Modern Constraint Answer Set Solving

Dissertation
von
Max Ostrowski

zur Erlangung des akademischen Grades
Doktor der Naturwissenschaften (Dr. rer. nat.)
in der Wissenschaftsdisziplin
"Wissensverarbeitung und Informationssysteme"

eingereicht an der
Mathematisch-Naturwissenschaftlichen Fakultät
der Universität Potsdam

angefertigt am 26. Februar 2018, Institut für Informatik
Professur Wissensverarbeitung und Informationssysteme
betreut von Prof. Dr. Torsten Schaub

This work is licensed under a Creative Commons License:
Attribution International
To view a copy of this license visit
http://creativecommons.org/licenses/by/4.0/

Published online at the
Institutional Repository of the University of Potsdam:
URN urn:nbn:de:kobv:517-opus4-407799
http://nbn-resolving.de/urn:nbn:de:kobv:517-opus4-407799

Abstract

Answer Set Programming (ASP) is a declarative problem solving approach, combining a rich yet simple modeling language with high-performance solving capabilities. Although this has already resulted in various applications, certain aspects of such applications are more naturally modeled using variables over finite domains, for accounting for resources, fine timings, coordinates, or functions. Our goal is thus to extend ASP with constraints over integers while preserving its declarative nature. This allows for fast prototyping and elaboration tolerant problem descriptions of resource related applications. The resulting paradigm is called Constraint Answer Set Programming (CASP).

We present three different approaches for solving CASP problems. The first one, a lazy, modular approach combines an ASP solver with an external system for handling constraints. This approach has the advantage that two state of the art technologies work hand in hand to solve the problem, each concentrating on its part of the problem. The drawback is that inter-constraint dependencies cannot be communicated back to the ASP solver, impeding its learning algorithm. The second approach translates all constraints to ASP. Using the appropriate encoding techniques, this results in a very fast, monolithic system. Unfortunately, due to the large, explicit representation of constraints and variables, translation techniques are restricted to small and mid-sized domains. The third approach merges the lazy and the translational approach, combining the strength of both while removing their weaknesses. To this end, we enhance the dedicated learning techniques of an ASP solver with the inferences of the translating approach in a lazy way. That is, the important knowledge is only made explicit when needed.

By using state of the art techniques from neighboring fields, we provide ways to tackle real world, industrial size problems. By extending CASP to reactive solving, we open up new application areas such as online planning with continuous domains and durations.

Zusammenfassung

Die Antwortmengenprogrammierung (ASP) ist ein deklarativer Ansatz zur Problemlösung. Eine ausdrucksstarke Modellierungssprache erlaubt es, Probleme einfach und flexibel zu beschreiben. Durch sehr effiziente Problemlösungstechniken, konnten bereits verschiedene Anwendungsgebiete erschlossen werden. Allerdings lassen sich Probleme mit Ressourcen besser mit Gleichungen über Ganze oder Reelle Zahlen lösen, anstatt mit reiner Boolescher Logik. In dieser Arbeit erweitern wir ASP mit Arithmetik über Ganze Zahlen zu Constraint Answer Set Programming (CASP). Unser Hauptaugenmerk liegt dabei auf der Erweiterung der Modellierungssprache mit Arithmetik, ohne Performanz oder Flexibilität einzubüßen.

In einem ersten, bedarfsgesteuertem, modularen Ansatz kombinieren wir einen ASP Solver mit einem externen System zur Lösung von ganzzahligen Gleichungen. Der Vorteil dieses Ansatzes besteht darin, dass zwei verschiedene Technologien Hand in Hand arbeiten, wobei jede nur ihren Teil des Problems betrachten muss. Ein Nachteil der sich daraus ergibt ist jedoch, dass Abhängigkeiten zwischen den Gleichungen nicht an den ASP Solver kommuniziert werden können. Das beeinträchtigt die Lernfähigkeit des zu Grunde liegenden Algorithmus. Der zweite von uns verfolgte Ansatz übersetzt die ganzzahligen Gleichungen direkt nach ASP. Durch entsprechende Kodierungstechniken erhält man ein sehr effizientes, monolithisches System. Diese Übersetzung erfordert eine explizite Darstellung aller Variablen und Gleichungen. Daher ist dieser Ansatz nur für kleine bis mittlere Wertebereiche geeignet. Die dritte Methode, die wir in dieser Arbeit vorstellen, vereinigt die Vorteile der beiden vorherigen Ansätze und überwindet ihre Kehrseiten. Wir entwickeln einen lernenden Algorithmus, der die Arithmetik implizit lässt. Dies befreit uns davon, eine möglicherweise riesige Menge an Variablen und Formeln zu speichern, und erlaubt es uns gleichzeitig dieses Wissen zu nutzen.

Das Ziel dieser Arbeit ist es, durch die Kombination hochmoderner Technologien, industrielle Anwendungsgebiete für ASP zu erschliessen. Die verwendeten Techniken erlauben eine Erweiterung von CASP mit reaktiven Elementen. Das heißt, dass das Lösen des Problems ein interaktiver Prozess wird. Das Problem kann dabei ständig verändert und erweitert werden, ohne dass Informationen verloren gehen oder neu berechnet werden müssen. Dies eröffnet uns neue Möglichkeiten, wie zum Beispiel reaktives Planen mit Ressourcen und Zeiten.

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig angefertigt, nicht anderweitig zu Prüfungszwecken vorgelegt und keine anderen als die angegebenen Hilfsmittel verwendet habe. Sämtliche wissentlich verwendeten Textausschnitte, Zitate oder Inhalte anderer Verfasser wurden ausdrücklich als solche gekennzeichnet.

Potsdam, 26. Februar 2018

____

Max Ostrowski

Acknowledgments

This is for all people, constantly asking "Are you still not finished with your PhD?" — It is done. At last.

First and foremost, I would like to thank my supervisor Prof. Dr. Torsten Schaub for his continuous support. He manages a great team of researchers, while respecting various ways of living and working and never even bothered when you had to take care of your child or simply work from home. Also, this work would not have been possible without the daily chat with my colleagues, quite some of them I call friends now. Finally, and most importantly, I thank my wife and friends for supporting me all the time.

This work was partially funded by DFG grant SCHA 550/9.

Contents

1 Introduction 3
1.1 Contributions . . . 4
1.2 Outline . . . 7

2 Background 9
2.1 Constraint Answer Set Programming . . . 10
2.2 Logic Programs . . . 10
2.3 Constraint Satisfaction Problems . . . 12
2.4 Constraint Stable Models . . . 13

3 Constraint Answer Set Programming via Conflict Driven Constraint Learning 15
3.1 Boolean Constraint Solving . . . 16
3.2 Basic Conflict Driven Constraint Learning . . . 20
3.3 Architecture . . . 23
3.4 Extended Conflict Driven Constraint Learning . . . 26
3.5 Reason and Conflict Reduction . . . 29
3.6 Implementation Techniques . . . 33
3.6.1 Lazy Reasons . . . 33
3.6.2 Global Constraints . . . 34
3.6.3 Initial Lookahead . . . 34
3.6.4 Optimization . . . 35
3.7 Evaluation . . . 35
3.8 Conclusion . . . 41

4 Encoding Constraint Satisfaction Problems 43
4.1 Normalizing Constraints . . . 44
4.2 Encoding Linear Constraints . . . 45
4.3 Nogoods of Constraint Satisfaction Problems . . . 47
4.4 Encoding Constraint Satisfaction Problems . . . 50
4.4.1 Encoding Constraint Answer Set Programs . . . 57
4.5 Evaluation . . . 58
4.6 Conclusion . . . 61

5 Lazy Nogood and Variable Generation 63
5.1 Constraint Stable Models in Terms of Propagators 63
5.2 System Design 65
5.2.1 Architecture 65
5.2.2 Language 66
5.2.3 Algorithms 70
5.3 Distinguished Features 78
5.4 Evaluation 86

6 Multi-Shot Constraint Answer Set Programming 101
6.1 Multi-Shot Solving 101
6.2 Incremental Programs 102
6.3 Incremental N-Queens 103
6.4 Planning with Durations 108

7 Related Work 111
7.1 Logic Programs Modulo Theories 111
7.2 Comparing Different Semantics and Systems 112

8 Conclusion 123

List of Figures

2-1 Workflow of ASP. 9

3-1 Architecture of *clingcon* 1 + 2. 25
3-2 Evaluating filtering techniques wrt. conflict size.. 39
3-3 Evaluating filtering techniques wrt. runtime. 39

4-1 Architecture of *aspartame*. 44
4-2 A graphical representation of the linear constraint $x + y \leq 7$. 46

5-1 Architecture of *clingcon* 3. 65

List of Tables

3.1 Computing the stable model $\{light, night, switchOn, x < 7, x \geq 22\}$ with CDCL(P_1). . . . 24
3.2 Conflict analysis of $\{\mathbf{T}light, \mathbf{F}\{switchOn\}, \mathbf{F}\{\sim night\}\}$. . . . 24
3.3 Comparing *clingo* 2.0.2, *adsolver*, and *clingcon* 1. . . . 36
3.4 Evaluating lazy nogood generation within *clingcon*. . . . 37
3.5 Evaluating filtering techniques. . . . 40
3.6 The effects of theory propagation wrt. runtime. . . . 40
3.7 Initial Lookahead (*I.L.*). . . . 41

4.1 Experiments comparing different encodings with sugar. . . . 59
4.2 Experiments comparing different encodings for alldifferent. . . . 60

5.1 Order literals of different views of one variable. . . . 78
5.2 Constraint logic programs using reified $\mathbf{T}(x > 7) \Leftrightarrow x > 7$ and half-reified $\mathbf{T}(x > 7)' \Rightarrow x > 7$ constraints. . . . 84
5.3 Default configuration D of *clingcon* 3.2.0. . . . 87
5.4 Comparison of different features of *clingcon* 3.2.0 on the benchmark set of the *minizinc* competition 2015. Shown are scores of how often a configuration is better than another. Bold numbers indicate the best configuration for the benchmark class. . . . 88
5.5 Comparing *clingcon* 3.2.0 DT with different state of the art CP solvers on the *minizinc* competition 2015 benchmark set. . . . 93
5.6 Comparison of different CASP systems on the two dimensional strip packing problem. . . . 95
5.7 Comparison of different CASP systems on the incremental scheduling problem. . . . 98
5.8 Comparison of different CASP systems on the weighted sequence problem. 99
5.9 Comparison of different CASP systems on the reverse folding problem. 100

6.1 Comparison of different incremental n-queens programs. . . . 107

7.1 Feature comparison of different systems. . . . 113

Chapter 1

Introduction

Answer Set Programming (ASP;[15, 87, 95]) is a declarative problem solving approach, combining a rich yet simple modeling language with high-performance solving capabilities. This approach has already resulted in various applications. Among them, a decision support systems for NASA shuttle controllers [7, 98], product configuration [111], scheduling [70], timetabling [12], shift design [1] and various reasoning tools in systems biology [16, 65]. However, certain aspects of such applications are more naturally modeled using variables over finite domains, accounting for resources, fine timings, coordinates, or functions. Consider a complex planning problem, where the task is to schedule different machines, each of them consuming and producing goods, while at the same time using energy. All these resources, like runtime, power, fuel and storage can hardly be handled using purely propositional approaches, as a problem inherent to all these solving approaches is the grounding bottleneck. That is, the need to explicitly represent every combination of values in a constraint. Therefore, a dedicated treatment of variables and constraints over integers is needed, as done in constraint processing (CP;[38, 106]).

Our goal is to extend ASP with constraints over integers while preserving its declarative nature and excellent performance. The resulting paradigm is called Constraint Answer Set Programming (CASP) and can be used for fast prototyping of elaboration tolerant problem descriptions of resource related applications. Groundbreaking work on enhancing ASP with CP techniques for handling (integer) arithmetics was conducted in [19, 90, 91]. Based on firm semantical underpinnings, this approach provides a family of ASP languages parameterized by different constraint classes. While [19] develops a high-level algorithm viewing both ASP and CP solvers as black boxes, [91] embeds a black-boxed CP solver into a traditional, backtracking based ASP solver. This resulted in two approaches to handle ASP with constraints. Unfortunately, neither of them uses elaborated conflict-driven learning techniques from Satisfiability Checking (SAT;[23, 94]), and therefore is not matching the performance of state of the art problem solving solvers. We address this problem and propose several alternative ways to combine modern ASP and CP solving techniques to handle CASP problems.

1.1 Contributions

This thesis focuses on CASP and how to solve problems within this paradigm. Our goal is to conserve the declarative nature and elaboration tolerance of ASP and develop refined techniques to tackle real world, industrial problems. Therefore, we concentrate on state of the art solving techniques like learning algorithms as well as modeling techniques such as multi-shot solving. We now present the four main parts of our contribution, all of them published.

Constraint Answer Set Programming via Conflict Driven Constraint Learning We define CASP, pursuing a semantic approach that is based on a propositional language rather than a multi-sorted, first-order language, as used in [19, 90, 91]. This allows us to use conflict-driven constraint learning (CDCL;[62]) technology for solving propositional problems. These learning algorithms are the state of the art solution to Boolean satisfiability problems and have been well researched since the mid-90s. We use and extend these sophisticated algorithms for solving CASP problems. Our extension follows the so-called lazy approach of advanced Satisfiability Modulo Theories (SMT;[97]) solvers by abstracting from the constraints in a specialized theory. The idea is as follows. During solving, the ASP solver passes its (partial) information to a CP solver, which checks the implied constraints via constraint propagation. As a result, it either signals that no solution exists or, if possible, extends the knowledge base of the ASP solver. To facilitate learning within the ASP solver, however, each inference must be justified by providing a “reason” for the underlying algorithms. Yet, to the best of our knowledge, this is not supported by off-the-shelf CP solvers.[1]

We show the correctness of our approach by proving the relation between the definition of CASP and its characterization using Boolean propositions. As a consequence, we develop an algorithmic framework for conflict-driven ASP solving that integrates CP solving capabilities while overcoming the aforementioned difficulty. This results in the system *clingcon* 1, outperforming previous approaches. Additionally, it can handle optimization over constraint variables and global constraints. In a second step, the algorithmic framework is extended by filtering techniques based on Irreducible Inconsistent Sets (IIS;[68, 120]). This technique strengthens the provided conflicts and improves the learning capabilities of the whole approach.

Encoding Constraint Satisfaction Problems For solving Constraint Satisfaction Problems (CSPs), the preferred method is not so clear and new approaches have been developed during the last years. Having a standard, non-learning CP solver has the benefit of supporting special (global) constraint propagators for various kinds of constraints. An implicit variable/domain representation supports huge or even infinite domains. On the other hand, encoding finite linear CSPs as propositional formulas and solving them by using modern solvers for SAT [73] has proven to be a highly effective approach by the award-winning *sugar*[2] system. This system transforms a

[1] Advanced SMT solvers, like [97], address this through handcrafted theory solvers.
[2] `http://bach.istc.kobe-u.ac.jp/sugar`

CSP into a propositional formula in Conjunctive Normal Form (CNF). The translation relies on the order encoding [31, 116], and the resulting CNF formula can be solved by an off-the-shelf SAT solver. We elaborate upon an alternative approach based on ASP and present the resulting *aspartame*[3] framework. It constitutes an ASP-based CP solver similar to *sugar*. The major difference between *sugar* and *aspartame* rests upon the implementation of the translation of CSPs into Boolean constraint problems. While *sugar* implements a translation into CNF in Java, *aspartame* starts with a translation into a set of facts. These facts are combined with a general-purpose ASP encoding for CP solving (also based on the order encoding), to be solved by an ASP solver. We extend the used techniques to provide an ASP library for solving CASP.

Lazy Nogood and Variable Generation Our first approach presented in this section used to handle CASP consists of a learning ASP solver in combination with a non-learning CP solver. Without learning algorithms, such CP solvers rest upon an implicit variable representation. It permits huge domains and thus avoids the grounding bottleneck, but also restricts information exchange which impedes the CDCL algorithm used by the learning ASP solver. On the other hand, the translation approach, encoding CASP using ASP, explicitly represents integer variables and therefore benefits from the full power of CDCL. The granularity induced by this representation provides accurate conflict and propagation information. The downside of this is its limited scalability due to the size of the translation. We therefore present an approach combining the use of CDCL with an explicit representation that overcomes the named weaknesses of the two approaches. Inspired by the work of [44, 99, 118], we are using dedicated propagators to implicitly represent the encoding of the constraints and create the necessary propositions and variables whenever needed. Hence, we neither need to make the constraints nor the variables explicit a priori but create them on demand. In combination with a generic, declarative theory language and sophisticated preprocessing techniques, we provide a full fledged implementation of a modern CASP solver, named *clingcon* 3. We evaluate our system and compare it with state of the art CP and CASP solvers. Also, we provide a tool for translating CP benchmarks in the *minizinc* format into the internal ASP format. This enables the CASP community to take advantage of a whole new class of benchmarks.

Multi-Shot Constraint Answer Set Programming Multi-shot ASP solving [57, 58, 59] is about solving continuously changing logic programs in an operative way. This can be controlled via reactive procedures that loop on solving while reacting, for instance, to outside changes or previous solving results. Such reactions may entail the addition or retraction of rules that the operative approach can accommodate by leaving the unaffected program parts intact within the solver. This avoids re-grounding and benefits from heuristic scores and constraints learned over time. Evolving constraint logic programs can be extremely useful in dynamic applications to add new resources, set observed variables, and add or relieve restrictions on capacities. To extend multi-shot solving to CASP, *clingcon* 3 allows us to add and delete constraints in order

[3] `https://potassco.org/labs/2016/09/20/aspartame.html`

to capture evolving CSPs. New resources can be added using additional constraint variables and domains. While restricting variables by adding constraints and rules to the constraint logic program is easy, increasing their capacity is not. The key to this is lazy variable generation that allows us to avoid making huge domains explicit. For this purpose, we start with a virtually maximum domain that is restrained by retractable constraints. The domain is then increased by relaxing these constraints. After providing an example, we present a first approach to plan with durations using the presented techniques. Therefore, we took an extended version of the famous Yale shooting problem considering actions with durations.

We conclude the thesis with a comparison of different approaches to constraint answer set solving by contrasting around 20 systems.

Publications Most parts of this thesis have been published in international journals and proceedings of international conferences. We give below the chapters and the respective publications. Note that several additions like theorems and proofs have been made to the publications. Definitions and declarations have been unified.

Chapter 2 and 3

- M. Gebser, M. Ostrowski, and T. Schaub. Constraint answer set solving. In P. Hill and D. Warren, editors, *Proceedings of the Twenty-fifth International Conference on Logic Programming (ICLP'09)*, volume 5649 of *Lecture Notes in Computer Science*, pages 235–249. Springer-Verlag, 2009

- M. Ostrowski and T. Schaub. ASP modulo CSP: The clingcon system. *Theory and Practice of Logic Programming*, 12(4-5):485–503, 2012

Chapter 4

- M. Banbara, M. Gebser, K. Inoue, M. Ostrowski, A. Peano, T. Schaub, T. Soh, N. Tamura, and M. Weise. aspartame: Solving constraint satisfaction problems with answer set programming. In F. Calimeri, G. Ianni, and M. Truszczyński, editors, *Proceedings of the Thirteenth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'15)*, volume 9345 of *Lecture Notes in Artificial Intelligence*, pages 112–126. Springer-Verlag, 2015

Chapter 5 and 6

- M. Banbara, B. Kaufmann, M. Ostrowski, and T. Schaub. Clingcon: The next generation. *Theory and Practice of Logic Programming*, 17(4):408–461, 2017

- T. Janhunen, R. Kaminski, M. Ostrowski, T. Schaub, S. Schellhorn, and P. Wanko. Clingo goes linear constraints over reals and integers. *Theory and Practice of Logic Programming*, 17(5-6):872–888, 2017

The development of the different solving strategies for CASP problems was inspired by my work on different problems in systems biology, like the evolution of ontologies [100], automated network reconstruction [45] and Boolean network identification [71, 101].

1.2 Outline

In Chapter 2, we capture the necessary background for CASP. We start with an informal description of the workflow for using ASP and why it is necessary to extend it to CASP. After a description of CASP, we formally introduce ASP and define its solutions using the Gelfond-Lifschitz reduct on logic progams. Since CASP is an amalgamation of ASP and CP, a proper definition of a CSP is given. This finally allows us to characterize the solutions of a CASP problem as constraint stable models.

In Chapter 3, we start with a characterization of a logic program and its solutions (stable models) using nogoods and Boolean assignments. This characterization is extended with *externals* [44], and we come up with a basic CDCL algorithm for solving such programs. To compute constraint stable models, an extension of the basic algorithm is provided and exemplified. To improve the learning facilities of CDCL, designated techniques are presented to improve the interplay between ASP and CP. Finally, we propose the CASP systems *clingcon* 1 and *clingcon* 2 and evaluate their performance wrt. previously developed CASP approaches.

In Chapter 4, we contrast the learning approach with an eager, translational concept by translating a CSP into Boolean nogoods. After a short discourse about normalizing non-linear constraints into linear ones, we give a concise overview of several encoding techniques, such us direct, logarithmic, support, range, and order encoding. As the latter enforces bound consistency on linear constraints and can reduce tractable CSP to tractable SAT [103], we use it to characterize constraint stable models using an ASP encoding. Therefore, CASP problems can be encoded using exactly the same encoding. Finally, we evaluate the presented techniques and compare them wrt. the state of the art CP solver *sugar*.

In Chapter 5, the learning and the translational approach are combined using lazy nogood and variable generation. As a foundation for this approach, we characterize solutions of CASP in terms of nogoods and propagators. This is followed by a description of the design of the CASP solver *clingcon* 3, defining its architecture, input language and algorithms. As this system represents nogoods of the order encoding implicitly, new preprocessing techniques are developed. To assess the performance of our approach, we start with an extensive evaluation of its features. Then, we compare our system to state of the art CP solvers, using benchmarks from the *minizinc* competition 2016. Finally, CASP benchmarks are used to evaluate and confront our systems with other modern CASP solvers.

In Chapter 6, an extension to multi-shot ASP using CASP is presented. Given the circumstance that lazy variable generation can handle variables with huge domains, we elaborate on the resulting possibilities for multi-shot solving. In fact, we can add and remove constraints as well as increase or decrease the domain of variables. We

show these features using an encoding for the n-queens problem and evaluate different techniques in terms of performance, number of atoms and nogoods. Afterwards, we apply these techniques to encode a variant of the well known Yale shooting problem.

In Chapter 7, we conclude the thesis with an overview of related systems and paradigms. Therefore, we start with a definition of our CASP semantics in terms of the more general framework of ASP modulo theories. We then briefly compare it with other CASP paradigms, namely $\mathcal{AC}$ [90], $\mathcal{EZ}$ [5, 9], $\mathcal{ASPMT}$ [18], Here and There with Constraints [26], and Bound Founded ASP [3, 4]. Also, several systems have been developed and we shortly address their features and shortcomings.

Chapter 2

Background

ASP combines a high level modeling language with state of the art Boolean constraint solving technology. In this paradigm, it is unnecessary to describe how to solve the problem but to define the problem [56, 78]. This workflow is shown in Figure 2-1. After

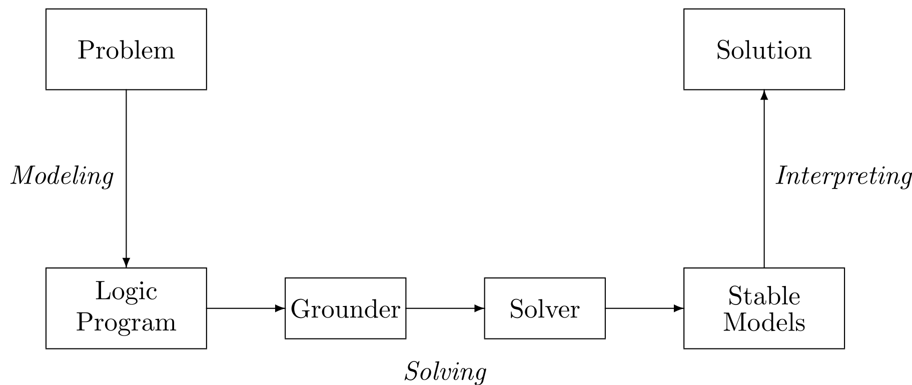

Figure 2-1: Workflow of ASP.

describing the problem, it is modeled as a logic program using first order variables. The grounding process is responsible for replacing all occurrences of first order variables with constants from the Herbrand universe [86]. Afterwards, the grounded program is solved using an ASP solver, computing the stable models of it. Modern ASP solvers are based on CDCL algorithms using nogoods. The stable models of the program can then be interpreted as solutions to the problem.

The process of grounding enables us to use fast CDCL algorithms to solve the problem. One drawback is the combinatorial nature of this process that can result in a very large representation of the problem, called the grounding bottleneck. This becomes a problem when modeling resources or functions over large, finite domains. To partially overcome this problem, CASP uses constraints over integer or real valued variables. One main goal of this thesis is to provide and compare efficient approaches to overcome the grounding bottleneck using CASP.

In this chapter, we give the definitions of CASP and its prerequisites.

- We start with the basics of constraint logic programs and give reference to their

origin. Since constraint logic programs form a combination of logic programs and CSPs, we:

- define logic programs and the semantics of ASP using the Gelfond-Lifschitz reduct [66],
- present the general concept of a CSP over integer variables and its solutions, and finally
- combine these concepts to define the semantics of constraint logic programs.

2.1 Constraint Answer Set Programming

CASP has been developed to combine the advantages of ASP and CP. Thus keeping a declarative modeling language in combination with a very fast Boolean search engine and extending it with capabilities to handle constraints over large domains. It has its origins in [19, 90, 91] which have been unified in [80]. In the constraints, integer or real valued variables in ASP programs are used and enable ASP to handle large quantities instead of Boolean states in a natural way. It has been proven useful for expressing constraints over resources, timings, and others. Logic programs, involving Boolean and integer variables and constraints are called *constraint logic programs*.

A constraint logic program consists of a logic program P over disjoint sets $\mathcal{A}, \mathcal{C}$ of propositional variables, and an associated CSP $(\mathcal{V}, D, C)$. Elements of $\mathcal{A}$ and $\mathcal{C}$ are referred to as regular and constraint atoms, respectively. The CSP consists of a set of integer variables $\mathcal{V}$, a set D of corresponding variable domains, and a set C of constraints.

2.2 Logic Programs

A constraint logic program P consists of rules of the form[1]

$$a_0 \leftarrow a_1, \ldots, a_m, \sim a_{m+1}, \ldots, \sim a_n \tag{2.1}$$

where $0 \leq m \leq n$ and $a_0 \in \mathcal{A}$ and each $a_i \in \mathcal{A} \cup \mathcal{C}$ is an atom for $1 \leq i \leq n$.

As an example[2], consider the logic program P_1:

[1] We present our approach in the context of normal logic programs, though it readily applies to disjunctive logic programs with aggregates — as does *clingcon* 3.
[2] This example was inspired by [9].

Example 1

$$switchOn \leftarrow {\sim} switchOff \tag{2.2}$$
$$switchOff \leftarrow {\sim} switchOn \tag{2.3}$$
$$light \leftarrow switchOn \tag{2.4}$$
$$light \leftarrow {\sim} night \tag{2.5}$$
$$night \leftarrow (x < 7) \tag{2.6}$$
$$night \leftarrow (x \geq 22) \tag{2.7}$$
$$sleep \leftarrow switchOff, night \tag{2.8}$$

This program contains regular atoms *switchOn*, *switchOff*, *light*, *night*, and *sleep* from $\mathcal{A}$ along with the constraint atoms $(x < 7)$ and $(x \geq 22)$ from $\mathcal{C}$. Accordingly, x is an integer variable in $\mathcal{V}$. The programs intuitive meaning is that we can either switch the light on or off. There is light, if we switched it on or it is not night. Despite other opinions, we define night to be between 22:00 (10 P.M.) and 7:00 (7 A.M.). The last line states that we can sleep if it is night and the light is switched off.

We need the following auxiliary definitions. We define $head(r) = a_0$ as the heads of rules r in (2.1), $body(r) = \{a_1, \ldots, a_m, {\sim} a_{m+1}, \ldots, {\sim} a_n\}$ as its body, and $atom(r) = \{a_0, a_1, \ldots, a_m, a_{m+1}, \ldots, a_n\}$. Moreover, we let $head(P) = \{head(r) \mid r \in P\}$, $body(P) = \{body(r) \mid r \in P\}$, $body_P(a) = \{body(r) \mid r \in P, head(r) = a\}$, and $atom(P) = \{atom(r) \mid r \in P\}$. While $body(r)^+ = \{a_1, \ldots, a_m\}$ denotes the positive part of the body of a rule r, $body(r)^- = \{a_{m+1}, \ldots, a_n\}$ denotes the negative one. We can project $body(r)$ on a set of atoms $\mathcal{A}$, written as $body(r)|_{\mathcal{A}} = (body(r)^+ \cap \mathcal{A}) \cup \{{\sim} a \mid a \in body(r)^- \cap \mathcal{A}\}$. If $body(r) = \emptyset$, r is called a fact. If $head(r)$ is missing, r is called an integrity constraint and r stands for $x \leftarrow body(r), {\sim} x$ where x is a new atom.[3] Whenever $\mathcal{C} = \emptyset$, a constraint logic program is called a normal logic program.

In ASP, the semantics of a normal logic program P is given by its *stable models* [64, 66]. These are defined using the reduct of a normal logic program. The *reduct*, P^X, of P relative to a set of atoms X is defined by

$$P^X = \{head(r) \leftarrow body(r)^+ \mid r \in P, body(r)^- \cap X = \emptyset\}$$

Note that P^X is a positive program possessing a unique $\subseteq$-minimal model (cf. [41]). Given this, X is a *stable model* of normal logic program P, if X itself is the $\subseteq$-minimal model of P^X. Note that any stable model of P is a $\subseteq$-minimal model of P as well, while the converse does not hold in general. A logic program is *tight*, if it does not have any cycles in its positive dependency graph

$$G(P) = (atom(P), \{(a, b) \mid r \in P, a \in body(r)^+, b \in head(r)\})$$

[3] As syntactic sugar, a rule $c \leftarrow a_1, \ldots, a_m, {\sim} a_{m+1}, \ldots, {\sim} a_n$ with a constraint atom $c \in \mathcal{C}$ in the head stands for $\leftarrow a_1, \ldots, a_m, {\sim} a_{m+1}, \ldots, {\sim} a_n, {\sim} c$.

While P_1 is a tight program, the program

$$a \leftarrow b$$
$$b \leftarrow a$$

is non-tight. We refer to [15] for a comprehensive introduction to ASP.

To extend this characterization to programs with constraint atoms, it is important to realize that the truth value of such atoms is determined external to the program. In CASP, this is reflected by the requirement that constraint atoms must not occur in the head of rules.[4] Hence, treating constraint atoms as regular ones leaves them unfounded, which means that they would never occur in a $\subseteq$-minimal model of P.

2.3 Constraint Satisfaction Problems

A *constraint satisfaction problem* (CSP) is a triple $(\mathcal{V}, D, C)$, where $\mathcal{V}$ is a set of *variables* with respective *domains* D, and C is a set of *constraints*. The domain of a variable $v \in \mathcal{V}$ is given by $D(v) \in D$. The complement of a constraint $c \in C$ is denoted as $\overline{c}$. We require C to be closed under complements. Following [38], a constraint c is a pair (S, R) consisting of a k-ary *relation* R defined on a vector of variables $S \in V^k$, called the *scope* of R. That is, for $S = (v_1, \dots, v_k)$, we have $R \subseteq D(v_1) \times \cdots \times D(v_k)$. Given that our relations have a schema, we adapt the definition of relations and tuples used in database theory for making the ordering of values within tuples obsolete (cf. [46, p. 141]). For our examples, we often use constraints c of the form

$$a_1 v_1 + \cdots + a_k v_k \circ b \tag{2.9}$$

where a_i, b are integers, $v_i \in \mathcal{V}$ for $1 \leq i \leq k$ and $\circ \in \{<, \leq, >, \geq, =, \neq\}$. The scope of such an arithmetic constraint can easily be obtained as $c = ((v_1, \dots, v_k), \{(x_1, \dots, x_k) \mid a_1 x_1 + \cdots + a_k x_k \circ b,\ x_i \in D(v_i), 1 \leq i \leq k\})$.

In Example 1, we have $x \in \mathcal{V}$ and let $D(x) = \{0, \dots, 23\}$. Since we require C to be closed under complements, it contains both $x < 7$ and its complement $x \geq 7$, as well as $x \geq 22$ and its complement $x < 22$.

An assignment $\mathbf{C} : v \in \mathcal{V} \mapsto d \in D(v)$ *satisfies* a constraint $c = (\{v_1, \dots, v_k\}, R)$, if $(\mathbf{C}(v_1), \dots, \mathbf{C}(v_k)) \in R$. A set C of constraints is called *conflicting*, if there does not exist an assignment $\mathbf{C}$ that satisfies all constraints $c \in C$. We let $sat_{\mathbf{C}}(C)$ denote the set of all constraints in C satisfied by $\mathbf{C}$. Following [42], we call $(\mathbf{C}, sat_{\mathbf{C}}(C))$ a *configuration* of $(\mathcal{V}, D, C)$. For instance, the assignment $\mathbf{C} = \{x \mapsto 5\}$ satisfies the constraints $x < 7$ and $x < 22$. Accordingly, $(\{x \mapsto 5\}, \{x < 7, x < 22\})$ is a configuration of $(\{x\}, \{D(x)\}, \{x < 7, x \geq 7, x \geq 22, x < 22\})$.

Moreover, we rely on the CP concept of a *view*. Following [108], a *view* on a variable x is an expression $ax + b$ for integers a, b; its *image* is defined as $img(ax+b) = \{ax + b \mid x \in D(x)\}$.[5] Since a view $ax + b$ can always be replaced with a fresh

[4] In alternative semantic settings, theory atoms may also occur as rule heads (cf. [53]).

[5] Any linear expression with one variable can be converted to an expression of the form $ax + b$.

variable y along with a constraint $y = ax + b$, we may use them nearly everywhere where we would otherwise use variables. For a view v, we define $lb(v)$ and $ub(v)$ as the smallest/largest value in $img(v)$.[6] Then, $prev(d, v)$ ($next(d, v)$) is a function mapping a value d to the largest (smallest) element d' in $img(v)$ which is smaller (larger) than d if $d > lb(v)$ ($d < ub(v)$), otherwise it is $-\infty$ (∞). In our example, we have $lb(x) = 0$ and $ub(x) = 23$, and for instance $prev(17, 2x + 3) = 15$, $next(5, x) = 6$, and $prev(0, x) = -\infty$, respectively.

2.4 Constraint Stable Models

Given a constraint logic program P over regular atoms $\mathcal{A}$ and constraint atoms $\mathcal{C}$ associated with CSP $(\mathcal{V}, D, C)$. We identify constraint atoms in $\mathcal{C}$ with constraints in C via a function $\gamma : \mathcal{C} \mapsto C$. Sometimes we abuse notation and use γ to identify a set of constraints atoms with a set of constraints. Given an assignment $\mathbf{C} : \mathcal{V} \mapsto D$ and the constraint logic program P, we define the *constraint reduct* as

$$P^{\mathbf{C}} = \{head(r) \leftarrow body(r)|_{\mathcal{A}} \mid r \in P, body(r) \in body^{C}_{\mathbf{C}}(P)\}$$

where $body^{C}_{\mathbf{C}}(P) = \{body(r) \mid r \in P, \gamma(body(r)^{+}) \cap \mathcal{C} \subseteq sat_{C}(\mathbf{C}), \gamma(body(r)^{-}) \cap sat_{C}(\mathbf{C}) = \emptyset\}$ is the set of bodies where all constraints are satisfied. This basically means that all constraints are evaluated wrt. the constraint assignment and removed from the logic program.

In our example, we associate the constraint atom $(x < 7)$ with the constraint $x < 7$, or formally, $\gamma(x < 7) = x < 7$. The constraint assignment $\mathbf{C} = \{x \mapsto 5\}$ satisfies the constraints $\{x < 7, x < 22\} \subseteq C$. Its constraint reduct $P^{\mathbf{C}}$ is therefore given as

$$\begin{aligned}
switchOn &\leftarrow {\sim} switchOff\\
switchOff &\leftarrow {\sim} switchOn\\
light &\leftarrow switchOn\\
light &\leftarrow {\sim} night\\
night &\leftarrow\\
sleep &\leftarrow switchOff, night
\end{aligned}$$

We can now define the constraint stable models of a constraint logic program in compliance with [64].

[6] Note that for a view of the form $1x + 0$ we have $D(x) = img(x)$.

Definition 1

Let P be a constraint logic program over regular atoms $\mathcal{A}$ and constraint atoms $\mathcal{C}$ associated with CSP $(\mathcal{V}, D, C)$. Furthermore, let $\mathbf{C} : \mathcal{V} \mapsto D$ be an assignment of integer variables and $X \subseteq \mathcal{A} \cup \mathcal{C}$ a set of atoms.
Then, $(X, \mathbf{C})$ is a *constraint stable model* of P iff $X \cap \mathcal{C} = \{c \mid \gamma(c) \in sat_{\mathbf{C}}(C)\}$ and $X \cap \mathcal{A}$ is the $\subseteq$-smallest model of the program $(P^{\mathbf{C}})^X$.

Accordingly, our example yields the following constraint stable models

$$\begin{array}{ll} X & \mathbf{C} \\ \{switchOn, light\} & x \in \{7, \ldots, 21\} \\ \{switchOn, light, night, (x < 7)\} & x \in \{0, \ldots, 6\} \\ \{switchOn, light, night, (x \geq 22)\} & x \in \{22, 23\} \\ \{switchOff, light\} & x \in \{7, \ldots, 21\} \\ \{switchOff, night, sleep, (x < 7)\} & x \in \{0, \ldots, 6\} \\ \{switchOff, night, sleep, (x \geq 22)\} & x \in \{22, 23\} \end{array} \tag{2.10}$$

where $x \in \{m, \ldots, n\}$ means that either $x \mapsto m$, or $x \mapsto m + 1$, ..., or $x \mapsto n$.

Chapter 3

Constraint Answer Set Programming via Conflict Driven Constraint Learning

This chapter shows the relation of ASP to Boolean constraint solving and extends it to the paradigm of CASP. We demonstrate one way of handling CASP problems using state of the art techniques from SAT, ASP, and CP. In particular we show the following.

- In view of our focus on computational aspects, we deal with Boolean assignments and constraints. We give a corresponding characterization of a logic program and its stable models.
- We extend this characterization to logic programs with externals and give a definition of constraint stable models in terms of these.
- A description of a basic CDCL algorithm for logic programs with externals is given and exemplified.
- To solve constraint logic programs, an extension of the shown algorithm is provided that computes constraint stable models.
- Since the extended CDCL algorithm uses an external CP solver to handle the constraint part of the problem, the reason and conflict handling (a core part of CDCL) is reduced. Several techniques are presented to overcome this problem.
- After presenting distinguished implementation techniques of this approach, we evaluate the system wrt. previously developed approaches to solve constraint logic programs.

The definition of CASP, the system description of *clingcon* and part of its evaluation have been published in our paper [64]. The reduction methods presented in Section 3.5 and the corresponding evaluations are detailed in [102]. An elaborate description of ASP, the basic CDCL algorithm and its modifications were added. Finally, a

theorem establishing the relation between the constraint reduct and logic programs with externals is given and proved.

3.1 Boolean Constraint Solving

The basic idea of CDCL-based ASP solving is to map inferences from rules as in (2.1) to unit propagation on Boolean constraints. Our description of this approach follows the one given in [56].

Accordingly, we represent Boolean assignments, $\mathbf{B}$, over a set of atoms $\mathcal{A} \cup \mathcal{C}$ by sets of *signed literals* $\mathbf{T}a$ or $\mathbf{F}a$ standing for $a \mapsto \mathbf{T}$ and $a \mapsto \mathbf{F}$, respectively, where $a \in \mathcal{A} \cup \mathcal{C}$. The complement of a signed literal σ is denoted by $\overline{\sigma}$. We define $\mathbf{B}^{\mathbf{T}} = \{a \in \mathcal{A} \cup \mathcal{C} \mid \mathbf{T}a \in \mathbf{B}\}$ and $\mathbf{B}^{\mathbf{F}} = \{a \in \mathcal{A} \cup \mathcal{C} \mid \mathbf{F}a \in \mathbf{B}\}$. We can also project an assignment to a set of atoms $\mathbf{B}|_{\mathcal{A}} = \{\mathbf{T}a \mid a \in \mathbf{B}^{\mathbf{T}} \cap \mathcal{A}\} \cup \{\mathbf{F}a \mid a \in \mathbf{B}^{\mathbf{F}} \cap \mathcal{A}\}$. Then, an assignment $\mathbf{B}$ is *complete*, if $\mathbf{B}^{\mathbf{T}} \cap \mathbf{B}^{\mathbf{F}} = \emptyset$ and $\mathbf{B}^{\mathbf{T}} \cup \mathbf{B}^{\mathbf{F}} = \mathcal{A} \cup \mathcal{C}$. For instance, the assignment $\{\mathbf{T}\mathit{switchOn}, \mathbf{F}\mathit{switchOff}, \mathbf{F}\mathit{light}, \mathbf{T}\mathit{night}, \mathbf{F}\mathit{sleep}, \mathbf{F}(x < 7), \mathbf{T}(x \geq 22)\}$ is complete wrt. the atoms in Example 1.

Boolean constraints are represented as *nogoods*. A nogood is a set of signed literals representing an invalid partial assignment. A nogood δ is *violated* by a Boolean assignment $\mathbf{B}$ whenever $\delta \subseteq \mathbf{B}$. A complete Boolean assignment is a *solution* of a set of nogoods, if it violates none of them. Given a Boolean assignment $\mathbf{B}$ and a nogood δ such that $\delta \setminus \mathbf{B} = \{\sigma\}$ and $\overline{\sigma} \notin \mathbf{B}$, we say that δ is *unit* wrt. $\mathbf{B}$ and *asserts* the *unit-resulting literal* $\overline{\sigma}$. For a set Δ of nogoods and an assignment $\mathbf{B}$, *unit propagation* is the iterated process of extending $\mathbf{B}$ with unit-resulting literals until no further literal is unit-resulting for any nogood in Δ.

With these concepts in mind, the Boolean constraints induced by a normal logic program P can be captured as the nogoods of its completion and loop formulas. To define the completion, the equivalence of a body $\beta = \{p_1, \dots, p_m, \sim p_{m+1}, \dots, \sim p_n\} \in \mathit{body}(P)$ with all of its atoms is expressed in Equations 3.1 and 3.2.

$$\delta(\beta) = \{\mathbf{F}\beta, \mathbf{T}p_1, \dots, \mathbf{T}p_m, \mathbf{F}p_{m+1}, \dots, \mathbf{F}p_n\} \tag{3.1}$$

$$\Delta(\beta) = \{\{\mathbf{T}\beta, \mathbf{F}p_1\}, \dots, \{\mathbf{T}\beta, \mathbf{F}p_m\}, \{\mathbf{T}\beta, \mathbf{T}p_{m+1}\}, \dots, \{\mathbf{T}\beta, \mathbf{T}p_n\}\} \ . \tag{3.2}$$

An atom β is introduced for every body $\{p_1, \dots, p_m, \sim p_{m+1}, \dots, \sim p_n\} \in \mathit{body}(P)$. The corresponding literals $\mathbf{T}\beta$ and $\mathbf{F}\beta$ are called body literals.

Equations 3.3 and 3.4 handle the equivalence of all atoms $a \in \mathcal{A}$ with its supporting bodies $\mathit{body}_P(a) = \{\beta_1, \dots, \beta_k\}$.

$$\Delta_P(a) = \{\,\{\mathbf{F}a, \mathbf{T}\beta_1\}, \dots, \{\mathbf{F}a, \mathbf{T}\beta_k\}\,\} \tag{3.3}$$

$$\delta_P(a) = \{\mathbf{T}a, \mathbf{F}\beta_1, \dots, \mathbf{F}\beta_k\} \ . \tag{3.4}$$

The Boolean constraints induced by the completion of normal logic program P

over $\mathcal{A}$ can be captured as follows:

$$\Delta_P = \{\delta(\beta) \mid \beta \in body(P)\} \cup \{\delta \in \Delta(\beta) \mid \beta \in body(P)\} \quad (3.5)$$
$$\cup \{\delta_P(a) \mid a \in atom(P)\} \cup \{\delta \in \Delta_P(a) \mid a \in atom(P)\}$$

According to [83], exponentially many loop nogoods Λ_P may be needed to characterize the stable models of a logic program. We therefore define:

$$\Lambda_P = \bigcup_{\substack{U \subseteq atom(P),\\ EB_P(U)=\{B_1,\ldots,B_k\}}} \{\{\mathbf{T}a, \mathbf{F}B_1, \ldots, \mathbf{F}B_k\} \mid a \in U\} \quad (3.6)$$

where $EB_P(U) = \{body(r) \mid r \in P, head(r) \in U, body(r) \cap U = \emptyset\}$.

Definition 2: [56]

Let P be a normal logic program over a set of atoms $\mathcal{A}$ and let $\mathbf{B}$ be a (unique) solution of the set $\Delta_P \cup \Lambda_P$ of nogoods.
Then, a set of atoms $X = \mathbf{B}^{\mathbf{T}} \cap \mathcal{A}$ is a stable model of P.

In Example 1, the nogoods $\delta_{P_1}(switchOn)$ and $\Delta_{P_1}(switchOn)$ are

$$\{\mathbf{T}switchOn, \mathbf{F}\{\sim switchOff\}\} \text{ and } \{\mathbf{F}switchOn, \mathbf{T}\{\sim switchOff\}\}.$$

Similarly, the body $\{\sim switchOff\}$ of Rule (2.2) gives rise to nogoods

$$\delta(\{\sim switchOff\}) = \{\mathbf{F}\{\sim switchOff\}, \mathbf{F}switchOff\}.$$

and

$$\Delta(\{\sim switchOff\}) = \{\mathbf{T}\{\sim switchOff\}, \mathbf{T}switchOff\}$$

Hence, once an assignment contains $\mathbf{T}switchOn$, we may derive $\mathbf{F}switchOff$ via unit propagation (using both the first and the last nogood).

To extend this characterization to programs with constraint atoms, it is important to realize that the truth value of such atoms is determined external to the program. For instance, in our example, we would get from both $\delta_{P_1}(x < 7)$ and Λ_{P_1} the nogood $\{\mathbf{T}(x < 7)\}$, which would set $(x < 7)$ permanently to false. To address this issue, we handle $\mathcal{C}$ as external atoms.

Logic Programs With Externals The Boolean constraints induced by the completion of logic programs over regular atoms $\mathcal{A}$ and externals $\mathcal{C}$ as defined in [44] are:

$$\Delta_P^{\mathcal{C}} = \{\delta(\beta) \mid \beta \in body(P)\} \cup \{\delta \in \Delta(\beta) \mid \beta \in body(P)\} \quad (3.7)$$
$$\cup \{\delta_P(a) \mid a \in atom(P) \setminus \mathcal{C}\} \cup \{\delta \in \Delta_P(a) \mid a \in atom(P) \setminus \mathcal{C}\}. \quad (3.8)$$

Then, the loop nogoods of a logic program with externals are defined as

$$\Lambda_P^{\mathcal{C}} \;=\; \bigcup_{\substack{U \subseteq atom(P)\setminus\mathcal{C},\\ EB_P(U)=\{B_1,\dots,B_k\}}} \{\{\mathbf{T}a, \mathbf{F}B_1, \dots, \mathbf{F}B_k\} \mid a \in U\}\,. \tag{3.9}$$

These equations are obtained by replacing $atom(P)$ in the qualification of Equation (3.5) and Equation (3.6) with $atom(P) \setminus \mathcal{C}$.

Theorem 3.1.1

Let P be a constraint logic program over regular atoms $\mathcal{A}$ and constraint atoms $\mathcal{C}$ associated with the CSP $(\mathcal{V}, D, C)$. Furthermore, let $\mathbf{C} : V \to D$ be an assignment of integer variables.
Then, $(X, \mathbf{C})$ is a constraint stable model of P iff $X = \mathbf{B}^{\mathbf{T}} \cap (\mathcal{A} \cup \mathcal{C})$ for a (unique) solution $\mathbf{B}$ of the set $\Delta_P^{\mathcal{C}} \cup \Lambda_P^{\mathcal{C}} \cup \{\{\mathbf{F}c\} \mid \gamma(c) \in sat_{\mathbf{C}}(C)\} \cup \{\{\mathbf{T}c\} \mid \overline{\gamma(c)} \in sat_{\mathbf{C}}(C)\}$ of nogoods.

Accordingly, our example yields the constraint stable models in 2.10. For instance, the very first constraint stable model corresponds to the Boolean assignment $\{\mathbf{T}switchOn, \mathbf{T}light, \mathbf{F}switchOff, \mathbf{F}night, \mathbf{F}sleep, \mathbf{F}(x < 7), \mathbf{F}(x \geq 22)\}$ paired with the constraint variable assignment $\{x \mapsto 7\}$.

Proof 3.1.1 *Given Definition 1, $X \cap \mathcal{C} = \{c \mid \gamma(c) \in sat_{\mathbf{C}}(C)\}$ and $X \cap \mathcal{A}$ is the $\subseteq$-smallest model of the reduct $(P^{\mathbf{C}})^X$. According to Definition 2, $X \cap \mathcal{A}$ is a stable model of $P^{\mathbf{C}}$ and $X \cap \mathcal{A} = \mathbf{A}^{\mathbf{T}}$ for a (unique) solution $\mathbf{A}$ of $\Delta_{P^{\mathbf{C}}} \cup \Lambda_{P^{\mathbf{C}}}$. Given $\mathcal{A} \cap \mathcal{C} = \emptyset$ (by definition), we have that $\mathbf{A} \cup \{\mathbf{T}c \mid \gamma(c) \in sat_{\mathbf{C}}(C)\} \cup \{\mathbf{F}c \mid \overline{\gamma(c)} \in sat_{\mathbf{C}}(C)\}$ is a solution of $\Delta_{P^{\mathbf{C}}} \cup \Lambda_{P^{\mathbf{C}}} \cup \Theta_{\mathbf{C}}$ where $\Theta_{\mathbf{C}} = \{\{\mathbf{F}c\} \mid \gamma(c) \in sat_{\mathbf{C}}(C)\} \cup \{\{\mathbf{T}c\} \mid \overline{\gamma(c)} \in sat_{\mathbf{C}}(C)\}$. We now show that the set of solutions wrt. $\mathcal{A} \cup \mathcal{C}$ for $\Delta_{P^{\mathbf{C}}} \cup \Lambda_{P^{\mathbf{C}}} \cup \Theta_{\mathbf{C}}$ is equivalent to the set of solutions of $\Delta_P^{\mathcal{C}} \cup \Lambda_P^{\mathcal{C}} \cup \Theta_{\mathbf{C}}$.*

We can apply an assignment $\mathbf{B}$ to a set of nogoods Σ,

$$\Sigma \diamond \mathbf{B} = \{\delta \setminus \mathbf{B} \mid \delta \in \Sigma, \forall \sigma \in \mathbf{B}, \overline{\sigma} \notin \delta\}$$

by removing all occurrences of $\sigma \in \mathbf{B}$ from all of the nogoods and removing all nogoods that contain $\overline{\sigma}$. We can unit propagate a set of nogoods

$$unit(\Sigma) = \{\Sigma \diamond \mathbf{B} \mid \mathbf{B} = \{\overline{\sigma} \mid \{\sigma\} \in \Sigma\}\}$$

by applying $\Sigma \diamond \{\overline{\sigma}\}$ for all unit nogoods $\{\sigma\} \in \Sigma$. With $unit(\Sigma)^*$ we denote the fixpoint of this function.

To check whether the solutions of $\Delta_{P^{\mathbf{C}}} \cup \Lambda_{P^{\mathbf{C}}} \cup \Theta_{\mathbf{C}}$ and $\Delta_P^{\mathcal{C}} \cup \Lambda_P^{\mathcal{C}} \cup \Theta_{\mathbf{C}}$ are the same, we first compare the nogoods from $unit(\Delta_{P^{\mathbf{C}}})^*$ and $unit(\Delta_P^{\mathcal{C}} \cup \Theta_{\mathbf{C}})^*$. We now show this comparison for each part of the completion nogoods separately. Recall that $body_{\mathbf{C}}^{C}(P) = \{body(r) \mid r \in P, \gamma(body(r)^+) \cap \mathcal{C} \subseteq sat_{\mathbf{C}}(C), \gamma(body(r)^-) \cap sat_{\mathbf{C}}(C) = \emptyset\}$ is the set of bodies where all constraints are satisfied. We split the completion nogoods

into parts to show that

$$unit(\Delta_{P^{\mathbf{C}}})^* = unit(\Delta_P^{\mathcal{C}} \cup \Theta_{\mathbf{C}})^* \ .$$

For this, the following four equivalences have to hold:

I $unit(\{\delta(\beta|_{\mathcal{A}}) \mid \beta \in body^C_{\mathbf{C}}(P)\})^* = unit(\{\delta(\beta) \mid \beta \in body(P)\} \cup \Theta_{\mathbf{C}})^*$

II $unit(\bigcup_{\beta \in body^C_{\mathbf{C}}(P)} \Delta(\beta|_{\mathcal{A}}))^* = unit(\bigcup_{\beta \in body(P)} \Delta(\beta) \cup \Theta_{\mathbf{C}})^*$

III $unit(\{\delta_{P^{\mathbf{C}}}(a) \mid a \in atom(P) \setminus \mathcal{C}\})^* = unit(\{\delta_P(a) \mid a \in atom(P) \setminus \mathcal{C}\} \cup \Xi_{\mathbf{C}} \cup \Theta_{\mathbf{C}})^*$

IV $unit(\bigcup_{a \in atom(P) \setminus \mathcal{C}} \Delta_{P^{\mathbf{C}}}(a))^* = unit(\bigcup_{a \in atom(P) \setminus \mathcal{C}} \Delta_P(a) \cup \Xi_{\mathbf{C}} \cup \Theta_{\mathbf{C}})^*$

where $\Xi_{\mathbf{C}} = \{\{\mathbf{T}\beta\} \mid \beta \in body(P) \setminus body^C_{\mathbf{C}}(P)\}$ *is the set of nogoods that implies all bodies to be false that contain a constraint literal that is false wrt.* $\mathbf{C}$*. We let*

$$\beta = \{p_1, \ldots, p_m, {\sim}p_{m+1}, \ldots, {\sim}p_n, c_1, \ldots, c_k, {\sim}c_{k+1}, \ldots, {\sim}c_l\}$$

where $p_i \in \mathcal{A}, c_j \in \mathcal{C}, 1 \leq i \leq n, 1 \leq j \leq l$.

To prove I, for every $\beta \in body^C_{\mathbf{C}}(P)$*, we have that*

$$\delta(\beta|_{\mathcal{A}}) = \{\mathbf{F}\beta, \mathbf{T}p_1, \ldots, \mathbf{T}p_m, \mathbf{F}p_{m+1}, \ldots, \mathbf{F}p_n\}$$

while

$$\delta(\beta) = \{\mathbf{F}\beta, \mathbf{T}p_1, \ldots, \mathbf{T}p_m, \mathbf{F}p_{m+1}, \ldots, \mathbf{F}p_n, \mathbf{T}c_1, \ldots, \mathbf{T}c_k, \mathbf{F}c_{k+1}, \ldots, \mathbf{F}c_l\}$$

It is easy to see that $unit(\{\delta(\beta|_{\mathcal{A}})\})^* = unit(\{\delta(\beta)\} \cup \Theta_{\mathbf{C}})^*$ *as* $body^C_{\mathbf{C}}(P)$ *contains only bodies where all constraint literals are true wrt.* $\mathbf{C}$ *and* $unit()^*$ *removes all these literals from* $\delta(\beta)$ *as* $\Theta_{\mathbf{C}}$ *implies all constraint literals that are true wrt.* $\mathbf{C}$.

For all other bodies $\beta \in body(P) \setminus body^C_{\mathbf{C}}(P)$*, we have that* $P^{\mathbf{C}}$ *removes the whole rule and therefore does not require any nogood. On the other hand we have that* $\delta(\beta)$ *contains at least one constraint literal which is false wrt.* $\mathbf{C}$*. Therefore,* $unit(\{\delta(\beta)\} \cup \Theta_{\mathbf{C}})^* = \emptyset$*, as unit propagation removes all nogoods that contain a false literal wrt.* $\mathbf{C}$ *(as* $\Theta_{\mathbf{C}}$ *implies all constraint literals that are true wrt.* $\mathbf{C}$*).*

To prove II, for every $\beta \in body^C_{\mathbf{C}}(P)$*, we have that*

$$\Delta(\beta|_{\mathcal{A}}) = \{\{\mathbf{T}\beta, \mathbf{F}p_1\}, \ldots, \{\mathbf{T}\beta, \mathbf{F}p_m\}, \{\mathbf{T}\beta, \mathbf{T}p_{m+1}\}, \ldots, \{\mathbf{T}\beta, \mathbf{T}p_n\}\}$$

contains only atoms from $\mathcal{A}$ *while*

$$\Delta(\beta) = \Delta(\beta|_{\mathcal{A}}) \cup \{\{\mathbf{T}\beta, \mathbf{F}c_1\}, \ldots, \{\mathbf{T}\beta, \mathbf{F}c_k\}, \{\mathbf{T}\beta, \mathbf{T}c_{k+1}\}, \ldots, \{\mathbf{T}\beta, \mathbf{T}c_l\}\}$$

Due to $\Theta_{\mathbf{C}}$*, all literals* $\mathbf{F}c_1, \ldots, \mathbf{F}c_k, \mathbf{T}c_{k+1}, \ldots, \mathbf{T}c_l$ *are false wrt.* $\mathbf{C}$*, and these nogoods are removed via unit propagation leaving* $unit(\Delta(\beta|_{\mathcal{A}}))^* = unit(\Delta(\beta) \cup \Theta_{\mathbf{C}})^*$.

For all other bodies $\beta \in body(P) \setminus body^C_{\mathbf{C}}(P)$*, we have that* $P^{\mathbf{C}}$ *removes the whole rule and therefore does not require any nogood. Given* $\Delta(\beta)$ *as before, we*

know that at least one constraint literal of $\mathbf{F}c_1, \ldots, \mathbf{F}c_k, \mathbf{T}c_{k+1}, \ldots, \mathbf{T}c_l$ *is true wrt.* $\mathbf{C}$*, and unit propagation creates the nogood* $\{\mathbf{T}\beta\}$*. This again removes all nogoods containing* $\mathbf{T}\beta$*, resulting in* $unit(\Delta(\beta) \cup \Theta_{\mathbf{C}})^* = \emptyset$*. Furthermore, we know that* $\bigcup_{\beta \in body(P) \setminus body^{\mathcal{C}}_{\mathbf{C}}(P)} unit(\Delta(\beta) \cup \Theta_{\mathbf{C}})$ *creates the nogoods* $\Xi_{\mathbf{C}}$*, which are needed to prove III and IV.*

To prove III, we consider all atoms $a \in atom(P) \setminus \mathcal{C}$ *where* $body_P(a) = \{\beta_1, \ldots, \beta_i, \beta_{i+1}, \ldots, \beta_j\}$ *and* $\{\beta_{i+1}, \ldots, \beta_j\} = body(P) \setminus body^{C}_{\mathbf{C}}(P)$*. We know that*

$$\delta_{P^{\mathbf{C}}}(a) = \{\mathbf{T}a, \mathbf{F}\beta_1, \ldots, \mathbf{F}\beta_i\}$$

where $body_{P^{\mathbf{C}}}(a) = \{\beta_1, \ldots, \beta_i\} \subseteq body^{C}_{\mathbf{C}}(P)$ *and*

$$\delta_P(a) = \{\mathbf{T}a, \mathbf{F}\beta_1, \ldots, \mathbf{F}\beta_i, \mathbf{F}\beta_{i+1}, \ldots, \mathbf{F}\beta_j\}$$

Given that $\Xi_{\mathbf{C}}$ *implies these false bodies, we can easily see that* $unit(\{\delta_{P^{\mathbf{C}}}(a)\})^* = unit(\{\delta_P(a)\} \cup \Xi_{\mathbf{C}})^*$*.*

To prove IV, considering all atoms as before, We know that

$$\Delta_{P^{\mathbf{C}}}(a) = \{\, \{\mathbf{F}a, \mathbf{T}\beta_1\}, \ldots, \{\mathbf{F}a, \mathbf{T}\beta_i\}\, \}$$

and

$$\Delta_P(a) = \Delta_{P^{\mathbf{C}}}(a) \cup \{\{\mathbf{F}a, \mathbf{T}\beta_{i+1}\}, \ldots, \{\mathbf{F}a, \mathbf{T}\beta_j\}\}$$

Since all the extra bodies in $\Delta_P(a)$ *are false, the additional nogoods are removed by unit propagation leaving* $unit(\Delta_{P^{\mathbf{C}}}(a))^* = unit(\Delta_P(a) \cup \Xi_{\mathbf{C}})^*$*.*

As we have shown that all parts I to IV of the completion coincide, we conclude that $unit(\Delta_{P^{\mathbf{C}}})^* = unit(\Delta^{\mathcal{C}}_P \cup \Theta_{\mathbf{C}})^*$*.*

Given the false bodies in $body(P) \setminus body^{C}_{\mathbf{C}}(P)$ *it is easy to see that* $unit(\Lambda_{P^{\mathbf{C}}})^* = unit(\Lambda^{\mathcal{C}}_P \cup \Xi_{\mathbf{C}})^*$*.*

We have shown that the nogoods

$$unit(\Delta_{P^{\mathbf{C}}} \cup \Lambda_{P^{\mathbf{C}}} \cup \Theta_{\mathbf{C}})^* = unit(\Delta^{\mathcal{C}}_P \cup \Lambda^{\mathcal{C}}_P \cup \Theta_{\mathbf{C}})^*$$

and therefore conclude that $\Delta_{P^{\mathbf{C}}} \cup \Lambda_{P^{\mathbf{C}}} \cup \Theta_{\mathbf{C}}$ *has the same solutions wrt.* $\mathcal{A} \cup \mathcal{C}$ *as* $\Delta^{\mathcal{C}}_P \cup \Lambda^{\mathcal{C}}_P \cup \Theta_{\mathbf{C}}$*.* ■

3.2 Basic Conflict Driven Constraint Learning

For computing the stable models of a normal logic program with externals we use a Conflict Driven Constraint Learning (CDCL) [62] algorithm, which is effective for SAT [32, 88, 89, 123]. The CDCL algorithm is based on unit propagation of nogoods, conflict analysis, non-chronological backtracking and nogood learning. We shortly explain CDCL and its sub-functions in Algorithm 1. For this, we sometimes see an assignment $\mathbf{B}$ as a sequence of literals, following the order they have been added (assigned) to $\mathbf{B}$. With $\mathbf{B}[\sigma]$, we denote the subsequence of all literals that have been added before σ. We also introduce the notion of a decision level, where $dl(\sigma)$ denotes

Algorithm 1: CDCL

Input : A normal logic program P over $\mathcal{A}$ and externals $\mathcal{C}$.
Output : A stable model of P.

```
1  B ← A ∪ C ∪ {body(r) | r ∈ P}                    // set of atoms
2  B ← ∅                                              // (Boolean) assignment
3  ∇ ← ∅                                              // set of (dynamic) nogoods
4  dl ← 0                                             // decision level
5  loop
6      (B, ∇) ← Propagation(B, ∇)
7      Σ ← {δ | δ ∈ Δ_P^C ∪ ∇, δ ⊆ B}
8      if Σ ≠ ∅ then
9          if dl = 0 then return unsatisfiable
10         (δ', dl) ← ConflictAnalysis_P(∇, B, δ) for some δ ∈ Σ
11         ∇ ← ∇ ∪ {δ'}
12         B ← B \ {σ | σ ∈ B, dl(σ) > dl}
13     else if B^T ∪ B^F = B then
14         return (B^T ∩ (A ∪ C))
15     else
16         σ_d ← Select(B, B)
17         dl ← dl + 1
18         B ← B ∘ σ_d
```

the decision level on which this literal was assigned.

The algorithm starts with an empty assignment $\mathbf{B}$ and decision level 0. The goal is to gradually fill the assignment until it contains exactly one literal per atom. The positive literals then represent a solution to the normal logic program.

Our algorithm executes the following steps in order, until we have either found a solution or assured that no such solution exists. First, we propagate on the nogoods of the normal logic program in line 6 which is implemented in detail by Algorithm 2. After propagation reached a fixpoint, we check if we have any conflicting nogoods in lines 7 – 8. If we encounter a conflict on decision level 0 (line 9), the problem is *unsatisfiable* and the algorithm terminates. Every other conflict gets analyzed in line 10 and we backtrack to the decision level which is returned by ConflictAnalysis (Algorithm 3). In this case we also add the conflict nogood to the set of learned nogoods ∇. If we are not having a conflict, we either have a complete assignment in line 14 and return a stable model of P, or we select a new literal to be assigned/joined to the partial assignment based on some decision heuristic in line 16.

We now detail propagation in Algorithm 2. Propagation adds the set of literals that are implied by nogoods in lines 5 – 6 to the assignment $\mathbf{B}$. We propagate in a loop until we either encounter a conflict in line 2 or have reached a fixpoint in line 11. In the case of ASP, there are two ways of propagation. The first one is unit propagation on the nogoods in $\Delta_P^{\mathcal{C}}$ and ∇ (see lines 3 – 6). For every unit nogood, we add the

Algorithm 2: PROPAGATION

Global : A normal logic program P over $\mathcal{A}$ and externals $\mathcal{C}$.
Input : A Boolean assignment $\mathbf{B}$ and a set ∇ of nogoods.
Output : A (Boolean) assignment and a set of nogoods.

1 **loop**
2   **if** $\delta \subseteq \mathbf{B}$ for some $\delta \in \Delta_P^{\mathcal{C}} \cup \nabla$ **then return** $(\mathbf{B}, \nabla)$
3     $\Sigma \leftarrow \{\delta \mid \delta \in \Delta_P^{\mathcal{C}} \cup \nabla, \delta \setminus \mathbf{B} = \{\sigma\}, \overline{\sigma} \notin \mathbf{B}\}$
4   **if** $\Sigma \neq \emptyset$ **then**
5   **foreach** $\delta \in \Sigma$ such that $\delta \setminus \mathbf{B} = \{\sigma\}$ **do**
6             $\mathbf{B} \leftarrow \mathbf{B} \circ \overline{\sigma}$
7   **else**
8         $\Sigma \leftarrow$ UFSPROPAGATION$_P(\mathbf{B})$
9   **if** $\Sigma \neq \emptyset$ **then**
10             $\nabla \leftarrow \nabla \cup \Sigma$
11   **if** $\Sigma = \emptyset$ **then return** $(\mathbf{B}, \nabla)$

unit-resulting literal $\overline{\sigma}$ to the assignment $\mathbf{B}$. In this way, we capture all inferences from these sets of nogoods. If unit propagation has reached a fixpoint, we propagate on $\Lambda_P^{\mathcal{C}}$ in lines $8-10$. As this set of nogoods can be exponential in size [83] we are not representing it explicitly but use the dedicated algorithm UFSPROPAGATION which returns a set of unit and/or conflicting nogoods based on the current partial assignment. These nogoods are added to the learned nogoods ∇. If UFSPROPAGATION does not return any new nogoods, propagation has reached a fixpoint and the algorithm terminates.

Whenever we encounter a conflicting nogood in the CDCL algorithm, we need to analyse and simplify it such that it contains only one literal of the current decision level. The CONFLICTANALYSIS is detailed in Algorithm 3. Here, we resolve literals σ from a conflicting nogood δ until we only have one literal of the highest decision level left in this nogood. A literal σ is resolved in line 5 using a nogood ε which is asserting literal σ for an assignment $\mathbf{B}[\sigma]$. Combining those two nogoods $(\delta \setminus \{\sigma\}) \cup (\varepsilon \setminus \{\overline{\sigma}\})$ yields a new conflicting nogood without σ. Since ε was used to assert σ, all new literals added by the union of the two nogoods have a decision level which is smaller than or equal to that of σ. Finally, there is only one literal of the current decision level left and we return the new nogood and the second highest decision level. This is the level that is used for backjumping in the CDCL algorithm.

To improve the understanding of the whole CDCL algorithm we compute a solution of program P_1 from Example 1. Table 3.1 shows the additions to the assignment and the nogoods that are used for propagation. Note that the atoms $\mathcal{C} = \{(x < 7), (x \geq 22)\}$ are external atoms. We start with selecting $\mathbf{T}$*light* and $\mathbf{T}$*sleep* in line 16 of Algorithm 1 as there is nothing to propagate in line 6. Afterwards we resume with propagation. With $\mathbf{T}\{\mathit{switchOff}, \mathit{night}\}$, we denote the assignment of a body literal. It has to be true as it is the only body to derive *sleep*. The nogood

Algorithm 3: ConflictAnalysis

Global : A normal logic program P over $\mathcal{A}$ and externals $\mathcal{C}$.
Input : A set ∇ of nogoods, a (Boolean) assignment $\mathbf{B}$, and a conflicting nogood δ.
Output : A derived nogood and a decision level.

```
1 loop
2   foreach σ ∈ δ such that δ \ B[σ] = {σ} do
3     k ← max{dl(σ′) | σ′ ∈ δ \ {σ}}
4     if k = dl(σ) then
5       δ ← (δ \ {σ}) ∪ (ε \ {σ̄}) where ε ∈ Δ_P^C ∪ ∇, ε \ B[σ] = {σ̄}
6     else return (δ, k)
```

$\delta_P(sleep) = \{\mathbf{T}sleep, \mathbf{F}\{switchOff, night\}\}$ is used to assert this. Now, we propagate $\mathbf{T}night$ and $\mathbf{T}switchOff$ and their respective bodies $\mathbf{F}\{\sim night\}$ and $\mathbf{F}\{\sim switchOff\}$. As already shown before, given $\mathbf{T}switchOff$, we derive $\mathbf{T}switchOn$. This leads us to a conflict in line 7 of Algorithm 1. We use ConflictAnalysis in Algorithm 3 to analyze the conflicting nogood $\{\mathbf{T}light, \mathbf{F}\{switchOn\}, \mathbf{F}\{\sim night\}\}$. We replace the literal that was added last viz. $\mathbf{F}\{switchOn\}$ with $\mathbf{F}switchOn$ using the nogood $\{\mathbf{T}\{switchOn\}, \mathbf{F}switchOn\}$. The intermediate results of resolving this nogood are shown in Table 3.2. Here, δ shows the conflicting nogood, which is resolved with ε using the highlighted literals. The final conflict nogood is $\{\mathbf{T}light, \mathbf{T}\{switchOff, night\}\}$. This new nogood tells us that in no solution to the problem the light is on while it is night and the switch is off. As it contains only one literal of the current decision level 2, we undo the assignment and backtrack to level 1 in line 12 of Algorithm 1 and learn the nogood $\{\mathbf{T}light, \mathbf{T}\{switchOff, night\}\}$ by adding it to the set ∇. Now, we start the loop again and directly propagate $\mathbf{F}\{switchOff, night\}$ from the newly learned nogood. Since this is the only supporting body for the atom *sleep*, it must be false. As we cannot propagate any further literals, we add $\mathbf{T}night$ in line 16. By propagation, we derive that $\mathbf{T}switchOn$ as the light is on and it is night. After assigning all other atoms via unit propagation, we are left with two constraint atoms $(x < 7)$ and $(x \geq 22)$. We simply decide to make them both true (not shown in the table) using the select function. Having a complete assignment we can now return the solution $\mathbf{B}^{\mathbf{T}} \cap (\mathcal{A} \cup \mathcal{C}) = \{light, night, switchOn, (x < 7), (x \geq 22)\}$. Note that this solution is not a constraint stable model, as $x < 7$ and $x \geq 22$ are contradictory, but until now, no consistency check of the constraints is involved.

3.3 Architecture

For implementing CASP, we decided to draw upon experiences made in the area of SMT [97]. SMT enriches SAT with different theories like equalities and uninterpreted functions, bit-vector or floating point arithmetic, difference logic, or linear real/integer/mixed arithmetics. The goal is to find a solution to the SAT part of the problem

dl	$\mathbf{B}$	δ	*using*	Line
1	$\mathbf{T}light$			16
2	$\mathbf{T}sleep$			16
	$\mathbf{T}\{switchOff, night\}$	$\{\mathbf{T}sleep, \mathbf{F}\{switchOff, night\}\}$	$\delta_P(sleep)$	6
	$\mathbf{T}night$	$\{\mathbf{T}\{switchOff, night\}, \mathbf{F}night\}$	$\Delta(\{switchOff, night\})$	6
	$\mathbf{F}\{\sim night\}$	$\{\mathbf{T}\{\sim night\}, \mathbf{T}night\}$	$\Delta(\{\sim night\})$	6
	$\mathbf{T}switchOff$	$\{\mathbf{T}\{switchOff, night\}, \mathbf{F}switchOff\}$	$\Delta(\{switchOff, night\})$	6
	$\mathbf{F}\{\sim switchOff\}$	$\{\mathbf{T}\{\sim switchOff\}, \mathbf{T}switchOff\}$	$\Delta(\{\sim switchOff\})$	6
	$\mathbf{F}switchOn$	$\{\mathbf{T}switchOn, \mathbf{F}\{\sim switchOff\}\}$	$\delta_P(switchOn)$	6
	$\mathbf{T}\{\sim switchOn\}$	$\{\mathbf{F}\{\sim switchOn\}, \mathbf{F}switchOn\}$	$\delta(\{\sim switchOn\})$	6
	$\mathbf{F}\{switchOn\}$	$\{\mathbf{T}\{switchOn\}, \mathbf{F}switchOn\}$	$\Delta(\{switchOn\})$	6
		$\{\mathbf{T}light, \mathbf{F}\{switchOn\}, \mathbf{F}\{\sim night\}\}$		7
		$\{\mathbf{T}light, \mathbf{T}\{switchOff, night\}\}$	$dl = 1$	10
1	$\mathbf{F}\{switchOff, night\}$	$\{\mathbf{T}light, \mathbf{T}\{switchOff, night\}\}$	∇	6
	$\mathbf{F}sleep$	$\{\mathbf{T}sleep, \mathbf{F}\{switchOff, night\}\}$	$\delta_P(sleep)$	6
2	$\mathbf{T}night$			16
	$\mathbf{F}\{\sim night\}$	$\{\mathbf{T}\{\sim night\}, \mathbf{T}night\}$	$\Delta(\{\sim night\})$	6
	$\mathbf{T}\{switchOn\}$	$\{\mathbf{T}light, \mathbf{F}\{switchOn\}, \mathbf{F}\{\sim night\}\}$	$\delta_P(light)$	6
		...		

Table 3.1: Computing the stable model $\{light, night, switchOn, x < 7, x \geq 22\}$ with CDCL(P_1).

	δ	ε
0	$\{\mathbf{T}light, \mathbf{F}\{switchOn\}, \mathbf{F}\{\sim night\}\}$	$\{\mathbf{T}\{switchOn\}, \mathbf{F}switchOn\}$
1	$\{\mathbf{T}light, \mathbf{F}switchOn, \mathbf{F}\{\sim night\}\}$	$\{\mathbf{T}switchOn, \mathbf{F}\{\sim switchOff\}\}$
2	$\{\mathbf{T}light, \mathbf{F}\{\sim switchOff\}, \mathbf{F}\{\sim night\}\}$	$\{\mathbf{T}\{\sim switchOff\}, \mathbf{T}switchOff\}$
3	$\{\mathbf{T}light, \mathbf{T}switchOff, \mathbf{F}\{\sim night\}\}$	$\{\mathbf{T}\{switchOff, night\}, \mathbf{F}switchOff\}$
4	$\{\mathbf{T}light, \mathbf{T}\{switchOff, night\}, \mathbf{F}\{\sim night\}\}$	$\{\mathbf{T}\{\sim night\}, \mathbf{T}night\}$
5	$\{\mathbf{T}light, \mathbf{T}\{switchOff, night\}, \mathbf{T}night\}$	$\{\mathbf{T}\{switchOff, night\}, \mathbf{F}night\}$
6	$\{\mathbf{T}light, \mathbf{T}\{switchOff, night\}\}$	

Table 3.2: Conflict analysis of $\{\mathbf{T}light, \mathbf{F}\{switchOn\}, \mathbf{F}\{\sim night\}\}$.

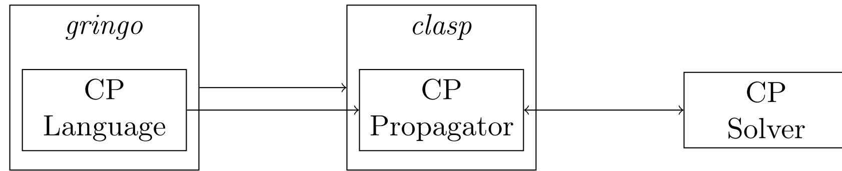

Figure 3-1: Architecture of *clingcon* 1 + 2.

that is consistent with the theory. Various techniques [97] have been developed:

The Lazy Approach abstracts from the constraints in a specialized theory. Every constraint is represented by a Boolean atom. This allows for a compact handling of the theory and dedicated theory algorithms. The SAT solver takes care of a valid assignment to the Boolean variables while dedicated algorithms use this assignment to check consistency with respect to the theory. The drawbacks of this approach are that the collaboration between the SAT and the theory solver is not easy, as modern CDCL algorithms usually need reason and conflict nogoods.

Partial Assignment Checks are used to check consistency of an incomplete assignment with respect to the theory. They find inconsistent assignments much earlier during the search but come at the cost of additional theory checks. It is possible to use incomplete consistency checks which can detect that a partial assignment is inconsistent. These algorithms are usually very fast but cannot detect inconsistency in all cases. For linear constraints, bound propagation can be such a test.

Theory Propagation does not only check consistency with the theory but also allows to infer knowledge from the theory given a partial assignment. With this knowledge, dedicated theory algorithms can extend the Boolean assignment by literals directly inferred from the theory.

Incremental Propagation allows the theory solver to have an internal state. As in CDCL, the assignment gets extended step by step, new constraints are added that must be checked. Some solvers can take advantage of the fact that a subset of the constraints has already been checked. So propagation does not need to redo all of its work and just needs to incorporate the new constraints. This can increase the performance of the theory propagation drastically.

The Eager Approach translates theory atoms into Boolean propositions. For some theories, there exist efficient translations. This approach makes full use of the learning capabilities of a SAT solver. One drawback is that the encoding into clauses or nogoods can be too big for certain theories to be solvable. Also, some translation may produce fewer clauses while sacrificing the propagation strength of unit propagation.

We decided on a lazy approach for our first system named *clingcon* 1.[1] We therefore use a CP solver to check for consistency of our assignment by abstracting from the integer arithmetic constraints [97]. To profit from the development of different CP solvers, we focus on an approach taking an unmodified, black-box CP solver. Hence we have no insight into the CP system. The award winning solver *gecode*[2] is used in our system. We also developed our algorithms in a way that it is easy to exchange the CP solver. The architecture of our system is shown in Figure 3-1. We extended the input language of the ASP grounder *gringo* by a language for constraints over integers and changed the ASP solver *clasp* to contain an additional CP propagator. This propagator then calls the black box system *gecode* for CP propagation. While the ASP solver uses an extended CDCL algorithm, the CP solver checks the constraints implied by the partial assignment for consistency. Incremental theory propagation is also applied, which means that not only the consistency is checked, but also if new knowledge can be derived and extend the Boolean assignment. We now show how we extended the CDCL algorithm to handle constraint logic programs.

3.4 Extended Conflict Driven Constraint Learning

Given a constraint logic program P over $\mathcal{A}$ and externals $\mathcal{C}$ associated with the CSP $(\mathcal{V}, D, C)$, we want to compute a constraint stable model $(X, \mathbf{C})$. Following Theorem 3.1.1, we develop a modified CDCL algorithm to find a solution $\mathbf{B}$ to

$$\Delta_P^{\mathcal{C}} \cup \Lambda_P^{\mathcal{C}} \cup \{\{\mathbf{F}c\} \mid \gamma(c) \in sat_{\mathbf{C}}(C)\} \cup \{\{\mathbf{T}c\} \mid \overline{\gamma(c)} \in sat_{\mathbf{C}}(C)\}$$

such that $X = \mathbf{B^T} \cap (\mathcal{A} \cup \mathcal{C})$ and $X \cap \mathcal{C} = \{c \mid \gamma(c) \in sat_{\mathbf{C}}(C)\}$. We extend γ to signed literals over $\mathcal{C}$ as follows:

$$\gamma(\sigma) = \begin{cases} \overline{\gamma(a)} & \text{if } \sigma = \mathbf{F}a, a \in \mathcal{C} \\ \gamma(a) & \text{if } \sigma = \mathbf{T}a, a \in \mathcal{C} \end{cases}$$

For instance, we have $\gamma(\mathbf{F}(x < 7)) = x \geq 7$. When applied to a set of literals, γ refers to its set of constraints.

We use CDCL Algorithm 1 and extend it by a CP oracle. The input of our new Algorithm 4 is a constraint logic program. As all differences to Algorithm 1, this is highlighted. Several major changes have been done. First of all, the PROPAGATION function has been extended to also accept a CSP and there is a consistency check added if we encounter a complete Boolean assignment. If we have such a complete Boolean assignment (line 13) we are not sure whether there exists an assignment $\mathbf{C}$ such that all constraints $\gamma(\mathbf{B}|_{\mathcal{C}})$ are satisfied. Therefore, we use the function LABELING$_{(\mathcal{V},D,C)}(\mathbf{B}|_{\mathcal{C}})$ in line 14 which returns either an assignment $\mathbf{C}$ such that $sat_{\mathbf{C}}(C) = \gamma(\mathbf{B}|_{\mathcal{C}})$, or *conflicting*. In the first case, we simply return a constraint stable model. In the latter, we take the complete assignment $\mathbf{B}|_{\mathcal{C}}$ projected on the constraint atoms as a conflict

[1] Its successor *clingcon* 2, using the same architecture, is described in Section 3.5.
[2] `http://www.gecode.org`

Algorithm 4: CDCL-CASP

Input : A constraint logic program P over $\mathcal{A}$ and externals $\mathcal{C}$ associated with $(\mathcal{V}, D, C)$.

Output : A constraint stable model of P.

```
1  𝓑 ← 𝓐 ∪ 𝓒 ∪ {body(r) | r ∈ P}                       // set of atoms
2  B ← ∅                                                 // (Boolean) assignment
3  ∇ ← ∅                                                 // set of (dynamic) nogoods
4  dl ← 0                                                // decision level
5  loop
6     (B, ∇) ← Propagation(B, ∇)
7     Σ ← {δ | δ ∈ Δ_P^𝓒 ∪ ∇, δ ⊆ B}
8     if Σ ≠ ∅ then
9        if dl = 0 then return unsatisfiable
10       (δ', dl) ← ConflictAnalysis_P(∇, B, δ) for some δ ∈ Σ
11       ∇ ← ∇ ∪ {δ'}
12       B ← B \ {σ | σ ∈ B, dl(σ) > dl}
13    else if B^T ∪ B^F = 𝓑 then
14       C ← Labeling_(𝓥,D,C)(B|_𝓒)
15       if C = conflicting then
16          (δ, dl) ← ConflictAnalysis_P(∇, B, B|_𝓒)
17          ∇ ← ∇ ∪ {δ}
18          B ← B \ {σ | σ ∈ B, dl(σ) > dl}
19       else return (B^T ∩ (𝓐 ∪ 𝓒), C)
20    else
21       σ_d ← Select(𝓑, B)
22       dl ← dl + 1
23       B ← B ∘ σ_d
```

and resolve it (lines 15 – 18) as before.[3]

Also, the propagation Algorithm 5 is extended by CSP propagation. Whenever no new knowledge can be derived by unit propagation or $\textsc{UfsPropagation}_P(\mathcal{C}, \mathbf{B})$, we call $\textsc{CspPropagation}(\mathbf{B}|_{\mathcal{C}})$ to invoke an external CP solver for propagation.[4] Given a set of literals $\mathbf{B}$, it either returns a set of newly derived literals $\mathbf{B}'$ or *conflicting*. All constraints $\gamma(\mathbf{B}')$ are inferred by the external CP solver from $\gamma(\mathbf{B}|_{\mathcal{C}})$. If it returns *conflicting*, we return and add the currently assigned constraint literals $\mathbf{B}|_{\mathcal{C}}$ as a conflicting nogood (line 13). One major difficulty is that the CDCL Algorithm requires conflicts and reasons which cannot be obtained easily from a black box system. We

[3] Note that in the case that we have found a solution, the labeling function always returns the best solution in terms of the CSP optimization. The currently best value for a solution is stored internally and a global optimal solution is guaranteed this way.

[4] Of course this can be improved by only doing constraint propagation if there is a change in the assignment wrt. the constraint atoms $\mathcal{C}$.

Algorithm 5: PROPAGATION

Global : A constraint logic program P over $\mathcal{A}$ and externals $\mathcal{C}$ associated with $(\mathcal{V}, D, C)$.
Input : A Boolean assignment $\mathbf{B}$ and a set ∇ of nogoods.
Output : A (Boolean) assignment and a set of nogoods.

```
1  loop
2      if δ ⊆ B for some δ ∈ Δ_P^C ∪ ∇ then return (B, ∇)
3      Σ ← {δ | δ ∈ Δ_P^C ∪ ∇, δ \ B = {σ}, σ̄ ∉ B}
4      if Σ ≠ ∅ then
5          foreach δ ∈ Σ such that δ \ B = {σ} do
6              B ← B ∘ σ̄
7      else
8          Σ ← UfsPropagation_P(C, B)
9          if Σ ≠ ∅ then
10             ∇ ← ∇ ∪ Σ
11         else
12             B′ ← CspPropagation(B|_C)
13             if B′ = conflicting then return (B, ∇ ∪ Reduce_(V,D,C)(B|_C))
14             foreach σ ∈ B′ do ∇ ← ∇ ∪ Reduce_(V,D,C)(B|_C ∪ {σ̄})
15             if B′ = ∅ then return (B, ∇)
```

therefore incorporated the REDUCE function which we present in the next section. For the moment, let us assume it to be the identity function. In the case that we deduce a set of literals $\mathbf{B}'$ in line 14, we add a unit nogood $\mathbf{B}|_{\mathcal{C}} \cup \{\overline{\sigma}\}$ for each $\sigma \in \mathbf{B}'$. We reduce the size of this nogood with the REDUCE function as explained below. Note that propagation always favors unit propagation before UFSPROPAGATION before CSPPROPAGATION. So we only call the costly CSP propagation once we have reached a fixpoint with the other (cheaper) propagation.

To improve the understanding of the changes to the basic CDCL algorithm, we compute a constraint stable model for Example 1. We start Algorithm 4 with an empty assignment $\mathbf{B}$ and program P_1. We reuse the updates to the assignment and the nogoods that are used in Table 3.1 exactly as before. We can do this because we have not yet decided a truth value for a constraint atom and therefore CSPPROPAGATION always returns the empty set. The only atoms left are $(x < 7)$ and $(x \geq 22)$, so we select $\mathbf{T}(x < 7)$ in line 21 of Algorithm 4. Given this, we have that $\mathbf{B}|_{\mathcal{C}} = \{\mathbf{T}(x < 7)\}$ and therefore CSPPROPAGATION$(\{\mathbf{T}(x < 7)\}) = \{\mathbf{F}(x \geq 22)\}$, as $x < 7$ implies $x < 21$. Since this is not conflicting, we extend ∇ by $\{\mathbf{T}(x < 7), \mathbf{T}(x \geq 22)\}$ in line 14 of Algorithm 5. Now, unit propagation extends the assignment by $\mathbf{F}(x \geq 22)$ in lines 3–6 reaching a fixpoint. We have a complete assignment and label the variables in $\mathcal{V}$ in line 14 of Algorithm 4. One possible labeling for the variables in $\mathcal{V}$ is $\mathbf{C} = \{x \mapsto 5\}$. This assignment is consistent with the constraints in $sat_{\mathbf{C}}(C) = \{x < 7, x < 21\}$. Therefore, we return the constraint stable model

Algorithm 6: DELETION FILTERING

Global : A CSP $(\mathcal{V}, D, C)$.
Input : A list of literals $\mathbf{B} = [\sigma_1, \ldots, \sigma_n]$.
Output : A (shortened) list of literals.

```
1 i ← 1
2 while i ≤ |B|
3     if CspPropagation([σ_1, ..., σ_{i−1}, σ_{i+1}, ..., σ_{|B|}]) = conflicting then
4         B ← [σ_1, ..., σ_{i−1}, σ_{i+1}, ..., σ_{|B|}]
5     else
6         i ← i + 1
7 return B
```

$(\{light, night, switchOn, (x < 7)\}, \{x \mapsto 5\})$.

3.5 Reason and Conflict Reduction

A crucial part of a CDCL system are short reasons and conflicts. Having the partial assignment as reason/conflict hinders propagation, as not every part of it is actually needed. We now show how to use a technique to produce a so called Irreducible Inconsistent Set (IIS;[30, 68, 77, 120]). For this, an algorithm similar to Algorithm 6 is proposed in [30]. It reduces a set of conflicting constraints to a conflicting subset minimal set. The basic idea is to remove one constraint after the other and check if the set is still conflicting. If this is the case, we can permanently remove this constraint and continue. If not, this constraint is a crucial part of the conflict and stays in the set.

To show the effects of the reduction method in Algorithm 6, we use an extended set of constraint atoms over variables $\mathcal{V} = \{x, y, z, w\}$ and domains $D = \{D(v) = \{0, \ldots, 8\} \mid v \in \mathcal{V}\}$. Reconsider the execution of the extended CDCL algorithm in Section 3.4 and suppose that we found the assignment

$$\mathbf{B} = \{\mathbf{T}(x + y > 8), \mathbf{T}(z = 0), \mathbf{T}(w = 0), \mathbf{T}(y = x),\\ \mathbf{F}(x + z > 8), \mathbf{F}(x + w > 8), \mathbf{T}(y - x = 1)\}$$

which is conflicting wrt. CspPropagation$(\mathbf{B}|_{\mathcal{C}})$. We want to reduce it, as it contains a lot of unrelated information. In the first step, we remove $\mathbf{T}(x + y > 8)$ from the assignment, but the induced CSP is still conflicting. Hence, that we can safely remove it. The same works for the next two literals $\mathbf{T}(z = 0)$ and $\mathbf{T}(w = 0)$. Then, we try to remove $\mathbf{T}(y = x)$ from the assignment in line 3 but detect that the induced CSP is no longer conflicting, so this constraint is necessary for the set of constraints to be conflicting. We continue by removing $\mathbf{F}(x + z > 8)$ and $\mathbf{F}(x + w > 8)$ while the CSP is still conflicting. Finally, we try to remove $\mathbf{T}(y - x = 1)$ but find out that this constraint is necessary for the conflict. We are left with a reduced conflict clause

Algorithm 7: FORWARD FILTERING

Global : A CSP $(\mathcal{V}, D, C)$.
Input : A list of literals $\mathbf{B} = [\sigma_1, \ldots, \sigma_n]$.
Output : A (shortened) list of literals.

```
1 B′ ← []
2 while CspPropagation(B′) ≠ conflicting
3     L ← B′
4     i ← 1
5     while CspPropagation(L) ≠ conflicting
6         L ← L ∘ σ_i
7         i ← i + 1
8     B′ ← B′ ∘ σ_i
9 return B′
```

$\{\mathbf{T}(y = x), \mathbf{T}(y - x = 1)\}$ which is the real cause of the conflict and a smaller nogood.

Precise conflicts and reasons are a crucial part of every CDCL algorithm. Therefore, we present several methods to improve the trivial conflicts we get from the blackbox CP solver approach.

Forward Filtering With *gecode*, we decided for a modern CP solver that supports incremental propagation. In fact, propagation is done in a constraint space where the restricted domains of the variables are stored after propagation. Usually, propagation cannot be undone, but adding a constraint to a solver is simple, as it constrains the variables even more. Removing a constraint (as it is shown in Algorithm 6) is usually not possible. So propagation has to start from scratch with a (reduced) set of constraints every time. Therefore, we use the forward filtering Algorithm 7. It tries to avoid resetting the constraint space and incrementally adds constraints to a testing constraint space in lines 5−7. If the testing constraint space becomes *conflicting*, we add the last literal to the result, as this is a crucial part of the inconsistency. We redo this until the result becomes *conflicting*. This algorithm is designed to achieve a better performance with a standard incremental CP solver. Consider again our example, $\mathbf{B} = \{\mathbf{T}(x + y > 8), \mathbf{T}(z = 0), \mathbf{T}(w = 0), \mathbf{T}(y = x), \mathbf{F}(x + z > 8), \mathbf{F}(x + w > 8), \mathbf{T}(y - x = 1)\}$. We add $\mathbf{T}(x + y > 8)$ to the testing list $\mathbf{L}$. This constraint alone is not conflicting, so we keep adding constraints until $\mathbf{L} = \mathbf{B}$ in lines 5−7. Now, we know that the last constraint $\mathbf{T}(y - x = 1)$ is indispensable for the inconsistency. Restarting the loop, with $\mathbf{L} = [\mathbf{T}(y - x = 1)]$ in line 3, we can add $\mathbf{T}(x + y > 8)$, $\mathbf{T}(z = 0)$, $\mathbf{T}(w = 0)$, and $\mathbf{T}(y = x)$ until $\mathbf{L}$ becomes conflicting again. Now, since we have that $\mathbf{B}' = [\mathbf{T}(y - x = 1), \mathbf{T}(y = x)]$ our algorithm detects a conflict in line 2 and returns this reduced set. We found exactly the same irreducible inconsistent set as with the deletion filtering Algorithm 6 but did most of the constraint propagation incrementally.

Algorithm 8: RANGE FILTERING

Global : A CSP $(\mathcal{V}, D, C)$.
Input : A list of literals $\mathbf{B} = [\sigma_1, \ldots, \sigma_n]$.
Output : A (shortened) list of literals.

```
1 B' ← []
2 i ← n
3 while CspPropagation(B') ≠ conflicting
4 |   B' ← B' ∘ σ_i
5 |   i ← i − 1
6 return B'
```

Backward Filtering The basic idea of this algorithm is the same as in Algorithm 7. With backward filtering, we reverse the order of the inconsistent constraint list. Therefore, we first test the last assigned constraint and iterate to the first. In this way, we accommodate the fact that one of the literals that was decided on the current decision level has to be included in the conflicting nogood. Otherwise, we would have recognized the conflict before.

Range Filtering Still trying to reduce the amount of propagation that we need, Algorithm 8 does not aim at computing an irreducible list of constraints, but tries to approximate a smaller one to find a tradeoff between reduction of size and runtime of the algorithm. Therefore, as shown in Algorithm 8, we move through the reversed list of constraints $\mathbf{B}$ and add constraints to the result $\mathbf{B}'$ until it becomes inconsistent. In our example, we reduce the inconsistent list by the elements $\mathbf{T}(x + y > 8)$, $\mathbf{T}(z = 0)$, and $\mathbf{T}(w = 0)$, as this is the first part of the list of constraints that is unnecessary for the conflict.

Connected Component Filtering With Algorithm 9, we make use of the structure of the constraints. It only adds constraints that share some of their variables and starts with the last constraint from the list. While $\mathcal{V}'$ contains all variables that occur in the testing list, the function VARS(σ) returns all variables of the constraint $\gamma(c)$ where $\sigma = \mathbf{T}c$ or $\mathbf{F}c$. We start the loop by remembering how many constraint variables we have seen so far (line 6). Now, we iterate over all constraints that contain some of the already inspected variables in $\mathcal{V}'$ (lines 8−9). We add them to the test list $\mathbf{L}$ and also extend $\mathcal{V}'$. As in Algorithm 7, once $\mathbf{L}$ becomes conflicting, the last constraint is added to the result $\mathbf{B}'$. If $\mathbf{B}'$ is already conflicting, we are done and have a minimal list of constraints. Otherwise, we restart the loop, restricting the set of variables $\mathcal{V}'$ to the variables in $\mathbf{B}'$. Furthermore, we only iterate over the constraints from the test list $\mathbf{B}'$, as this list is already conflicting. If we cannot find a conflicting list of constraints (this can be the case if the last constraint of the input list $\mathbf{B}$ is not contained in the minimal list of constraints), we add all variables of all constraints in $\mathbf{B}$ to $\mathcal{V}'$. This algorithm takes account of the internal structure of the constraints. In

Algorithm 9: CONNECTED COMPONENT FILTERING

Global : A CSP $(\mathcal{V}, D, C)$.
Input : A list of literals $\mathbf{B} = [\sigma_1, \ldots, \sigma_n]$.
Output : A (shortened) list of literals.

```
1  if size(B) = 0 then return ∅
2  B' ← []
3  L ← []
4  V' ← VARS(σ_n)
5  while CspPropagation(B') ≠ conflicting
6      count ← |V'|
7      i ← size(B)
8      while CspPropagation(L) ≠ conflicting and i ≥ 0
9          if V' ∩ VARS(σ_i) ≠ ∅ then
10             L ← L ∘ σ_i
11             V' ← V' ∪ VARS(σ_i)
12         i ← i − 1
13     if CspPropagation(L) = conflicting then
14         B' ← B' ∘ σ_i
15         V' ← {v | v ∈ VARS(σ), σ ∈ B'}
16         B ← REMOVE(L, σ_i)
17         L ← B'
18     if count = |V'| then V' ← {v | v ∈ VARS(σ), σ ∈ B}
19 return B'
```

our example, the last constraint is $\mathbf{T}(y - x = 1)$, so $\mathcal{V}' = \{y, x\}$ and we completely ignore $\mathbf{T}(z = 0)$ and $\mathbf{T}(w = 0)$.

Connected Component Range Filtering This algorithm is a combination of connected component filtering and range filtering. We simply stop Algorithm 9 at line 13 and return $\mathbf{L}$. As with range filtering, this algorithm does not return a minimal, but smaller list of constraints.

Reducing Reasons To use the REDUCE function also for minimizing reasons, the function CSPPROPAGATION($\mathbf{B}$) needs to be deterministic, monotone, and reaching a fixpoint on the assignment $\mathbf{B}$ of constraint literals. For all CSPs $(\mathcal{V}, D, C)$ and assignments $\mathbf{B}$, $\mathbf{B}' =$ CSPPROPAGATION($\mathbf{B}$). The function reaches a fixpoint, if $\emptyset =$ CSPPROPAGATION($\mathbf{B} \cup \mathbf{B}'$) or $\mathbf{B}' = \mathit{conflicting}$. It is monotone, if for every assignment $\mathbf{D} \supseteq \mathbf{B}$, $\mathbf{D}' =$ CSPPROPAGATION($\mathbf{D}$), either $\mathbf{D}' = \mathit{conflicting}$ or $\mathbf{D}' \supseteq \mathbf{B}'$.

Given a partial assignment $\mathbf{B}$, CSPPROPAGATION($\mathbf{B}$) returns a set $\mathbf{B}'$. Since the function is monotone, no subset of $\mathbf{B}$ is conflicting. As the call on $\mathbf{B}$ returns $\mathbf{B}'$, and it computes a fixpoint, any call to $\mathbf{B} \cup \{\sigma\}$ where $\sigma \in \mathbf{B}'$ is also not conflicting. Hence CSPPROPAGATION($\mathbf{B} \cup \{\overline{\sigma}\}$) is conflicting, as the function is monotone. The REDUCE

function can shrink a conflicting set, so $\text{Reduce}_{(\mathcal{V},D,C)}(\mathbf{B} \cup \{\overline{\sigma}\})$ does not remove $\overline{\sigma}$ from the set (as it is no longer conflicting), and returns a (minimal) nogood asserting σ.

In fact, the Reduce function can be used to shrink unit nogoods as it can be seen in Algorithm 5 line 14. Consider the following propagation

$$\{\mathbf{F}(y-x=1)\} = \text{CspPropagation}(\{\mathbf{T}(x+y>8), \mathbf{T}(z=0), \mathbf{T}(w=0), \mathbf{T}(y=x)\})$$

To reduce the unit nogood, we call $\text{Reduce}_{(\mathcal{V},D,C)}(\{\mathbf{T}(x+y>8), \mathbf{T}(z=0), \mathbf{T}(w=0), \mathbf{T}(y=x)\} \cup \{\mathbf{T}(y-x=1)\})$ on line 14 in Algorithm 5. The filtering Algorithm 6 reduces it to $\{\mathbf{T}(y=x), \mathbf{T}(y-x=1)\}$, a nogood that is unit and minimal wrt. the original assignment.

3.6 Implementation Techniques

After developing an algorithm to compute constraint stable models, we now present different techniques that are implemented in *clingcon* 1 and 2. These techniques either improve the performance of our approach or add new features relevant for CASP solving. Finally, we evaluate the presented techniques on standard CASP problems, comparing our system to previous approaches tackling CASP.

3.6.1 Lazy Reasons

All algorithms to reduce reasons that we proposed so far are developed with the idea in mind to avoid the function CspPropagation, as it can be costly and is called quite often. A different approach to avoid calling our CP solver is to reduce and store the asserting nogoods only lazily. Hence, instead of adding a (reduced) nogood to the learned nogoods ∇, every time that CspPropagation returns a set of literals $\mathbf{B}'$ in Algorithm 5, we simply store the *lazy nogoods* $\{\{\overline{\sigma}, \epsilon\} \mid \sigma \in \mathbf{B}'\}$ where ϵ denotes an internal data structure that stores a reference to the last literal σ' in the assignment $\mathbf{B}$. Unit propagation can still add $\sigma \in \mathbf{B}'$ to the assignment, but at this point in computation there is no need to know the rest of the nogood. Actually, we only need the complete nogood whenever σ occurs in a conflict. Conflict analysis in Algorithm 3 uses the actual nogood to resolve σ. As only a lazy nogood was stored, we have to recreate that nogood. Given that we have stored σ' in the internal data structure ϵ, we can simply replace ϵ with $\mathbf{B}[\sigma']$. The restored nogood is $\{\overline{\sigma}\} \cup \mathbf{B}[\sigma']$, as originally intended. Now, it is also the time to call one of the costly reduction functions to reduce this nogood to a smaller one. The advantages of this method are that:

- we only use a reduction filter on the nogood if it is actually used in conflict analysis and
- we are not adding the nogoods to ∇ such that unit propagation is slowed down by the amount of nogoods that we add.

Algorithm 10: CSP LOOKAHEAD

Input : A CSP $(\mathcal{V}, D, C)$ and a set of constraint atoms $\mathcal{C}$.
Output : A set of nogoods Σ.

```
1 Σ ← ∅
2 let σ ∈ {Tc, Fc | c ∈ C} in
3     δ ← CspPropagation({σ})
4     if δ = conflicting then
5         Σ ← Σ ∪ {σ}
6     else
7         foreach σ' ∈ δ do Σ ← Σ ∪ {{σ, σ̄'}}
8 return Σ
```

The last point in this can also be a drawback as the nogood is not added for further propagation which means that CSPPROPAGATION has to derive the same literals again during the search and no shortcut via a nogood is created.

3.6.2 Global Constraints

Global constraints are known to speed up the computation in traditional CP solvers. Within our framework, it is possible to use any global constraint that is supported by the blackbox CP solver. Nevertheless, global constraints usually cannot be reified in such solvers and therefore, are only allowed as facts in the input language of *clingcon* 1 and 2. Currently, two global constraints are implemented in these systems. The global distinct, where a set of terms must be different from each other. And the global *count* constraint, which counts the number of terms that are equal to a certain term d.

3.6.3 Initial Lookahead

As shown in [122], initial lookahead on constraints can be very helpful in the context of SMT. It makes implicit knowledge (stored in the propagators of the theory solver) explicitly available to the propositional solver. We use this feature as a preprocessing step, restricted to constraint literals. As we can see in Algorithm 10, for every constraint literal $\sigma \in \{\mathbf{T}c, \mathbf{F}c \mid c \in \mathcal{C}\}$ we call CSPPROPAGATION. If σ is conflicting, we add a unary nogood $\{\sigma\}$ such that unit propagation can infer $\overline{\sigma}$. If it is not conflicting, we we add a binary nogood $\{\sigma, \overline{\sigma'}\}$ for every literal σ' which is implied by CSPPROPAGATION. In this way, binary relations between constraints become explicitly available to the ASP solver. For example, $\mathbf{T}(z = 0)$ implies $\mathbf{F}(x + z > 8)$ or $\mathbf{T}(x + y > 8)$ implies $\mathbf{F}(y = 0)$.[5] Initial lookahead can be enabled/disabled in the solver using the command line option `--csp-initial-lookahead`.

[5] Recall that $D(v) = \{0, \dots, 8\}$ for all $v \in \mathcal{V}$.

3.6.4 Optimization

Our systems *clingcon* 1 and 2[6] supports optimization statements over integer variables. Also multiple objective functions are supported. For this, we benefit from the techniques that are available in the CP solver. The optimization can be controlled with two command line options. Use `--csp-opt-val` to set an initial value for the optimization statements. Only better solutions are enumerated. The other option is `--csp-opt-all` which lets the solver enumerate all solutions less or equal to the bound found last. So to compute all optimal solutions, one first computes one optimal solution and then calls the solver again with the optimal value as input to `--csp-opt-val`.

3.7 Evaluation

We developed the system *clingcon* 1, which can solve constraint answer set programs. It extends the ASP system *clingo* 2.0.2[7] with the generic CP solver *gecode* 2.2.0[8].

The used techniques are established in the area of SMT and proven to be efficient. We first compare our approach to the *adsolver* [91], which also uses a lazy approach. It lacks features of CDCL and uses traditional ASP solving algorithms, based on DPLL-style backtracking, combined with a solver for difference logic. In fact, *adsolver*'s implementation relies on *lparse* [113] and *smodels* [109]. The implementation described in [90] allows the usage of difference constraints of the form $x - y > c$ for variables x, y, and constant c; at most one such constraint is allowed within integrity constraints. The underlying CP solver is handcrafted and thus supports incremental solving and backtracking. In fact, no learning is done in the system. Our experiments consider a benchmark suite stemming from the decision support systems for NASA shuttle controllers [7, 8, 98]. It involves mapping logical time steps on real-time and has already been used to appraise *adsolver* in [91]. We compare the system *adsolver* 1.55 with *clingcon* 1. All benchmarks are run on a 3.4GHz Linux system using a time limit of 600s and 3GB memory restriction. We also encoded the problem in pure ASP and solved it using the ASP solver *clingo*. In Table 3.3, the runtime (in seconds) of the different approaches is presented. A — denotes a timeout of 600 seconds. We show the results for 12 randomly picked sample instances for different amount of time steps. The pure ASP encoding with the solver *clingo* can only solve 5 time steps and we observed memory exhaustion on all instances using 7 time steps. The dedicated CASP system *adsolver* can solve up to 13 time steps on some instances and outperforms the pure ASP approach by far. The new *clingcon* system, using a dedicated CDCL approach and no filtering techniques performs even better on all instances, by up to two orders of magnitude. This clearly shows that CASP benefits from the combination of a learning algorithm like CDCL with a dedicated CP solver.

We evaluate how lazy nogoods without any filtering algorithms affect the performance. In Table 3.4, we can see that not learning the asserting nogoods helps to

[6] as well as *clingcon* 3

[7] `https://potassco.org/clingo/`

[8] `http://www.gecode.org`

	clingo	*adsolver*				*clingcon* 1			
timesteps	*5*	*5*	*7*	*11*	*13*	*5*	*7*	*11*	*13*
3-0/025	162	14	51	460	365	1	1	4	5
3-0/050	173	31	108	471	—	1	2	8	12
3-0/100	175	448	188	—	—	1	2	5	7
3-0/125	165	19	60	224	—	1	1	4	8
5-0/025	174	28	107	—	—	1	2	10	15
5-0/050	163	13	42	204	497	1	1	4	7
5-0/100	168	21	66	282	514	1	1	4	5
5-0/125	174	32	104	429	—	1	2	6	10
8-0/025	177	41	140	—	—	1	2	9	7
8-0/050	167	18	54	215	—	1	1	4	8
8-0/100	165	13	41	208	—	1	2	4	5
8-0/125	162	16	53	246	519	1	1	4	7
avg	169	58	84	378	558	1	2	5	8

Table 3.3: Comparing *clingo* 2.0.2, *adsolver*, and *clingcon* 1.

avoid timeouts even for 20 time steps. Exhaustive nogood recording slows down unit propagation more than the additional inferences would help the solver. To make the learned nogoods more useful, we evaluate different reduction methods in the next step.

We have shown that using CDCL to solve CASP problems has some potential. A broader set of benchmarks is used to evaluate the different techniques we proposed.

Two Dimensional Strip Packing The task is to position a given set of squares into a rectangular strip of dimension $n \times m$ without overlapping [110]. This problem is directly taken from the third ASP Competition [28].[9] The position of the corners of the squares is represented using integer variables. Within ASP we check whether two squares overlap.

Incremental Scheduling As presented in the third ASP Competition, incremental scheduling is inspired by a real world application in commercial printing [6]. It is about simulating an online scheduling process where the schedule needs to be up to date while jobs are added and equipment goes off. All jobs have a duration, a deadline, a device and an impact. There is a set of precedences between the jobs that has to be respected. If a job is not finished before its deadline, it gets a tardiness based on its impact. The goal is to find a schedule having an overall tardiness below a given minimum.

[9] `https://www.mat.unical.it/aspcomp2011`

	lazy					learn				
timesteps	*5*	*7*	*11*	*13*	*20*	*5*	*7*	*11*	*13*	*20*
3-0/025	1	1	4	5	17	1	1	4	5	18
3-0/050	1	2	6	11	27	1	2	8	12	30
3-0/100	1	2	10	12	38	1	2	5	7	33
3-0/125	1	1	4	10	133	1	1	4	8	31
5-0/025	1	2	5	14	66	1	2	10	15	118
5-0/050	1	1	4	10	241	1	1	4	7	—
5-0/100	1	1	4	6	25	1	1	4	5	49
5-0/125	1	2	6	9	81	1	2	6	10	73
8-0/025	1	2	11	12	222	1	2	9	7	—
8-0/050	1	1	4	7	457	1	1	4	8	32
8-0/100	1	2	5	6	26	1	2	4	5	23
8-0/125	1	1	4	6	17	1	1	4	7	18
avg	1	2	5	9	113	1	2	5	8	135

Table 3.4: Evaluating lazy nogood generation within *clingcon*.

Quasi Group Completion Quasi Group Completion has been used for benchmarking CP systems in [69]. Given an $n \times n$ square where some of the fields are already filled with numbers from 1 to n. The task is to complete the square with numbers from 1 to n such that all numbers in each row/column are different from each other. We tested 78 instances of size 20×20, partially filled with random numbers.

Weighted Sequence Tree Also, presented in the third ASP Competition, the problem is inspired by finding an optimal join order in the cost-based query optimizer of Oracle [81]. Given a set of nodes S and a maximum cost m, each node i has weight w_i, integer cardinality c_i, and a color $r \in \{red, green, blue\}$. The goal is to find a sequence for all nodes such that the cost of that sequence is below a certain threshold. As the weight and cardinality of a node is given, we can choose the color and position in the sequence ourselves. The cost of the first node is 0, whereas every other node i has the cost

$$cost(i) = \left\{ \begin{array}{ll} w_i + c_i & if\ r_i = green \\ cost(i-1) + w_i & if\ r_i = red \\ cost(i-1) + c_i & if\ r_i = blue \end{array} \right\}$$

The cost of the sequence is the sum of the costs of all leaves.

Unfounded Set Check To increase the spectrum of benchmarks, we conceived a new collection of benchmarks which make use of the CP solver to handle the unfounded set check UfsPropagation for some normal logic programs. Therefore, we reify [52, 60] logic programs, in our case we take:

- *Labyrinth* – the problem of guiding an avatar through a dynamically changing labyrinth to certain fields [28],
- *HashiwoKakero* – a logic puzzle game, and
- *HamiltonianCycle* – finding a cycle in a graph that uses every node only once.

Using a reified program, we can reason about its structure. In particular, we can add an encoding that does the unfounded set check using level-mapping as proposed in [48, 74, 96]. We assign a level to every atom in a strongly connected component and use the CP solver to find a valid mapping. Using this translation, we can solve any non-tight logic program using the CP solver for the unfounded set check. All instances and encodings are freely available.[10]

Settings We run our benchmarks single-threaded on a cluster with 24×8 cores with 2.27GHz each. We restricted each run to use 4GB RAM. In all our benchmarks we used the standard configuration of *clingcon* 2.0, unless stated otherwise. This also means that we used *lazy reasons*, a feature that is enabled by default. So every reason that is generated by the CP solver is not reduced during propagation but when it is needed during conflict analysis.

Reason and Conflict Reduction First, we analyze how much the different conflict and reason reduction methods presented in Section 3.5 differ in size of conflicts and average runtime. As conflicts and reasons are strongly interacting in the CDCL framework, we test the combination of all our proposed algorithms. We denote the filtering algorithms with the following shortcuts: s(*Simple*), b(*Backward Filtering*), f(*Forward Filtering*), c(*Connected Component Filtering*), r(*Range Filtering*) and o(*Connected Component Range Filtering*). We name the filtering algorithm for reasons first, separated by a slash from the algorithm used to filter conflicts. To denote the configuration using *Range Filtering* for reasons and *Forward Filtering* for conflicts, we simply write r/f. The configuration evaluated so far, without applying any reduction methods, is therefore denoted s/s.

We start by showing the impact on average conflict size of all configurations using a heat map in Figure 3-2. It shows the reduction of the conflict size in percentage relative to the worst configuration. The rows represent the used algorithms for reason filtering, the columns represent the algorithms for filtering conflicts. So the worst configuration is represented by a totally black square and a configuration that reduces the average conflict size by half is gray. A completely white field means that the conflict size has been reduced to zero. As we can see in Figure 3-2, the average conflict size is reduced

[10] `http://www.cs.uni-potsdam.de/clingcon/benchmarks.html`

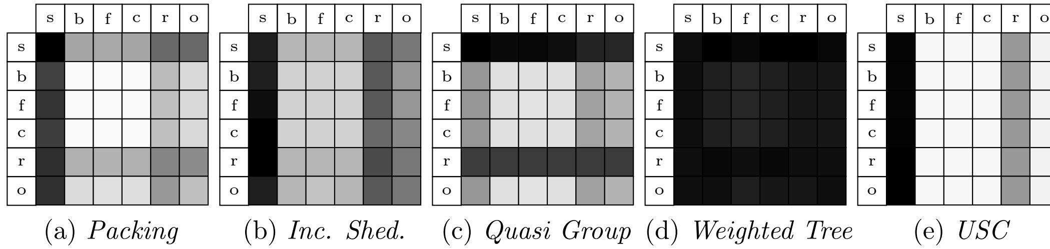

Figure 3-2: Evaluating filtering techniques wrt. conflict size..

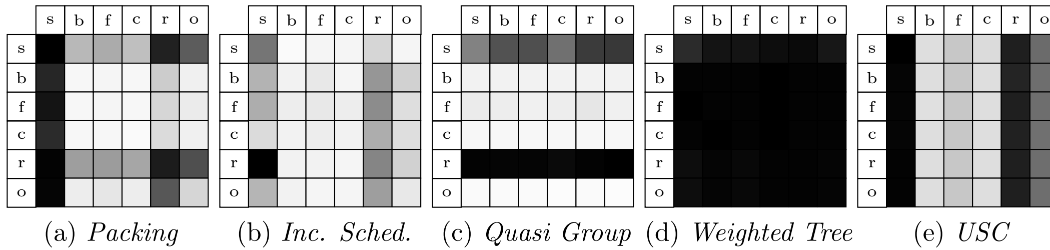

Figure 3-3: Evaluating filtering techniques wrt. runtime.

by all combinations of filtering algorithms. Furthermore, we see that the first row and column, respectively, is usually darker than the others, which indicates that filtering either only conflicts or only reasons is not enough. Also, we see that for the *Unfounded Set Check (USC)* benchmark the filtering of reasons does not have any effect. This is due the encoding of the problem. Since nearly no propagation takes place, no reasons are computed at all. The shades on the *Range Filtering* rows/columns (denoted by r) clearly show that the *Range Filtering* produces larger conflicts. But this is improved by incorporating structure to the filtering algorithm using *Connected Component Range Filtering.*

Next, we want to see if the reduction of the average conflict size also pays off in terms of runtime. Therefore, Figure 3-3 shows the heat map for average runtime. A black square denotes the slowest configuration, while a gray one is twice as fast. As we can clearly see, the reduction of runtime coincides with the reduction of conflict size in most cases. Furthermore, we can see a clear speedup for all benchmark classes except *Weighted Sequence Tree* using the filtering algorithms.

Table 3.5 compares the *Simple* version s/s without using any filtering algorithms, with the configuration o/b (reducing reasons using *Connected Component Range Filtering* and reducing conflicts using *Backward Filtering*), as it has the lowest number of timeouts. It shows the average runtime in seconds together with the number of timeouts, and the average conflict size. We note a speedup of around one order of magnitude on all benchmarks except *Weighted Tree.* The same picture is given for the reduction of conflict size. Whenever the average conflict size (acs) is reduced, this also pays off in terms of runtime.

Instances (#number)	time s/s	time o/b	acs s/s	acs o/b
Packing(50)	888(49)	63(0)	293	40
Inc. Sched.(50)	30(01)	3(0)	15	5
Quasi Group(78)	390(28)	12(0)	480	56
Weighted Tree(30)	484(07)	574(18)	31	31
USC(132)	721(104)	92(1)	454	13

Table 3.5: Evaluating filtering techniques.

CspPropagation on	partial assignments	complete assignments
Packing	63	571
Inc. Sched.	3	11
Quasi Group	12	19
Weighted Tree	574	546
USC	92	82

Table 3.6: The effects of theory propagation wrt. runtime.

Theory Propagation Theory propagation is a technique used in SMT to infer knowledge from the theory given only a partial assignment. By disabling this partial check we can see whether this technique has an influence on the performance of our solver. Table 3.6 shows the average runtime in seconds, with and without theory propagation on our benchmark sets with configuration o/b. We see that it speeds up the solving of the *Packing* problem by an order of magnitude. On other benchmarks relinquishing some propagation strength can improve the overall speed of the system. This is the case as CspPropagation does not infer any useful knowledge on these problems and the time is better spend using unit propagation.

Initial Lookahead In Table 3.7 we evaluate the effects of initial lookahead. We see that the inferred binary nogoods are quite helpful. For the *Quasi Group* and the *Weighted Tree* problem we reduce the number of timeouts and achieve a speedup. On other problems, either no simple nogoods can be inferred or are not helpful. Nevertheless, the time spent on the lookahead was negligible on all of our benchmarks.

Global Constraints To confirm that global constraints speed up the computation, we modeled the *Quasi Group* problem using the global constraint distinct. Using the distinct constraint, we achieve an average runtime of 220 seconds and 18 timeouts over all instances while the decomposed version is much slower. It uses a cubic number of

instances (#number)	time (timeouts)	time with *I.L*	time of *I.L.*	nogoods from *I.L*
Packing(50)	888(49)	882(49)	5	7970
Inc. Sched.(50)	30(01)	40(02)	0	73
Quasi Group(78)	390(28)	355(24)	9	105367
Weighted Tree(30)	484(07)	312(04)	0	1520
USC(132)	721(104)	719(103)	3	1

Table 3.7: Initial Lookahead (*I.L.*).

inequalities and used 390 seconds on average and had 28 timeouts. *Clingcon* confirms, that global constraints are handled more efficiently than their explicit decomposition.

3.8 Conclusion

We introduced a novel approach to integrate constraint processing capabilities into modern ASP solvers based on CDCL. Our semantic approach relies on a propositional language rather than a multi-sorted, first-order language, as used in [19, 90, 91]. With *clingcon* 1 and 2 we developed a system for solving CASP. We are able to use state of the art CP solvers and do not need to implement any CP propagation or optimization techniques as this is done completely in the external CP solver. In this way, we profit from new developments in this area. We provide an open source version of our system using the grounder *gringo*, the ASP solver *clasp*, and the CP solver *gecode*. The software and full language documentation is available for download.[11] We showed that our method based on a modern CDCL algorithm is very suitable for solving CASP and is in alignment with the latest developments in SMT. By following the lazy approach of advanced SMT solvers by abstracting from the constraints in a specialized theory [97], we outperform traditional CASP systems (based on DPLL-style backtracking) by orders of magnitude. Having a declarative input language is a big advantage over pure SMT systems, complex hybrid problems can now easily be expressed using CASP.

A major difficulty in our endeavor was the lack of CP solvers providing an interface supporting conflict-driven learning. We addressed this problem by developing a new algorithmic framework for re-engineering minimal reasons/conflicts. This enables the ASP solver to learn about the structure of the theory, even if the theory solver does not give any information about it (the CP solver acts as a blackbox). We furthermore show that while applying these filtering methods, knowledge is discovered that is valuable for the overall search process and can therefore speed up the search by orders of magnitude. We point out that the developed techniques regarding the filtering methods can also be applied to other theories then CP.

[11] `https://sourceforge.net/projects/potassco/files/clingcon/` For newer versions please visit: `https://potassco.org/clingcon/`

Chapter 4

Encoding Constraint Satisfaction Problems

Encoding finite linear Constraint Satisfaction Problems (CSPs; [106]) as propositional formulas and solving them by using modern SAT solvers has proven to be a highly effective approach by the award-winning *sugar*[1] system. The CP solver *sugar* reads a CSP instance and transforms it into a propositional formula in Conjunctive Normal Form (CNF). The translation relies on the order encoding [31, 116], and the resulting CNF formula can be solved by an off-the-shelf SAT solver.

In this chapter, we elaborate upon an alternative approach based on ASP and present the resulting *aspartame*[2] framework, serving two purposes. First, *aspartame* provides a library for solving CSPs as part of an encompassing logic program. Second, it constitutes an ASP-based CP solver similar to *sugar*. The major difference between *sugar* and *aspartame* rests upon the implementation of the translation of CSPs into Boolean constraint problems. While *sugar* implements a translation into CNF in Java, *aspartame* starts with a representation as a set of facts.[3] Its architecture is given in Figure 4-1. When used as a library, this set of facts (representing the CSP) must be supplied by the user. In turn, these facts are combined with a general-purpose ASP encoding for CP solving (also based on the order encoding), which is subsequently instantiated by an off-the-shelf ASP grounder, in our case *gringo*. The resulting propositional logic program is then solved by an off-the-shelf ASP solver (here *clasp*). The high-level approach of ASP has obvious advantages. First, instantiation is done by general-purpose ASP grounders rather than dedicated implementations. Second, the elaboration tolerance of ASP makes it easy to maintain and modify the encoding. Therefore, it is easy to experiment with novel or heterogeneous encodings. However, the question is whether the high-level approach of *aspartame* matches the performance of the more dedicated *sugar* system. We empirically address this question by contrasting the performance of both CP solvers, while fixing the back-end solver to *clasp*, used as both a SAT and an ASP solver.

[1] `http://bach.istc.kobe-u.ac.jp/sugar`

[2] `https://potassco.org/labs/2016/09/20/aspartame.html`

[3] When used as CP solver, *aspartame* re-uses *sugar*'s front-end for parsing and normalizing (non-linear) CSPs. We also extended *sugar* to produce a fact-based representation.

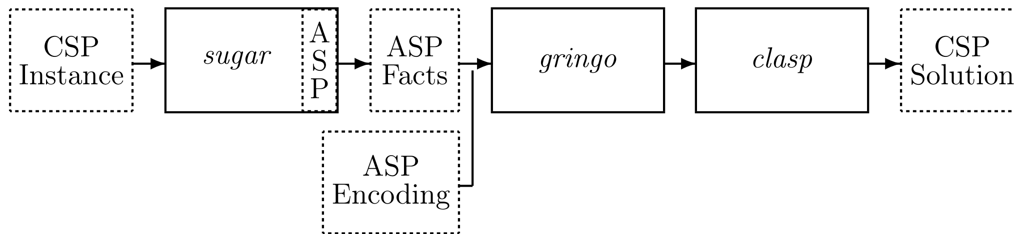

Figure 4-1: Architecture of *aspartame*.

From an ASP perspective, we gain insights into advanced modeling techniques for solving CSPs. The ASP encoding, implementing CP solving with *aspartame*, has the following features:

- usage of function terms to abbreviate structural subsums,
- avoidance of (artificial) intermediate integer variables (to split sum expressions),
- a collection of encodings for the distinct constraint.

This chapter is structured as follows.

- As the presented translation techniques can handle linear constraints, we introduce a set of constraints that can easily be transformed into linear constraints.
- After a short overview of several known translation techniques for linear constraints, we define the nogoods of a CSP using the order encoding. Given this, we develop a characterization of constraint stable models using nogoods only.
- We show how to represent and encode a CSP using a basic ASP encoding. Afterwards, several improvements are discussed.
- It is possible to extend *aspartame* to handle CASP problems. We give a short example on how this is done.
- Finally, we evaluate the presented techniques of our system and also compare it to the state of the art CP solver *sugar*.

Parts of this chapter have been published in [11]. We extended the publication by a more elaborate definition of the order encoding. In order to represent CASP using this encoding, reified, linear constraints are used.

4.1 Normalizing Constraints

This chapter uses a technique to translate linear constraints of the form

$$a_1v_1 + \cdots + a_kv_k \leq b \tag{4.1}$$

into a set of nogoods, given that a_i, b are integers, and v_i are integer variables for $1 \leq i \leq k$. We introduce some basic translation techniques that allow us to handle any constraint of the form

$$a_1v_1 + \cdots + a_kv_k \circ b \tag{4.2}$$

where $\circ \in \{<, >, \leq, \geq, =, \neq\}$. While translating

$$\begin{aligned}
a_1v_1 + \cdots + a_kv_k < b \text{ into } \quad & a_1v_1 + \cdots + a_kv_k \leq b - 1, \\
a_1v_1 + \cdots + a_kv_k \geq b \text{ into } & -a_1v_1 - \cdots - a_kv_k \leq -b, \text{ and} \\
a_1v_1 + \cdots + a_kv_k > b \text{ into } & -a_1v_1 - \cdots - a_kv_k \leq -b - 1
\end{aligned}$$

is trivial, other constraints require more elaborate techniques.

Given a constraint logic program P over regular atoms $\mathcal{A}$ and constraint atoms $\mathcal{C}$ associated with CSP $(\mathcal{V}, D, C)$, we translate a constraint $\gamma(c)$, $c \in \mathcal{C}$ of the form $a_1v_1 + \cdots + a_kv_k = b$, by introducing new rules. We therefore translate P into a new constraint logic program P' over regular atoms $\mathcal{A} \cup \{c\}$ and constraint atoms $\mathcal{C} \setminus \{c\} \cup \{l, r\}$. We let $\gamma(l) = a_1v_1 + \cdots + a_kv_k \leq b$ and $\gamma(r) = -a_1v_1 - \cdots - a_kv_k \leq -b$[4] and add the new rule $c \leftarrow l, r$ to P'. Let us have a look at the following example program.

Example 2

$$a \leftarrow (x + y > 8), (z - w = 42)$$

The translation process changes the first constraint and adds an additional rule for the second constraint.

$$\begin{aligned}
a &\leftarrow (-x - y \leq -9), c \\
c &\leftarrow (z - w \leq 42), (-z + w \leq -42)
\end{aligned}$$

For inequality constraints $\gamma(c) \equiv a_1v_1 + \cdots + a_kv_k \neq b$ we propose a similar translation process adding the rules

$$\begin{aligned}
c &\leftarrow (a_1v_1 + \cdots + a_kv_k \leq b - 1) \\
c &\leftarrow (-a_1v_1 - \cdots - a_kv_k \leq -b - 1).
\end{aligned}$$

As our implementation only supports constraints that can be normalized to the form $a_1v_1 + \cdots + a_kv_k \leq b$, we assume that the constraint logic program has been normalized in the presented way if not mentioned otherwise.

4.2 Encoding Linear Constraints

Given a linear constraint, several translations into Boolean formulas exists. The translated constraints then can be solved using CDCL like algorithms while learning

[4] This is equivalent to $a_1v_1 + \cdots + a_kv_k \geq b$.

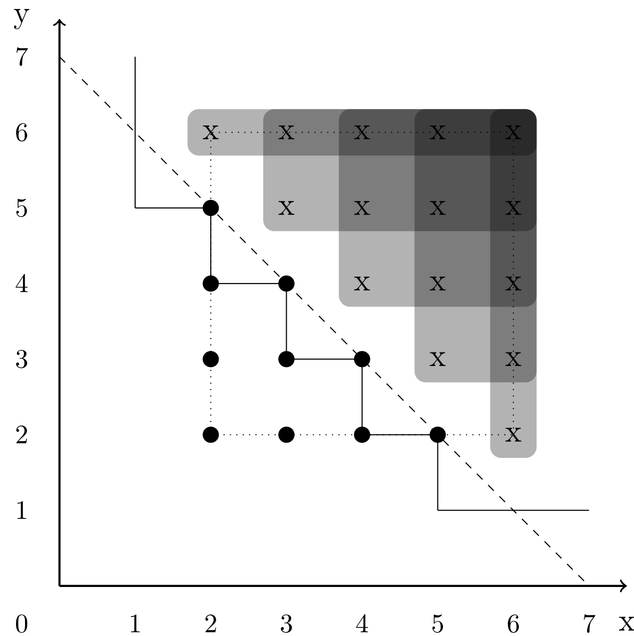

Figure 4-2: A graphical representation of the linear constraint $x + y \leq 7$.

inter-constraint dependencies. We give a brief overview over common translations and explain their advantages and drawbacks. For a more detailed view, [43] describes different translations, their propagation strengths and their complexity.

Direct Encoding The direct encoding [121] is the most obvious way to encode a linear constraint. It uses one atom for each domain value d of each variable v, where the atom has the meaning $v = d$. For a linear constraint, for each forbidden combination of values a nogood is created. E.g. for constraint $x + y \leq 7$, the nogood $\{\mathbf{T}(x = 5), \mathbf{T}(y = 4)\}$ is part of the encoding. The linear constraint $x + y \leq 7$ is given by the dashed line in Figure 4-2 and every dot on the left side of it is a valid assignment for $D(x) = D(y) = \{2, \ldots, 6\}$. Each `x` is represented by one nogood in the direct encoding, representing a forbidden assignment of the two variables. This encoding technique does not achieve arc consistency[5] by unit propagation.

Logarithmic Encoding The logarithmic encoding [121] is optimized for the size of the encoding. For every variable v, we have one atom for each bit of the binary representation of its domain. So atom v_j means that the value of v has the jth bit set to true in its binary notation. For a linear constraint, each forbidden combination is ruled out by a nogood describing the combination of forbidden bits. For the constraint used above, the nogood $\{\mathbf{T}(x_1), \mathbf{T}(x_3), \mathbf{T}(y_3)\}$ is part of the encoding and expresses the same condition as the nogood shown for the direct encoding. The logarithmic encoding also does not achieve arc consistency by unit propagation and its propagation

[5] Every admissible value of a variable is consistent (wrt. a constraint) with some admissible value of another variable.

strength is weak. The advantage of this encoding technique is its compactness.

Support Encoding The support encoding [67] can be used either with the direct or the logarithmic encoding. The idea is to encode the supported regions instead of the conflicting ones. In some cases, this technique reduces the number of nogoods.

Range encoding In a range encoding, a variable v and its range $l \leq v \leq u$ are encoded by propositional variables v_l^u for every possible range. Hence, a quadratic number of atoms is needed to represent a variable. Channelling constraints are introduced to say that v_l^u implies $v_{prev(l,v)}^u$ and $v_l^{next(u,v)}$. The advantage of this encoding is that holes in the domain can be propagated.

Order Encoding The order encoding [31, 115, 116] introduces exactly one atom per domain value per variable. The atom $(v \leq d)$ has the meaning $v \leq d$ for a variable v and its values $d \in D(v)$. Channelling constraints are introduced to say that $(v \leq d)$ implies $(v \leq \mathit{next}(d, v))$. For a linear constraint, one nogood can remove a whole set of conflicting values, as we can see in Figure 4-2. Each gray box is represented by a nogood using the order encoding, e.g. $\{\mathbf{F}(x \leq 2), \mathbf{F}(y \leq 4)\}$ for ruling out every solution where x is greater than 2 and y is greater than 4. A detailed description of the order encoding is to be given in Section 4.3.

Compact Order Encoding The compact order encoding [14, 117] is a mixture between the order and the logarithmic encoding. Each variable is represented using digits of base B. If B is the size of the domain, this is equivalent to the order encoding and if $B = 2$ this is the logarithmic encoding. This encoding is made for representing larger variables by using a bitwise representation and sacrificing propagation strength.

4.3 Nogoods of Constraint Satisfaction Problems

We decided on the order encoding technique to translate a CSP into a set of nogoods, as it is well established [103, 114]. To this end, given a CSP $(\mathcal{V}, D, C)$, we let $\mathcal{O}_\mathcal{V}$ stand for the set of order atoms associated with variables in $\mathcal{V}$. Whenever the set $\mathcal{V}$ is clear from the context, we drop it and simply write $\mathcal{O}$. More precisely, we introduce an *order atom* $(v \leq d) \in \mathcal{O}$ for each constraint variable $v \in \mathcal{V}$ and value $d \in D(v), d \neq \mathit{ub}(v)$. We refer to signed literals over $\mathcal{O}$ as *signed order literals*.

Now, we are ready to map a linear CSP into a set of nogoods. First, we need to make sure that each variable in $\mathcal{V}$ has exactly one value from its domain in D. To this end, we define the following set of nogoods.

$$\Phi(\mathcal{V}, D) = \{\{\mathbf{T}(v \leq d), \mathbf{F}(v \leq \mathit{next}(d, v))\} \mid v \in \mathcal{V}, d \in D(v), \mathit{next}(d, v) < \mathit{ub}(v)\} \tag{4.3}$$

Intuitively, each such nogood stands for an implication "$(v \leq d) \Rightarrow (v \leq d+1)$". In Example 1, we get the following nogoods.

$$\Phi(\{x\}, \{D(x)\}) = \{\{\mathbf{T}(x \leq 0), \mathbf{F}(x \leq 1)\}, \ldots, \{\mathbf{T}(x \leq 21), \mathbf{F}(x \leq 22)\}\}. \quad (4.4)$$

Second, we need to establish the relation between constraint atoms $\mathcal{C}$ and their associated linear constraints in C. Following [49], a *reified constraint* is an equivalence "$\mathbf{T}c \Leftrightarrow \gamma(c)$" where $c \in \mathcal{C}$; it is decomposable into two *half-reified* constraints "$\mathbf{T}c \Rightarrow \gamma(c)$" and "$\mathbf{F}c \Rightarrow \overline{\gamma(c)}$".

To translate constraints into nogoods, we need to translate expressions of the form $av + b \leq 0$ for $v \in \mathcal{V}$ and integers a, b into signed ordered literals.[6] Following [116], we then define $(av + b \leq 0)^\ddagger$ as

$$(av + b \leq 0)^\ddagger = \begin{cases} (v \leq \lfloor \frac{-b}{a} \rfloor)^\dagger & \text{if } a > 0 \\ \overline{(v \leq \lceil \frac{-b}{a} \rceil - 1)^\dagger} & \text{if } a < 0 \end{cases}$$

where $(v \leq d)^\dagger$ is defined for $lb(v) \leq d < ub(v)$ as

$$(v \leq d)^\dagger = \begin{cases} \mathbf{T}(v \leq d) & \text{if } d \in D(v) \\ \mathbf{T}(v \leq prev(d, v)) & \text{if } d \notin D(v) \end{cases}$$

If $d \geq ub(v)$ then $(v \leq d)^\dagger = \mathbf{T}\emptyset$; if $d < lb(v)$ then $(v \leq d)^\dagger = \mathbf{F}\emptyset$, where $\emptyset$ stands for the empty body.[7] Expressing constraint $x < 7$ from Example 1 in terms of signed order literals results in $(1 \cdot x + (-6) \leq 0)^\ddagger = \mathbf{T}(x \leq 6)$. The signed literal $\mathbf{T}(x \leq 6)$ indicates that 6 is the largest integer satisfying the constraint. Also, we get the signed literals $(x \leq 0)^\dagger = \mathbf{F}\emptyset$ and $(x \leq 23)^\dagger = \mathbf{T}\emptyset$.

We sometimes use $<$,$>$, or $\geq$ as operators in these expressions and implicitly convert them to the normal form $av + b \leq 0$ to be used in this translation. Accordingly, the complementary constraint yields $(x > 6)^\ddagger = ((-1) \cdot x + 7 \leq 0)^\ddagger = \overline{(x \leq \lceil \frac{-7}{-1} \rceil - 1)^\dagger} = \mathbf{F}(x \leq 6)$.

The actual relation between the constraint atoms in $\mathcal{C}$ and their associated linear constraints in C is established via the following nogoods.

$$\Psi(\mathcal{C}) = \bigcup_{c \in \mathcal{C}} \psi(\mathbf{T}c, \gamma(c)) \cup \psi(\mathbf{F}c, \overline{\gamma(c)}) \ . \quad (4.5)$$

For all constraint atoms $c \in \mathcal{C}$ associated with the linear constraint $\gamma(c) = \sum_{i=1}^{n} a_i v_i \leq b$ in C, we define for both of its half-reified constraints the set of nogoods

$$\psi(\mathbf{T}c, \textstyle\sum_{i=1}^{n} a_i v_i \leq b) = \{\{\mathbf{T}c\} \cup (\delta \setminus \{\mathbf{T}\emptyset\}) \mid \delta \in \phi(\sum_{i=1}^{n} a_i v_i \leq b), \mathbf{F}\emptyset \notin \delta\} \quad (4.6)$$

$$\psi(\mathbf{F}c, \overline{\textstyle\sum_{i=1}^{n} a_i v_i \leq b}) = \{\{\mathbf{F}c\} \cup (\delta \setminus \{\mathbf{T}\emptyset\}) \mid \delta \in \phi(\overline{\textstyle\sum_{i=1}^{n} a_i v_i \leq b}), \mathbf{F}\emptyset \notin \delta\} \quad (4.7)$$

[6] Any linear inequality using $<, >, \leq, \geq$, and one variable can be converted into this form.

[7] We use $\mathbf{T}\emptyset$ and $\mathbf{F}\emptyset$ as representatives for tautological and unsatisfiable signed literals; they are removed in (4.6) and (4.7) below.

where

$$\phi(\sum_{i=1}^{n} a_i v_i \le b) = \left\{ \begin{array}{ll} \{(a_1 v_1 > b)^\ddagger\} & \text{if } n = 1 \\ \{(a_1 v_1 \ge d)^\ddagger\} \cup \delta & \text{if } n > 1 \\ \quad \delta \in \phi(\sum_{i=2}^{n} a_i v_i \le b - d), & \\ \quad d \in img(a_1 v_1) & \end{array} \right\} \tag{4.8}$$

Note that nogoods with $\mathbf{T}\emptyset$ and $\mathbf{F}\emptyset$ are simplified in (4.6) and (4.7). Also, observe that the definition of ϕ is recursive although this does not show up with our simple examples.

In Example 1, we obtain

$$\psi(\mathbf{T}(x < 7), x \le 6) = \{\{\mathbf{T}(x < 7), \mathbf{F}(x \le 6)\}\} \tag{4.9}$$
$$\psi(\mathbf{F}(x < 7), -x \le -7) = \{\{\mathbf{F}(x < 7), \mathbf{T}(x \le 6)\}\} \tag{4.10}$$

and

$$\psi(\mathbf{T}(x \ge 22), -x \le -22) = \{\{\mathbf{T}(x \ge 22), \mathbf{T}(x \le 21)\}\} \tag{4.11}$$
$$\psi(\mathbf{F}(x \ge 22), x \le 21) = \{\{\mathbf{F}(x \ge 22), \mathbf{F}(x \le 21)\}\} \tag{4.12}$$

Taken together, these nogoods realize the aforementioned equivalence between the constraint atoms $(x < 7), (x \ge 22)$, and their associated constraints. Note that $(x < 7)$ and $(x \ge 22)$ are constraint atoms in $\mathcal{C}$, while $(x \le 6)$ and $(x \le 21)$ are order atoms in $\mathcal{O}$ and thus belong to the encoding of the constraints. For further illustration, reconsider the Boolean assignment $\mathbf{B}|_{\mathcal{C}} = \{\mathbf{F}(x < 7), \mathbf{F}(x \ge 22)\}$ inducing the first constraint stable models in (2.10). Applying unit propagation, we get $\mathbf{F}(x \le 6)$ via (4.10) and in turn $\mathbf{F}(x \le 5)$ to $\mathbf{F}(x \le 1)$ via the nogoods in $\Phi(\{x\}, \{D(x)\})$ in (4.3). Similarly, making $\mathbf{F}(x \ge 22)$ true yields $\mathbf{T}(x \le 21)$, via the nogoods in (4.12). All in all, a CSP $(\mathcal{V}, D, C)$ is characterized by the nogoods in $\Phi(\mathcal{V}, D)$ and $\Psi(\mathcal{C})$.

While in Section 3.1 the corresponding constraint variable assignment $\mathbf{C}$ is determined externally, it can be directly extracted from a solution $\mathbf{B}$ for $\Phi(\mathcal{V}, D)$ by means of the following functions: The upper bound for a view v relative to a Boolean assignment $\mathbf{B}$ is given by $ub_{\mathbf{B}}(v) = \min(\{ub(v)\} \cup \{d \mid d \in img(v), (v \le d)^\ddagger \in \mathbf{B}\})$ and its lower bound by $lb_{\mathbf{B}}(v) = \max(\{lb(v)\} \cup \{d \mid d \in img(v), (v \ge d)^\ddagger \in \mathbf{B}\})$. Then, $\mathbf{C}(v) = lb_{\mathbf{B}}(v) = ub_{\mathbf{B}}(v)$ for all $v \in \mathcal{V}$. Accordingly, the above Boolean assignment corresponds to the solutions $\{\{x \mapsto v\} \mid v \in \{7, \ldots, 21\}\}$.

Proposition 4.3.1

Let P be a constraint logic program over regular atoms $\mathcal{A}$ and constraint atoms $\mathcal{C}$ associated with the CSP $(\mathcal{V}, D, C)$ and let $\mathbf{B}$ be a solution of the set $\Phi(\mathcal{V}, D) \cup \Psi(\mathcal{C})$ of nogoods.
Then, $\mathbf{B}|_{\mathcal{C}} = \{\mathbf{T}c \mid \gamma(c) \in sat_{\mathbf{C}}(C)\} \cup \{\mathbf{F}c \mid \overline{\gamma(c)} \in sat_{\mathbf{C}}(C)\}$ iff $\mathbf{C} = \{v \mapsto lb_{\mathbf{B}}(v) \mid v \in \mathcal{V}\}$.

Proof 4.3.1 *The correctness follows directly from the correctness of the order encoding as shown in [116].*

Combining the nogoods stemming from the logic program and its associated CSP, we obtain the following characterization of constraint logic programs.

Theorem 4.3.2

Let P be a constraint logic program over regular atoms $\mathcal{A}$ and constraint atoms $\mathcal{C}$ associated with the CSP $(\mathcal{V}, D, C)$.
Then, $(X, \mathbf{C})$ is a constraint stable model of P iff $(\mathbf{C}, sat_{\mathbf{C}}(C))$ is a configuration for $(\mathcal{V}, D, C)$, and $X = \mathbf{B}^{\mathbf{T}} \cap (\mathcal{A} \cup \mathcal{C})$ for any (unique) solution $\mathbf{B}$ of the set $\Delta_P^{\mathcal{C}} \cup \Lambda_P^{\mathcal{C}} \cup \Psi(\mathcal{C}) \cup \Phi(\mathcal{V}, D)$ of nogoods and $\mathbf{C} = \{v \mapsto lb_{\mathbf{B}}(v) \mid v \in \mathcal{V}\}$ being a total assignment over $\mathcal{V}$.

Proof 4.3.2 *The proof of this theorem is obtained by replacing* $\{\{\mathbf{F}c\} \mid \gamma(c) \in sat_{\mathbf{C}}(C)\} \cup \{\{\mathbf{T}c\} \mid \overline{\gamma(c)} \in sat_{\mathbf{C}}(C)\}$ *in Theorem 3.1.1 with* $\Phi(\mathcal{V}, D) \cup \Psi(\mathcal{C})$. *The correctness of the solution is assured by Proposition 4.3.1.*

4.4 Encoding Constraint Satisfaction Problems

For using *aspartame* as a CP solver, we extended the front-end of the *sugar* system by an output component representing CSPs in terms of ASP facts. The latter also constitute the CSP instances when using *aspartame* as a library. As usual, the resulting facts can then be combined with a first-order encoding processable with off-the-shelf ASP systems. In what follows, we describe *aspartame*'s fact format and we present dedicated ASP encodings utilizing function terms to capture substructures in CSP instances.

Fact Format Facts express the variables and constraints of a CSP instance in the syntax of ASP grounders like *gringo*. Their format is easiest explained on the set of clauses from Example 3 with $D(x) = D(y) = D(z) = \{1, 2, 3\}$ and $D(b) = \{0, 1\}$. Each clause is the disjunction of its elements, and its representation is shown in Listing 4.1.

Example 3

$$C_1 = \{distinct(x, y, z)\}$$
$$C_2 = \{b, 4x - 3y + z \leq 0\}$$
$$C_3 = \{\neg b, -4x + 3y \leq -6\}$$

While facts of the predicate `var`/2 provide labels of Boolean variables, like b, the predicate `var`/3 includes a third argument for declaring the domains of integer variables, like x, y, and z. Domain declarations rely on function terms `range(`l`,`u`)`, standing for

```
1 var(bool,b).      var(int,(x;y;z),range(1,3)).

3 constraint(1,global(alldifferent,arg(x,arg(y,arg(z,nil))))).
4 constraint(2,b).
5 constraint(2,op(le,op(add,op(add,op(mul,4,x),op(mul,-3,y)),op(mul,1,z)),0)).
6 constraint(3,op(neg,b)).
7 constraint(3,op(le,op(add,op(mul,-4,x),op(mul,3,y)),-6)).
8 objective(minimize,x).
```

Listing 4.1: Facts representing the CSP from above.

integer intervals $[l, u]$. While one term, `range(1,3)`, suffices for the common domain $\{1, 2, 3\}$ of x, y, and z (line 1 in Listing 4.1), in general, several intervals can be specified (via separate facts) to form non-contiguous domains. Note that the interval format for integer domains offers a compact fact representation of domains; e.g., the single term `range(1,10000)` captures a domain of 10000 elements. Furthermore, the usage of meaningful function terms avoids any need for artificial labels to refer to domains or parts thereof.

The literals of constraint clauses are also represented by means of function terms. In fact, the second argument of `constraint`/2 in line 3 of Listing 4.1 stands for $\mathit{distinct}(x, y, z)$ from the constraint clause C_1 in Example 3, identified by the first argument of `constraint`/2. Since each fact of predicate `constraint`/2 is supposed to describe a single literal only, constraint clause identifiers establish the connection between individual literals of a clause. The function term expressing the distinct constraint includes an argument list of the form `arg(`v_1`,arg(`$\dots$`,arg(`v_n`,nil)`$\dots$`))`, in which $v_1, \dots, v_n$ refer to integer variables. In line 3 of Listing 4.1, a distinct constraint over arguments $\vec{v}$ is declared via `global(alldifferent,`$\vec{v}$`)`; at present, `alldifferent` is a fixed keyword in facts used by *aspartame*, but support for other kinds of global constraints can be added in the future. The more complex term of the form `op(le,`Σ`,`m`)` in line 5 stands for a linear inequality $\Sigma \leq m$. In particular, the inequality $4x - 3y + z \leq 0$ from C_2 is represented by nested `op(add,`Σ`,`av`)` terms whose last argument av and deepest Σ part are of the form `op(mul,`a`,`v`)`; such nesting corresponds to the precedence $(((4 * x) + (-3 * y)) + (1 * z)) \leq 0$. The representation by function terms captures linear inequalities of arbitrary arity and, as with integer intervals, associates (sub)sums with canonical labels. Finally lines $4-5$ denote the clause C_2. In line 6, a Boolean constraint, e.g. $op(neg, b)$ using Boolean variables and the operator *neg* for classical negation is shown. To state the objective functions for minimizing/maximizing a set of variables we use the predicate `objective(minimize,`x`)` where x denotes an integer variable that is to be minimized.

First-Order Encoding In addition to a dedicated output component of *sugar* for generating ASP facts, *aspartame* comes with several alternative first-order ASP encodings for solving CSP instances. In the following, we first describe a basic encoding that implements the order encoding techniques [114, 116] from Section 4.3 concisely, and then present optimizations and extensions for the distinct constraints. We also show that these *aspartame* encodings can be used as library for solving CASP and

constraint optimization problems. To concentrate on the encoding of linear constraints and distinct, we simplified the encoding and stripped off capabilities for handling Boolean variables and restrict ourselves to unary constraint clauses in the following. Our encodings are given in the language of the grounder *gringo* 4 that features the Lua scripting language to help with more complex calculations during grounding.

Basic Encoding In Listing 4.2 we show auxiliary predicates that we use in our encoding. Given a CSP instance $(\mathcal{V}, D, C)$, for each integer variable $v \in \mathcal{V}$, the domain values of the variables are kept in a Lua table, and then the lower and upper bounds of each v are calculated via Lua and captured in the second arguments of `lb`/2 and `ub`/2 respectively in lines 1−3. Each literal of a constraint clause is classified into a distinct constraint expressed by `alldiff`/1 or a linear inequality by `wsum`/1 in lines 5−7. The identifier in the first argument of `constraint`/2 is removed in lines 6−7, since the *aspartame* encoding presented here is restricted to unary constraint clauses. For each linear inequality $\sum_{i=1}^{n} a_i v_i \leq b$, in line 6, $\sum_{i=1}^{n} a_i v_i$ is sorted in descending order of $|D(v_i)|$ via Lua, and then the constraint is captured in the argument of predicate `wsum`/1. Each sum Σ in `op(_,`Σ`,_)` is decomposed into (sub)sums in lines 9−10, and then the lower and upper bounds of them are calculated and captured in the second arguments of `inf`/2 and `sup`/2 respectively in lines 13−19. In line 11, a predicate `unary_exp(`e`)` is generated if e is an unary expression of the form `op(mul,`a_i`,`v_i`)`.

```
1  var(V) :- var(int,V,_).
2  lb(V,@getLB(V)) :- var(V).
3  ub(V,@getUB(V)) :- var(V).

5  global(I,global(Func,Args)) :- constraint(I,global(Func,Args)).
6  wsum(@sortWsum(L))          :- constraint(I,L); not global(_,L).
7  alldiff(Args)               :- global(_,global(alldifferent,Args)).

9  exp(E)     :- wsum(op(_,E,_)).
10 exp(E1;E2) :- exp(op(add,E1,E2)).
11 unary_exp(op(mul,A,V)) :- exp(op(mul,A,V)).

13 inf(op(mul,A,V),A*LB)  :- exp(op(mul,A,V)); A > 0; lb(V,LB).
14 inf(op(mul,A,V),A*UB)  :- exp(op(mul,A,V)); A < 0; ub(V,UB).
15 inf(op(add,E1,E2),A+B) :- exp(op(add,E1,E2)); inf(E1,A); inf(E2,B).

17 sup(op(mul,A,V),A*UB)  :- exp(op(mul,A,V)); A > 0; ub(V,UB).
18 sup(op(mul,A,V),A*LB)  :- exp(op(mul,A,V)); A < 0; lb(V,LB).
19 sup(op(add,E1,E2),A+B) :- exp(op(add,E1,E2)); sup(E1,A); sup(E2,B).
```

Listing 4.2: Auxiliary predicates

Encoding Integer Variables To define the order atoms $\mathcal{O}$ and nogoods $\Phi(\mathcal{V}, D)$ from Equation (4.3) we use Listing 4.3. The first line introduces an order atom `p(v,d)`, meaning $(v \leq d)$, for every $v \in \mathcal{V}$ and $d \in D(v)$. We use a choice rule [56] to say that we can chose any truth-assignment for them. To ensure that each variable has exactly one value, lines 2−3 introduce the nogoods from $\Phi(\mathcal{V}, D)$. Note that the Lua function `getDom(`v`)` returns $D(v)$, and `getSimpGT(`a, v`)` $= next(a, v)$. For illustration, consider an integer variable $x \in \{2, 3, 4, 5, 6\}$ represented by `var(int,`x`,range(2,6))` as an ASP fact. The grounded, propositional logic program is

```
{p(x,2), p(x,3), p(x,4), p(x,5), p(x,6)}.
:- p(x,2); not p(x,3).
:- p(x,3); not p(x,4).
:- p(x,4); not p(x,5).
:- p(x,5); not p(x,6).
:- not p(x,6).
```

resulting in the following stable models:

stable models	interpretation
{p(x,6)}	$x \mapsto 6$
{p(x,5),p(x,6)}	$x \mapsto 5$
{p(x,4),p(x,5),p(x,6)}	$x \mapsto 4$
{p(x,3),p(x,4),p(x,5),p(x,6)}	$x \mapsto 3$
{p(x,2),p(x,3),p(x,4),p(x,5),p(x,6)}	$x \mapsto 2$

Note that the nogoods from $\Phi(\mathcal{V}, D)$ are represented using integrity constraints.

```
1 { p(V,A) : A = @getDom(V) } :- var(V).
2 :- p(V,A); not p(V,B); B = @getSimpGT(A,V); A < UB; ub(V,UB).
3 :- not p(V,UB); var(V); ub(V,UB).
```

Listing 4.3: Encoding of Integer Variables

Encoding of Linear Constraints The nogoods of a linear constraint are recursively defined as $\phi(\sum_{i=1}^{n} a_i v_i \leq b)$ in Equation (4.8). Following this equation, the first rule in Listing 4.4 describes the reduction of an (n)-ary constraint $\sum_{i=1}^{n} a_i v_i \leq b$ ($n \geq 2$) in line 3 to an $(n-1)$-ary constraint $\sum_{i=2}^{n} a_i v_i \leq b-d$ (line 1), by removing the last unary expression of the constraint and subtracting d from the right hand side. This is done for all values $d \in img(a_1 v_1)$. As a condition to the rule, the literal representing $(a_n v_n \geq d)^{\ddagger}$ is added in lines 4–5. The Lua function getSimpLE(d − 1, v) simply returns $prev(d, v)$, and lines 4–5 correspond to the transformation $()^{\ddagger}$ which is described in Section 4.3. This encoding procedure can be optimized by considering the validity and inconsistency of the recursive part $\sum_{i=2}^{n} a_i v_i \leq b - d$, which can reduce the number of nogoods. The validity and inconsistency of the recursive part can be captured by inequalities $b - d \geq \sup(\sum_{i=2}^{n} a_i v_i)$ and $b - d < \inf(\sum_{i=2}^{n} a_i x_i)$, respectively, where inf and sup indicate the lower and upper bounds of the linear expressions. When the recursive part is valid, the nogood containing it is unnecessary. When it is inconsistent, the literal of the recursive part can be removed, and moreover only one such nogood is sufficient. Based on the observations above, the function getDomOpt() restricts the values of d in such a way that the recursive part becomes neither valid nor inconsistent, if $(a_n v_n \geq d)^{\ddagger}$ holds. In the second rule, only one wsum(op(le, op(mul, a, v), $b - Inf$)) corresponding to $a_1 v_1 \leq b - \inf(\sum_{i=2}^{n} a_i v_i)$ is generated if there exists at least one value in $img(a_1 v_1)$ such that the recursive part becomes inconsistent.

```
1  wsum(op(le,X,B-A*D)) :-
2          not unary(X) : opt_binary == 1;
3          wsum(op(le,op(add,X,op(mul,A,V)),B)),
4          not p(V,D') : A > 0, D' = @getSimpLE(D-1,V);
5              p(V,D)  : A < 0;
6          D = @getDomOpt(V,A,B-Sup,B-Inf), inf(X,Inf), sup(X,Sup).

8  wsum(op(le,op(mul,A,V),B-Inf)) :-
9          wsum(op(le,op(add,X,op(mul,A,V)),B)),
10         B-Inf < Sup, inf(X,Inf), sup(op(mul,A,V),Sup).
```

Listing 4.4: Encoding of $\sum_{i=1}^{n} a_i v_i \leq b$ $(n \geq 2)$

```
1  :- wsum(op(le,op(mul,A,V),B)), Inf <= B, B < Sup,
2            inf(op(mul,A,V),Inf), sup(op(mul,A,V),Sup),
3            not p(V,B') : A > 0;
4                p(V,B') : A < 0;
5            B' = @getLE(B,A,V).

7  :- wsum(op(le,op(mul,A,V),B)), Inf > B, inf(op(mul,A,V),Inf).
```

Listing 4.5: Encoding of $a_1 v_1 \leq b$

Now, that we have a recursive definition for constraints with an arity of two or above, the encoding of unary expressions is described in Listing 4.5. If the unary expression $a_1 v_1 \leq b$ is neither valid nor inconsistent, the first rule translates the expression directly to its order literal $(a_1 v_1 \leq b)^{\ddagger}$, using the Lua function $\texttt{getLE}(v, a, b) = \mathit{prev}(b+1, av)$. It therefore ensures that, whenever a unary expression is true, the respective order literal has to hold. If the unary expression is inconsistent, the second rule ensures that it never holds.

We refer to the encoding of Listings 4.2–4.5 as *basic encoding*. This encoding can concisely implement CP solving based on the order encoding techniques by utilizing the feature of function terms. Moreover, it proposes an alternative approach to splitting sum expressions. In fact, the basic encoding splits them by generating the instances of predicate `wsum`/1 during recursive encoding, rather than by introducing intermediate integer variables during preprocessing like *sugar*'s CSP-to-CSP translation. It is noted that global constraints such as *alldifferent* and others need to be translated into linear inequalities by *sugar*'s front-end and then encoded by the basic encoding.

Optimized Encoding The basic encoding can generate redundant clauses for linear inequalities of size two. Consider $x + y \leq 7$ represented by a function term $\texttt{op}(\texttt{le}, \texttt{op}(\texttt{add}, \texttt{op}(\texttt{mul}, 1, \texttt{x}), \texttt{op}(\texttt{mul}, 1, \texttt{y})), 7)$, where $D(x) = D(y) = \{2, 3, 4, 5, 6\}$. The resulting propositional logic program is as follows.

```
                             :- not p(y,5). :- not p(x,5).
wsum(op(le,op(mul,1,x),2)) :- not p(y,4). :- wsum(op(le,op(mul,1,x),2)); not p(x,2).
wsum(op(le,op(mul,1,x),3)) :- not p(y,3). :- wsum(op(le,op(mul,1,x),3)); not p(x,3).
wsum(op(le,op(mul,1,x),4)) :- not p(y,2). :- wsum(op(le,op(mul,1,x),4)); not p(x,4).
```

```
1  :- wsum(op(le,op(add,op(mul,A1,V1),op(mul,A1,V2)),B)),
2          not p(V2,C) : A1 > 0 , C = @getSimpLE(D-1,V2);
3              p(V2,D) : A1 < 0;
4          not p(V1,E) : A1 > 0;
5              p(V1,E) : A1 < 0;
6          D = @getDomOpt(V2,A1,B-Sup,B-Inf), inf(op(mul,A1,V1),Inf),
7          sup(op(mul,A1,V1),Sup), E = @getLE(B-A1*D,A1,V1).
```

Listing 4.6: Encoding of $a_1 v_1 + a_2 v_2 \leq b$

```
1  alldiffArg(arg(V,A),1,V,A) :- alldiff(arg(arg(V,A),nil)).
2  alldiffArg(N,I+1,V,A) :- alldiffArg(N,I,_,arg(V,A)).

4  val(V,A) :- var(int,V,_), p(V,A), not p(V,@getSimpLE(A-1,V)), alldiffArg(_,_,V,_).

6  alldiffRange(A,LB,UB) :- alldiff(arg(A,nil)),
7          LB = #min {L,V : var(int,V,range(L,_)), alldiffArg(A,_,V,_)},
8          UB = #max {U,V : var(int,V,range(_,U)), alldiffArg(A,_,V,_)}.
```

Listing 4.7: Auxiliary predicates for the *alldifferent* constraints

The intermediate instances of `wsum`/1 are redundant and can be removed. This issue can be fixed by the *optimized encoding* which is an extension of the basic encoding by adding only the one rule of Listing 4.6 and by setting the constant `opt_binary` to 1. The rule of Listing 4.6 represents the special case of the first rule in Listing 4.4 for $\sum_{i=1}^{n} a_i v_i \leq b$ with $(n = 2)$ and does not generate any intermediate instances of `wsum`/1. However, we keep generating such intermediate instances for $n > 2$ because they can be shared by different linear inequalities and can be effective in reducing the number of nogoods. For the above example, the optimized encoding generates the following.

```
:- not p(y,5). :- not p(x,5).
:- not p(y,4); not p(x,2). :- not p(y,3); not p(x,3). :- not p(y,2); not p(x,4).
```

Alldifferent encodings In this part, we present different encodings for the alldifferent constraint. In Listing 4.7 we show all the auxiliary atoms that we need for the forthcoming encodings. In the first two lines we create the predicate `alldiffArg`(`n`, `i`, `v`, `a`) to collect all variables v and coefficients a for the alldifferent constraint n. The consecutive index i is used to access all variables in an order. Using these atoms, we create a direct encoding for all variables that occur in an alldifferent constraint in line 4. We use the atoms `val`(`v`, `a`) to denote that $v = a$. Furthermore, we create a maximum range of all variables that occur in an alldifferent constraint in lines $6-8$ using `alldiffRange`(`n`, `lb`, `ub`). We refer to these values as the range of the alldifferent constraint.

We now present different encodings for the alldifferent constraint. The simplest one, *alldiffA*, is presented in Listing 4.8. It ensures that two distinct variables that occur in the same alldifferent constraint do not take the same value using the atom

```
1  :- alldiffRange(CI,LB,UB), X = LB..UB, val(V1,X), val(V2,X),
2     alldiffArg(CI,_,V1,_), alldiffArg(CI,_,V2,_), V1 < V2.
```

Listing 4.8: Encoding A for alldifferent.

```
1  :- alldiffRange(CI,LB,UB),
2     X = LB..UB, 2{val(V,X) : alldiffArg(CI,_,V,_)}.
```

Listing 4.9: Encoding B for alldifferent.

$\texttt{val}(\texttt{v}, \texttt{a})$. This encoding produces a quadratic number of rules (in the number of variables occurring in the constraint) for every value x in the range of the constraint. The encoding *alldiffB* from Listing 4.9 does the same, this time using a cardinality constraint. Any value x in the range of the alldifferent constraint is not allowed to be can be assigned to at most one variable. This encoding uses exactly one rule for every value x.

The encoding *alldiffC* from Listing 4.10 is a more sophisticated one. For every variable $v_1, \ldots, v_n$ in an alldifferent constraint c we derive an atom $\texttt{seen}(\texttt{c}, \texttt{i} - 1, \texttt{x})$ that means that at least one variable v_j with $j \geq i$ has taken the value x. In line 3, we forbid that a variable v_j has the same value as a variable v_i with $i < j$.

The last encoding, *alldiffD*, in Listing 4.11 uses Hall intervals [21], and does not use the direct encoding from Listing 4.7. In the first line, we compute all Hall intervals of maximum size H ($\{[s, s+h] \mid s \in [lb, ub-h], 0 \leq h \leq H\}$), for the interval $[lb, ub]$ of an alldifferent constraint. Next, we create a fresh integer variable with the domain $\{0, 1\}$ for every Hall interval in lines $3-4$. In the lines $6-8$ we create a reified constraint $hallvar(v, lb, ub) \Leftrightarrow v \geq lb \wedge v \leq ub$ to ensure that a Hall interval variable is one, whenever its variable v is inside the Hall interval and zero otherwise. The last four rules create a constraint for each Hall interval $[l, u]$ that ensures that the sum of all hallvar variables is less than $u - l$, practically ensuring that no two variables have the same value. With this encoding we can achieve stronger propagation on alldifferent constraints using unit propagation at the cost of producing more rules. We can control this behaviour with the hallsize $\texttt{hall} = H$, having better propagation but also producing more constraints giving higher values of H. Note that hall propagation can not be done efficiently using unit propagation [22].

Objective Functions To minimize an integer variable, the `objective`/2 predicate is used in Listing 4.12. For every variable x we want to minimize we add $\texttt{p}(x, d)$ with a weight of $d - prev(d, x)$ to an ASP minimize statement for every $d \in D(x), d \neq lb(x)$. The Lua function $\texttt{getSimpLE}(d-1, v)$ simply returns $prev(d, v)$. So for minimizing x where $D(x) = \{0, 1, 5, 6\}$ the statement grounds to

```
#minimize{(1-0),x : p(x,1); (5-1),x : p(x,5); (6-5),x : p(x,6)}.
```

```
1 seen(CI,I-1,X) :- alldiffArg(CI,I,V,_), 1 < I, val(V,X).
2 seen(CI,I-1,X) :- seen(CI,I,X), 1 < I.
3 :- alldiffArg(CI,I,V,_), val(V,X), seen(CI,I,X).
```

Listing 4.10: Encoding C for alldifferent.

```
1  hInterval(CI,S,E) :- alldiffRange(CI,LB,UB), H=0..hall, S=LB..UB-H, E=S+H.

3  hVar(V,S,E) :- hInterval(CI,S,E), alldiffArg(CI,_,V,_).
4  var(int,hVar(V,S,E),range(0,1)) :- hVar(V,S,E).

6  :- p(hVar(V,S,E),0), hVar(V,S,E), p(V,@getSimpLE(E,V)), not p(V,@getSimpLE(S-1,V)).
7  :- not p(hVar(V,S,E),0), hVar(V,S,E), p(V,@getSimpLE(S-1,V)).
8  :- not p(hVar(V,S,E),0), hVar(V,S,E), not p(V,@getSimpLE(E,V)).

10 lastArg(CI,X) :- alldiffArg(CI,X,_,_), not alldiffArg(CI,X+1,_,_).
11 hCtor(CI,S,E,1,op(mul,1,hVar(V,S,E))) :- alldiffArg(CI,1,V,_), hInterval(CI,S,E).
12 hCtor(CI,S,E,N+1,op(add,Old,op(mul,1,hVar(V,S,E)))) :- hCtor(CI,S,E,N,Old),
13                                   alldiffArg(CI,N+1,V,_), hInterval(CI,S,E).
14 constraint(hc(CI,S,E),op(le,C,E-S+1)) :- hCtor(CI,S,E,Last,C), lastArg(CI,Last).
```

Listing 4.11: Encoding D for alldifferent.

4.4.1 Encoding Constraint Answer Set Programs

We developed an encoding for linear constraints and also the alldifferent constraint. This allows us to solve CSPs over finite integers just as the *sugar* system. We now elaborate on an extension of this encoding. Although *aspartame* was not specifically designed for CASP, it can be used to express CASP problems quite easily. By simply using the `wsum/1` atom in a problem encoding combined with the encodings for *aspartame*, we can use constraints in our ASP problem specification. As an example, we reconsider the two dimensional strip packing problem from Section 3.7.

Two Dimensional Strip Packing To encode this problem, the set of rectangles is represented by a set of facts `r(I,W,H)`. Each `I` identifies a rectangle with width `W` and height `H`. The task is to fit all into a container of width `w` and height `ub`. This time, we minimize the height of the container. Based on the problem description in [11], Listing 4.13 encodes the problem using `var/3` for describing integer variables and their domain, `le(x,c,y)` intending to express $x + c \leq y$ and `objective/2` to minimize the total height. Whenever an arrangement of squares is chosen in lines $6-7$ and `le(x,c,y)` holds, we derive a `wsum/1` atom in line 10 which ensures that the constraint $x + c \leq y$ holds using the aspartame encoding.

This method has some problems related to the input language of *aspartame*. The fact format was designed to be automatically generated from a CSP encoding. It is not convenient to use constraints in the body of a rule (as they cannot easily be made external) nor to create (n)-ary linear constraints[8] because of the nested structure that was chosen for performance reasons.

[8] Constraints whose length depends dynamically on the input instance.

```
1  #minimize{  (D-P),V,D : p(V,D), objective(minimize,V), P=@getSimpLE(D-1,V) }.
```

Listing 4.12: Encoding objective functions.

```
1  var(int,   x(I), range(0,w-W))  :- r(I,W,H).
2  var(int,   y(I), range(0,ub-H)) :- r(I,W,H).
3  var(int, height, range(0,ub)).
4  objective(minimize, height).

6  1 { le(x(I),WI,x(J)) ; le(x(J),WJ,x(I)) ; le(y(I),HI,y(J)) ; le(y(J),HJ,y(I)) } :-
7          r(I,WI,HI), r(J,WJ,HJ), I < J.
8  le(y(I),H,height) :- r(I,W,H).

10 wsum(op(le,op(add,op(mul,1,X),op(mul,-1,Y)),-C)) :- le(X,C,Y).
```

Listing 4.13: Encoding of 2sp problems

4.5 Evaluation

As mentioned, *aspartame* re-uses *sugar*'s front-end for parsing and normalizing CSPs. Hence, it accepts the same input formats, viz. XCSP[9] and *sugar*'s native CSP format.[10] For this, we implemented an output hook for *sugar* that provides us with the resulting CSP instance in *aspartame*'s fact format. This format can also be used for directly representing linear arithmetic constraints within standard ASP encodings used for CASP. In both cases, the resulting facts are then used for grounding a dedicated ASP encoding (via the ASP grounder *gringo*). In turn, the resulting propositional logic program is passed to the ASP solver *clasp* that returns an assignment, representing a solution to the original CSP instance.

Our empirical analysis considers all instances of `GLOBAL` categories in the 2009 CSP Competition.[9] We ran them on a cluster of Linux machines equipped with dual Xeon E5520 quad-core 2.26GHz processors and 48GB RAM. We separated grounding and solving times, and imposed on each a limit of 1800s and 16GB. While we count a grounding timeout as 1800s, we penalize unsuccessful solving with 1800s if either solving or grounding does not finish in time

At first, we analyze the difference between the *basic encoding* and its refinements from Section 4.4. To this end, Table 4.1 contrasts the results obtained from different ASP encodings as well as *sugar* (2.2.1). The name of the benchmark class and the number of instances is given in the first column. In each setting, the *trans* column shows the average time used for translating CSP problems into their final propositional format. For this purpose, *aspartame* uses *gringo* (4.5), while *sugar* uses a dedicated implementation resulting in a CNF in DIMACS format. Analogously, the *solve* column gives the average time for each benchmark class, showing the number of translation (to^t) and total timeouts (to). In all cases, we use *clasp* 3.1.1 as back-end ASP or SAT solver, respectively, in its ASP default configuration `tweety`. Comparing the

[9] `http://www.cril.univ-artois.fr/CPAI09`
[10] `http://bach.istc.kobe-u.ac.jp/sugar/package/current/docs/syntax.html`

instances	basic trans	basic solve	basic to^t	basic to	optimized trans	optimized solve	optimized to^t	optimized to	optimizednosplit trans	optimizednosplit solve	optimizednosplit to^t	optimizednosplit to	sugar trans	sugar solve	sugar to^t	sugar to
CabinetStart(40)	1800	1800	40	40	53	20	0	0	8	2	0	0	4	12	0	0
QG3(7)	2	515	0	2	2	515	0	2	2	515	0	2	2	514	0	2
QG4(7)	2	291	0	1	2	278	0	1	2	278	0	1	2	269	0	1
QG5(7)	1	168	0	0	1	71	0	0	1	71	0	0	1	60	0	0
QG6(7)	3	257	0	1	4	257	0	1	4	257	0	1	2	257	0	1
QG7	3	263	0	1	4	259	0	1	4	259	0	1	2	258	0	1
Squares(37)	49	385	0	5	162	229	0	4	33	278	0	4	4	271	0	4
SquaresUnsat(37)	49	745	0	15	160	683	0	13	33	728	0	13	4	660	0	13
Bibd1011(6)	19	5	0	0	30	3	0	0	15	6	0	0	9	6	0	0
Bibd1213(7)	28	13	0	0	42	20	0	0	20	15	0	0	7	2	0	0
Bibd6(10)	5	1	0	0	8	1	0	0	4	1	0	0	3	1	0	0
Bibd7(14)	6	1	0	0	9	1	0	0	5	1	0	0	4	1	0	0
Bibd8(7)	9	14	0	0	14	3	0	0	7	11	0	0	4	2	0	0
Bibd9(10)	13	3	0	0	20	2	0	0	9	3	0	0	6	4	0	0
BibdVariousK(29)	17	343	0	4	23	298	0	4	14	324	0	5	6	266	0	3
bqwh15106(10)	0	0	0	0	1	0	0	0	1	0	0	0	1	0	0	0
bqwh18141(10)	1	0	0	0	1	0	0	0	1	0	0	0	1	0	0	0
Cjss(10)	61	1084	0	6	97	1085	0	6	86	1091	0	6	24	944	0	5
Compet02(20)	199	67	0	0	1165	200	2	2	1164	200	2	2	22	9	0	0
Compet08(16)	17	146	0	1	90	16	0	0	91	16	0	0	73	463	0	4
CostasArray(11)	3	577	0	3	15	381	0	2	16	514	0	3	2	362	0	2
LatinSquare(10)	1	180	0	1	2	180	0	1	2	180	0	1	1	180	0	1
MagicSquare(18)	1057	1179	10	11	1208	1103	11	11	1444	1400	14	14	629	756	6	7
Medium(5)	305	117	0	0	1717	1446	4	4	1721	1446	4	4	31	10	0	0
Nengfa(3)	113	19	0	0	777	5	0	0	770	5	0	0	4	6	0	0
pigeons(19)	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
Pseudo(100)	122	466	5	23	142	482	7	26	12	382	0	18	95	488	5	26
Rcpsp(39)	0	0	0	0	1	0	0	0	0	0	0	0	1	0	0	0
RcpspTighter(39)	0	0	0	0	1	0	0	0	0	0	0	0	1	0	0	0
Small(5)	14	2	0	0	87	1	0	0	87	1	0	0	4	1	0	0
total	210	412	55	114	163	273	24	78	124	274	20	75	44	252	11	70

Table 4.1: Experiments comparing different encodings with sugar.

basic encoding with the *optimized encoding*, we observe that the latter significantly reduces both solving and grounding timeouts (mainly due to the *CabinetStart* class). Next, we investigate the impact of the recursive structure of our encodings. For this, we disabled splitting of linear constraints within *sugar*'s translation. This usually leads to an exponential increase in the number of clauses for *sugar*. The results are given in column *optimizednosplit* of Table 4.1. In fact, disabled splitting performs as good as the optimized encoding with splitting. In some cases, it even improves performance. Since splitting constraints the right way usually depends heavily on heuristics, our recursive translation offers a heuristic-independent solution to this problem. Finally, although *aspartame* and *sugar* are at eye height regarding solving time and timeouts, *aspartame* falls short by an order of magnitude when it comes to translating CSPs into propositional format. Here the dedicated implementation of

sugar[11] clearly outperforms the grounder-based approach of *aspartame*. On the other hand, our declarative approach allows us to easily modify and thus experiment with different encodings.

This flexibility was extremely useful when elaborating upon different encodings. While for the benchmarks used in Table 4.1 the alldifferent constraints are translated to linear constraints with the help of *sugar*, we now handle them by an encoding. To this end, Table 4.2 compares four alternative encodings for handling alldifferent

	Encoding A				Encoding B				Encoding C				Encoding D				Encoding $B^{\dagger}$			
instances	trans	solve	to^{t}	to	trans	solve	to^{t}	to	trans	solve	to^{t}	to	trans	solve	to^{t}	to	trans	solve	to^{t}	to
CabinetStart(40)	54	20	0	0	53	20	0	0	53	20	0	0	65	23	0	0	8	2	0	0
QG3(7)	2	515	0	2	2	515	0	2	2	515	0	2	3	515	0	2	2	515	0	2
QG4(7)	2	276	0	1	2	278	0	1	2	283	0	1	3	290	0	1	2	278	0	1
QG5(7)	1	56	0	0	1	68	0	0	1	60	0	0	3	149	0	0	1	68	0	0
QG6(7)	4	257	0	1	4	257	0	1	4	257	0	1	6	258	0	1	4	257	0	1
QG7	4	259	0	1	4	260	0	1	4	259	0	1	4	261	0	1	4	260	0	1
Squares(37)	161	229	0	4	161	230	0	4	161	230	0	4	166	232	0	4	33	281	0	4
SquaresUnsat(37)	158	683	0	13	158	682	0	13	158	682	0	13	168	701	0	13	32	726	0	13
Bibd1011(6)	30	3	0	0	30	3	0	0	31	3	0	0	32	3	0	0	15	6	0	0
Bibd1213(7)	42	20	0	0	42	20	0	0	43	20	0	0	43	24	0	0	20	15	0	0
Bibd6(10)	8	1	0	0	8	1	0	0	8	1	0	0	14	1	0	0	4	1	0	0
Bibd7(14)	9	1	0	0	9	1	0	0	9	1	0	0	9	1	0	0	5	1	0	0
Bibd8(7)	14	3	0	0	14	3	0	0	14	3	0	0	14	3	0	0	7	11	0	0
Bibd9(10)	20	2	0	0	20	2	0	0	20	2	0	0	21	3	0	0	9	3	0	0
BibdVariousK(29)	23	298	0	4	23	298	0	4	23	298	0	4	26	300	0	4	14	324	0	5
bqwh15106(10)	0	0	0	0	0	0	0	0	0	0	0	0	4	1	0	0	0	0	0	0
bqwh18141(10)	0	0	0	0	0	0	0	0	0	0	0	0	5	1	0	0	0	0	0	0
Cjss(10)	99	1085	0	6	99	1085	0	6	99	1085	0	6	97	1085	0	6	87	1091	0	6
Compet02(20)	834	22	0	0	836	18	0	0	840	21	0	0	1095	268	2	2	842	19	0	0
Compet08(16)	236	463	0	4	232	455	0	4	230	475	0	4	633	1498	0	13	233	455	0	4
CostasArray(11)	5	503	0	2	5	349	0	2	5	494	0	3	10	517	0	3	6	343	0	2
LatinSquare(10)	0	180	0	1	0	180	0	1	0	180	0	1	2	180	0	1	0	180	0	1
MagicSquare(18)	1192	1107	11	11	1192	1103	11	11	1191	1104	11	11	1196	1106	11	11	1442	1401	14	14
Medium(5)	1510	36	0	0	1499	34	0	0	1503	35	0	0	1700	1458	4	4	1507	34	0	0
Nengfa(3)	10	2	0	0	9	1	0	0	9	2	0	0	67	117	0	0	9	1	0	0
pigeons(19)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Pseudo(100)	142	482	7	26	142	482	7	26	142	482	7	26	142	483	7	26	12	382	0	18
Rcpsp(39)	1	0	0	0	1	0	0	0	1	0	0	0	1	0	0	0	0	0	0	0
RcpspTighter(39)	1	0	0	0	1	0	0	0	1	0	0	0	1	0	0	0	0	0	0	0
Small(5)	62	1	0	0	62	1	0	0	61	1	0	0	92	8	0	0	62	1	0	0
total	148	269	18	76	148	266	18	76	148	269	18	77	174	326	24	92	110	264	14	72

Table 4.2: Experiments comparing different encodings for alldifferent.

presented above. Our experiments show however, that simple translations using binary inequalities like A and B are as good as more complex ones like C and even outperform more sophisticated ones as D. The last column shows the combination of non-splitting linear constraints (in *sugar*) and handling the alldifferent constraint with translation

[11] The timeouts of *sugar* during translation are always due to insufficient memory.

B. This is currently the best performing combination of encodings and constitutes the default setting of *aspartame* (2.0.0).[12]

4.6 Conclusion

As regards pure CP solving, our approach can be seen as a first-order alternative to SAT-based approaches like *sugar* [116]. The resulting system *aspartame* relies on high-level ASP encodings and delegates both the grounding and solving tasks to general-purpose ASP systems. Furthermore, these encodings can be used as a library for solving CSPs as part of an encompassing logic program, as it is done in the framework of CASP.

Although the performance of the underlying SAT solver is crucial, the SAT encoding plays an equally important role [104]. Among them, we find the direct [121], support [67], logarithmic [121], order [31, 116], and compact order [117] encoding. The order encoding showed good performance for a wide range of CSPs [31, 93, 99, 110]. In fact, the SAT-based CP solver *sugar* won the `GLOBAL` category at the 2008 and 2009 CP solver competitions [79]. Also, the SAT-based CP solver BEE [92] and the CLP system B-Prolog [124] use this encoding. In fact, the order encoding provides a compact translation of arithmetic constraints, while also maintaining bounds consistency by unit propagation. Interestingly, it has been shown that the order encoding is the only existing SAT encoding that can reduce tractable CSP to tractable SAT [103].

We have focused our empirical comparison on the SAT-based CP solver *sugar*. This is motivated as follows. First, *sugar* follows a rather similar approach as *aspartame*. Second, we could compare *sugar* and *aspartame* in a uniform setting by fixing the back-end solver to *clasp*. Furthermore, *sugar* has been extensively compared to genuine CP solvers in the literature, which also puts *aspartame*'s performance in context. In particular, the empirical success of *sugar* suggests that CP solving via state of the art Boolean constraint solvers should be the option of choice whenever the size required for a Boolean representation can be afforded. Although *aspartame* does not fully match the performance of *sugar* from a global perspective, the picture is fragmented and leaves room for further improvements, especially for the translation process.

Before we compare *aspartame* with our previously proposed CASP approach, we extend the translational approach by making it "online". In fact, instead of translating all constraints a priori, we aim at translating them on the fly during the solving process. This approach combines the advantages of the order encoding with the extended CDCL approach from Section 3.4 and is to be presented in the next chapter.

[12] The system is available at `https://potassco.org/labs/2016/09/20/aspartame.html`

Chapter 5

Lazy Nogood and Variable Generation

In this chapter we develop an approach for handling CASP using lazy nogood and variable generation techniques. These are particularly efficient for solving linear constraints over variables with huge domains. The main part of this chapter was already published in [13]. It is structured as follows:

- We first characterize CASP in terms of nogoods and propagators. This forms the foundation for the lazy nogood learning approach.
- Next, we describe the design of our new system *clingcon* 3, including its architecture and its input language.
- The algorithms from Section 3.4 are extended to use constraint propagators and lazy variable generation. In combination with the approach presented in Chapter 4, a partial translation of the problem can be achieved.
- We now detail some of the distinguishing features of our system. These features include preprocessing and runtime features adjusted to fit our approach.
- To assess the usefulness of our approach, we start with an extensive evaluation of its features. Then, we compare our system to state of the art CP solvers, using benchmarks from the *minizinc* competition 2016. Finally, CASP benchmarks are evaluated and our system is confronted with other modern CASP systems, including the *aspartame* and *clingcon* 2 system presented before.

5.1 Constraint Stable Models in Terms of Propagators

The basic idea of lazy nogood generation is to make the nogoods in $\Phi(\mathcal{V}, D)$ and $\Psi(\mathcal{C})$ (Equations (4.3) and (4.5) from Section 4.3) only explicit on demand. This is done by propagators corresponding to the respective set of nogoods. A popular example of

this is the unfounded-set-check algorithm in ASP solvers that only makes the nogoods in Λ_P (3.6) explicit on demand.

Following [44], a *propagator* for a set Θ of nogoods is a function Π_Θ mapping a Boolean assignment $\mathbf{B}$ to a subset of Θ such that for each total assignment $\mathbf{B}$: if $\delta \subseteq \mathbf{B}$ for some $\delta \in \Theta$, then $\delta' \subseteq \mathbf{B}$ for some $\delta' \in \Pi_\Theta(\mathbf{B})$. That is, whenever there is a nogood in Θ violated by an assignment $\mathbf{B}$, then $\Pi_\Theta(\mathbf{B})$ yields a violated nogood, too. A propagator Π_Θ is *conflict optimal*, if for all partial assignments $\mathbf{B}$, the violation of a nogood in Θ by $\mathbf{B}$ implies that some nogood in $\Pi_\Theta(\mathbf{B})$ is violated by $\mathbf{B}$. Π_Θ is *inference optimal*, if it is conflict optimal and $\Pi_\Theta(\mathbf{B})$ contains all unit nogoods of Θ wrt. $\mathbf{B}$.

We obtain the following extension of Theorem 4.3.2.

Theorem 5.1.1

Let P be a constraint logic program over regular atoms $\mathcal{A}$ and constraint atoms $\mathcal{C}$ associated with the CSP $(\mathcal{V}, D, C)$ and let Π_Θ be a propagator for $\Theta = \Lambda_P^{\mathcal{C}}$, $\Psi(\mathcal{C})$, and $\Phi(\mathcal{V}, D)$, respectively.
Then, $\mathbf{B}$ is a solution of the set

$$\Delta_P^{\mathcal{C}} \cup \Lambda_P^{\mathcal{C}} \cup \Psi(\mathcal{C}) \cup \Phi(\mathcal{V}, D)$$

of nogoods iff $\mathbf{B}$ is a solution of the set

$$\Delta_P^{\mathcal{C}} \cup \Pi_{\Lambda_P^{\mathcal{C}}}(\mathbf{B}) \cup \Pi_{\Psi(\mathcal{C})}(\mathbf{B}) \cup \Pi_{\Phi(\mathcal{V},D)}(\mathbf{B})$$

of nogoods.

This theorem tells us that the nogoods in $\Psi(\mathcal{C})$, $\Phi(\mathcal{V}, D)$, and $\Lambda_P^{\mathcal{C}}$ must not be explicitly represented but can be computed by corresponding propagators Π_Θ that add them lazily on demand.

Proof 5.1.1 *This theorem follows directly from the definition of a propagator, as a propagator* $\Pi_\Theta(\mathbf{B})$ *always contains a conflicting nogood iff* Θ *contains a conflicting nogood wrt. a complete assignment* $\mathbf{B}$*.*

To relax the restrictions imposed by this theorem, the idea is to compile out a subset of constraints and variables of the CSP while leaving the others subject to lazy nogood generation. This is captured by the following corollary to Theorem 5.1.1.

Corollary 5.1.1 *Let* P *be a constraint logic program over regular atoms* $\mathcal{A}$ *and constraint atoms* $\mathcal{C}$ *associated with the CSP* $(\mathcal{V}, D, C)$ *and let* Π_Θ *be a propagator for* $\Theta = \Lambda_P^{\mathcal{C}}$, $\Psi(\mathcal{C} \setminus \mathcal{C}')$*, and* $\Phi(\mathcal{V} \setminus \mathcal{V}', D \setminus D')$*, respectively, for arbitrary subsets* $\mathcal{C}' \subseteq \mathcal{C}$, $\mathcal{V}' \subseteq \mathcal{V}$*, and* $D' \subseteq D$*.*
Then, $\mathbf{B}$ *is a solution of the set*

$$\Delta_P^{\mathcal{C}} \cup \Lambda_P^{\mathcal{C}} \cup \Psi(\mathcal{C}) \cup \Phi(\mathcal{V}, D)$$

of nogoods iff $\mathbf{B}$ *is a solution of the set*

$$\Delta_P^{\mathcal{C}} \cup \Psi(\mathcal{C}') \cup \Phi(\mathcal{V}', D') \cup \Pi_{\Lambda_P^{\mathcal{C}}}(\mathbf{B}) \cup \Pi_{\Psi(\mathcal{C}\setminus\mathcal{C}')}(\mathbf{B}) \cup \Pi_{\Phi(\mathcal{V}\setminus\mathcal{V}', D\setminus D')}(\mathbf{B})$$

of nogoods.

This correspondence nicely reflects upon the basic idea of our approach. While the entire set of loop nogoods $\Lambda_P^{\mathcal{C}}$ is handled by the unfounded set propagator $\Pi_{\Lambda_P^{\mathcal{C}}}$ as usual, the ones capturing the CSP is divided among the explicated nogoods in $\Psi(\mathcal{C}') \cup \Phi(\mathcal{V}', D')$ and the implicit ones handled by the propagators $\Pi_{\Psi(\mathcal{C}\setminus\mathcal{C}')}$ and $\Pi_{\Phi(\mathcal{V}\setminus\mathcal{V}', D\setminus D')}$. Note that variables and domain elements are often only dealt with implicitly through their induced order atoms in $\mathcal{O}$.

5.2 System Design

We now present the *clingcon* 3 system, a lazy nogood and variable generating CASP solver. It combines the previously presented approaches by extending the CDCL approach with a propagator that uses the order encoding. We also propose a new CASP input language based on the general theory language framework [53] provided by *gringo*.

5.2.1 Architecture

clingcon 3 is an extension of the ASP system *clingo* 5, which itself relies on the grounder *gringo* and the solver *clasp*. The architecture of *clingcon* 3 is given in Figure 5-1. More precisely, *clingcon* uses *gringo*'s capabilities to specify and process

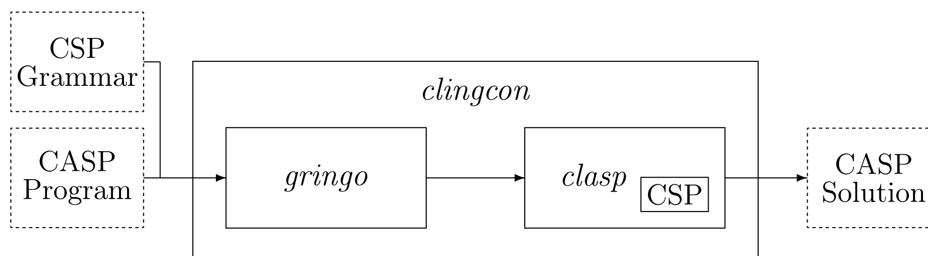

Figure 5-1: Architecture of *clingcon* 3.

customized theory languages. For this, it is sufficient to supply a grammar fixing the syntax of constraint-related expressions. As detailed in Section 5.2.2, this allows us to express linear constraints similar to standard ASP aggregates by using first-order variables. Unlike this, *clingcon* extends *clasp* in several ways to accommodate its lazy approach to constraint solving. First, *clasp*'s preprocessing capabilities are extended to integrate linear constraints. Second, dedicated propagators are added to account for lazy constraint propagation. Both extensions are detailed in Section 5.3. And finally,

a special output module was created to integrate CSP solutions. Notably, *clingcon* 3 pursues a lazy yet two-fold approach to constraint solving that allows for making a part of the nogoods in $\Phi(\mathcal{V}, D) \cup \Psi(\mathcal{C})$ explicit during preprocessing, while leaving the remaining constraints implicit and the creation of corresponding nogoods subject to the constraint propagator. In this way, a part of the CSP can be put right up front under the influence of CDCL-based search. All other constraints are only turned into nogoods when needed. Accordingly, only a limited subset of order atoms from $\mathcal{O}$ (the atoms that represent the integer variables) must be introduced at the beginning; further ones are only created if they are needed upon the addition of new nogoods. This is also called lazy variable generation.

It is worth mentioning that both the grounding and the solving component of *clingcon* can also be used separately via *clingo*'s option '`--mode`'. That is, the same result as with *clingcon* is obtained by passing the output of '`clingcon --mode=gringo`' to '`clingcon --mode=clasp`'. The intermediate result of grounding a CASP program is expressed in the *aspif* format [54] that accommodates both the regular ASP part of the program as well as its constraint-based extension. This modular design allows others to take advantage of *clingcon*'s infrastructure for their own CASP solvers. Also, external front ends can be used for generating ground CASP programs; e.g. the *flatzinc* translator used in Section 5.4.

Finally, extra effort was taken to transfer *clasp* specific features to *clingcon*'s solving component. This includes multi-threading [63], unsatisfiable core techniques [2], multi-criteria optimization [55], domain-specific heuristics [61], multi-shot solving [57, 59], and *clasp*'s reasoning modes like enumeration, intersection and union of models. Vocabulary-sensitive reasoning modes like projective enumeration and domain-specific heuristics can be used via auxiliary atoms.

5.2.2 Language

As mentioned, the treatment of the extended input language of CASP programs can be mapped onto *gringo*'s theory language capabilities [53]. For this, it is sufficient to supply a corresponding grammar fixing the syntax of the language extension. The one used for *clingcon* is given in Listing 5.1. The grammar is named `csp` and consists of two parts, one defining theory terms in lines $2-27$ and another defining theory atoms in lines $29-33$. All regular terms are implicitly included in the respective theory terms. These are then used to represent constraint-related expressions that are turned by grounding into linear constraint atoms using predicate `&sum`, domain restrictions using predicate `&dom`, directives `&show` and `&minimize`, and the predefined global constraint `&distinct`.[1]

Before delving into further details, let us illustrate the resulting syntax by the CASP program for two dimensional strip packing given in Listing 5.2, already introduced in Section 3.7 and Section 4.4.1. Given a set of rectangles, each represented by a fact `r(I,W,H)` where `I` identifies a rectangle with width `W` and height `H`, the task is to fit all into a container of width `w` and height `ub` while minimizing the needed height

[1] Which is equivalent to the alldifferent constraint from Section 4.4.

```
#theory csp {
    dom_term {
    + : 5, unary;
    - : 5, unary;
    .. : 1, binary, left;
    * : 4, binary, left;
    + : 3, binary, left;
    - : 3, binary, left
    };
    linear_term {
    + : 5, unary;
    - : 5, unary;
    * : 4, binary, left;
    + : 3, binary, left;
    - : 3, binary, left
    };
    show_term {
    / : 1, binary, left
    };
    minimize_term {
    + : 5, unary;
    - : 5, unary;
    * : 4, binary, left;
    + : 3, binary, left;
    - : 3, binary, left;
    @ : 0, binary, left
    };

    &dom/0 : dom_term, {=}, linear_term, any;
    &sum/0 : linear_term, {<=,=,>=,<,>,!=}, linear_term, any;
    &distinct/0 : linear_term, any;
    &show/0 : show_term, directive;
    &minimize/0 : minimize_term, directive
}.
```

Listing 5.1: Language Syntax

```
&dom{0..w-W}  = x(I) :- r(I,W,H).
&dom{0..ub-H} = y(I) :- r(I,W,H).

1 { le(x(I),WI,x(J));
    le(x(J),WJ,x(I));
    le(y(I),HI,y(J));
    le(y(J),HJ,y(I)) } :- r(I,WI,HI), r(J,WJ,HJ), I < J.

&sum{VI; C} <= VJ :- le(VI,C,VJ).

&dom{0..ub} = height.
&sum{y(I); H} <= height :- r(I,W,H).
&minimize {height}.
&show {height}.
```

Listing 5.2: Two Dimensional Strip Packing

```
1   r(a,5,2). r(b,2,3). r(c,2,2).

3   &dom{0..(6-5)}  = x(a). &dom{0..(6-2)}  = x(b). &dom{0..(6-2)}  = x(c).
4   &dom{0..(10-2)} = y(a). &dom{0..(10-3)} = y(b). &dom{0..(10-2)} = y(c).

6   1 <= { le(x(a),5,x(b)); le(x(b),2,x(a));
7          le(y(a),2,y(b)); le(y(b),3,y(a)) }.
8   1 <= { le(x(a),5,x(c)); le(x(c),2,x(a));
9          le(y(a),2,y(c)); le(y(c),2,y(a)) }.
10  1 <= { le(x(b),2,x(c)); le(x(c),2,x(b));
11         le(y(b),3,y(c)); le(y(c),2,y(b)) }.

13  &sum{ x(a); 5 } <= x(b) :- le(x(a),5,x(b)).
14  &sum{ x(b); 2 } <= x(a) :- le(x(b),2,x(a)).
15  &sum{ y(a); 2 } <= y(b) :- le(y(a),2,y(b)).
16  &sum{ y(b); 3 } <= y(a) :- le(y(b),3,y(a)).
17  &sum{ x(a); 5 } <= x(c) :- le(x(a),5,x(c)).
18  &sum{ x(c); 2 } <= x(a) :- le(x(c),2,x(a)).
19  &sum{ y(a); 2 } <= y(c) :- le(y(a),2,y(c)).
20  &sum{ y(c); 2 } <= y(a) :- le(y(c),2,y(a)).
21  &sum{ x(b); 2 } <= x(c) :- le(x(b),2,x(c)).
22  &sum{ x(c); 2 } <= x(b) :- le(x(c),2,x(b)).
23  &sum{ y(b); 3 } <= y(c) :- le(y(b),3,y(c)).
24  &sum{ y(c); 2 } <= y(b) :- le(y(c),2,y(b)).

26  &dom{ 0..10 } = height.

28  &sum{ y(a); 2 } <= height.
29  &sum{ y(b); 3 } <= height.
30  &sum{ y(c); 2 } <= height.

32  &minimize{ height }.
33  &show{ height }.
```

Listing 5.3: Two Dimensional Strip Packing Example

of the container. The first two lines of Listing 5.2 restrict the domain of the left lower corner of each rectangle `I`. The respective instantiations of `x(I)` and `y(I)` yield constraint variables denoting the `x` and `y` coordinate of `I`, respectively. Note that in both lines the consecutive dots '`..`' construct a theory term '`0..w-W`' and '`0..ub-H`' once `w` and `ub` are replaced, respectively. The choice rule in lines $4-7$ lets us choose among all combinations of two rectangles, that is, which one is left, right, below or above. At least one of these relations must hold so that no two rectangles overlap. Atoms of form `le(VI,C,VJ)` indicate that coordinate `VI`+`C` must be less than or equal to `VJ`. This property is enforced by the linear constraint in line 9. Finally, to minimize the overall height of (stacked) rectangles, we introduce the variable `height`. This variable's value has to be greater than or equal to the `y` coordinate of any rectangle `I` plus the rectangle's height `H`. This ensures that `height` is greater or equal to the height of the highest rectangle. Finally, `height` is minimized in line 13.

Now, if we take the three rectangles `r(a,5,2)`, `r(b,2,3)`, `r(c,2,2)` along with `ub=10` and `w=6`, we obtain the ground program in Listing 5.3. The domains of the constraint variables giving the x- and y-coordinates are delineated in line 3 and 4. Note that in contrast to regular ASP the grounder leaves terms with the theory symbol `..` intact. The orientation of each pair of rectangles is chosen in lines $6-11$. If for

example `le(x(c),2,x(b))` becomes true, that is, rectangle c is left of b, then the constraint $x(c) + 2 \leq x(b)$ is enforced in line 22. After setting the domain for the `height` variable in line 26, we restrict it to be greater or equal to the top y-coordinate of all rectangles in lines 28 − 30. Line 32 enforces the minimization of this variable. A solution with minimal `height` consists of the regular atoms `le(y(b),3,y(a))`, `le(y(c),2,y(a))`, and `le(x(c),2,x(b))`, and the constraint variable assignment $\{\texttt{height} \mapsto 5, \texttt{y(c)} \mapsto 1, \texttt{x(c)} \mapsto 2, \texttt{x(a)} \mapsto 1, \texttt{x(b)} \mapsto 4, \texttt{y(a)} \mapsto 3, \texttt{y(b)} \mapsto 0\}$. Of course other minimal configurations exist.

We have seen above how seamlessly theory atoms capturing constraint-related expressions can be used in logic programs. We detail below the five distinct atom types featured by *clingcon* and refer the interested reader for a general introduction to theory terms and atoms to [53].

Actual constraints are represented by the theory atoms `&dom`, `&sum`, and `&distinct`. All three can occur in the head and body of rules, as indicated by `any` in lines 29 − 31 in Listing 5.1. We discuss below their admissible format after grounding. In the following, a linear expression is a sum of integers, products of integers, or products of an integer and a constraint variable.

Domain constraints are of form $\texttt{\&dom}\{d_1; \dots; d_n\} = t$ where

- each d_i is a domain term of form
  - u or
  - $v..w$

  where u, v, w are constraint variable free linear expressions[2] and
- t is a linear expression containing exactly one constraint variable.

Then, the previous expression represents the constraint $t \in \bigcup_{i=1}^{n} [\![d_i]\!]$, where $[\![d]\!] = \{u\}$ if $d = u$, $[\![d]\!] = \{v, \dots, w\}$ if $d = v..w$, and undefined otherwise.

This constraint can be used to set the domain of variables where even non-contiguous domains can be used by having $n > 1$. For example $\texttt{\&dom}\{1..3; 5\} = x$ represents the constraint $x \in \{1, \dots, 3\} \cup \{5\}$.

Linear constraints are of form $\texttt{\&sum}\{t_1; \dots; t_n\} \circ t_{n+1}$ where

- each t_i is a linear expression containing at most one constraint variable, and
- $\circ$ is one of the operators `<=,=,>=,<,>,!=`.

This expression represents the linear constraint $(t_1 + \cdots + t_n) \circ t_{n+1}$, which can be translated into one or two linear constraints as described in Section 4.1.

Distinct constraints are of form $\texttt{\&distinct}\{t_1; \dots; t_n\}$ where each t_i is a linear expression containing at most one constraint variable. Such an expression stands for the constraints $t_i \neq t_j$ for $0 \leq i < j \leq n$.

[2] Linear expressions that evaluate to a constant value.

The distinct (or alldifferent) constraint is one of the most common global constraints in CP. We use it to show how global constraints can be incorporated into the language.

The two remaining theory atoms provide directives, similar to their regular counterparts.

Output directives are of form `&show`$\{s_1; \dots; s_n\}$ where each s_i is a show term of form

- f/m where f is a function symbol and m a positive integer or
- t, where t is a constraint variable.

While the latter adds variable t to the list of output variables, the first one adds all variables of the form $f(t_1, \dots, t_m)$ (where t_i is a term) as output variables. For all constraint stable models, the value of the output variables is shown in a solution.

Minimize directives are of form `&minimize`$\{m_1; \dots; m_n\}$ where each m_i is a minimize term of form $t_i@l_i$ and t_i being a linear expression with at most one constraint variable. Since we support multi-objective optimization, l_i is an integer stating the priority level. Whenever $@l_i$ is omitted, l_i assumed to be zero. Priorities allow for representing lexicographically ordered minimization objectives. As in regular ASP, higher levels are more significant than lower ones.

Let us make precise how minimize statements induce optimal constraint stable models. Let P be a constraint logic program associated with $(\mathcal{V}, D, C)$. For a variable assignment $\mathbf{C}$ and an integer l, define $\sum_l^{\mathbf{C}}$ as the sum of all values $a \cdot \mathbf{C}(v) + c$ for all occurrences of minimize terms $av + c@l$ in all minimize statements in P. A constraint stable model $(X, \mathbf{C})$ of P wrt. $(\mathcal{V}, D, C)$ is non-optimal if there is a constraint stable model $(X', \mathbf{C}')$ such that $\sum_l^{\mathbf{C}'} < \sum_l^{\mathbf{C}}$ and $\sum_{l'}^{\mathbf{C}'} = \sum_{l'}^{\mathbf{C}}$ for all $l' > l$, and optimal otherwise. Maximization can be achieved by multiplying each minimize term by -1.

Note that the set of constraints supported by *clingcon* is only a subset of the constraints expressible with the syntax fixed in Listing 5.1. While for example expressions with more than one constraint variable are well-formed according to the syntax, they are not supported by *clingcon*.

5.2.3 Algorithms

As already mentioned, *clingcon* 3 pursues a lazy approach to constraint solving that distinguishes two phases. During preprocessing, any part of the nogoods representing a CSP can be made explicit and thus put right away under the influence of CDCL based solving. Unlike this, the remaining constraints are at first kept implicit and their corresponding nogoods are only added via constraint propagators to CDCL solving when needed. This partitioning of constraints constitutes a trade-off. On the one

hand, constraint propagators are usually slower than unit propagation, in particular, when dealing with sets of nogoods of moderate size because of modern SAT techniques such as the two-watched-literals scheme [123]. On the other hand, translating all constraints is often impracticable, in particular, when dealing with very large domains. Hence, a good trade-off is to restrict the translation to "small constraints" in order to benefit from the high performance of CDCL solving and to unfold "larger constraints" only on demand.

In what follows, we make *clingcon*'s two-fold approach precise by presenting algorithms for translation and propagation of constraints before discussing implementation details in Section 5.3.

Partial Translation. Following Corollary 5.1.1, a subset $\mathcal{C}' \subseteq \mathcal{C}$ of the constraint atoms is used to create the set of nogoods $\Psi(\mathcal{C}')$. Therefore, Algorithm 11 creates a set of nogoods that is equivalent to $\psi(\sigma, a_1v_1 + \cdots + a_nv_n \leq b)$, as defined in (4.6) and (4.7); in turn, they are used to create $\Psi(\mathcal{C}')$ as shown in (4.5). To this end, it is initially engaged by $\textsc{Translate}(\{\sigma\}, a_1v_1 + \cdots + a_nv_n \leq b)$. We start the algorithm

Algorithm 11: TRANSLATE

Input : A set of signed literals δ and a linear constraint $a_1v_1 + \cdots + a_nv_n \leq b$.
Output : A set of nogoods.

```
1 Σ ← ∅
2 d ← next(b − Σ_{j=2}^n ub(a_j v_j), a_1 v_1)
3 while d ≤ ub(a_1 v_1)
4 |   if d + Σ_{j=2}^n lb(a_j v_j) ≤ b then
5 |   |   Σ ← Σ ∪ TRANSLATE(δ ∪ {(a_1 v_1 ≥ d)^‡}, a_2 v_2 + ··· + a_n v_n ≤ b − d)
6 |   else
7 |   |   return Σ ∪ {δ ∪ {(a_1 v_1 ≥ d)^‡}}
8 |   d ← next(d, a_1 v_1)
9 return Σ
```

by having σ in the set of literals δ, and setting d to the smallest value greater than $b - \sum_{j=2}^{n} ub(a_jv_j)$ in the image of a_1v_1. This is the smallest value needed to violate the constraint. If d and the least sum $\sum_{j=2}^{n} lb(a_jv_j)$ added by all other views is still less than b in line 4, we have to recursively translate the rest of the constraint, while subtracting d from the right-hand side in Line 5. Otherwise the constraint is already violated and we return all nogoods created so far in line 7. We iteratively increase d in line 8 and repeat this process (line 3) for all values in $img(a_1v_1)$. Note that this also involves adding all order atoms $\mathcal{O}_{\Psi(\mathcal{C}')} = \bigcup_{\delta\in\Psi(\mathcal{C}')} \delta^{\mathbf{T}} \cup \delta^{\mathbf{F}}$ included in the created nogoods $\Psi(\mathcal{C}')$ to the solver.

Which constraints to translate is subject to heuristics and command line options, as explained in Section 5.3.

Extended Conflict Driven Constraint Learning. After translating a part of the problem into a set of nogoods $\Psi(\mathcal{C}')$, using the order atoms $\mathcal{O}_{\Psi(\mathcal{C}')} \subseteq \mathcal{O}$, we explain

Algorithm 12: CDCL-CASP

Input : A constraint logic program P over $\mathcal{A}$ and externals $\mathcal{C}$ associated with $(\mathcal{V}, D, C)$, a set of constraint atoms $\mathcal{C}' \subseteq \mathcal{C}$, and a set of order atoms $\mathcal{O}_{\Psi(\mathcal{C}')} \subseteq \mathcal{O}$.

Output : A constraint stable model of P.

```
1  𝓑 ← 𝓐 ∪ 𝓒 ∪ {body(r) | r ∈ P} ∪ 𝓞_Ψ(𝓒')      // set of atoms
2  B ← ∅                                        // (Boolean) assignment
3  ∇ ← ∅                                        // set of (dynamic) nogoods
4  dl ← 0                                       // decision level
5  loop
6      (𝓑, B, ∇) ← Propagation(𝓑, B, 𝓒', 𝓒, ∇)
7      Σ ← {δ | δ ∈ Δ_P^𝓒 ∪ ∇, δ ⊆ B}
8      if Σ ≠ ∅ then
9          if dl = 0 then return unsatisfiable
10         (δ', dl) ← ConflictAnalysis_P(∇, B, δ) for some δ ∈ Σ
11         ∇ ← ∇ ∪ {δ'}
12         B ← B \ {σ | σ ∈ B, dl(σ) > dl}
13     else if B^T ∪ B^F = 𝓑 then
14         if lb_B(v) ≠ ub_B(v) for some v ∈ 𝓥 then
15             𝓑 ← 𝓑 ∪ Split_{𝓥,D}(𝓑)
16         else
17             return (B^T ∩ (𝓐 ∪ 𝓒), {v ↦ lb_B v | v ∈ 𝓥})
18     else
19         σ_d ← Select(𝓑, B)
20         dl ← dl + 1
21         B ← B ∘ σ_d
```

how to solve the remaining constraint logic program P over $\mathcal{A}$ and $\mathcal{C}$ associated with $(\mathcal{V}, D, C)$. Our algorithmic approach follows the one in [44], where a modified CDCL algorithm supporting external propagators is presented. We extend our Algorithm 1 with lazy nogood and variable generation in Algorithm 12. The algorithm relies upon a growing set of Boolean variables $\mathcal{B}$, which is initiated with all atoms (regular, constraint, and the set of the order atoms in $\mathcal{O}_{\Psi(\mathcal{C}')}$), and subsequently expanded by further order atoms. Accordingly, the Boolean assignment $\mathbf{B}$ is restricted to atoms in $\mathcal{B}$, and recorded nogoods are accumulated in ∇. Starting with an empty assignment, the Propagation method (Line 6), extends the assignment $\mathbf{B}$ with propagated literals, adds new nogoods to ∇ and extends the set of atoms $\mathcal{B}$. This method is detailed below in Algorithm 13. When encountering a conflicting assignment (Line 7), we either backjump (lines $10-12$) or, if we cannot recover from the conflict, return *unsatisfiable*. Whenever all atoms in $\mathcal{B}$ are assigned (Line 13), we check whether a complete assignment for the variables in $\mathcal{V}$ is obtained from $\mathbf{B}$ in Line 14. If this is the case, we return the constraint stable model $(\mathbf{B}^\mathbf{T} \cap (\mathcal{A} \cup \mathcal{C}), \{v \mapsto lb_\mathbf{B}(v) \mid v \in \mathcal{V}\})$.

Otherwise, $\textsc{Split}_{\mathcal{V},D}(\mathcal{B},\mathbf{B})$ creates a new order atom for the constraint variable with the currently largest domain that splits the domain in half. If we face an incomplete assignment, we extend it using the $\textsc{Select}$ function.

Algorithm 13: PROPAGATION

Global : A constraint logic program P over $\mathcal{A}$ and externals $\mathcal{C}$ associated with $(\mathcal{V}, D, C)$.
Input : A set of atoms $\mathcal{B}$, a Boolean assignment $\mathbf{B}$, two sets of constraint atoms $\mathcal{C}'$ and $\mathcal{C}$, and a set of learned nogoods ∇.
Output: A set of atoms, a Boolean assignment, and a set of learned nogoods.

```
1  loop
2      if δ ⊆ B for some δ ∈ Δ_P^C ∪ Ψ(C') ∪ ∇ then return (B, B, ∇)
3      Σ ← {δ | δ ∈ Δ_P^C ∪ ∇, δ \ B = {σ}, σ̄ ∉ B}
4      if Σ ≠ ∅ then
5          foreach δ ∈ Σ such that δ \ B = {σ} do
6              B ← B ∘ σ̄
7      else
8          Σ ← UfsPropagation_P(B)
9          if Σ ≠ ∅ then
10             ∇ ← ∇ ∪ Σ
11         else
12             Σ ← CspPropagation(B, C \ C', B)
13             for δ ∈ Σ do  B ← B ∪ δ^T ∪ δ^F
14             ∇ ← ∇ ∪ Σ
15             if Σ = ∅ then return (B, B, ∇)
```

Algorithm 13 reflects the proceeding of our propagators and is an extension of the basic propagation Algorithm 2. At first, unit propagation is run on the completion nogoods $\Delta_P^{\mathcal{C}}$, the nogoods from the partial translation $\Psi(\mathcal{C}')$, and finally the already learned nogoods ∇ on lines 3 − 6. Then, propagator $\Pi_{\Lambda_P^{\mathcal{C}}}$ is engaged via UFSPROPAGATION (line 8). If it does not add any new nogoods to ∇, CSPPROPAGATION is called (line 12). This method acts as a propagator, returning a set of nogoods Σ. Since some of these nogoods may use new order atoms not introduced so far, we dynamically extend the set of atoms $\mathcal{B}$ by the atoms in $\delta^{\mathbf{T}} \cup \delta^{\mathbf{F}}$ stemming from the added nogoods $\delta \in \Sigma$.

New nogoods produced by any propagator are added to the set ∇ of recorded nogoods and propagation resumes afterwards (lines 6 and 10). Notably, CSPPROPAGATION is not run until a fixpoint is obtained. However, its set of returned nogoods remains non-empty until a fixpoint is reached. In this way, unit propagation interleaves with constraint propagation while delaying more complex propagation. In all, since unit propagation is much faster, it always precedes unfounded set propagation, which again precedes constraint propagation. This order reflects the complexity of the respective propagators, so that the faster the propagation, the sooner it is engaged.

Lazy Variable Generation. Realizing CspPropagation as a propagator for $\Pi_{\Psi(\mathcal{C})}$ and $\Pi_{\Phi(\mathcal{V},D)}$ allows for lazy nogood generation and for capturing inferences of the order encoding. However, to be effective, lazy variable generation requires a different set of constraints to be propagated. For illustration, suppose CspPropagation is a propagator for $\Psi(\mathcal{C}) \cup \Phi(\mathcal{V}, D)$. Considering example program P_1 along with $\mathbf{T}(x < 7) \in \mathbf{B}$ results in CspPropagation$(\emptyset, \emptyset, \{\mathbf{T}(x < 7)\}) = \{\{\mathbf{T}(x < 7), \mathbf{F}(x \leq 6)\}\}$, which is a subset of $\Psi(\mathcal{C})$ according to (4.9). This nogood comprises the order atom $(x \leq 6)$ which is added to $\mathcal{B}$ in line 13. Having this nogood, unit propagation adds in turn $\mathbf{T}(x \leq 6)$ to the assignment in lines 3−6. Then, CspPropagation$(\{(x \leq 6)\}, \emptyset, \{\mathbf{T}(x \leq 6)\})$ yields the nogoods $\{\{\mathbf{T}(x \leq 6), \mathbf{F}(x \leq 7)\}, \dots, \{\mathbf{T}(x \leq 21), \mathbf{F}(x \leq 22)\}\}$ belonging to $\Phi(\mathcal{V}, D)$ and produces the corresponding order atoms $\{(x \leq 7), \dots, (x \leq 22)\}$. We see that once a certain upper bound $\mathbf{T}(v \leq x) \in \mathbf{B}$ is found, all order atoms in $\{(v \leq x') \mid x' > x, x' \in D(v), x' < ub(v)\}$ are added to $\mathcal{B}$. Similarly, if a lower bound $\mathbf{F}(v \leq x) \in \mathbf{B}$ is fixed, all order atoms $\{(v \leq x') \mid x' \leq x, x' \in D(v)\}$ are added to $\mathcal{B}$. To avoid adding superfluous order atoms, we let CspPropagation be a propagator for $\Psi(\mathcal{C}) \cup \Phi'(\mathcal{V}, D)$ where

$$\Phi'(\mathcal{V}, D) = \{\{\mathbf{T}(v \leq d), \mathbf{F}(v \leq e)\} \mid v \in \mathcal{V}, d \in D(v), e \in D(v), d < e < ub(v)\}.$$

Although $\Phi'(\mathcal{V}, D)$ is a superset of $\Phi(\mathcal{V}, D)$, CspPropagation only adds nogoods from $\Phi'(\mathcal{V}, D)$ whose order atoms have already been introduced, that is, $\{(v \leq d), (v \leq e)\} \subseteq \mathcal{B}$. While $\Phi(\mathcal{V}, D)$ contains for each variable v a linear number of nogoods of form $\{\mathbf{T}(v \leq d), \mathbf{F}(v \leq \mathit{next}(d, v))\}$, $\Phi'(\mathcal{V}, D)$ contains a quadratic number of nogoods for each variable. The nogoods in $\Phi(\mathcal{V}, D)$ allow for propagating the truth value of one order literal to its adjacent one. Unlike this, $\Phi'(\mathcal{V}, D)$ contains redundant nogoods that allow for propagating the truth value of one order literal to all greater ones by means of nogoods of form $\{\mathbf{T}(v \leq d), \mathbf{F}(v \leq e)\}$ for all values $e \in D(v)$ such that $d < e < ub(v)$. Instead of "chaining" all values together, the latter nogoods allow us to directly infer any greater value. Since we restrict our propagator for $\Phi'(\mathcal{V}, D)$ to only return nogoods where all order atoms are included in $\mathcal{B}$, no new order atoms are created. In our example, as $ub_{\mathbf{B}}(x) = 6$, the next iteration of this optimized CspPropagation function now returns $\{\mathbf{T}(x \geq 22), \mathbf{T}(x \leq 21)\}$ which is part of $\Psi(\mathcal{C})$ according to (4.11). As this introduces the order atom $(x \leq 21)$, CspPropagation also returns $\{\mathbf{T}(x \leq 6), \mathbf{F}(x \leq 21)\}$, which is a subset of $\Phi'(\mathcal{V}, D)$. It is easy to see that all intermediate atoms $\{(x \leq 6), \dots, (x \leq 20)\}$ are not introduced and we directly "jump" to the necessary atoms. Using these two nogoods, unit propagation extends the assignment by $\mathbf{T}(x \leq 21)$ and $\mathbf{F}(x \geq 22)$ in lines 3−6 reaching a complete assignment. Also $\mathcal{B}$ has been extended to now contain the order atom $(x \leq 21)$. New nogoods produced by any propagator are added to the set ∇ of recorded nogoods and propagation resumes afterwards (line 10 and line 13). Notably, CspPropagation is not run until a fixpoint is obtained. However, its set of returned nogoods remains non empty until a fixpoint is reached. In this way, unit propagation interleaves with constraint propagation while delaying more complex propagation. In all, as unit propagation is much faster, it always precedes unfounded set propagation, which again precedes constraint propagation. This order reflects the complexity of the respective

propagators, so that the faster the propagation, the sooner it is engaged.

As we have restricted x between the values $lb_{\mathbf{B}}(x) = 0$ and $ub_{\mathbf{B}}(x) = 6$, SPLIT adds the order atom $(x \leq 3)$. The SELECT function in line 18 extends $\mathbf{B}$ by e.g. $\mathbf{F}(x \leq 3)$, and we have to add another order atom $(x \leq 5)$ in line 16. After selecting $\mathbf{F}(x \leq 5)$ we have $lb_{\mathbf{B}}(x) = ub_{\mathbf{B}}(x) = 6$ and return the constraint stable model $(\{light, night, switchOn, (x < 7)\}, \{x \mapsto 6\})$ in line 15. Note that only 4 of 24 order atoms have been added to the set of atoms. This allows us to handle variables with huge domains as potentially only a small portion of its order atoms has to be added to the system.

Constraint Propagation. CSPPROPAGATION is depicted in Algorithm 14 and consists of two parts (lines 1–10 and lines 11–21). The first part starts with selecting

---

Algorithm 14: CSPPROPAGATION

Global : A constraint logic program P over $\mathcal{A}, \mathcal{C}$ associated with $(\mathcal{V}, D, C)$.
Input : A set of atoms $\mathcal{B}$, a set of constraint atoms $\mathcal{C}'$, and a Boolean assignment $\mathbf{B}$.
Output : A set of nogoods.

```
1  Σ ← ∅                                              // an empty set of nogoods
2  for v ∈ V do
3     if T(v ≤ d) ∈ B for some d ∈ D(v) then
4        ub ← min {d | d ∈ D(v), T(v ≤ d) ∈ B}
5        Σ ← Σ ∪ {{T(v ≤ ub), F(v ≤ x)} | x > ub, (v ≤ x) ∈ B, T(v ≤ x) ∉ B}
6     if F(v ≤ d) ∈ B for some d ∈ D(v) then
7        lb ← max {d | d ∈ D(v), F(v ≤ d) ∈ B}
8        Σ ← Σ ∪ {{T(v ≤ x), F(v ≤ lb)} | x < lb, (v ≤ x) ∈ B, F(v ≤ x) ∉ B}
9  if Σ ≠ ∅ then return Σ
10 for c ∈ C \ C' do
11    if Tc ∈ B then
12       Σ ← Σ ∪ PROPAGATEBOUNDS(B, Tc ⇒ γ(c))
13    else if Fc ∈ B then
14       Σ ← Σ ∪ PROPAGATEBOUNDS(B, Fc ⇒ ¬γ(c))
15    else
16       Σ ← Σ ∪ PROPAGATEREIFICATION(B, Tc ⇒ γ(c))
17       Σ ← Σ ∪ PROPAGATEREIFICATION(B, Fc ⇒ ¬γ(c))
18    if Σ ≠ ∅ then return Σ
19 return ∅
```

---

the unit nogoods from $\Phi'(\mathcal{V}, D)$. For every variable $v \in \mathcal{V}$, we check if it already has an upper bound ub (lines 3–4) given by $\mathbf{T}(v \leq ub) \in \mathbf{B}$. If this is the case, we add the nogoods

$$\{\{\mathbf{T}(v \leq ub), \mathbf{F}(v \leq x)\} \mid x > ub, (v \leq x) \in \mathcal{B}, \mathbf{T}(v \leq x) \notin \mathbf{B}\}$$

to Σ to ensure consistency of all order atoms $(v \leq x) \in \mathcal{B}$ where $x > ub$ that are not already true. Lines $6-8$ do the same for the tightest lower bound of the variable. If any nogoods are found, they are immediately returned in line 10. The PROPAGATION function continues with unit propagation on the new nogoods. The second part of the constraint propagation (lines $11-21$), generating the nogoods in $\Psi(\mathcal{C} \setminus \mathcal{C}')$ lazily, is only done if all order atoms are properly propagated, i.e. no new nogoods have been generated in the first part (lines $1-10$). This is detailed in the next paragraph.

To generate the nogoods in $\Psi(\mathcal{C} \setminus \mathcal{C}')$ lazily, Algorithm 14 uses functions PROPAGATEBOUNDS and PROPAGATEREIFICATION for half-reified constraint $\mathbf{T}c \Rightarrow \gamma(c)$ and $\mathbf{F}c \Rightarrow \overline{\gamma(c)}$, respectively, for each $c \in \mathcal{C} \setminus \mathcal{C}'$. In the respective algorithms 15 and 16, we consider four different strengths of propagation, denoted by ps. A strength of 1 means that our propagator only produces conflicting nogoods. A strength of 2 means that it additionally checks if yet undecided constraints became true. Strength 3 furthermore adds unit nogoods that also propagate the bounds of the variables in a constraint if it is already decided to be true, whereas strength 4 also computes optimized nogoods for yet undecided constraints. The propagators are conflict optimal and for strength 4 even inference optimal. We divided our propagator into two algorithms, handling reified constraints of form $\sigma \Rightarrow a_1v_1 + \cdots + a_nv_n \leq b$. Algorithm 15 is only called if $\sigma \in \mathbf{B}$. Whenever σ is true, we check whether the constraint $a_1v_1 + \cdots + a_nv_n \leq b$

Algorithm 15: PROPAGATEBOUNDS

Global : An integer ps.
Input : A Boolean assignment $\mathbf{B}$ and a half-reified constraint $\sigma \Rightarrow a_1v_1 + \cdots + a_nv_n \leq b$.
Output : A set of nogoods.

```
1  Σ ← ∅                                                  // An empty set of nogoods
2  if Σ_{j=1}^n ub_B(a_j v_j) ≤ b then return ∅
3  if ps ≤ 2 then
4  |   if Σ_{j=1}^n lb_B(a_j v_j) > b then
5  |   |   Σ ← {{σ} ∪ {(a_j v_j ≥ lb_B(a_j v_j))‡ | 1 ≤ j ≤ n}}
6  |   return Σ
7  for i = 1..n do
8  |   cur ← b − Σ_{j=1,j≠i}^n lb_B(a_j v_j)
9  |   if cur < ub_B(a_i v_i) then
10 |   |   Σ ← Σ ∪ {{σ, (a_i v_i > cur)‡} ∪ {(a_j v_j ≥ lb_B(a_j v_j))‡ | 1 ≤ j ≤ n, j ≠ i}}
11 |   |   if cur < lb_B(a_i v_i) then return Σ
12 return Σ
```

can be falsified. If it can never be falsified, e.g., the sum of the current upper bounds already satisfies the constraint in line 1, we are done. If we only have propagation strength 1 or 2, we check in line 3 whether the sum of the current lower bounds is already above the bound b. In this case, we simply return the current lower bounds of the views as a nogood, since the constraint is already violated. For example, take the

constraint $\sigma \Rightarrow x + y \leq 9$ with $D(x) = D(y) = \{1, \ldots, 15\}$ and the current lower and upper bounds $lb_{\mathbf{B}}(x) = 7$, $ub_{\mathbf{B}}(x) = 10$, $lb_{\mathbf{B}}(y) = 5$, and $ub_{\mathbf{B}}(y) = 12$. The sum of the lower bounds $7 + 5$ is greater than 9, and so the constraint is violated. Therefore, we add the nogood $\{\sigma, (x \geq 7)^{\ddagger}, (y \geq 5)^{\ddagger}\}$. If the propagation strength is greater than 2 (lines $7-11$), we try to find new upper bounds for the views of the constraint. For this purpose, *cur* represents the maximal value that $a_i v_i$ can take without violating the constraint. All other views $a_j v_j$ ($j \neq i$) contribute at least their current lower bound to the sum. In our example, this means that $cur = 9 - 5 = 4$. If this value is less than the current upper bound of $a_i v_i$ (line 8), we create a nogood that allows us to propagate the new upper bound. In the example, this is $\{\sigma, (x > 4)^{\ddagger}, (y \geq 5)^{\ddagger}\}$. Compared to the nogood that was created in line 4, this nogood is stronger as the required minimum for x is lower. If *cur* is even below the current lower bound of $a_i v_i$, we have a conflict and stop eagerly (line 11). Since $cur = 4$ and $lb_{\mathbf{B}}(x) = 7$, this is the case in our example. This algorithm has linear complexity $O(n)$, but since we consider domains/images with holes, finding the literal $(a_i v_i > cur)^{\ddagger}$ is actually $O(log(|D(v_i)|))$ which raises the overall complexity for propagation strength greater than 2.

Algorithm 16 is only called if neither $\sigma \in \mathbf{B}$ nor $\overline{\sigma} \in \mathbf{B}$, e.g. whenever σ is unknown, and propagation strength is at least 2 (line 1). If the sum of all current

Algorithm 16: PropagateReification

Global : An integer ps.
Input : A Boolean assignment $\mathbf{B}$ and a half-reified constraint $\sigma \Rightarrow a_1 v_1 + \cdots + a_n v_n \leq b$.
Output : A set of nogoods.

```
1  if ps = 1 then return ∅
2  low ← Σ_{j=1}^{n} lb_B(a_j v_j)
3  if low > b then
4      δ ← {σ}
5      if ps < 4 then
6          δ ← δ ∪ {(a_j v_j ≥ lb_B(a_j v_j))‡ | 1 ≤ j ≤ n}
7      else
8          for j ∈ 1..n do
9              low′ ← low − lb_B(a_j v_j)
10             cur ← next(b − low′, a_j v_j)
11             δ ← δ ∪ {(a_j v_j ≥ cur)‡}
12             low ← low′ + cur
13     return {δ}
```

lower bounds on the left hand side is greater than b (lines $2-3$), the constraint can never become satisfied. Given a propagation strength below 4, we simply create a nogood based on the current lower bounds. In our example, this is the same nogood $\{\sigma, (x \geq 7)^{\ddagger}, (y \geq 5)^{\ddagger}\}$ generated in Algorithm 15. If the propagation strength is 4 (lines $8-13$), we try to find a sum of the views that is minimally greater than b. In

our example, we start with a lower bound $low = 12$. By subtracting $lb_{\mathbf{B}}(x)$, we get $low' = 5$. This leaves us with $cur = next(9-5, x) = 5$ adding $(x \geq 5)^{\ddagger}$ to the nogood δ. In the second iteration, we now have to find a sufficient lower bound for y that violates the constraint. We see that this value is 5, adding $(y \geq 5)^{\ddagger}$ to δ in line 11 resulting in the nogood $\{\sigma, (x \geq 5)^{\ddagger}, (y \geq 5)^{\ddagger}\}$. Again, the complexity of the refined search is higher but also the produced nogoods are stronger. Note that as an optimization, PropagateBound and PropagateReification are only called if the bounds of the variables of the constraints have changed. The propagation strength is set using the option `--prop-strength`.

5.3 Distinguished Features

We now present different features of our new solver *clingcon* 3. We start with the preprocessing features that aim at reducing the size of domains, number of variables and prepare the program for solving. Then, we elaborate on the constraint solving techniques used by *clingcon*. After presenting the algorithmic framework of *clingcon* 3, we now describe some of its specific features. Many of them aim at reducing the sizes of domains and the number of variables, while others address special functionalities, like global constraints or multi-objective optimization over integer variables, respectively. When we refer in the following to the truth values of atoms, we consider a partial assignment obtained by propagation and/or preprocessing.

Views. A view $av + b$ can be represented with the same set of order atoms as its variable v [118]. Consider the view $-5v + 7$ together with the domain $D(v) = \{1, 2, 3, 4, 5\}$. We show how the order atoms of v are used to encode constraints over the view in *clingcon*. The view $-5v + 7$ has the following values in its image: $img(-5v + 7) = \{-18, -13, -8, -3, 2\}$. The order literals for $\{(v \leq x)^{\ddagger} \mid x \in D(v)\}$ and $\{(-5v+7 \leq x)^{\ddagger} \mid x \in img(-5v+7)\}$ are given in Table 5.1. We see that the set of

Expression	Image	Order Literals				
v	$\{1, 2, 3, 4, 5\}$	$\mathbf{T}(v \leq 1)$	$\mathbf{T}(v \leq 2)$	$\mathbf{T}(v \leq 3)$	$\mathbf{T}(v \leq 4)$	$\mathbf{T}\emptyset$
$-5v + 7$	$\{-18, -13, -8, -3, 2\}$	$\mathbf{F}(v \leq 4)$	$\mathbf{F}(v \leq 3)$	$\mathbf{F}(v \leq 2)$	$\mathbf{F}(v \leq 1)$	$\mathbf{T}\emptyset$

Table 5.1: Order literals of different views of one variable.

order atoms used for these literals is the same. By allowing views instead of variables, we avoid introducing new variables (for views). In fact, neither the XCSP [107] nor the *flatzinc*[3] format allow for using views in global constraints. For instance, a distinct constraint over the set of views $\{1000v_1, 1000v_2, 1000v_3, 1000v_4, 1000v_5\}$ translates into the same nogoods as a distinct constraint over $\{v_1, v_2, v_3, v_4, v_5\}$. Due to the restriction to use variables, according solvers like *sugar* [116] introduce auxiliary

[3] http://www.minizinc.org/downloads/doc-1.3/flatzinc-spec.pdf

variables $v'_i = 1000v_i$ for $1 \leq i \leq 5$. If $D(v_i) = \{1, \dots, 10\}$, bound propagation yields the domains $D(v'_i) = \{1000 \cdot 1, \dots, 1000 \cdot 10\} = \{1000, \dots, 10000\}$.[4] Furthermore, around 220000 nogoods for the equality constraints are created. By handling views directly, we avoid introducing these auxiliary variables and constraints in *clingcon* 3.

The same holds for minimization statements. Views on variables such as $3 * v_2$ or $-v_3$ allow for weighting variables during minimization as well as maximization, without the need of introducing auxiliary variables and additional constraints.

Non-Contiguous Integer Domains. We represent domains of variables (and images of views) as sorted lists of ranges like $[1..3, 7..12, 39..42]$ instead of single ranges like $[1..42]$. This has the advantage that we can represent domains with holes directly, without any additional constraints. Introducing order atoms for such a non-contiguous domain produces fewer atoms ($3 + 6 + 4 - 1 = 12$ in this example[5]) than for a domain only represented with two bounds (41). A drawback of this representation is that the lookup for a certain value d in the domain becomes logarithmic, as we rely upon binary search in the list of ranges. This is frequently done in Algorithms 11, 15, and 16 whenever a calculated value d leads to searching for a literal $(v \leq d)^{\ddagger}$.

Equality Processing. To minimize the numbers of atoms and nogoods that have to be created during a translation or solving process, we need to reduce the numbers of integer variables. To accomplish this, we consider the equalities in a CSP that include only two integer variables, and replace all occurrences of the first variable with a view on the second variable in all other constraints. Consider a constraint logic program P over $\mathcal{A}$ and $\mathcal{C}$ associated with $(\mathcal{V}, D, C)$. For each element $\gamma(\sigma) \in C$ of the form $ax + c_1 = by + c_2$ (or $ax + c_1 \neq by + c_2$) where σ is true (false), a, b, c_1, c_2 are integers, and $x, y \in \mathcal{V}$, we successively replace constraints in C. For this, we normalize the constraint $\gamma(\sigma)$ to $ax = by + c$ where x is lexicographically smaller than y and multiply all constraints in C containing variable y with b and replace $by + c$ by ax in them. The domain of x is made domain consistent such that $ad \in img(by + c)$ holds for all $d \in D(x)$. Afterwards, we remove $\gamma(\sigma)$ from C and y from $\mathcal{V}$. Note that by replacing variables, new equalities may arise, which we process until a fixpoint is reached.

For illustration, consider the following set C of constraints.

$$a = 2b \tag{5.1}$$
$$b = 2c \tag{5.2}$$
$$c = 2d \tag{5.3}$$
$$d = 2e \tag{5.4}$$
$$e = 2f \tag{5.5}$$
$$a + 14d - 3f + b \leq -g \tag{5.6}$$

[4] As done in the *sugar* system.

[5] The -1 is due to the fact that we do not need an atom to represent ≤ 42, as it is trivially true.

And assume that the constraint literals associated with the first 5 constraints are true. Furthermore, let $D(x) = \{-2^{12}, \ldots, 2^{12}\}$ where $x \in \{a, b, c, d, e, f, g\}\}$. Without any simplification, we have 7 variables, all with a domain size of 8193. By simply translating these constraints, we would create $2 * 7 * 8193 = 114688$ order and direct atoms and around 118 million nogoods. Let us show how equality processing allows us to significantly reduce these numbers in our example. To begin with, we multiply the constraint in (5.6), viz. $a + 14d - 3f + b \leq -g$, with 2 and replace $-6f$ with $-3e$ using the constraint in (5.5). This yields $2a + 28d - 3e + 2b \leq -2g$. Also, (5.5) allows us to restrict the domain of e to $D(e) = \{-2^{11}, \ldots, 2^{11}\}$. We then remove $e = 2f$ from the set of constraints and f from the set of variables. We repeat this procedure for all other equalities. To replace e, we again multiply the obtained constraint by 2, yielding $4a + 56d - 6e + 4b \leq -4g$, and replace $6e$ with $3d$ using (5.4). This results in $4a + 53d + 4b \leq -4g$. Again, we remove $d = 2e$ and variable e, and obtain $D(d) = \{-2^{10}, \ldots, 2^{10}\}$. Using (5.3), we multiply by 2 and replace $106d$ with $53c$ which leads to the constraint $8a + 53c + 8b \leq -8g$. To remove c, the constraint in (5.2) is used to replace $106c$ with $53b$ resulting in $16a + 69b \leq -16g$. In the last step, we apply (5.1) to get $32a + 69a \leq -32g$ which simplifies to $101a \leq -32g$. As a result, the overall set of constraints is thus reduced to a single constraint $101a \leq -32g$. This constraint uses only two variables with domains $D(a) = \{-2^7, \ldots, 2^7\}$ and $D(g) = \{-2^{12}, \ldots, 2^{12}\}$. All other constraints and variables have been removed. To translate this constraint, we need $256 + 8192 = 8448$ order atoms and 268 nogoods.

Our approach to equivalence processing is inspired by Boolean *Equi-propagation* [93], which directly replaces the order atoms of one variable with the other. Directly using integer variables, without considering the order literal representation, allows us to use this technique also in the context of lazy variable generation. Here, it reduces the number of variables, which leads to shorter constraints, which ultimately reduces the number of nogoods in the translation process.

Equality preprocessing is done once in *clingcon* 3, before the actual solving starts and can be controlled using the command line option `--equality-processing`.

Distinct Translation. *clingcon* features two alternatives for translating global distinct constraints. Assume that constraint atom c represents a distinct constraint over a set $\{v_1, \ldots, v_n\}$. Since we represent distinct constraints in terms of rules and other linear constraints, this constraint atom becomes a regular atom and is used in the head of rules.

The first method to handle this constraint uses a quadratic number of new, regular atoms $neq(v_i, v_j)$ for all $1 \leq i < j \leq n$ together with the rules

$$neq(v_i, v_j) \leftarrow (v_i - v_j \leq 1)$$
$$neq(v_i, v_j) \leftarrow (v_j - v_i \leq 1)$$

to represent that two variables are unequal. By adding the following rule to the

program

$$
\begin{aligned}
c \leftarrow neq(v_1, v_2), neq(v_1, v_3), \ldots, neq(v_1, v_n),& \\
neq(v_2, v_3), \ldots, neq(v_2, v_n),& \\
\ddots \quad \vdots& \\
neq(v_{n-1}, v_n)&
\end{aligned}
$$

clingcon ensures that c is only true if all variables are distinct from each other.

The second alternative uses a so-called *direct encoding* [121]. For each value $d \in \bigcup_{i=1}^{n} img(v_i)$, we ensure that at most one variable from $\{v_1, \ldots, v_n\}$ takes this value. Therefore, we introduce regular atoms of form $eq(v_i, d)$ for all these variables together with the rule

$$eq(v_i, d) \leftarrow (v_i \leq d), (-v_i \leq -d) \tag{5.7}$$

representing that $v_i = d$. Furthermore, we add a cardinality constraint [109] for each value d to the effect that no two or more variables may have the same value, viz.

$$c' \leftarrow 2\ \{eq(v_1, d), \ldots, eq(v_n, d)\}$$

The new regular atom c' is true if two or more variables have the same value d. If this is not the case, the distinct constraint atom holds via the rule:

$$c \leftarrow {\sim} c'$$

We reuse the direct encoding atoms $eq(v_i, d)$ for other distinct constraints. Note that introducing all direct encoding atoms also involves the creation of corresponding order atoms before the solving process. So no variable from a distinct constraint can be created lazily. This is also the reason why this option is not enabled in *clingcon* by default and distinct constraints are translated using inequalities. The use of the direct encoding along with cardinality constraints is enabled with the option `-distinct-to-card`.

Pigeon Hole Constraints. To enhance the propagation strength when translating distinct constraints in *clingcon*, we add rules for the lower and upper bounds. Consider the constraint atom c for a distinct constraint over $\{v_1, \ldots, v_n\}$ and let $U = \bigcup_{i=0}^{n} img(v_i)$, l be the nth smallest element in U, and u be the nth greatest element in U. We add the rules:

$$
\begin{aligned}
&\leftarrow c, (v_1 > u), \ldots, (v_n > u) \\
&\leftarrow c, (v_1 < l), \ldots, (v_n < l)
\end{aligned}
$$

where as before, c is treated as regular atom.

So given a distinct constraint over $\{v_1, v_2, v_3\}$ with $D(v_i) = \{1, \ldots, 10\}$ for $1 \leq i \leq 3$

we add the rules

$$\leftarrow c, (v_1 > 8), (v_2 > 8), (v_3 > 8)$$
$$\leftarrow c, (v_1 < 3), (v_2 < 3), (v_3 < 3)$$

This forbids all variables to have a value greater than eight or to have a value less than three. This feature only causes a constant overhead in the number of rules. It can be controlled using the option `--distinct-pigeon`.

Permutation Constraints. A distinct constraint over $\{v_1, \ldots, v_n\}$ where $U = \bigcup_{i=1}^{n} img(v_i)$ and $|U| = n$ induces a permutation on the variables. Let c be the constraint atom representing this global constraint. In this special case, we can add the rules

$$\leftarrow c, \sim eq(v_1, d), \ldots, \sim eq(v_n, d) \qquad \text{for all } d \in U.$$

These rules enforce that each value is taken at least once.

For example, given a distinct constraint over $\{v_1, v_2, v_3\}$ with $D(v_i) = \{1, \ldots, 3\}$ for $1 \leq i \leq 3$ we add the rules

$$\leftarrow c, \sim eq(v_1, 1), \sim eq(v_2, 1), \sim eq(v_3, 1)$$
$$\leftarrow c, \sim eq(v_1, 2), \sim eq(v_2, 2), \sim eq(v_3, 2)$$
$$\leftarrow c, \sim eq(v_1, 3), \sim eq(v_2, 3), \sim eq(v_3, 3)$$

This feature introduces direct encoding atoms along with the respective rules and order atoms in (5.7). Since these atoms cannot be treated lazily, this feature is disabled by default but can be controlled using the option `--distinct-permutation`.

Sorting. Sorting constraints by descending coefficients is known to avoid redundant nogoods in the translation process [114]. Also, systems like *sugar* sort constraints by smallest domain first, and when tied, with largest coefficient. *clingcon* can either sort by coefficient or domain size first, in decreasing or increasing order. The option `--sort-coefficient` controls the sorting of the constraints.

Splitting Constraints. Considering that directly translating a linear constraint $a_1v_1 + \cdots + a_nv_n \leq b$ with the order encoding leads to an exponential number of nogoods, we split long constraints into shorter ones by introducing new variables. Thereby we adapt the heuristics of *sugar*. We only split a constraint if the number of variables is greater than α and if its translation produces more than β nogoods. If both conditions hold, we recursively split a constraint into α parts. The new constraints have the form $a_kv_k + \cdots + a_lv_l = v_l^k$ where $1 \leq k \leq l \leq n$. α and β are freely configurable. By default, splitting is disabled in *clingcon*, but α and β can be changed with options `--split-size` and `--max-nogoods-size`.

Symmetry Breaking. When splitting a constraint like $a_1v_1 + a_2v_2 + a_3v_3 \leq b$, we get the constraints $a_1v_1 + a_1v_2 = v_2^1$ and $v_2^1 + a_3v_3 \leq b$. Equations like $a_1v_1 + a_1v_2 = v_2^1$ are represented as conjunctions of $a_1v_1 + a_1v_2 \leq v_2^1$ and $a_1v_1 + a_1v_2 \geq v_2^1$ as shown in Section 4.1. By dropping the latter inequality, we obtain an equi-satisfiable set of constraints being smaller than before but admitting more (symmetric) solutions, as v_2^1 freely varies. Symmetry breaking should therefore be enabled if one wants to enumerate all solutions without duplicates. This form of symmetry breaking is usually skipped in SAT-based CSP solvers like *sugar*. This option is set via `--break-symmetries`.

Domain Propagation. To create the domain of variables like v_n^1 in the aforementioned constraints of form $a_1v_1 + \cdots + a_nv_n = v_n^1$, we may use bound propagation. For example, the constraint $42x + 1337z = y$ where $D(x) = D(z) = \{0, 1\}$ results in the domain $D(y) = \{42 \cdot lb(x) + 1137 \cdot lb(z), \ldots, 42 \cdot ub(x) + 1137 \cdot ub(z)\} = \{0, \ldots, 1379\}$. Using domain propagation instead leads to the much smaller domain $D(y) = \{42d_x + 1137d_z \mid d_x \in D(x), d_z \in D(z)\} = \{0, 42, 1337, 1379\}$. However, we restrict domain propagation to preprocessing by default, as it has an exponential runtime. *clingcon* allows for controlling domain propagation by setting a threshold on the domain size; this is set by option `--domain-size`.

Translate Constraints. Following a two-fold approach, *clingcon* can translate some constraints while leaving others to constraint propagators as shown in Section 5.2.3. *clingcon* provides the option `--translate-constraints=m` to decide which constraints to translate or not. The translation depends on the estimated number of nogoods $\prod_{i=1}^{n-1} |D(v_i)|$ that Algorithm 11 produces for a constraint $a_1v_1 + \cdots + a_nv_n \leq b$. If this number is below the threshold `m`, *clingcon* translates the constraint. Also all order atoms used in these nogoods are created.

Redundant Nogood Check. A nogood δ is said to be stronger than a nogood δ', iff for all literals $(v > d)^\ddagger \in \delta$, there exists a literal $(v > d')^\ddagger \in \delta'$ such that $d \leq d'$ and v is a view. Whenever a nogood is created in line 7 in Algorithm 11, we compare it to the previously created one. If one of them is stronger, we only keep the stronger one, otherwise, we keep both. This feature allows *clingcon* to remove some redundant nogoods during the translation process. It is especially useful if the constraints are not sorted by descending coefficients. The check just adds constant overhead to the translation process but avoids creating a significant amount of nogoods. For instance, translating the famous *send more money* problem results in 628 nogoods among which 327 are redundant, when using `--split-size=3`. This feature can be triggered using option `--redundant-nogood-check`.

Don't Care Propagation. Suppose we want to express that $(x > 7)$ should hold whenever a holds; otherwise we do not care whether $(x > 7)$ holds or not. A corresponding constraint logic program is given in the first row of Table 5.2 together with its constraint stable models. In the standard case for CASP, the constraint atom is reified with its constraint via $\mathbf{T}(x > 7) \Leftrightarrow x > 7$. In the case that a is true, the

logic program P	constraint stable models of P
$\{a\}$ $\leftarrow a, \sim(x > 7)$	$\{(\{a, (x > 7)\}, \quad \{x \mapsto d\}) \mid d \in \{8, \ldots, 10\}\} \cup$ $\{(\{(x > 7)\}, \quad \{x \mapsto d\}) \mid d \in \{8, \ldots, 10\}\} \cup$ $\{(\emptyset, \quad \{x \mapsto d\}) \mid d \in \{1, \ldots, 7\}\}$
$\{a\}$ $\leftarrow a, \sim(x > 7)'$ $\leftarrow \sim a, (x > 7)'$	$\{(\{a, (x > 7)'\}, \quad \{x \mapsto d\}) \mid d \in \{8, \ldots, 10\}\} \cup$ $\{(\emptyset, \quad \{x \mapsto d\}) \mid d \in \{1, \ldots, 10\}\}$

Table 5.2: Constraint logic programs using reified $\mathbf{T}(x > 7) \Leftrightarrow x > 7$ and half-reified $\mathbf{T}(x > 7)' \Rightarrow x > 7$ constraints.

constraint atom $(x > 7)$ has to be true. The reification ensures that x is greater than 7, leading to three different assignments $\{\{x \mapsto d\} \mid d \in \{8, \ldots, 10\}\}$ for variable x. In the case that a is false, the constraint atom $(x > 7)$ can either be true or false. The first case results in the same three assignments, while the latter corresponds to seven others, viz. $\{\{x \mapsto d\} \mid d \in \{1, \ldots, 7\}\}$, as the reification imposes that the constraint $x > 7$ does not hold, basically enforcing $x \leq 7$. We note that in case a is false, the constraint imposed on x is either $x > 7$ or $x \leq 7$. Hence, there is actually no restriction on the assignment of x. We exploit this observation by replacing $(x > 7)$ with a new constraint atom $(x > 7)'$ and adding the rule $\leftarrow \sim a, (x > 7)'$. The idea is that atom $(x > 7)'$ imposes $(x > 7)$ as a half-reified constraint, meaning that x is enforced to be greater than 7 only if the constraint atom $(x > 7)'$ is true, i.e. $\mathbf{T}(x > 7)' \Rightarrow x > 7$. We obtain exactly the same stable models in terms of the regular atoms and integer variable assignments, as depicted in the second row of Table 5.2. The difference between these two programs lies in the assignment of the constraint atoms. The additional rule $\leftarrow \sim a, (x > 7)'$ ensures that the constraint atom $(x > 7)'$ is false, whenever a is false. Since we connect the constraint atom with its constraint using a half-reified constraint, this constraint has no effect on the assignment of x, resulting in $\{\{x \mapsto d\} \mid d \in \{1, \ldots, 10\}\}$. Although the number of constraint stable models stays the same, the number of different Boolean assignments is reduced.

This technique is called *Don't Care Propagation* [119]. All constraint atoms that only occur in integrity constraints and only positively (negatively) in the whole program are don't care atoms. *clingcon* fixes the truth value of don't care atoms to false (true), if all integrity constraints containing the atom have at least one literal being false under the current assignment. Don't care propagation can be useful in SAT, but it has even more potential to be helpful in CASP/SMT, since we not only reduce the search space but also the theory propagator has to handle only one half-reified constraint per don't care atom. Hence, only half of the inferences have to be checked. This technique is not specifically designed for CSP but it can also be used for other theories. Don't care propagation is controlled using the option `--dont-care-propagation`.

Order Atom Generation. When translating a constraint, all order atoms for all its integer variables must be available. By not translating all constraints, we also do not need to create all order atoms. Some of them can be created on the fly during propagation. With this in mind, it might still be useful to create a certain number of order atoms per variable in a preprocessing step. *clingcon* can create n atoms evenly spread among the domain values of a variable v. So if we have a domain $D(v) = \{1, \dots, 10, 90, \dots, 100\}$ and create four order atoms we use $(v \leq 3), (v \leq 8), (v \leq 92)$, and $(v \leq 97)$. These order atoms allow the solver to split the domain during the search. Option `--min-lits-per-var=n` adds at least $min(n, |D(v)| - 1)$ order atoms for each variable v.

Explicit Binary Order Nogoods. Some order atoms are created before solving. Therefore, it can also be beneficial to create a subset of the order nogoods $\Phi'(\mathcal{V}, D)$ in advance, as shown in Corollary 5.1.1. Given that we created the set of order atoms $\{(v \leq x_1), \dots, (v \leq x_n)\}$ for a variable $v \in \mathcal{V}$ where $x_i < x_{i+1}$ for $1 \leq i \leq n$, the explicit order nogoods

$$\{\{\mathbf{T}v \leq x_1, \mathbf{F}v \leq x_2\}, \dots, \{\mathbf{T}v \leq x_{n-1}, \mathbf{F}v \leq x_n\}\}$$

can also be created. To introduce these binary order nogoods for all order atoms that have been created before the solving process, the option `--explicit-binary-order` can be used.

Objective Functions. We support multi-objective optimization on sets of views. For all views $av{+}c$ subject to minimization, we use the signed order literals $(av{+}c \geq d)^\ddagger$ with weight

$$\begin{cases} d - prev(d, av + c) & \text{if } d > lb(av + c) \\ d & \text{if } d = lb(av + c) \end{cases}$$

for all values $d \in img(av + c)$ in an ASP minimize statement. This minimizes the total sum of the set of views. By using native ASP minimize statements, *clingcon* reuses *clasp*'s branch and bound and unsatisfiable core based techniques [2]. For instance, for minimizing $3x$ where $D(x) = \{1, 3, 7\}$, we have the following weighted literals in the (internal) ASP minimize statement $(3x \geq 3)^\ddagger = 3$, $(3x \geq 9)^\ddagger = 6$, and $(3x \geq 21)^\ddagger = 12$. In terms of ASP-pseudo-code this amounts to a minimize statement of form $\#minimize\{6 : {\sim} x \leq 1; 12 : {\sim} x \leq 3\}$ although order literals are not part of the input language. $(3x \geq 3)^\ddagger$ evaluates to true, while $(3x \geq 9)^\ddagger$ and $(3x \geq 21)^\ddagger$ can be expressed via order literals as ${\sim}(x \leq 1)$ and ${\sim}(x \leq 3)$, respectively.

Flattening Objective Functions. Minimizing the value of an integer variable y that is included in a constraint $\gamma(\sigma) = (a_1v_1 + \cdots + a_nv_n = y)$ where σ is true, is equivalent to minimizing the value of $a_1v_1 + \cdots + a_nv_n$. Directly using the views a_iv_i strengthens the nogoods used to represent the minimize statement. The con-

straint $a_1v_1 + \cdots + a_nv_n = y$ can be removed if y is not used anywhere else.[6] In fact, this pattern occurs quite often in our *minizinc* benchmark set. Replacing variable y with its constituents $a_1v_1 + \cdots + a_nv_n$ can be controlled with the option `--flatten-optimization`.

Reduced Nogood Learning. Whenever CspPropagation in Algorithm 13 and 14 derives a nogood, it is possible to not add it to the store of learned nogoods ∇ but rather keep it implicit and only add it if it is really needed for conflict analysis. The internal interface of *clasp* supports such a behavior. While the learned nogoods ∇ improve the strength of unit propagation, too many nogoods decrease its performance. Therefore, lazily adding these nogoods when they are actually needed can improve unit propagation. To disable the storage of nogoods and handle them implicitly, *clingcon* provides option `--learn-nogoods`.

5.4 Evaluation

In this section, we evaluate the afore-presented features and compare *clingcon* with other systems. We performed all our benchmarks on an Intel Xeon 3.40GHz processor with Debian GNU/Linux 3.8. We used a timeout of 1800 seconds and restricted main memory to 6GB. In all tests, we count a memory out as a timeout. The experiments are split into three sections. First, we evaluate the presented features and discuss corresponding configurations of *clingcon*. Second, we compare *clingcon* with state of the art CP solvers using the benchmark classes of the *minizinc* competition 2015. And finally, we contrast *clingcon* with other CASP systems using different CASP problems.

To evaluate the presented techniques, we give a comprehensive comparison in Table 5.4. To concentrate on the CP techniques of *clingcon* 3.2.0, we use the CP benchmarks of the *minizinc* competition 2015.[7] We removed the benchmark classes *large scheduling* and *project planning* as they cannot be translated into the *flatzinc* format without the use of special global constraints. For all other classes, we used the *mzn2fzn*[8] toolchain to convert all instances to *flatzinc* while removing all non-linear and global constraints except for distinct. This functionality is provided by *mzn2fzn*, which translates non-supported constraints away. We use the standard translation provided by *mzn2fzn* to handle all benchmark classes. In this way, even problems using constraints on sets, non-linear equations, or complex global constraints can be handled by solvers restricted to basic linear constraints. For making this benchmark suite available to the CASP community, we build a converter from *flatzinc* to the *aspif* format [54] used by *clingcon*; it is called *fz2aspif*.[9] To evaluate the different features, we modified the scoring system of the *minizinc* competition, which is based on the Borda count evaluation technique. On a per instance basis, a configuration gets one point for every other configuration being worse. A configuration is considered

[6] We keep the constraint to correctly print y in a solution.
[7] `http://www.minizinc.org/challenge2015/challenge.html`
[8] `http://www.minizinc.org/software.html`
[9] `https://potassco.org/labs/2016/12/02/fz2aspif.html`

Option	Value	Explanation
--equality-processing	true	Enable equality processing
--distinct-to-card	false	Translate distinct constraints using inequalities
--distinct-pigeon	true	Use pigeon hole constraints
--distinct-permutation	false	Not using permutation constraints
--sort-coefficient	false	Sort by domain size first
--sort-descend-coefficient	true	Sort using decreasing coefficients
--sort-descend-domain	false	Sort using increasing domain sizes
--split-size	-1	Not splitting constraints
--max-nogoods-size	1024	Not splitting constraints with less than 1024 nogoods
--translate-constraints	10000	Translate constraints with less than 10000 nogoods
--break-symmetries	true	Break symmetries when splitting
--domain-size	10000	Use 10000 as a threshold for domain propagation
--redundant-nogood-check	true	Enable redundant nogood check when translating
--dont-care-propagation	true	Enable don't care propagation
--min-lits-per-var	1000	Introduce at least 1000 order atoms per variable
--flatten-optimization	true	Flatten the objective function
--prop-strength	4	Use highest propagation strength 4
--explicit-binary-order	false	Not explicitly creating nogoods from $\Phi'(\mathcal{V}, D)$
--learn-nogoods	true	Add all learned nogoods to ∇ immediately

Table 5.3: Default configuration D of *clingcon* 3.2.0.

worse, if either the found optimization value is at least 1% lower, or if it has the same optimization value but is slower. A configuration is considered slower if it is at least 5 seconds slower. Classes marked with * are decision problems (all others are optimization problems); classes containing the global distinct constraint are marked with †. We have exactly five instances per class.

The following discussion refers to the results shown in Table 5.4. The columns used for comparison are named in the paragraph heading. Column D presents the default configuration of *clingcon* given in Table 5.3. All other listed configurations differ only in one or two options from this default in order to test specific techniques. For instance, for evaluating equality processing, we compare default configuration D, using equality processing, with configuration NE, disabling equality processing. Thus, except for `--equality-processing`, all other options remain unaltered.

Equality Processing (D, NE) To evaluate the influence of equality processing, we compare default configuration D (with equality processing) with configuration NE (without equality processing). This feature improves performance on nearly all benchmark classes significantly. By simply removing constraints and variables the underlying CSP gets easier to solve.

instances	D	NE	DT	NP	PO	SC	SP	ST	NS	D_1	D_2	D_3	T_1	T_2	T_3	T_4	NR	ND	M_1	M_2	M_3	NF	P_1	P_2	P_3	EO	RL
*costas**†	38	47	15	39	41	38	27	27	24	27	27	26	12	33	0	0	16	38	36	38	40	38	12	12	38	32	**60**
cvrp†	49	38	90	61	65	49	51	44	52	51	51	51	31	67	41	0	34	55	93	95	97	49	9	9	49	**100**	66
freepizza	**121**	111	**121**	**121**	**121**	**121**	47	26	49	51	47	47	79	**121**	**121**	26	26	100	99	107	102	115	109	113	120	120	109
gfd-schedule	105	69	103	102	102	103	52	11	47	54	54	54	58	102	102	0	36	50	84	**114**	112	103	81	81	105	102	93
grid-colour	**130**	**130**	**130**	**130**	**130**	**130**	**130**	**130**	**130**	**130**	**130**	**130**	**130**	**130**	**130**	**130**	**130**	**130**	**130**	**130**	**130**	**130**	**130**	**130**	**130**	**130**	**130**
is†	118	111	116	119	111	118	45	0	45	44	45	44	96	118	118	0	0	93	**128**	53	56	118	0	0	117	90	80
mapping	106	87	106	106	106	106	33	0	18	30	33	33	104	112	110	0	0	**124**	84	109	111	106	0	0	106	97	112
knapsack	66	55	64	66	66	64	7	2	7	3	7	7	65	65	66	0	2	66	44	73	76	57	**99**	**99**	64	40	34
*nmseq**	100	100	100	100	98	99	16	27	25	25	28	28	100	67	100	0	28	100	88	79	100	96	0	0	92	39	**130**
opd	89	82	88	89	89	89	**125**	**125**	121	**125**	**125**	**125**	52	89	88	0	**125**	63	44	89	90	88	79	79	88	78	83
open-stacks†	96	73	116	107	77	95	57	54	71	65	58	59	33	96	77	21	55	95	79	95	103	92	61	61	94	**117**	54
p1f†	66	66	**100**	70	71	66	35	65	35	36	35	35	84	66	66	0	65	75	57	74	76	71	64	64	66	78	68
radiation	125	118	125	125	124	115	62	43	40	52	62	62	108	107	103	0	50	125	112	119	110	101	68	68	125	**128**	40
roster	**130**	**130**	**130**	**130**	**130**	**130**	**130**	**130**	**130**	**130**	**130**	**130**	**130**	**130**	**130**	0	**130**	**130**	**130**	**130**	**130**	**130**	**130**	**130**	**130**	**130**	**130**
spot5	86	86	86	86	86	86	105	75	105	100	105	105	86	86	86	0	78	86	68	111	68	86	102	102	86	**120**	106
tdtsp	77	69	77	77	77	77	94	0	35	94	94	94	47	76	30	0	0	3	70	59	**119**	77	44	12	84	46	75
triangular	15	15	15	15	15	15	38	11	38	38	38	15	56	15	15	0	11	15	15	15	15	8	**77**	**77**	15	33	27
zephyrus	72	47	72	72	72	72	62	0	62	62	62	62	64	72	72	0	0	72	71	**103**	0	72	0	0	46	72	7
total	1589	1434	**1654**	1615	1581	1573	1116	770	1034	1117	1131	1107	1335	1552	1455	177	786	1420	1432	1593	1535	1537	1065	1037	1555	1552	1404

Table 5.4: Comparison of different features of *clingcon* 3.2.0 on the benchmark set of the *minizinc* competition 2015. Shown are scores of how often a configuration is better than another. Bold numbers indicate the best configuration for the benchmark class.

Distinct Translation (D, DT) Translating global distinct constraints into cardinality rules prevents order atoms from being created lazily. The default configuration D translates them into a set of inequalities. The translation using cardinality constraints in column DT performs better on *cvrp*, *open-stacks*, and *p1f*, while it performs worse on the benchmark class *costas*. As long as the domain size is small, this feature can be useful for problems using distinct constraints. The configuration DT performs best of all tested configurations.

Pigeon Hole Constraints (D, NP) Since pigeon hole constraints add only constant overhead in the number of nogoods, they are enabled in configuration D. Disabling their addition, slightly increases performance on benchmark classes containing distinct constraints (marked with †), as witnessed in column NP. Although these constraints have no positive effect on the benchmarks at hand, we keep this feature enabled by default since it increases propagation strength.

Permutation Constraints (D, PO) Unlike pigeon hole constraints, permutation constraints introduce direct encoding atoms which prevents lazy variable generation for some constraint variables. This is the reason why this feature is disabled by default in configuration D. We enabled it in column PO. Again, this feature only influences benchmark classes containing distinct constraints. It improves performance for the *cvrp* class but decreases it on the other classes. The impact of this feature depends upon the respective problem.

Sorting (D, SC) As we cannot account for all combinations of sorting mechanisms, we evaluate this feature only on the cases discussed in [114]. Default configuration D implements the one in *sugar*; it sorts by smallest domain first and prefers larger coefficients. The alternative sorting recommended in [114] first sorts on larger coefficients and afterwards uses the smaller domain. This behavior is enforced by setting `--sort-coefficient=true` and reflected in column SC. We see that both sorting methods yield a similar performance when applied to our lazy nogood generating approach.

Splitting Constraints (D, SP, T_4, ST) Splitting constraints into smaller ones is mandatory for any translation-based approach using the order encoding to avoid an exponential number of nogoods. We restricted our evaluation to a splitting size of 3, as done in *sugar*. The default configuration D of *clingcon* does not split any constraints. The effect of splitting constraints into ternary ones (`--split-size=3`) is reflected by column SP; it performs poorly in our lazy nogood generating setting because it introduces many new constraints and variables. On the other hand, when translating all constraints (`--translate-constraints=-1`) as shown in column T_4, the split into constraints of up to three variables (using `--translate-constraints=-1` and `--split-size=3`) increases performance significantly, as witnessed by column ST. We conclude that splitting constraints is not necessary for lazy nogood generating solvers but essential for translational approaches that use the order encoding.

Symmetry Breaking (*SP*, *NS*) Splitting constraints introduces auxiliary variables that may lead to redundant solutions. Symmetry breaking eliminates such redundancies and has only an effect when splitting constraints. This is why it is interesting to compare column *SP* (`--split-size=3`) where symmetry breaking is enabled with column *NS* (`--split-size=3` and `--break-symmetries=false`) where it is disabled. In both cases, all constraints are split into ternary ones. The additional constraints remove symmetric solutions from the search space and therefore seem to be beneficial, especially on classes *tdtsp*, *radiation*, and *mapping*.

Domain Propagation (*D_1*, *D_2*, *SP*, *D_3*) To investigate the impact of domain propagation during preprocessing, we tested four different configurations that all split constraints into ternary ones (`--split-size=3`). They only differ in using the options `--domain-size=0` (no domain propagation) in column D_1, `--domain-size=1000` in column D_2, `--domain-size=10000` in column *SP*, and `--domain-size=-1` (unlimited domain propagation) in column D_3. We observe that unlimited domain propagation reduces performance in benchmark class *triangular* but has no significant influence otherwise. The other tested configurations have no influence on the runtime of the benchmarks. We assume that domain propagation does not prune the domain enough to make a considerable difference. For the default configuration of *clingcon*, we decided to restrict it to a reasonable number (10000) which leaves it enabled for mid-sized domains.

Translate Constraints (*T_1*, *T_2*, *D*, *T_3*, *T_4*) We have already seen that translating all constraints as shown in column T_4 is not very beneficial. Now, we evaluate whether the translation of "small" constraints improves performance through a mixture of "translating small constraints" and "handling larger ones lazily". Therefore, we compare the results obtained with option `--translate-constraints=0` (no constraints are translated) in column T_1, with T_2 where `--translate-constraints=1000` (translate constraints that produce up to 1000 nogoods) is used, with *D* using `--translate-constraints=10000` (up to 10000 nogoods), with T_3 using `--translate-constraints=50000` (up to 50000 nogoods), and T_4 using `--translate-constraints=-1` (all constraints are translated). There is a trade-off on the size of constraints to translate. While translating small constraints (constraints that produce up to 1000 nogoods) improves performance, the translation of larger constraints decreases it again. On some benchmarks, like *triangular* and *p1f*, translating no constraints is beneficial. Also, translating all constraints in T_4 performs worst of all tested configurations.

Redundant Nogood Check (*ST*, *NR*) To evaluate this feature, we decided to translate all constraints (`--translate-constraints=-1`). Since this configuration is not producing good results for a comparison (most of the time the translation is simply too large to be finished), we additionally split the constraints into ternary ones with option `--split-size=3`. With this, we compare the configuration with redundancy check in column *ST* with *NR* where

redundancy checking is disabled (`--redundant-nogood-check=false`). The redundant nogood check is fast and simply removes redundant nogoods from the order encoding. Benchmark classes like *costas* and *cvrp* perform better with the reduced set of nogoods, while redundant nogoods are beneficial for *gfd-schedule* and *radiation*.

Don't Care Propagation (*D*, *ND*) is enabled by default and removes unnecessary implications from the problem. Disabling (`--dont-care-propagation=false`) this feature in column *ND* decreases performance.

Order Atom Generation (M_1, *D*, M_2, M_3) Adding order atoms lazily is mandatory to handle large domains. We now evaluate the effect of adding a small amount of order atoms eagerly for every constraint variable, evenly spread among its domain values. We compare column M_1 using `--min-lits-per-var=0` (adding no atoms), with D using `--min-lits-per-var=1000` (adding 1000 order atoms per variable), with M_2 using `--min-lits-per-var=10000` (adding 10000), and M_3 using `--min-lits-per-var=-1` (adding all order atoms). Adding no order atoms in advance drastically reduces performance of the system while adding 1000 to 10000 order atoms achieves best performance. When adding too many or even all order atoms before solving, performance is again decreased, especially on classes with large domains like *zephyrus*. Also, note that the tested benchmark classes are very sensitive to this option as adding atoms beforehand may influence the heuristic of the search.

Flattening Objective Functions (*D*, *NF*) is a feature well received by this benchmark set. All *flatzinc* encodings contain only one variable subject to minimization. On most benchmark classes this variable simply represents the sum of a set of variables. Adding this set directly to the objective function avoids adding an unnecessary and probably large constraint and also improves propagation strength of the learned nogoods. Unlike D, configuration *NF* disables this feature via `--flatten-optimization=false`. We observe that flattening the optimization statement increases the performance on many benchmark classes.

Lazy Nogood Generation (P_1, P_2, P_3, *D*) We now evaluate the four propagation strengths described before where `--prop-strength=1` is reflected by the results in column P_1, `--prop-strength=2` by the ones in column P_2, `--prop-strength=3` in column P_3, and `--prop-strength=4` in the default configuration D. We see that a high propagation strength is important. Especially propagating changed bounds with `--prop-strength=3` is necessary for many benchmark classes. Interestingly, less propagation performs best for the classes *knapsack* and *triangular* where constraint propagation is not dominating the search but still takes time. On these classes, configurations with propagation strength 1 or 2 spend less time on CspPropagation and more on pure CDCL search, as attested by a much higher number of choices.

Explicit Binary Order Nogoods (*D*, *EO*) Default configuration D does not introduce explicit binary order nogoods $\Phi(\mathcal{V}, D)$ but uses a propagator for captur-

ing the corresponding inferences lazily. The `--explicit-binary-order=true` option reflected in column *EO* creates these nogoods explicitly for all order atoms created during preprocessing, leaving the others subject to lazy nogood propagation. Although, overall performance of the implicit binary order nogoods is better, for some benchmark classes like *cvrp* and *spot5* using binary order nogoods explicitly is the best choice. This is one of the options for which it is hard to find a clear cut default setting and that needs consideration for each benchmark class.

Reduced Nogood Learning (*D*, *RL*) *clingcon*'s default configuration D adds all nogoods returned by CspPropagation to the set of learned nogoods (viz. ∇ in Algorithm 13). Lazily adding these nogoods when they are actually needed for conflict analysis is achieved with `--learn-nogoods`; the results are shown in column *RL*. The average performance of adding nogoods lazily is inferior to the one obtained by learning all nogoods. Nevertheless, the latter setting performs best on *costas* and *nmseq*, the two decision problems in our benchmark set. Future work has to investigate which of the nogoods have to be learned and which of them can be added lazily.

Configuration DT is the configuration with the highest overall score. Nevertheless, *clingcon*'s default configuration is more conservative since it allows for using lazy variable generation in all cases. For instance, with configuration DT it is impossible to run the multi-shot n-queens example presented in the next Chapter 6, because 2^{30} order atoms would have to be created per queen in order to use cardinality constraints for the distinct constraint.

Next, we compare *clingcon* to state of the art CP solvers on the same set of benchmarks with the same scoring system. The second column of Table 5.5 shows configuration DT of *clingcon* 3. This is the best configuration of the internal comparison in Table 5.4, which is obtained using the command line option `--distinct-to-cardinality=true`. We compare it to *g12fd* (Mercury FD Solver), which is the G12 FlatZinc interpreter's default solver, taken from the *minizinc* 2.0.11 package.[10] Furthermore, we have taken *gecode* 4.4.0,[11] a well-known classical CP solver. Also, the lazy clause generating solvers *minisatid* 3.11.0 [35][12] as well as *chuffed*,[13] the best solver of the *minizinc* competition 2015.[14] Finally, we compare to *picatsat* 2.0,[15] a CP solver that won the second place at the *minizinc* competition 2016 by translating constraints into SAT using a logarithmic encoding. We ran *g12fd* and *gecode* with `--ignore-user-search` to disable any special heuristic given in the problem encodings for all solvers. In the competition, this is called "free search". To measure the core performance of the systems, it is most instructive to consider *chuffed′* and *picatsat′*

[10] `http://www.minizinc.org/software.html`
[11] `http://www.gecode.org`
[12] With some bugfixes. Special thanks to Bart Bogaerts for his great support on this work.
[13] `https://github.com/geoffchu/chuffed` — SHA 5b379ed9942ee59e8684149eae3fec1af426f6ee
[14] It did not participate in the ranking as it is was entered by the organizers. It ran outside of competition and was faster than the winning system.
[15] `http://picat-lang.org`

instances	*clingcon*	*g12fd*	*gecode*	*minisatid*	*chuffed*	*chuffed′*	*picatsat*	*picatsat′*
costas*†	6	0	**19**	9	8	8	11	11
cvrp†	24	3	6	5	**30**	22	4	6
freepizza	**35**	15	0	31	26	26	3	3
gfd-schedule	10	9	12	27	**28**	**28**	20	14
grid-colour	**35**	1	8	34	23	23	31	31
is†	17	15	2	17	**35**	32	19	11
mapping	17	14	0	17	**29**	**29**	23	16
knapsack	19	11	14	8	5	5	**26**	**26**
nmseq*	24	15	**25**	3	21	21	5	5
opd	25	6	4	15	21	17	31	**32**
open-stacks†	20	0	9	8	**35**	**35**	21	16
p1f†	21	0	22	3	**34**	21	5	8
radiation	28	7	7	12	**33**	**33**	18	14
roster	**35**	**35**	0	**35**	**35**	**35**	29	28
spot5	20	10	0	19	16	16	**35**	31
tdtsp	8	**35**	10	1	31	24	10	9
triangular	15	**29**	5	26	23	24	12	12
zephyrus	3	20	7	10	26	26	**30**	**30**
total	362	225	150	280	**459**	425	333	303

Table 5.5: Comparing *clingcon* 3.2.0 *DT* with different state of the art CP solvers on the *minizinc* competition 2015 benchmark set.

which use the two solvers on exactly the same set of constraints as *clingcon*. Hence, all non-linear and global constraints (except distinct) are translated using *mzn2fzn* in the same way for all systems.[16]

The results in Table 5.5 show that *clingcon*[17] outperforms established systems such as *g12fd*, *gecode*, *minisatid*, and even *picatsat*. There are also different benchmark classes where solvers dominate each other and vice versa. We point out that *gecode* has special propagators for many non-linear and global constraints that have been used in the benchmarks. Also *chuffed*, as a lazy clause generating solver, has propagators

[16] Unfortunately, we were unable to compare to the lazy clause generating system *g12lazy*, as it produced wrong results on some of the benchmarks and is no longer maintained. We cannot convert the competition benchmarks to a format readable by *sugar*, as existing converters are outdated and not compatible anymore.

[17] Note that the Borda Count scores are relative to the compared systems, and therefore are different for the same configuration of *clingcon* in Table 5.4 and 5.5.

for many other constraints and can therefore handle some of the benchmark classes much better. As we are building a CASP system, we refrain from supporting a broad variety of global constraints, as some of them can be modeled in ASP. So for a better comparison on the features of *clingcon*, we translated all non-linear and global constraints except for distinct in the columns *chuffed′* and *picatsat′* into linear ones. Here, we see that these systems profit from the dedicated treatment of global constraints but that the base performance of *clingcon* is comparable. In general, *clingcon* does not match the performance of the best solver of the *minizinc* competition 2015 but on benchmark classes like *freepizza*, *grid-colour*, *opd*, *knapsack*, and *spot5*, it even outperformed *chuffed*. We conclude that *clingcon*, despite being a CASP system, is at eye level with state of the art CP solvers but cannot top the best lazy clause generating systems.

Finally, we compare *clingcon* against six other CASP systems.

- *inca* [43] with the option `--linear-bc`,[18] a lazy nogood generating system not supporting lazy variable generation.
- *clingcon* 2 [102], using *gecode* 3.7.3 as a black-box CP solver.
- *ezcsp* 1.6.24 [9], also pursuing a black-box approach but using CP solver B-Prolog 7.4 with ASP solver *clasp*.
- *aspartame* [11], a system using an eager translation of the constraint part by means of an ASP encoding.
- *ezsmt* 1.0.0 [82], translating CASP programs to SMT, solved by SMT solver *z3* 4.2.2.
- *clingo* 5.1.0, a pure ASP solver to measure the influence of the CP part on solving.

The first benchmark class is the two dimensional strip packing problem [110]; its encoding is shown in Listing 5.2. In Table 5.6, column *clingo* 5 reflects the results obtained with a highly optimized ASP encoding, using a handcrafted order encoding. Time is given in seconds, letting - denote a timeout of 1800 seconds. The best objective value computed so far is given in the columns headed with opt. For *aspartame*, we have taken an encoding provided in [11]. For the other systems such as *clingcon* 2, *clingcon* 3, and *inca*, we adjusted the syntax for the linear constraints. We refrained from comparing with *ezcsp* or *ezsmt* as both systems are not supporting optimization of integer variables. The bottom row counts the number of times a system performed best. We clearly see that *clingcon* 2 is outperformed even by the manual ASP encoding. The new *clingcon* 3 system performs best. The translational approach of *aspartame* is close to the *inca* system, and both perform better than the manual ASP approach. According to [110], these results compete with dedicated, state of the art systems.

[18] This option was recommended by the authors of the system for these kind of benchmarks.

instances	*clingo* 5 time	*clingo* 5 opt	*aspartame* time	*aspartame* opt	*clingcon* 2 time	*clingcon* 2 opt	*inca* time	*inca* opt	*clingcon* 3 time	*clingcon* 3 opt
BENG01	9	30	20	30	-		916	30	**2**	**30**
BENG02	-	58	**1336**	**57**	-		-	58	-	58
BENG03	-	87	-	85	-		-	85	**775**	**84**
BENG04	-	111	-	**108**	-		-	**108**	-	**108**
BENG05	-	141	-	**136**	-		-	**136**	-	**136**
BENG06	1226	36	32	36	-		**5**	**36**	23	36
BENG07	-	69	-	**68**	-		-	69	-	**68**
BENG08	-		-		-		-	104	-	**103**
BENG09	-		-		-		-		-	**128**
BENG10	-		-		-		-		-	**158**
CGCUT01	**1**	**23**	**1**	**23**	-	26	-	25	**0**	**23**
CGCUT02	-	66	-	**65**	-		-	67	-	**65**
CGCUT03	-		-		-		-		-	
GCUT01	-	1016	5	1016	**0**	**1016**	**0**	**1016**	**0**	**1016**
GCUT02	-	1242	-	1195	-		-	**1190**	-	**1190**
GCUT03	-		134	1803	-		**1**	**1803**	12	1803
GCUT04	-		-		-		-		-	
HT01	**1**	**20**	**1**	**20**	-	22	346	20	**0**	**20**
HT02	8	20	5	20	-	25	77	20	**1**	**20**
HT03	**1**	**20**	**1**	**20**	-		10	20	**0**	**20**
HT04	840	15	33	15	-		-	16	**8**	**15**
HT05	12	15	**9**	**15**	-		**8**	**15**	13	15
HT06	14	15	8	15	-		359	15	**1**	**15**
HT07	-	31	**175**	**30**	-		-	31	-	31
HT08	**1284**	**30**	-	31	-		-	31	-	31
HT09	-	31	272	37	-		-	31	**41**	**30**
NGCUT01	**0**	**23**	**0**	**23**	**1**	**23**	**0**	**23**	**0**	**23**
NGCUT02	**2**	**30**	**1**	**30**	-	33	80	30	**0**	**30**
NGCUT03	**2**	**28**	**2**	**28**	-		**1**	**28**	**0**	**28**
NGCUT04	**0**	**20**	**0**	**20**	**0**	**20**	**0**	**20**	**0**	**20**
NGCUT05	**0**	**36**	**0**	**36**	-		**0**	**36**	**0**	**36**
NGCUT06	8	31	**1**	**31**	-		**0**	**31**	**0**	**31**
NGCUT07	**0**	**20**	**0**	**20**	**0**	**20**	**0**	**20**	**0**	**20**
NGCUT08	**1**	**33**	**1**	**33**	-	36	38	**33**	**0**	**33**
NGCUT09	**87**	**50**	-	50	-	57	-	50	1549	50
NGCUT10	6	80	**1**	**80**	-	81	**0**	**80**	**0**	**80**
NGCUT11	**4**	52	**1**	**52**	-	55	**0**	**52**	**0**	**52**
NGCUT12	-	87	**3**	87	-		**0**	**87**	**0**	**87**
#single best		13		21		4		16		**28**

Table 5.6: Comparison of different CASP systems on the two dimensional strip packing problem.

The next benchmark classes are incremental scheduling, weighted sequence, and reverse folding, all stemming from the ASP competition.[19] Encodings for *clingo*, *ezcsp*,[20] *ezsmt*, and *clingcon* 2 have been taken from [82] in combination with instances from the ASP competition.[21] We changed the pure ASP encoding for *clingo* slightly for a better grounding performance. For these classes, we could not provide an encoding for *aspartame*, as its prototypical CASP support does not allow for modeling parametrized n-ary constraints.

For incremental scheduling, *inca* produces wrong results due to its usage of an intermediate version of *gringo*, viz. 3.0.92. The runtime in seconds for incremental scheduling is shown in Table 5.7. We see that *clingcon* 2 improves on the dedicated ASP encoding. In fact, incremental scheduling is a true CASP problem where the pure ASP encoding can be improved by using CP. While the black-box approach of *ezcsp* performs worst, *ezsmt* and *clingcon* 3 clearly dominate this comparison.[22] The enhanced preprocessing techniques and the lazy variable generation of *clingcon* even outperforms the industrial SMT solver *z3* (as used in *ezsmt*).

For the weighted sequence problem, we see in Table 5.8 that *inca*, *clingo*, *ezsmt*, and *clingcon* 3 perform well on this benchmark set, while *clingcon* 2 could not compete with the timings of the other systems and *ezcsp* does not solve any of them. Again, time is shown in seconds and - denotes a timeout of 1800 seconds. We also see that the performance of the pure ASP encoding is in the same range as that of the winning CASP systems. Hence, the ASP solving part clearly dominates the CSP part. This also explains the slightly worse performance of *clingcon* 3 due to its heavy preprocessing of the CSP part.

For the reverse folding problem, we compare the same systems as before. Table 5.9 gives the running time in seconds. While all CASP systems improve upon the pure ASP encoding, *clingcon* 2 and *clingcon* 3 perform best on this benchmark class. The preprocessing overhead of *clingcon* 3 does not pay off in terms of runtime on this benchmark class, making it perform slightly worse than *clingcon* 2. Of the two lazy nogood generating solvers *inca* and *clingcon* 3, the latter performs better due to lazy variable generation, as not all order atoms have to be generated before solving. While the black-box approach of *ezcsp* can solve the problem, the translation to SMT by *ezsmt* performs even better. We conclude that this is also due to the fact that no auxiliary atoms for an encoding of the constraints are used in *ezsmt*. A closer inspection revealed that the number of choices for *inca* and *clingcon* 2 is below 100 on average. For this problem, the ASP part is dominated by the CSP part. This is also the reason why the pure ASP encoding produces a memory out on all instances (it was not able to ground all constraints).

We conclude that *clingcon* 3 improves significantly upon its predecessor *clingcon* 2, is comparable to state of the art CP systems, and the currently fastest CASP system

[19] `http://aspcomp2015.dibris.unige.it/LPNMR-comp-report.pdf`

[20] To be comparable, we used the encoding without *cumulative* constraint.

[21] We refrained from using the other three benchmark classes from this source as the available instances were too easy to solve to produce informative results.

[22] The time to run the completion and translation processes for *ezcsp* and *ezsmt* is not included in the tables.

available. All benchmarks, encodings, instances and results are available online.[23] Finally, we present an approach to multi-shot CASP solving that makes use of the presented features such as lazy variable generation.

[23] `https://potassco.org/clingcon`

instances	*clingo* 5	*clingcon* 2	*clingcon* 3	*ezcsp*	*ezsmt*
020-inc	302	**1**	**0**	**0**	**0**
028-inc	-	16	**4**	-	**5**
044-inc	-	518	149	-	**116**
063-inc	335	**0**	**1**	-	**0**
083-inc	268	-	**1**	-	**0**
096-inc	719	298	**1**	-	**0**
106-inc	470	-	**2**	-	**1**
158-inc	355	**4**	**1**	**0**	**1**
175-inc	-	83	**6**	-	**4**
181-inc	-	**5**	**1**	-	**2**
184-inc	425	-	**1**	-	**1**
211-inc	-	-	**194**	-	799
214-inc	-	-	**7**	-	76
230-inc	-	77	**9**	-	24
256-inc	-	-	-	-	-
257-inc	-	-	-	-	-
266-inc	-	-	**767**	-	-
334-inc	-	-	-	-	-
338-inc	-	-	-	-	-
362-inc	-	-	-	-	-
# single best	0	4	**14**	2	11

Table 5.7: Comparison of different CASP systems on the incremental scheduling problem.

instances	*clingo* 5	*inca*	*clingcon* 2	*clingcon* 3	*ezcsp*	*ezsmt*
01-tree	**6**	**5**	**6**	**1**	-	**1**
02-tree	**8**	**1**	**8**	**3**	-	**5**
05-tree	**5**	**1**	**4**	**1**	-	**3**
06-tree	**3**	**2**	19	**1**	-	**3**
07-tree	**3**	**4**	**1**	**1**	-	**4**
11-tree	**2**	**0**	6	**1**	-	**0**
15-tree	**2**	**3**	**1**	**1**	-	**1**
16-tree	**2**	**4**	15	**5**	-	**1**
22-tree	4	**0**	**4**	**1**	-	**0**
23-tree	**3**	**0**	**4**	**0**	-	**3**
26-tree	**2**	**3**	51	**4**	-	**1**
29-tree	7	**1**	26	**2**	-	**5**
33-tree	**9**	**7**	95	15	-	**9**
35-tree	24	**10**	231	30	-	26
38-tree	**7**	17	30	**10**	-	**11**
39-tree	**6**	**7**	330	46	-	23
40-tree	**3**	9	398	47	-	14
41-tree	36	**14**	**18**	**13**	-	34
49-tree	**7**	**3**	220	22	-	14
53-tree	**5**	8	297	33	-	**2**
#single best	**17**	**17**	8	14	0	15

Table 5.8: Comparison of different CASP systems on the weighted sequence problem.

instances	*clingo* 5	*inca*	*clingcon* 2	*clingcon* 3	*ezcsp*	*ezsmt*
07-reverse	-	**1**	**0**	**1**	11	**1**
11-reverse	-	**1**	**0**	**1**	9	**1**
15-reverse	-	**1**	**1**	**1**	**6**	**1**
18-reverse	-	**2**	**1**	**2**	27	**1**
20-reverse	-	**6**	**3**	4	60	10
24-reverse	-	12	**5**	**8**	272	49
28-reverse	-	11	**5**	**7**	107	**8**
31-reverse	-	20	**8**	36	128	73
34-reverse	-	25	**11**	20	625	112
35-reverse	-	35	**15**	23	353	96
39-reverse	-	40	**18**	**23**	682	212
44-reverse	-	-	**33**	**38**	-	339
47-reverse	-	**6**	**4**	**4**	86	**4**
49-reverse	-	**7**	**4**	**4**	67	**4**
50-reverse	-	**2**	**1**	**2**	12	**4**
#single best	0	8	**15**	12	1	8

Table 5.9: Comparison of different CASP systems on the reverse folding problem.

Chapter 6

Multi-Shot Constraint Answer Set Programming

This chapter shows how CASP can be extended to multi-shot solving. In particular, we show the following.

- An overview of multi-shot solving in the context of CASP and its applications.
- An introduction to incremental programs using the n-queens problem.
- An example on planning with durations using the spoiled Yale shooting scenario.

Parts of this chapter have been published in [75] and [13].

6.1 Multi-Shot Solving

As mentioned, a major design objective of *clingcon* 3 is to transfer *clingo*'s functionalities to CASP solving. A central role in this is played by multi-shot solving [57, 58, 59] because it allows for casting manifold reasoning modes. More precisely, multi-shot solving is about solving continuously changing logic programs in an operative way. This can be controlled via reactive procedures that loop on solving while reacting, for instance, to outside changes or previous solving results. These reactions may entail the addition or retraction of rules that the operative approach can accommodate by leaving the unaffected program parts intact within the solver. This avoids re-grounding and benefits from heuristic scores and nogoods learned over time.

To extend multi-shot solving to CASP, our propagators allow for adding and deleting constraints in order to capture evolving CSPs. Evolving constraint logic programs can be extremely useful in dynamic applications, for example, to:

- add new resources in a planning domain,
- set the value of an observed variable measured using sensors,
- add restrictions to reduce the capacity of containers, or

- increase their capacity depending on external systems like weather forecast etc.

The presented propagators provide means for all these issues. New resources can be added using additional constraint variables and domains. Values can be limited by adding constraints and rules to the constraint logic program. Due to our monotone treatment of CSPs in CASP, it is always possible to add new constraint atoms. Since they are not allowed to occur in rule heads they to not interfere with the completion of the logic program. Hence, we can combine (and therefore extend) two constraint logic programs under exactly the same restrictions that apply to normal logic programs (cf. [57, 58]).

While confining variables is easy, accomplished by adding constraints on those variables, increasing their capacity is addressed via lazy variable generation. That is, we start with a virtually maximum domain that is restrained by retractable constraints. The domain is then increased by relaxing these constraints. Importantly, the order atoms representing the active domain are only generated when needed. This avoids introducing a large amount of atoms, especially in the non-active area of the domain. As an example, consider the variable x and its domain $D(x) = \{1, \ldots, 10^9\}$ having one billion elements. By adding the constraint $x \leq 10$, only the first 10 values are valid assignments. After retracting $x \leq 10$ and adding $x \leq 20$, only the first 20 values constitute the search space. Since order atoms are only introduced in the actual search space, no atoms are introduced for the huge amount $(10^9 - 20)$ of other values. Using this technique, CASP can deal with increasing domains within reasonable space.

For illustration, let us consider the well-known n-queens puzzle for demonstrating how to incrementally add new constraints and constraint variables to a constraint logic program and how to remove constraints from it. Afterwards, we show how planning with durations can be achieved using *clingcon* 3. We therefore add durations to the actions of the famous Yale shooting problem.

6.2 Incremental Programs

To illustrate how seamlessly *clingcon* integrates CASP and multi-shot solving, we apply *clingo*'s exemplary Python script for incremental solving to model different incremental versions of the n-queens puzzle in CASP. Multi-shot solving in *clingo* relies on two directives [57, 58], the `#program` directive for regrouping rules and the `#external` directive for declaring atoms as being external to the program at hand. The truth value of such external atoms is set via *clingo*'s API. *Clingo*'s incremental solving procedure is provided in Python and loops over increasing integers until a stop criterion is met. It presupposes three groups of rules declared via `#program` directives. At step 0, the programs named `base` and `check(n)` are ground and solved for $\mathtt{n} = 0$. Then, in turn programs `check(n)` and `step(n)` are added for $\mathtt{n} > 0$ and the obtained program is grounded and solved. Other names and components are definable by appropriate changes to the Python program. Stop criteria can be the satisfiability or unsatisfiability of the respective program at each iteration. In addition, at each step `n` an external atom `query(n)` is introduced; it is set to true for the current iteration `n` and false for all previous instances with smaller integers than `n`. Although

we reproduce the exemplary Python program from *clingo*'s example pool in Listing 6.1, we must refer the reader to [57] for further details.

```
1  #script (python)

3  import clingo

5  def get(val, default):
6      return val if val != None else default

8  def main(prg):
9      imin   = get(prg.get_const("imin"), clingo.Number(0))
10     imax   = prg.get_const("imax")
11     istop  = get(prg.get_const("istop"), clingo.String("SAT"))

13     step, ret = 0, None
14     while ((imax is None or step < imax.number) and
15            (step == 0 or step < imin.number or (
16               (istop.string == "SAT"     and not ret.satisfiable) or
17               (istop.string == "UNSAT"   and not ret.unsatisfiable) or
18               (istop.string == "UNKNOWN" and not ret.unknown)))):
19         parts = []
20         parts.append(("check", [step]))
21         if step > 0:
22             prg.release_external(clingo.Function("query", [step-1]))
23             parts.append(("step", [step]))
24             prg.cleanup()
25         else:
26             parts.append(("base", []))
27         prg.ground(parts)
28         prg.assign_external(clingo.Function("query", [step]), True)
29         ret, step = prg.solve(), step+1
30 #end.

32 #program check(t).
33 #external query(t).
```

Listing 6.1: Incremental mode of *Clingo*5

6.3 Incremental N-Queens

The CASP encoding of the incremental n-queens puzzle in Listing 6.2 demonstrates the addition and removal of constraints and also shows how variable domains are dynamically increased. As usual, the goal is to put n queens on an $n \times n$ chess board such that no two queens threaten each other. Here, however, this is done for an increasing sequence of integers n such that the queens puzzle for n is obtained by extending the one for $n-1$. While the first line of Listing 6.2 includes the Python program in Listing 6.1, the next one includes the grammar from Listing 5.1. Line 3 suppresses the output of regular atoms. The remaining encoding makes use of two features of *clingo*'s exemplary incremental solving procedure, viz. subsequently grounding and solving rules regrouped under program `step(n)` and the external atom `query(n)`.[1] In Listing 6.2, all rules in lines $7-17$ are regrouped under subprogram

[1] Strictly speaking, lines $1-3$ belong to the program `base` that is treated once at the beginning (cf. Listing 6.1 and [57] for details).

```
#include "incmode.lp".
#include "csp.lp".
#show.

#program step(n).

pos(n).

&sum{ q(n) } > 0.
&sum{ q(X) } <= n :- pos(X), query(n).

&distinct{ q(X)     : pos(X) }.

&distinct{ q(X)+X-1 : pos(X) }.
&distinct{ q(X)-X+1 : pos(X) }.

&show{ q(n) }.
```

Listing 6.2: Incremental n-queens encoding Q_1 (`incqueens.lp`)

`step(n)`. The Python program Listing 6.1 makes *clingcon* in turn solve the empty program, then program `step(1)`, then program `step(1)` and `step(2)` together, then both former programs and `step(3)`, etc. This is done by keeping the previous programs in the solver and by replacing parameter `n` in lines $7-17$ with the respective integer when grounding the added subprogram. Thus, at each step `n` a fact '`pos(n).`' is added to the solver (cf. line 7). The heads of line 9 and 10 represent the linear constraints

$$q(n) > 0 \qquad \text{and} \qquad q(x) \leq n \;\; \text{for } x \in \{1, \ldots, n\} \; .$$

At each step `n`, the integer variable `q(n)` is introduced and required to be a positive integer. Moreover, all integer variables `q(1)` to `q(n)` are required to take values less or equal than n. However, while the former constraint is unconditional, the latter are subject to the external atom `query(n)`. The functioning of Listing 6.1 ensures that only `query(n)` is true while `query(s)` is false for all `s` $<$ `n`. In this way, the domain of all constraint variables `q(1)` to `q(n)` is increased by one at each step. Lines 12-15 in Listing 6.1 add distinct constraints to the effect that no two queens can be placed on the same row or diagonal of the board. Line 17 simply instructs *clingcon* to add `q(n)` to the output constraint variables.

In the following, we detail the grounding process for this example. The base program simply consists of the first 3 lines of the original encoding. Afterwards, program `step(1)` is grounded, adding the first constraints of the problem. The result is shown in Listing 6.3. The first variable `q(1)` is introduced and its lower bound is fixed to 1 in line 3. Its upper bound is also restricted to 1 but here only if `query(1)` holds. This is only the case of `n=1` when solving program `step(1)` (line 4). In all subsequent cases, `query(1)` is false, and hence $q(1) \leq 1$ is not imposed anymore. Accordingly, the atom `&sum{q(1)} <= 1` can vary freely (since it is an external constraint atom). Don't care propagation, described in Section 5.3, addresses such atoms and removes them from the system.

As solving the 1-queen problem is uninteresting, the second solving step adds

```
pos(1).

&sum{ q(1) } > 0.
&sum{ q(1) } <= 1 :- query(1).

&distinct{ q(1) }.

&distinct{ q(1) }.
&distinct{ q(1) }.

&show{ q(1) }.
```

Listing 6.3: Grounded incremental n-queens program `step(1)`.

```
pos(2).

&sum{ q(2) } > 0.
&sum{ q(1) } <= 2 :- query(2).
&sum{ q(2) } <= 2 :- query(2).

&distinct{ q(1), q(2) }.

&distinct{ q(1), q(2)+1 }.
&distinct{ q(1), q(2)-1 }.

&show{ q(2) }.
```

Listing 6.4: Grounded incremental n-queens program `step(2)`.

program `step(2)` shown in Listing 6.4. We are now solving the second step and `query(1)` will no longer be true, which amounts to removing the rule from line 4 in Listing 6.3. The new step adds two rules for this instead (lines 4–5) and restricts all variables to be less than or equal 2. Also, additional distinct constraints are added involving `q(2)`. The next step again removes the rules in lines 4–5 by making `query(2)` false and adds a new restriction (lines 4–6 in Listing 6.5). In this way, we not only add new variables at each step, but also increase the upper bounds of existing ones. For solving the third step, the grounded rules of all three steps are taken together, only `query(3)` is set to true, and all previously added instances of `query/1` are false. This leaves us with the following set of constraints that have to hold in any solution for this step.

$$
\begin{array}{lll}
q(1) > 0 & q(2) > 0 & q(3) > 3 \\
q(1) \leq 3 & q(2) \leq 3 & q(3) \leq 3 \\
\mathit{distinct}\{q(1)\} & \mathit{distinct}\{q(1), q(2)\} & \mathit{distinct}\{q(1), q(2), q(3)\} \\
 & \mathit{distinct}\{q(1), q(2)+1\} & \mathit{distinct}\{q(1), q(2)+1, q(3)+2\} \\
 & \mathit{distinct}\{q(1), q(2)-1\} & \mathit{distinct}\{q(1), q(2)-1, q(3)-2\}
\end{array}
$$

Listing 6.6 shows a run of Listing 6.2 up to 10 steps. Setting the stop criterion to `UNKNOWN` makes sure that the process neither terminates upon satisfiable nor unsatisfiable result.

```
1  pos(3).

3  &sum{ q(3) } > 0.
4  &sum{ q(1) } <= 3 :- query(3).
5  &sum{ q(2) } <= 3 :- query(3).
6  &sum{ q(3) } <= 3 :- query(3).

8  &distinct{ q(1),q(2),q(3) }.

10 &distinct{ q(1), q(2)+1, q(3)+2 }.
11 &distinct{ q(1), q(2)-1, q(3)-2 }.

13 &show{ q(3) }.
```

Listing 6.5: Grounded incremental n-queens program `step(3)`.

```
1  $ clingcon incqueens.lp -c imax=10 -c istop=\"UNKNOWN\"
2  clingcon version 3.2.0
3  Reading from incqueens.lp
4  Solving...
5  Answer: 1
6
7  Solving...
8  Answer: 1
9  q(1)=1
10 Solving...
11 Solving...
12 Solving...
13 Answer: 1
14 q(4)=2 q(3)=4 q(2)=1 q(1)=3
15 Solving...
16 Answer: 1
17 q(5)=3 q(1)=1 q(2)=4 q(3)=2 q(4)=5
18 Solving...
19 Answer: 1
20 q(6)=5 q(5)=3 q(1)=2 q(2)=4 q(3)=6 q(4)=1
21 Solving...
22 Answer: 1
23 q(7)=6 q(6)=3 q(5)=5 q(1)=2 q(2)=4 q(3)=1 q(4)=7
24 Solving...
25 Answer: 1
26 q(8)=7 q(7)=3 q(6)=1 q(5)=6 q(1)=4 q(2)=2 q(3)=5 q(4)=8
27 Solving...
28 Answer: 1
29 q(9)=3 q(8)=6 q(7)=8 q(6)=5 q(5)=2 q(1)=1 q(2)=4 q(3)=7 q(4)=9
30 SATISFIABLE
31
32 Models       : 8+
33 Calls        : 10
34 Time         : 0.075s (Solving: 0.02s 1st Model: 0.02s Unsat: 0.00s)
35 CPU Time     : 0.070s
```

Listing 6.6: Running Listing 6.2 (Q_1; `incqueens.lp`)

Measure	Q_1	Q_2	Q_3
Time	138s	10s	16s
Atoms	55k	55k	32k
Static Nogoods	24k	5k	2k
Dynamic Nogoods	1181k	320k	301k

Table 6.1: Comparison of different incremental n-queens programs.

A closer look at the distinct constraints in lines 12 – 15 of Listing 6.2 reveals quite some redundancy. This is because the constraints added at each step supersede the ones added previously, and they all coexist in the system. For example, at Step 3 the system contains 3 instances of line 12, namely `&distinct{q(1)}`, `&distinct{q(1),q(2)}`, and `&distinct{q(1),q(2),q(3)}`. Clearly, the first two constraints are redundant in view of the third but remain in the system. To avoid this redundancy, we can make use of the external atom `query(n)` to remove the redundant distinct constraints at each step in the same way we tighten the upper bound of variable domains. This amounts to replacing lines 12 – 15 in Listing 6.2 with the ones given in Listing 6.7.

```
12 &distinct{ q(X)      : pos(X)} :- query(n).

14 &distinct{ q(X)+X-1  : pos(X)} :- query(n).
15 &distinct{ q(X)-X+1  : pos(X)} :- query(n).
```

Listing 6.7: Retracting Constraints, encoding Q_2

Although the last modification guarantees that the system bears no redundant distinct constraints,[2] it leads to adding and removing the same restrictions over and over again. For example, the constraint that `q(1)` and `q(2)` must have different values is included in every distinct constraint after step 1. And this information is retracted and re-added at each step. This is avoided by the constraints in Listing 6.8. This formulation only adds constraints for the new variable `q(n)` at each step `n` and stays clear from retracting any constraints.

```
12 &sum{ q(X)     } != q(n)      :- X=1..n-1.

14 &sum{ q(X)+X-1 } != q(n)+n-1 :- X=1..n-1.
15 &sum{ q(X)-X+1 } != q(n)-n+1 :- X=1..n-1.
```

Listing 6.8: Partial Constraints, encoding Q_3

Evaluation Table 6.1 gives a comparison of the three different encodings for the incremental n-queens problem for 30 steps. The first row gives the respective total running time of *clingcon* 3.2.0. The second one reports the total number of introduced atoms. The third one gives the sum of static nogoods generated at each step, and the

[2] Given that don't care propagation is enabled by default.

```
1 #include "incmode.lp".
2 #include "csp.lp".

4 #program base.
5 action(wait). action(load). action(shoot).
6 duration(load,25). duration(shoot,5). duration(wait,36).
7 unloaded(0).
8 &sum { at(0) } = 0.
9 &sum { armed(0) } = 0.
```

Listing 6.9: Yale shooting instance

last one the sum of dynamic nogoods ∇ generated by lazy constraint propagation. We observe that the initial encoding Q_1 performs worst in all aspects. The inherent redundancy of Q_1 is reflected by the high number of dynamic nogoods generated by the constraint propagator. This is the source of its inferior overall performance. Unlike this, the two alternative approaches bear less redundancy, as reflected by their much lower number of dynamic nogoods. In Q_2, this is achieved by eliminating duplicate inferences from redundant constraints. Although Q_3 even further reduces the number of atoms as well as static and dynamic nogoods, its runtime is slightly inferior. This is arguably due to the usage of elementary linear constraints rather then global distinct constraints (and the pigeon hole constraints which are enabled by default).

6.4 Planning with Durations

To show how constraint answer set solving can be used in the domain of planning, we adapted the spoiled Yale shooting scenario from [27]. The goal is to kill turkeys. To this aim, we have a gun and two actions, load and shoot. If we load, the gun becomes loaded. If we shoot, it kills the turkey, if the gun was loaded for no more than 35 minutes. Otherwise the gun powder spoils. We model this incremental planning problem using CASP. We start by including the incremental Python program and the CSP grammar in the first two lines of Listing 6.9. This listing is the base program. All actions and their durations are introduced in Lines 5 and 6. At the initial situation, the gun is unloaded as described in Lines 7 to 9. Listing 6.10 characterizes the dynamic part of the problem; it is grounded for each step `n`. Line 3 enforces that exactly one action is done per step. The exact time every step takes place is denoted by the integer variables `at(n)`. The difference between two consecutive time steps is exactly the duration of the respective action (Line 4). The next three lines ensure the inertia of the fluents, e.g., the gun stays loaded/unloaded if it was loaded/unloaded before, and the turkey stays dead. Lines 10 and 11 use the integer variable `armed(n)` to denote for how long the weapon is loaded at step `n`. Whenever it is unloaded, `armed(n)` is 0, otherwise it is increased by the duration of the last action. The upcoming four lines (13–16) encode the conditions and effects of the actions. When we load the gun, it becomes loaded, when we shoot, it becomes unloaded. If we shoot and the gun was loaded for no longer than 35 minutes (and thus the gun powder is unspoiled),

```
1  #program step(n).

3  1{do(X,n) : action(X)}1.
4  &sum { at(n); -1*at(N') } = D :- do(X,n); duration(X,D); N'=n-1.

6  loaded(n)   :- loaded(n-1); not unloaded(n).
7  unloaded(n) :- unloaded(n-1); not loaded(n).
8  dead(n)     :- dead(n-1).

10 &sum { armed(n) } = 0 :- unloaded(n-1).
11 &sum { armed(n); -1*armed(N') } = D :- do(X,n); duration(X,D); N'=n-1; loaded(N').

13 loaded(n)   :- do(load,n).
14 unloaded(n) :- do(shoot,n).
15 dead(n)     :- do(shoot,n); &sum { armed(n) } <= 35.
16 :- do(shoot,n), unloaded(n-1).
```

Listing 6.10: Spoiled Yale shooting scenario

```
1  #program check(n).
2  :- not dead(n); query(n).
3  :- not &sum { at(n) } <= 100; query(n).
4  :- do(shoot,n); not &sum { at(n) } > 35.
```

Listing 6.11: Query for the Yale Shooting Scenario.

then the turkey is dead. The last line ensures that we cannot shoot if the gun is not loaded. Together with the initial situation and the actions from Listing 6.9 this encodes the spoiled Yale shooting problem, and any solution represents an executable plan. Listing 6.11 adds a query to our problem. In Line 2 we ensure that the turkey is dead in step `n`. Because we added the external atom `query(n)` to this rule, it is only active if we are actually solving step `n` and is removed in all other steps. The next line ensures that we kill the turkey within 100 minutes. And as an additional constraint, we added some preparation time such that we are not allowed to shoot in the first 35 minutes. It is possible to solve this problem within three steps.

There exist two constraint stable models $(X, \mathbf{C})$ at this time point:[3]

$X \cap \mathcal{A}$			constraint assignment	
	`unloaded(0)`,		$at(0) \mapsto 0$,	$armed(0) \mapsto 0$,
`do(wait,1)`,	`unloaded(1)`,		$at(1) \mapsto 36$,	$armed(1) \mapsto 0$,
`do(load,2)`,	`loaded(2)`,		$at(2) \mapsto 61$,	$armed(2) \mapsto 0$,
`do(shoot,3)`,	`unloaded(3)`,	`dead(3)`	$at(3) \mapsto 66$,	$armed(3) \mapsto 5$
	`unloaded(0)`,		$at(0) \mapsto 0$,	$armed(0) \mapsto 0$,
`do(load,1)`,	`loaded(1)`,		$at(1) \mapsto 25$,	$armed(1) \mapsto 0$,
`do(load,2)`,	`loaded(2)`,		$at(2) \mapsto 50$,	$armed(2) \mapsto 25$,
`do(shoot,3)`,	`unloaded(3)`,	`dead(3)`	$at(3) \mapsto 55$,	$armed(3) \mapsto 30$

which means that we either wait before loading and shooting, or load the gun instead of waiting, such that the gun is loaded twice before shooting.

Conclusion We have shown that multi-shot solving can be extended to the paradigm of CASP. Solving planning or scheduling problems involving ressources can now be done in an incremental way. Especially the possibility to increase the domains of the variables will be a helpful feature in modeling these problems. This cannot be done easily with a translational approach, due to the grounding of variables with huge domains.

[3] $\mathcal{A}$ denotes the set of regular atoms.

Chapter 7

Related Work

This chapter gives an overview on the area of CASP. We compare different semantics and their implementations. We start with a definition of our semantics in terms of the more general framework of ASP modulo Theories [54]. Afterwards, a comprehensive list of features of available CASP systems is presented and their pros and cons are discussed. A subset of this list has already been published in [75].

7.1 Logic Programs Modulo Theories

According to [54], a logic program P modulo theories is a logic program over two disjoint alphabets, $\mathcal{A}$ and $\mathcal{T}$, consisting of regular and *theory atoms*. In what follows, we simplify the definition by concentrating only on one theory T. Accordingly, P is a set of rules of the form (2.1) over atoms $\mathcal{A} \cup \mathcal{T}$. In analogy to input atoms from `#external` directives [57], $\mathcal{T}$ is partitioned into *defined* theory atoms $\mathcal{T} \cap \mathit{head}(P)$ and *external* theory atoms $\mathcal{T} \setminus \mathit{head}(P)$. In order to reflect different forms of theory propagation, $\mathcal{T}$ is partitioned into *strict* theory atoms $\mathcal{T}_e$ and *non-strict* theory atoms $\mathcal{T}_i$, i.e., $\mathcal{T}_e \cap \mathcal{T}_i = \emptyset$ and $\mathcal{T}_e \cup \mathcal{T}_i = \mathcal{T}$. The strict theory atoms in $\mathcal{T}_e$ resemble the treatment of reified constraints ($\mathbf{T}a \Leftrightarrow c$), where a is a theory atom and c is a constraint. Non-strict theory atoms in $\mathcal{T}_i$ are treated like half-reified constraints ($\mathbf{T}a \Rightarrow c$).[1]

A set $\mathcal{S} \subseteq \mathcal{T}$ is called a T-solution, if T is consistent with the conditions expressed by elements of $\mathcal{S}$ as well as complements of conditions associated with the false strict theory atoms $\mathcal{T}_e \setminus \mathcal{S}$. We identify the theory T with a set Δ_T of *theory nogoods* such that, given a total assignment $\mathbf{B}$, we have that $\delta \subseteq \mathbf{B}$ for some $\delta \in \Delta_T$ iff there is no T-solution $\mathcal{S}$ such that $\{a \mid a \in \mathcal{T}, \mathbf{T}a \in \mathbf{B}\} \subseteq \mathcal{S}$ and $\{a \mid a \in \mathcal{T}_e, \mathbf{F}a \in \mathbf{B}\} \cap \mathcal{S} = \emptyset$. That is, the nogoods in Δ_T reject $\mathbf{B}$ iff no T-solution (i) includes all theory atoms in $\mathcal{T}$ that are assigned to true by $\mathbf{B}$ and (ii) excludes all strict theory atoms in $\mathcal{T}_e$ assigned to false by $\mathbf{B}$. This semantic condition establishes a (one-to-one) correspondence between T-stable models of P and solutions for $(\Delta_P^{\mathcal{T}\setminus \mathit{head}(P)} \cup \Lambda_P^{\mathcal{T}\setminus \mathit{head}(P)}) \cup \Delta_T$.

[1] That is, $\mathcal{T}_e$ stands for equivalences and $\mathcal{T}_i$ for implication.

Proposition 7.1.1

Let P be a logic program over theory T with regular atoms $\mathcal{A}$ and theory atoms $\mathcal{T}$.
Then, $(\mathbf{B}|_{\mathcal{A}\cup\mathcal{T}})^{\mathbf{T}}$ is a T-stable model of P iff $\mathbf{B}$ is a solution of $\Delta_P^{\mathcal{T}\setminus head(P)} \cup \Lambda_P^{\mathcal{T}\setminus head(P)} \cup \Delta_T$.

This proposition is taken from [54] where the definition of the completion exactly coincides with the definition of the completion for logic programs with externals.

CASP as Logic Program over Theories We now express a constraint logic program as a logic program over theories.

Theorem 7.1.2

Let P be a constraint logic program over regular atoms $\mathcal{A}$ and constraint atoms $\mathcal{C}$ associated with the CSP $T = (\mathcal{V}, D, C)$. We let $\mathcal{C}$ be the set of theory atoms, $\mathcal{C}_e = \mathcal{C}$, $\mathcal{C}_i = \emptyset$, and $head(P) \cap \mathcal{C} = \emptyset$.
Then, $(X, \mathbf{C})$ is a constraint stable model of P iff X is a T-stable model of P.

Proof 7.1.1 *This theorem directly follows from Proposition 7.1.1 and Theorem 4.3.2, as the definitions are identical in the case of CASP.*

Intuitively, in a constraint logic program, all constraint atoms are strict, and they are not allowed to occur in any head of a rule. Therefore, CASP is an instance of logic programs modulo CSP.

7.2 Comparing Different Semantics and Systems

Here we give an overview of the development of CASP, its different semantics, languages and systems. We start with the groundwork that was done with the development of the language $\mathcal{AC}$ and go on alphabetically. For each approach to CASP, we then give an example of its syntax, its restrictions and possibilities, and explain the semantic differences. Also, the corresponding systems implementing the approach and a short description of the used technologies are given. Table 7.1 summarizes their features. The first column "translation" depicts whether the system translates CASP to ASP, SMT or Mixed Integer Linear Programming (MILP). Hence, once the input program is translated to the target paradigm, only a solver for the target formalism is needed. This is also one of the big advantages of the translational approaches. They profit from the features and performance of the respective target formalisms e.g. SMT, MILP or ASP, and no new solver needs to be developed. A drawback is the translation itself, that can result in large propositional representations or weak propagation strength. The second column marks systems that use an explicit representation for the integer variables. This is the case when using some form of encoding and usually results in

a large number of propositional atoms to represent variables with large domains. If non-linear constraints like quadratic equations and global constraints are supported, this is shown in the next column. Afterwards, the support for real numbers and optimization over integer or real numbers is depicted. The last column marks whether the used formalism supports a system intrinsic way to handle default values for integer or real variables.

	translation	explicit	non-linear	reals	optimization	defaults
acsolver	-	-	✓	✓	-	-
adsolver	-	-	-	-	-	-
aspartame	✓	✓	-	-	✓	-
aspmt2smt	✓	-	✓	✓	-	✓
bfasp	✓	✓[1]	-	-	✓	✓
clingo[DL]	-	-	-	-	-	-
clingo[LP]	-	-	-	✓	✓[2]	-
clingcon 1	-	-	✓	-	✓	-
clingcon 2	-	-	✓	-	✓	-
clingcon 3	✓[3]	✓[1]	✓[4]	-	✓	-
dingo	✓	-	-	-	-	-
dlvhex[CP]	-	-	✓	-	✓	-
ezcsp	-	-	✓	✓	-	-
ezsmt	✓	-	✓	✓	-	-
inca	-	✓	✓	-	✓	-
lc2casp	✓[3]	✓[1]	✓[4]	-	✓	✓
mingo	✓	-	-	✓[5]	-	-
minisatid	-	✓[1]	✓	-	✓	-
msvm	✓	✓	-	-	-	✓

[1] Can be generated lazily.
[2] Optimization is done locally for every stable model.
[3] Allows for partial translation of the problem.
[4] Translation of distinct into linear constraints.
[5] Only for the variables, not any constants.

Table 7.1: Feature comparison of different systems.

$\mathcal{AC}$-programs To the best of our knowledge, $\mathcal{AC}$ was the first approach to create a CASP language and semantics, combining ASP and CR-Prolog. $\mathcal{AC}$ uses regular and constraint sorts. The idea is to avoid grounding constraint variables and only ground variables over regular sorts. It furthermore distinguishes between regular

predicates (over regular sorts), constraint predicates (simple arithmetic expressions over constraint sorts), defined predicates (defined in terms of constraint predicates), and mixed predicates (over regular and constraint sorts). Thus $\mathcal{AC}$ allows constraint variables to occur in predicates in the head of rules. Two implementations for this language are proposed in [90]. The *acsolver* is based on *lparse*, *Surya* and *CLP(R)*. The *adsolver* in turn, on *lparse*, *smodels* and a self-made, incremental difference logic solver. Both solvers cannot handle defined predicates. They use classical ASP solving techniques such as DPLL-style backtracking [33] and therefore support no advanced learning features. Nevertheless, they support theory propagation on partial assignments. Program P_1 from Example 1 can be written in the following *adsolver* syntax.

```
1  :- constants
2     x :: real[0..23].

4  switchOn  <- not switchOff.
5  switchOff <- not switchOn.
6  light <- switchOn.
7  light <- not night.
8  night <- time(x) & x=X & X < 7.
9  night <- time(x) & x=X & X >= 22.
10 sleep <- switchOff & night.
```

The solutions for this program are the same as in (2.10).

aspartame The *aspartame* system (Section 4.4) was not developed as a CASP system but as a CP solver. It uses an ASP encoding to solve CSP problems by translating the constraints into a logic program using the order encoding. It is possible to combine a CSP with a logic program to form a constraint answer set problem, although the encoding itself is not elaboration tolerant. This can be seen from the syntax for Example 1.

```
1  var(bool,"early"). var(bool,"late").
2  var(int,"x",range(0,23)).
3  switchOn  :- not switchOff.
4  switchOff :- not switchOn.
5  light :- switchOn.
6  light :- not night.
7  night :- not p("early",0).
8  night :- not p("late",0).
9  sleep :- switchOff, night.
10 constraint(1,op(neg,"early")).
11 constraint(1,op(le,op(mul,1,"x"),6)).
12 constraint(2,"early").
13 constraint(2,op(le,op(mul,-1,"x"),-7)).
14 constraint(3,op(neg,"late")).
15 constraint(3,op(le,op(mul,-1,"x"),-22)).
16 constraint(4,"late").
17 constraint(4,op(le,op(mul,1,"x"),21)).
```

To produce an encoding with the same semantics as in our example, we currently have to reify the constraints with Boolean variables manually. Therefore, we add two clauses for $\mathbf{T}early \Leftrightarrow x < 7$ and $\mathbf{T}late \Leftrightarrow x \geq 22$ respectively (lines 10−13 and lines 14−17). With $p(early, 0)$ and $p(late, 0)$ we can access the truth value of the Boolean constraint variables *early* and *late*, where $p(early, 0)$ means that *early* is false.

$\mathcal{ASPMT}$ The functional stable model semantics [18] defines non-monotonic reasoning on non-Boolean variables. This allows for expressing, i.e. defaults in an elaboration tolerant way. Consider the following bath tub example. A tub starts with an amount of 100 units of water. It can be refilled by using the tap, actually gaining one unit every time step. The CASP program looks like this:

$$\leftarrow \sim amount_T = amount_{T+1}, \sim opentap_T$$
$$\leftarrow \sim amount_T + 1 = amount_{T+1}, opentap_T$$

The first rule is stating that the amount of water stays the same when the tap is closed, while the second rule increases the amount by one for every time step where the tap is open. If we now add a possibility to empty the tub, like pulling the plug, we need to change the first rule to account for this new action.

$$\leftarrow \sim amount_T = amount_{T+1}, \sim opentap_T, \sim pullplug_T$$
$$\leftarrow \sim amount_T + 1 = amount_{T+1}, opentap_T$$
$$\leftarrow \sim amount_T - 1 = amount_{T+1}, pullplug_T$$

Now, the first rule states that the amount of water stays the same when the tap is closed and we have not pulled the plug. As we have to change the rules when adding an action, this is not elaboration tolerant. $\mathcal{ASPMT}$ can use integer variables in the head of rules, and therefore solves this problem elegantly. The following program

$$amount_{T+1} = X \leftarrow \sim\sim amount_{T+1} = X, amount_T = X$$
$$amount_{T+1} = X \leftarrow amount_T = X - 1, opentap$$

solves this problem by using a default. The first rule (using a double negation) actually says that the amount stays the same, if there is no reason to believe something else. This program can now easily be extended by other actions without the need to change the first rule.

$$amount_{T+1} = X \leftarrow amount_T = X + 1, pullplug$$

In [18], it is shown that CASP is a special case of $\mathcal{ASPMT}$ where the theory is fixed to CSP and intensional constants are limited to propositional constants only, and do not allow function constants.

While the first implementation called *msvm* eliminates the intensional functions by grounding them and afterwards solving the problem using *gringo* and *clasp*, a second implementation called *aspmt2smt* translates a logic program with constraints into SMT. It uses the grounder *gringo* and the SMT solver *z3*, and can therefore also handle constraints over real values. Unlike with *ezsmt*, the SMT solver cannot be changed in this system and is hard wired. The implementation is also restricted to tight logic programs. Program P_1 from Example 1 can be written in the following syntax.

```
1  :- constants
2     x :: real[0..23].
3  {time(x) = V}.
4  switchOn  <- not switchOff.
5  switchOff <- not switchOn.
6  light <- switchOn.
7  light <- not night.
8  night <- time(x)=X & X < 7.
9  night <- time(x)=X & X >= 22.
10 sleep <- switchOff & night.
```

We see in line 3 that we need to have an (in this case empty) reason for the variable `time(x)` to get a value.

Bound Founded ASP The bound founded approach [3, 4] defines CP variables similar to $\mathcal{ASPMT}$ and HT_C (see below). CP variables can occur in assignments in the head of a rule. Instead of justifying all possible values for the head variable, bound founded ASP only justifies the minimal (maximal) value that can be derived by a rule. Its implementation extends the lazy nogood generating CP solver *chuffed*.

***clingo*[DL]** The *clingo* derivative *clingo*[DL] [75] accepts a subset of the theory of linear constraints namely *Quantifier-Free Integer Difference Logic* (QF-IDL) dealing with constraints of the form $x - y \leq k$, where x and y are integer variables and k is an integer constant. Despite its restriction, QF-IDL can be used to naturally encode timing related problems, e.g., scheduling or timetabling, and provides the additional advantage of being solvable in polynomial time. *clingo*[DL] uses *clingo*'s theory interface [54] to integrate a propagator that checks during search whether the current set of implied difference constraints is satisfiable. Using *clingo*'s theory language, the syntax for program P_1 from Example 1 is:

```
1 &diff{ x-0 } <= 23.
2 &diff{ 0-x } <= 0.
3 switchOn  :- not switchOff.
4 switchOff :- not switchOn.
5 light :- switchOn.
6 light :- not night.
7 night :- &diff{ x-0 } <= 6.
8 night :- &diff{ 0-x } <= -22.
9 sleep :- switchOff, night.
```

To use the *strict* semantics (see Section 7.1) on theory atoms, we use the command line option `-c strict=1`. This system is part of the potassco tool collection `https://potassco.org/labs/clingodl/`.

***clingo*[LP]** The *clingo* derivative *clingo*[LP] [75] combines ASP with the theory of linear constraints over either reals or integers. As *clingo*[DL] above, it uses the same theory interface to check whether the current set of implied linear constraints is satisfiable. On that account, it uses the Linear Programming (LP) solver *cplex* or *lpsolve*. The syntax for program P_1 from Example 1 is

```
1 &dom{0..23} = x.
2 switchOn  :- not switchOff.
3 switchOff :- not switchOn.
4 light :- switchOn.
5 light :- not night.
6 night :- &sum{x} < 7.
7 night :- &sum{x} >= 22.
8 sleep :- switchOff, night.
```

Strict semantics is enabled using the command line option `-c nstrict=0`. This system is also available at `https://potassco.org/labs/clingolp/`.

clingcon CASP as defined in Section 2.1 and 2.4 is proven to be a syntactic variant of $\mathcal{AC}^-$ which is $\mathcal{AC}$ without defined predicates [80]. It does not allow constraints to occur in the head of rules. We have developed three different versions of *clingcon*, that implement CASP.

***clingcon* 1** uses modern CDCL techniques with learning as presented in Section 3.4, and is based on *gringo*, *clasp*, and *gecode*. It checks partial assignments for consistency with the theory and does theory propagation to infer knowledge from the CP solver. The used CP solver is incremental, meaning that there is no reason to recompute the CP part for every propagation step. To the best of our knowledge, it is also the first CASP solver that supports objective functions on the constraint variables, being able to handle optimization problems.

***clingcon* 2** is an extension of *clingcon* 1, introducing reason and conflict reduction. We showed in Section 3.7 that the learning capabilities of the systems are improved by this technique. *clingcon* 1 and 2 share the same syntax. Program P_1 from Example 1 can be written as follows.

```
1 $domain(0..23).
2 switchOn  :- not switchOff.
3 switchOff :- not switchOn.
4 light :- switchOn.
5 light :- not night.
6 night :- x $< 7.
7 night :- x $>= 22.
8 sleep :- switchOff, night.
```

***clingcon* 3** uses the modular syntax of *gringo* [53] as shown in Section 5.2.2. The syntax for program P_1 from Example 1 has therefore been changed to the following.

```
1 &dom{0..23} = x.
2 switchOn  :- not switchOff.
3 switchOff :- not switchOn.
4 light :- switchOn.
5 light :- not night.
6 night :- &sum{x} < 7.
7 night :- &sum{x} >= 22.
8 sleep :- switchOff, night.
```

As depicted in Section 5.2, it supports lazy nogood and lazy variable generation

by using a self made CP propagator for linear constraints. It can also handle multi-shot problems by solving them incrementally.

dingo The translator *dingo* [76] combines ASP with Difference Logic (DL;[20, 50]) constraints. All these constraints of the form $x+k \geq y$ can efficiently be handled using a cycle detection algorithm for a weighted graph. It translates the CASP program into difference logic. It thereby preserves the ASP semantics for Boolean atoms and treats constraints similar to CASP. While translating the completion of the program into linear constraints is easy, a level ranking characterization [85] is used to capture the loop formulas of a logic program. Afterwards, an SMT solver with support for difference logic, for example *z3*, can be used to solve the problem. Program P_1 from Example 1 can be written in the following syntax.

```
1  int(x).
2  int(zero).
3  dl_le(zero,zero,0).
4  dl_le(x,zero,23).
5  dl_le(zero,x,0).
6  switchOn  :- not switchOff.
7  switchOff :- not switchOn.
8  light :- switchOn.
9  light :- not night.
10 night :- dl_le(x,zero,6).
11 night :- dl_le(zero,x,-22).
12 sleep :- switchOff, night.
```

Constraint Hex Programs Constraint Hex Programs have been developed in [37, 112]. The definition of constraint stable models is based on the constraint reduct as defined in Section 2.4. The stable models themselves are defined using the Faber-Leone reduct [47] unlike the Gelfond-Lifschitz [66] reduct we are using. In the solver *dlvhex*[cp], constraint hex programs are then translated to standard hex programs with the special case of CP as theory. The implementation also uses *gringo* and *clasp* and the CP solver *gecode*. Recent versions of DLVHex [105] support theory propagation on partial assignments and apply an irreducible inconsistent set algorithm, like the ones shown in Section 3.5. According to [37, 112], performance is slightly worse than the one of *clingcon* 2. These two systems are very similar to each other and therefore the syntax for Example 1 is the same.

$\mathcal{EZ}$ Designed as a description language for CP problems, $\mathcal{EZ}$ was presented in [5, 9]. Given a logic program, certain predicates in the answer set (stable model) X like `cspdomain(fd)`, `cspvar(x,1,10)` and `required(x < 7)` form a CSP. If there exists a solution $\mathbf{C}$ to the CSP, then X is called a weak answer set to the problem. Both solutions together $\langle X, \mathbf{C}\rangle$ are called extended weak answer set. $\mathcal{EZ}$ is a syntactic variant of language $\mathcal{AC}^-$ where constraints occur only in integrity constraints [80]. The language does not support constraints directly in rules but allows the predicate *required* in the head. Example 1 using the *required* predicate can be written in the $\mathcal{EZ}$ language as:

```
1  cspdomain(fd).
2  cspvar(x,0,23).
3  switchOn  :- not switchOff.
4  switchOff :- not switchOn.
5  light :- switchOn.
6  light :- not night.
7  night :- not day.
8  day   :- not night.
9  required(x<7 \/ x>21) :- night.
10 sleep :- switchOff, night.
```

Note that we had to introduce an additional atom `day` to cope with constraints in rule bodies. In a preprocessing step, a simple integrity constraint is added to connect the *required* atom with a constraint.

$$\leftarrow required(x < 7 \bigvee x > 21), \sim(x < 7 \vee x > 21)$$

Due to the nature of the language, constraints are treated as non-strict (see Section 7.1), i.e. $\mathbf{T}a \Rightarrow c$ for an atom a and a constraint c. This is because the constraint c is part of the constraint problem if a is in a stable model, but $\bar{c}$ does not have to be part of the CP if a is not. This explains the restriction that constraints are only allowed in integrity constraints. Consider the following line of our example program, as if we would have used the constraint in the body:

$$night \leftarrow (x < 7)$$

It has the following extended weak answer sets:

X	$\mathbf{C}$
$\{\}$	$x \in \{0, \ldots, 23\}$
$\{night, (x < 7)\}$	$x \in \{0, \ldots, 6\}$

where $D(x) = \{0, \ldots, 23\}$. We see that the first line contains solutions where e.g. $x = 0$ but *night* is not true. In fact, $\mathcal{EZ}$ yields unexpected results if constraints are used in rule bodies of non-integrity constraints. This observation was also made in [10, 75]. We are aware of two implementations that directly use $\mathcal{EZ}$ as an input language.

ezcsp supports three different integration schemas [9]. The black-box integration schema is a loose coupling of the ASP and the CP solver. *ezcsp* uses *smodels* and *clasp* as ASP solvers and *bprolog* or *sicstus prolog* as CP solvers. Once a stable model is computed by the ASP solver, it is translated into a CSP and checked with the external CP solver. If the check fails, another solution is computed. Therefore, all knowledge gathered so far is lost and the systems are started from scratch, adding an integrity constraint to avoid finding the same solution again. The grey-box integration schema uses an incremental ASP solver instead, keeping the learnt knowledge after every stable model. Finally, the white-box integration schema is a tight integration of the CP solver into the ASP solving

process. This schema allows for theory propagation on partial assignments and is the strongest approach wrt. CDCL learning. Since *ezcsp* uses a *Prolog* system as CP solver it can also handle constraints over real numbers. On the other hand, it cannot compute optimal solutions wrt. an objective function.

ezsmt reuses the input language $\mathcal{EZ}$ [82],[2] and translates tight CASP programs to the SMT-LIB standard format [17]. These SMT programs can then be solved by any SMT solver, for example *z3* [36]. The translation technique is easy, as simply the nogoods of the completion (with the constraints as external atoms) need to be converted to clauses for the SMT solver. As *ezsmt* can only handle tight logic programs, no loop formulas are needed.

For an extensive comparison of $\mathcal{AC}$, CASP, and $\mathcal{EZ}$, we refer to [80].

inca The *inca* [43] system implements CASP semantics as described in Section 2.4. It uses an extended CDCL algorithm similar to the one proposed in Algorithm 12. It generated lazy nogoods using a propagator for linear constraints and various other propagators for the distinct constraint. Since *inca* also uses the order encoding but no lazy variable generation, $\Phi(\mathcal{V}, D)$ and $\mathcal{O}_{\mathcal{V}}$ have to be introduced beforehand. This prevents *inca* from handling variables with huge domains. Program P_1 from Example 1 can be written in the following syntax.

```
1 #var $x = 0..23.
2 switchOn  :- not switchOff.
3 switchOff :- not switchOn.
4 light :- switchOn.
5 light :- not night.
6 night :- $x #< 7.
7 night :- $x #>= 22.
8 sleep :- switchOff, night.
```

Here and There with Constraints The theory of Here and There with Constraints (HT_C;[26]) also combines default reasoning with constraints similar to $\mathcal{ASPMT}$. It actually extends the logic of Here and There (HT) [72] to handle constraints. It is therefore more general than CASP. A special case, HT with linear constraints uses constraint assignments in the head of a rule and constraint checks in the body. Variables take a value if they are defined and stay unassigned otherwise. The bath tub problem can be encoded in the following elaboration tolerant way.

```
1 &assign {amount(T+1) := amount(T)} :- not &sum{amount(T+1)} != amount(T).
2 &assign {amount(T+1) := amount(T)+1} :- opentap.
3 &assign {amount(T+1) := amount(T)-1} :- pullplug.
```

In [26], we provide a translation for this special case of HT_C with linear constraints to

[2] http://www.unomaha.edu/college-of-information-science-and-technology/natural-language-processing-and-knowledge-representation-lab/software/ezsmt.php

CASP. We developed the translator *lc2casp*[3] that transforms a program written in a *gringo* 5 syntax extension, used to represent these programs, into the input language of *clingcon* 3. Lazy variable generation is needed, as the translation step cannot provide domains for the variables. The syntax for program P_1 from Example 1 has therefore to be adapted using assignments.

```
1 &dom{0..23} = x.
2 &assign {x := 0..23}.
3 switchOn  :- not switchOff.
4 switchOff :- not switchOn.
5 light :- switchOn.
6 light :- not night.
7 night :- &sum{x} < 7..
8 night :- &sum{x} >= 22.
9 sleep :- switchOff, night.
```

In line 2, we use an unconditioned assignment to allow x to take any value. This is necessary to emulate the semantics of CASP and resembles the same rule that was added for the semantics of $\mathcal{ASPMT}$.

mingo The *mingo* system translates ASP in combination with linear constraints [84] to Mixed Integer Programming (MIP). *Mingo* follows the same approach as *dingo* [76]. The translated program is a set of linear constraints which can be solved by a MIP solver like *cplex*. Due to the nature of these solvers, constraint variables over reals are possible. Program P_1 from Example 1 can be written in the following syntax.

```
1  int(x).
2  mgeq(1,x,0).
3  mleq(1,x,23).
4  switchOn  :- not switchOff.
5  switchOff :- not switchOn.
6  light :- switchOn.
7  light :- not night.
8  night :- mleq(1,x,6).
9  night :- mgeq(1,x,22).
10 sleep :- switchOff, night.
```

idp The *idp* system [34] can solve typed first order logic with inductive definitions ($FO(ID)$). It supports arithmetic constraints over integers, similar to CASP. Unlike CASP, it uses a typed input language. The implementation uses an extension of *minisat*, called *minisatid*, a CDCL-based SAT solver with lazy clause and variable generation. It uses an algorithm similar to Algorithm 12 but based on SAT instead of ASP.

[3] http://www.cs.uni-potsdam.de/lc2casp/

Chapter 8

Conclusion

By combining the declarative modeling approach of ASP with solving capabilities of CP, we made CASP applicable to real world, industrial size problems that involve resources like machine and railway scheduling, factory layouting, SQL based query optimization, etc. The main challenge to overcome was to combine the handling of huge domains with dedicated learning techniques to keep the desired performance. In addition to this, we used reactive solving techniques to enhance the versatility of the approach even more. It is now the first time that we can combine all the benefits that we have from ASP, having an elaboration tolerant, declarative framework and a fast solving technology (CDCL), with the possibilities to handle resources using non-Boolean variables, intrinsic to CP. This thesis gives an overview on CASP, its origins and usages, and evaluates different techniques for handling it.

To define CASP, we pursue a semantic approach that is based on a propositional language. Following the approach of SMT, we abstract from the constraints in a specialized theory. By replacing the semantics of SAT with the one of ASP, we had to account for the modeling language of ASP and also change the underlying solver technology. This led to the development of a full fledged general theory language that is easily extendible and can handle arbitrary kinds of theories [53]. With the propositional characterization, we developed an extension of the CDCL algorithm for handling constraint logic programs, by using a modern CP solver to check partial assignments for consistency. This resulted in the modern CASP solver *clingcon* 1, combining the ASP solver *clasp* with the black box CP solver *gecode*. We found out that propagation on partial assignments is important on most problems. Furthermore, special filtering techniques are applied to improve the interplay of CDCL and external CP solvers.

A different approach is pursued by translating a CSP into nogoods. We developed an elaboration tolerant ASP encoding to handle this translation using the order encoding. The translated CSP can be solved using a highly optimized ASP solver. Unfortunately, performance degrades with increasing domain size of the integer variables. We found out that dedicated preprocessing techniques are needed to further improve on grounding.

Solvers that translate CSPs into SAT or ASP profit from the raw speed and the learning capabilities of CDCL. Unfortunately, they suffer from the grounding

bottleneck in case of large domains. Some encoding techniques (like the logarithmic encoding) have been developed to reduce grounding size but sacrifice propagation strength. This led to the development of lazy nogood generating solvers. They improve on the grounding bottleneck, as an exponential number of nogoods is represented implicitly. Furthermore, learned nogoods can be forgotten without sacrificing completeness using standard CDCL techniques. Unfortunately, the representation of the variables has to be made explicit which is linear in size of the domain for the order encoding. This results in large or impossible groundings for huge domains. Lazy variable generation is a solution to this. As there is no need to represent variables in advance, it perfectly complements lazy nogood generation and results in excellent performance. Therefore, combining the advantages of the learning approach using a CDCL algorithm with the translation method using the order encoding, a lazy nogood generating system was devised. It was extended with lazy variable generation to cope with huge domains. It supports dedicated preprocessing features, a generic input language, and the possibility of translating a part of the CSP. The evaluation shows that a high configurability is important to cope with different problems.

This lazy approach enabled us to development constraint multi-shot solving, which is a useful paradigm for solving planning problems with resources. It allows us to gradually build up a system instead of solving a huge problem at once, also reacting to outside behaviour. The devised system can add and remove constraints as well as increase or decrease the domain of variables.

Finally, an overview of other CASP systems and paradigms is given. This comparison has shown that our systems provide a lot of features, such as multi-shot solving and optimization, in combination with good performance.

While CASP is a perfect paradigm for solving problems involving resources, encoding them using a language that naturally handles defaults can be beneficial. Extending the framework of CASP with non-monotone constraint variables like it was done in $\mathcal{ASPMT}$, *Here and There with Constraints*, and Bound Founded ASP definitively increases the elaboration tolerance and modeling convenience. In [26], we translate the theory of Here and There with Linear Constraints into CASP. We presented a tool called *lc2casp* that extends our input language by exactly these features, compiling the result back to CASP, to be readily solved with *clingcon* 3. So no new solving paradigms or systems need to be developed for this enhanced CASP approach.

In the future, encodings specifically designed for lazy variable generation need to be explored. It has proven to be a well suited method for handling CP, and redundant information or large groundings are no longer the bottlenecks, rather propagation strength and complexity get into focus. Therefore, also combinations of encodings (order, range, direct) may be beneficial in lazy variable generating approaches. Another point that can be learned from conventional CP solvers is heuristics. As ASP can modify the heuristic of the underlying CDCL search engine in a declarative way [61], this needs to be extended to constraint variables to mimic and extend traditional CP heuristics.

Bibliography

[1] M. Abseher, M. Gebser, N. Musliu, T. Schaub, and S. Woltran. Shift design with answer set programming. *Fundamenta Informaticae*, 147(1):1–25, 2016.

[2] B. Andres, B. Kaufmann, O. Matheis, and T. Schaub. Unsatisfiability-based optimization in clasp. In Dovier and Santos Costa [40], pages 212–221.

[3] R. Aziz. Bound founded answer set programming. *CoRR*, abs/1405.3367, 2014.

[4] R. Aziz, G. Chu, and P. Stuckey. Stable model semantics for founded bounds. *Theory and Practice of Logic Programming*, 13(4-5):517–532, 2013.

[5] M. Balduccini. Representing constraint satisfaction problems in answer set programming. In W. Faber and J. Lee, editors, *Proceedings of the Second Workshop on Answer Set Programming and Other Computing Paradigms (ASPOCP'09)*, pages 16–30, 2009.

[6] M. Balduccini. Industrial-size scheduling with ASP+CP. In Delgrande and Faber [39], pages 284–296.

[7] M. Balduccini, M. Gelfond, and M. Nogueira. Answer set based design of knowledge systems. *Annals of Mathematics and Artificial Intelligence*, 47(1-2):183–219, 2006.

[8] M. Balduccini, M. Gelfond, M. Nogueira, and R. Watson. The USA-Advisor: A case study in answer set planning, 2001.

[9] M. Balduccini and Y. Lierler. Integration schemas for constraint answer set programming: a case study. In E. Lamma and T. Swift, editors, *Technical Communications of the Twenty-ninth International Conference on Logic Programming (ICLP'13)*, volume 13(4-5) of *Theory and Practice of Logic Programming, Online Supplement*, 2013.

[10] M. Balduccini and Y. Lierler. Constraint answer set solver EZCSP and why integration schemas matter. Unpublished draft, 2016. Available at: `http://works.bepress.com/yuliya_lierler/64/`.

[11] M. Banbara, M. Gebser, K. Inoue, M. Ostrowski, A. Peano, T. Schaub, T. Soh, N. Tamura, and M. Weise. aspartame: Solving constraint satisfaction problems with answer set programming. In F. Calimeri, G. Ianni, and M. Truszczyński,

editors, *Proceedings of the Thirteenth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'15)*, volume 9345 of *Lecture Notes in Artificial Intelligence*, pages 112–126. Springer-Verlag, 2015.

[12] M. Banbara, K. Inoue, B. Kaufmann, T. Schaub, T. Soh, N. Tamura, and P. Wanko. teaspoon: Solving the curriculum-based course timetabling problems with answer set programming. In E. Burke, L. Di Gaspero, B. McCollum, A. Schaerf, and E. Özcan, editors, *Proceedings of the Eleventh International Conference of the Practice and Theory of Automated Timetabling (PATAT'16)*, pages 13–32, 2016.

[13] M. Banbara, B. Kaufmann, M. Ostrowski, and T. Schaub. Clingcon: The next generation. *Theory and Practice of Logic Programming*, 17(4):408–461, 2017.

[14] M. Banbara, N. Tamura, and T. Tanjo. Proposal of a compact and efficient SAT encoding using a numeral system of any base. In Y. Ben-Haim and Y. Naveh, editors, *Proceedings of the Second International Workshop on the Cross-Fertilization Between CSP and SAT (CSPSAT'11)*, 2011.

[15] C. Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press, 2003.

[16] C. Baral, K. Chancellor, N. Tran, N. Tran, A. Joy, and M. Berens. A knowledge based approach for representing and reasoning about signaling networks. In *Proceedings of the Twelfth International Conference on Intelligent Systems for Molecular Biology/Third European Conference on Computational Biology (ISMB'04/ECCB'04)*, pages 15–22. Oxford University Press, 2004.

[17] C. Barrett, P. Fontaine, and C. Tinelli. The SMT-LIB Standard: Version 2.5, 2015. Available at `http://www.SMT-LIB.org`.

[18] M. Bartholomew and J. Lee. Functional stable model semantics and answer set programming modulo theories. In F. Rossi, editor, *Proceedings of the Twenty-third International Joint Conference on Artificial Intelligence (IJCAI'13)*, pages 718–724. IJCAI/AAAI Press, 2013.

[19] S. Baselice, P. Bonatti, and M. Gelfond. Towards an integration of answer set and constraint solving. In M. Gabbrielli and G. Gupta, editors, *Proceedings of the Twenty-first International Conference on Logic Programming (ICLP'05)*, volume 3668 of *Lecture Notes in Computer Science*, pages 52–66. Springer-Verlag, 2005.

[20] R. Bellman. On a routing problem. *Quarterly of Applied Mathematics*, 16:87–90, 1958.

[21] C. Bessiere, G. Katsirelos, N. Narodytska, C. Quimper, and T. Walsh. Decompositions of all different, global cardinality and related constraints. In Boutilier [24], pages 419–424.

[22] C. Bessiere, G. Katsirelos, N. Narodytska, and T. Walsh. Circuit complexity and decompositions of global constraints. In Boutilier [24], pages 412–418.

[23] A. Biere, M. Heule, H. van Maaren, and T. Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2009.

[24] C. Boutilier, editor. *Proceedings of the Twenty-first International Joint Conference on Artificial Intelligence (IJCAI'09)*. AAAI/MIT Press, 2009.

[25] A. Brodsky, editor. *Proceedings of the Twenty-fifth IEEE International Conference on Tools with Artificial Intelligence (ICTAI'13)*. IEEE Computer Society Press, 2013.

[26] P. Cabalar, R. Kaminski, M. Ostrowski, and T. Schaub. An ASP semantics for default reasoning with constraints. In R. Kambhampati, editor, *Proceedings of the Twenty-fifth International Joint Conference on Artificial Intelligence (IJCAI'16)*, pages 1015–1021. IJCAI/AAAI Press, 2016.

[27] P. Cabalar, R. Otero, and S. Pose. Temporal constraint networks in action. In W. Horn, editor, *Proceedings of the Fourteenth European Conference on Artificial Intelligence (ECAI'00)*, pages 543–547. IOS Press, 2000.

[28] F. Calimeri, G. Ianni, F. Ricca, M. Alviano, A. Bria, G. Catalano, S. Cozza, W. Faber, O. Febbraro, N. Leone, M. Manna, A. Martello, C. Panetta, S. Perri, K. Reale, M. Santoro, M. Sirianni, G. Terracina, and P. Veltri. The third answer set programming competition: Preliminary report of the system competition track. In Delgrande and Faber [39], pages 388–403.

[29] M. Carro and A. King, editors. *Technical Communications of the Thirty-second International Conference on Logic Programming (ICLP'16)*, volume 52. Open Access Series in Informatics (OASIcs), 2016.

[30] J. Chinneck and E. Dravinieks. Locating minimal infeasible constraints sets in linear programs. In *ORSA Journal On Computing*, volume 3, pages 157–168. Operations Research Society of America, 1991.

[31] J. Crawford and A. Baker. Experimental results on the application of satisfiability algorithms to scheduling problems. In B. Hayes-Roth and R. Korf, editors, *Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI'94)*, pages 1092–1097. AAAI Press, 1994.

[32] A. Darwiche and K. Pipatsrisawat. Complete algorithms. In Biere et al. [23], chapter 3, pages 99–130.

[33] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5:394–397, 1962.

[34] B. De Cat, B. Bogaerts, M. Bruynooghe, and M. Denecker. Predicate logic as a modelling language: The IDP system. *CoRR*, abs/1401.6312, 2014.

[35] B. De Cat, B. Bogaerts, J. Devriendt, and M. Denecker. Model expansion in the presence of function symbols using constraint programming. In Brodsky [25], pages 1068–1075.

[36] L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In C. Ramakrishnan and J. Rehof, editors, *Proceedings of the Fourteenth International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08)*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer-Verlag, 2008.

[37] A. De Rosis, T. Eiter, C. Redl, and F. Ricca. Constraint answer set programming based on HEX-programs. In D. Inclezan and M. Maratea, editors, *Proceedings of the Eighth Workshop on Answer Set Programming and Other Computing Paradigms (ASPOCP'15)*, 2015.

[38] R. Dechter. *Constraint Processing.* Morgan Kaufmann Publishers, 2003.

[39] J. Delgrande and W. Faber, editors. *Proceedings of the Eleventh International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'11)*, volume 6645 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, 2011.

[40] A. Dovier and V. Santos Costa, editors. *Technical Communications of the Twenty-eighth International Conference on Logic Programming (ICLP'12)*, volume 17. Leibniz International Proceedings in Informatics (LIPIcs), 2012.

[41] W. Dowling and J. Gallier. Linear-time algorithms for testing the satisfiability of propositional Horn formulae. *Journal of Logic Programming*, 1:267–284, 1984.

[42] C. Drescher. *Conflict-driven constraint answer set solving.* PhD thesis, Computer Science and Engineering, Faculty of Engineering, UNSW, 2015.

[43] C. Drescher and T. Walsh. A translational approach to constraint answer set solving. *Theory and Practice of Logic Programming*, 10(4-6):465–480, 2010.

[44] C. Drescher and T. Walsh. Answer set solving with lazy nogood generation. In Dovier and Santos Costa [40], pages 188–200.

[45] M. Durzinsky, W. Marwan, M. Ostrowski, T. Schaub, and A. Wagler. Automatic network reconstruction using ASP. *Theory and Practice of Logic Programming*, 11(4-5):749–766, 2011.

[46] R. Elmasri and S. Navathe. *Fundamentals of database systems.* Addison-Wesley, 1994.

[47] W. Faber, G. Pfeifer, and N. Leone. Semantics and complexity of recursive aggregates in answer set programming. *Artificial Intelligence*, 175(1):278–298, 2011.

[48] F. Fages. Consistency of Clark's completion and the existence of stable models. *Journal of Methods of Logic in Computer Science*, 1:51–60, 1994.

[49] T. Feydy, Z. Somogyi, and P. Stuckey. Half reification and flattening. In J. Lee, editor, *Proceedings of the Seventeenth International Conference on Principles and Practice of Constraint Programming (CP'11)*, volume 6876 of *Lecture Notes in Computer Science*, pages 286–301. Springer-Verlag, 2011.

[50] L. Ford and D. Fulkerson. *Flows in networks*. Princeton University Press, 1962.

[51] J. Gallagher and M. Gelfond, editors. *Technical Communications of the Twenty-seventh International Conference on Logic Programming (ICLP'11)*, volume 11. Leibniz International Proceedings in Informatics (LIPIcs), 2011.

[52] M. Gebser, R. Kaminski, B. Kaufmann, M. Lindauer, M. Ostrowski, J. Romero, T. Schaub, and S. Thiele. *Potassco User Guide*. University of Potsdam, second edition edition, 2015.

[53] M. Gebser, R. Kaminski, B. Kaufmann, M. Ostrowski, T. Schaub, and P. Wanko. Theory solving made easy with clingo 5. In Carro and King [29], pages 2:1–2:15.

[54] M. Gebser, R. Kaminski, B. Kaufmann, M. Ostrowski, T. Schaub, and P. Wanko. Theory solving made easy with clingo 5 (extended version). Available at `http://www.cs.uni-potsdam.de/wv/publications/`, 2016. Extended version of [53].

[55] M. Gebser, R. Kaminski, B. Kaufmann, and T. Schaub. Multi-criteria optimization in answer set programming. In Gallagher and Gelfond [51], pages 1–10.

[56] M. Gebser, R. Kaminski, B. Kaufmann, and T. Schaub. *Answer Set Solving in Practice*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan and Claypool Publishers, 2012.

[57] M. Gebser, R. Kaminski, B. Kaufmann, and T. Schaub. *Clingo* = ASP + control: Preliminary report. In M. Leuschel and T. Schrijvers, editors, *Technical Communications of the Thirtieth International Conference on Logic Programming (ICLP'14)*, volume 14(4-5) of *Theory and Practice of Logic Programming, Online Supplement*, 2014. Available at `http://arxiv.org/abs/1405.3694v1`.

[58] M. Gebser, R. Kaminski, B. Kaufmann, and T. Schaub. Multi-shot ASP solving with clingo. *CoRR*, abs/1705.09811, 2017.

[59] M. Gebser, R. Kaminski, P. Obermeier, and T. Schaub. Ricochet robots reloaded: A case-study in multi-shot ASP solving. In T. Eiter, H. Strass, M. Truszczyński, and S. Woltran, editors, *Advances in Knowledge Representation, Logic Programming, and Abstract Argumentation: Essays Dedicated to Gerhard Brewka on the Occasion of His 60th Birthday*, volume 9060 of *Lecture Notes in Artificial Intelligence*, pages 17–32. Springer-Verlag, 2015.

[60] M. Gebser, R. Kaminski, and T. Schaub. Complex optimization in answer set programming. *Theory and Practice of Logic Programming*, 11(4-5):821–839, 2011.

[61] M. Gebser, B. Kaufmann, R. Otero, J. Romero, T. Schaub, and P. Wanko. Domain-specific heuristics in answer set programming. In M. desJardins and M. Littman, editors, *Proceedings of the Twenty-Seventh National Conference on Artificial Intelligence (AAAI'13)*, pages 350–356. AAAI Press, 2013.

[62] M. Gebser, B. Kaufmann, and T. Schaub. Conflict-driven answer set solving: From theory to practice. *Artificial Intelligence*, 187-188:52–89, 2012.

[63] M. Gebser, B. Kaufmann, and T. Schaub. Multi-threaded ASP solving with clasp. *Theory and Practice of Logic Programming*, 12(4-5):525–545, 2012.

[64] M. Gebser, M. Ostrowski, and T. Schaub. Constraint answer set solving. In P. Hill and D. Warren, editors, *Proceedings of the Twenty-fifth International Conference on Logic Programming (ICLP'09)*, volume 5649 of *Lecture Notes in Computer Science*, pages 235–249. Springer-Verlag, 2009.

[65] M. Gebser, T. Schaub, S. Thiele, B. Usadel, and P. Veber. Detecting inconsistencies in large biological networks with answer set programming. In M. Garcia de la Banda and E. Pontelli, editors, *Proceedings of the Twenty-fourth International Conference on Logic Programming (ICLP'08)*, volume 5366 of *Lecture Notes in Computer Science*, pages 130–144. Springer-Verlag, 2008.

[66] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In R. Kowalski and K. Bowen, editors, *Proceedings of the Fifth International Conference and Symposium of Logic Programming (ICLP'88)*, pages 1070–1080. MIT Press, 1988.

[67] I. Gent. Arc consistency in SAT. In F. van Harmelen, editor, *Proceedings of the Fifteenth Eureopean Conference on Artificial Intelligence (ECAI'02)*, pages 121–125. IOS Press, 2002.

[68] J. Gleeson and J. Ryan. Identifying minimally infeasible subsystems of inequalities. In *ORSA Journal On Computing*, volume 2, pages 61–63. Operations Research Society of America, 1990.

[69] C. Gomes and B. Selman. Problem structure in the presence of perturbations. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI'97)*, pages 221–226. AAAI/MIT Press, 1997.

[70] G. Grasso, S. Iiritano, N. Leone, V. Lio, F. Ricca, and F. Scalise. An ASP-based system for team-building in the Gioia-Tauro seaport. In M. Carro and R. Peña, editors, *Proceedings of the Twelfth International Symposium on Practical Aspects of Declarative Languages (PADL'10)*, volume 5937 of *Lecture Notes in Computer Science*, pages 40–42. Springer-Verlag, 2010.

[71] C. Guziolowski, L. Paulevé, M. Ostrowski, T. Schaub, and A. Siegel. Boolean network identification from multiplex time series data. In O. Roux and J. Bourdon, editors, *Proceedings of the Thirteenth International Conference on Computational Methods in Systems Biology (CMSB'15)*, volume 9308 of *Lecture Notes in Bioinformatics*, pages 170–181. Springer-Verlag, 2015.

[72] A. Heyting. Die formalen Regeln der intuitionistischen Logik. In *Sitzungsberichte der Preussischen Akademie der Wissenschaften*, page 42–56. Deutsche Akademie der Wissenschaften zu Berlin, 1930. Reprint in Logik-Texte: Kommentierte Auswahl zur Geschichte der Modernen Logik, Akademie-Verlag, 1986.

[73] J. Huang. Universal Booleanization of constraint models. In P. Stuckey, editor, *Proceedings of the Fourteenth International Conference on Principles and Practice of Constraint Programming (CP'08)*, volume 5202 of *Lecture Notes in Computer Science*, pages 144–158. Springer-Verlag, 2008.

[74] T. Janhunen. Some (in)translatability results for normal logic programs and propositional theories. *Journal of Applied Non-Classical Logics*, 16(1-2):35–86, 2006.

[75] T. Janhunen, R. Kaminski, M. Ostrowski, T. Schaub, S. Schellhorn, and P. Wanko. Clingo goes linear constraints over reals and integers. *Theory and Practice of Logic Programming*, 17(5-6):872–888, 2017.

[76] T. Janhunen, I. Niemelä, and M. Sevalnev. Computing stable models via reductions to difference logic. In E. Erdem, F. Lin, and T. Schaub, editors, *Proceedings of the Tenth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'09)*, volume 5753 of *Lecture Notes in Artificial Intelligence*, pages 142–154. Springer-Verlag, 2009.

[77] U. Junker. Quickxplain: Conflict detection for arbitrary constraint propagation algorithms. *IJCAI'01 Workshop on Modelling and Solving problems with constraints*, 2001.

[78] B. Kaufmann, N. Leone, S. Perri, and T. Schaub. Grounding and solving in answer set programming. *AI Magazine*, 37(3):25–32, 2016.

[79] C. Lecoutre, O. Roussel, and M. van Dongen. Promoting robust black-box solvers through competitions. *Constraints*, 15(3):317–326, 2010.

[80] Y. Lierler. Relating constraint answer set programming languages and algorithms. *Artificial Intelligence*, 207:1–22, 2014.

[81] Y. Lierler, S. Smith, M. Truszczynski, and A. Westlund. Weighted-sequence problem: ASP vs CASP and declarative vs problem-oriented solving. In C. V. Russo and N. Zhou, editors, *Proceedings of the Eleventh International Symposium on Practical Aspects of Declarative Languages (PADL'12)*, volume 7149 of *Lecture Notes in Computer Science*, pages 63–77. Springer-Verlag, 2012.

[82] Y. Lierler and B. Susman. SMT-based constraint answer set solver EZSMT (system description). In Carro and King [29], pages 1:1–1:15.

[83] F. Lin and Y. Zhao. ASSAT: computing answer sets of a logic program by SAT solvers. *Artificial Intelligence*, 157(1-2):115–137, 2004.

[84] G. Liu, T. Janhunen, and I. Niemelä. Answer set programming via mixed integer programming. In G. Brewka, T. Eiter, and S. McIlraith, editors, *Proceedings of the Thirteenth International Conference on Principles of Knowledge Representation and Reasoning (KR'12)*, pages 32–42. AAAI Press, 2012.

[85] G. Liu and J. You. Level mapping induced loop formulas for weight constraint and aggregate logic programs. *Fundamenta Informaticae*, 101(3):237–255, 2010.

[86] J. Lloyd. *Foundations of Logic Programming*. Symbolic Computation. Springer-Verlag, 1987.

[87] V. Marek and M. Truszczyński. Stable models and an alternative logic programming paradigm. In K. Apt, V. Marek, M. Truszczyński, and D. Warren, editors, *The Logic Programming Paradigm: a 25-Year Perspective*, pages 375–398. Springer-Verlag, 1999.

[88] J. Marques-Silva, I. Lynce, and S. Malik. Conflict-driven clause learning SAT solvers. In Biere et al. [23], chapter 4, pages 131–153.

[89] J. Marques-Silva and K. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, 1999.

[90] V. Mellarkod and M. Gelfond. Integrating answer set reasoning with constraint solving techniques. In J. Garrigue and M. Hermenegildo, editors, *Proceedings of the Ninth International Symposium on Functional and Logic Programming (FLOPS'08)*, volume 4989 of *Lecture Notes in Computer Science*, pages 15–31. Springer-Verlag, 2008.

[91] V. Mellarkod, M. Gelfond, and Y. Zhang. Integrating answer set programming and constraint logic programming. *Annals of Mathematics and Artificial Intelligence*, 53(1-4):251–287, 2008.

[92] A. Metodi and M. Codish. Compiling finite domain constraints to SAT with BEE. *Theory and Practice of Logic Programming*, 12(4-5):465–483, 2012.

[93] A. Metodi, M. Codish, and P. Stuckey. Boolean equi-propagation for concise and efficient SAT encodings of combinatorial problems. *Journal of Artificial Intelligence Research*, 46:303–341, 2013.

[94] D. Mitchell. A SAT solver primer. *Bulletin of the European Association for Theoretical Computer Science*, 85:112–133, 2005.

[95] I. Niemelä. Logic programs with stable model semantics as a constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence*, 25(3-4):241–273, 1999.

[96] I. Niemelä. Stable models and difference logic. *Annals of Mathematics and Artificial Intelligence*, 53(1-4):313–329, 2008.

[97] R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Solving SAT and SAT modulo theories: From an abstract Davis-Putnam-Logemann-Loveland procedure to DPLL(T). *Journal of the ACM*, 53(6):937–977, 2006.

[98] M. Nogueira, M. Balduccini, M. Gelfond, R. Watson, and M. Barry. An A-prolog decision support system for the space shuttle. In I. Ramakrishnan, editor, *Proceedings of the Third International Symposium on Practical Aspects of Declarative Languages (PADL'01)*, volume 1990 of *Lecture Notes in Computer Science*, pages 169–183. Springer-Verlag, 2001.

[99] O. Ohrimenko, P. Stuckey, and M. Codish. Propagation via lazy clause generation. *Constraints*, 14(3):357–391, 2009.

[100] M. Ostrowski, G. Flouris, T. Schaub, and G. Antoniou. Evolution of ontologies using ASP. In Gallagher and Gelfond [51], pages 16–27.

[101] M. Ostrowski, L. Paulevé, T. Schaub, A. Siegel, and C. Guziolowski. Boolean network identification from perturbation time series data combining dynamics abstraction and logic programming. *Biosystems*, 149:139–153, 2016.

[102] M. Ostrowski and T. Schaub. ASP modulo CSP: The clingcon system. *Theory and Practice of Logic Programming*, 12(4-5):485–503, 2012.

[103] J. Petke and P. Jeavons. The order encoding: From tractable CSP to tractable SAT. In K. Sakallah and L. Simon, editors, *Proceedings of the Fourteenth International Conference on Theory and Applications of Satisfiability Testing (SAT'11)*, volume 6695 of *Lecture Notes in Computer Science*, pages 371–372. Springer-Verlag, 2011.

[104] S. Prestwich. CNF encodings. In Biere et al. [23], pages 75–97.

[105] C. Redl. The dlvhex system for knowledge representation: recent advances (system description). *Theory and Practice of Logic Programming*, 16(5-6):866–883, 2016.

[106] F. Rossi, P. van Beek, and T. Walsh, editors. *Handbook of Constraint Programming*. Elsevier Science, 2006.

[107] O. Roussel and C. Lecoutre. XML representation of constraint networks: Format XCSP 2.1. *CoRR*, abs/0902.2362, 2009.

[108] C. Schulte and G. Tack. Views and iterators for generic constraint implementations. In P. van Beek, editor, *Proceedings of the Eleventh International Conference on Principles and Practice of Constraint Programming (CP'05)*, volume 3709 of *Lecture Notes in Computer Science*, pages 118–132. Springer-Verlag, 2005.

[109] P. Simons, I. Niemelä, and T. Soininen. Extending and implementing the stable model semantics. *Artificial Intelligence*, 138(1-2):181–234, 2002.

[110] T. Soh, K. Inoue, N. Tamura, M. Banbara, and H. Nabeshima. A SAT-based method for solving the two-dimensional strip packing problem. *Fundamenta Informaticae*, 102(3-4):467–487, 2010.

[111] T. Soininen and I. Niemelä. Developing a declarative rule language for applications in product configuration. In G. Gupta, editor, *Proceedings of the First International Workshop on Practical Aspects of Declarative Languages (PADL'99)*, volume 1551 of *Lecture Notes in Computer Science*, pages 305–319. Springer-Verlag, 1999.

[112] O. Stashuk. Integrating constraint programming into answer set programming. Master Thesis, TU Wien, 2013.

[113] T. Syrjänen. Lparse 1.0 user's manual, 2001.

[114] N. Tamura, M. Banbara, and T. Soh. Compiling pseudo-Boolean constraints to SAT with order encoding. In Brodsky [25], pages 1020–1027.

[115] N. Tamura, A. Taga, S. Kitagawa, and M. Banbara. Compiling finite linear CSP into SAT. In F. Benhamou, editor, *Proceedings of the Twelfth International Conference on Principles and Practice of Constraint Programming (CP'06)*, volume 4204 of *Lecture Notes in Computer Science*, pages 590–603. Springer-Verlag, 2006.

[116] N. Tamura, A. Taga, S. Kitagawa, and M. Banbara. Compiling finite linear CSP into SAT. *Constraints*, 14(2):254–272, 2009.

[117] T. Tanjo, N. Tamura, and M. Banbara. Azucar: A SAT-based CSP solver using compact order encoding. In A. Cimatti and R. Sebastiani, editors, *Proceedings of the Fifteenth International Conference on Theory and Applications of Satisfiability Testing (SAT'12)*, volume 7317 of *Lecture Notes in Computer Science*, pages 456–462, Berlin, Heidelberg, 2012. Springer-Verlag.

[118] F. Thibaut and P. Stuckey. Lazy clause generation reengineered. In I. Gent, editor, *Proceedings of the Fifteenth International Conference on Principles and Practice of Constraint Programming (CP'09)*, volume 5732 of *Lecture Notes in Computer Science*, pages 352–366. Springer-Verlag, 2009.

[119] C. Thiffault, F. Bacchus, and T. Walsh. Solving non-clausal formulas with DPLL search. In M. Wallace, editor, *Proceedings of the Tenth International Conference on Principles and Practice of Constraint Programming (CP'04)*, volume 3258 of *Lecture Notes in Computer Science*, pages 663–678. Springer-Verlag, 2004.

[120] J. van Loon. Irreducibly inconsistent systems of linear inequalities. *European Journal of Operational Research*, 8(3):283–288, 1981.

[121] T. Walsh. SAT versus CSP. In R. Dechter, editor, *Proceedings of the Sixth International Conference on Principles and Practice of Constraint Programming (CP'00)*, volume 1894 of *Lecture Notes in Computer Science*, pages 441–456. Springer-Verlag, 2000.

[122] Y. Yu and S. Malik. Lemma learning in SMT on linear constraints. In A. Biere and C. Gomes, editors, *Proceedings of the Ninth International Conference on Theory and Applications of Satisfiability Testing (SAT'06)*, volume 4121 of *Lecture Notes in Computer Science*, pages 142–155. Springer-Verlag, 2006.

[123] L. Zhang, C. Madigan, M. Moskewicz, and S. Malik. Efficient conflict driven learning in a Boolean satisfiability solver. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD'01)*, pages 279–285. ACM Press, 2001.

[124] N. Zhou. The SAT compiler in B-prolog. The Association for Logic Programming Newsletter, March 2013, 2013. `http://www.cs.nmsu.edu/ALP/2013/03/the-sat-compiler-in-b-prolog/`.