Proceedings
of the
Eighth International Workshop
on
Constraint Handling Rules

September 8th, 2011
Cairo, Egypt

Preface

This volume contains the papers presented at CHR 2011, the Eighth International Workshop on Constraint Handling Rules, held on September 8th, 2011 in Cairo, Egypt at the occasion of the Second International Summer School on Constraint Handling Rules.

This workshop was the eighth in a series of annual CHR workshops. 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. Previous workshops on Constraint Handling Rules were organized in 2004 in Ulm (Germany), in 2005 in Sitges (Spain) at ICLP, in 2006 in Venice (Italy) at ICALP, in 2007 in Porto (Portgual) at ICLP, in 2008 in Hagenberg (Austria) at RTA, in 2009 in Pasadena (California, USA) at ICLP and in 2010 in Edinburgh (Scotland) at ICLP.

More information about CHR is available on the CHR website[1]. The papers from all previous editions of the CHR workshop (as well as many other CHR-related papers) are also available for download from the CHR website[2].

This edition of the CHR workshop received eight full paper submissions, of which one was retracted before reviewing started. Each submission was carefully reviewed by three program committee members or additional reviewers. The program committee maintained a very high standard of quality and decided to accept five papers. One of the accepted papers, *CHR for Spoken and other Biological Languages* by Verónica Dahl, was "upgraded" to an invited talk. The program also includes an invited talk by Andrea Triossi: *Boosting CHR through Hardware Acceleration.*

We are grateful to all the authors of the submitted papers, the invited speakers, the program committee members, and the reviewers for their time and efforts. We would also like to thank the local organizers, Slim Abdennadher, Amira Zaki and Ahmed Abousafy, at the German University of Cairo for the excellent organization of this workshop and the summer school on CHR with which it was collocated.

August 2011 Jon Sneyers
Ulm

[1] CHR website: `http://dtai.cs.kuleuven.be/CHR`
[2] CHR bibliography: `http://dtai.cs.kuleuven.be/CHR/biblio.shtml`

Workshop Organization

Program Chair

Jon Sneyers	K.U.Leuven, Belgium

Program Committee

Slim Abdennadher	German University in Cairo, Egypt
Henning Christiansen	Roskilde University, Denmark
François Fages	INRIA Rocquencourt, France
Thom Frühwirth	University of Ulm, Germany
Maurizio Gabbrielli	University of Bologna, Italy
Rémy Haemmerlé	Technical University of Madrid, Spain
Eric Monfroy	UTFSM, Chile and University of Nantes, France
Paolo Pilozzi	K.U.Leuven, Belgium
Peter Stuckey	University of Melbourne, Australia
Armin Wolf	Fraunhofer FIRST, Germany

Reviewers

Henning Christiansen	Eric Monfroy
Thom Frühwirth	Paolo Pilozzi
Rémy Haemmerlé	Jon Sneyers
Dragan Ivanovic	Peter Stuckey
Jacopo Mauro	Armin Wolf

Table of Contents

Invited Talks

Technical Papers

Application Papers

Boosting CHR through Hardware Acceleration

Andrea Triossi

DAIS, Università Ca' Foscari Venezia, Italy
triossi@unive.it

Abstract. The aim of this talk is to present a general framework for compiling Constraint Handling Rules (CHR) to a low level hardware description language (HDL). The benefit introduced by a CHR based hardware synthesis is twofold: it increases the abstraction level of the common synthesis work-flow and it can give significant speed up to the execution of a CHR program. I will describe a practical method that set CHR as starting point for a hardware description and afterwards I will show how to integrate the generated hardware code, deployed in a Field Programmable Gate Array (FPGA), within the traditional software execution model of CHR. The result is a prototype CHR execution engine composed of a general purpose processor coupled with a specialized hardware accelerator. The former executes a CHR specification while the latter unburdens the processor by executing in parallel the most computational intensive rules. The talk will have a practical focus, illustrating the achieved performance obtained by a prototype system architecture.

The motivations that gave rise to the development of a novel technique to synthesize behavioral hardware components starting from a declarative programming language are basically two. Firstly it introduces a high level of abstraction in an environment that is traditionally described at low level and then secondly it results in hardware blocks that can be interfaced with the high level language execution giving relevant time improvements. Therefore we developed a completely general framework that allows to synthesize reconfigurable hardware easily employable in a wide range of application, since a small modification to the high level code affects a huge portion of low level HDL code (resulting in a remarkable time saving for the programmer). Moreover the generated hardware code is fully compliant with the commonly adopted ones and hence can be easily integrated in the existing hardware project. Once hardware can be directly compiled from a high level language, we implemented a coupled system constituted by a traditional general purpose CPU and a hardware accelerator deployed on a Field Programmable Gate Array (FPGA). Thus such sole system will be compiled from a single high level language specification through a double compilation path resulting in a efficient execution engine. In the following paragraphs we briefly introduce our contributions to the two aforementioned research areas.

High Level Synthesis The ability to integrate an ever greater number of transistors within a chip and the consequent complexity growth of the features that

can be implemented in hardware demand for new languages and tools suitable for describing the different nature of the hardware devices. First goal of this work is to synthesize hardware starting from a language at a level higher than the ones of the commonly used behavioral HDLs. This should let programmers to easily focus on system behavior rather than on low level implementation details. The design procedure identified in [4] can be applied to a declarative paradigm rather than traditional imperative languages, inheriting all the well-known benefits for the programmer.

We identify in CHR [3] the high level language that can fulfill the hardware description. Its plain and clear semantics makes it suitable for concurrent computation, since programs for standard operational semantics can be easily interpreted in a parallel computation model [2]. In our work we want to fully exploit these features, highlighting the relations with the parallel characteristics of the target gate-level hardware.

We set the general outline of a hardware implementation of a CHR subset able to comply with the restricted bounds that hardware imposes. We will show how to efficiently accommodate in hardware concurrent rules execution. The level of parallelization achieved provides a time efficiency comparable with that obtained with a HDL design. The proposed optimizations then revealed that for certain classes of algorithms the degree of parallelization achievable in hardware is only limited by the available space resource. Clearly our work is not intended to be an answer to the problem of the high level hardware synthesis, but it attempts to highlight how a declarative programming language as CHR can contribute to ease the burden of the hardware designers. Several example of simple CHR programs will be presented showing the generality of application of the proposed framework and pointing out how we can develop hardware compiled to solve completely different tasks with minimal changes in the program's rules.

Hardware accelerator Hardware designers have used FPGAs for many years as single-chip accelerated application but recently, due to the increased devices density, FPGAs are being the focus of system designers attention as well. This change of perspective sparked the debate on what could be the best language for describing a mixed hardware/software system. The lack of a suitable standard gave rise to a plethora of high-level languages (HLLs) oriented to increase the productivity in the description of reconfigurable computing devices [1].

We will show that CHR can be efficiently used directly at system level: the generated hardware can be employed as specialized hardware accelerator for a general purpose processor. In such a way we achieved a close synergy between high level software and hardware because the former facilitates the synthesis of the latter and the latter clears the way for a fast execution of the former.

The entire system compilation is split into two branches related to the hardware and software parts. The compilation begins from an annotated program in which the programmer highlights the rules that have to be executed by the hardware accelerator. The hardware compilation consists of the application of our high level synthesis technique [5] that results in a bit stream directly de-

ployable in a FPGA. On the other side the standard software compilation will be necessary altered due to the removal of some rules from the full program specification. The execution of the removed rules will be embedded in a custom made built-in Prolog predicate. When it is called all the constraints of a specific type are sent to the hardware accelerator that, once it has terminated the computation, will send the resulting constraints back to the constraint store.

The proposed hardware based approach increases the performance of constraint programming trying to achieve a stronger coupling between algorithms and platforms. A custom hardware based on FPGA is used as accelerator for CHR code: the traditional program execution is then interfaced with a custom accelerator engine. We will present case studies in which the high level hardware compilation produces the needed hardware code for the accelerator, and simple CHR rules, added to the program, let the CPU cooperates with the accelerator itself. Finally we should remark that the system integration is made smoother by the adoption of the same language for the software and hardware description confirming the possibility to adopt CHR as system level language.

References

1. E. El-Araby, P. Nosum, and T. El-Ghazawi. Productivity of High-Level Languages on Reconfigurable Computers: An HPC Perspective. In *IEEE International Conference on Field-Programmable Technology*. IEEE, 2007.
2. T. Frühwirth. Parallelizing Union-Find in Constraint Handling Rules Using Confluence Analysis. In *Proc. of the 21st Conference on Logic Programming (ICLP)*, volume 3668 of *LNCS*, pages 113–127. Springer, 2005.
3. T. Frühwirth. *Constraint Handling Rules*. Cambridge University Press, 2009.
4. G. Hu, S. Ren, and X. Wang. A comparison of C/C++-based Software/Hardware Co-design Description Languages. In *Proc. of the 9th International Conference for Young Computer Scientists*, pages 1030–1034. IEEE, 2008.
5. A. Triossi, S. Orlando, A. Raffaetà, F. Raiser, and T. Frühwirth. Constraint-based hardware synthesis. In *Proc. of 24th Workshop on (Constraint) Logic Programming*, 2010.

CHR for Spoken and other Biological Languages

Veronica Dahl

Department of Computer Science
Simon Fraser University
Burnaby, B.C., Canada
veronica@cs.sfu.ca

Abstract. CHR is now well established as an invaluable tool for computing and other formal science applications. Much less studied is their use in the humanistic sciences. In this article we bring together our personal view on applying Constraint Reasoning as embodied in CHR to joining the humanistic with the formal sciences, through the link of language– understood in a broad sense as including both spoken languages and molecular biology languages. and we try to distill from these heterogeneous enterprises some common threads that can possibly lead to an embryonic model of humanistic investigation through CHR. The applications we cover include such themes as a dual processing scheme for both human and biological languages; decoding nucleic acid strings through human language; DNA replication as a model for computational linguistics; multi-disciplinary biological knowledge representation for early cancer diagnosis; RNA-inspired analysis of poetry; parsing medical text into de-identified databases; and biological concept-formation.
Keywords: CHR, CHRG, assumptions, abduction, Hyprolog, nucleic acid mining, computational linguistics, concept formation, parsing, medical text processing,

1 Introduction

Instantaneous communication made possible by contemporary technology is favouring more cross-disciplinary interactions than ever, as well as accelerating those within each field. And it was high time, because it is becoming clearer that some disciplines simply need to join forces with others. For instance, an unprecedented volume growth of biological data over the past few years (most notably, human language text produced in the form of articles, books, web sites, etc., and genetic code text in nucleic acid language, such as DNA sequences) has created formidable challenges for their timely and interrelated processing. Traditional methods in biology for processing and making sense of such information can no longer keep up with the exponentially growing information tsunami. Methodologies that pertain to the Natural Language Understanding field of AI are now being exploited to analyze biological sequences, which is uncovering similarities between the languages of molecular biology and spoken languages such as English. Such similarities might help explain the curious fact first discussed in [24],

that many techniques used in bioinformatics, even if developed independently, may be seen to be grounded in linguistics.

This observation, still valid, resonates deeply with this author, whose obsession with language in all forms has led her, through sometimes unconventional interdisciplinary enterprises that often try to bridge the humanistic and the formal sciences, to search for ever higher while executable levels of description that can transfer into different disciplines.

During this search, the CHR paradigm [19] as embedded in CHRG [7] and HyProlog [5] has become central in many ways, which we discuss in this paper. The main extensions we use are abduction (the unsound but useful inference of p as a possible explanation for q given that p implies q) and assumptions (resources that are globally available as from their inception while being backtrackable) [17, 5], both subject to consistency with a special type of facts: integrity constraints. These extensions allow us to move beyond the limits of classical logic to explore possible cause and what-if scenarios– as befits the needs for flexibility of linguistic applications.

2 Background

2.1 CHRG

CHR Grammars, or CHRGs for short, are to CHR what Definite Clause Grammars (DCGs) are to Prolog, and are executed as CHR programs that provide robust parsing with an inherent treatment of ambiguity.

For instance, the CHRG rules

```
token(leucine)::> codon([u,u,a]).
token(leucine)::> codon([u,u,g]).
```

are equivalent to the CHR rules:

```
token(X0,X1,leucine)==> codon(X0,X1,[u,u,a]).
token(X0,X1,leucine)==> codon(X0,X1,[u,u,g]).
```

where the word-boundary arguments, X0 and X1, are now explicit.

2.2 Hyprolog

Hyprolog [5] is an extension of Prolog and of CHR with assumptions and abduction.

Abduction is agreed upon as a powerful technique in logic programming but its actual use in practice appears to be rather limited since most available systems are research prototypes implemented using inefficient metaprogramming techniques. Assumptive logic programming is related to abduction but provides explicit creation and consumption of hypotheses plus scoping principles inspired by linear logic. It can be hard-wired as in Bin-Prolog [20] or incorporated into any Prolog version with some loss of speed.

As an example of the use of assumptions, here is a graph walking program which avoids loops simply because assumed facts are usable (i.e. consumed upon their successful unification with a goal) at most once. "c" stands for "connected, so c(Node,NodeList) denotes that Node is connected with each element of NodeList. Assumptions are preceded by a plus sign, and consumptions by a minus sign.

```
path(X,X,[X]).
path(X,Z,[X|Xs]):-linked(X,Y),path(Y,Z,Xs).

linked(X,Y):- -c(X,Ys), member(Y,Ys).

start(Xs):-
  +c(1,[2,3]), +c(2,[1,4]), +c(3,[1,5]), +c(4,[1,5]),
  path(1,5,Xs).
```

By executing `?-start(Xs)`, we will avoid loops like `1-2-1 and 1-2-4-1` and obtain the expected paths:

```
  Xs=[1,2,4,5];
  Xs=[1,3,5]
```

In [5] we showed that abduction and assumptions can be integrated with traditional Prolog programs without any significant slow-down in execution speed or other burdens for the programmer.

This was achieved by using a trivial extension for abducible predicates and assumptions written in CHR . From the user's end, the notation and processing remain consistent with ordinary Prolog, including for integrity constraints, which are expressed in CHR (as embedded in Prolog). The only visible difference with ordinary Prolog is that certain predicates are declared as abducibles, that assumptions are identified through notation (plus for assuming and minus for consuming), and that abducibles (as is standard with abducibles) never appear as clause heads .

For instance, to abduce that it rained or that a sprinkler was on from the information that the grass is wet, under an integrity constraint that if the trees are dry it cannot have rained last night, all we need to write (aside from calling the Hyprolog module and declaring our abducibles as such) is:

```
grass(wet):- rained_last_night.
grass(wet):- sprinkler_was_on.

trees(dry), rained_last_night ==> fail.
```

To the best of our knowledge, HyProlog provides one of the most efficient implementations of abductive logic programming, perhaps the most efficient one, and the price for this is a limited support of negation, as detailed in [5].

3 Language Understanding

3.1 On spoken languages and the two cultures

Humans interpret spoken languages through clues much beyond the literal meaning strictly conveyed in the text being interpreted; for instance metaphor, irony, and pronoun reference are usually directly understood even if metaphor often points to a different domain than the discourse's, irony conveys roughly the opposite of what is literally being said, and several entities may potentially be referred to by the same pronoun. Contextual and pragmatic knowledge present in the hearer's mind but hard for a computer to store and pertinently retrieve makes humans quite unbeatable in language understanding compared to computers. Two major breakthroughs have nevertheless considerably advanced the state-of-the-art in the past few decades:

- Logic based processing tools have made it possible for high level formulations of human languages to be directly executable while remaining close to human formulations, such as those familiar to linguists (documented for instance yearly at the European Summer School of Logic, Language and Informatics series).
- The computer industry's bonanza in terms of speed and memory has made brute force approaches more affordable and in many cases successful. In particular, statistical and probabilistic language models have had engineering success in providing an accurate simulation of some linguistic phenomena [21].

These two developments are sometimes seen as competing, and there is an ongoing discussion in linguistic circles as to which is "best", cf. for instance [21].

It is our thesis that applying CHR –or its grammatical counterpart, CHRG– to processing language promotes combinations of these two approaches which can largely keep the best of both worlds. We shall develop this thesis in what follows.

3.2 Parsing spoken human languages

CHR allows for high level while executable descriptions of language much as logic grammars do. In addition, its bottom-up emphasis probably makes it simpler to grasp for minds with a penchant for the concrete, and for those for whom the word logic equates with "difficult".

As well, it facilitates interaction between different language levels, by making it possible to access several constraints, possibly coming from disparate language modules, through the same rule. Thus, CHR rules lend themselves beautifully to the specialized task of describing linguistic formalisms which view a grammar as a set of constraints, and parsing as a constraint satisfaction problem. One interesting example are Property Grammars (PG) [1], described through properties between constituents plus conditions under which some can be relaxed. Faced

with incomplete or incorrect input, the parser still delivers results rather than failing and indicates the reasons of anomaly through a list of satisfied and a list of violated properties. For instance, a PG parse of the noun phrase "every blue moon" results in a set of satisfied properties (e.g. *linear precedence* holds between the determiner and the noun, between the adjective and the noun, and between the determiner and the adjective; the noun's requirement for a determiner is satisfied, etc.) and a set of unsatisfied properties, which is empty for this example. Some properties can be relaxed, e.g. a language tutoring system geared to Spanish speaking students might want to relax the linear precedence constraint between the adjective and the noun, so that for instance "Every moon blue" would be accepted even though ungrammatical in English, but the violation would be indicated by placing the violated relationship in the set of unsatisfied properties, thus signaling to the student an unexpected word ordering in the language he or she is trying to learn.

One of the problems with constraint-based approaches is that constraints are usually expressed over high-level objects or structures. This is the case for example in HPSG [22], in which complex feature-structures must first be built before constraints can be evaluated. Similarly, Optimality Theory [23] also generates a set of structures (or candidate structures) and then uses constraints to filter this set. In our approach, any constraint can be evaluated at any time for any set of categories. Such an evaluation dynamically adds new information: the satisfaction of a *selection* constraint instantiates the syntactic category it describes. But this instantiation is conceived almost as a side effect of evaluation: satisfying constraints does not rely on the knowledge of the upper-level category. In other words, the hierarchical information is no longer preponderant in the parsing process. This means that one can evaluate subsets of constraints, for example in the case of applications that only need NP recognition. Achieving robustness in the face of incomplete information or noise is thus also made easier.

We have developed a parsing scheme for PGs in terms of CHRG [11] which validates the model of property centered parsing with respect to efficiency, while preserving the level of generality of this theory. To illustrate our parsing scheme for PGs, we now show the core rule, which combines two consecutive categories (one of which is of type `XP` or obligatory) into a third, by testing each of the properties on the pair and creating the new property lists through property inheritance [11]. Its form is described in Fig. 1.

This rule first tests that one of the two categories is of type `XP` (a phrase category) or obligatory (i.e., the head of an `XP`), and that the other category is an allowable constituent for that `XP`. It then successively tests each of the PG properties among those categories, incrementally building as it goes along the lists of satisfied and unsatisfied properties. Finally, it infers a new category of type `XP` spanning both these categories, with the finally obtained `Sat` and `Unsat` lists as its characterization.

In practice, this rule unfolds into two symmetric parts, to accomodate the situation in which the XP category appears before the category Cat which is to be incorporated into it.

```
cat(Cat,Features1,Graph1,Sat1,Unsat1):(Start1,End1),
cat(Cat2,Features2,Graph2,Sat2,Unsat2):(End1,End2) ::>
    xp_or_obli(Cat2,XP), ok_in(XP,Cat),
    acceptable(precedence(XP,Start1,End1,End2,Cat,Cat2,Sat1,Unsat1,SP,UP,BP)),
    acceptable(dependency(XP,Start1,End1,End2,Cat,Features1,Cat2,
        Features2,SP,UP,SD,UD,BD)),
    build_tree(XP,Graph1,Graph2,Graph,ImmDaughters),
    acceptable(unicity(Start,End2,Cat,XP,ImmDaughters,SD,UD,SU,UU,BU)),
    acceptable(requirement(Start,End2,Cat,XP,ImmDaughters,SU,UU,SR,UR,BR)),
    acceptable(exclusion(Start,End2,Cat,XP,ImmDaughters,SR,UR,Sat,Unsat,BE))
    | cat(XP,Features2,Graph,Sat,Unsat).
```

Fig. 1. New Category Inference

We note that direct interpretation, moreover, guarantees a better evolution of the system: it can better adjust to changes in the theory and to experimental stages. This has for instance allowed us to add new semantic properties for specific application purposes [14].

4 From linguistic to medical applications: early diagnoses, de-identification

Not only language levels can be simultaneously accessed by a single rule: also non-linguistic components of an AI system can. This facilitates, for instance, co-operation between knowledge repositories and linguistic components of a same system, e.g. the grammatical component of an AI system can determine the semantic type of a linguistic argument in collaboration with an ontology component of the same system, or even with web search.

Thus, [2] takes inspiration from the linguistic developments above reported (in particular, from the author's rendition of Property Grammars [11], which relies exclusively on constraints, controls the parse through head-driven analysis, and provides a direct interpretation, while preserving all theoretical properties at the implementation level) to propose a CHR based model of multidisciplinary information which can combine heterogeneous clues about the development of oral and lung cancer at the early stages along with the patients medical history and behavioral risk analysis, for a more accurate diagnostic of the probability of the lesions advancing to cancer or not. Each element of data involved in the analysis presents very different characteristics, and the model is robust in that it will still reach useful conclusions even when not all of the data is available for a given patient. Underlying this model is an extension of the CHR-based Concept Formation model [12], which evolved from [11].

Interestingly, the addition of probabilities in the early cancer diagnosis model is done on a rule by rule basis, rather than as a separate module. From this point of view we already observe the "best of both worlds" situation: while purely probabilistic or statistical models of language provide little insight, as observed

in the discussion in [21], the incorporation of probabilities into rules that do provide cognitive insight by relating concepts links them into those insights, for mutual complementation.

As said, sometimes in a parsing process lexical, syntactic and semantic information must cooperate dynamically in order to zoom onto the precise meaning of a natural language sentence or discourse. In these cases we have found CHR to be particularly helpful, given that constraints springing from different grammar components can be conjured into the same CHR rule, which gets informed from all these sources simultaneously. A case in point is [15], which proposes a methodology that exploits this feature in order to develop a model of medical document de-identification.

De-identification is the process of automatic removal of all personally identifying Private Health Information (PHI) from medical records, while protecting the integrity of the data as much as possible [26]. In the current state of the art, although most of the performance metrics reported in every other paper, hits at least some 90% performance measure, most of the times, several restrictions to the input text and to the target output have been assumed. These mean we still need a human assistant to do at least the final scanning if not re-processing.

Our work on de-identification deals with privacy sensitive texts that are rich sources of research information by extracting the knowledge these texts represent and feeding them into a database, and by marking any sensitive fields syntactically for eventual removal by the system. The system takes into account what type of research will be conducted in order to protect identifying fields as needed. Thus researchers, instead of accessing the text, can query the database for the information they require, while having no access to specific identities. The hybrid nature of constraints involved (coming from different grammatical levels, or combining semi-structured with free text elements) can be elegantly handled by CHR's multi-headed rules. To illustrate the power of combining constraints from heterogeneous sources, lexical entries in the grammar for a hospital admissions application can glean information from an ontological component, resulting in lexical-semantic constraints that can deal with ambiguities such as:

```
enter(patient-X,hospital-Y).  (as in "admitted into the hospital")
enter(patient-X,state-Y)  (as in "entered into a comma")
```

The parser can then keep track of potential referents for pronouns and other referential terms through the use of assumptions. Interestingly, disambiguation and anaphora resolution can cooperate with each other: semantic types allow us to differentiate between a patient named Huntington and a disease named so; thus, further ensure the correct identification of a referent, as the following discourse and corresponding representations exemplify.

"Huntington entered the hospital on April 16, 2010. This patient should be tested for Huntington."

```
+entered(patient-id(huntington), hospital-id(universalcures),
```

```
date-id(16-04-2010)).
must-test-for(patient-P,disease-huntington)
```

Our parser's anaphora resolution system will instantiate P with id(huntington) and correspondingly mark the relation "must-test-for" as an assumption. The explicit mention of the type ("patient") in the subject of the second sentence serves as a corroboration to the anaphora resolution system that we are referring indeed to the Huntington typed as a patient, in the first sentence. If marked otherwise, the two types would not have matched. If the second sentence were "He should be tested for Huntington", the type gleaned from the first sentence for this individual would simply carry over, together with his name, into the term representing it. Of course, even for humans there will be cases in which even context leaves us clueless, as in "Huntington won". We are content if our proposed methodology allows us to deal with ambiguity with as much success as humans can.

5 Biological Languages

5.1 The languages of Nucleic Acid

Biological sequence analysis is resorting more and more to AI methods, given the astounding rate at which such information grew over the last decade. Old methodologies for processing it can no longer keep up with this rate of growth. AI methods such as logic programming and constraint reasoning have been coming to the rescue, generating a fascinating and interdisciplinary field. In particular, methodologies that pertain to the natural language processing field of AI are now being exploited to analyze biological sequences, which is uncovering similarities between the languages of molecular biology and human languages. [13], identifies some of the forms that tend to repeat in both human and molecular biology languages, and proposes a uniform treatment through CHR, regardless of the area of application.

Some of the forms found both in nucleic acid strings (made up of the "words" or nucleotides A, C, T, U and G) and in natural languages are relatively simple yet not necessarily easy to parse: e.g. *palindromes* (sequences that read the same from left-to-right or from right-to-left, as the Spanish sentence, modulo blank spaces: "Dabale arroz a la zorra el abad", or as the DNA sequence A C C T G G T C C A). Their length can vary, and their position within in a string is unpredictable. *Tandem repeats* (where a substring repeats again right away) also appear in both types of languages, as "tut" does in "Tut, tut, it looks like rain", or as C G A within the sequence C C A T C G A C G A U A). In human languages, repetition can appear literally or in more involved phenomena such as full conjunctive clauses, where the surface forms are not the same, but the structure repeats around some coordinating word like "and", "or", "but" (as in "Slowly but surely, ...", where the tandem repeat is between two adverbs, and a mediating conjunction intervenes).

In addition to these basic structures, which can be found in linear sequences, pairings of nucleotides, which attract each other, form more complex structures, where the sequences fold into three dimensions: the nucleotide A tends to pair with T, and C with G. These are called *Watson-Crick* or *canonical base pairs*. These base pairings result in structures or motifs of a variety of forms, such as helix, hairpin loop, bulge loop and internal loop. One of the widely occurring complex structures in molecular biology is the *pseudoknot* which has been proved to play an important role for the functions of RNA. A simple pseudoknot is formed by pairing some of the bases in a hairpin loop that are supposed to stay unpaired, with bases outside the loop. If we draw for natural language sentences some of the links between, for instance, a clause's antecedent and the clause itself, we obtain similarly shaped figures.

5.2 Decoding nucleic acid through spoken language

High level code for analyzing nucleic acid strings can be written by computer specialists in reasonably useful and efficient ways. Still, it is the prerogative of computer specialists to write such code, even if in interaction with biology experts, and in some cases, of specialists in Artificial Intelligence. These are used to instructing computers to "think" logically and to conduct effective searches of large problem spaces by endowing their computer programs with reasoning capabilities, often based in executable incarnations of logic. The ability to encode such solutions in a suitable AI language is an acquired skill which requires extensive knowledge of the language, practice with writing programs in that language, and a lot of programming discipline and ingenuity. It moreover requires a suitable level of interdisciplinary communication skills in order to clearly capture the precise description of what is to be done, from the biologist who is interested in the results and for whom his own jargon is second nature. This involves the development of a common jargon or at least an understanding of the other's jargon for each of the disciplines involved. Not an easy task, but one in which good breakthroughs have been made and which advances at a quick pace.

In [9] we propose human language itself as the high level query language for decoding. We aim at an even higher level of interaction with computers- one in which biologists are given the software tools for commanding computers through their own human language such as English, to extract genetic information of their interest which is encoded in DNA strings. Ultimately we aim at doing away with the need to resort to a computer specialist- a task which seems formidable and perhaps is so in its full generality, but for which subtasks exist which are useful enough, quite impressive, and feasible.

Specifically, we extend a series of DNA decoding primitives written in CHRG, such as those proposed in [4], into human language primitives which can then be used to automatically program the decoding of a nucleic acid string from a sentence that describes in plain language (English, French, etc.) what needs to be analysed within the string. Some efficiency concerns are tackled by appropriate constraints and thus remain invisible to the user, except in their effects.

Both the parser and the DNA decoding toolkit components of the system use CHR. Our approach allows for eager discarding of wrong lines of reasoning, as well as for paraphrases of a given question without ill-effects in the execution, and with consequent gain in the richness of the input accepted. As well, it permits a cooperative integration between both components, by allowing one of them to inform the other one through integrity constraints in CHR.

6 Cross-fertilizations between human and biological languages

6.1 A dual processing scheme for both human and biological languages

David Searls proved that the grammar of nucleic acid language is in fact non-deterministic and ambiguous and moreover not context-free [24]. These features are shared by so-called natural languages such as English.

Based upon these and other similarities, [13] proposes a model of human language processing, called Synalysis, built around CHRG. Inspired by biological sequence replication and nucleotide bindings, this model can express and implement both analysis and synthesis in the same stroke. This is akin to biological mechanisms, such as DNA substring repair, in which a string is analysed while being synthesized elsewhere. The uses of Synalysis are exemplified around the language processing phenomenon of long distance dependencies, which also presents in molecular biology since it involves relating two substrings (of either human or biological language text) which might be arbitrarily far apart from each other. Our proposed model is suitable to those language processing frameworks known as *compositional*, where the representations obtained for the whole are composed out of partial representations obtained for the parts.

This research relies on CHR for the following reasons:

- ambiguity is inherently treated because all possibilities resulting from ambiguous input are expressed in the constraint store
- long-distance dependencies, including strings that repeat arbitrarily far apart, are easily conveyed through multi-headed rules
- in its grammatical version, CHRG, we are spared from explicitly manipulating the input and output arguments and can specify context explicitly
- memoing, a well-established technique for human language processing which is also used in our dual model, is inherently available in CHR

Our research in [13] also shows that, while for the simpler of the forms we have identified as being common to both molecular biology and human language sequences, a uniform treatment through CHR is adequate in both disciplines, more complex forms might require the complement of heuristic rules. In our own research we have incorporated them through probabilities implemented in ad-hoc fashion, suiting our needs, because CHRiSM [25] was not yet available. It would be interesting to restate our results in terms of CHRiSM, since its present

availability may well prove to be an additional reason to favor CHR approaches to a unified processing scheme for both human and biological languages. In any case, we found that adding the needed probabilities to our CHR formulation was a straightforward enough task, giving further proof to our thesis that the two cultures can cooperate rather than compete.

6.2 Literary applications: an RNA-inspired analysis of poetry

The style in which an RNA molecule folds in space obeys laws of nucleotide binding and attraction which are encoded in its primary structure, that is, in the sequence of nucleotides conforming it. Natural language sentences can also be viewed as encodings for a structure in space- in this case, a parse tree- which exhibits relationships or bindings between different parts of the sentence. In [3], we presented a novel methodology –chrRNA– for addressing the bioinformatics problem of finding an RNA sequence which folds into a given structure. In [8] we explored the possibilities in adapting this methodology to the problem of parsing poems that follow specific stylistic trends, e.g. because they belong to the same author. Just as chrRNA involves a very simple grammar, which augmented by probabilities can lead to approximate but still useful solutions for a problem that has been proved to be NP-hard, and these probabilities encode the molecule's "style", as it were, adapting our method to computational linguistics involves resorting to stylistic probabilities observed in a given author's poetic production in order to aid in the parse of a given poem of the same author, or to aid in determining authorship itself. This methodology can also be applied to authorship determination.

As a simple example, consider the following sentence, adapted from Nicolas Guillen's poem "La Guitarra": "Dejo el borracho en su coche, dejo el cabaret sombrio". This can be parsed into one sentence, which explicitly and in English would correspond to "The drunkard left the sombre cabaret (by traveling) in his car". Any Spanish reader would understand that the verb's repetition (dejo=left) is for poetic effect, rather than a "new" main verb. A machine analyzer, however, would recognize two sentences, corresponding either to: "(Someone) left the drunkard in his car, and (the same person) left the sombre cabaret, or "Someone left the drunkard in his car, and the sombre cabaret left". The first interpretation is likely when one considers that implicit subjects are very common in Spanish (so any reasonable Spanish analyzer would conceive it); the second one is nonsensical for humans but plausible from syntax alone, and therefore, a fair candidate for a poetically uninformed parser. Note that while the state-of-the-art in parsing would allow us to choose between these two interpretations, perhaps by paying the price of including semantic type information to preclude non animated subjects such as "the cabaret" for movement verbs such as "dejo", the interpretation as a single sentence with verb repetition would remain inaccessible, to the best of our knowledge, to state-of-the-art parsers, including those meant to analyze poetry. Algorithmic approaches to poetry have been around for a relatively long time, but they mostly focus on generating poetry by automated or semi-automated means (e.g. [18]), and as such, belong to the general field of

Electronic Writing. Automated poetry analysis, on the other hand, remains a bit more elusive, and concentrates on the more mechanizable subtasks, such as automated analysis of sound and meter.

As in the case of RNA design [3], we can encode probability values that will allow us to determine, in case of ambiguity, which possible analysis is more likely. Thus we can encode CHR rules with probabilities to the effect that when a verb is not the initial word in a sentence, the noun phrase that follows it is likely to be a direct object rather than a subject, so we can analyze it as the direct object of that verb. This rule will be used for instance for "dejo el cabaret sombrio".

Likewise, we can indicate through probabilities that initial verbs in Guillen's poetry are likely to appear before the subject noun phrase just for stylistic effect, as in "dejo el borracho" (literally, "left the drunkard").

7 Towards a model of humanistic investigation through CHR

We can distill, from all the discussed body of research, a series of features that make the CHR paradigm especially promising as potentially leading to a model of humanistic investigation:

- modularity allows for straightforward change for experimentation purposes
- concreteness is promoted by naturalness of bottom-up thinking
- flexibility, e.g. top-down and bottom-up strategies can coexist as needed
- potential for combining heterogeneous sources, which also serves for long distance dependency phenomena
- robustness: partial results are possible even if some data needed for a complete result is missing
- inherent treatments of ambiguity and memoing
- straightforward addition of probabilities, which are ubiquitously needed, in normal CHR or CHRG rules
- recent availability of CHRiSM for further syntactic sugar

In addition, we note that as embedded in our language processing tools (Hyprolog, CHRG), it presents the following further advantageous features:

- non-classical inference such as abduction and assumptions add flexibility and make it easy to explore what-if scenarios
- automatic handling of input and output arguments in the case of grammars.
- working store elements can come from a variety of disparate sources, and thus they lend themselves ideally for incorporating multi-agents that collaborate in tasks that require intelligent interactions with non-grammatical kinds of agents.

Crucial to humanistic investigation in any discipline is the formation of concepts in flexible enough a way that their properties can be enforced or relaxed as

needed. For this reason, we put forward that the CHR based paradigm of Concept Formation [12] may be appropriate in this respect. But as we have also seen, probabilities tend to play a major role in many of the applications described, as well as being important for natural language processing itself, an area which is relevant to all areas.

As proof of concept, we have abstracted, from recent different realizations of the linguistically inspired Concept Formation paradigm, a multi-agent model for Biological Concept Formation which can be considered as a computational metaphor for the (biological) mind, with direct executability implications [10]. Due to the generalized use of Constraint Handling Rules or their grammatical counterpart, we are able to integrate human language processing techniques into our approach which are not only useful for all types of concept formation but also allow us a smooth integration of human language processing agents, as well as their interactions with the knowledge base agents. Another interesting feature of our proposal is its robustness: due to the capability of relaxing some of the properties involved in concept formation, results that can be useful are provided even in the absence of all the information "necessary" to form the concepts in question.

Concept formation rules are applicable to many other AI and cognitive problems as well, most notably, those involving the need to reason with incomplete or incorrect concepts.

Another important facet of humanistic investigation is parsing spoken language itself, which will be important in any discipline. The parsing model needed for Language Intelligence as we understand it (i.e., much beyond keywords and syntactic variants) must satisfy three main requirements: ability to decode text into knowledge bases, flexibility to accommodate the imperfections and imprecision typical of spontaneous human language use, while exploiting its rich expressive power to good advantage, and good potential to blend in, and cooperate with, semantic web technologies. Adapting the new family of Abductive Grammars [6, 16] holds great promise in this respect, because of their built-in ability to construct knowledge bases from language sentences. As well, our constraint-based rendition of Property Grammars [11]) holds great promise because of their focus on yielding useful results even for imperfect input (involving noise, incorrect input, and incomplete input).

8 Concluding Thoughts

Just as our root discipline – Philosophy – needed to slowly separate into a myriad of disciplines and sub-disciplines in order to develop methods specific to each, to achieve depth, etc., we believe we are now at a time in which an inverse process of integration needs to happen: in specializing, some disciplines have become unnaturally disconnected from others or from the whole, with the result that the broad view of the forest is sometimes lost, and that parallels that could be exploited cannot even be seen. A reconnection from the more mature present standpoints of these different branches seems in order and in any case,

is simply happening. We hope to have shown, in a very modest way and for just a few of the disciplines needing this, that the CHR paradigm can constitute a good pivot around which to perform the needed reconnection, and that whereas constraint programming seems more and more focussed on solving "classic" OR problems and benchmarking, it is not mostly about gaining 1 ms on instance i of problem p, but is also a powerful descriptive tool for addressing interdisciplinary applications with the mighty advantage of direct executability.

Acknowledgement

Support from NSERC's Discovery Grant 31611024 is gratefully acknowledged. I also thank the anonymous referees for very useful comments on this article's first draft.

References

1. Blache, P.: Property Grammars, a Fully Constraint-Based Theory. In: Constraint Solving and Language Processing, H. Christiansen et al. (eds), LNAI 3438, Springer (2005)
2. Barranco-Mendoza, A. Persaoud, D.R., Dahl, V.: A Property-Based Model for Lung Cancer Diagnosis. Poster, RECOMB 2004, San Diego (2004) 27–31
3. Bavarian, M. , Dahl, V. : Constraint-Based Methods for Biological Sequence Analysis. Journal of Universal Computing Science (2006) 12(11) 1500–1520
4. Bel Enguix, G., Jimenez-Lopez, M.D., and Dahl, V.: Mining Linguistics and Molecular Biology Texts through Specialized Concept Formation. Poster, NLPCS09
5. Christiansen, H. and Dahl, V.: HYPROLOG: a New Logic Programming Language with Assumptions and Abduction. LNCS 3668: 159-173 (2005)
6. Christiansen, H., Dahl, V.: Abductive Logic Grammars. In: Ono, H., Kanazawa, M., de Queiroz, R.J.B. (eds.) WoLLIC 2009. LNCS, vol. 5514, pp. 170–181. Springer, Heidelberg (2009)
7. Christiansen, H.: CHR Grammars. Journal on Theory and Practice of Logic Programming. 5, 467–501. (2005)
8. Dahl, V., Jimenez-Lopez, M.D., and Perriquet, O. Poetic RNA: Adapting RNA Design Methods to the Analysis of Poetry. IN: PAAMS 2010, Vol. 2, pp. 403-410, Springer Verlag, ISBN 978-3-642-12383-2. (2010)
9. Dahl, V.: Decoding Nucleic Acid Strings through Human Language. Bel-Enguix, G., Jiménez-López, M.D. (eds.) Language as a Complex System: Interdisciplinary Approaches. Cambridge Scholars Publishing (2010)
10. Dahl, V., Barahona, P., Bel-Enguix, G., Kriphal L.: Biological Concept Formation Grammars- A Flexible, Multiagent Linguistic Tool for Biological Processes. LAMAS (2010)
11. Dahl, V., Blache, P.: Directly Executable Constraint Based Grammars. In: Journées Francophones de Programmation en Logique avec Contraintes, Angers, France (2004)
12. Dahl, V., Voll, K.: Concept Formation Rules: An Executable Cognitive Model of Knowledge Construction. In: 1st International Workshop on Natural Language Understanding and Cognitive Sciences. Porto, Portugal (2004)

13. Dahl, V., Maharshak, E.: DNA Replication as a Model for Computational Linguistics. . In J. Mira et al. (Eds.): IWINAC09 (Best Paper Award). LNCS, vol. 5601, pp. 346-355. Springer-Verlag, Heidelberg (2009)
14. Dahl, V., Gu, B.: Semantic Property Grammars for Knowledge Extraction from Biomedical Text. In: 22nd International Conference on Logic Programming (2006)
15. Dahl, V., Saghaei, S., Schulte, O.: Parsing Medical Text into De-identified Databases. In: 1st International Workshop on AI Methods for Interdisciplinary Research in Language and Biology, Rome (2011)
16. Dahl, V.: From Speech to Knowledge. In: Pazienza, M.T. (Ed.) Information Extraction: towards scalable, adaptable systems. Springer, LNAI 1714, Springer, pp. 49-75 (1999)
17. Dahl, V. and Tarau, P.: Assumptive Logic Programming, Proc. ASAI'04, Cordoba (2004)
18. Mendelowitz, E.: Drafting poems: inverted potentialities. In: International Multimedia Conference, Proceedings of the 14th Annual ACM International Conference on Multimedia, pp. 1047-1048. Santa Barbara (2006).
19. Fruhwirth, T.W.: Constraint Handling Rules. Cambridge University Press, ISBN 9780521877763, 2009.
20. Tarau, P. (2011) The BinProlog Experience: Architecture and Implementation Choices for Continuation Passing Prolog and First-Class Logic Engines. CoRR abs.1102.1178.
21. Norvig, P.: On Chomsky and the Two Cultures of Statistical Learning. http://norvig.com/chomsky.html.
22. Pollard C. & I. Sag (1994), *Head-driven Phrase Structure Grammars*, CSLI, Chicago University Press.
23. Prince A. & Smolensky P. (1993), *Optimality Theory: Constraint Interaction in Generative Grammars*, Technical Report RUCCS TR-2, Rutgers Center for Cognitive Science.
24. Searls, D.: The Language of Genes. Nature, 420, 211–217 (2002)
25. Jon Sneyers, J., Meert, W., Vennekens, J. CHRiSM: Chance Rules induce Statistical Models. In: Proc. CHR'09 Workshop, Pasadena, CA (2009)
26. Uzuner, O., Luo, Y., and Szolovits, P.: Evaluating the state-of-the-art in automatic deidentification. Journal of the American Medical Informatics Association, 14(5):55063 (2007)

Angelic CHR

Thierry Martinez

EPI Contraintes, INRIA Paris-Rocquencourt,
BP105, 78153 Le Chesnay Cedex, France.
thierry.martinez@inria.fr

Abstract. Implementations of CHR follow a committed-choice forward-chaining execution model: the non-determinism of the abstract semantics is partly refined with extra-logical syntactic convention on the program order and possibly notations for weighted semantics (with priorities or probabilities), and partly left unspecified in the underlying compiler. This paper proposes an alternative execution model which explores all the possible choices, by opposition to the committed-choice strategy. This execution model is angelic in the sense that if there exists a successful execution strategy (with respect to a given observable), then this strategy will be found. Formally, the set of computed goals is complete with respect to the set of the logical consequences of the interpretation of the initial goal in linear logic. In practice, this paper introduces a new data representation for sets of goals, the derivation nets. Sharing strategies between computation paths can be defined for derivation nets to make execution algorithmically tracktable in some cases where a naive exploration would be exponential. Control for refined execution is recovered with the introduction of user constraints to encode sequencing, fully captured in the linear-logic interpretation. As a consequence of angelic execution, CHR rules become decomposable while preserving accessibility properties. This decomposability makes natural the definition in angelic CHR of meta-interpreters to change the execution strategy. More generally, arbitrary computation can be interleaved during head matching, for custom user constraint indexation and deep guard definition.

1 Introduction

Since the introduction of Prolog, logic programming has been living with the dichotomy: "programs = logic + control". As every logic-based language, the declarativity of Constraint Handling Rules [7,15] relies on the logical interpretation of the programs. However, this interpretation hides syntactic conventions, like the order of the rules, distinguished abbreviations such as propagation rules, and annotations which control the effective execution of the program. These control features are formally described in a hierarchy of semantics, from the abstract semantics ω_{va} [7] to more fine-grained semantics, describing the handling of propagations (the theoretical semantics ω_{t}), of the rule ordering (the refined semantics ω_{r} [5]), or of annotations like priorities [4] or probabilities [13].

CHR enjoys two logical interpretation: the first to have been introduced historically interprets rules and goals as first-order classical logic formulae; more recently [2], an interpretation as first-order linear logic [8] formulae has been given. The latter provides a finer reading of the dynamics of the rules and will be the logical interpretation considered in this paper.

All these semantics are correct with respect to the linear-logic interpretation: if a configuration is reachable through any of these operational semantics, then this configuration is indeed a linear-logic consequence of the initial goal. However, only the abstract semantics enjoys completeness: the purpose of all other semantics is to provide syntactic construction to force the execution to choose some particular branches. The downside is that these scheduling choices escape the declarative framework provided by logic. The programmer should ensure that the scheduler can only make the good choice, either by writing a confluent program or by relying on extra-logical traits (order of the rules, priorities, etc.) to drive the scheduler.

Focusing on completeness entails the exploration of all the logical consequences of the interpretation of the initial goal in linear logic. For this purpose, we propose angelic scheduling as an alternative execution model for CHR. Observationaly, the scheduler always makes the good choice: more precisely, if a successful (i.e., non-blocking) choice exists, it will be explored. Accessible configurations exactly match the set of logical consequences of the linear interpretation of the initial goal: the operational behavior is fully described by the linear-logic interpretation, including the control. More formally, linear logic is the most faithful logic for CHR [2], since it captures the non-monotonous evolution of configurations. Control structures like sequencing and branching have natural encoding in this logic and their usage for CHR have already been showed through the log-linear encoding of RAM machines [14].

In angelic settings, the atomicity of head consumption is not essential, in opposition to the committed-choice case. Since absence of user constraints cannot be observed, partial head consumptions just lead to silent unsuccessful computation branches. This property allows the interleaving of arbitrary computations between multiple head consumptions. Meta-interpreters for CHR rules can therefore be written by sequencing the consumption of the successive parts of the head. Specific representations can be chosen for some heads to enable user-defined indexation strategy. To reduce the combinatorial explosion among computation branches, the formalism of derivation nets is introduced: this formalism provides a graphical representation for sets of computation paths. Non-determinism during the execution of a CHR program can be intrinsic to the rule dynamics, and all choices should be explored, but the abstract operational semantics suffers from a large part of scheduling non-determinism between independent paths of the computation that should be quotiented for a tractable execution. The derivation nets are a convenient representation to define sharing strategies between computation paths to eliminate scheduling non-determinism. Two decidable sharing strategies are explored in this paper. The first strategy shows that optimal sharing is decidable but is computationaly expensive. The second one is polynomial in the

worst case and induces essentially a constant overhead in practice while being optimal relatively to a conservative interpretation of user constraint identity.

In the following section, angelic semantics is formally defined through derivation nets, and sharing strategies are presented. In Section 3, the specificity of angelic programming is formally explored through the decomposition property of head consumption and control mechanisms. In Section 4, concrete usage of angelism are given for meta-interpreter implementation, custom indexing definition and deep guards in CHR.

Note that this application of angelic semantics to CHR is only a preliminary work. A prototype implementation of angelic semantics along the lines presented here has been developped for the LCC language[1] (Linear-logic Concurrent Constraint), and despite showing good asymptotical behavior, large implementation work still has to be done to optimize execution time and memory usage. Despite divergences in the syntax, we have shown that LCC and CHR are equivalent for the abstract semantics considered here [11]. Therefore, the implemented prototype can already be used to execute the examples given in this article, modulo their trivial encoding in LCC.

Angelic semantics have been identified as the natural semantics for Concurrent Constraint (CC) programming languages [9] since the very beginning of the introduction of this language family: in this forward-chaining framework, the set of accessible computations is more natural to link with a logical interpretation than a particular computation path. However, the CC language and its angelic semantics is considered in [9] as an abstract language to reason about concurrency-related questions that can be captured in this formalism: there is no consideration about implementation. Moreover whenever CC languages have only a monotonous interpretation in classical logic, LCC and CHR handle non-monotonous traits with consumptions.

2 Angelic Semantics

In this section, derivation nets are first introduced for Constraint Simplification Rules, the fragment of CHR without propagation. Sharing strategies are introduced to algorithmically build derivation nets that reduce scheduling non-determinism. Derivation nets are then generalized to the full CHR language through the separation of CHR store between linear and persistent constraints, in the sense of the $\omega_!$ semantics [3].

An (oriented) *multigraph* is a pair $(V; \mathbf{i})$ where V is a set of *vertices* and $\mathbf{i} : V \times V \to \mathbb{N}$ is an *incidence function* giving the weight of the *edge* between each pair of vertices (with the convention that identifies the absence of edge with an edge of weight 0). Equivalenty, $\mathbf{i}$ is a multiset of binary edges in $V \times V$. For each vertex v, the multiset of *prevertices* ${}^\bullet v$, vertices that lead to v, is defined by the characteristic function $u \mapsto \mathbf{i}(u, v)$ and the multiset of *postvertices* $v^\bullet$, vertices that come from v, is defined by the characteristic function $u \mapsto \mathbf{i}(v, u)$.

[1] `http://contraintes.inria.fr/~tmartine/silcc`

A multigraph is *bipartite* if V is the disjoint union of two sets $V_1 \uplus V_2$ such that $\mathbf{i}(v, v') = 0$ for all $v, v' \in V_i$ for $i = 1$ or 2. An (oriented) *multihypergrap*h is a tuple (V, E, i) such that $(V \uplus E, i)$ is a bipartite multigraph: V is the set of the *vertices of the multihypergraph* and E are the *hyperarcs*. For each hyperarc $e \in E$, ${}^\bullet e$ is the set of *input vertices* of e and $e^\bullet$ is the set of *output vertices* of e. A *labeled multihypergraph* is a tuple (V, E, i, ℓ) such that (V, E, i) is a multihypergraph and $\ell : V \uplus E \to A$ is a mapping from vertices and hyperarcs to an alphabet of *labels* A.

2.1 Derivation nets for Constraint Simplification Rules (CSR)

Given a language for built-in constraints $\mathcal{L}_b$ equipped with a constraint theory $\mathcal{T}$ and a language for user-defined constraints $\mathcal{L}_u$, a CSR program is a set of constraint simplification rules.

Definition 1. *A* constraint simplification rule *has the form*

$$n @ H \Leftrightarrow G | B_b, B_u$$

where n is the name *of the rule, the* head *H is a multi-set of user-defined constraints, the* guard *G is a built-in constraint, and the* body *is a conjunction of a built-in constraint B_b and a multi-set of user-defined constraints B_u.*

Consider the following rules describing the calculus of a two-dimensional scalar product with a concurrent product.

Example 1 (Concurrent two-dimensional scalar product).

```
init @ scalar(X1, Y1, X2, Y2, P) ⇔
  product(X1, X2, X), product(Y1, Y2, Y), sum(X, Y, P) .
product @ product(A, B, C) ⇔
  V is A * B, value(C, V).
sum @ sum(A, B, C), value(A, VA), value(B, VB) ⇔
  V is VA + VB, value(C, V) .
```

There are essentially two possible derivations in the abstract semantics from a query `scalar(X1, Y1, X2, Y2, P)`, revealing scheduling non-determinism, depending upon which of the two products is evaluated first: `product(X1, X2, X)` or `product(Y1, Y2, Y)`.

We introduce derivation nets to describe sharing strategies which quotient these scheduling choices.

Definition 2. *A derivation net for a CSR program P is a labeled multi-hypergraph $(V, E, \mathbf{i}, \ell)$, where the vertices V are labeled with built-in or user-defined constraints and the hyperarcs E are labeled with rule names ($\ell : V \uplus E \to \mathcal{L}_b \uplus \mathcal{L}_u \uplus \mathcal{N}$), such that for each hyperarc $e \in E$, there exists a rule $\langle n @ H \Leftrightarrow G | B_b, B_u \rangle \in P$ and a renaming ρ for fresh variables occurring in the rule with*

- $\ell(e) = n$,
- $\ell({}^\bullet e) = H\rho \uplus G'$,
- $\ell(e^\bullet) = B_b\rho \uplus B_u\rho \uplus G'$.

with G' a logical consequence of G under the hypotheses of the theory $\mathcal{T}$.

The following derivation net shows the quotiented derivation path for the scalar product.

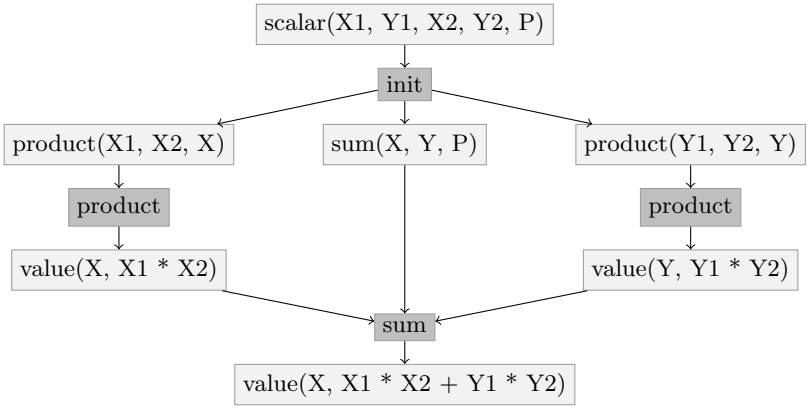

Fig. 1. Derivation net for the query `scalar(X1, Y1, X2, Y2, P)` with the scalar product program (Example 1)

Derivation nets can be equipped with a Petri-net semantics: vertices can be viewed as places marked with tokens that give their number of occurrences in the constraint store. Hyperarcs give the transitions between the places. Compared to the interpretation of CHR programs in Petri-nets [1] that interprets programs themselves as (a colored extension of) Petri-nets independently from the execution, the nets considered here give interpretations for partial executions of programs and grow as long as the execution continues.

Definition 3. *A* marking *is a multiset of vertices.* Derivations *are given by a binary relation $\to_{\mathrm{d}}$ between markings such that $m \to_{\mathrm{d}} m'$ if there exists a hyperarc e such that ${}^\bullet e \subseteq m$ (*i.e.*, for all v, ${}^\bullet e(v) \leqslant m(v)$) and for all v, $m'(v) = m(v) - {}^\bullet e(v) + e^\bullet(v)$.*

Markings are multisets of user constraints and built-in constraints: as such, they can be identified to CHR configurations.

Theorem 1 (Correction). *If there exists a derivation $m \to_{\mathrm{d}} m'$, then there exists a transition in the abstract semantics from the configuration m to the configuration m'.*

Moreover, a derivation net can always be extended with a new hyperarc for any possible transition, leading to new vertices according to the goal of the associated rule. By iterating this construction, it is possible to define a potentially infinite derivation net representing all the possible transitions.

Theorem 2 (Completeness). *For any initial configuration m, there exists a (possibly infinite) derivation net such that if m' is a configuration accessible from m in the abstract semantics, then m' is a marking accessible from m by derivation.*

Angelic execution consists therefore in the iterative construction of such a complete derivation net, keeping only the hyperarcs which are involved in a reachable marking from the initial configuration. Since the exploration is potentially infinite, the exploration should be done in breadth first to give an equal chance of execution to every computation path.

2.2 Sharing strategies

Derivation nets do not structurally force any sharing to reduce scheduling non-determinism. This is typically the case for *simpagation* rules: a simpagation rule is of the form $n@H_1 \backslash H_2 \Leftrightarrow B$ where H_1 is a *persistent head*. Such a rule has the same logical interpretation as $n@H_1, H_2 \Leftrightarrow H_1, B$. Two firings of simpagation rules with a common persistent head can lead to two computation paths whether which rule is fired before the other.

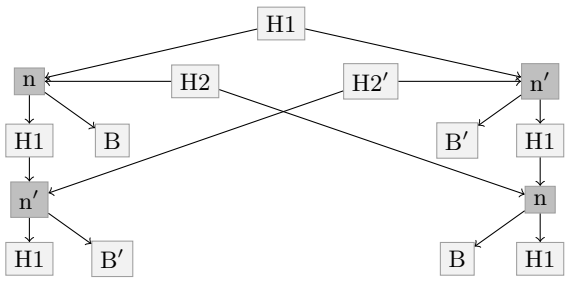

Fig. 2. Derivation net without sharing for the query `H1, H2, H2'` with two simpagation rules `n @ H_1 H_2 ⇔ B` and `n' @ H_1 H_2' ⇔ B'`

This non-determinism can be reduced considering the derivation net where each simpagation rule is a hyperarc such that the vertex of the persistent head is the same both for input and output.

The iterative construction of such a derivation net can be done by sharing all equal user constraints to the same vertex: interpreting the derivation net as a Petri net, testing the reachability of a hyperarc reduces to testing the reachability in a Petri net, which is decidable [12] but computationaly expensive [10].

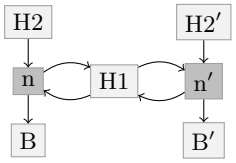

Fig. 3. Derivation net with sharing for the query `H1, H2, H2'` with two simpagation rules `n @ H_1 H_2 ⇔ B` and `n' @ H_1 H_2' ⇔ B'`

Proposition 1. *The complete derivation net where all hyperarcs are reachable and all equal user constraints are shared to the same vertex can be iteratively constructed by solving an EXPSPACE-complete problem for each new hyperarc.*

However, in the context of derivation nets where all cycles are trivial, that is to say that every cycle consist in only one hyperarc having some vertices both as input and output, then there exists a log-linear algorithm to decide the reachability. The case of trivial cycles is particularily important as those cycles appear naturally when considering simpagation rules.

- Preparation: at each new vertex creation v0, computes a table $t(v_0)$ which associates each ancestor vertice v (outside trivial cycles) to its potential immediate successor hyperarc e (there is at most one!): $t(v_0) : v \mapsto e$ With balanced binary trees, logarithmic time cost for each vertex.
- To check if a binary hyperarc e_0 between v_0 and v_1 introduces a conflict:
  1. choose one of the predecessor vertex, say v_0, (preferably the one with least ancestors)
  2. let $t \leftarrow t(v_1) + (v_1 \mapsto e_0)$
  3. begin with $v \leftarrow v_0$,
  4. for each predecessor vertex v' of each predecessor hyperarc e of v,
  5. if $t(v')$ is defined, succeeds if $t(v') = e$ or v' in trivial cycle, else fails,
  6. if not, let $t \leftarrow t + (v' \mapsto e)$ and recursively go to 4 for $v \mapsto v'$.

In worst case, logarithmic cost (table search) for each ancestor.

In practice, either hyperarcs between neighbours ($t(v_0)$ is often defined) or hyperarcs between a vertex and a top-level ask (with few ancestors).

This algorithm gives a polynomial construction for completing the derivation net at each execution step.

2.3 Derivation nets in presence of Propagation Rules

As far as the abstract semantics is concerned, propagation rules $n@H \Rightarrow G|B_b, B_u$ are shortcuts for $n@H \Leftrightarrow G|H, B_b, B_u$: the head is restored after firing the rule. This non-consumption leads to trivially infinite computation paths that can be avoided with the $\omega_!$ semantics [3]. In terms of derivation nets, distinguishing the set of vertices between linear and persistent constraints make construction rules

of derivation nets being refined to prevent this trivial non-termination. The strict output vertices of a hyperarc are marked as persistent if and only if all the input vertices are persistent or if it is a propagation (that is to say, if all input vertices are in a trivial cycle).

3 Angelic Programming

3.1 Head Decomposability

In the abstract semantics, a CHR rule can only observe the presence of a user constraint, not the absence: firing is monotonic relatively to the store. As a consequence, if observations are restricted to the side effects of the firing of some rules, silent partial consumption of the head of these rules cannot be observed. Such an observable is motivated by the fact that the body of fired rules is the place where side effects can happen. In particular, multiple headed rules can be rewritten in 2-headed rules by the introduction of fresh intermediary user constraints (carrying the context variables if any).

The rule

```
a, b, c, d ⇔ print("side effect")
```

and the set of rules

```
a ⇔ f1
f1, b ⇔ f2
f2, c ⇔ f3
f3, d ⇔ print ("side effect")
```

are equivalent provided that `f1, f2, f3` are fresh user constraints that do not appear elsewhere neither in the program nor in the initial goal.

This equivalence does not hold in general with a committed-choice scheduler since premature consumptions of `a` and `b` can prevent other rules to be fired even if `c` and `d` never appear. On the contrary, premature consumptions in angelic settings will only lead to blocking computation branches that will not prevent other branches to be explored. This property can benefit to the implementation: only 2-headed rules have to be considered. More precisely, all CHR rules can be translated to rules with two heads where one of them is an intermediary user constraint.

Such a translation makes trivial cycles of simpagations become non-trivial: the algorithm presented above can nevertheless be adapted in this case, since all hyperarcs involved in the cycle only introduce intermediary user constraints. Therefore, a hyperarc cannot be unreachable due to the consumption of such constraints by another rules.

3.2 Controlling the Angelism

User constraints can be introduced to explicitely sequence the execution of rules. The following program produces as side-effect a unspecified permutation of `a`, `b`, `c` when launched with the goal start.

```
start ⇔ a, b, c.
a ⇔ print("a").
b ⇔ print("b").
c ⇔ print("c").
```

The order can be fixed by the introduction of fresh intermediary constraints to mark the step of the sequence (carrying the context variables if any).

```
start ⇔ s0, a, b, c.
a, s0 ⇔ s1, print("a").
b, s1 ⇔ s2, print("b").
c, s2 ⇔ print("c").
```

Operationaly, such an explicit sequencing forces the derivation net to have a linear path instead of branching hyperarcs. Formally, the sequence is coded declaratively in the logic instead of being left to the conventional implementation of the comma sequence operator.

4 Applications

4.1 Angelism for CHR∨

Angelic execution can be seen as a search among scheduling. Therefore, CHR∨ rules where there can be multiple bodies, leading to a search for a successful one, can be encoded as multiple rules with the same head.

The rule `N @ H ⇔ G | B1 ; ...; Bn.` is encoded as the set of n rules `N1 @ H ⇔ G | B1.`, ..., `Nn @ H ⇔ G | Bn.` Angelic execution ensures that consequences of `B1`, ..., `Bn` are explored.

4.2 Meta-interpreters

The decomposability of heads allow CHR meta-interpreters to be conveniently written. Suppose that a rule is coded with a user constraint `rule(N @ H ⇔ G | B)` where `H` is a list of heads and with a proper encoding for the body `B` where user constraints are marked with the functor `ucstr`, then the following rules code a meta-interpreter.

```
first_head @ rule(_N @ H ⇔G, B), ucstr(H0) ⇔
  copy_term((H, G, B), ([H0 | T0], G0, B0)) |
  match(T0, G0, B0).
matching_end @ match([], G, B) ⇔call(G) |
  call(B).
matching_cont @ match([H | T], G, B), ucstr(H) ⇔
  match(T, G, B).
```

This meta-interpreter uses the decomposability of rules with match as intermediary user constraint.

Example 2. The scalar product example (Example 1) is encoded into the following constraints. A derivation net for the meta-interpretation of this program is given in Fig. 4.

```
rule(init @ [scalar(X1, Y1, X2, Y2, P)] ⇔true |
  ucstr(product(X1, X2, X)),
  ucstr(product(Y1, Y2, Y)),
  ucstr(sum(X, Y, P))).
rule(product @ [product(A, B, C)] ⇔true |
  V is A * B,
  ucstr(value(C, V))).
rule(sum @ [sum(A, B, C), value(A, VA), value(B, VB)] ⇔true |
  V is VA + VB,
  ucstr(value(C, V))).
```

4.3 User-defined Indexation

Thanks to the dependency book-keeping operated by derivation nets, rules can reformulate user constraints to another form while keeping the dependency relation between the original user constraint and the reformulation. For instance, suppose that the underlying implementation only makes indexing on the principal functor of the user constraint arguments. The following rule decomposes a nested term in a user constraint into another user constraint with this term as root argument.

```
c(f(X)) ⇔ cf(X) .
```

Then consuming `cf(X)` in another rule is equivalent to consuming `c(f(X))`: therefore, rules having heads matching on `c(X)` in general and rules having heads matching on `cf(X)` in particular can coexist and consume observationnaly the same user constraints.

4.4 Deep Guards

In a previous paper [6], we proposed a framework to extend the built-in guard language in CHR with a mechanism allowing to trigger CHR computation during guard entailment checking. This framework suffers from the trigerred computation having to satisfy sanity conditions not to pollute the store in case of non-entailment. Deep guards, that is to say CHR rules whose guards involve arbitrary CHR computations, can be trivially implemented with angelism. In the general form, a rule with a deep guard is written as follows.

```
H ⇔ (G => C) | B.
```

with the convention that the rule can only be fired if, once `H` has been consumed, the CHR goal `G` entails the CHR store `C`. Such a rule can be rewritten as follows, where mark is fresh (carrying the context variables if any).

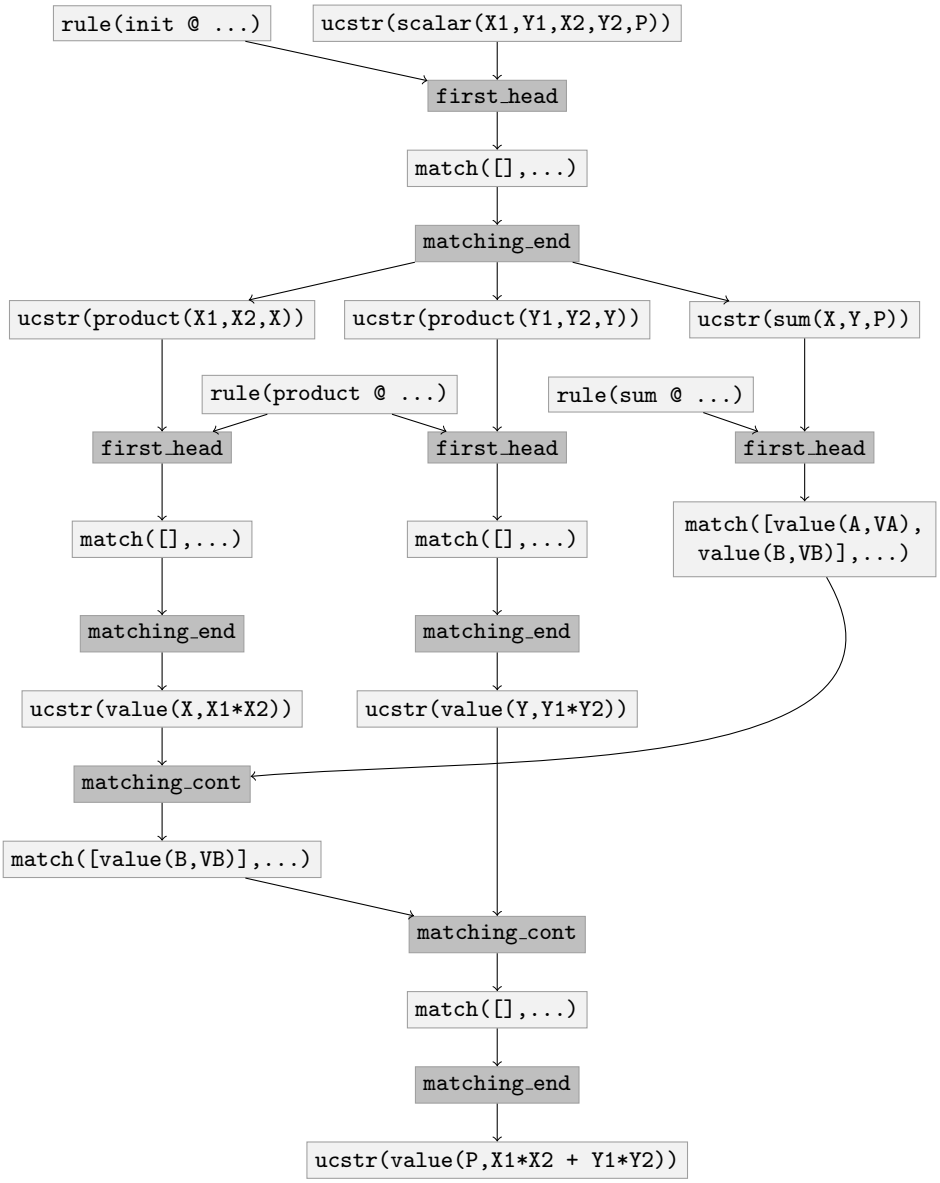

Fig. 4. Meta-interpretation of the query `ucstr(scalar(X1,Y1,X2,Y2,P))` with the meta-interpreted program given in Example 2

```
H ⇔ G, mark.
mark, C ⇔ B.
```

This is only correct in the case of an angelic execution since `H` is consumed before the execution of `G`: if this execution does not lead to `C`, other branches of execution should be explored.

5 Conclusion

We have described a new execution model for CHR, the angelic semantics, which computes all the reachable configuration from an initial CHR goal. We introduced a notion of derivation nets for graphical represention of CHR execution paths. These derivation nets allow the description of sharing strategies which make the angelic semantics tractable in practice. In these settings, we illustrate how angelic semantics can result to a fully declarative language, where control is captured by the logical interpretation. Proposed applications give natural solutions in the angelic execution model to questions which are still open with committed-choice: the existence of CHR meta-interpreters, the redefinition of specific user constraint representations, for indexation in particular, and more generally the interleaving of arbitrary computation between head consumption, allowing deep guards. The implementation still has to be done in the light of what has been already implemented for LCC. This work can begin with a simple meta-interpreter of CHR written in angelic LCC. However, even if we believe that such an implementation is possible, a lot of work remains to be explored in terms of compilation techniques to make the performance competitive with committed-choice implementations. This work is a move forward to more declarativity, for reducing the gap between the logical interpretation and the effective implementation of the semantics. We hope for more theoretical development of algorithms taking benefits of the angelic semantics, as well as progress for implementation efficiency.

References

1. Hariolf Betz. Relating coloured petri nets to constraint handling rules. In *Proceedings of the forth Constraint Handling Rules Workshop CHR'07*, pages 33–47, 2007.
2. Hariolf Betz and Thom W. Frühwirth. A linear-logic semantics for constraint handling rules. In *Proceeding of CP 2005, 11th*, pages 137–151, 2005.
3. Hariolf Betz, Frank Raiser, and Thom Frühwirth. A complete and terminating execution model for Constraint Handling Rules. *Theory and Practice of Logic Programming*, 10(4-6):597–610, 2010.
4. Leslie De Koninck, Tom Schrijvers, and Bart Demoen. User-definable rule priorities for CHR. In *Proceedings of PPDP'07, International Conference on Principles and Practice of Declarative Programming, Wroclaw, Poland*, pages 25–36. ACM Press, 2007.

5. Gregory J. Duck, Peter J. Stuckey, Maria Garcia De La Banda, and Christian Holzbaur. The refined operational semantics of constraint handling rules. In *In 20th International Conference on Logic Programming (ICLP'04*, pages 90–104. Springer-Verlag, 2004.
6. François Fages, Cleyton Mario de Oliveira Rodrigues, and Thierry Martinez. Modular CHR with ask and tell. In Thom Frühwirth and Tom Schrijvers, editors, *Proceedings of the fifth Constraint Handling Rules Workshop CHR'08*, 2008.
7. Thom W. Frühwirth. *Constraint Handling Rules*. Cambridge University Press, 2009.
8. Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50(1), 1987.
9. Radha Jagadeesan, Vasant Shanbhogue, and Vijay A. Saraswat. Angelic nondeterminism in concurrent constraint programming. Technical report, Xerox Parc, 1991.
10. R. J. Lipton. The reachability problem requires exponential space. Technical Report 62, New Haven, Connecticut: Yale University, Department of Computer Science, Research, January 1976.
11. Thierry Martinez. Semantics-preserving translations between linear concurrent constraint programming and constraint handling rules. In *Proceedings of PPDP'10, International Conference on Principles and Practice of Declarative Programming, Edinburgh, UK*, pages 57–66. ACM, 2010.
12. Ernst W. Mayr. An algorithm for the general petri net reachability problem. In *Proceedings of the thirteenth annual ACM symposium on Theory of computing*, STOC '81, pages 238–246, New York, NY, USA, 1981. ACM.
13. Jon Sneyers, Wannes Meert, Joost Vennekens, Yoshitaka Kameya, and Taisuke Sato. CHR(PRISM)-based probabilistic logic learning. *Theory and Practice of Logic Programming*, 10(4-6):433–447, 2010.
14. Jon Sneyers, Tom Schrijvers, and Bart Demoen. The computational power and complexity of constraint handling rules. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 31(2):1–42, 2009.
15. Jon Sneyers, Peter Van Weert, Tom Schrijvers, and Leslie De Koninck. As time goes by: Constraint Handling Rules – a survey of CHR research between 1998 and 2007. *Theory and Practice of Logic Programming*, 2, January 2010.

Towards a Generic Trace for Rule Based Constraint Reasoning

Armando Gonçalves[1,2], Pierre Deransart[2], Marcos Aurélio Almeida da Silva[3] and Jacques Robin[1]

[1] Universidade Federal de Pernambuco, Recife, Brazil
[2] INRIA, Centre de Paris-Rocquencourt, France
[3] Université Pierre-Marie Curie, Paris, France

Abstract. CHR is a very versatile programming language that allows programmers to declaratively specify constraint solvers. An important part of the development of such solvers is in their testing and debugging phases. Current CHR implementations support those phases by offering tracing facilities with limited information. In this paper, we propose a new trace for CHR which contains enough information to analyze any aspects of CHR$^\vee$ execution at some general abstract level. This approach is based on the idea of generic trace. Such a trace is formally defined as an extension of the $\omega_r^\vee$ semantics. It is currently prototyped in a SWI Prolog based CHR implementation.

1 Introduction

CHR (Constraint Handling Rules)[9] is a uniquely versatile and semantically well-founded programming language. It allows programmers to specify constraint solvers in a very declarative way. An important part of the development of such solvers is in their testing and debugging phases. Current CHR implementations support those phases by offering tracing facilities with limited information.

In this paper, we propose a new trace for CHR which contains enough information, including source code ones, to analyze any aspects of CHR$^\vee$ execution at some abstract level, general enough to cover several implementations and source level analysis. Although the idea of formal specification based tracer is not new (see for example [14]), the main novelty leads in the generic aspect of the trace. Most of the existing implementations of CHR like in [11,13,2,20] include a tracer with specific CHR ports, but without formal specification, nor consideration with regards to different kind of usages than debugging.

The notion of generic trace has been informally introduced and used for defining portable CLP(FD) tracer and portable applications [1,15]. We propose here to use this approach to specify a tracer for rule based inference engine like CHR$^\vee$. A generic trace has three main characteristics: it is "high level" in the sense that it is independent from particular implementations of CHR, it has a specified semantics (Observational Semantics) and can be used to implement debugging tools or applications. In [18] it is shown that it can be adapted for software component based programming.

We present a generic trace for CHR$^\vee$ based on its refined operational semantics $\omega_r^\vee$ [5], and describe a first prototype developed for SWI-Prolog CHR$^\vee$ engine. The implementation consists of combining the original trace of the SWI engine with source code information to get generic trace events, and then, allowing the user to filter these events using a SQL-based language.

This paper is organized as follows. Section 2 gives a short introduction to generic traces. Section 3 presents CHR, its $\omega_r^\vee$ semantics and the observational semantics, OS-CHR$^\vee$, defining the generic trace. Section 4, the CHR-SWI-Prolog based prototype. Section 5 presents the experimentation. Discussion and conclusions are in the two last sections.

A full version of this paper can be found in [12].

2 Generic Trace, Observational Semantics, and Subtrace

The concept of *generic trace* has been first introduced in [15], formally defined in [6,7], and a first application to CHR presented in [18]. A generic trace is a trace with a specification based on a partial operational semantics applicable to a family of processes. We give here its main characteristics and the way to specify a generic trace.

2.1 Preliminaries

A *trace* consists of an initial state s_0 followed by an ordered finite or infinite sequence of *trace events*, denoted $< s_0, \overline{e} >$. $\mathcal{T}$ is a set of traces (finite or infinite). A *prefix* (finite, of size t) of a trace $T =< s_0, \overline{e_n} >$ (finite or infinite, here of size $n \geq t$) is a partial trace $U_t =< s_0, \overline{e_t} >$ which corresponds to the t first events of T, with an initial state at the beginning. $\mathcal{T}$ may contain any prefixes of its elements.

A trace can be decomposed into segments containing trace events only, except prefixes which start with a state. An associative operator of concatenation will be used to denote sequences concatenations (denoted ++). The neutral element is [] (empty sequence). A segment (or prefix) of size 0 is either an empty sequence or a state.

Traces are used to represent the evolution of systems by describing the evolution of their state. A state of the system is described by a given finite set of parameters and a state corresponds to a set of values of parameters. Such states will be said *virtual* as they correspond to states of the observed system, but they are not actually traced. We will thus distinguish between actual and virtual traces.

- the *actual traces* ($\mathcal{T}^w$) are a way to observe the evolution of a system by generating traces. The events of an actual trace have the form $e = (a)$ where a is an *actual state* described by a set of *attributes values*. An actual states is described by a finite set of attributes. Actual traces corresponds to sequences of events produced by a tracer of an observed system. They usually encode virtual states changes in a synthetic manner.

– the *virtual traces* ($\mathcal{T}^v$) corresponds to the sequence of the virtual states such that for each transition in the system between two virtual states, it corresponds an actual trace event. The virtual trace events have the form $e = (r, s)$ where r is a *type of action* associated with a state transition and s, called *virtual state*, the new state reached by the transition and described by a set of *parameters*. Virtual traces correspond to sequences of virtual states of the observed system which produced the actual trace, together with the kind of action which produced the virtual state transition.

The correspondence between both kinds of traces is specified by two functions $E : \mathcal{T}^v \rightarrow \mathcal{T}^w$ and $I : \mathcal{T}^w \rightarrow \mathcal{T}^v$, respectively the *extraction* and the *reconstruction function*, as illustrated by the figure 1.

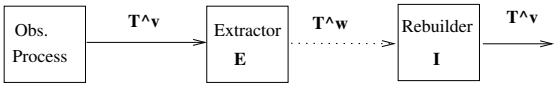

Fig. 1. Extraction and Reconstruction

The idea is that the actual generated trace contains as much information as possible in such way that the virtual trace can be reconstructed from the actual one. In other words, the extraction is done without loss of information. Such a property of the traces is called *faithfulness* and, if we denote Id_v (resp. Id_w) the identity between virtual traces (resp. actual traces), it states that $E \circ I = Id_v$ (composition) or $E = I^{-1}$, and $I \circ E = Id_w$ (or $I = E^{-1}$).

2.2 Components in Trace Design

When designing a trace, several components must be taken into consideration. They are depicted in the Figure 2.

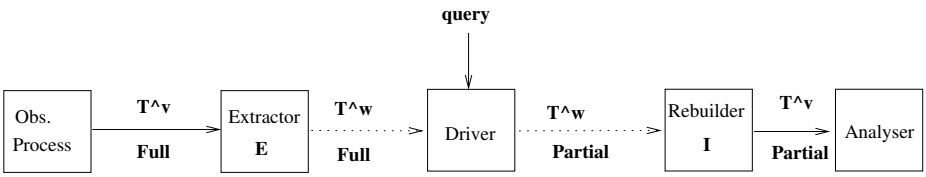

Fig. 2. Components in Trace Design

1. The observed process whose behavior is modeled by a virtual trace (sequence of successive virtual states) T^v.

2. An *extractor* component which encodes the virtual trace into the actual one T^w. This component corresponds in practice to the tracer formalized by the extraction function E.
3. The *driver* which realizes the actual trace filtering according to some *trace query*. In this paper we limit its role to select a subtrace of the so called *full trace*.
4. The *rebuilder* which may reconstruct from a full or partial actual trace a full or partial virtual trace. This is possible only if there is no loss of information (faithfulness property). The rebuilder is formalized by the reconstruction function I.
5. The *analyzer*, which corresponds to some debugging tool or particular application, working with the full trace or a partial one.

Notice that in practice the three first components may be combined in such a way that for a given query the driver may select directly a subset of the virtual trace, thus avoiding to extract and encode a full actual trace before selecting a subtrace.

In this paper we focus on three first components: observed process, extractor and driver (but restricted to a query). The two first ones correspond to the description of the evolution of the observed system defined by a state transition relation (defining its virtual traces) and the production of the corresponding actual trace, together called the *Observational Semantics*. General and formal definitions can be found in [7].

2.3 Characteristics of a Generic Trace

The idea of generic trace meets the needs of independent trace specification and portability. It is intended to specify a process or an algorithm by its observed behavior, i.e. the trace of abstracted operations that it is expected to implement. The level of description must be general enough to include family of processes, and the level of granularity must be sufficiently refined to be used by a family of applications. This may be the case for example for applications such as monitoring, debugging, visualization tools, or any application using the generic trace.

Definition 1 (Generic Trace (GT)).
Given a family of processes $p \in P$, each of them equipped to produce traces $\mathcal{T}_p$, a set of traces $\mathcal{T}_g$ is generic if, for each process p in the family, there exists an abstraction of its traces which is a subtrace of $\mathcal{T}_g$, that is:
$\forall p \in P,\ \exists\ \mathcal{T}$ *such that* $Abs(\mathcal{T}_p, \mathcal{T}) \wedge Sub(\mathcal{T}_g, \mathcal{T})$.

A more formal definition can be found in [12].
Three questions are then worth posing:

– How to ensure that the trace produced by some process is compliant with the GT?

– Can the GT be used in application development, with the guarantee that the application will work with any compliant process?
– Can the GT be extended to handle more processes in such a way that existing applications will still work?

Here are some possible answers.

Compliance to the Generic Trace

A trace of a process is compliant with the GT if it satisfies the definition 1, i.e. there exists a subtrace of the GT which is an abstraction of a subtrace of the one of the process. It is thus possible either to implement straightforwardly the GT as it is (in this case the process produces exactly the GT), or to prove that the traces a process p may generate verify $\exists \mathcal{T}', Abs(\mathcal{T}_p, \mathcal{T}') \wedge Sub(\mathcal{T}_g, \mathcal{T}')$.

Building tools with the Generic Trace

The interest of a generic trace is that it facilitates the development of tools that can be used with all compliant processes. The development is made considering that the tool uses at least a sub-GT covering sufficiently many processes. Thus it is possible to adapt the tool to the process p by abstracting the trace generated by the process (without any modification) to get a GT. This can be done at le level of the process (the process can use any tool) or at the level of the tool (the tool can be run with this particular process). The figure 3 illustrates these two ways to adapt processes with compliant tracer and tools.

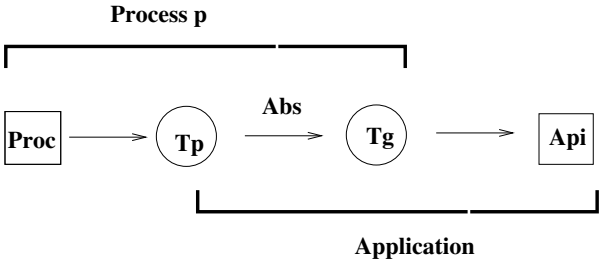

Fig. 3. Use of a Generic Trace: process or application adaptation

The fact that the GT has a formal specification makes it possible to realize a prototype (executable specification) which shall be itself a new compliant process. It is thus possible to use such a prototype to develop and test tools. This development method guarantees that any tool made on the top of the GT will be able to work with any compliant processes.

Generic Trace Extensions

As long as an extension of the GT preserves the fact that a process is compliant w.r.t. a subtrace of the extended GT, they still are compliant w.r.t. the extended GT. It is sufficient to ensure that any GT extension preserves the sub-

traces. This guarantees that the compliant processes will continue to be usable by tools using the original GT.

2.4 Generic Trace Specification

By definition, if the observational semantics of a generic trace is faithful, a subtrace of a virtual trace is a subtrace of the corresponding actual trace, from which the original virtual subtrace can be reconstructed. This is illustrated by figure 2. A query applied to the actual trace selects a partial actual trace in such a way that the resulting partial trace can be transformed in a partial virtual trace (the one from which the partial actual trace could be extracted). A practical consequence is that the definition of a generic trace should be given by a faithful observational semantics.

In practice, the generic trace specification consists of an operational semantics corresponding to some abstract level of process observation, instrumented to produce an actual trace. The level of description (granularity of the events) should be chosen in such a way that this abstract operational semantics can be abstracted from each particular semantics of each process of the family. Symmetrically, it is requested that the abstract operational semantics can be "implemented" in each process of the family.

The faithfulness property of the observational semantics guarantees that the generic actual trace preserves the whole information concerning the process behavior, which can be deduced from the observation level corresponding to the given operational semantics.

3 Generic Trace for CHR$^\vee$

In this section we introduce the generic trace proposed for CHR$^\vee$. It is based on the refined Theoretical Operational Semantics for CHR, $\omega_r^\vee$, as defined in [5].

Such semantic is declarative enough to cover most of the CHR implementation. It is the case for ECLiPSe Prolog [2] and SWI-Prolog [20] whose operational semantics can be viewed as a refinement of $\omega_r^\vee$ (conversely $\omega_r^\vee$ can be viewed as an abstraction of the semantics of these implementations).

3.1 Operational Semantics $\omega_r^\vee$

We define CT as the constraint theory which defines the semantic of the built-in constraints and thus models the internal solver which is in charge of handling them. We assume it supports at least the equality built-in. We use $[H|T]$ to indicate the first (H) and the remaining (T) terms in a list or stack, + for pushing elements into stack, ++ for sequence concatenation and [] for empty sequences. We use the notation $\{a_0, \ldots, a_n\}$ for both bags and sets. Bags are sets which allow repeats. We use $\cup$ for set union and $\uplus$ for bag union, and $\{\}$ to represent both the empty bag and the empty set. The identified constraints have the form $c\#i$, where c is a user-defined constraint and i a natural number.

They differentiate among copies of the same constraint in a bag. We also assume the functions $chr(c\#i) = c$ and $id(c\#i) = i$.

An execution state $\mathcal{E}$ is a tuple $\langle A, S, B, T\rangle_n$, where

- A is the execution stack;
- S is the UDCS (User Defined Constraint Store), a bag of identified user defined constraints;
- B is the BICS (Built-in Constraint Store), a conjunction of constraints;
- T is the Propagation History, a set of sequences for each recording the identities of the user-defined constraints which fired a rule;
- n is the next free natural used to number an identified constraint.

Current alternatives are denoted as ordered sequence of execution states, $\mathcal{L} = [\mathcal{E}_1, \mathcal{E}_2, ...\mathcal{E}_n]$ where $\mathcal{E}_1$ is the active execution state and $[\mathcal{E}_2, ..., \mathcal{E}_n]$ the remaining alternatives.

The initial configuration is represented by $\mathcal{E}_0 = [\langle A, \{\}, true, \{\}\rangle_1]$. The top of execution stack A is a constraint that will be processed and its initial value is determined by the initial goal. The transitions are applied non-deterministically until no transition is applicable. These transitions are defined as follows:

Solve+Wake $[\langle [c|A], S, B, T\rangle_n |\mathcal{L}] \mapsto [\langle wakeup(S, c, B) + A, S, c \wedge B, T\rangle_n |\mathcal{L}]$, where c is built-in and $wakeup(S, c, B)$ is a function implementing the *wake-up policy*[9]

Activate $[\langle [c|A], S, B, T\rangle_n |\mathcal{L}] \mapsto [\langle [c\#n : 1|A], \{c\#n\} \uplus S, B, T\rangle_{n+1} |\mathcal{L}]$, where c is user-defined constraint

Reactivate $[\langle [c\#i|A], S, B, T\rangle_n |\mathcal{L}] \mapsto [\langle [c\#n : 1|A], S, B, T\rangle_n |\mathcal{L}]$, where c is user-defined constraint

Apply $[\langle A, H_1 \uplus H_2 \uplus S, B, T\rangle_n |\mathcal{L}] \mapsto [\langle C + A, H_1 \uplus S, e \wedge B, T'\rangle_n |\mathcal{L}]$ where exists a rule $r@H_1' \backslash H_2' \Leftrightarrow g|C$ and a matching substitution e, such that $chr(H_1) = e(H_1')$, $chr(H_2) = e(H_2')$ and $CT \models B \supset \exists(e \wedge g)$ and the entry $\{(r, id(H_1)++id(H_2))\} \notin T$ and $T' = T \cup \{(r, id(H_1)++id(H_2))\}$.

Drop $[\langle [c\#i : j|A], S, B, T\rangle_n |\mathcal{L}] \mapsto [\langle A, S, B, T\rangle_n |\mathcal{L}]$, where there is no occurrence j for c in the program.

Default $[\langle [c\#i : j|A], S, B, T\rangle_n |\mathcal{L}] \mapsto [\langle [c\#i : j + 1|A], S, B, T\rangle_n |\mathcal{L}]$, if no other transition is possible in the current state.

Split $[\langle [c_1 \vee ... \vee c_m|A], S, B, T\rangle_n |\mathcal{L}] \mapsto [\sigma_1, ..., \sigma_m|\mathcal{L}]$, where $\sigma_i = \langle [c_i|A], S, B, T\rangle_n$, for $1 \leq i \leq m$. This transition implements depth-first, other search strategies can be implemented by easily changing this definition.

Fail $[\mathcal{E}|\mathcal{L}] \mapsto \mathcal{L}$, This transition is called automatically if $\mathcal{E}$ is a failed state. By definition a failed state occurs when the Built-in store is false.

3.2 Generic Trace

We introduce here informally the generic trace of CHR$^\vee$. Each transition in the $\omega_r^\vee$ semantics should generate an actual trace event.

- **Wake** *a built-in constraint (BIC) is solicited*
  This event has 4 attributes and will be written as: $[Wake, c, wakeup(S, c, B), n]$. It will happen when a solve transitions is performed.

- **ActivateRDC** *activate a Rule defined constraint (RDC)*
  This event has 3 attributes and will be written as: $[ActivateRDC, c, n]$. It will happen when a solve transitions is performed, getting the RDC c from the top of the execution stack and activating it.
- **ReactivateRDC** *Activate a Rule defined constraint with justification*
  This event has 4 attributes, including: a RDC and a Wake event, which is the justification of the most recent Reactivate transition. It will be written: $[Reactivate, c\#i : j, [Wake, b, n], m]$.
- **TryRule** *attempt to apply a Rule*
  This event has 7 attributes, including: rule name, the active constraint, the constraints that match the *keep*, the constraint that matches the *remove*, the guard. It will be written:
  $[TryRule, ruleName, activeConstraint, keep, remove, guard, n]$,
  where $activeConstraint$ has the form $c\#i : j$.
- **ApplyRule** *apply the rule*
  After trying a rule, if the guard is true, the apply rule event will trigger. It has 7 attributes, including: the last corresponding TryRule event, added RDCs, preserved Head, removed RDCs, added BICs. The last three attributes are obtained from the rule's body. It will be written: $[ApplyRule, [TryRule, ...], addedRDCs, keep, remove, addedBICs, n]$
- **Drop** *drop a constraint*
  This event has 3 attributes, including a RDC, and will be written: $[Drop, c\#i : j, n]$. It corresponds to the end of the execution of c, where c was an active constraint.
- **Default** *numbering incrementation*
  Occurs when virtual states numbering is incremented. It has 3 attributes and will be writen: $[Default, j, n]$, where j is the new value.
- **Split** *create a disjunction*
  Occurs when a rule is disjunctive. It has 3 attributes, including the ApplyRule event with the rule whose body has the disjunction. It will be writen: $[Split, [ApplyRule, ...], n]$.
- **Fail** *the rule application fails*
  Occurs when the Built-In store is false. It has 3 attributes, including the ApplyRule with the ultimate rule tested before failure. It will be written: $[Fail, [ApplyRule, ...], n]$. n is the numbering of the failed state.

All the variables which occur in the initial goal will keep their original name in all their occurrences in the generic trace. The formal definition of trace generation will be given in the next section 3.3.

3.3 Observational Semantics of CHR$^\vee$ (OS-CHR$^\vee$)

We specify the observational semantics of CHR$^\vee$, OS-CHR$^\vee$, on the top of the operational semantics of section 3.1. The current generated actual trace is denoted N, an ordered sequence of the trace events. The initial configuration will be represent as $\mathcal{E} = [\langle A, \{\}, true, \{\}\rangle_1], N = []$

Solve+Wake $[\langle [c|A], S, B, T\rangle_n |\mathcal{L}], N$
$\mapsto [\langle wakeup(S,c,B) + A, S, c \wedge B, T\rangle_n |\mathcal{L}], N{+}{+}[Wake, c, wakeup(S,c,B), n]$
where $SolveCond$
Activate $[\langle [c|A], S, B, T\rangle_n |\mathcal{L}], N$
$\mapsto [\langle [c\#n : 1|A], \{c\#n\} \uplus S, B, T\rangle_{n+1} |\mathcal{L}], N{+}{+}[ActivateRDC, c, n]$
where c is a rule-defined constraint
Reactivate $[\langle [c\#i|A], S, B, T\rangle_n |\mathcal{L}], N$
$\mapsto [\langle [c\#n : 1|A], \{c\#n\} \uplus S, B, T\rangle_n |\mathcal{L}], N{+}{+}[ReactivateRDC, c, wake(c,N)]$
where $CondReac$ (see below)
Apply.1 $[\langle [c\#i : j|A], H_1 \uplus H_2 \uplus S, B, T\rangle_n |\mathcal{L}], N$
$\mapsto [\langle [c\#i : j|A], H_1 \uplus H_2 \uplus S, B, T\rangle_n |\mathcal{L}], N{+}{+}[TryRule, r, c\#i : j, H_1, H_2, g, n]$
where $CondApp1$ (see below).
Apply.2 $\langle A, H_1 \uplus H_2 \uplus S, B, T\rangle_n, N$
$\mapsto \langle C + A, H_1 \uplus S, B, T'\rangle_n, N{+}{+}$
$[ApplyRule, tryRule(N), addRDCs(C), H_1, H_2, addBICs(C), n]$
where $CondApp2$ (see below).
Drop $[\langle [c\#i : j|A], S, B, T\rangle_n |\mathcal{L}], N$
$\mapsto [\langle A, S, B, T\rangle_n |\mathcal{L}], N{+}{+}[Drop, c\#i : j, n]$ where c is an active constraint.
Default $[\langle [c\#i : j|A], S, B, T\rangle_n |\mathcal{L}], N$
$\mapsto [\langle [c\#i : j+1|A], S, B, T\rangle_n |\mathcal{L}], N{+}{+}[Default, j+1, n]$.
Split $[\langle [c_1 \vee ... \vee c_m|A], S, B, T\rangle_n |\mathcal{L}], N$
$\mapsto [\sigma_1, ..., \sigma_m|\mathcal{L}], N{+}{+}[Split, rule(N), n]$ where $CondSplit$ (see below).
Fail $[\mathcal{E}|\mathcal{L}], N \mapsto \mathcal{L}, N{+}{+}[Fail, rule(N), n]$ where $CondFail$ (see below).

The conditions appearing in our observation semantics are defined as follows:

- $SolveCond$: c is built-in, and $wakeup(S,c,B)$ defines which CHR constraints of S are woken by adding the constraint c to the built-in store B.
- $CondReac$: the function $wake : Constraint, Trace \mapsto Wake$ is responsible for selecting the Wake event that justifies the Reactivate.
- $CondApp1$: there exists a rule $r@H_1' \backslash H_2' \Leftrightarrow g|C$ and a matching substitution e, such that $chr(H_1) = e(H_1')$, $chr(H_2) = e(H_2')$ and $\{(r, id(H_1){+}{+}id(H_2))\} \notin T$.
- $CondApp2$: C is the body of the rule $r@H_1' \backslash H_2' \Leftrightarrow g|C$. The $tryRule : Trace \mapsto TryRule$ will retrieve the TryRule event generated by Apply.1, It will search for the event in the trace log, normally the event TryRule will be one step back. $addRDCs : Body \mapsto Sequence(RDC)$ will select only the RDCs on the body; the function $addBICs : Body \mapsto Sequence(BIC)$ will select the BICs on the body. Same conditions of $Apply.1$ plus $CT \models B \supset \exists(e \wedge g)$ and $T' = T \cup \{(r, id(H_1){+}{+}id(H_2))\}$.
- $CondSplit$: where $\sigma_i = \langle [A_i|A], S, B, T\rangle_n$, for $1 \leq i \leq m$, and $rule : Trace \mapsto ApplyRule$ is a function that will retrieve the cause of the split, a disjunctive rule.
- $CondFail$: n is the numbering of the failed state $\mathcal{E}$, and $rule(N)$ is a function that will retrieve the cause of the failure.

4 Prototyping of a generic CHR$^\vee$ Trace Engine using SWI Prolog

A generic CHR$^\vee$ tracer for SWI-Prolog was developed. In Section 5.1, we introduce the SWI Prolog debug output trace produced when executing CHR rule bases. Section 5.2 presents our mapping from the produced trace into OS-CHR$^\vee$

4.1 Running Example

The generic trace will be illustrated on a simple disjunctive graph-coloring problem. The following CHR$^\vee$ rules define a graph coloring solution:

```
node1@ node(r1,C) ==> (C = r ; C = b ; C = g).
node2@ node(r2,C) ==> (C = b ; C = g).
node3@ node(r3,C) ==> (C = r ; C = b).
node4@ node(r4,C) ==> (C = r ; C = b).
node5@ node(r5,C) ==> (C = r ; C = g).
node6@ node(r6,C) ==> (C = r ; C = g; C = t).
node7@ node(r7,C) ==> (C = r ; C = b).
startGraph@ edges<=> edge(r1,r2), edge(r1,r3), edge(r1,r4),
edge(r1,r7), edge(r2,r6), edge(r3,r7), edge(r4,r5), edge(r4,r7),
edge(r5,r6), edge(r5,r7).
wrong@ edge(Ri,Rj), node(Ri,Ci), node(Rj,Cj) ==> Ci = Cj | false.
l1@ l([ ],[ ]) <=> true.
l2@ l([R|Rs],[C|Cs]) <=> node(R,C), l(Rs,Cs).
```

This CHR base handles a graph-coloring problem with at most 3 colors where any two nodes connected by a common edge must not have the same color. The constrain `node(r1,C)` means that node r1 has color C, the `startGraph` rule defines edges between nodes of a graph and the `wrong` rule assures that two nodes will have different colors. A small part of the trace from the following goal "edges, l([r1,r7,r4,r3,r2,r5,r6],[C1,C7,C4,C3,C2,C5,C6])." is depicted:

```
CHR: (1) Insert: node(r1,_G234) # <384>
CHR: (2) Call: node(r1,_G234) # <384>
CHR: (2) Try: node(r1,_G234) # <384> ==> _G234=r;_G234=b;_G234=g.
CHR: (2) Apply: node(r1,_G234) # <384> ==> _G234=r;_G234=b;_G234=g.
...
CHR: (2) Insert: node(r7,_G235) # <386>
CHR: (3) Call: node(r7,_G235) # <386>
CHR: (3) Try: node(r7,_G235) # <386> ==> _G235=r;_G235=b.
CHR: (3) Apply: node(r7,_G235) # <386> ==> _G235=r;_G235=b.
CHR: (4) Wake: node(r7,r) # <386>
CHR: (4) Try: node(r1,r) # <384>, edge(r1,r7) # <376>,
              node(r7,r) # <386> ==> r=r | false.
CHR: (4) Apply: node(r1,r) # <384>, edge(r1,r7) # <376>,
```

```
                node(r7,r) # <386> ==> r=r | false.
CHR: (3) Fail: node(r7,r) # <386>
CHR: (4) Wake: node(r7,b) # <386>
...
```

This subset of the execution is responsible for trying the value C1 and C7 as red then backtracking because C1 and C7 cannot have the same colors.

Informal definitions of the trace events of SWI-Prolog can be found here[4]. Some problems occur when an analysis of the trace is needed: the try/apply transition has no rule name, it's very difficult to link the name of the generated var with the name of the variable passed as goal since all vars were renamed and there isn't an efficient way to query it.

4.2 Understanding SWI-Prolog Trace

SWI-Prolog's default search strategy implemented is depth-first, the parameter *depth* indicates the transaction's actual level in the search tree and *id* is the constraint's unique identifier. The trace output fits the following pattern: `CHR: (depth)  Instruction: constraint(terms) #<ID>`.
Small parts of the trace will be shown and explained.

```
CHR: (0) Insert: edges # <372>
CHR: (1) Call: edges # <372>
```

The trace produced by these two ports is responsible for removing a constraint from the goal and insert it in the execution stack. Notice that in SWI's trace they always appear together.

```
CHR: (2) Exit: edge(r1,r2) # <373>
```

The computation over the active constraint is finished.

```
CHR: (3) Try: node(r7,_G235) # <386> ==> _G235=r;_G235=b.
CHR: (3) Apply: node(r7,_G235) # <386> ==> _G235=r;_G235=b.
```

The trace produced by Try and Apply‘s port only happens together and It means that a rule was tried and applied respectively.

```
CHR: (4) Wake: node(r7,r) # <386>
```

The Wake port is traced when a built-in is solved, in this case the constraint was reactivated because C7 = r.

```
CHR: (2) Redo: node(r1,b) # <412>
```

An active constraint starts looking for an alternative solution.

[4] `http:/www.swi-prolog.org/pldoc/doc_for?object=section(2,'7.4',swi('/doc/Manual/debugging.html'))`

4.3 Transforming SWI Tracer into a Generic Tracer

The SWI's output is not enough to perform a translation to OS-CHR$^{\vee}$. We do need information about what was the goal passed and access to the source-code. The inputs and outputs of the algorithm is illustrated by figure 4. The Translator's algorithm will be explained by example.

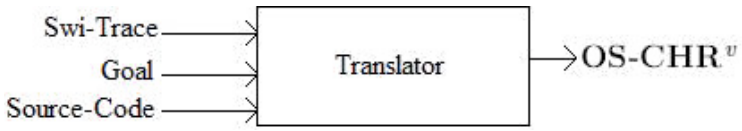

Fig. 4. Translator's Input/Output

Some ports have direct connection with OS-CHR$^{\vee}$: Call and Exit. All others ports will need a computation using the generated SWI trace. The Insert port is ignored because it is redundant with the port Call.

```
CHR: (0) Call: edges # <372> -> [ActivateRDC,[edges,372],372]
CHR: (2) Exit: edge(r1,r2) # <373> ->[Drop,[edge,r1,r2,373],373]
```

For the tryRule map, we have to look the source code and try to find what is the rule name for that transition, and while generating the trace we keep track of the active constraint.

```
CHR: (5) Try: node(r1,r) # <384>,edge(r1,r4) # <375>, node(r4,r) #
<388> ==> r=r | false. -> [TryRule, ruleName,activeConstraint,
[keep,[node,r1,r,384],[edge,r1,r4,375],[node,r4,r,388]],[remove],
[guard, [r=r]], 388]
```

ApplyRule is the most complicated map, we have to link(@) with the tryRule and check if it has a disjunctive body. In this case, we have keep track correctly of the link, with a possible failure status; it can generate a lot of trace events depending on how many constraints were added/removed, and possibly a split transition. The link function will recover the real name of the variable, in this case _G235 = C7

```
CHR: (3) Apply: node(r7,_G235) # <386> ==> _G235=r;_G235=b.
-> [ApplyRule,@TryRule,[addedRDCs],[node,r7,link(_G235)][removeRDCs],
[bic,[link(_G235) = r],[link(_G235) = b]],388] ++
[Split,@ApplyRule,388]
```

For the Wake port the we have to look to previous values of the trace and determine what BIC solving fired this transition and also a Reactivate event will be produced. In this case `C7 = r` was the cause.

```
CHR: (4) Wake: node(r7,r) # <386> -> [Wake,bicSource,388]++
[Reactivate,[node,r7,r,386],@Wake,388]
```

The Fail port will produce a Fail event with its cause, a rule that propagates to false.

```
CHR: (4) Fail: node(r4,r) # <388> -> [Fail, @ApplyRule,388]
```

The redo port only indicates a backtrack event and It's not used to produce the generic trace.

4.4 Trace Querying

The produced generic trace is represented by a sequence of Java objects. The language we choose for querying the trace is the SQL for Java Objects (JoSQL); its implementation can be found here[5].

These are some examples of query in JoSQL: (on a trace of example 4)

- SELECT * FROM trace WHERE type ='ApplyRule' AND (name='wrong' OR name='node1' OR name='node2') Will select the trace of the execution of rules: wrong, node1,node2.
- SELECT * FROM trace WHERE type ='Split' OR type ='Fail' Will select all split and fail transition.
- SELECT addedRDCs,removedRDCs,addedBICs FROM trace WHERE type ='ApplyRule'

The last query is more general and can be used by any application which need to handle a current state of the constraint store.

5 Experimentation

To evaluate our approach 3 benchmarks were set: 10-Queens, primes and a compiled example of scheduling from CHORD[4], available on its test folder, the reason for choosing a CHORD example was the complexity, more than 100 rules. All results are shown in the following table.

All the experiments were performed on a PC with Pentium Core 2 Duo processor running at 2,4 GHz, with 4 GB of RAM and 1.5GB were reserved to the Java heap. The prolog process and our traces are two different process as described by Langevine[16].

CHR$^\vee$ tracing evaluation (time)				
Problems	No Trace	Swi Trace	OS-CHR$^\vee$	**Size of the Trace (SWI/OS-CHR$^\vee$)**
scheduling	0.1s	0.2s	0.27s	0.5M / 0.5MB
primes	57s	1min	1min 05s	5.4MB / 6.3MB
10-Queens	7s	1 min 14s	1min 25s	59.7MB / 71.7MB
graphColoring	0.007s	0.025s	0.083s	14.5KB / 21KB

The following queries were done:

[5] http://josql.sourceforge.net/

```
scheduling> g []. %starts chord computation
primes> candidates(8000). %calculates primes upto 8000
10-Queens> solveall(10,N,S). % give all solutions for 10 Queens.
graphColoring> edges, l([r1,r7,r4,r3,r2,r5,r6],[C1,C7,C4,C3,C2,C5,C6]).
%graph with 7 edges
```

As we can observe the generation of trace events is very time consuming, but the feature of querying it as we parse to our definition normally adds a negligible amount of time. For the selection of objects, tests executed on JoSQL show that that a list of 1,000,000 generic trace events can be queried in about 1.5s.

6 Discussion

Several aspects of such a generic trace were explored on [18], in particular its relations with component software development, the use of the fluent calculus to prototype traces and the use of object oriented specification methods. The generic trace presented in that work is thus limited to the simple theoretical operational semantics ω_t [10] and therefore is less precise than the one given here.

Our approach of the observational semantics rely to abstract interpretation. The OS is similar to the "Observable Semantics" of Lucas [17] or the partial trace semantics of Cousot [3]. The parameters used to describe the execution states are, as expressed by Lucas, "syntactic objects used to represent the conduct of operational mechanisms". The traces are abstract representations of CHR$^\vee$ semantics which allow to take into account the sole details we want to consider as common to different implementations. The (abstraction) relations between a generic trace and the traces of specific implementations of solvers are explored in [6], together with a compliance proof method. Furthermore the generic trace contains a set of details considered as useful in several debugging tasks with several levels of refinement or observation. It could be enriched according to different needs [6], or refined without changing the semantics of the already existing one.

This way to proceed is opposite to the frequently adopted approach as, in particular, in [19], where a set of (visual) debugging tools is defined together with their input data, which consists of a restricted trace containing the minimal needed information. In our approach, we specify a semantically rich trace which can be used as input data for a potentially larger set of tools. The choice of the data to trace is made on the basis of a high level operational semantics, not on the basis of some specific debugging need. However the generic trace is designed in such a way that most of debugging tools devoted to the analysis of CHR resolution behavior may find in this trace what they need. As a consequence, based on this observational semantics, the work of implementation of the tracer and the work of designing debugging tools can be performed independently.

[6] An extensive study about the needs for constraint debugging can be found in [8].

One may however feel that implementing a full generic trace is too much work demanding or that the resulting tracer performance will be considerably slow down. It has been shown in [16] that a generic approach may have more advantages than drawbacks in the sense that there may be a good trade-off between a very detailed generic trace (based on a more refined operational semantics) and the use of a trace driver able to query efficiently the generic trace, with a significant improvement in portability of debugging tools. We have shown here, that the implementation of the CHR$^\vee$ generic trace in SWI-Prolog CHR implementation can easily be performed on the top of an existing tracer, resulting in an efficient generic tracer.

7 Conclusion

We have presented a first observational semantics of CHR$^\vee$, a formal specification of a CHR$^\vee$ generic tracer, and a first prototype based on a CHR SWI-Prolog implementation. This approach shows that the generic trace can be easily and efficiently implemented on existing CHR$^\vee$ implementations. The interest of the "generic approach" leads in the portability of analysis tools developed on the basis of this trace and the variety of possible trace based applications.

We do not claim that the CHR$^\vee$ observational semantics which is presented here is the ultimate one. More refined observational semantics could be considered or inclusion of several levels of refinements (for example combining with Prolog semantics in Prolog based implementations); we just have shown that this approach can be realistic and useful in a great variety of CHR based software development.

Future work will concern more experimentation and improvements of the generic trace, OO based CHR implementation including a generic trace, and generic trace for hybrid constraint solvers.

References

1. A. Aggoun, T. Baudel, P. Deransart, M. Ducassé, J.-D. Fekete, and N. Jussien. Generic Trace Format for Constraint Programming, GenTra4CP, Version 2.1. Technical report, INRIA Paris-Rocquencourt, École des Mines de Nantes, INSA de Rennes, Université d'Orléans, Cosytec and ILOG, July 2004. `http://contraintes.inria.fr/OADymPPaC/Public/Trace/index.html`.
2. A. Aggoun & al. ECLiPSe User Manual. Release 5.3. 2001.
3. P. Cousot and R. Cousot. Systematic design of program transformation frameworks by abstract interpretation. In *Proc. of POPL 2002*, pages 178–190, 2002.
4. Marcos Aurélio Almeida da Silva. CHORD: Constraint Handling Object-oriented Rules with Disjunctions. Master's thesis, Universidade Federal de Pernambuco, February 2009.
5. L. De Koninck, T. Schrijvers, and B. Demoen. Search Strategies in CHR(Prolog). In *CHR 2006: Proc. 3rd Workshop on Constraint Handling Rules*, pages 109–124, 2006.

6. P. Deransart. Generic Traces and Constraints, GenTra4CP Revisited, May 2011. `http://hal.inria.fr/hal-00597033`.
7. P. Deransart. Towards a Trace Meta-Theory, July 2011. Working document `http://hal.inria.fr/inria-00443648` (almost in French).
8. Pierre Deransart, Manuel V. Hermenegildo, and Jan Maluszynski, editors. *Analysis and Visualization Tools for Constraint Programming, Constraint Debugging (DiSCiPl project)*, volume 1870 of *Lecture Notes in Computer Science*. Springer, 2000.
9. Thom Frühwirth. *Constraint Handling Rules*. Cambridge University Press, 2009.
10. Thom Frühwirth and Slim Abdennadher. *Essentials of constraint programming*. Springer-Verlag, 2003.
11. Thom Frühwirth and Pascal Brisset. High-level implementations of Constraint Handling Rules. Technical Report ECRC-95-20, European Computer-Industry Research Centre, Munchen, Germany, 1995.
12. Armando Gonçalves, Pierre Deransart, Marcos Aurelio, and Jacques Robin. Towards a Generic Trace for Rule Based Constraint Reasoning. Technical Report to appear, INRIA Paris-Rocquencourt, September 2011.
13. C. Holzbaur and Thom Frühwirth. Constraint Handling Rules Reference Manual for Sicstus Prolog, July 1998. Technical Report TR-98-01.
14. E. Jahier, M. Ducassé, and O. Ridoux. Specifying Prolog trace models with a continuation semantics. In K.-K. Lau, editor, *Proc. of LOgic-based Program Synthesis and TRansformation*, London, July 2000. Technical Report Report Series, Department of Computer Science, University of Manchester, ISSN 1361-6161. Report number UMCS-00-6-1.
15. Ludovic Langevine, Pierre Deransart, and Mireille Ducassé. A Generic Trace Schema for the Portability of CP(FD) Debugging Tools. In K.R. Apt, F. Fages, F. Rossi, P. Szeredi, and Jozsef Vancza, editors, *Recent Advances in Constraints*, number 3010 in LNAI. Springer Verlag, May 2004.
16. Ludovic Langevine and Mireille Ducassé. Design and implementation of a tracer driver: Easy and efficient dynamic analyses of constraint logic programs. *Theory and Practice of Logic Programming, Cambridge University Press*, 8(5-6), Sep-Nov 2008.
17. Salvador Lucas. Observable Semantics and Dynamic Analysis of Computational Processes. Technical Report LIX/RR/00/02, Laboratoire d'Informatique LIX, 2000.
18. Rafael Oliveira and Pierre Deransart. Towards a Generic Framework to Generate Explanatory Traces of Constraint Solving and Rule-Based Reasoning. Technical Report RR-7165, INRIA Paris-Rocquencourt, December 2009.
19. Helmut Simonis, Paul Davern, Jacob Feldman, Deepak Mehta, Luis Quesada, and Mats Carlsson. A Generic Visualization Platform for CP. In Karen Petrie, editor, *Proceedings of the 16th International Conference on Principles and Practice of Constraint Programming*, St Andrews, Scotland, September 2010.
20. J. Wielemaker. SWI-Prolog 5.6 Reference Manual. *Department of Social Science Informatics, University of Amsterdam, Amsterdam, Marz*, 2006.

Modeling Dependent Events with CHRiSM for Probabilistic Abduction

Henning Christiansen[1] and Amr Hany Saleh[2]

[1] Research group PLIS: Programming, Logic and Intelligent Systems
Department of Communication, Business and Information Technologies
Roskilde University, P.O.Box 260, DK-4000 Roskilde, Denmark
E-mail: henning@ruc.dk
[2] Department of Computer Science and Engineering
Faculty of Media Engineering and Technology
The German University in Cairo, Egypt
E-mail: amr.shehata@student.guc.edu.eg

Abstract. Most earlier approaches to probabilistic abductive logic programming are based on the assumption that abducibles represent independent events. This enables efficient and incremental calculation of probabilities, but may not be suitable for all real world problems. As an attempt to introduce such dependencies in a logic programming setting, we have applied CHRiSM, which is a recent probabilistic extension to Constraint Handling Rules, for specification and evaluation of probability distributions over dependent abducibles. It is shown that this principle integrates well with earlier work on probabilistic abduction based on CHR, generalizing it to handle such dependencies. We present our first experiments with a working implementation that show the potential for interesting applications. On the other hand, our experience underlines problems concerning high complexity due to lacking incremental methods for probability calculation.

1 Introduction

Abduction refers here to the process of reasoning to find explanations (unknown facts), that can explain given observations, and often this definition is refined so we search for a *best* explanation; see, e.g., [8] for more background. Abduction has been studied in the context of logic programming, and adding probabilities provides a way of giving priorities to different explanations, thus providing a meaningful characterization of a "best" solution. With few exceptions, earlier work on probabilistic abduction in logic programming has been based on the assumption that abducible facts are instances of mutually independent random variables. This implies a restricted expressive power but also a straightforward way of calculation probabilities in an incremental way. Earlier work [3] in this direction has shown that the language of Constraint Handling Rules, CHR [9], is well suited for describing implementations with a variety of characteristics, e.g., adapting for best-first search. Here we apply a recent probabilistic extension CHRiSM [21] to extend this work with dependent probabilities, using a

specific form of CHRiSM programs as plug-in modules to characterize such joint probability distributions over abducibles and explanation.

The approach handles negation of abducibles and points forward to a treatment of non-ground and existentially quantified abducibles, which often cause problems in implementations of abductive logic programming. We can also point to other open problems: the calculation of probabilities does not scale well to large sets of abducibles, and useful operations such as subsumption and entailment checking seem to require a detailed knowledge about the actual probability distribution over possible worlds which may not be easily captured. Also, while our semantics in a natural way captures programs with potentially infinitely many abducibles, the way we use CHRiSM excludes this option.

In section 2, we introduce a probabilistic possible worlds semantics as a general theoretical setting for probabilistic logic programming and use it to define abstract syntax and semantics for such programming languages. In section 3, we introduce CHRiSM [21] and describe our approach for using it to define the desired probability distributions. Section 4 explains how probability distributions defined using CHRiSM can be integrated with CHR programs that implement search over the clauses of the given probabilistic logic program, including best-first. Section 5 discusses a few possible enhancements of these strategies. Section 6 discusses selected related work and section 7 gives a conclusion, including suggestions for future work.

2 Basic Concepts

We have chosen a probabilistic possible worlds semantics as a firm theoretical basis, which makes it possible for us to derive – rather than postulate – properties that are reasonable for probability distributions used for the abducibles in a probabilistic abductive logic program. This semantics introduced below is similar to the one used by [16] and adapted to here to our specific case.

2.1 Probabilistic Possible Worlds

We consider here a world (or state of affairs) as given by the truth values of a set of world properties. Intuitively, this set may be so huge that it is impossible for an observer to perceive or name them all in any feasible way. For technical simplicity, we assume the set of world properties to be countable, but larger cardinalities could have been allowed at the price of replacing the summations in the following definition by integrals or other limit constructions. There are no assumptions about probabilistic independency of such properties.

Definition 1. *Assume a set of* world properties $\mathcal{X}$. *A* world *is a mapping from each* $x \in \mathcal{X}$ *to* $\{true, false\}$*; a world* w *may also be viewed as a set of literals, called* world literals*, containing* x *when* x *is true in* w *and* $\neg x$ *otherwise.*

A probability distribution *over worlds is a mapping* P *into* $[0; 1]$ *with*

$$\sum_{w \in W} P(w) = 1.$$

A world w with $P(w) > 0$ is called a possible *world. The set of possible worlds is denoted $\mathcal{W}$. A* sub-world *s is a finite set of world literals with, for no $x \in \mathcal{X}$, both $x \in s$ and $\neg x \in s$; any property x with neither $x \in s$ or $\neg x \in s$ is* undefined *in s. For sub-world s and possible world w, we write $w \models s$ iff $x \in w$ for any $x \in s$ and $x \notin w$ for any $\neg x \in s$; the* world set *for s is defined as $\mathcal{W}(s) = \{w \in \mathcal{W} \mid w \models s\}$.*

The probability of a sub-world s is defined as follows.

$$P(s) = \sum_{w \in \mathcal{W}(s)} P(w)$$

A sub-world s with $P(s) > 0$ is called a possible *sub-world.*

The notion of a sub-world may model what is relevant or perceivable for an observer under the given circumstances, which means that is not interesting or perhaps not possible for the observer to distinguish between different elements of $\mathcal{W}(s)$ for a given sub-world s. Notice that the probability of a sub-world is given by a typically infinite but countable sum which is well-defined due to the initial assumptions.

Whenever a world literal y is the negation $\neg x$ of world property x, we let $\neg y$ refer to x, and the usage of y being undefined in some s means the same as x being undefined in s. It the rest of this section, we assume that a set of world properties $\mathcal{X}$ with a probability distribution P is given. We observe the following properties that follow immediately from the definition.

Proposition 1. *Let s_1, s_2 and s be sub-worlds, x a world literal undefined in s, and let $\emptyset$ refer to the empty sub-world.*

- *Whenever $s_1 \subseteq s_2$, we have that $\mathcal{W}(s_2) \subseteq \mathcal{W}(s_1)$.*
- *Whenever $s_1 \subseteq s_2$, we have that $P(s_1) \geq P(s_2)$.*
- *Whenever $s_1 \subseteq s_2$ and s_1 is not possible, neither is s_2.*
- $P(s) = P(s \cup \{x\}) + P(s \cup \{\neg x\})$
- $P(\emptyset) = 1$.
- *There exists at least one possible world.*

The following property states that a suitable, joint probability distribution over a set of logical literals is compatible with a probabilistic possible world semantics.

Proposition 2. *Let M be a set of ground atoms induced by given sets of predicates and function symbols, and let S be the collection of all sets of non-contradictory literals made from a subset of M. Let, furthermore, P be a function $S \to [0; 1]$ that satisfies the following conditions.*

- *Whenever, for $s_1, s_2 \in S$ with $s_1 \subseteq s_2$, it holds that $P(s_1) \geq P(s_2)$.*
- *Whenever, for $s \in S$ and x being a literal of M with $x, \neg x \notin S$, it holds that $P(s) = P(s \cup \{x\}) + P(s \cup \{\neg x\})$.*
- $P(\emptyset) = 1$.

Then M and P' induces a probabilistic possible world semantics, cf. def. 1, where P' is the restriction of P to 2^M. Furthermore, the generalization of P' to sub-worlds coincides with P.

2.2 Probabilistic Abductive Logic Programs

Here we define the abstract syntax and semantics of probabilistic abductive logic programs, ProbALPs, and characterize some of their important properties. No explicit integrity constraints are included as they can be embedded in the probability distribution by setting zero probability for undesired combinations of abducibles.

Disjoint and countably infinite sets of constants, function symbols, variables and predicates are assumed as usual, together with two disjoint sets of predicates, *abducibles* and *defined* (the latter a.k.a. *program* or *user defined*). Terms and atoms are built in the usual way; a *program literal* is either a defined atom or a perhaps negated abducible atom. Instances and ground instances are defined in the usual way. We assume a *naming function* that maps any ground abducible atom into a unique world property.

Definition 2. *Given a probabilistic possible world semantics, cf. def. 1, and the nomenclature introduced above, a* probabilistic abductive logic program, ProbALP, *is a finite set of* program clauses*, each of the form*

$$h :\!- b_1, \ldots, b_n$$

where h is a defined atom called the head*, $n \geq 0$ and $b_1, \ldots, b_n$, called the* body*, consisting of program literals. A* query *is a conjunction of ground atoms. An* explanation *is a finite existentially quantified set of abducible literals; the quantifier is as usual left implicit in our notation.*

A standard completion semantics [4] is assumed for program clauses. Thus, for a ProbALP *Prog* and explanation E, it is well-defined, whether a given query is true in *Prog* extended with E, denoted $Prog, \exists E \models A$.

A ground explanation is, via the naming function, considered equivalent to a sub-world, and the notion of world sets is generalized accordingly. The world set $\mathcal{W}(E)$ for an explanation E is defined as the union of the world sets for each ground instance of E, and the probability $P(E)$ is defined as $\sum_{w \in \mathcal{W}(E)} P(w)$.

For any two explanations E_1 and E_2 with $\mathcal{W}(E_2) \subseteq \mathcal{W}(E_1)$, we say that E_1 *subsumes* E_2 and that E_1 is more general than E_2. An explanation E is *consistent* whenever $P(E) > 0$ (or, equivalently, that $\mathcal{W}(E) \neq \emptyset$). Whenever $\mathcal{W}(E_1) = \mathcal{W}(E_2)$, we say that E_1 and E_2 are *equivalent*. Whenever E_1 subsumes E_2 and they are not equivalent, we say that the subsumption is *strict*.

The following example indicates that subsumption in the context of dependent abducibles is more complicated than in the independent case.

Example 1. Independently of the underlying set of possible worlds, we have that $\{a\}$ subsumes $\{a, b\}$, and $\{a(X)\}$ subsumes $\{a(17)\}$, both judgments made by purely syntactic arguments. Whether $\{a(X), b(17)\}$ subsumes $\{c(12)\}$ (or the reverse for that matter) depends on the specific semantics, and in general we cannot expect a decision procedure for such judgments.

We observe the following properties about subsumption that follow immediately from the definition.

Proposition 3. *Whenever E_1 subsumes (resp. strictly subsumes, and is equivalent with) E_2, it holds that $P(E_1) \geq P(E_2)$ (resp. $P(E_1) > P(E_2)$, and $P(E_1) = P(E_2)$).*

Whenever E is an explanation and A is an abducible literal, it holds that E is equivalent with $E \cup \{A\}$ iff $P(E) = P(E \cup \{A\})$.

We do not employ these properties in our implementation, but they demonstrate that an effective procedure for evaluation probabilities may give rise to a way of simplifying the presentation of explanations into some minimal form.

Definition 3. *Consider a given ProbALP Prog and a query Q. An* abductive answer for *(or* explanation of*) Q is a consistent explanation E such that for any possible world $w \in \mathcal{W}(E)$ it holds that*

$$Prog, w \models Q.$$

An explanation E of Q is minimal *if not subsumed by another explanation of Q.*

Notice that we used a semantic characterization for minimality rather than a syntactic one based on, e.g., the number of literals or a subset relationship.

Abductive procedures, such as those we describe in this paper, tend to produce minimal proof explanations. In order to avoid a lengthy, technical definition, we introduce this notion informally: a *proof explanation* for a given query Q is an explanation E which is generated through a specific proof of Q (using an SLD proof rule) by incorporating into E those abducible literals that are encountered in the clause instances that are applied.

Example 2. Consider a program with abducible predicates a and b consisting of the two clauses $\{p \texttt{:-} a, b;\ p \texttt{:-} a, \neg b\}$. Then $\{a\}$ is a minimal explanation of p, and $\{a, b\}$ as well as $\{a, \neg b\}$ are minimal proof explanations.

In the case of independent abducibles, proof minimal explanations may be compacted into minimal ones by syntactic rules based on opposite literals as inherent in this example, cf. [3]. However, for dependent abducibles, it requires obviously additional rules that depend on the underlying possible world semantics.

3 Using CHRiSM to Define Possible Worlds Semantics

3.1 An Overview of CHRiSM

CHRiSM [21] is an extension to CHR with probabilistic rules and a probabilistic semantics based on the probabilistic-logic language PRISM [19].

Derivation is seen as a probabilistic process, in the sense that different rules have a certain probability of firing. In this way each derivation is assigned a probability, and the probability of a particular final state is calculated from all possible derivation leading to it. In these calculations, all choices made by CHRiSM are considered independent, but as we see below, a CHRiSM program can define a variety of dependent distributions over sets of constraints.

CHRiSM adds two types of probability statements; it gives the possibility of adding a probability of firing a rule as well as probabilities for selection among the alternatives of a disjunction in the body of the rule. For example:

```
0.7 ?? a ==> c.
a ==> 0.8 ?? b ; c.
```

The first rule indicates a probability of 0.7 to fire in a state that contains the constraint `a`. The second one indicates a probability of 0.8 to add constraint `b` (and not `c`) and probability 0.2 to add constraint `c` (and not `b`) to a state that contains the constraint `a`. CHRiSM provides two sorts of queries for probabilities.

1. `prob( (`Q `<==>` A`) , P)`. Assigns to variable `P` the probability that a derivation starting from an initial state Q ends in a final state whose constraints are exactly those given by A.
2. `prob( (`Q `===>` A`) , P)`. Assigns to `P` the probability that a derivation starting from an initial state Q ends in a final state that contains as a subset those given by A.

In addition, A may contain negated constraints $\sim C$, with the meaning that the constraint C should not appear in the final states considered; this principle is called negation by absence. For more details about CHRiSM, see [21].

3.2 Probability Distributions for ProbALPs using CHRiSM

Our approach to use CHRiSM to define probabilities is to write a CHRiSM program of a specific form that we call a *Probability Defining CHRiSM program*, PDCP. Such a program includes the following constraints,

- a constraint for each abducible predicate with the same arity,
- a special constraint `start/0`,
- a special constraint `failure/0` that must not appear in any rule.

To fit with current limitations in the CHRiSM system, a program should be range-restricted[3] so that no non-ground constraints are created during its execution. Intuitively `start` serves as a "world creator" that defines all possible final states of interest in a probabilistic way, whereas `failure` is never included in such a final state, meaning that $\sim$`failure` via negation as absence characterizes all non-failed, final states. An explanation should be represented as a conjunction of literals without duplicates, and with the negative ones indicated by CHRiSM's negation as absence.

Provided that 1) abducibles always are ground, and 2) the CHRiSM program never produces a failure or loops, we can take the following CHRiSM query as a definition of $P(E)$.

(i) `prob((start ===>` E`), P)`

[3] A program is range-restricted whenever any variable in a rule body appears also in the head of that rule.

I.e., the variable `P` is bound to $P(E)$. We observe that the function P fulfills the conditions of proposition 2, so the probability distribution thus defined is compatible with a possible world semantics. To see this, notice that CHRiSM sums probabilities over exactly those final states that play the role of $\mathcal{W}(E)$.

Example 3. Consider a the following CHRiSM program viewed as PDCP.

```
0.5 ?? start ==> a.
0.1 ?? a ==> b.
```

It indicates a dependency between `a` and `b`: logically it embeds `b` $\rightarrow$ `a` (i.e., if you have `b`, you must also have `a`). It gives, for example, $P(\{\texttt{a}\}) = 0.5$ and $P(\{\texttt{b}\}) = 0.05$, whereas $P(\{\texttt{a},\texttt{b}\}) = 0.05$. From this, we can see the following.

- Abducibles `a` and `b` are indeed dependent as $P(\{\texttt{a},\texttt{b}\}) \neq P(\{\texttt{a}\}) \times P(\{\texttt{b}\})$.
- Proposition 3 confirms that explanations $\{\texttt{a},\texttt{b}\}$ and $\{\texttt{b}\}$ are equivalent.

This can be generalized to PDCPs that apply failure for integrity checking. This involves the problem that some probability mass is lost, i.e., the sum of probabilities for final and non-failed states is < 1. However, when we normalize the probability found by `prob` using the probability of reaching a non-failed final state, the conditions of definition 2 are established. More precisely, the following query will bind variable `P` to the desired probability of explanation E.

(ii) `prob((start ===> ~failure), PN), prob((start ===> E), P1), P is P1/PN.`

The following example illustrates the use of failure for integrity checking as well as more elaborate CHRiSM rules than those of the previous example.

Example 4. We consider a PDCP that may be used for reasoning about the weather in a period of three days covering `yesterday`, `today` and `tomorrow`. The weather can be either `sunny` or `rainy` a particular day, but not both (in other word, the classification describes the prevailing weather that day). The `weather/2` constraint represents abducibles relating weather and days.

```
start ==> 0.6 ?? weather(sunny,yesterday) ; weather(rainy,yesterday).
start ==> 0.6 ?? weather(sunny,today)     ; weather(rainy,today).
start ==> 0.6 ?? weather(sunny,tomorrow)  ; weather(rainy,tomorrow).

0.7 ?? weather(X,yesterday), weather(X,today) ==> weather(X,tomorrow).
weather(X,D), weather(Y,D) ==> X=Y.
```

The first three CHRiSM rules state a particular "background probability" for each weather type, 0.6 for sunny and $1 - 0.6 = 0.4$ for rainy. The fourth rule increases the probability for a given weather if it has lasted for two days already, and the last rule is a CHR rule indicating that only one type of weather is possible on a particular day.

It can bee seen that some derivations beginning from `start` lead to failure. If, for example, the first three rules produce `weather(rain,yesterday)`,

`weather(rain,today)` and `weather(sunny,tomorrow)`, and the fourth rule decides to fire, the derivation fails and we loose the probability mass $0.4 \times 0.4 \times 0.6 \times 0.7$. It can be verified that the following query, which is the part of *(ii)* that calculates the normalization factor,

```
prob((start ===> ~failure), PN)
```

produces the value $1-(0.4\times0.4\times0.6\times0.7+0.6\times0.6\times0.4\times0.7) = 0.832$. Using the probability distribution defined by *(ii)* and the PCDP above, we obtain, for example, $P($`weather(sunny,yesterday)`$) = P($`weather(sunny,today)`$) = 0.4$ and $P($`weather(sunny,tomorrow)`$) \approx 0.6404$. The increase in the overall probability for sun tomorrow is an effect of the the fourth rule that emphases the general majority for sunny weather.

As the factor used for normalization is specific for a given PDCP, we need only calculate it once for that program. The following example shows the code lines needed for an implementation, which also fixes a problem with the currently available implementation of CHRiSM.

Example 5. In the current implementation of CHRiSM, version 0.2 (downloaded August 2011), the `prob` predicate fails when there are no final states for the given query and where it, according to the specification of [21] referred above, should succeed with probability 0.

The following predicate `probNorma` corrects for this problem and incorporates the normalization needed for programs with failures, cf. query *(ii)* above. The purpose of the `init` predicate is to have the normalization factor evaluated only once, prior to the use of `probNorma`.

```
probNorma(Cs,PN):-
    (prob((start ===> Cs),P) -> true ; P=0),
    non_failure_prob(N),
    PN is P/N.

:- dynamic non_failure_prob/1.

init:-
    (prob((start ===> ~failure),P) -> true ; P=0),
    asserta(non_failure_prob(P)).
```

3.3 Experiments using CHRiSM for Non-ground Explanation

The currently available implementation of CHRiSM is only guaranteed to provide correct results for ground queries, and the precise meanings of queries *(i)* and *(ii)* above for non-ground E are not obvious from [21]. We made some promising experiments with the available CHRiSM implementation, which involved small change in one of CHRiSM's internal utilities.

Queries with variables in positive literals do sum the probabilities of the correct set of final states, including when sharing occurs. For example:

```
prob((start ===> a(X),b(X)), N)
```

Here exactly those final states are counted in which there is a pair of abducibles `a(`x`),b(`x`)` for some term x.

Negative, non-ground literals are handled correctly when preceded by a positive literal containing the same variable. For example:

```
prob((start ===> a(X), ∼b(X)), N)
```

Here exactly those final states are counted in which there is an abducibles `a(`x`)` and no `b(`x`)` for the same term x.

For the ground case that we have studied above, we emphasized that duplicate constraints must be avoided in explanations when we give them to the `prob` predicate. This is due to fact that tests for membership in final states made by `prob` is designed according to the multiset semantics of CHRiSM. So for example `prob((start===>a,a),P)` will only count states that include two or more occurrences of `a`. This problem appears also when we have an explanation of the form $E_1 = \{\texttt{a(X)}, \texttt{a(1)}\}$. According to our semantics, this explanation is equivalent with $E_2 = \{\texttt{a(1)}\}$, and to get the correct probability from `prob`, we must use the form E_2 for reasons we just pointed out. However, we should not remove such internally subsumed literals in our interpreters for ProbALPs: it may be the case that an explanation such as E_1 is specialized by `X` being instantiated to, say, `2` so the explanation becomes $\{\texttt{a(2)}, \texttt{a(1)}\}$ which is not subsumed by E_2 which is unaffected by the instantiation of `X`.

4 Integration of Probability Distributions in CHRiSM into Interpreters for ProbALP in CHR

Implementations in CHR of probabilistic abduction with independent probabilities for the abducibles have been described in [3]. Abductive logic programs are compiled into query interpreters in CHR that produce proof minimal explanations. One of the advantages of using CHR is a flexible control which makes it easy to adopt these query interpreters to work best first, calculating the most probable answers in order of decreasing probability, as long as the user asks for more solutions.

We sketch here the methods of [3], adapting them to an example of a simplistic ProbALP; only the calculation of probabilities differs. The referenced paper gives correctness proofs for its special case, which seems straightforward to adapt to the present setting. Here we proceed in an informal way.

We consider an example program with defined predicates `p/1`, `q/1`, ..., and abducibles `a/1`, ... It is assumed that the abducibles and their joint probability distribution is defined in terms of PDCP as described in section 3. The logic program includes among others the following clauses for `p/1`.

```
p(X):- q(X), a(X).
p(1).
```

We start explaining a query interpreter that works depth-first and which requires the ProbALP to be range-restricted, i.e., abducibles and other predicate calls are consistently ground.

The fundamental constraint in the interpreter is the following `explain/3`. An instantiated constraint in the store represents a branch of a computation that proceeds in an adapted SLD manner. The meaning of a call of the form

`explain(`Q`, `E`, `P`)`

is that subquery Q remains in order to have found a proof for the initial query; E is the partial explanation of abducibles encountered so far, and P is the probability of E. Both Q and E are given as lists of literals. When Q has become empty, E is a proof minimal explanation for the initial query. Before giving the details, we define a suitable top-level predicate and the CHR rule that applies when an explanation has been found.

```
explain(Q):-  explain(Q, [], 1).

explain([],E,P) <=>
     print explanation E with its probability P.
```

The program clauses for predicate `p/1` is compiled into the following CHR clause.

```
explain( [p(X)|G], E, P) <=>
    rename([p(X)|G]+E,[p(Xr1)|Gr1]+E1),
    explain([q(Xr1),a(Xr1)|Gr1], E1, P),
    (X=1 -> explain(G, E, P) ; true).
```

This rule applies for a branch with a query having `p(...)` as its "current" subgoal and a continuation referred to by the variable `G`. It rewrites the given branch into new branches, potentially one for each clause in the definition of `p/1`. The renaming predicate `rename(`A`,`A'`)` assigns to A' a copy of A whose variables are replaced in a consistent manner by new variables, and should be called for each such new branch in order to avoid cluttering up alternative instantiations of the variables. Notice, however, that for the last one, we can bypass renaming as the involved variables anyhow are not used for other purposes.

As shown in the example rule above, a clause with a distinct variable in each of its arguments (here: only one) is translated into a straightforward rewriting of the query. The last clause, being that fact `p(1)` implies a unification which may fail, and if that happens, the branch will silently disappear.

Executing an abducible predicate means to add it to the current explanation (if it is not there already) and re-calculate the probability. An abducible predicate is compiled into a CHR rule as follows plus a completely analogous one for negated abducibles.

```
explain([a(X)|G],E,P) <=>
    (member(a(X),E) -> E1 = E ; E1 = [a(X)|E]),
    prob_ex(E1, P1),
    explain(G,E1,P1).
```

The `prob_ex` predicate is an auxiliary predicated that does a little formatting and calls the CHRiSM program to obtain the probability of the new explanation. Notice that it removes duplicate literals before continuing. The `probNorma` predicate is the auxiliary shown in example 5 above which generates a suitable call to CHRiSM's `prob`.

```
prob_ex(E, P):-
    turn the list E into a conjunction EConj using commas,
    probNorma(EConj, P).
```

What has been shown so far is sufficient to turn any range-restricted ProbALP into a depth-first query interpreter. In order to produce a best-first interpreter, we can keep the same basic structure but add a few additional auxiliary constraints to ensure that only the `explain` constraint with the highest probability (in its third argument) can be involved in a rule application; this presents no conceptual difficulties, and we refer to [3] for the details.

Example 6. We consider the PDCP introduced in example 4 and extend it with two constraints `weekday` and `tomorrow`, and the following rule.

```
start ==> 0.7143 ?? workday(tomorrow) ; weekend(tomorrow).
```

It defines a probability of `tomorrow` being either a workday or a day in the weekend which, however, cannot both be the case at the same time. Together with the following clauses it forms a ProbALP.

```
plan_for_tomorrow(work):-  weather(rainy,today), weather(rainy,tomorrow).

plan_for_tomorrow(beach):- weekend(tomorrow), weather(sunny,tomorrow).

plan_for_tomorrow(beach):- weather(sunny,yesterday),
                           weather(sunny,today), weather(sunny,tomorrow).
```

We have compiled this into a best-first query interpreter along the lines above, and here we show some queries and answers. For the following query, one answer is produced.

```
?- explain([plan_for_tomorrow(work)])
Most probably solution:
  [weather(rainy,tomorrow),weather(rainy,today)]
  P=0.192307692307692
Another and less probable explanation? ;
No (more) solutions
```

The next query leads to two different solutions, one for each of the relevant program clauses.

```
?- explain([plan_for_tomorrow(beach)]).
Most probably solution:
  [weather(sunny,tomorrow),weather(sunny,today),weather(sunny,yesterday)]
  P=0.259615384615385
Another and less probable explanation? ;
```

```
Most probably solution:
  [weather(sunny,tomorrow),weekend(tomorrow)]
  P=0.182957884615385
Another and less probable explanation? ;
No (more) solutions
```

It should be noticed that these two solutions overlap in the sense that they have a possible world in common, as their union is consistent. Finally, if we extend the query as follows with an "observed" abducible, we miss out the first of these of these answers as there is no final state containing both `workday(tomorrow)` and `weekend(tomorrow)`.

```
?- explain([plan_for_tomorrow(beach),workday(tomorrow)]).
Most probably solution:
  [workday(tomorrow),weather(sunny,tomorrow),weather(sunny,today),
       weather(sunny,yesterday)]
  P=0.185443269230769
Another and less probable explanation? ;
No (more) solutions
```

This probability is smaller than for the analogous solution for the previous query as the contribution for the world containing `weekend(tomorrow)` plus `weather(sunny, ···)` for all three days.

5 Enhancements for Defining Probabilities using CHRiSM

In this section we show useful extensions to PDCPs that are easy to integrate with the implementation principle shown above. It is shown how a notion of domains may be associated with specific arguments of abducible predicates in order to ensure range-restrictedness. We introduce also an observation module that allows to introduce pre-observed abducibles in the initial constraint store when probabilities are computed.

5.1 Domains that Ensure Ground Abducibles

Consider an example concerning a set of girls $\{Ann, Sue, Eva, \ldots\}$ of size n. A PDCP should be used to describe the probability of each girl being blonde. In principle this may be done using n rule, one for each girl. To avoid this, we might naively suggest the following rule which is not range-restricted, and whose logical meaning does not either express the intended meaning.

```
0.7 ?? start ==> blonde(X).
```

To cope with such unbound variables, we can introduce domain declarations and associate a domain to specific arguments exemplified as follows.

```
domain(girls, [ann, sue, eva, tia]).
type(blonde, 1, girls).
```

The last declaration associates the domain `girls` to the first argument of the abducible predicate `blonde`. With these declarations, we can automatically compile the knowledge of the domains into to relevant PDCP rules, exemplified as follows.

```
0.7 ?? start, constant(girls,X) ==> blonde(X).
```

To get everything to work, the domain declarations should be incorporated into the probability queries performed by the query interpreters as follows.

```
prob((constant(girls,ann),constant(girls,eva),...,start ===> E),P).
```

This saves typing and results in a better program structure, but does not scale well with respect to efficiency for large domains.

5.2 Pre-Observed Facts Module

By pre-observed facts we refer to abducibles that are known to hold before a query is given to some ProbALP. These may be utilized in our implementation by including them in the initial states for probability queries. Assuming, for example, that abducibles `a` and `b` are pre-observed, the probability queries should be done as follows.

```
prob((a, b, start ===> E), P).
```

This may in some cases reduce complexity by removing early in the computation all states that are inconsistent with `a` and `b`. For some PDCPs, the probabilities found in this way and normalized accordingly corresponds to conditional probabilities, but in general we cannot count on this.

6 Comparison with Related Work

Abductive logic programming has been studied since the early 1990ies; see [6, 11] for an overview. Extending with probabilities as a means to prioritize among alternative explanations has been treated by several authors in varying settings, mostly under the assumption of independent probabilities for abducibles. Among the most influential work, we find Poole's which stems also from the early 1990ies [15]. Our own recent work on expressing probabilistic abduction in CHR (with independent probabilities) extends previous work with integrity constraints and the option of integrating with external constraint solvers; see [3] which also gives a detailed overview of related work concerned with abduction using CHR. PRISM [19] is a highly generic probabilistic-logic language which can work with an unlimited number of independent random variables and as a special case it can also express abduction (with independent abducibles). There are other recent works on abduction in different logic-based frameworks that are not directly comparable to ours, and which allow probabilistic dependencies [17, 2].

The notion of Possible Worlds has been used in philosophy and linguistics in attempts to cover pragmatic aspects of meaning, i.e., the relationship between representation and the real world or collections of hypothetical worlds (or states of affairs); see, e.g., the work by Stalnaker as summarized in [22]. It is used by [13] to explain the semantics of counterfactual statements and is standard for modal logics [12]. Probabilistic possible worlds have been proposed by several authors. It is difficult to trace a single origin, but we may mention some influential papers, e.g., [14, 1, 7]. The formulation we have used is similar to [16] that provides a concise introduction and overview; see also [10]. Although our terminology is different, there is also a strong resemblance to Sato's distribution semantics [18] and an earlier work by Dantsin [5].

Several of the referenced and other similar works state to have a possible worlds semantics but quite often this is not formalized or described in detail. In the present paper, we used possible worlds as the defining framework to characterize desired properties for the probability distributions, and our distributions using CHRiSM was shown to conform with those.

Recent work by Simari and Subrahmanian [20] describes an approach to a special form of abduction in a limited logic language with intervals of probabilities associated to clauses. It is based on a sort of possible worlds, each being a collection of derived properties (which is unconventional), and it stresses also the absence of independency assumptions. However, the setting is quite different from ours and most other definitions of Abductive Logic Programming, and a direct comparison is difficult.

We are not aware of other work than our own that uses a probabilistic possible world semantics to define dependent distributions over abducible predicates, which are applied for generation of probability weighted explanations for given observations. In addition, we propose a specific method for defining such probability distributions.

7 Conclusion

In this work, we have defined a semantics of probabilistic abductive logic programs that supports dependent probabilities for abducibles. We introduced a special kind of CHRiSM programs called *probability defining CHRiSM programs*, which is consistent with this semantics. There is an appealing similarity between the way a possible world semantics sums up probabilities over possible worlds and the way CHRiSM sums probabilities over final states. However, this way of using CHRiSM is not very efficient and blocks effectively for using an unlimited number of abducibles.

Our theoretical model can handle both negated and non-ground abducibles, but the current version of CHRiSM cannot handle our encoding of non-ground abducibles, except in a few simple cases. We may anticipate that future releases of CHRiSM provide a clarified semantics in these respects.

We have also shown how such probability defining CHRiSM programs can be incorporated into query interpreters written in CHR. Moreover, we consid-

ered facilitating the definition of probability distributions using CHRiSM with the introduction of domains for abducible predicates and suggested a way to incorporate pre-observed abducible facts.

7.1 Time Complexity Problems

Our implementation of probability calculations uses CHRiSM in a way that forces the enumeration of all possible final states each time a probability is asked for. This introduces a factor into the overall time complexity which is exponential in the number of abducibles.

There may be several ways to approach this problem. In principle we could generate all possible worlds (i.e, final CHRiSM states) once and for all before the program execution starts, and represent them as an array of bit vectors and provide a fast matching to determine in which worlds a given (partial) explanation is actually true; this may be a viable solution in specific cases which even may be supported by specialized hardware. However, the development of incremental methods may be more suited as explanations gradually get specialized by adding more literals or unifying variables; this is how it is typically done for independent probabilities, but for the dependent case, there is no obvious universal solution. The notion of unambiguous CHRiSM programs suggested by [21] may be a useful restriction in the search for incremental methods.

7.2 Operations on Explanations

Subsumption and entailment of explanations as well as compaction of different explanations into more general (and thus more probable) ones can be computed efficiently for independent abducibles, as e.g., studied by [3]. However, our use of CHRiSM as a sort of blackbox makes this difficult, since these operations depend on properties embedded in the possible worlds semantics. Clearly more research needs to be done here.

Acknowledgements. This work is supported by the project "Logic-statistic modelling and analysis of biological sequence data" funded by the NABIIT program under the Danish Strategic Research Council.

References

1. John C. Bigelow. Possible worlds foundations for probability. *Journal of Philosophical Logic*, 5(3):299–320, 1976.
2. Jianzhong Chen, Stephen Muggleton, and José Carlos Almeida Santos. Learning probabilistic logic models from probabilistic examples. *Machine Learning*, 73(1):55–85, 2008.
3. Henning Christiansen. Implementing probabilistic abductive logic programming with Constraint Handling Rules. In Tom Schrijvers and Thom W. Frühwirth, editors, *Constraint Handling Rules*, volume 5388 of *Lecture Notes in Computer Science*, pages 85–118. Springer, 2008.

4. Keith L. Clark. Negation as failure. In Hervé Gallaire and Jack Minker, editors, *Logic and Data Bases*, pages 293–322. Perseus Publishing, 1978.
5. Evgeny Dantsin. Probabilistic logic programs and their semantics. In Andrei Voronkov, editor, *RCLP*, volume 592 of *Lecture Notes in Computer Science*, pages 152–164. Springer, 1991.
6. Marc Denecker and Antonis C. Kakas. Abduction in logic programming. In Antonis C. Kakas and Fariba Sadri, editors, *Computational Logic: Logic Programming and Beyond*, volume 2407 of *Lecture Notes in Computer Science*, pages 402–436. Springer, 2002.
7. Ronald Fagin and Joseph Y. Halpern. Uncertainty, belief, and probability. *Computational Intelligence*, 7:160–173, 1991.
8. Peter A. Flach and Antonis C. Kakas, editors. *Abduction and Induction: Essays on their relation and integration.* Kluwer Academic Publishers, April 2000.
9. Thom Frühwirth. *Constraint Handling Rules.* Cambridge University Press, August 2009.
10. Joseph Y. Halpern. A logical approach to reasoning about uncertainty: a tutorial. In Xabier Arrazola, Kepa Korta, and Francis Jeffry Pelletier, editors, *Discourse, Interaction, and Communication*, pages 141–155. Kluwer, 1998.
11. A.C. Kakas, R.A. Kowalski, and F. Toni. The role of abduction in logic programming. *Handbook of Logic in Artificial Intelligence and Logic Programming,* vol. 5, Gabbay, D.M, Hogger, C.J., Robinson, J.A., (eds.), Oxford University Press, pages 235–324, 1998.
12. Saul Kripke. Semantical considerations on modal logics. *Acta Philosophica Fennica*, 16:83–94, 1963.
13. David Lewis. *Counterfactuals.* Blackwells, 1973.
14. Nils J. Nilsson. Probabilistic logic. *Artificial Intelligence*, 28(1):71–87, 1986.
15. David Poole. Logic programming, abduction and probability - a top-down anytime algorithm for estimating prior and posterior probabilities. *New Generation Computing*, 11(3):377–400, 1993.
16. David Poole, Alan Mackworth, and Randy Goebel. *Computational intelligence: a logical approach.* Oxford University Press, 1998.
17. Sindhu Raghavan and Raymond Mooney. Bayesian abductive logic programs. In *Proceedings of the AAAI-10 Workshop on Statistical Relational AI (Star-AI 10)*, pages 82–87, Atlanta, GA, July 2010.
18. Taisuke Sato. A statistical learning method for logic programs with distribution semantics. In Leon Sterling, editor, *Logic Programming, Proceedings of the Twelfth International Conference on Logic Programming*, pages 715–729. Leon Sterling, 1995.
19. Taisuke Sato. A glimpse of symbolic-statistical modeling by PRISM. *J. Intell. Inf. Syst.*, 31(2):161–176, 2008.
20. Gerardo I. Simari and V. S. Subrahmanian. Abductive inference in probabilistic logic programs. In Manuel V. Hermenegildo and Torsten Schaub, editors, *ICLP (Technical Communications)*, volume 7 of *LIPIcs*, pages 192–201. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2010.
21. Jon Sneyers, Wannes Meert, Joost Vennekens, Yoshitaka Kameya, and Taisuke Sato. CHR(PRISM)-based probabilistic logic learning. *Theory and Practice of Logic Programming*, 10(4-6):433–447, 2010.
22. Robert Stalnaker. On the representation of context. *Journal of Logic, Language and Information*, 7(1):3–19, 1998.

Model Transformation using Constraint Handling Rules as a basis for Model Interpretation

Marcel Dausend and Frank Raiser

Institute of Software Engineering and Compiler Construction
Ulm University, Ulm, Germany

Abstract. In this paper, we present a model transformation approach aiming to simplify automatic processing of UML state machine models, especially for interpretation. The main requirements are easing the implementation of the interpreter and reducing the number of calculations necessary to execute a model.
Our model transformation preserves the semantics and is implemented using CHR. The result of the transformation is an UML state machine model based on the concept of compound transitions. Furthermore we provide an interpreter for those models which supports a comprehensive subset of UML state machine concepts, i. a. junction, fork, join.
Our preliminary results show that state machine interpreters can profit from the former model transformation. It simplifies certain aspects of the interpreter implementation and positively affects the performance of the interpreter, e.g. regarding transition selection and transition execution.

Keywords: Constraint Handling Rules (CHR), Unified Modeling Language (UML), model transformation, model interpretati!on

1 Introduction

The Unified Modeling Language (UML)[1] is the de facto standard in Software Engineering for modeling software systems. An important and popular application of UML models is automatic processing like static evaluation, or interpretation, and also code generation. Interpretation of UML state machines poses a hard problem for several reasons: regarding numerous static as well as dynamic aspects, handling complex models, and dealing with non formal UML semantics, etc.

Optimization of UML state machine interpreters, programs that execute a given model by interpretation, is a challenging topic, especially enhancing the performance without violating UML semantics or excluding certain concepts. We think, there are at least two effective strategies for optimization: enhancing the implementation, and/or simplifying the model before interpretation.

In this paper, we illustrate how the interpretation can profit from model simplification by former model transformation (cf. Sec. 4). Our strategy of model

transformation is to consolidate information spread over the model advantageously at specific model elements. The basis for this transformation is the UML concept of compound transitions, i.e. a whole transition path between sets of states contained in a state machine model. We implemented a model transformation using Constraint Handling Rules (CHR) to make compound transitions explicit by folding multiple transitions. Thereby, information needed for interpretation is no longer spread over the model. In consequence, the interpretation (cf. Sec. 5) can profit by a simplified transition selection algorithm as well as transition execution.

We implemented the model transformation using CHR, a general-purpose programming language, formally defined as a state transition system that is both rule- and logic-based (cf. Sec. 3). CHR displays the same tendency as many rule- or logic-based approaches in that there already exist hundreds of scientific publications on it, yet we are only just beginning to witness its adoption by industry users.

2 State Machines

In this section, we introduce the basic concepts of UML state machines. For further reading on this subject (cf. [1–3]).

State machines are a language package of UML used to model behavior. Each behavior model is based on a context provided by a classifier that defines properties, operations, visibility, etc. A state machine can define its own context or can be executed in an existing context if it is invoked by a different behavior within that context.

The static structure of UML is defined by the UML meta-model given by means of class diagrams and a textual specification for each element following the structure: Generalizations, Description, Attributes, Associations, Constraints, Semantics, and Notation. Figure 1 gives an overview of a part of the UML state machine meta-model. All these elements and their related constraints and semantics (cf. [1, Ch. 15]) are relevant for both our model transformation and execution of the resulting models by interpretation.

The two main concepts of the meta-model (cf. Fig. 1) are classes to define elements, and associations to define the relations between classes. The class StateMachine is associated with the class Region by a composition. This means that a state machine must have at least one region that only exists as long as its state machine. Each region contains some vertexes and transitions between them. The UML distinguishes between different kinds of vertexes using inheritance. A Vertex of type State can be simple, composite, or orthogonal. A so-called Pseudostate is used as intermediate targets of a transition path. Each kind of pseudo state (cf. PseudostateKind in Fig. 1) has its own semantics taking into account individual aspects of the dynamics of state changes. Behavioral aspects are realized by associating Behavior, Trigger, and Constraint elements with a State and/or Transitions.

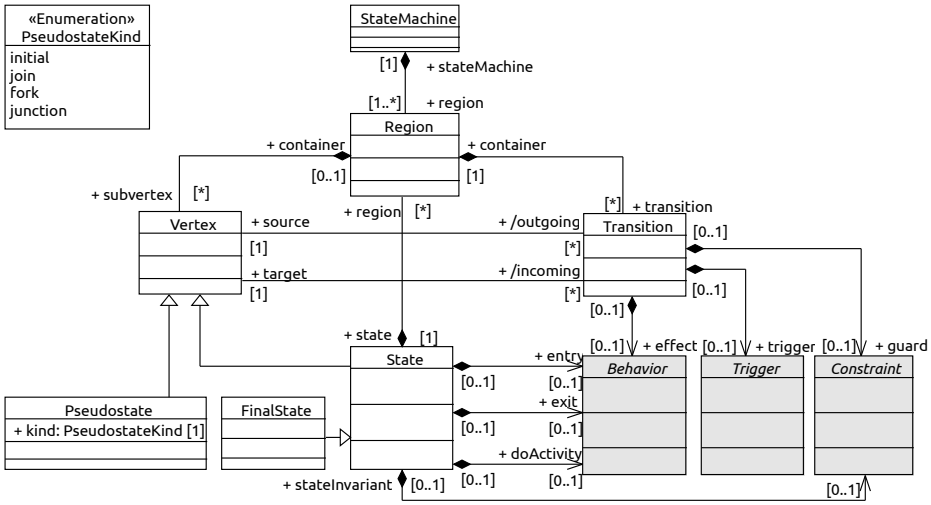

Fig. 1. Excerpt of the UML state machine meta-model showing all model elements for which we support model transformation and interpretation. (Behavior, Trigger and Constraint are classes from different UML meta-model packages.)

The dynamic aspects of state machines are given by a textual description [1]. The configuration of a state machine is represented by a set of active states called active state configuration.

"Junction vertices are semantic-free vertices that are used to chain together multiple transitions" [1, P. 557]. They allow branching of transitions into several paths where a path is selected at runtime based on the active state configuration.

Figure 2 shows a state machine containing one region *rtop* with an initial, a fork *fork1*, and a join pseudo state *join1*, and two states *a* and *b*. The transitions *t3* up to *t7* are connected to either *fork1* or *join1* pseudo state and form a compound transition. A compound transition leads from a set of states to another set of states. The states *a* and *b* are orthogonal states, which means in terms of UML that these states have at least two regions. State *a* has two regions: *r1* containing state *a1*, and region *r2* with an initial state and a sub-state *a2*. State *b* owns three regions *r3*, *r4*, and *r5*. Each region contains a state which is the target of one transition *t6*, *t7*, or *t8*. Region *r5* contains an initial pseudo state. which is the source of *t8*.

3 Constraint Handling Rules

Constraint Handling Rules has been introduced in the early 90ies [4]. The article [4] summarized its early development and was the main reference for the following decade, but was recently replaced by a new book [5].

CHR distinguishes user-defined or CHR constraints from built-in constraints. The latter are provided by a host system, e.g. , Prolog or Java. A CHR program

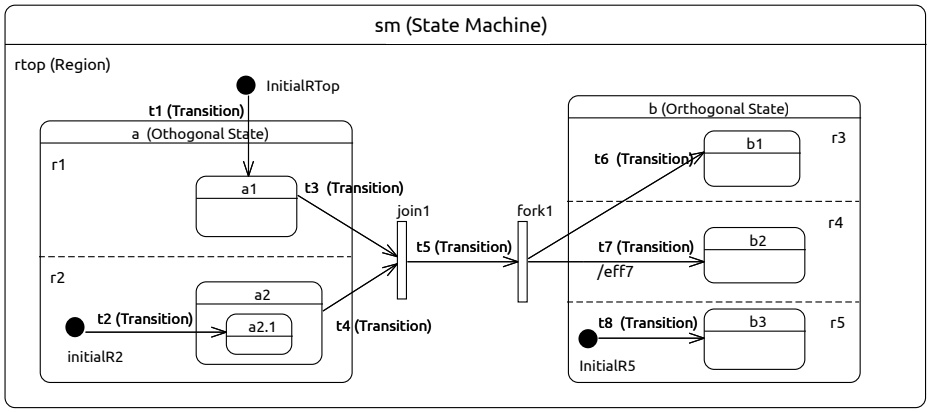

Fig. 2. A UML state machine containing two orthogonal states a and b.

consists of a finite set of rules, for which the remainder of this section discusses their syntax and semantics.

Definition 1 (CHR Rules [6]).

A CHR program $\mathcal{P}$ *is a finite set of* rules *of the form* $H_1 \backslash H_2 \Leftrightarrow G \mid B$, *where* H_1 *and* H_2 *– called the* head *– are multisets of CHR constraints,* G *– called the* guard *– is a conjunction of built-ins, and* $B = B_c, B_b$ *– called the* body *– consists of a multiset* B_c *of CHR constraints and a conjunction* B_b *of built-ins.* H_1 *is referred to as the* kept head *and* H_2 *as the* removed head.

There are different specializations of CHR rules, depending on the head:

- Simpagation rules *(*$H_1 \neq \emptyset \wedge H_2 \neq \emptyset$*) written as* $H_1 \backslash H_2 \Leftrightarrow G \mid B$
- Simplification rules *(*$H_1 = \emptyset \wedge H_2 \neq \emptyset$*) written as* $H_2 \Leftrightarrow G \mid B$
- Propagation rules *(*$H_1 \neq \emptyset \wedge H_2 = \emptyset$*) written as* $H_1 \Longrightarrow G \mid B$

Every CHR rule can optionally be preceded by an identifier *(or* name*) followed by* @*. Finally, if the guard* G *is* $\top$ *it may be omitted together with the* $\mid$ *character.*

Variables that occur in the rule, but not in its head, are called local variables.

There exist multiple operational semantics of CHR rules in the literature [7]. A concise formal definition for the equivalence-based operational semantics [8] is given below. It is suitable for our semantics-related discussions in this work, while our implementation relies on the refined semantics [9] instead. The refined semantics removes multiple sources of non-determinism from the equivalence-based operational semantics, and hence, forms the basis of many available CHR implementations.

Definition 2 (Operational Semantics ω_e).

For a CHR program $\mathcal{P}$*, the state transition system* $(\Sigma_e/\equiv_e, \rightarrowtail_e)$ *is given in Table 1. The transition is based on a variant of a rule* r *in* $\mathcal{P}$ *such that its local*

variables are disjoint from the variables occurring in the representation of the pre-transition state.

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

Table 1. State Transition System ω_e with Equivalence Classes

In contrast to other definitions, a CHR state in ω_e is an equivalence class that contains all possible syntactic formulations. This results in the concise definition of the state transition system given in Table 1. As can be seen from this inference rule, a rule application removes the H_2 part of a head and adds the body to the current state if the guard G is satisfied.

In this work, we interpret UML diagrams as typed graphs. Earlier work [6] discussed how graphs, more precisely graph transformation systems, can be embedded in CHR. The mapping itself is fairly intuitive and encodes nodes and edges as CHR constraints. Additionally, to achieve proper semantics for the graph transformation systems, the degree of nodes is encoded explicitly in the corresponding CHR constraints and consistently updated by rules. Due to the intuitive nature of the encoding, we omit a detailed discussion here, and instead refer the interested reader to [6].

4 Transformation into Compound State Machine Models

In this section we introduce the supported concepts (cf. Sec. 4.1), explain and demonstrate our model transformation by means of our example (cf. Sec.4.2), and characterize the resulting models (cf. Sec. 4.3).

> In general, a compound transition is an acyclical unbroken chain of transitions joined via join, junction, choice, or fork pseudostates that define [a] path from a set of source states (possibly a singleton) to a set of destination states, (possibly a singleton). [1, P. 589]

Our model transformation basically creates a single compound transition for every transition path of a state machine model – we call the resulting model a *compound state machine* model. Such a model offers considerable advantages for further processing of state machines – in our case interpretation.

4.1 Supported UML Concepts

Our transformation concept preserves all information of the source model such that the resulting model is capable to substitute the original model. In the following, we refer to the most relevant UML concepts taken into account for our model transformation (cf. Fig. 1).

A simple compound transition comprises a source and a target State connected via a path of two Transitions with an intermediate Junction within the same region. To support this kind of compound transitions exit- and entry-Behavior of States, Trigger, Guard, and Effect of a Transition, and Junction are considered for model transformation.

Composite States contain one Region, e.g. with other States. Transitions from (or to) such an inner State (cf. state *a2.1* in Fig. 2) cross the border of at least one composite state so that their exit- and entry-Behavior are respected as well.

Orthogonal States extend the concept of compound transitions by allowing parallel Regions within a State (cf. states *a* and *b* in Fig. 2). Join and Fork enable modeling of explicit transitions between sets of states. The source or target of a compound transition is a set of States (cf. Fig. 2).

Further UML concepts are realized by our interpreter based on models resulting from our model transformation. Apart from explicit transitions, implicit ones, e.g. a default transition of a region (cf. Fig. 2), are handled by our interpreter. Furthermore, concepts like ConnectionPoints and Histories are not affected by our model transformation and therefore can be considered as well. Decisions are not taken into account because their semantics influence the transition selection by allowing side effects on guards of outgoing transitions during traversing.

4.2 Model Transformation with CHR

We illustrate the transformation by means of the example shown in Fig. 2. The transformation consists of several steps: preprocessing of the UML source model and its conversion into CHR code, model transformations called "transition folding" and "transition lifting", and post-processing. The main steps of the general model transformation are folding transition paths, lifting of transitions, and generating code.

The UML source model is assumed to be valid according to the UML specification [1]. The preprocessing ensures that suitable termination conditions can be found for recursive transformation rules. Therefore, every incoming transition of Junctions, if more than one exists, is assigned to its own new Junction. The outgoing transitions of the new Junctions are copies of those of the original Junction.

The resulting UML model is considered as CHR input as basis for model transformation. According to the Fork and its related elements in Fig. 2 a formal description, given as 'XML Metadata Interchange' file (cf. Lst. 1.1), is converted into equivalent CHR code (cf. Lst. 1.2).

Preprocessing rules We defined nine CHR constraints according to the meta-model of UML state machines [1]. These CHR constraints reflect the UML elements of state machines and the relation between them, e.g. a state as a source or target of a transition (cf. Lsts. 1.1, 1.2). Nesting relations between elements are made explicit introducing an `owner` constraint, e.g. `owner(`*`fork1`*`,` *`rtop`*`)` means that *fork1* is owned by region *rtop*. The ownership of Transitions is

not determined by the UML specification. Therefore, we consider a TRANSITION to be owned by the least common ancestor REGION of its source and target STATES. For example transition *t2* is owned by region *r2* whereas *t4* is owned by *rtop* (cf. Fig. 2). As last preprocessing step, all else guards of transitions are made explicit by setting them to the conjunction of all negated guards of the transitions originating from the same source vertex. This step eases further processing of guards during transformation as well as interpretation.

```
...
<region xmi:id="4" name="rtop">
...
  <subvertex xmi:type="uml:Pseudostate" xmi:id="16"
      name="Fork" kind="fork1"/>
  <subvertex xmi:type="uml:State" xmi:id="17" name="b">
    <region xmi:id="22" name="r3">
      <subvertex xmi:type="uml:State" xmi:id="23" name="b1"/>
    </region>
    <region xmi:id="24" name="r4">
      <subvertex xmi:type="uml:State" xmi:id="25" name="b2"/>
    </region> ...
  </subvertex>
  <transition xmi:id="27" name="t5" source="10" target="16"/>
  <transition xmi:id="28" name="t6" source="16" target="23"/>
  <transition xmi:id="29" name="t7" source="16" target="25"
      effect="eff7"/>
</region>
...
```

Listing 1.1. XMI Definition for the relevant model elements of our example.

```
...
region(rtop, 0, 11), owner(rtop, sm),
vertex(b1, 2), owner(b1, r3), vertex(b2, 2), owner(b2, r4),
vertex(fork1, 4), pstate(fork1, fork), owner(fork1, rtop),
...
trans(t5, [join1], [fork1], [join1], [fork1], [], [], []),
    src(t5, join1), tgt(t5, fork1), owner(t5, rtop),
trans(t6, [fork1], [b1], [fork1], [b1], [], [], []),
    src(t6, fork1), tgt(t6, b1), owner(t6, rtop),
trans(t7, [fork1], [b2], [fork1], [b2], [], [], [eff7]),
    src(t7, fork1), tgt(t7, b2), owner(t7, rtop)
...
```

Listing 1.2. Part of the CHR input model relevant for folding the transitions connected to the fork.

Transition folding. Folding means merging two consecutive TRANSITIONS of a transition path which are coupled via either a FORK, JOIN or JUNCTION. For each of these pseudo states two CHR rules for folding are defined. One for the base case and one for the recursion case. The interesting transformation part in

our example (cf. Fig. 2) deals with the transition path between the states *a1*, *a2*, *b1*, and *b2* involving PSEUDOSTATES *join1* and *fork1* as well as TRANSITIONS *t3* to *t7*. All these TRANSITIONS are recursively transformed into a single semantics preserving compound transition.

Regarding FORKS, the aim of each recursive transformation step is to remove one outgoing transition and shift its information to a different outgoing transition. If only one outgoing transition remains, the base case removes the fork and both transitions and adds its information to a new transition. In Fig. 2 three transitions (*t5*-*t7*) are connected to the fork *fork1*.

The CHR constraints (before transformation) are shown in Lst. 1.2. Each CHR constraint has arguments to reflect the information of the UML model and additional information. The constraint `region` has three arguments: a name, depth, and degree. The `argument` degree is a counter for related elements, e.g. vertexes and transitions. Region *rtop* is the topmost region of the state machine and has a depth of zero by definition. At the beginning of the transformation, region *rtop* is related to 11 elements, i.e. 11 other constraints make use of rtop. The constraint `vertex` has two arguments, name and degree. The `pstate` constraint indicates the kind of pseudo state, i.e. *fork1* is of type FORK. `Transition` has eight arguments: the name of the transition, a list of states to deactivate, a list of states to activate, a list of source states, a list of target states, a list of triggers, a list of guards, and a list of effects. Thus, it holds all information of a compound transition.

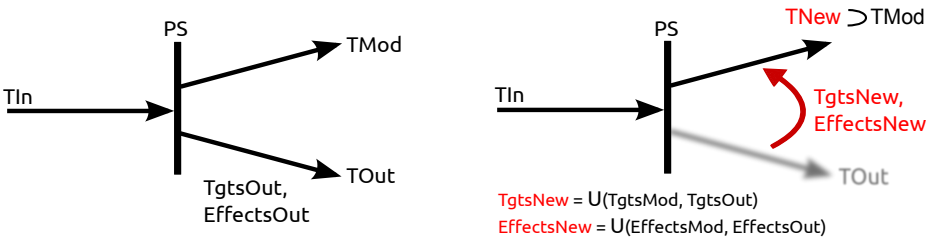

Fig. 3. Folding transformation applied on transitions connected to a fork pseudo state with at least two outgoing transitions (recursion case).

The head of the CHR rule in Lst. 1.3 describes the situation before the folding transformation. This situation is present in Lst. 1.2 and is shown schematically in Fig. 3 (left). The schematic arguments are mapped to the example as follows: *TIn* to *t5*, *PS* to *fork1*, *TMod* to *t6*, and *TOut* to *t7*. *TOut* targets vertex *V2* which is mapped to *b2*. All transitions are owned by region *rtop*. Because it is not determined by the rule which transition has to be chosen for removal, *TOut* (*t7*) is selected non-deterministically for removal.

The application of the rule removes the constraints for two outgoing transitions (*TMod* and *TOut*) of a pseudo state *PS*. These transitions are removed by applying the rule whereas a new one (*TNew*) is constructed to substitute *TMod*.

```
tgt(TIn, PS), pstate(PS, fork),
src(TMod, PS), tgt(TMod, _),
\
trans(TOut, _, _, _, TgtsOut, _, _, EffectsOut), src(TOut,
    PS), tgt(TOut, V2),
trans(TMod, _, ActsMod, SrcsMod, TgtsMod, TriggersMod,
    GuardsMod, EffectsMod),
vertex(V2, DV2),
vertex(PS, DPS)
<=>
append(EffectsOut, EffectsMod, EffectsNew),
append(TgtsMod, TgtsOut, TgtsNew),
trans(TNew, _, ActsMod, SrcsMod, TgtsNew, TriggersMod,
    GuardsMod, EffectsNew),
make_lca_owner(TIn, TMod, TNew), % compute and assign owner of TNew
remove_owner(TMod), remove_owner(TOut), % decrease degree of owners
TNew = TMod,
DV21 is DV2-1, vertex(V2, DV21),   % decrease degree of V2 and PS
DPS1 is DPS-1, vertex(PS, DPS1).
```

Listing 1.3. CHR rule of the recursive case to fold a Fork.

In order to preserve all information of both removed transitions, all arguments of *TMod* are assigned to *TNew*. Since UML only allows Effects on outgoing transitions, the effects of *TOut* and *TMod* are merged to a new list of effects for *TNew*. The lists of target vertexes are treated similarly. Finally, the degrees of the affected vertexes and regions (which own the transitions) are adjusted. As a consequence of removing *TOut* (*t7*), the degrees of *PS* (*fork1*), *V2* (*b2*), and the owning regions (*rtop*) have changed. Therefore, the `vertex` constraints are in the removed head and are recreated in the body with a degree which is decreased by one. The degree of the topmost region is decreased by two and increased by one by the `remove_owner` and `make_lca_owner` constraints (cf. Lst. 1.3). The resulting set of constraints is given in Lst. 1.4.

```
...
region(rtop, 0, 10), owner(rtop, sm),
vertex(b1, 2), owner(b1, r3), vertex(b2, 1), owner(b2, r4),
vertex(fork1, 3), pstate(fork1, fork), owner(fork1, rtop),
...
trans(t5, [join1], [fork1], [join1], [fork1], [], [], []),
    src(t5, join1), tgt(t5, fork1), owner(t5, rtop),
trans(t6, [fork1], [b1, b2], [fork1], [b1, b2], [], [],
    [eff7]), src(t6, fork1), tgt(t6, b1), owner(t6, rtop),
...
```

Listing 1.4. The CHR constraints after a transformation where Lst. 1.3 is applied on Lst. 1.2 (t6 $\sim$ *TMod* and *t7* $\sim$ *TOut*)

For the base case of the above transformation, the fork *PS* has exactly one incoming and one outgoing transition. Then, both transitions are removed and

a new transition is inserted as substitution. Unlike the recursive case the list of states to be deactivated, triggers, and guards from the incoming transition are taken into account.

Transition lifting. Another important concept of our model transformation is lifting of source and target transition ends. The lifting process is the last step before a transition is transformed into code. If a transition execution is performed leading to a state outside the region of its source state, all states whose borders are crossed have to be either deactivated or activated. The lifting transformation collects this information and stores it in the corresponding arguments of the constraint `trans`. A transition that cannot be lifted any more is transformed into code.

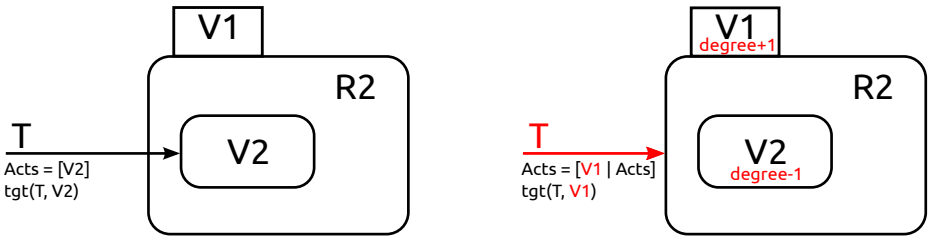

Fig. 4. Transformation to lift a target end of a transition to its surrounding state. This transformation collects the information about which states have to be activated during the execution and writes it to the modified transition.

```
region(R2, _, _), owner(R2, V1), owner(V2, R2)
\
vertex(V1, DV1), vertex(V2, DV2),
trans(T, Deacts, Acts, Srcs, Tgts, Trigger, Guard, Effect),
tgt(T, V2)
<=>
tgt(T, V1), % redefine target for transition T
trans(T, Deacts, [V1|Acts], Srcs, Tgts, Trigger, Guard,
   Effect), % append V1 to list of Acts
DV11 is DV1+1, vertex(V1, DV11), % increase degree of V1
DV21 is DV2-1, vertex(V2, DV21). % decrease degree of V2
```

Listing 1.5. The CHR rule lifts the target of a transition T from a vertex V1 to a vertex V2 where V2 owns the region R2 which contains V1.

Figure 4 illustrates the lifting of a transition's target end (left to right). If a vertex *V2* is a target of a transition *T*, and it is owned by a region *R2* then the transition end is "lifted" to the region's owner vertex *V1*. This condition is expressed by the kept head of the CHR rule in Lst. 1.5. The degree of those vertexes, the list of to be activated vertexes (*Acts*), and the target for the transition *T* are changed by the rule. Lifting results in increasing the degree of *V1* by one and in decreasing the degree of *V2* by one. In the case of execution of *T*, *V1* has to be activated before entering *V2*. Therefore, *V1* is appended to list *Acts* of *T*. Finally, the constraint `tgt` for *T* is changed from `tgt(T, V2)` to `tgt(T, V1)`.

Lifting terminates, if no parent vertex for *V2* exists. A analogous lifting is performed on source transition ends to construct the list *Deacts* of states to be deactivated.

Post-processing As a consequence, both lists – *Deacts* and *Acts* – contain an identical prefix of undesired vertexes. These vertexes are common ancestor states of the source and target of the transition, i.e. the transition connects vertexes inside a common parent vertex. Post-processing rules are added to eliminate these prefixes from the lists *Deacts* and *Acts* of a `transition`. As mentioned above, further rules deal with other aspects of the compound transition UML model (cf. Sec. 4.1), e.g. transition paths involving JUNCTIONS.

At last, all sub-models which are atomic in terms of our model transformation are translated into code constraints reflecting the final UML model. This model is the basis for our interpreter.

4.3 Model Transformation Summary

The resulting model according to Sec. 4.2 can be characterized as follows: The model consists of only STATES, REGIONS, TRANSITIONS, and elements which are unaffected by the transformation. Except for TRANSITIONS originating from INITIAL PSEUDO STATES, every TRANSITION directly connects two sets of STATES. Each TRANSITION holds all information necessary for its execution. This information is no longer distributed over the model. All "else"-GUARDS are now explicit, such that they can be analyzed in a context-free manner. Paths, that constitute a non regular compound transition (i.e. unsatisfiable GUARDS, or multiple TRIGGERS) are removed from the model.

5 UML Interpreter for Compound State Machines

The interpretation of UML models in general is a challenging topic. The complexity of UML itself and its imprecise semantics are very hard problems for automatic processing of UML models. Unclarities and ambiguities according to the semantics are resolved by following the semantics defined by [10, 11]. Currently we do not know any interpreter for models of UML state machines based on the concept of compound transitions. For that reason we implemented a prototypical interpreter as proof of concept.

In the following we outline the technical basis and concepts of the interpreter. Thereby, we focus on similarities and differences to traditional implementation concepts [12] and exemplarily refer to the state machine semantics of [10].

The interpreter is implemented in the Scala programing language[1]. Additionally, the actors package is used to support multi threading and ease communication between objects, e.g. states and regions. The interpreter comprises the Scala classes Application, StateMachine, State, Region, Transition, and Event. The class Application implements a given UML model according to our transformation result. Each other class realizes the behavior of the UML concepts their

[1] http://www.scala-lang.org/

name refer to. All these classes are implemented as agents to support parallelism and simplify communication.

First, we enumerate some concepts of the interpreter which are similar to traditional UML interpreter implementations. The interpreter is implemented in an object-oriented manner where behavioral objects correspond to UML objects. Some superordinate activities of the UML model execution are done by the state machine, e.g. event selection and event dispatching.

From our point of view, the main differences compare to other interpreters can be found in transition selection and transition execution. The UML requires so-called run-to-completion processing [1, p. 580]. Therefore, two points have to be guaranteed:

1. "an event occurrence can only be taken from the pool and dispatched if the processing of the previous current occurrence is fully completed.", i.e. a compound transition may only have one trigger.
2. "an event occurrence will never be processed while the state machine is in some intermediate and inconsistent situation."

As consequence, one has to assure that if a state configuration is left another stable state configuration will be reached. To guarantee this, it is necessary to check if all guards of a transition path will evaluate to true and that no more than one trigger is contained on the whole path (cf. [10]). This calculation costs a lot of resources at runtime.

For the transition selection based on our transformed model, where every transition already is a compound transition, both number of triggers and guard conditions can directly be checked without needing to traverse the model.

A state configuration change is controlled by the selected transition. The state change is organized in three phases: Deactivation of the transition's source states, execution of its effects, and activation of its target states.

For instance, executing the compound transition in Fig. 2 means, that state **a** and all its sub-states are exited, then the transition's effects are executed, and at last **b** and its sub-states are entered.

By introducing a propagation concept, the implementation executes transitions in compliance to the ordering of actions as specified in the UML specification [1]. The interpreter was tested on several different of our transformed models. So far, the implementation is limited to those concepts mentioned in Sec. 4.1.

6 Discussion and Conclusion

In this paper, we have presented a concept for model transformation of state machines aiming information consolidation by folding and lifting transformations applied on compound transitions of UML state machines (cf. Sec. 2). The resulting models are used for model interpretation. In conclusion, the implementation of the interpreter profits by simplified transition selection algorithm as well as transition execution.

Refactoring of UML state machines by graph transformation was already performed by [13]. They use model transformation to normalize diagrams for better understanding of the semantics in terms of the notation itself. In the contrary, our model transformation focuses on transformations that simplify automatic processing of state machines, especially for interpretation.

Folli and Mens [14] introduce model transformations to refactor class and state machine diagrams by AGG as proof of concept for graph transformation for model refactoring.

We chose a pragmatic approach for both, definition and interpretation of our model transformation using CHR (cf. Sec. 4) and implementation of the interpreter (cf. Sec. 5). Our approach provides precise and straightforward definitions of models and model transformations.

As basis for a short comparison to related model transformation approaches, we briefly classify our transformation according to the feature model of [15]. We confine our application to unidirectional model-to-model transformation. We use CHR which naturally supports the following model transformation features: variables, patterns, logic expressions, i.e. both, executable as well as non-executable logic, separation of left-hand side from right-hand side of a rule, in-place transformation, and conditional rule selection. Furthermore, scheduling, iteration and phasing are supported implicitly. Certainly, CHR supports no scoping, i.e. all rules are checked for an entire CHR input.

Several current model transformation approaches tend to QVT [16], the model transformation standard of the OMG [17, 18]. These approaches and other ones provide powerful (diagrammatic) specification of transformation systems based on user definable meta-models [19, 20, 14, 21]. These approaches are notably beneficial for large scaled model driven development approaches. Certainly, those techniques, e.g. bidirectional transformation approaches according to QVT, maybe more complicated also if not needed as in our case. Unfortunately, the current QVT Mapping Language is far less intuitive and easy-to-use in case of unidirectional transformations [18]. [22] compare QVT and graph transformation by transforming UML state machines into Communication Sequential Processes specifications. They conclude that both approaches are rather similar. Graph transformation has its power in the clear operational idea which enhances rule specification whereas the above mentioned approaches benefit from the bidirectionality idea. According to [23] graph transformation can support bi-directionality just as well.

Others transformation languages are domain specific. For example XSLT[2], a declarative XML-based language for model transformation of XML documents seems to be suitable for our approach, too. Agreeing with [15], we think that the scalability of XSLT has limitations regarding manual implementation of model transformation. Handwritten XSLT based implementations are hard to maintain because verbosity and readability issues of XMI and XSLT. CHR provides a good

[2] XSTL (eXtensible Stylesheet Language Transformation) is a XML technology used for describing (mainly syntactic) transformations between XML files.

readability, clear semantics and additionally offers a good basis for profitability of certain concepts of an implementation.

Several approaches using model transformation successfully apply Prolog, CHR and/or CLP for different purposes, e.g. as transformation engine [24, 19, 25]. They agree that logic programming and constraint programming are valuable for model transformation.

Pretschner and Lotzbeyer [24] state that CLP and CHR are well suited for interactive generation techniques based on symbolic model execution as demonstrated in [26], too. VIATRA [19] uses Prolog for the declaration of the transformation including control flow graphs and graph transformation rules. The author states that automated transformation from source to target model using Prolog, which is hidden from the user, provides a more efficient solution as by using XSLT.

Our interpreter is implemented in Scala and is based on the formally defined UML semantics of [11]. The interpreter was implemented as proof of concept to test our resulting models and to analyze the benefits for the implementation itself. In our opinion the transition selection as well as the transition execution algorithms of our implementation profit by localization of model information resulting in a better performance of our interpretation.

7 Future Work

We are currently extending our model transformation approach using CHR, and the preliminary results are encouraging. Based on these results, the most relevant topics to substantiate, apply and extend our current work are:

- Proving of the semantics preserving property for each CHR rule
- Analyzing our approach to achieve qualitative as well as quantitative results about its benefits and drawbacks
- Applying our approach on other, similar UML diagrams, e.g. Activity Diagrams

Aside these topics, our approach can be applied to simplify code generation or test case generation approaches.

References

1. OMG. Unified Modeling Language Superstructure v2.3, 2010.
2. R. Miles and K. Hamilton. *Learning UML 2.0.* O'Reilly Media, Inc., 2006.
3. B. Selic. The Theory and Practice of Modelling Language Design for Model-Based Software Engineering — A Personal Perspective. *Lecture Notes in Computer Science*, 6491:290–321, 2011.
4. T. Frühwirth. Theory and Practice of Constraint Handling Rules. *Journal of Logic Programming, Special Issue on Constraint Logic Programming*, 37:95–138, 1998.
5. T. Frühwirth. *Constraint Handling Rules.* Cambridge University Press, 2009.
6. F. Raiser. *Graph Transformation Systems in Constraint Handling Rules: Improved Methods for Program Analysis.* PhD thesis, Ulm University, Germany, 2010.

7. J. Sneyers, P. Van Weert, T. Schrijvers, and L. De Koninck. As Time Goes By: Constraint Handling Rules – A Survey of CHR Research between 1998 and 2007. *Theory and Practice of Logic Programming*, 10(1):1–47, 2010.
8. T. Frühwirth and F. Raiser, editors. *Constraint Handling Rules – Compilation, Execution, and Analysis*. Books on Demand GmbH, Norderstedt, 2011.
9. G. J. Duck, P. J. Stuckey, M. García de la Banda, and C. Holzbaur. The Refined Operational Semantics of Constraint Handling Rules. In B. Demoen and V. Lifschitz, editors, *ICLP '04*, volume 3132 of *Lecture Notes in Computer Science*, pages 90–104. Springer-Verlag, September 2004.
10. M. Dausend. Entwicklung einer ASM-Spezifikation der Semantik der Zustandsautomaten der UML 2.0. Master's thesis, Universität Ulm, 2007.
11. J. Kohlmeyer. *Eine formale Semantik für die Verknüpfung von Verhaltensbeschreibungen in der UML 2*. PhD thesis, Universität Ulm, 2009.
12. A. Kirshin, D. Dotan, and A. Hartman. A UML Simulator Based On a Generic Model Execution Engine. *Lecture Notes in Computer Science*, 4364:324, 2007.
13. M. Gogolla and F. Parisi-Presicce. State Diagrams in UML: A Formal Semantics using Graph Transformations. In *Workshop on Precise Semantics for Modelling Techniques*, volume TUM-I9803, pages 55–72. TU München, 1998.
14. A. Folli and T. Mens. Refactoring of UML models using AGG. *ECEASST*, 8, 2007.
15. K. Czarnecki and S. Helsen. Classification of Model Transformation Approaches. In *2nd OOPSLA'03 Workshop on Generative Techniques in the Context of MDA*, pages 1–17, Anaheim, CA, USA, 2003.
16. OMG. Meta Object Facility (MOF) 2.0 Query/View/Transformation. 1.1, 2011.
17. A. Königs. Model Transformation with Triple Graph Grammars. In *Model Transformations in Practice Satellite Workshop of MODELS*, 2005.
18. A. Balogh and D. Varró. Advanced Model Transformation Language Constructs in the VIATRA2 Framework. In *Proceedings of the 2006 ACM Symposium on Applied Computing*, pages 1280–1287. ACM, 2006.
19. D. Varró. Automated Program Generation for and by Model Transformation Systems. *Applied Graph Transformation (AGT'02)*, pages 161–174, 2002.
20. L. Geiger and A. Zündorf. Statechart Modeling with Fujaba. *Electronic Notes in Theoretical Computer Science*, 127(1):37–49, 2005.
21. B. Schätz. Formalization and Rule-Based Transformation of EMF Ecore-Based Models. *Software Language Engineering*, pages 227–244, 2009.
22. J.M. Küster, S. Sendall, and M. Wahler. Comparing two Model Transformation Approaches. In *Workshop on OCL and Model Driven Engineering*, 2004.
23. A. Cicchetti, D. Di Ruscio, R. Eramo, and A. Pierantonio. Logical constraints for managing non-determinism in bidirectional model transformations. In *Model-Driven Engineering, Logic and Optimization: friends or foes? (MELO 2011)*, 2011.
24. A. Pretschner and H. Lötzbeyer. Model Based Testing with Constraint Logic Programming: First Results and Challenges. In *Proc. 2nd ICSE Intl. Workshop on Automated Program Analysis, Testing and Verification (WAPATV'01)*, volume 2, Toronto, 2001.
25. J.M. Almendros-Jiménez and L. Iribarne. ODM-based UML Model Transformations using Prolog. In *Model-Driven Engineering, Logic and Optimization: friends or foes? (MELO 2011)*, 2011.
26. A. Ciarlini and T. Frühwirth. Automatic Derivation of Meaningful Experiments for Hybrid Systems. In *Proc. ACM SIGSIM Conf. on Artificial Intelligence, Simulation, and Planning (AIS'00)*, 2000.

Author Index