UPPSALA
UNIVERSITET

Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology 1591

Analysis, synthesis and application of automaton-based constraint descriptions

MARÍA ANDREÍNA FRANCISCO RODRÍGUEZ

ACTA
UNIVERSITATIS
UPSALIENSIS
UPPSALA
2017

ISSN 1651-6214
ISBN 978-91-513-0132-7
urn:nbn:se:uu:diva-332149

Dissertation presented at Uppsala University to be publicly examined in ITC 2446, Polacksbacken, Lägerhyddsvägen 2, Uppsala, Friday, 15 December 2017 at 13:15 for the degree of Doctor of Philosophy. The examination will be conducted in English. Faculty examiner: Reader Christopher Jefferson (University of St Andrews).

Abstract

Francisco Rodríguez, M. A. 2017. Analysis, synthesis and application of automaton-based constraint descriptions. *Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology* 1591. 79 pp. Uppsala: Acta Universitatis Upsaliensis. ISBN 978-91-513-0132-7.

Constraint programming (CP) is a technology in which a combinatorial problem is modelled as a conjunction of constraints on variables ranging over given initial domains, and optionally an objective function on the variables. Such a model is given to a general-purpose solver performing systematic search to find constraint-satisfying domain values for the variables, giving an optimal value to the objective function. A constraint predicate (also known as a global constraint) does two things: from the modelling perspective, it allows a modeller to express a commonly occurring combinatorial substructure, for example that a set of variables must take distinct values; from the solving perspective, it comes with a propagation algorithm, called a propagator, which removes some but not necessarily all impossible values from the current domains of its variables when invoked during search.

Although modern CP solvers have many constraint predicates, often a predicate one would like to use is not available. In the past, the choices were either to reformulate the model or to write one's own propagator. In this dissertation, we contribute to the automatic design of propagators for new predicates.

Integer time series are often subject to constraints on the aggregation of the features of all maximal occurrences of some pattern. For example, the minimum width of the peaks may be constrained. Automata allow many constraint predicates for variable sequences, and in particular many time-series predicates, to be described in a high-level way. Our first contribution is an algorithm for generating an automaton-based predicate description from a pattern, a feature, and an aggregator.

It has previously been shown how to decompose an automaton-described constraint on a variable sequence into a conjunction of constraints whose predicates have existing propagators. This conjunction provides the propagation, but it is unknown how to propagate it efficiently. Our second contribution is a tool for deriving, in an off-line process, implied constraints for automaton-induced constraint decompositions to improve propagation. Further, when a constraint predicate functionally determines a result variable that is unchanged under reversal of a variable sequence, we provide as our third contribution an algorithm for deriving an implied constraint between the result variables for a variable sequence, a prefix thereof, and the corresponding suffix.

Keywords: constraint programming, constraint predicates, global constraints, automata, automaton-described constraint predicates, automaton-induced constraint decompositions, implied constraints, time-series constraints, transducers, automaton invariants

María Andreína Francisco Rodríguez, Department of Information Technology, Division of Computing Science, Box 337, Uppsala University, SE-75105 Uppsala, Sweden.

ISSN 1651-6214
ISBN 978-91-513-0132-7
urn:nbn:se:uu:diva-332149 (http://urn.kb.se/resolve?urn=urn:nbn:se:uu:diva-332149)

Dedicated to my daughter Abril

List of Papers

This dissertation is based on the following papers, which are referred to in the text by their Roman numerals:

I M. A. Francisco Rodríguez, P. Flener, and J. Pearson:
Automatic generation of descriptions of time-series constraints.
In: M. Virvou (editor), ICTAI 2017. IEEE Computer Society, 2017 (in press).

II M. A. Francisco Rodríguez, P. Flener, and J. Pearson:
Generation of implied constraints for automaton-induced decompositions.
In: A. Brodsky (editor), ICTAI 2013, pages 1076–1083. IEEE Computer Society, 2013.

III M. A. Francisco Rodríguez, P. Flener, and J. Pearson:
Implied constraints for AUTOMATON constraints.
In: G. Gottlob, G. Sutcliffe, and A. Voronkov (editors), GCAI 2015. EasyChair Proceedings in Computing, volume 36, pages 113–126, 2015.

IV E. Arafailova, N. Beldiceanu, R. Douence, P. Flener, M. A. Francisco Rodríguez, J. Pearson, and H. Simonis:
Time-series constraints: Improvements and application in CP and MIP contexts.
In: C.-G. Quimper (editor), CP-AI-OR 2016. Lecture Notes in Computer Science, volume 9676, pages 18–34. Springer, 2016.

V N. Beldiceanu, M. Carlsson, P. Flener, M. A. Francisco Rodríguez, and J. Pearson:
Linking prefixes and suffixes for constraints encoded using automata with accumulators.
In: B. O'Sullivan (editor), CP 2014. Lecture Notes in Computer Science, volume 8656, pages 142–157. Springer, 2014.

VI E. Arafailova, N. Beldiceanu, M. Carlsson, P. Flener, M. A. Francisco Rodríguez, J. Pearson, and H. Simonis:
Systematic derivation of bounds and glue constraints for time-series constraints.
In: M. Rueher (editor), CP 2016. Lecture Notes in Computer Science, volume 9892, pages 13–29. © Springer, 2016.

Reprints were made with permission from the publishers.

Comments on my Participation

Paper I

I was the lead researcher and lead writer. My advisors contributed to the discussions.

Paper II

I was the lead researcher and lead writer. My advisors contributed to the discussions and writing.

Paper III

I was the lead researcher and contributed to the writing. My advisors contributed to the discussions and writing.

Paper IV

I was the lead researcher and lead writer of the section *Improved Generation of Implied Constraints* (Section 5), as well as the lead writer of the section *Benchmark on CP and MIP Solvers* (Section 6). I contributed to the discussions and writing of the rest of the paper.

Paper V

I contributed to the discussions and writing.

Paper VI

I was the lead researcher and lead writer of the section *Glue Constraints for Time-Series Constraints* (Section 3). I contributed to the discussions and writing of the rest of the paper.

Other Publications

E. Arafailova, N. Beldiceanu, R. Douence, M. Carlsson, P. Flener, M. A. Francisco Rodríguez, J. Pearson, and H. Simonis:
Global Constraint Catalog, Volume II, Time-Series Constraints.
In: Computing Research Repository, arXiv:1609.08925, September 2016.
Available at `http://arxiv.org/abs/1609.08925`.

M. A. Francisco Rodríguez, P. Flener, and J. Pearson:
Consistency of constraint networks induced by automaton-based constraint specifications.
In: A. Rendl and Ch. Beck (editors), Proceedings of ModRef 2011, the 10th International Workshop on Constraint Modelling and Reformulation, held at CP 2011, 2011.
Available at `http://www-users.cs.york.ac.uk/~frisch/ModRef/11`.

Contents

1 Introduction 13
1.1 Constraint Programming 14
1.2 Contributions 16
1.3 Outline of the Dissertation 18

2 Describing Constraints by Automata 19
2.1 Finite Automata and Regular Languages 19
2.2 Describing Constraints by Deterministic Finite Automata 21
2.3 Describing Constraints by Predicate Automata 22
2.4 Describing Constraints by Automata with Accumulators 24
2.5 Describing Constraints by Predicate Automata with Accumulators 26

3 Time-Series Constraints 28
3.1 Definitions 28
3.2 Specifying a Pattern by a Transducer 30
3.3 Synthesising Automaton-Based Descriptions of Time-Series Constraint Predicates from a Transducer 33

4 Implied Constraints for Automaton-Induced Constraint Decompositions 38
4.1 Implied Constraints 38
4.2 Linear Implied Constraints 40
4.2.1 Linear Implied Constraints from a Constraint Checker 42
4.2.2 Linear Implied Constraints from an Automaton 44
4.3 Glue Constraints 50

5 Summaries of Papers 56
I Automatic Generation of Descriptions of Time-Series Constraints 56
II Generation of Implied Constraints for Automaton-Induced Decompositions 57
III Implied Constraints for AUTOMATON Constraints 58
IV Time-Series Constraints: Improvements and Application in CP and MIP Contexts 59
V Linking Prefixes and Suffixes for Constraints Encoded Using Automata with Accumulators 60
VI Systematic Derivation of Bounds and Glue Constraints for Time-Series Constraints 61

6 Related Work 62
6.1 Constraints Over Formal Languages 62
6.2 Other Types of Automata 63
6.3 Quantitative Properties of Data Streams 64
6.4 Improving Propagation of Automaton-Induced Constraint Decompositions 64
6.5 Time-Series Constraints 65

7 Conclusion 67
7.1 Contributions 67
7.2 Future Work 68

8 Glossary 71

Sammanfattning på svenska 73

References 76

Acknowledgements

"Thank you very much," said Alice.

Through the Looking-Glass
LEWIS CARROLL

There are many people that I would like to thank who have helped me grow and develop over the last few years. First and foremost, I would like to thank my advisors Pierre Flener and Justin Pearson. It is impossible for me to put into words how much your support has meant to me. I cannot imagine anyone else being better suited to be my advisors than you. You have been there for me every step of the way while I went down the rabbit hole exploring the mysterious world of automata, constraints, and academic research. You have always been there to give me your advice and your most honest opinion. Thank you for everything.

Thank you to my co-authors Ekaterina Arafailova, Nicolas Beldiceanu, Mats Carlsson, and Helmut Simonis (in alphabetical order). Working with you has brought me so much joy, and led me down a fascinating path.

I would like to thank the womENcourage 2015 team, the members of the UU ACM-W Student Chapter, and the members of the Gender Equality Group at the IT department. It was a pleasure working with you.

For making my life as a PhD student a lot of fun, and for supporting and encouraging me when I least expected it, I would like to thank (in alphabetical order) Gustav Björdal, Åsa Cajander, Sofia Cassel, Virginia Grande, Farshid Hassani, Jean-Noël Monette, Aletta Nylén, Johannes Åman Pohjola, Palle Raabjerg, Joseph Scott, Michael Thuné, Tobias Wrigstad, and Yunyun Zhu. Thank you all for putting up with the madness!

Thank you to all my dear friends for filling my days with a mixture of cake, candy, coffee, penguins, sheep, unicorns, flowers, princesses, and baby-eating monsters. You know who you are!

I would like to especially thank Sofia Cassel, Virginia Grande, and Joseph Scott: you are the best! I couldn't have done it without you! Seriously, I would have been really hungry! A profound thank you to Sofia Cassel for translating the sammanfattning.

My family have been an endless source of support. Thanks to my parents, Marisol and Juan Antonio, who are always there for me, my sister Stephanie, my favourite aunt Mary Nancy, and the whole Furlan-Capriles family.

To my dear husband Gabriel: I wouldn't have done this without you! Thank you for always believing in me, for your patience, and for your constant support and encouragement (despite your best intentions!). There's not enough iron ore in this planet to make all the thimbles I'd like to send you.

And finally, for no other reason than smiling at me every morning and hugging me when I get home, I would like to thank our daughter Abril: you're my favourite person in the whole world!

About the cover

The cover is a representation of Alice's Adventures in Wonderland in the style of Jimmy Liao. The artwork was kindly provided by the talented Yunyun Zhu.

1. Introduction

> "Where shall I begin, please your Majesty?" he asked."Begin at the beginning," the King said gravely, "and go on till you come to the end: then stop."
>
> *Alice's Adventures in Wonderland*
> LEWIS CARROLL

Consider a nonogram like the one in Figure 1.1. A *nonogram* is a puzzle in the form of a grid in which cells must be filled in black or white according to the numbers at the left and top of the grid, called *clues*, in order to reveal a hidden picture. For example, the nonogram in Figure 1.1 hides a picture of a teapot. Each clue indicates the lengths of the unbroken stretches of black cells in a given row or column. For example, the clue '4 8 3' means that there are three stretches of black cells of length four, eight, and three respectively, with at least one white cell between successive stretches.

With a nonogram, as with most interesting puzzles and real-world combinatorial problems, there are simply too many possible ways of filling in the cells for finding a solution by trial and error in reasonable time. One way of approaching a nonogram, and solving it, is to frame it as a constraint satisfaction problem.

In this chapter we first introduce the reader in Section 1.1 to the basic concepts of constraint programming, and then we list our contributions in Section 7.1. Finally, we give in Section 1.3 an outline of the rest of the dissertation.

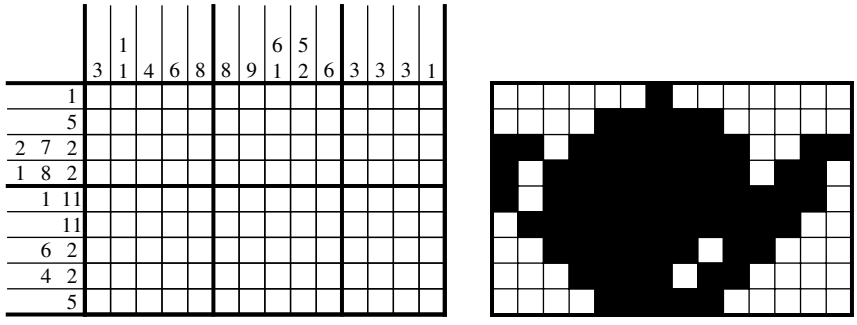


Figure 1.1. A nonogram puzzle (left) and its unique solution (right): a teapot.

1.1 Constraint Programming

Constraint programming (CP) [47] is a declarative programming paradigm for modelling and solving combinatorial problems. Constraint programming is currently successfully applied to many real-world application areas such as scheduling [1, 7], packing [25], and rostering [22].

The idea behind constraint programming is that the user specifies the constraints that should hold among decision variables and a general-purpose constraint solver is used to find a solution.

For example, consider again the nonogram puzzle. Each unknown in the problem, namely each of the cells in the grid, is called a *decision variable*. Each decision variable V_i can take values in a given domain, denoted $\mathrm{dom}(V_i)$. In a nonogram puzzle, the domain of each decision variable is the set $\{\mathrm{w},\mathrm{b}\}$, where the domain value 'w' stands for white and 'b' for black. Moreover, solutions are distinguished from non-solutions by constraints, which are the limitations to the values that the decision variables can take simultaneously.

A *constraint* is a pair $\gamma(\mathcal{V})$, where $\mathcal{V}$ is a tuple of decision variables $\langle V_1,\ldots,V_n\rangle$ and γ is a set of tuples of length n from some given domain. The tuple $\mathcal{V}$ is referred to as the *scope* of the constraint. For example, the constraint $\textsc{AllDifferent}(V_1,\ldots,V_n)$ holds if and only if all the n decision variables in $\langle V_1,\ldots,V_n\rangle$ take n distinct values.

A *solution* to a constraint $\gamma(V_1,\ldots,V_n)$ is some assignment to all its decision variables, $V_1 = d_1,\ \ldots,\ V_n = d_n$, such that the tuple $\langle d_1,\ldots,d_n\rangle$ belongs to γ and each d_i is in $\mathrm{dom}(V_i)$. For example, consider the constraint $\textsc{AllDifferent}(V_1,V_2,V_3)$, where the decision variables V_1, V_2, and V_3 can take values in $\{1,2,3,4\}$. A solution to $\textsc{AllDifferent}(V_1,V_2,V_3)$ is, among others, the assignment $V_1 = 1$, $V_2 = 3$, $V_3 = 4$. Back to our nonogram example, each clue constrains the values that the cells of a give row or column can take. A solution to a given clue is a colour assignment to the cells of the corresponding row or column such that the clue is satisfied.

A *constraint satisfaction problem* (CSP) is a conjunction of constraints, together with the domains of its decision variables. A constraint satisfaction problem is sometimes considered as a set of constraints, with implicit conjunction between the constraints of the set. For example, the conjunction:

$$\textsc{AllDifferent}(V_1,V_2,V_3) \wedge V_1 + V_3 = 4 \tag{1.1}$$

with $\mathrm{dom}(V_i) = \{1,2,3,4\}$, is a constraint satisfaction problem. Note that a nonogram puzzle can be *modelled*, in a fully declarative way, as a constraint satisfaction problem. We will show below an elegant way to do so.

A *solution* to a constraint satisfaction problem is an assignment to all its decision variables that is a solution to all its constraints simultaneously. For example, a solution to the constraint satisfaction problem (1.1) is the assignment $V_1 = 1$, $V_2 = 2$, $V_3 = 3$. Note that the assignment $V_1 = 1$, $V_2 = 3$, $V_3 = 4$

is a solution to the constraint ALLDIFFERENT(V_1, V_2, V_3) but not a solution to the constraint problem (1.1).

Nonogram puzzles are usually designed to have a unique solution, but CSPs in general can have any number of solutions, including none. For example, the unique solution to the nonogram in Figure 1.1 depicts a teapot. Nevertheless, it can be the case that for a given constraint satisfaction problem some solutions are measurably better than other solutions, and the goal is to find a best possible solution: then we instead call it a *constrained optimisation problem* (COP). For example, we could be interested in solutions to (1.1) where the value of V_2 is as large as possible, as is the case with the assignment $V_1 = 1$, $V_2 = 4$, $V_3 = 3$, for instance. The principles discussed here are all easily extensible to COPs, but details are omitted for brevity.

Constraint predicates are an important component in modern CP solvers. A constraint predicate does two things: from the modelling perspective, it allows a modeller to express concisely a commonly occurring combinatorial structure of constraint problems; from the solving perspective, it comes with an algorithm, called a *propagator*, that removes impossible domain values. The removal of impossible values by a given propagator can in turn trigger other propagators, and this process continues until a common fixpoint is reached, that is, a point when none of the propagators can remove any more domain values. The calculation of this fixpoint is interleaved with a backtracking systematic search until a solution is found.

A *global* constraint predicate, such as ALLDIFFERENT, constrains a non-fixed number of decision variables. Although modern CP solvers have many global constraint predicates, often a global constraint predicate that one is looking for is not there. In the past, the choices were either to reformulate the model or to write one's own propagator, as it can be seamlessly added to a CP solver. For example, a *time series* is here a sequence of integers, corresponding to measurements taken over a time interval. Time series are common in many application areas, such as the output of electric power stations over multiple days [17], the manpower required in a call centre [6], or the daily capacity of a hospital clinic over a period of years. Time series are often constrained by physical or organisational limits. For example, the number of inflexions may be constrained, or the sum of the peak maxima, or the minimum of the valley widths, but such global constraint predicates are not readily available in most CP solvers.

One way to reformulate a global constraint is to decompose it. A *decomposition* of a global constraint $\gamma(\mathcal{V})$ is a polynomial-time transformation of $\gamma(\mathcal{V})$ into a conjunction N of constraints for whose predicates there already are propagators, and possibly new decision variables, such that N preserves the set of tuples that belong to $\gamma(\mathcal{V})$. For example, the global constraint ALLDIFFERENT$(V_1, \ldots, V_n)$ can be decomposed into the disequalities $V_i \neq V_j$, where $1 \leq i < j \leq n$. These disequality constraints collectively give the semantics of the global constraint predicate and provide the propagation.

In [13, 43], a framework is given where a global constraint predicate can be described in a relatively simple and high-level way by a deterministic finite automaton. The idea behind an *automaton-based description* of a constraint predicate is to describe what it means for a constraint with that predicate to be satisfied in terms of the accepting paths of the automaton. For example, in a nonogram puzzle, a row constrained to contain two stretches of black cells, of lengths 4 and 3 in this order, separated by at least one white cell but preceded and followed by any amounts of white cells, can be checked by an automaton equivalent to the regular expression $w^*b^4w^+b^3w^*$. Based on the automaton, the framework of [13] decomposes a constraint with the described global constraint predicate into a conjunction of constraints for whose predicates there already are propagators. Such a decomposition is known as an *automaton-induced decomposition* of the constraint. Since this is non-standard background material, we provide a tutorial in Chapter 2.

It is known that, in general, the propagation of the automaton-induced decomposition of a constraint cannot eliminate all impossible values from the domains of the decision variables. In this dissertation we tackle this problem.

1.2 Contributions

In this dissertation we work mainly in two areas: automatically generating automaton-based descriptions of time-series constraint predicates, and automatically improving the propagation of automaton-induced constraint decompositions. We now outline our challenges and contributions for each area. An overview of the terminology and our contributions and how they relate to other work can be seen in Figure 1.2.

Generating Automaton-Based Descriptions of Time-Series Constraint Predicates

In Paper I we show how to synthesise automaton-based descriptions of time-series constraint predicates directly from a regular expression. We do so in two steps: first, we characterise the large class of regular expressions that can be handled by the synthesis of automaton-based descriptions of time-series constraint predicates in [11], making it possible to decide when the synthesiser is applicable; and second, we give an algorithm for, together with the synthesiser in [11], automatically generating automaton-based descriptions of time-series constraint predicates directly from such a regular expression, because the synthesiser of [11] requires the user to provide a handcrafted low-level intermediate representation.

Together with the synthesiser of [11] and the decomposition framework of [13], this work can be seen as providing an automated way to design check-

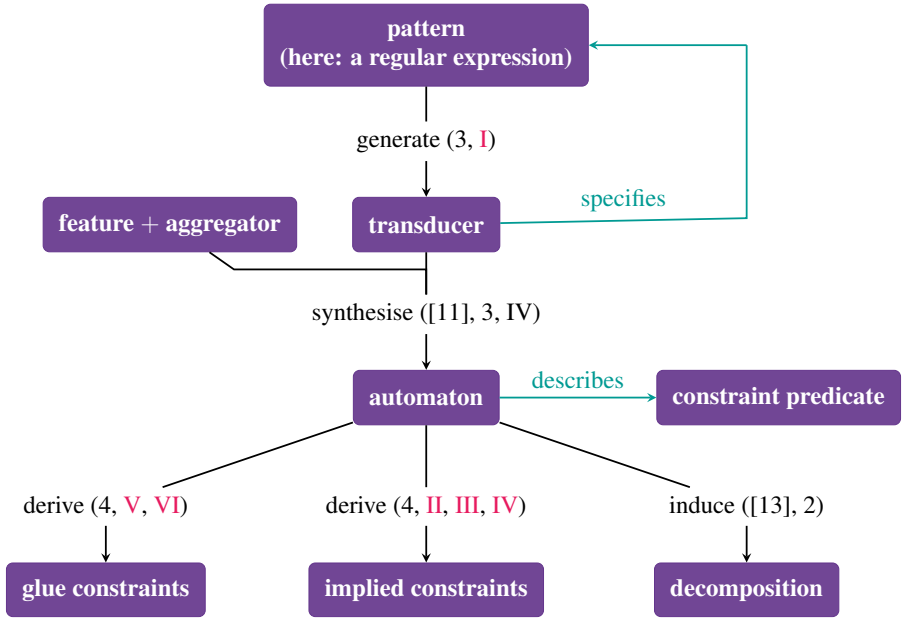


Figure 1.2. Our work and terminology in context. The main contributions of this dissertation are highlighted in red. Roman numbers refer to papers in the appendix and unbracketed Arabic numbers refer to chapters.

ers and decompositions for time-series constraint predicates without expert knowledge on automaton-based constraint descriptions.

Improving Automaton-Induced Constraint Decompositions

In Papers II–VI we show how to derive implied constraints from automaton-induced constraint decompositions. An *implied constraint* is a constraint that is logically implied by other constraints [54]. It does not change the set of solutions, but the idea is that adding it to a model might reduce the time required to solve the problem due to additional propagation.

First, in Papers II–IV we show how to derive implied constraints directly from an automaton-based constraint description, which can be added to the corresponding automaton-induced constraint decomposition.

Second, consider a constraint predicate for a sequence of decision variables functionally determining a result variable that is unchanged under sequence reversal. When such a constraint predicate is described using an automaton, we show in Papers V–VI how to derive, for the automaton-induced constraint decomposition, an implied constraint between the result variables for a sequence of decision variables, a prefix thereof, and the corresponding suffix.

This work can be seen as providing an automated way to improve propagation for automaton-induced constraint decompositions.

1.3 Outline of the Dissertation

Chapter 2 recapitulates the required background on classical automata theory and introduces the reader to the automaton-based description of a constraint predicate [13, 43].

Chapter 3 introduces the reader to time series and time-series constraint predicates. In particular, we define the class of time-series constraint predicates for which we are able to synthesise automaton-based constraint predicate descriptions automatically.

Chapter 4 introduces the reader to implied constraints for automaton-induced constraint decompositions. In Section 4.2 we present our tool ImpGen and show how it can be used to derive automatically linear implied constraints directly from an automaton. In Section 4.3 we define a new kind of implied constraint, called *glue constraints*, and show how to derive such constraints.

Chapter 5 summarises each of the included papers. Chapter 6 provides an overview of related work. In Chapter 7 we conclude and present possible future work.

To make this dissertation self-contained, we define other used concepts in Chapter 8.

An overview of the terminology introduced in each chapter and how the topics relate to each other can be seen in Figure 1.2.

2. Describing Constraints by Automata

> "Besides, that's not a regular rule: you invented it just now."
>
> *Alice's Adventures in Wonderland*
> LEWIS CARROLL

This chapter recapitulates the standard theory of automata (see also, e.g., [36]). We introduce the reader to finite automata and regular languages (Section 2.1) and then we define the AUTOMATON constraint predicate in three stages: first its particular case that is also known as the REGULAR constraint predicate [43] (Section 2.2), and then two orthogonal extensions, namely predicate automata (Section 2.3) and automata with accumulators[1] (Section 2.4). Finally, we compose the two extensions into predicate automata with accumulators (Section 2.5).

2.1 Finite Automata and Regular Languages

A *deterministic finite automaton* (*DFA*) [36], or *automaton* for short, is a tuple $\langle Q, \Gamma, \delta, \rho_0, Q_a \rangle$ where Q is the finite set of states; Γ is the finite alphabet; ρ_0 is a state in Q denoting the initial state; Q_a is a subset of Q denoting the accepting states; and δ is a total function from $Q \times \Gamma$ to Q denoting the transition function. If $\delta(\rho, a) = \rho'$, then we say that there is a transition from state ρ to state ρ' that *consumes* alphabet symbol a; this is here often written as:

$$\rho \xrightarrow{a} \rho'$$

A *word* is here a sequence of symbols from a given alphabet. Let Γ^* denote the infinite set of words built from Γ, including the empty word, denoted ε. The *extended transition function* $\widehat{\delta}\colon Q \times \Gamma^* \to Q$ for words (instead of symbols) is recursively defined by $\widehat{\delta}(\rho, \varepsilon) = \rho$ and $\widehat{\delta}(\rho, wa) = \delta(\widehat{\delta}(\rho, w), a)$ for a word w and symbol a. Note that both δ and $\widehat{\delta}$ are total functions. A word $w = a_1 a_2 \cdots a_{n-1} a_n$ is *accepted* by the automaton if there is a chain of transitions:

$$\rho_0 \xrightarrow{a_1} \rho_1 \xrightarrow{a_2} \ldots \xrightarrow{a_{n-1}} \rho_{n-1} \xrightarrow{a_n} \rho_n$$

[1] Automata with accumulators are called counter automata in Paper II, and memory-DFAs in Paper III and Paper V.

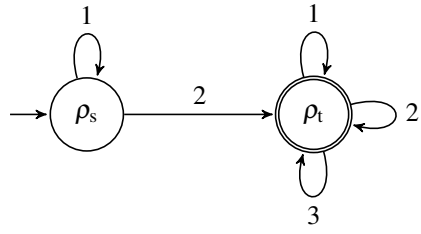


Figure 2.1. DFA for the regular expression $1^*2(1|2|3)^*$.

such that $\rho_n \in Q_a$, that is if $\widehat{\delta}(\rho_0, w) \in Q_a$.

One often uses pictures to define finite automata. For example, in Figure 2.1, we define an automaton with two states, $Q = \{\rho_s, \rho_t\}$, represented by circles, and an alphabet of three symbols, $\Gamma = \{1,2,3\}$, on the transitions. The initial state $\rho_0 = \rho_s$ is indicated by an arrow coming from nowhere, and an accepting state is represented by a double circle, and so $Q_a = \{\rho_t\}$. The transition function is represented by the annotated arrows, that is $\delta(\rho, a) = \rho'$ if there is an arrow from ρ to ρ' annotated with a. For each state, there is one outgoing arrow per alphabet symbol; any missing arrow is assumed to go to an implicit non-accepting state, on which there is a self-looping arrow for every symbol of the alphabet, so that no accepting state is reachable from that state. For example, in Figure 2.1, the missing transition from state ρ_s on symbol 3 goes to such an implicit non-accepting state.

A *language* is, in the formal sense, a set of words together with a set of formation rules. A *regular language* is a language that can be defined using a regular expression. Regular expressions describe patterns over words; for example, the regular expression $1^*2(1|2|3)^*$ over the alphabet $\Gamma = \{1,2,3\}$ defines the set of words that start with zero or more 1s, followed by exactly one 2, and ending with any number of symbols, possibly zero, from Γ. We say that $1^*2(1|2|3)^*$ *defines* a regular language. We denote the language defined by a regular expression σ by $\mathcal{L}(\sigma)$. For example, the words 2 and 121 are words in $\mathcal{L}(1^*2(1|2|3)^*)$, whereas the words 11 and 13 are not. We can also relate regular languages to automata: a language is regular if and only if every word in the language is accepted by a deterministic finite automaton. For this reason, we say that an automaton *accepts* a regular language $\mathcal{L}$, since it accepts all the words in $\mathcal{L}$ and rejects all the other ones. For example, the automaton in Figure 2.1 accepts the language of the regular expression $1^*2(1|2|3)^*$.

A *deterministic finite transducer* [48] is a tuple $\langle Q, \Gamma, \Gamma', \delta, \rho_0, Q_a \rangle$, where Q is the finite set of *states*, Γ is the finite *input alphabet*, Γ' is the finite *output alphabet*, $\delta \colon Q \times \Gamma \to Q \times \Gamma'^*$ is the *transition function*, which must be total, $\rho_0 \in Q$ is the *initial state*, and $Q_a \subseteq Q$ is the set of *accepting states*. When $\delta(\rho, a) = \langle \rho', a' \rangle$, there is a transition from state ρ to state ρ' upon consuming the input symbol a and *producing* the sequence a' of output symbols: we write

this as $\rho \xrightarrow{a:a'} \rho'$. Note that a deterministic finite automaton is a transducer without an output alphabet. In a graphical representation of a transducer, a transition is depicted by an arrow between two states, possibly the same, and is annotated by a consumed input symbol, followed by a colon and a sequence of produced output symbols (see Figure 3.4 for an example).

2.2 Describing Constraints by Deterministic Finite Automata

Any constraint (on a sequence of decision variables) whose extensional definition forms a regular language can be described by an automaton. In fact, any constraint on a finite sequence of decision variables that range over finite domains can be described by an automaton, since every finite language is a regular language. The REGULAR$(\mathcal{A}, \mathcal{V})$ constraint [13, 43] holds if the constraint described by the deterministic finite automaton $\mathcal{A}$ (or its equivalent regular expression) holds for the sequence $\mathcal{V}$ of decision variables, that is if $\mathcal{A}$ accepts the sequence of values of $\mathcal{V}$.

In practice, an automaton may however have a number of states that is exponential in the number of decision variables of the constraint, such as for the ALLDIFFERENT constraint predicate, as discussed in [43].

A REGULAR$(\mathcal{A}, \mathcal{V})$ constraint can be implemented either via a specialised propagator [43] or via decomposition into a conjunction of constraints [13]. We here take the latter approach because it will be more convenient when defining the extensions in Sections 2.3 and 2.4. For a given automaton $\mathcal{A} = \langle Q, \Gamma, \delta, \rho_0, Q_a \rangle$, we define a new constraint predicate T extensionally by the following set:

$$\{\langle q, a, q' \rangle \mid q \xrightarrow{a} q'\} \tag{2.1}$$

That is, $\mathrm{T}(q, a, q')$ is satisfied whenever there is a transition in $\mathcal{A}$ from state q to state q' that consumes symbol a. A REGULAR$(\mathcal{A}, \langle v_1, \ldots, v_n \rangle)$ constraint is then decomposed into the following conjunction of $n+2$ constraints, called the *transition constraints*:

$$q_0 = \rho_0 \wedge \mathrm{T}(q_0, v_1, q_1) \wedge \cdots \wedge \mathrm{T}(q_{n-1}, v_n, q_n) \wedge q_n \in Q_a \tag{2.2}$$

where $q_0, q_1, \ldots, q_{n-1}, q_n$ are new decision variables, called the *state variables*, with domain Q. For contrast, we call $v_1, \ldots, v_n$ the *problem variables*.

This decomposition actually works unchanged for *non-deterministic finite automata* (NFA), where δ is a relation rather than a total function (for example, see Figure 2.2), but we have elected to restrict our focus to deterministic ones, in order to ease the notation.

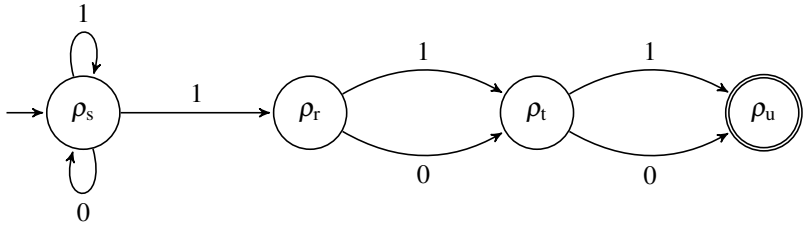


Figure 2.2. NFA for the regular expression $(0|1)^*1(0|1)^2$: all 0/1 sequences that have a 1 two characters from the end of the sequence.

2.3 Describing Constraints by Predicate Automata

The automata in [13] are more powerful than those in [43]: The alphabet symbols can be predicates on variables, and all predicates on an accepting path must be satisfied.

The definition presented here is parametrised by a suitable set of predicates. Let $\mathbf{Pred}_k$ be a set of k-ary predicates in some suitable language. That is, a predicate takes a vector, $\mathcal{P}$, of k values.

A *k-ary predicate automaton* is a tuple $\langle Q, \Gamma, \delta, \phi, \rho_0, Q_a \rangle$, where Q, Γ, δ, ρ_0, and Q_a are exactly as for a deterministic finite automaton, and ϕ is a function from Γ to $\mathbf{Pred}_k$. For all k-ary value vectors $\mathcal{P}$ and all distinct symbols a_1 and a_2 of Γ, we must have that $\phi(a_1)(\mathcal{P}) \wedge \phi(a_2)(\mathcal{P})$ is false (that is, any two predicates must be mutually exclusive). A sequence of k-ary vectors of values $\mathcal{P}_1\mathcal{P}_2\cdots\mathcal{P}_{n-1}\mathcal{P}_n$ is *accepted* by the automaton if there exists a chain of transitions

$$\rho_0 \xrightarrow{a_1} \rho_1 \xrightarrow{a_2} \ldots \xrightarrow{a_{n-1}} \rho_{n-1} \xrightarrow{a_n} \rho_n$$

such that $\rho_n \in Q_a$ and $\phi(a_i)(\mathcal{P}_i)$ is true for all $1 \leq i \leq n$. Such a chain of transitions can be written as

$$\rho_0 \xrightarrow{\phi(a_1)(\mathcal{P}_1)} \rho_1 \xrightarrow{\phi(a_2)(\mathcal{P}_2)} \ldots \xrightarrow{\phi(a_{n-1})(\mathcal{P}_{n-1})} \rho_{n-1} \xrightarrow{\phi(a_n)(\mathcal{P}_n)} \rho_n$$

Again, we often define k-ary predicate automata by pictures. The convention is similar to normal finite automata, except that the transition labels are predicates. We assume that each distinct predicate is associated with a distinct symbol of the alphabet Γ, and that the function ϕ is defined by the predicate labels in the picture.

For example, in Figure 2.3, the function ϕ could be defined by lambda expressions as follows: $\phi(1) = \lambda x, y : x = y$, $\phi(2) = \lambda x, y : x < y$, and $\phi(3) = \lambda x, y : x > y$. Consider the constraint that the sequence of decision variables $\mathcal{V}$ be lexicographically less than the sequence of decision variables $\mathcal{W}$, which is denoted by $\mathcal{V} <_{\text{lex}} \mathcal{W}$. For the fixed sequences $\mathcal{V} = \langle 1,2,5,6 \rangle$ and $\mathcal{W} = \langle 1,3,4,7 \rangle$, the sequence $\langle 1,1 \rangle\langle 2,3 \rangle\langle 5,4 \rangle\langle 6,7 \rangle$ of binary vectors, obtained by zipping $\mathcal{V}$ and $\mathcal{W}$ together, is accepted by the binary predicate automaton

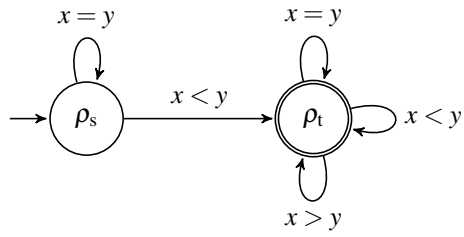


Figure 2.3. A k-ary predicate automaton with $k = 2$ describing the $<_{\text{lex}}$ constraint predicate.

($k = 2$) in Figure 2.3 because the transition chain

$$\rho_{\text{s}} \xrightarrow{1=1} \rho_{\text{s}} \xrightarrow{2<3} \rho_{\text{t}} \xrightarrow{5>4} \rho_{\text{t}} \xrightarrow{6<7} \rho_{\text{t}}$$

ends in the accepting state ρ_{t}.

Given a predicate automaton $\langle Q, \Gamma, \delta, \phi, \rho_0, Q_{\text{a}} \rangle$, the automaton $\langle Q, \Gamma, \delta, \rho_0, Q_{\text{a}} \rangle$ is referred to as the *underlying automaton* of the predicate automaton. For example, the automaton in Figure 2.1 is the underlying automaton of the predicate automaton in Figure 2.3.

In [13], the AUTOMATON$(\mathcal{A}, \mathcal{V})$ constraint holds if and only if the constraint described by the automaton $\mathcal{A}$ holds for the sequence $\mathcal{V}$ of decision variables, where $\mathcal{A}$ is a predicate automaton implemented with the help of reification. The constraint predicate T defined in (2.1) is used for the following $n+2$ transition constraints:

$$q_0 = \rho_0 \wedge \text{T}(q_0, S_1, q_1) \wedge \cdots \wedge \text{T}(q_{n-1}, S_n, q_n) \wedge q_n \in Q_{\text{a}} \tag{2.3}$$

These transition constraints are like (2.2), but are expressed for *new* decision variables $S_1, \ldots, S_n$, which are connected as follows to the sequence of problem variables $\mathcal{V}$ via the automaton predicates and reification: given an n-length sequence $\mathcal{V} = \langle \mathcal{V}_1, \ldots, \mathcal{V}_n \rangle$ of k-ary vectors of problem variables, we add the following n constraints, called the *signature constraints*:

$$\bigwedge_{i=1}^{n} \left(\bigwedge_{a \in \Gamma} (S_i = a \Leftrightarrow \phi(a)(\mathcal{V}_i)) \right) \tag{2.4}$$

where the S_i are called the *signature variables*, with domain Γ. Hence $\mathbf{Pred}_k$ contains whatever can be implemented as reified constraints in the underlying CP solver (note that most global constraint predicates can be reified [12]). For example, in Figure 2.3, the binary predicate automaton on the two sequences of variables $\mathcal{V} = \langle v_1, \ldots, v_n \rangle$ and $\mathcal{W} = \langle w_1, \ldots, w_n \rangle$ requires the transition constraints (2.3) and the following signature constraints for all $1 \leq i \leq n$:

$$(S_i = 1 \Leftrightarrow v_i = w_i) \wedge (S_i = 2 \Leftrightarrow v_i < w_i) \wedge (S_i = 3 \Leftrightarrow v_i > w_i)$$

2.4 Describing Constraints by Automata with Accumulators

While the class of constraint predicates that can be described by (predicate) automata is large (60 of the 381 constraint predicates of the *Global Constraint Catalogue* [10] are described that way), it is often the case that (predicate) automata are very large or specific to a problem instance. The second extension in [13] is the use of integer accumulators[2] that are initialised at the start and evolve through accumulator-updating operations coupled to the transitions of the automaton. Such automata with accumulators allow the capture of non-regular languages and yield (even for regular languages) automata that are often much smaller if not instance-independent and enable constraint predicates to be described succinctly or generically. The two extensions are orthogonal and can be composed, so we define this second extension in isolation.

Again, we give a definition that is parametric, namely on the class of accumulator-updating functions. An accumulator-updating operation consists of a sequence of assignments to some accumulators (the accumulators without assignments are left unchanged), possibly guarded by a condition on the current accumulator values and the variables. Let $\mathbf{AccUpdate}_\ell$ be a set of ℓ-ary accumulator-updating functions. That is, given a function $\psi \in \mathbf{AccUpdate}_\ell$ and a vector of accumulators $\mathcal{C} \in \mathbb{Z}^\ell$, we have that $\psi(\mathcal{C})$ is a new vector in $\mathbb{Z}^\ell$.

An *ℓ-ary automaton with accumulators* is a tuple $\langle Q, \Gamma, \delta, \rho_0, \mathcal{C}_0, Q_\mathrm{a}, \alpha \rangle$ where Q, Γ, ρ_0, and Q_a are exactly as for a deterministic finite automaton; vector $\mathcal{C}_0$ has the initial values of a vector $\mathcal{C}$ of ℓ accumulators; and δ is a function from $Q \times \Gamma$ to $Q \times \mathbf{AccUpdate}_\ell$. If $\delta(\rho, a) = (\rho', \psi)$ and $\psi(\mathcal{C}) = \mathcal{C}'$, then we write

$$(\rho, \mathcal{C}) \xrightarrow{a} (\rho', \mathcal{C}')$$

and similarly for its extended version $\widehat{\delta}$. A word $a_1 a_2 \cdots a_{n-1} a_n$ is *accepted* by the automaton if there is a chain of transitions

$$(\rho_0, \mathcal{C}_0) \xrightarrow{a_1} (\rho_1, \mathcal{C}_1) \xrightarrow{a_2} \ldots \xrightarrow{a_{n-1}} (\rho_{n-1}, \mathcal{C}_{n-1}) \xrightarrow{a_n} (\rho_n, \mathcal{C}_n)$$

such that $\rho_n \in Q_\mathrm{a}$. Finally, $\alpha \colon Q_\mathrm{a} \times \mathbb{Z}^k \to \mathbb{Z}$ is called the *acceptance function* and transforms the accumulators at an accepting state into an integer. Given a word w, the automaton with accumulators returns $\alpha(\widehat{\delta}(\langle \rho_0, \mathcal{C}_0 \rangle, w))$ if w is accepted. Note that δ, $\widehat{\delta}$, and α are total functions.

As with automata, one often uses pictures to define automata with accumulators. The set Q of states, the set Q_a of accepting states and the initial state ρ_0 are defined exactly as for an automaton. The transition function is also defined by the annotated arrows, but the label on the arrow of a transition consists of a symbol followed by an accumulator-updating operation between curly braces. That is $\delta(\rho, a) = (\rho', \psi)$ if there is an arrow from ρ to ρ' annotated with $a\ \{\psi\}$.

[2] Accumulators are called counters in [13] and in Paper II.

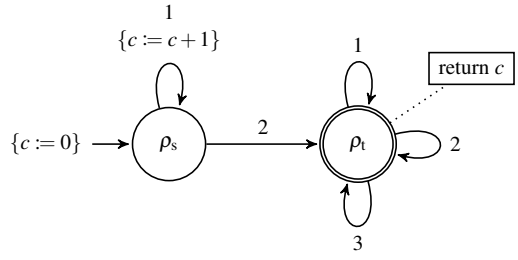


Figure 2.4. Automaton with $\ell = 1$ accumulator for the regular expression $1^*2(1|2|3)^*$. Accumulator c maintains the length of the longest prefix matching the regular expression 1^* of the sequence of symbols consumed so far.

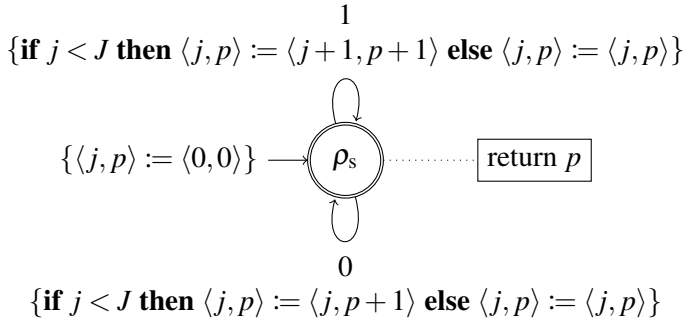


Figure 2.5. An ℓ-ary automaton with accumulators with $\ell = 2$ accumulators describing the JTHNONZEROPOS$(\mathcal{V}, J, P)$ constraint [10], which holds if and only if P is the position (counting from 1) of the J^{th} non-zero element of the sequence $\mathcal{V} = \langle V_1, \ldots, V_n \rangle$. Accumulator j maintains the number of non-zero values among the J first non-zero elements of $\mathcal{V}$, while accumulator p maintains the number of all values within that prefix of $\mathcal{V}$. Upon acceptance, the final value of the vector of accumulators $\langle j, p \rangle$ must be $\langle J, P \rangle$. The signature constraints are $S_i = 0 \Leftrightarrow V_i = 0$ and $S_i = 1 \Leftrightarrow V_i \neq 0$.

For example, in Figure 2.4, the self-loop on ρ_s depicts that $\delta(\rho_s, 1) = (\rho_s, \langle c + 1 \rangle)$ for all c. If an update corresponds to the identity function, then we do not depict it; for example, the three self-loops on ρ_t have no depicted updates, as $\langle c \rangle \coloneqq \langle c \rangle$. If an update involves only one accumulator, then we omit the angled brackets; for example, the self-loop on ρ_s has $c \coloneqq c + 1$ instead of $\langle c \rangle \coloneqq \langle c + 1 \rangle$. The acceptance function α transforms the vector of accumulators $\langle c \rangle$ at ρ_t into c, and is depicted by a box linked to ρ_t by a dotted line. Note that an accumulator-updating operation can also be guarded by a condition on the current accumulator values and the problem variables, as can be seen in Figure 2.5.

In [13], constraint predicates described by automata with accumulators are decomposed into transition constraints that are slightly extended to include information about the values of the accumulators. We define the transition

constraint predicate T extensionally by the following set:

$$\{\langle q,\mathcal{C},a,q',\mathcal{C}'\rangle \mid (q,\mathcal{C}) \xrightarrow{a} (q',\mathcal{C}')\}$$

An AUTOMATON$(\mathcal{A},\mathcal{V},R)$ constraint on a sequence of n problem variables, with $\mathcal{V} = \langle v_1,\ldots,v_n\rangle$, and an result parameter (either an integer constant or a decision variable), R, is then decomposed into the following conjunction of $n+4$ transition constraints:

$$\begin{aligned} q_0 = \rho_0 \wedge c_0 = \mathcal{C}_0 \wedge \mathrm{T}(q_0,c_0,v_1,q_1,c_1) \wedge \cdots \\ \wedge\, \mathrm{T}(q_{n-1},c_{n-1},v_n,q_n,c_n) \wedge q_n \in Q_\mathrm{a} \wedge \alpha(c_n) = R \end{aligned} \quad (2.5)$$

where $q_0,\ldots,q_n$ are state variables, with domain Q, while $c_0,\ldots,c_n$ are vectors of new integer decision variables, called *accumulator variables*.

Upon acceptance, we must have $\alpha(c_n) = R$; initially, we have $c_0 = \mathcal{C}_0$ where $\mathcal{C}_0$ is a parameter of the automaton. It is also important not to mix up the vectors of variables $c_0,\ldots,c_n$ with the vector c of accumulators of the automaton.

By abuse of language, when there is $\ell = 1$ accumulator, we often refer to vector $\mathcal{C}_0$ as the initial value (rather than the vector with the initial value), to vector $\mathcal{C}$ as an accumulator value, and to vector c_i as an accumulator variable.

2.5 Describing Constraints by Predicate Automata with Accumulators

A $\langle k,\ell\rangle$*-ary predicate automaton with accumulators*, or simply *automaton*, is an automaton that is both a k-ary predicate automaton and an ℓ-ary automaton with accumulators. A $\langle k,\ell\rangle$-ary predicate automaton with accumulators is a tuple $\langle Q,\Gamma,\delta,\phi,\mathcal{C}_0,\rho_0,Q_\mathrm{a},\alpha\rangle$ where Q, Γ, ρ_0, and Q_a are exactly as for a automaton; ϕ is a function from Γ to $\mathbf{Pred}_k$; vector $\mathcal{C}_0$ has the initial values of the ℓ accumulators; and δ is a function from $Q\times\Gamma$ to $Q\times\mathbf{AccUpdate}_\ell$.

For example, in Figure 2.6, we define a predicate automaton with accumulators where $Q = \{\rho_\mathrm{s},\rho_\mathrm{t}\}$ has two states, $\Gamma = \{1,2,3\}$ is an alphabet of three symbols, ϕ is the function defined by $\phi(1) = \lambda x,y : x = y$, $\phi(2) = \lambda x,y : x < y$, and $\phi(3) = \lambda x,y : x > y$, the accumulator c has the initial value $\mathcal{C}_0 = \langle 0\rangle$, $Q_\mathrm{a} = \{\rho_\mathrm{t}\}$ has one accepting state, and the transition function δ is as indicated with the annotated arrows. The arrow indicating the initial state of the automaton is preceded by the sequence of initialising assignments of the accumulators. The label on the arrow of a transition consists of a predicate followed by an accumulator-updating operation between curly braces.

Since a predicate automaton with accumulators consumes the signature variables S_i instead of the k-ary vectors of problem variables $\mathcal{V}_i$, the transition constraints (2.5) given in Section 2.4 for an AUTOMATON$(\mathcal{A},\mathcal{V},R)$ constraint,

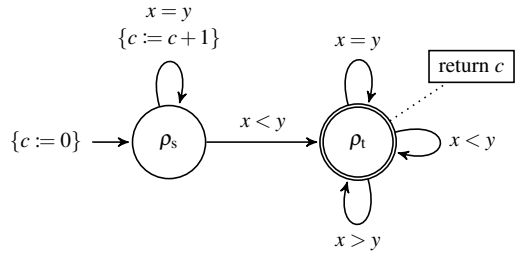


Figure 2.6. A $\langle 2,1\rangle$-ary predicate automaton with accumulators describing a constraint predicate on two sequences of decision variables $\mathcal{V}$ and $\mathcal{W}$ which holds if and only if $\mathcal{V} <_{\text{lex}} \mathcal{W}$ holds and accumulator c denotes the length of the longest common prefix between $\mathcal{V}$ and $\mathcal{W}$.

with $\mathcal{V} = \langle \mathcal{V}_1, \ldots, \mathcal{V}_n \rangle$, are transformed into the following:

$$q_0 = \rho_0 \wedge c_0 = \mathcal{C}_0 \wedge \mathrm{T}(q_0, c_0, S_1, q_1, c_1) \wedge \cdots \\ \wedge \mathrm{T}(q_{n-1}, c_{n-1}, S_n, q_n, c_n) \wedge q_n \in Q_{\mathrm{a}} \wedge \alpha(c_n) = R \qquad (2.6)$$

Even though the transition constraints are defined extensionally, they can be efficiently implemented using the CASE constraint predicate of SICStus Prolog [26] and the ELEMENT constraint predicate: see [9] for details.

We collectively refer to the signature variables S_i, accumulator variables c_i, and state variables q_i as the *induced variables* of the automaton.

3. Time-Series Constraints

"Oh dear! Oh dear! I shall be late!"

Alice's Adventures in Wonderland
LEWIS CARROLL

We introduce time series and time-series constraints. In Section 3.1 we define time series and explain how to describe time-series constraint predicates using the four-layered approach of [11]. In Section 3.2 we define *seed transducers* and show how to describe a time-series constraint predicate using such a transducer. Finally, in Section 3.3 we show how to synthesise from a seed transducer an automaton with accumulators that describes the predicate, from which the framework of [13] can be used to induce a decomposition of a constraint predicate.

3.1 Definitions

A *time series* is here a sequence of integers, corresponding to measurements taken over a time interval, such as the output of electric power stations over multiple days [17], the manpower required in a call centre [6], environmental data (temperature, humidity, CO_2 level) in buildings, or the daily capacity of a hospital clinic over a period of years. Time series are often constrained by physical or organisational limits, which restrict the evolution of a series. For example, the number of plateaus may be constrained, or the sum of the peak maxima, or the minimum of the valley widths.

In [11] it was shown that many useful constraints $\gamma(\langle X_1,\ldots,X_n\rangle,N)$ on an unknown time series $X = \langle X_1,\ldots,X_n\rangle$ of given length n can be described by a triple $\langle \pi, f, g\rangle$, where π is called a *pattern* and in this introductory chapter is one of the regular expressions in Figure 3.1 over the alphabet $\Sigma = \{\text{`<'}, \text{`='}, \text{`>'}\}$,[1] while $f \in \{\texttt{max}, \texttt{min}, \texttt{one}, \texttt{surface}, \texttt{width}\}$[2] is called a *feature*, and $g \in \{\texttt{Max}, \texttt{Min}, \texttt{Sum}\}$ is called an *aggregator*; integer variable N is constrained to be the aggregation, computed using g, of the list of values of feature f for all maximal words matching π in X. For example, given a time series, a constraint on the sum of the peak maxima can be specified by the aggregator $g = \texttt{Sum}$, the feature $f = \texttt{max}$, and the pattern $\pi = \texttt{Peak}$ (given in Example 1 below) corresponding to a peak within the time series. We denote a time-series constraint predicate specified by $\langle \pi, f, g\rangle$ as g_f_π.

[1] For a formal definition of pattern, see Paper I.

[2] Feature `one` corresponds to the value 1 for any pattern occurrence and it is used solely for the purpose of counting the number of pattern occurrences.

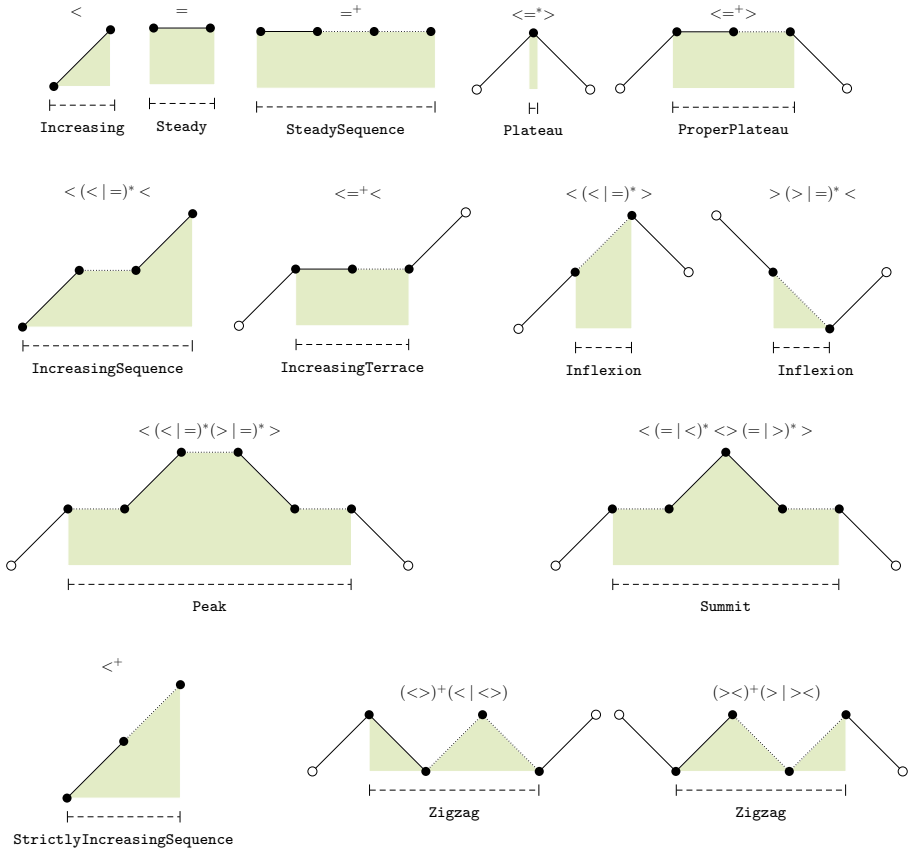


Figure 3.1. Illustration of the patterns in [11], with time on the horizontal axis and the measurements on the vertical axis: only the relative vertical positions of adjacent points matter, not their magnitudes. The width of the pattern is shown with a dashed line. Note that black points are part of a pattern occurrence, but not the white ones. Dash-dotted lines include an arbitrary number of points. Shaded areas approximate the surface of the pattern occurrence. Permuting the symbols '<' and '>' we obtain the remaining patterns in [11], namely `Decreasing`, `Plain`, `ProperPlain`, `DecreasingSequence`, `DecreasingTerrace`, `Valley`, `Gorge`, and `StrictlyDecreasingSequence`. (Adaptation of figures in [5].)

A sequence $S = \langle S_1, \ldots, S_{n-1} \rangle$, called the *signature* and containing *signature variables*, is linked to a time series $X = \langle X_1, \ldots, X_n \rangle$ via the *signature constraints* $(X_i < X_{i+1} \Leftrightarrow S_i = \text{'<'}) \wedge (X_i = X_{i+1} \Leftrightarrow S_i = \text{'='}) \wedge (X_i > X_{i+1} \Leftrightarrow S_i = \text{'>'})$ for all $i \in [1, n-1]$. We now introduce our running example.

Example 1. The time series $X = \langle 4,4,0,0,2,4,4,7,4,1,1,5,5,5,5,5,5,3 \rangle$ has the signature $S =$ '=>=<<=<>>=<=====>'. Consider the regular expression `Peak` = '<(<|=)*(>|=)*>': a *peak* within a time series corresponds to a

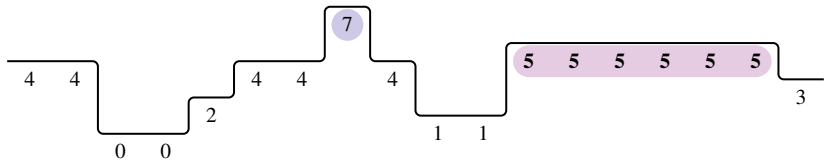


Figure 3.2. Visual representation of the MIN_MAX_PEAK$(X,5)$ constraint, with $X = \langle 4,4,0,0,2,4,4,7,4,1,1,5,5,5,5,5,5,3\rangle$.

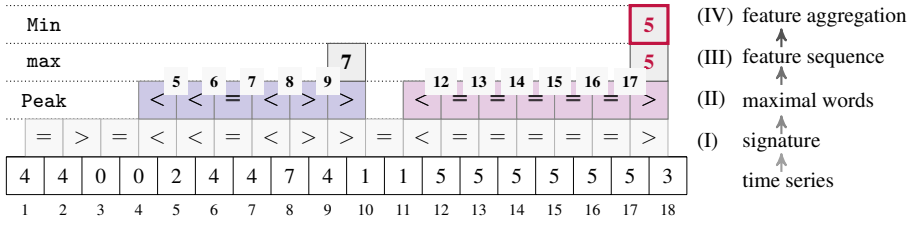


Figure 3.3. Describing time-series constraints as a function composition, exemplified on the MIN_MAX_PEAK$(X,5)$ constraint, with $X = \langle 4,4,0,0,2,4,4,7,4,1,1,5,5,5,5,5,5,3\rangle$. (Adaptation of a figure in [5].)

maximal word matching `Peak` in the signature. The time series X contains two peaks, namely $\langle 0,2,4,4,7,4,1\rangle$ and $\langle 1,5,5,5,5,5,5,3\rangle$, visible in Figure 3.2.

The `max` feature value of a peak is its highest value. The highest values of the two peaks in the time series X are 7 and 5 respectively.

Hence the lowest peak, obtained by using the aggregator `Min`, has as highest value $N = 5$, that is, the highest point of the lowest peak in the time series X. The underlying constraint is MIN_MAX_PEAK(X,N).

Figure 3.3 shows how to check MIN_MAX_PEAK$(X,5)$ by:

(I) building the signature by comparing adjacent values of the time series;
(II) finding in the signature all maximal words matching the regular expression `Peak` = '<(<|=)*(>|=)*>';
(III) computing the `max` feature value of each such peak; and
(IV) aggregating the feature values using the `Min` aggregator.

□

3.2 Specifying a Pattern by a Transducer

In [11] it was shown that many of the patterns for time-series constraint predicates can be specified by transducers. The output alphabet of such a transducer, called the *phase alphabet*, consists of symbols that denote the phases of identifying the maximal words matching a pattern in a signature. The symbols of the phase alphabet and their meaning are as follows:

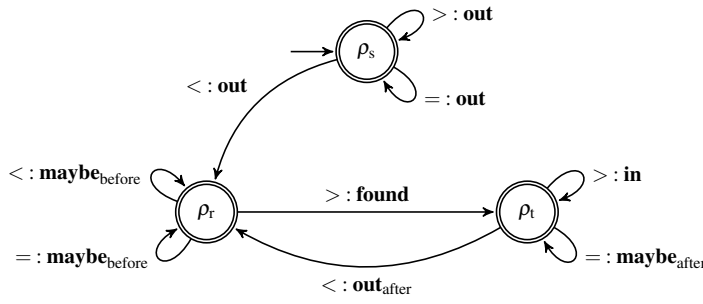


Figure 3.4. Transducer for `Peak` = '<(<|=)*(>|=)*>'.

- **found**: the symbol consumed is in a new pattern occurrence that may have started before and may be extended.
- **found**$_{\text{end}}$: the symbol consumed is the last symbol in a new pattern occurrence that may have started before.
- **maybe**$_{\text{before}}$: the symbol consumed may belong to a pattern occurrence, but this must be confirmed by producing a **found** or **found**$_{\text{end}}$.
- **out**$_{\text{reset}}$: the symbol consumed is outside any pattern occurrence and all the **maybe**$_{\text{before}}$ produced just before are outside any pattern occurrence.
- **in**: the symbol consumed is inside a pattern occurrence for which a **found** was already produced and all symbols between the one producing such a **found** and the one being consumed belong to the pattern occurrence.
- **maybe**$_{\text{after}}$: the symbol consumed may belong to a pattern occurrence for which a **found** was already produced, but this must be confirmed by producing **in** while consuming the rest of the signature.
- **out**$_{\text{after}}$: a pattern occurrence ended at the last **found** or **in** symbol produced.
- **out**: the symbol consumed is not in a pattern occurrence.

Each of the 20 patterns in [11] is described by what is there called a seed transducer. A *seed transducer* is a deterministic finite transducer with only accepting states, whose input alphabet, called the *topological alphabet*, is $\Sigma = \{\text{'<'}, \text{'='}, \text{'>'}\}$, and whose output alphabet is the phase alphabet.[3]

Example 2. Figure 3.4 shows a seed transducer with three states, $Q = \{\rho_s, \rho_r, \rho_t\}$, an input alphabet of three symbols, $\Gamma = \{\text{'<'}, \text{'='}, \text{'>'}\}$, and an output alphabet of six symbols, $\Gamma' = \{\textbf{out}, \textbf{maybe}_{\text{before}}, \textbf{found}, \textbf{in}, \textbf{maybe}_{\text{after}}, \textbf{out}_{\text{after}}\}$. The initial state is $\rho_0 = \rho_s$, and the set of accepting states is $Q_a = \{\rho_s, \rho_r, \rho_t\}$. For each state, there is one outgoing arrow per symbol of the input alphabet. □

[3]The phase alphabet is called the *semantic alphabet* in [11] and in Paper I.

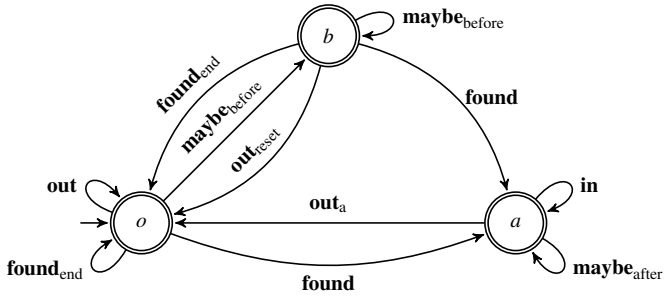


Figure 3.5. Automaton accepting the output language of a well-formed transducer. (Adaptation of a figure in [5]).

From now on we refer to seed transducers simply as transducers.

Example 3. Consider the transducer in Figure 3.4 for the pattern `Peak` = '<(<|=)*(>|=)*>' and the signature S = '=>=<<=<>>=<=====>'. The transitions are as follows:

$$\rho_s \xrightarrow{=:\textbf{out}} \rho_s \xrightarrow{>:\textbf{out}} \rho_s \xrightarrow{=:\textbf{out}} \rho_s \xrightarrow{<:\textbf{out}} \rho_r \xrightarrow{<:\textbf{maybe}_{\text{before}}} \rho_r$$
$$\xrightarrow{=:\textbf{maybe}_{\text{before}}} \rho_r \xrightarrow{<:\textbf{maybe}_{\text{before}}} \rho_r \xrightarrow{>:\textbf{found}} \rho_t \xrightarrow{>:\textbf{in}} \rho_t$$
$$\xrightarrow{=:\textbf{maybe}_{\text{after}}} \rho_t \xrightarrow{<:\textbf{out}_{\text{after}}} \rho_r \xrightarrow{=:\textbf{maybe}_{\text{before}}} \rho_r \xrightarrow{=:\textbf{maybe}_{\text{before}}} \rho_r$$
$$\xrightarrow{=:\textbf{maybe}_{\text{before}}} \rho_r \xrightarrow{=:\textbf{maybe}_{\text{before}}} \rho_r \xrightarrow{=:\textbf{maybe}_{\text{before}}} \rho_r \xrightarrow{>:\textbf{found}} \rho_t$$

The two **found** correspond to two peaks: the first one corresponds to the word from the first $\textbf{maybe}_{\text{before}}$ to the first **in** (i.e., the word $\textbf{maybe}^3_{\text{before}}$ **found in**); the second one corresponds to the word from just after the last $\textbf{out}_{\text{after}}$ to the last **found** (i.e., the word $\textbf{maybe}^5_{\text{before}}$ **found**). □

Seed transducers must obey a set of wellformedness conditions given in [11]. A transducer is *well-formed* with respect to a pattern π if the following conditions hold:

- Its output language is a subset of the language accepted by the automaton in Figure 3.5,
- All occurrences of π in a signature produce maximal words matching the regular expression $\textbf{maybe}^*_{\text{b}}(\textbf{found}_{\text{e}} \mid \textbf{found}(\textbf{maybe}^*_{\text{a}}\textbf{in})^*)$.

Each of the 20 transducers in [11] was hand-crafted and manually verified for wellformedness.

In Paper I we characterise the class of patterns that can be handled by the synthesis of automaton-based descriptions of time-series constraint predicates in [11], which makes it possible to decide when the synthesiser is applicable,

initialisation	$r \coloneqq \mathrm{id}_g^f$	$c \coloneqq \mathrm{id}_g^f$	$d \coloneqq \mathrm{id}_f$
return	$\phi_g(r,c)$		
phase		accumulator updates	
letters	update of r	update of c	update of d
out$_{\text{reset}}$			$d \coloneqq \mathrm{id}_f$
out$_{\text{after}}$	$r \coloneqq \phi_g(r,c)$	$c \coloneqq \mathrm{id}_g^f$	$d \coloneqq \mathrm{id}_f$
maybe$_{\text{before}}$			$d \coloneqq \phi_f(d,\delta_f^i)$
maybe$_{\text{after}}$			$d \coloneqq \phi_f(d,\delta_f^i)$
found		$c \coloneqq \phi_f(d,\delta_f^i)$	$d \coloneqq \mathrm{id}_f$
found$_{\text{end}}$	$r \coloneqq \phi_g(r,\phi_f(d,\delta_f^i))$		$d \coloneqq \mathrm{id}_f$
in		$c \coloneqq \phi_f(c,\phi_f(d,\delta_f^i))$	$d \coloneqq \mathrm{id}_f$
out			

(a) Original decoration table

Feature f	id_f	$\min_f$	$\max_f$	ϕ_f	δ_f^i
`one`	1	1	1	1	1
`width`	0	0	$+\infty$	$+$	1
`surface`	0	$-\infty$	$+\infty$	$+$	X_i
`max`	$-\infty$	$-\infty$	$+\infty$	max	X_i
`min`	$+\infty$	$-\infty$	$+\infty$	min	X_i

(b) Features

Aggregator g	ϕ_g	id_g^f
`Max`	min	$\min_f$
`Min`	max	$\max_f$
`Sum`	$+$	0

(c) Aggregators

Figure 3.6. (a) Decoration table used for synthesising an automaton from a transducer, a feature f, and an aggregator g; (b) Features: identity, minimum, and maximum values; the operators ϕ_f and δ_f^i recursively define the feature value v_u of a time series $\langle X_\ell,\ldots,X_u\rangle$ by $v_\ell = \phi_f(\mathrm{id}_f,\delta_f^\ell)$ and $v_i = \phi_f(v_{i-1},\delta_f^i)$ for $i > \ell$, where δ_f^i is the contribution of X_i to v_u. (c) Aggregators: operators and identity values relative a feature f. (Adaptation of figures in [5].)

and we give an algorithm for generating a wellformed seed transducer from such a pattern.

3.3 Synthesising Automaton-Based Descriptions of Time-Series Constraint Predicates from a Transducer

The synthesis of automaton-based descriptions of time-series constraint predicates in [11] relies on a declarative encoding of procedural knowledge into what is called a *decoration table*. The decoration tables are parametrised by features and aggregators, and define substitution rules on the transducers that allow an automaton with three accumulators to be synthesised. Recall

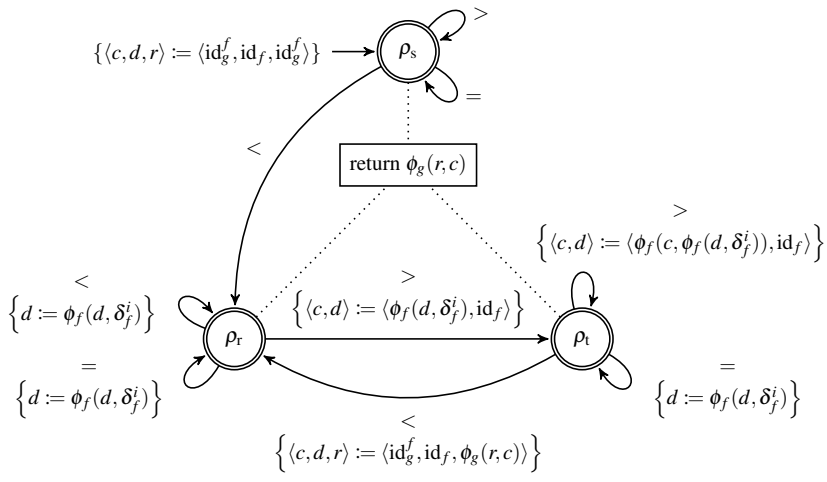


Figure 3.7. Automaton for any $g_f_$PEAK

that, from an automaton, the framework of [13] decomposes a constraint with the described global constraint predicate into a conjunction of constraints for whose predicates there already are propagators. These constraints collectively give the semantics of the described time-series constraint predicate and provide the propagation. The decoration table in Figure 3.6a provides a substitution rule for each symbol of the phase alphabet, where accumulator c stores the feature value of the *current* maximal word matching the given pattern, accumulator d stores the feature value of a *potential* part of a maximal word matching the pattern, and accumulator r stores the aggregated *result* value for the feature values of all the maximal words matching the pattern that have been consumed so far. Figures 3.6b and 3.6c provide the substitution rules for every feature and aggregator.

Example 4. Consider again the transducer for the `Peak` = '<(<|=)*(>|=)*>' pattern in Figure 3.4. After applying the substitution rules in Figure 3.6a to that transducer, we obtain the generic automaton in Figure 3.7 for any $g_f_$PEAK constraint predicate. Finally, after applying the substitution rules in Figure 3.6b and Figure 3.6c for the feature $f =$ `max` and the aggregator $g =$ `Min` to that generic automaton, we obtain the automaton Figure 3.8 for the MIN_MAX_PEAK constraint predicate. □

The future work in [11] included simplifying the synthesised automata, as they often have more accumulators and more complex accumulator updates than manually designed ones: this may weaken the induced decompositions of the constraint predicate described by the synthesised automaton.

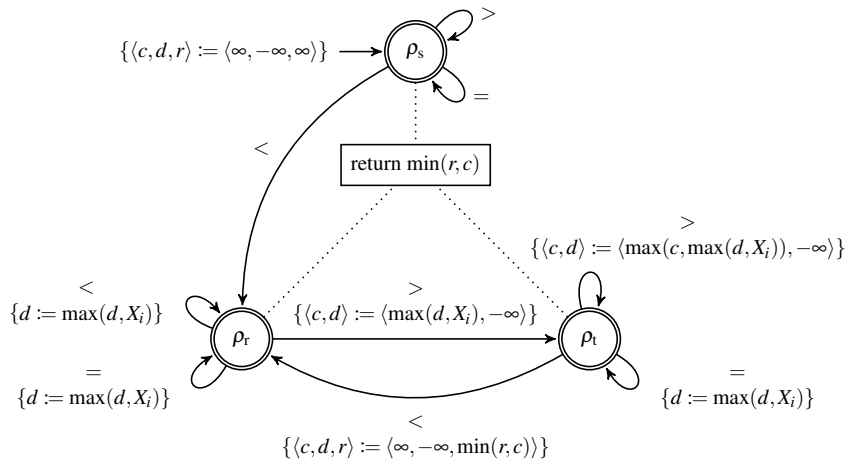


Figure 3.8. Automaton for MIN_MAX_PEAK

Example 5. Reconsider the automaton in Figure 3.8 for the MIN_MAX_PEAK constraint predicate from Example 4. Recall that this automaton has the accumulators c, d, and r, which respectively denote the feature value of the *current* peak, the feature value of a *potential* peak, and the aggregated *result* value for the feature values of all the peaks already encountered. Note that, for the particular case of MIN_MAX_PEAK, we know that when the transition from state ρ_r to state ρ_t is used, we are at the highest point of the current peak, and thus the accumulators c and d are redundant with accumulator r, and only r is needed. □

Rather than designing a procedural minimisation algorithm for automata with accumulators, in Paper IV we opted for capturing such procedural knowledge in a *declarative* and thus more easily reusable way: it suffices to specialise the decoration tables of [11] for some combinations of algebraic properties of pattern-feature-aggregator triples.

For example, for some combinations of pattern and feature, computing the feature value over a whole pattern occurrence gives the same result as computing it when a **found** symbol is produced, as seen in Example 5. This is the case of the feature `one` combined with any pattern, as well as the features `max` and `min` if the positions of the maximal and minimal values are uniquely determined by the pattern and coincide with producing a **found** symbol. This property is called *aggregate-once*, because the feature value of an occurrence depends only on the value of δ_f^i at the time a **found** symbol is produced, and so it can be aggregated immediately. Hence we need only one accumulator for aggregating.

initialisation	$r \coloneqq \mathrm{id}_g^f$
return	r
phase	
letters	update of r
$\mathbf{out}_{\mathrm{reset}}$	
$\mathbf{out}_{\mathrm{after}}$	$r \coloneqq \phi_g(r, \delta_f^i)$
$\mathbf{maybe}_{\mathrm{before}}$	
$\mathbf{maybe}_{\mathrm{after}}$	
found	
$\mathbf{found}_{\mathrm{end}}$	$r \coloneqq \phi_g(r, \delta_f^i)$
in	
out	

Figure 3.9. Optimised decoration table for aggregate-once constraint predicates. (Adaptation of a figure in [5].)

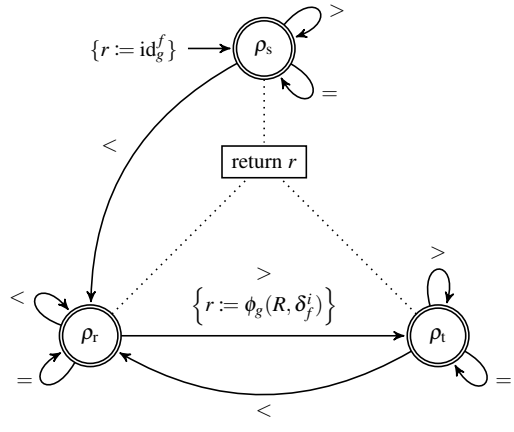


Figure 3.10. Simplified automaton for any g_f_PEAK, where f is either `one` or `max`.

Example 6. Consider again the transducer for the `Peak` = '<(<|=)*(>|=)*>' pattern in Figure 3.4. After applying the substitution rules of the decoration table in Figure 3.9 to that transducer, we obtain the generic automaton in Figure 3.10 for any g_f_PEAK constraint predicate where f is either `one` or `max`. Finally, after applying the substitution rules in Figure 3.6b and Figure 3.6c for the feature $f = \texttt{max}$ and the aggregator $g = \texttt{Min}$ to that generic automaton, we obtain the simplified automaton in Figure 3.11 for the MIN_MAX_PEAK constraint predicate. □

The simplification rules presented in Paper IV cover approximately 86% of the 266 time-series constraint predicates in [11].

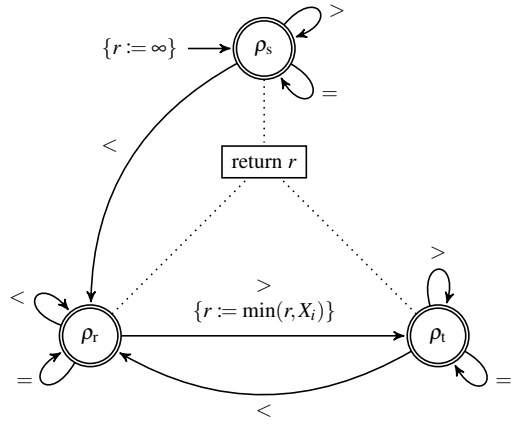


Figure 3.11. Simplified automaton for MIN_MAX_PEAK

4. Implied Constraints for Automaton-Induced Constraint Decompositions

> Ah, *that* is so hard that I fear I'm unable! For it holds it like glue.
>
> *Through the Looking-Glass*
> LEWIS CARROLL

We first introduce the reader to implied constraints (Section 4.1). We then define two orthogonal approaches to systematically generate implied constraints for automaton-induced constraint decompositions, namely linear implied constraints (Section 4.2) and glue constraints (Section 4.3).

4.1 Implied Constraints

An *implied constraint*, also called a *redundant constraint*, is a constraint that is logically implied by the constraints defining the problem [54]. It does not change the set of solutions, and hence it is logically redundant. The idea is that adding implied constraints to a model might reduce the time required to solve the problem due to additional propagation.

For example, consider the magic square problem. A *magic square* is an $n \times n$ square grid filled with the integers in the range $1, 2, \ldots, n^2$ such that each cell contains a distinct integer and that the sum of the integers in each row, column, and main diagonal is the same. That sum is called the *magic sum* of the magic square. In Figure 4.1 there is a magic square of order $n = 3$. Note that the magic square problem only requires the sum of each row, column, and main diagonal to be the same, as can be seen in the constraint model in Figure 4.2. Nevertheless, the magic sum can only take one particular value, which can be calculated before attempting to find a solution. We know that the square grid must be filled with the integers in the range $1, 2, \ldots, n^2$, that is, the sum of all the squares in the grid is:

$$1 + 2 + \cdots + n^2 = \frac{n^2(n^2+1)}{2}$$

Moreover, we know there are n rows (and n columns), so the sum of each row (and column) must be:

$$\frac{n^2(n^2+1)}{2n}$$

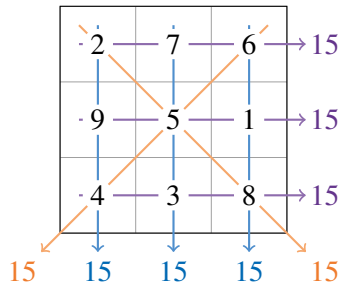


Figure 4.1. Magic square of order $n = 3$. The sum of the numbers in any row, column, or main diagonal is the same, namely 15.

```
magic_square_3([C11,C12,C13,C21,C22,C23,C31,C32,C33]) :-
    domain([C11,C12,C13,C21,C22,C23,C31,C32,C33], 1, 3*3),
    High is 3*3*3,
    Magic in 1..High,
    all_different([C11,C12,C13,C21,C22,C23,C31,C32,C33]),

    /* Row Constraints */
    C11 + C12 + C13 #= Magic,
    C21 + C22 + C23 #= Magic,
    C31 + C32 + C33 #= Magic,

    /* Column Constraints */
    C11 + C21 + C31 #= Magic,
    C12 + C22 + C32 #= Magic,
    C13 + C23 + C33 #= Magic,

    /* Diagonal Constraints */
    C11 + C22 + C33 #= Magic,
    C13 + C22 + C31 #= Magic.
```

Figure 4.2. Constraint model for the magic square problem of order $n = 3$ in SICStus Prolog [26] syntax.

Back to the magic square in Figure 4.1, the only possible value for the magic sum *magic* of a square of order $n = 3$ is 15. The exact value for the magic sum is determined by an implied constraint of the magic square problem, namely $magic = n(n^2 + 1)/2$. Without this implied constraint, partial assignments where the magic sum tentatively is not in the domain of *magic* can be formed, and eventually the search will have to backtrack when it cannot find a solution extending such a partial assignment.

In practice, it is not always simple to find good implied constraints, and they can be much more complex than equality constraints on a single variable. We now present two systematic approaches for deriving implied constraints for automaton-induced constraint decompositions, namely linear implied constraints (Section 4.2) and glue constraints (Section 4.3).

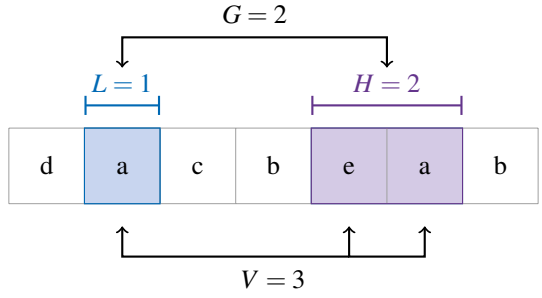


Figure 4.3. Visualisation of the instance GROUP($\langle$d,a,c,b,e,a,b$\rangle$,{a,e},2,3,2,1), where there are $G = 2$ groups of a total of $V = 3$ values from the set {a,e} in the sequence $\langle$d,a,c,b,e,a,b$\rangle$, the highest and lowest group sizes being $H = 2$ and $L = 1$.

4.2 Linear Implied Constraints

We now define the GROUP constraint predicate, to which we will be referring heavily throughout. We then give a motivating example, introducing our terminology and serving as running example throughout the rest of the chapter.

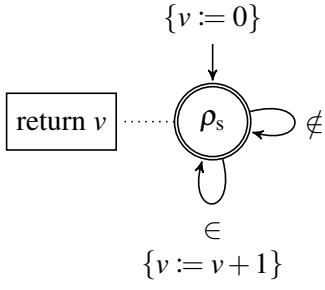
In a sequence, a *group* [10] is a maximal contiguous subsequence with values from a given set. The constraint GROUP(X,W,G,V,H,L) holds if there are G groups of a total of V values from the given set W in the possibly empty sequence X of variables, the highest and lowest group sizes being H and L respectively, with $H = 0 = L$ if $G = 0$. (Without loss of generality, we omit two parameters.)

For example, as seen in Figure 4.3, the instance GROUP$(X,\{\text{a},\text{e}\},2,3,2,1)$, where $X = \langle \text{d},\text{a},\text{c},\text{b},\text{e},\text{a},\text{b}\rangle$, holds since there are $G = 2$ groups of a total of $V = 3$ vowels from the set {a,e} in the sequence $X = \langle \text{d},\text{a},\text{c},\text{b},\text{e},\text{a},\text{b}\rangle$, namely the groups $\langle$a$\rangle$ and $\langle$e,a$\rangle$, the highest group size being $H = 2$ and the lowest group size being $L = 1$. The GROUP constraint predicate is very useful, for instance in staff rostering, where multiple counting constraints on the same sequence (the shift assignments of an employee over a planning horizon) are quite frequent.

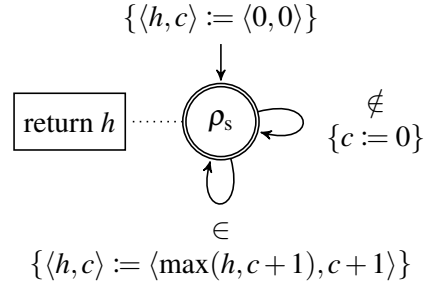
The GROUP constraint predicate has no known propagator. The reformulation of GROUP(X,W,G,V,H,L) in [10] by the conjunction GROUPG(X,W,G) $\wedge$ GROUPV(X,W,V) $\wedge$ GROUPH(X,W,H) $\wedge$ GROUPL(X,W,L) can be encoded using four AUTOMATON constraints involving automata with accumulators [13, 16] and the signature constraints:

$$(X_i \in W \;\Leftrightarrow\; S_i = \in) \wedge (X_i \notin W \;\Leftrightarrow\; S_i = \notin) \tag{4.1}$$

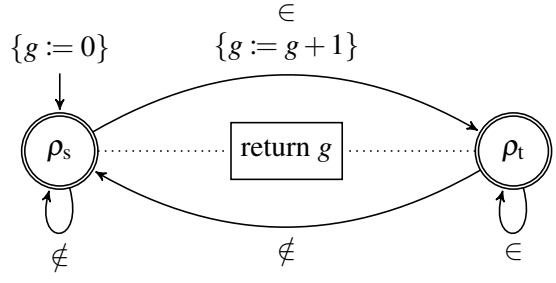
for all $i \in [1,n]$. See Figure 4.4 and Chapter 2 for details. In principle, we could compute the product automaton [36] of these four automata and just have a single AUTOMATON constraint: we do not do so here in order to be able to refer in our examples to the simpler automata. Note also that encoding any



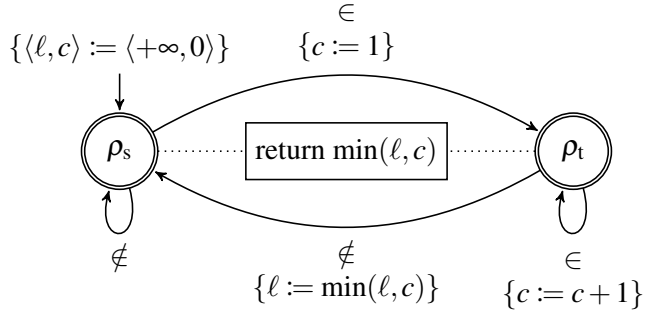
(a) Automaton $\mathcal{V}$ for GROUPV



(b) Automaton $\mathcal{H}$ for GROUPH



(c) Automaton $\mathcal{G}$ for GROUPG



(d) Automaton $\mathcal{L}$ for GROUPL

Figure 4.4. Automata with accumulators for the constraint predicates of the reformulation of the GROUP constraint.

of these four constraint predicates using REGULAR [43] requires designing a DFA whose size depends on the length of X.

We now show in Section 4.2.1 how to derive implied constraints from an imperative constraint checker, and then we show in Section 4.2.2 how to derive implied constraints directly from an automaton with accumulators.

Algorithm 1: Checker for the GROUPG(X, W, G) constraint

Data: Integer variable G, a set W, and a possibly empty sequence X of variables.

```
g := 0
q := ρs
i := 1
while i ≤ |X| do
    if X[i] ∈ W ∧ q = ρs then
        g := g + 1
        q := ρt
    else
        q := ρs
    i := i + 1
return G = g
```

4.2.1 Linear Implied Constraints from a Constraint Checker

Note that an automaton corresponds to a constraint checker. For example, a checker for a GROUPG(X, W, G) constraint is given in Algorithm 1 and can be obtained from the automaton in Figure 4.4c together with the signature constraints (4.1).

Informally, a *loop invariant* [27] for a given loop is a relation on the variables occurring in the loop: it should be true on entry into the loop and be guaranteed to remain true after every iteration of the loop. This means that on exit from a loop both its loop invariant and its loop termination condition can be guaranteed. For example, consider again Algorithm 1. The automatic loop invariant generator InvGen [33] derives, among others, the loop invariant:

$$g \leq |X| \tag{4.2}$$

In other words, there cannot be more groups than elements in a sequence X.

In order to transform loop invariants into implied constraints, we note that, for example, the invariant (4.2) is, by definition, satisfied at every iteration. At each iteration, some element of the sequence X is visited. Let decision variable G_i denote the value of the accumulator g after visiting i elements. In consequence, we write the invariant (4.2) as the following implied constraints:

$$G_i \leq |X| \tag{4.3}$$

for $0 \leq i \leq |X|$. Note that the bounds of the quantification correspond to the values of i before and after the loop. Moreover, note that to be able to use the implied constraints (4.3) we need the AUTOMATON constraint predicate to be implemented, in extension to how it is done in [13], by a decomposition that introduces variables representing the accumulators of the automaton after consuming each symbol of an argument sequence X of variables.

Algorithm 2: Checker for the GROUPG(X, W, G) constraint, keeping track of the two previous accumulator values

Data: Integer variable G, a set W, and a possibly empty sequence X of variables.

$g := 0$
$g_1 := 0$
$g_2 := 0$
$q := \rho_s$
$i := 1$
while $i \leq |X|$ **do**
  $g_2 := g_1$ // Keeping track of the previous value of g_1
  $g_1 := g$ // Keeping track of the previous value of g
  if $X[i] \in W \wedge q = \rho_s$ **then**
    $g := g + 1$
    $q := \rho_t$
  else
    $q := \rho_s$
  $i := i + 1$
return $G = g$

Consider again the automaton in Figure 4.4c: every path of two transitions increases the value of the accumulator g by at most 1. Let us extend the checker in Algorithm 1 in order to keep track of the two previous values of the accumulator g, yielding the checker in Algorithm 2. Accumulator g_1 denotes the value of accumulator g at the previous iteration, and accumulator g_2 denotes its value two iterations ago. We say that the *history length* is 2. From the checker in Algorithm 2, InvGen derives, among others, the loop invariant:

$$g_2 \leq g \leq g_2 + 1 \tag{4.4}$$

Note that g_1 does not appear in this invariant, as it is only used to keep track of the previous value of g. We translate the loop invariant (4.4) into the implied constraints:

$$G_{i-2} \leq G_i \leq G_{i-2} + 1 \tag{4.5}$$

for $1 < i \leq n$.

For more details about how to derive implied constraints from a checker, see Paper II. In particular, we also show how to modify constraint checkers in order to enable InvGen to derive disjunctive implied constraints. Moreover, we prove that implied constraints derived from constraint checkers improve propagation, even to the point of, for some constraint predicates, maintaining hyper-arc consistency.

Although the implied constraints obtained by deriving invariants from a checker corresponding to a given automaton are quite useful in practice, we

want both to make the process as automated as possible and to overcome the limitations of using InvGen. For example, we want to derive implied constraints (instead of invariants) directly from an automaton (instead of manually translating an automaton into a checker). Also, at the time InvGen only allows integer coefficients in $\{-1,0,1\}$. This limitation still exists, as a new version of InvGen has not been released so far.

4.2.2 Linear Implied Constraints from an Automaton

In Papers III–IV our approach to deriving constraints implied by the decomposition of an AUTOMATON$(\mathcal{A},S,R)$ constraint consists of three steps. First, using one half of Farkas' lemma and a template T for linear implied constraints, we set up a system N of non-linear constraints that model T being true at every state of the automaton $\mathcal{A}$ (Section 4.2.2). Second, we solve N, each solution providing an instantiation of T into a particular linear implied constraint (Section 4.2.2). Third, we eliminate uninteresting and propagation-redundant constraints from the derived set of implied constraints, and rank the remaining implied constraints by decreasing propagation strength (Section 4.2.2). We consider the user's final choice of implied constraints that are actually added to the decomposition to be a problem-specific rather than an automaton-specific task, so our tool focusses on *suggesting* implied constraints.

Implied Constraints: Template and Set-Up of the System N

In Paper III, we adapt the recipe of [50] for linear transition systems and develop ImpGen, our own tool for deriving implied constraints. A *linear transition system* does not have notions of consumption and acceptance of words, but is otherwise like an automaton with accumulators if every accumulator update of the latter is a linear expression on the accumulators. Everything that follows requires linearity, also of the implied constraints, so we now make that restriction.

One half of Farkas' lemma (e.g., [23]) says that a system of e linear inequalities $a_{i1}y_1+\cdots+a_{ik}y_k+b_i\geq 0$ over k real-valued variables y_j has another linear inequality $\alpha_1y_1+\cdots+\alpha_ky_k+\beta\geq 0$ over the same variables as a logical consequence if the latter is equal to a linear combination of the former, that is, if there exist e real numbers $\lambda_i\geq 0$ such that $\alpha_j=\sum_{i=1}^{e}\lambda_i a_{ij}$, for $1\leq j\leq k$, and $\beta\geq\sum_{i=1}^{e}\lambda_i b_i$. The following representation helps to see this:

$$\begin{array}{c|c} \lambda_1 & a_{11}y_1+\cdots+a_{1k}y_k+b_1\geq 0 \\ \vdots & \vdots \quad \vdots \quad \vdots \quad \vdots \\ \lambda_e & a_{e1}y_1+\cdots+a_{ek}y_k+b_e\geq 0 \\ \hline & \alpha_1y_1+\cdots+\alpha_ky_k+\beta\geq 0 \end{array}$$

If the i^{th} linear constraint is an equality, then the requirement $\lambda_i\geq 0$ is dropped.

The other half of Farkas' lemma gives a necessary and sufficient condition for a linear inequality to be a logical consequence of a system of linear inequalities. We do not need it as we deliberately do not aim at completeness and thus need not prove that a set of derived implied constraints is complete.

Let variable y_j denote the j^{th} accumulator of an automaton $\mathcal{A}$, with $1 \leq j \leq k$. Our template T for linear implied constraints for now is $\alpha_1 y_1 + \cdots + \alpha_k y_k + \beta \geq 0$, where the Greek letters denote the variables for which we will solve constraints. An instance of template T is true at every state of $\mathcal{A}$ if it is true at the start state of $\mathcal{A}$ and if its truth is preserved by every transition of $\mathcal{A}$. We now show how to encode this using Farkas' lemma in order to set up a system N of non-linear constraints that model T being true at every state of the automaton $\mathcal{A}$.

For the start state, we encode using Farkas' lemma that the point-wise initialisation equalities behind $\langle y_1, \ldots, y_k \rangle = I$ have T as a logical consequence, where I is the k-tuple of initial values of the accumulators of $\mathcal{A}$.

Example 7. Recall the automaton in Figure 4.4c for the GROUPG constraint predicate. There is only one accumulator, called g, which is initialised to 0. So the template for implied constraints is $\alpha_1 g + \beta \geq 0$ and it must be a logical consequence of the initialisation equality $g = 0$, that is:

$$\begin{array}{c|ll} \lambda_1 & g & = 0 \\ \hline & \alpha_1 g + \beta & \geq 0 \end{array}$$

Using Farkas' lemma, we get the constraints $\exists \lambda_1 : \alpha_1 = \lambda_1 \wedge \beta \geq 0$ for the system N. □

For each transition $(q, \langle y_1, \ldots, y_k \rangle) \xrightarrow{\sigma} (q', \langle y'_1, \ldots, y'_k \rangle)$ of $\mathcal{A}$, where each y'_j is a linear functional expression in terms of all the y_j, we encode using Farkas' lemma that template T has $T[y/y']$ as a template logical consequence, where $T[y/y']$ denotes T with every y_j substituted by y'_j. The resulting constraints are in general non-linear, as seen in the following example.

Example 8. Continuing from Example 7, first consider the transition from state ρ_s to state ρ_t, upon which accumulator g is incremented by 1. The desired template logical consequence $T[y/y']$ of T is $\alpha_1(g+1) + \beta \geq 0$, that is:

$$\begin{array}{c|lll} \lambda_2 & \alpha_1 \ g & +\beta & \geq 0 \\ \hline & \alpha_1(g+1) & +\beta & \geq 0 \end{array}$$

Using Farkas' lemma and rearranging, we get for each of these transitions the non-linear constraints $\exists \lambda_2 \geq 0 : \alpha_1 = \lambda_2 \alpha_1 \wedge \alpha_1 + \beta \geq \lambda_2 \beta$ for the system N.

Now consider all the remaining transitions, upon which the value of accumulator g does not change. The desired template logical consequence $T[y/y']$

of T is $\alpha_1 g + \beta \geq 0$, that is:

$$\begin{array}{c|ll} \lambda_3 & \alpha_1 g + \beta & \geq 0 \\ \hline & \alpha_1 g + \beta & \geq 0 \end{array}$$

Using Farkas' lemma and rearranging, we get the non-linear constraints $\exists \lambda_3 \geq 0 : \alpha_1 = \lambda_3 \alpha_1 \wedge \beta \geq \lambda_3 \beta$ for the system N. □

We now go beyond adapting the recipe of [50], by discussing four refinements of the ideas seen so far. The first three refinements are included in the first version of ImpGen in Paper III, while the last one is included only in its second version in Paper IV.

First, many implied constraints that provide extra propagation are expressed not only on the *current* values of the accumulators, but also on their values upon *previous* transitions, as already seen in Section 4.2.1. For example, only implied constraints that provide no extra propagation, such as $g \geq 0$, result from the solutions to the constraint system N we have set up in Examples 7 and 8: those examples were simple enough to explain all features of the procedure, but simpler than practical applications thereof. The following other example is enlightening.

Example 9. Consider again the automaton $\mathcal{G}$ in Figure 4.4c for the GROUPG constraint predicate, and its checker in Algorithm 1, which corresponds to $\mathcal{G}$. Recall that the checker in Algorithm 2 was obtained by extending the checker in Algorithm 1 with the idea of keeping track of the previous two values of accumulator g. The same idea can be used to extend directly the automaton in Figure 4.4c. Upon adding the initialisation $\langle g_2, g_1 \rangle := \langle 0, 0 \rangle$ to the start state, and adding the accumulator update $\langle g_2, g_1 \rangle := \langle g_1, g \rangle$ to each transition, we obtain the automaton in Figure 4.5. Applying Farkas' lemma to that automaton, we get the template $\alpha_1 g_2 + \alpha_2 g_1 + \alpha_3 g + \beta \geq 0$, so that, for instance, the implied constraint $g \leq g_2 + 1$ of (4.4) corresponds to $\alpha_1 = 1 = \beta \wedge \alpha_2 = 0 \wedge \alpha_3 = -1$. □

Our tool allows the user to indicate the history length h, so that appropriate accumulator terms are automatically added to the template T. Things scale at solving time when $h \cdot k$ grows, where k is the number of accumulators, since the process is off-line.

Second, the template T can be extended by adding a term $\alpha_{k+1} q$ with a variable q for the state at which the automaton is. This requires numbering the states. For example, this extension allows the tool to derive the implied constraint $g - g_1 \leq q$, where the state ρ_s of the automaton in Figure 4.4c (or Figure 4.5) has been numbered as 0 and state ρ_t as 1, meaning that g and g_1 are equal at the start state ρ_s and apart by at most one unit at state ρ_t. For example, this extension is actually necessary for deriving the implied constraint $g \leq g_2 + 1$ of Example 9, as we will see in Example 10. Our tool ImpGen al-

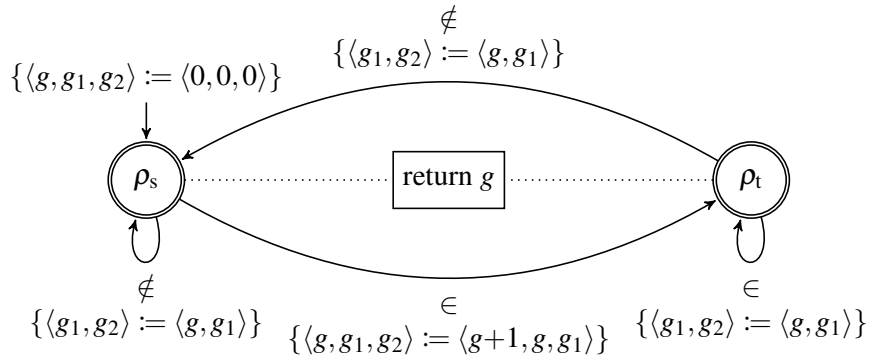


Figure 4.5. Automaton for GROUPG extended with a history length of 2

lows the user to switch on this option. Note that, when this option is switched on, the tool will automatically number the states, unless the user provides a particular way of numbering the states, in which case our tool will use the provided numbering.

Third, we have so far described how to derive implied constraints that are true at *every state* of $\mathcal{A}$. We can also make as many copies of the template as there are states in $\mathcal{A}$, so as to aim at deriving *state-specific* implied constraints. We must then use the appropriate template copies each time we apply Farkas' lemma for the start state or a transition. For example, an implied constraint specific to the start state ρ_s of the automaton in Figure 4.4c (or Figure 4.5) for the GROUPG predicate is $g_{i-1} = g_i$, derived from the implied constraints $g_{i-1} \leq g_i$ and $g_{i-1} \geq g_i$; we will see in Example 10 an encoding of this implied constraint when a term on the state variable q is added to the template. Our tool ImpGen allows the user to switch on this option.

Fourth, the template T can be extended by adding a term $\alpha_{k+2} i$ for the index i, that is the number of symbols consumed so far. For example, this extension allows the tool to derive the implied constraint $2g_i \leq i$. Our tool ImpGen allows the user to switch on this option.

Implied Constraints: Generation by Solving the System N

So far, we have shown how to set up a system N of non-linear constraints that are on the variables denoted by Greek letters in the template $\alpha_1 y_1 + \cdots + \alpha_k y_k + \alpha_{k+1} q + \alpha_{k+2} i + \beta \geq 0$ for implied constraints, but not on its accumulator variables y_j, state variable q, and index variable i. We now show how to solve N so that each solution provides an instantiation of the variables α_j, and β of the template, yielding an implied constraint on the accumulator variables y_j, state variable q, and index variable i.

Automata with accumulators are defined for integer accumulators, so it suffices to solve the non-linear constraint system N for *integer* values of the variables α_j, β, and λ_i: this is our second deliberate relaxation of completeness.

An implied constraint $\alpha_1 y_1 + \cdots + \alpha_k y_k + \alpha_{k+1} q + \alpha_{k+2} i + \beta \geq 0$ with rational values for the α_j, and β is equivalent to its multiplication by the least common multiplier of their denominators. Further, we reckon that for each variable a small finite integer interval centred on zero, such as from -5 to $+5$, suffices for finding many useful implied constraints: this is our last deliberate relaxation of completeness. Since our tool ImpGen is written in SICStus Prolog and reads automata in the SICStus Prolog syntax used in the Global Constraint Catalogue, we use the *finite*-domain CP solver of SICStus Prolog [26], although we could have used any integer programming solver: we solve upon linearising N by branching on values for the variables λ_i.

Inductive implied constraints [24] are those implied constraints that are provable by induction. Unfortunately, many implied constraints that provide extra propagation are not inductive, and so are not generated in one go, even if all options are switched on.

Example 10. Consider the implied constraint $g \leq g_2 + 1$ of (4.4). Let us again number state ρ_s of the automaton in Figure 4.4c as 0 and state ρ_t as 1. Deriving this implied constraint requires the prior knowledge that $g - g_1 \leq q$. It turns out that $g - g_1 \leq q$ actually *is* an inductive implied constraint, and so it is a solution to the system N. So let us add this implied constraint to the top side of each application of Farkas' lemma, with its own multiplier, and set up a second non-linear system N_2. For instance, the part of the system N_2 corresponding to the self-loop on state ρ_t in the automaton in Figure 4.4c, which does not modify the accumulator g, with a history length of 2, and including the state variable, is:

$$\begin{array}{c|l} \lambda_3 & \alpha_1 g + \alpha_2 g_1 + \alpha_3 g_2 + \alpha_{k+1} + \beta \quad \geq 0 \\ \lambda_4 & -g + \quad g_1 \qquad\qquad\qquad + 1 \quad \geq 0 \\ \hline & \alpha_1 g + \alpha_2 g \ + \alpha_3 g_1 + \alpha_{k+1} + \beta \quad \geq 0 \end{array}$$

Note that the variable q has been instantiated to the numerical name of the start state and the end state of the transition, both being ρ_t, namely 1. We get the non-linear constraints $\exists \lambda_3 \geq 0, \lambda_{31} \geq 0 : \alpha_1 + \alpha_2 = \lambda_3 \alpha_1 - \lambda_{31} \wedge \alpha_3 = \lambda_3 \alpha_2 + \lambda_{31} \wedge \lambda_3 \alpha_3 = 0 \wedge \alpha_{k+1} + \beta \geq \lambda_3(\alpha_{k+1} + \beta) + \lambda_{31}$ of N_2. It turns out that $g \leq g_2 + 1$ is now an inductive implied constraint, derived from a solution to N_2. □

Following the intuition of Example 10, after solving the system N, ImpGen can set up a new system N_2 by extending the system N with the implied constraints obtained by solving N: each implied constraint is added with its own multiplier to the top side of each application of Farkas' lemma. This process can be repeated by setting up a new system N_j by adding the implied constraints obtained from the system N_{j-1} to N, where $N = N_1$. Our method derives many implied constraints, each of which is inductive relative to those

derived before it. Note that the set of implied constraints derived from the system N_j is a superset of the implied constraints derived from the system N_{j-1}.

In practice, it is not necessary to add all the implied constraints derived from the system N_j into the new system N_{j+1}. It suffices to add an approximation of the set of generators, that is, we reduce the set of implied constraints by eliminating some of the inequalities that are a linear combination of other inequalities in the set of implied constraints. The solutions obtained are the same, but this greatly reduces the time and memory required to solve the system for large values of j.

Our tool ImpGen allows the user to indicate an upper bound u on the number of non-linear systems it will set up and solve; it will finish earlier if at fixpoint, that is if no new implied constraints are derived at some iteration. The whole derivation process is specific to an automaton but not to the constrained sequence, so it is off-line and can take an arbitrary amount of time. Our tool takes from seconds to days, depending on the parameters, especially u and the history length h.

Up to this point, we have shown how ImpGen handles automata where each accumulator update is a linear expression on accumulators. This includes increments and decrements by constant amounts (as in $c := c + 1$) or other accumulators (as in $c := c + \ell$), resets (as in $c := 0$), etc. This excludes updates via the 'max' and 'min' operators, for instance: this setting covers only 64 of the 266 time-series constraint predicates in the Time-Series Constraint Catalogue [5].

Towards handling *all* the time-series constraint predicates, we extend ImpGen to handle also conditional accumulator updates of the form $c := \text{if } \rho \text{ then } \phi \text{ else } \psi$, where ρ is a linear (in)equality and ϕ, ψ are linear expressions on accumulators: following an idea in [50], we extend the encoding of automaton transitions by allowing preconditions to be expressed. ImpGen now automatically first rewrites accumulator updates containing the binary 'min', 'max', or 'abs' operators into conditional updates. For example, the accumulator update on the arc from ρ_t to ρ_s in Figure 4.4d is rewritten as $\ell := \text{if } \ell > c \text{ then } c \text{ else } \ell$.

We also extend ImpGen to handle accumulator updates referring to the variables X_i and X_{i+1} when they are both linked to the S_i signature variable consumed on the transition: we do this by adding an accumulator x which always takes the value of the X_i variable, and add the signature constraint corresponding to the consumed signature symbol S_i to the top side of each application of Farkas' lemma, with its own multiplier.

The latest version of ImpGen, as described in Paper IV, covers *all* the 266 constraint predicates in the Time-Series Constraint Catalogue [5], as well as *all* the 56 constraint predicates with automaton-based descriptions in the Global Constraint Catalogue [10].

Implied Constraints: Redundancy Elimination and Selection
Some derived implied constraints are useless. For example, when all the α_j are zero, we can get an implied constraint like $5 \geq 0$, which is vacuously true and cannot improve propagation, but might slow it down. Other derived implied constraints are propagation-redundant. For example, the implied constraint $g_{i+1} \leq g_{i-1} + 1$ of Example 10 is redundant with $3g_{i+1} \leq 3g_{i-1} + 3$, and the former will give better propagation than the latter. As another example, the implied constraint $g_i + g_{i-1} \geq 0$ is redundant with the implied constraints $g_i \geq 0$ and $g_{i-1} \geq 0$ that result from the solutions to the constraint system N we have set up in Examples 7 and 8. Such redundancies stem from our not finding *generators* for the solutions to N.

Our tool ImpGen automatically eliminates useless and propagation-redundant constraints.

Finally, ImpGen *ranks* the implied constraints by decreasing estimated propagation strength when added to the automaton-induced CP decomposition: this is done based on a series of random instances. This enables *automated* selection via a top-m rule for a user-chosen parameter m.

Intuitively, the implied constraints derived by ImpGen can improve inference also for the MIP decomposition of the AUTOMATON constraint predicate in Paper IV because they are derived directly from an automaton and are not necessarily linear combinations of the linear inequalities in that decomposition [38]. Our experiments in Paper IV confirm that implied constraints that improve the propagation of the CP decomposition can also improve the inference of the MIP decomposition.

For more details and experimental results, see Papers II–IV.

4.3 Glue Constraints

Consider a constraint $\gamma(X,R)$ encoded using an AUTOMATON constraint, where variable R takes the same value for both the variable sequence X and its reverse X^{rev}. For example, consider again the instance $\text{GROUP}(X,\{\text{a},\text{e}\},2,3,2,1)$ from Figure 4.3, where $X = \langle \text{d},\text{a},\text{c},\text{b},\text{e},\text{a},\text{b}\rangle$. As can be seen in Figure 4.6, the instance $\text{GROUP}(X^{\text{rev}},\{\text{a},\text{e}\},2,3,2,1)$, where $X^{\text{rev}} = \langle \text{b},\text{a},\text{e},\text{b},\text{c},\text{a},\text{d}\rangle$ also holds since there are also $G = 2$ groups of a total of $V = 3$ vowels from the set $\{\text{a},\text{e}\}$ in the sequence X^{rev}, namely the groups $\langle \text{a},\text{e}\rangle$ and $\langle \text{a}\rangle$, the highest group size also being $H = 2$ and the lowest group size also being $L = 1$. We say that GROUP is its own *reverse*. This setting covers 45 of the 56 constraint predicates described using automata in the Global Constraint Catalogue [10] and 19 of the 20 time-series patterns in [11].

Given a partition of X into a prefix P and the corresponding suffix T, we derive in Paper V an implied constraint, called a *glue constraint*, shown to exist and be unique, between R, $\overrightarrow{R}$, and $\overleftarrow{R}$ when $\gamma(X,R)$, $\gamma(P,\overrightarrow{R})$, and $\gamma(T^{\text{rev}},\overleftarrow{R})$ hold.

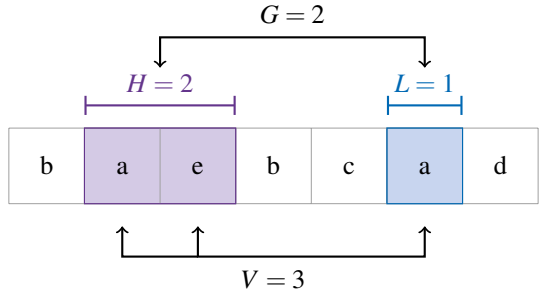


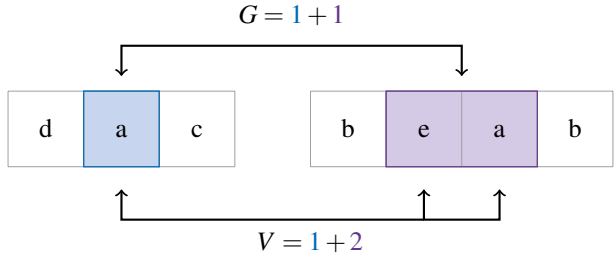
Figure 4.6. Visualisation of the instance GROUP($\langle$b,a,e,b,c,a,d$\rangle$,$\{$a,e$\}$,2,3,2,1), where there are $G = 2$ groups of a total of $V = 3$ values from the set $\{$a,e$\}$ both in the sequence $X = \langle$d,a,c,b,e,a,b$\rangle$ and its reverse $X^{\text{rev}} = \langle$b,a,e,b,c,a,d$\rangle$, the highest and lowest group sizes being $H = 2$ and $L = 1$.

A constraint $\gamma(V_1,\ldots,V_n)$ is a *total-function constraint* [12] if its variables V_i can be partitioned into two non-empty sets, X and R, such that for any assignment to the variables of X there is a *unique* assignment to the variables of R that satisfies γ. For example, the constraints GROUP(X,W,G,V,H,L), GROUPG(X,W,G), GROUPV(X,W,V), GROUPH(X,W,H), and GROUPL(X,W,L) are total-function constraints, where X and W uniquely determine G, V, H, and L. Also, signature constraints (see Chapter 2) are total-function constraints. However, ALLDIFFERENT$(X_1,\ldots,X_n)$ is not a total-function constraint, because, for the pairwise distinctness of the values of its variables, no subset of $\{X_1,\ldots,X_n\}$ uniquely determines its complement. The result set R may contain more than one variable, witness the SORT$(\langle X_1,\ldots,X_n\rangle,\langle Y_1,\ldots,Y_n\rangle)$ constraint [10], where the set $\{X_1,\ldots,X_n\}$ uniquely determines the variables of its sorted permutation $\langle Y_1,\ldots,Y_n\rangle$.

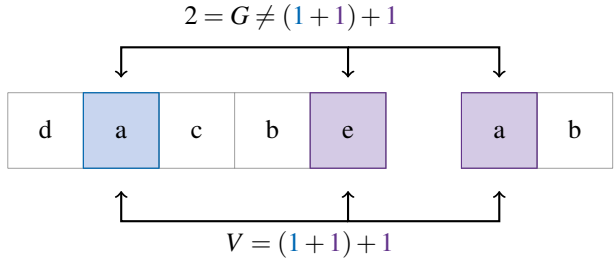
We write a constraint $\gamma(X,R)$ as $\gamma(X \to R)$ when the variables X functionally determine the variables R. We denote the reverse of a word or variable sequence w by w^{rev}. We now define our first core concept.

The *reverse* of a total-function constraint predicate γ is a total-function constraint predicate γ' such that both $\gamma(X \to R)$ and $\gamma'(X^{\text{rev}} \to R)$ hold, where X is a sequence of variables.

Example 11. The constraint predicates GROUP, GROUPG, GROUPV, GROUPH, and GROUPL are their own reverses. The constraint predicate LENGTHFIRSTSEQUENCE, where LENGTHFIRSTSEQUENCE(X,L) holds if L is the size of the *first* group of identical values within the sequence X of variables [10], does not have itself as reverse, but LENGTHLASTSEQUENCE, where LENGTHLASTSEQUENCE(X,L), holds if L is the size of the *last* group of identical values within X [10]. □



(a) Partition of $X = \langle \mathrm{d}, \mathrm{a}, \mathrm{c}, \mathrm{b}, \mathrm{e}, \mathrm{a}, \mathrm{b} \rangle$ into $P = \langle \mathrm{d}, \mathrm{a} \rangle$ and $T = \langle \mathrm{c}, \mathrm{b}, \mathrm{e}, \mathrm{a}, \mathrm{b} \rangle$.



(b) Partition of $X = \langle \mathrm{d}, \mathrm{a}, \mathrm{c}, \mathrm{b}, \mathrm{e}, \mathrm{a}, \mathrm{b} \rangle$ into $P = \langle \mathrm{d}, \mathrm{a}, \mathrm{c}, \mathrm{b}, \mathrm{e} \rangle$ and $T = \langle \mathrm{a}, \mathrm{b} \rangle$.

Figure 4.7. Visualisation of the instance $\textsc{Group}(\langle \mathrm{d}, \mathrm{a}, \mathrm{c}, \mathrm{b}, \mathrm{e}, \mathrm{a}, \mathrm{b} \rangle, \{\mathrm{a}, \mathrm{e}\}, 2, 3, 2, 1)$, where there are $G = 2$ groups of a total of $V = 3$ values from the set $\{\mathrm{a}, \mathrm{e}\}$ in the sequence $\langle \mathrm{d}, \mathrm{a}, \mathrm{c}, \mathrm{b}, \mathrm{e}, \mathrm{a}, \mathrm{b} \rangle$, the highest and lowest group sizes being $H = 2$ and $L = 1$.

We can now explain the key insight behind glue constraints using an example.

Example 12. Consider again the instance $\textsc{GroupV}(X, \{\mathrm{a}, \mathrm{e}\}, 3)$, where $X = \langle \mathrm{d}, \mathrm{a}, \mathrm{c}, \mathrm{b}, \mathrm{e}, \mathrm{a}, \mathrm{b} \rangle$. If we split the sequence $X = \langle X_1, \ldots, X_n \rangle$ into a non-empty prefix $P = \langle X_1, \ldots, X_i \rangle$ and the corresponding non-empty suffix $T = \langle X_{i+1}, \ldots, X_n \rangle$, with $1 \leq i < n$, then observe that the numbers V, $\overrightarrow{V}$, and $\overleftarrow{V}$ of group values respectively in the entire sequence X, the prefix P, and the reverse suffix T^{rev} are related by the glue constraint $V = \overrightarrow{V} + \overleftarrow{V}$. This constraint is implied by the conjunction of $\textsc{GroupV}(X, W, V)$, $\textsc{GroupV}(P, W, \overrightarrow{V})$, and $\textsc{GroupV}(T^{\mathrm{rev}}, W, \overleftarrow{V})$. In Figure 4.7 we show two partitions of X and the corresponding values for V, $\overrightarrow{V}$, and $\overleftarrow{V}$. Note that, while $\textsc{GroupV}(T^{\mathrm{rev}}, W, \overleftarrow{V})$ could be replaced above by $\textsc{GroupV}(T, W, \overleftarrow{V})$, this will be seen to be impossible in general with our approach, where the third implied constraint must be on the *reverse suffix*, not on the suffix itself. Consider again the instance $\textsc{GroupG}(X, \{\mathrm{a}, \mathrm{e}\}, 2)$, where $X = \langle \mathrm{d}, \mathrm{a}, \mathrm{c}, \mathrm{b}, \mathrm{e}, \mathrm{a}, \mathrm{b} \rangle$, and the two partitions of X in Figure 4.7. Observe that the numbers G, $\overrightarrow{G}$, and $\overleftarrow{G}$ of groups respectively in the entire sequence, the prefix, and the reverse suffix are not always related by the equality $G = \overrightarrow{G} + \overleftarrow{G}$. Indeed the total number of groups

	ρ_s
ρ_s	$V = \overrightarrow{v} + \overleftarrow{v}$

(a) Glue constraint for GROUPV

	ρ_s
ρ_s	$H = \max(\overrightarrow{h}, \overrightarrow{c} + \overleftarrow{c}, \overleftarrow{h})$

(b) Glue constraint for GROUPH

	ρ_s	ρ_t
ρ_s	$G = \overrightarrow{g} + \overleftarrow{g}$	$G = \overrightarrow{g} + \overleftarrow{g}$
ρ_t	$G = \overrightarrow{g} + \overleftarrow{g}$	$G = \overrightarrow{g} - 1 + \overleftarrow{g}$

(c) Glue constraint for GROUPG

	ρ_s	ρ_t
ρ_s	$L = \min(\overrightarrow{\ell}, \overrightarrow{c} + \overleftarrow{c}, \overleftarrow{\ell})$	$L = \min(\overrightarrow{\ell}, \overleftarrow{c}, \overleftarrow{\ell})$
ρ_t	$L = \min(\overrightarrow{\ell}, \overrightarrow{c}, \overleftarrow{\ell})$	$L = \min(\overrightarrow{\ell}, \overrightarrow{c} + \overleftarrow{c}, \overleftarrow{\ell})$

(d) Glue constraint for GROUPL

Figure 4.8. Glue constraints for the constraints of the reformulation of GROUP: a row index refers to the state of the corresponding automaton in Figure 4.4 reached by the prefix, and a column index refers to the state reached by the corresponding reverse suffix; recall that each of the four constraint predicates is its own reverse.

depends on whether or not the partition of X into P and T splits a group. We then want to post a GROUPG constraint together with the glue constraint for every split of the sequence V into a possibly empty prefix P and the corresponding possibly empty suffix T. Glue constraints for all the integer variables of GROUP are given in Figure 4.8. □

We need the AUTOMATON constraint predicate to be implemented, in extension to how it is done in [13], by a decomposition that introduces variables representing not only the accumulators but also the state of the argument automaton $\mathcal{A} = \langle Q, \Gamma, \delta, \rho_0, C_0, Q_a, \alpha \rangle$ after reading each symbol of an argument sequence X of variables. Upon reading the entire X, we have that $\widehat{\delta}(\langle \rho_0, I \rangle, X)$ is a tuple $\langle q, R \rangle$, where variable q represents the reached state of $\mathcal{A}$, and R is an array of k variables representing the k obtained accumulator values of $\mathcal{A}$. In Paper V we show that the result of $\mathcal{A}$ on a sequence X can be computed from only these state and accumulator variables, as they encode all information on X. We exploit this insight by constructing a glue constraint, which is unique and correctly constrains the result of $\mathcal{A}$ on a word X by combining the state and accumulator variables reached by a prefix P of X and the reverse of its corresponding suffix T.

Consider a total-function constraint $\gamma(X \to R)$, for which an automaton $\mathcal{A} = \langle Q, \Gamma, \delta, \phi, C_0, \rho_0, Q_a, \alpha \rangle$ consumes a signature S linked with the

sequence X by signature constraints, such that the variable R must be the result returned by $\mathcal{A}$ on S. Hence $\gamma(X \to R)$ can be encoded by $\text{AUTOMATON}(\mathcal{A}, S, R)$ and the signature constraints. Consider a split of S into the concatenation of the possibly empty prefix P and the corresponding possibly empty suffix T, that is $S = PT$, with $R = \alpha(\widehat{\delta}(\langle \rho_0, \mathcal{C}_0 \rangle, PT))$. Suppose, in addition to $\text{AUTOMATON}(\mathcal{A}, PT, R)$, we post the constraint $\text{AUTOMATON}(\mathcal{A}, P, \overrightarrow{R})$ on the prefix P, as well as the constraint $\text{AUTOMATON}(\mathcal{A}', T^{\text{rev}}, \overleftarrow{R})$ on the corresponding reverse suffix T^{rev}, where $\mathcal{A}' = \langle Q', \Gamma', \delta', \phi', \mathcal{C}'_0, \rho'_0, Q'_{\text{a}}, \alpha' \rangle$ is an automaton for the reverse of γ. Let $\widehat{\delta}(\langle \rho_0, \mathcal{C}_0 \rangle, P) = \langle \overrightarrow{Q}, \overrightarrow{\mathcal{C}} \rangle$ and $\widehat{\delta}'(\langle \rho'_0, \mathcal{C}'_0 \rangle, T^{\text{rev}}) = \langle \overleftarrow{Q}, \overleftarrow{\mathcal{C}} \rangle$. The relationship between the variables R, $\overrightarrow{\mathcal{C}}$, $\overleftarrow{\mathcal{C}}$, $\overrightarrow{Q}$, and $\overleftarrow{Q}$ gives rise to an implied constraint, called the *glue constraint*:

$$R = \text{GLUE}(\langle \overrightarrow{Q}, \overrightarrow{\mathcal{C}} \rangle, \langle \overleftarrow{Q}, \overleftarrow{\mathcal{C}} \rangle) \tag{4.6}$$

where GLUE is automaton-specific, and *independent* of P and T: it only depends on the reached tuples of state and accumulator variables.

We then post an AUTOMATON constraint and the glue constraint for every split of the signature S into a possibly empty prefix P and the corresponding possibly empty suffix T.

In Paper V we show that the glue constraints improves the propagation of the given automaton-induced decomposition. Note that the extra pruning achieved by the linear implied constraints in Section 4.2 is incomparable with that achieved by the glue constraints, as can be seen in the experimental results of Paper VI. We also show in Paper V the usefulness of glue constraints in the context of constraint-based local search [55], where glue constraints enable constant-time move probing.

In Paper V we show how to automatically derive the glue constraint for a useful class of constraints, namely those that can be encoded using an automaton with a single accumulator, which is initialised to zero at the start state and increased by a non-negative quantity at each transition as well as by the acceptance function. For instance, this class includes the GROUPV constraint predicate encoded using the automaton in Figure 4.4a and the GROUPG constraint predicate encoded using the automaton in Figure 4.4c. This setting excludes the GROUPH constraint predicate encoded using the automaton in Figure 4.4b and the GROUPL constraint predicate encoded using the automaton in Figure 4.4d. This setting covers 16 of the 56 constraint predicates described using automata in the Global Constraint Catalogue [10], and about one third of the constraint predicates in the Time-Series Constraint Catalogue [5]. The derivation of the glue constraint in Paper V for all the other constraint predicates outside of this class was ad hoc.

The *reverse of a pattern* is for the topological alphabet $\Sigma = \{\text{`<'}, \text{`='}, \text{`>'}\}$ a pattern that has the reverse order of its symbols and has all occurrences of the symbol '<' flipped into '>' and vice versa. In Paper VI we introduce *para-*

metric glue constraints and show that they can be derived *automatically* for time-series constraint predicates, which were introduced in [11]. A parametric glue constraint for a time-series constraint specified by $\langle \pi, f, g \rangle$ is specific to the pattern π, but generic in terms of the feature f and the aggregator g. We derive parametric glue constraints for the 19 of 20 patterns defined in [11] that also have a reverse pattern in [11]. Moreover, of the 56 constraint predicates in the Global Constraint Catalogue [10] that have their reverse constraint predicates in [10], some of them correspond to time-series constraint predicates as defined in Chapter 3. For example the GROUPH constraint predicate corresponds to a time-series constraint predicate where the aggregator g is `Max`, the feature f is `width`, and the accumulators $\langle c, r \rangle$ for time-series automata have the same semantics as the accumulators $\langle c, h \rangle$ of the automaton in Figure 4.4b. The accumulator d for time-series automata is unused because the concept of potential group does not exist, and thus it is eliminated. Note that the constraint GROUPL(X, W, L) is not a time-series constraint as defined in Chapter 3 because it is especially designed to hold if $L = 0$ when there are no groups of W in the variable sequence X. The equivalent time-series constraint would hold if $L = +\infty$ when there are no groups in the variable sequence X.

In Paper I we introduce an algorithm for automatically generating a seed transducer for a given pattern. Using that algorithm together with the automaton synthesiser in [11], it is now possible to generate automata for the reverse of any g_f_INFLEXION constraint predicate from the reverse of the INFLEXION pattern, which are the only constraint predicates in [11] without a reverse. In turn, this makes it possible to derive the parametric glue constraint for the INFLEXION pattern, thus covering *all* the 20 patterns in [11].

5. Summaries of Papers

> "Why, sometimes I've believed as many as six impossible things before breakfast."
>
> *Through the Looking Glass*
> LEWIS CARROLL

We summarise briefly the contents of Papers I–VI.

I Automatic Generation of Descriptions of Time-Series Constraints

In: M. Virvou (editor), ICTAI 2017. IEEE Computer Society, 2017 (in press).

We show how to apply the synthesiser of automaton-based time-series constraint predicate descriptions in [11] directly to high-level patterns, instead of low-level transducers. We do so in two steps: first, we characterise the large class of patterns, called *recognisable patterns*, which are the only patterns that can be handled by the synthesiser in [11], making it possible to decide when the synthesiser is applicable; and second, we give an algorithm for automatically generating a transducer from a recognisable pattern.

Our tool generates exactly the 20 handcrafted transducers of [11]. Our tool also generates the handcrafted transducers for the patterns `BumpOnDecreasingSequence` and `DipOnIncreasingSequence`, thus covering all 22 patterns in the Time-Series Constraint Catalogue [5]. By proving that our tool only generates well-formed transducers, we also prove that the transducers in [11] and [5] are well-formed.

This work contributes, in the context of time-series constraint predicates, to the systematic reconstruction of the Global Constraint Catalogue [10] that was previously advocated [15]. Note that, in this paper, we restrict the alphabet of the patterns to the topological alphabet $\Sigma = \{\text{`<'}, \text{`='}, \text{`>'}\}$, which is also the only input alphabet considered in [11], in order to ease the notation. However, both the characterisation of recognisable patterns and the algorithm for generating transducers from such patterns are independent of the input alphabet.

II Generation of Implied Constraints for Automaton-Induced Decompositions

In: A. Brodsky (editor), ICTAI 2013, pages 1076–1083. IEEE Computer Society, 2013.

We show how to improve propagation of automaton-induced constraint decompositions by means of systematically-derived implied constraints. Such implied constraints are derived off-line and are specific to the automaton but instance independent. From an analysis of constraint checkers corresponding to automata with accumulators, we identify loop invariants by means of an automatic invariant generator called InvGen [33], as well as by more involved loop-invariant generation techniques. After generating the loop invariants, we translate them into implied constraints and add them to the corresponding automaton-induced decomposition to improve propagation.

A key problem in automatic loop invariant generation is the inference of *disjunctive invariants*, which contain at least one disjunction and generally arise from the existence of conditionals in the loop body. In order to simplify the generation of disjunctive loop invariants, we manually use a technique proposed in [53] to decompose a constraint checker into a semantically equivalent sequence of loops, each of which has only conjunctive invariants. As a result, the disjunction of the conjunctive invariants of the loops is a disjunctive invariant of the original constraint checker. Note that InvGen can be used to generate invariants for each of the individual loops.

With the aim of generating loop invariants linking the values of the accumulators on a given iteration to the values of the accumulators in previous iterations, we systematically extend the constraint checkers to keep track of the previous values of the variables. The extension consists of adding new variables to the checker in such a way that the new variables store the values of the accumulators one or more iterations before.

We demonstrate the use of these invariant-generation techniques via the JTHNONZEROPOS[1] and GROUPG[2] constraint predicates. First, we show that the presence of these implied constraints does not increase the time or space complexity of computing the common fixpoint of the propagators of the decomposition. Second, we prove that in the presence of these implied constraints, domain consistency[3] is maintained on the automaton-induced decomposition of a JTHNONZEROPOS$(\mathcal{V},J,P)$ constraint. This example demonstrates that these implied constraints can be quite powerful. Finally, we also

[1]The JTHNONZEROPOS constraint predicate is called ITH_POS_DIFFERENT_FROM_ZERO in the Global Constraint Catalogue [10].

[2]The GROUPG constraint predicate is called NGROUP in Papers II and III, and it is a subconstraint predicate of the GROUP constraint predicate in the Global Constraint Catalogue [10].

[3]Domain consistency is called hyper-arc consistency (HAC) in Paper II

experimentally show that these implied constraints often improve propagation, incurring little or no time overhead.

III Implied Constraints for AUTOMATON Constraints

In: G. Gottlob, G. Sutcliffe, and A. Voronkov (editors), GCAI 2015. EasyChair Proceedings in Computing, volume 36, pages 113–126, 2015.

We continue the work of Paper II towards improving propagation of automaton-induced constraint decompositions. Here we present the first version of ImpGen (the second version of ImpGen is presented in Paper IV): a *fully automated* tool that reads an automaton with accumulators[4] in the SICStus Prolog syntax [26], and selects, in an off-line process, linear constraints that are implied by the automaton-induced constraint decomposition. We here focus on automata with accumulators where every accumulator update is a linear expression on the accumulators. This includes increments and decrements by constant amounts (as in $c := c + 1$) or by other accumulators (as in $c := c + \ell$), resets (as in $c := 0$), etc, but excludes updates using the 'max' and 'min' operators, for instance. This setting covers a large percentage of those in the Global Constraint Catalogue [10], but only a small percentage of those in the Time-Series Constraint Catalogue [5].

For better control compared to using an off-the-shelf loop-invariant generator (such as InvGen [33] in Paper II), we design our own tool to derive implied constraints, which works *directly* on the automaton (rather than on its manual translation into an imperative checker) and *directly* derives implied constraints (rather than generating loop invariants that have to be translated into implied constraints). Using one half of Farkas' lemma (e.g., [23]) and a linear template for implied constraints, we set up a system of non-linear constraints that model the template being true at every state of the automaton.

Our tool ImpGen goes beyond adapting Farkas' lemma for automata, by having extra parameters for controlling the quality of its results. First, as seen in Paper II, many implied constraints that provide extra propagation are expressed not only on the current values of the accumulators, but also on their values upon previous transitions. ImpGen allows the user to indicate the history length, so that appropriate accumulator-induced terms are added to the template. Second, the template can also be extended by adding a term for a state variable denoting the state at which the automaton is. This requires numbering the states: either the user provides a numerical value for each state, or the tool automatically assigns a numerical value to each state. Third, ImpGen can also generate state-specific implied constraints. Finally, ImpGen elim-

[4] Automata with accumulators are called counter automata in Paper II, and memory-DFAs in Paper III and Paper V.

inates uninteresting and propagation-redundant constraints from the derived set of implied constraints, so as to ease the user's choice of implied constraints that are actually added to the decomposition. The actual choice of which implied constraints are added to the decomposition is *problem*-specific and beyond the scope of our tool.

We experimentally show that a suitable choice, made by the user, among the tool-selected implied constraints can considerably improve solving time and propagation, both on an automaton-induced decomposition in isolation and on entire constraint problems containing the decomposition. In some cases we observe a big improvement in solving time compared to the decomposition alone, despite the overhead in running more propagators. For example, the automaton-induced decomposition of INFLEXION[5] [10] with the implied constraints is always faster than the decomposition alone, is about 35% faster on average, and has almost no failures on average. We obtain similar results for other automaton-induced constraint decompositions.

IV Time-Series Constraints: Improvements and Application in CP and MIP Contexts

In: C.-G. Quimper (editor), CP-AI-OR 2016. Lecture Notes in Computer Science, volume 9676, pages 18–34. Springer, 2016.

We describe and evaluate extensions to three different techniques related to automaton-induced constraint decompositions.

First, we improve the synthesiser of automaton-based constraint predicate descriptions in [11] for a large family of time-series constraints, so as to synthesise automata with fewer accumulators and simpler accumulator updates, improving both propagation time and memory usage. Note that the generated automata have non-linear accumulator updates (as in $c := \max(c,d)$).

Second, we decompose a constraint described by an automaton *with* accumulators into a linear-sized conjunction of linear inequalities, for use by a mixed-integer programming (MIP) solver.

Third, in order to cover the entire family of time-series constraint predicates of [11] and the Time-Series Constraint Catalogue [5], we extend ImpGen, our implied-constraint generator ImpGen of Paper III, so that it can handle non-linear accumulator updates. Moreover, we extend ImpGen to include a rank of the derived implied constraints by decreasing propagation strength, thereby easing the human selection of which implied constraints actually to use.

[5]The INFLEXION constraint predicate in [10] is called NBINFLEXION in the Time-Series Constraint Catalogue [5], where NB stands for 'number of' and corresponds to the feature `one` together with the aggregator `Sum`.

Our results show that the newly synthesised automata for time-series constraint predicates outperform the automata of [11], for both the automaton-induced CP and MIP decompositions, and that the newly derived implied constraints boost the inference, again for both the CP and MIP decompositions, as well as for both the automata of [11] and the newly synthesised ones.

We also evaluate CP and MIP solvers on a prototypical application modelled using time-series constraints. On their own, both the old and the new automata for time-series constraints perform quite poorly. Nevertheless, adding the top two ImpGen-derived implied constraints to the new automata allows us to find solutions for all problem instances.

V Linking Prefixes and Suffixes for Constraints Encoded Using Automata with Accumulators

In: B. O'Sullivan (editor), CP 2014. Lecture Notes in Computer Science, volume 8656, pages 142–157. Springer, 2014.

We consider constraints $\gamma(X,R)$ over a sequence of variables X, such that γ is described using an automaton with accumulators and functionally determines the result variable R, and where R is the same for both X and its reverse X^{rev}. This class of constraints is called *reversible constraints* and covers 45 of the 56 constraints that are described using automata with accumulators in the Global Constraint Catalogue [10], and 19 of the 22 patterns in the Time-Series Constraint Catalogue [5].[6]

Given a partition of X into a prefix P and the corresponding suffix T, we define an implied constraint, called *glue constraint*, shown to exist and be unique, between R, $\overrightarrow{R}$, and $\overleftarrow{R}$ when $\gamma(X,R)$, $\gamma(P,\overrightarrow{R})$, and $\gamma(T^{\text{rev}},\overleftarrow{R})$ hold.

After formalising reversible constraints and glue constraints, we first show how to hand-craft glue constraints for any reversible constraint described using an automaton with accumulators. Second, we show how to automatically derive glue constraints for a useful class of reversible constraints, namely those that can be encoded using an automaton with a single accumulator, which is initialised to zero at the start state and increased by a non-negative quantity at each transition as well as by the acceptance function. Third, we show how to further improve propagation in the presence of implied constraints on the result variables of multiple total-function constraints on the same sequence X of variables. The general idea is to take such an implied constraint and add it, together with the glue constraints, for the result variables of all prefixes (and corresponding suffixes) of X.

[6] The class of reversible constraints covers 19 of the 20 patterns in [11].

We evaluate the effectiveness and efficiency of the glue constraints, finding that using glue constraints and implied constraints on the results of all prefixes and suffixes greatly improves propagation. In particular, we use the glue constraint to compute, in constant time, the violation cost of $\gamma(X,R)$ when probing an assignment move in local search.

VI Systematic Derivation of Bounds and Glue Constraints for Time-Series Constraints

In: M. Rueher (editor), CP 2016. Lecture Notes in Computer Science, volume 9892, pages 13–29. Springer, 2016.

We give parametric ways of systematically deriving glue constraints, which are a particular kind of implied constraints introduced in Paper V, as well as aggregation bounds that can be added to the automaton-induced decomposition of the time-series constraints in [11] and in the Time-Series Constraint Catalogue [5].

First, we show how to systematically derive glue constraints, parametrised by the aggregator and feature functions, for any regular expression.

Second, we give a methodology for systematically deriving bounds, parametrised by a regular expression, on the result variable, for any pair of aggregator g and feature f, and then we demonstrate our methodology on the case when $g = \texttt{Max}$ and $f = \texttt{min}$.

Finally, we experimentally show the beneficial propagation impact of the derived glue constraints and bounds, both alone and together. In particular, we show that bounds and glue constraints on their own bring good reductions of the search space, and that their combinations can greatly improve propagation. For example, the combined approach in most cases reduces the number of backtracks by more than three orders of magnitude.

6. Related Work

"Curiouser and curiouser!" Cried Alice

Alice's Adventures in Wonderland
LEWIS CARROLL

We review related work along the main topics of this dissertation: automaton-based constraint predicate descriptions, automatically generating propagators or decompositions for constraint predicates, improving propagation of automaton-induced constraint decompositions, and time-series constraints.

6.1 Constraints Over Formal Languages

In Chapter 2 we explain regular languages and how to describe a constraint predicate by means of an automaton. In particular, we introduce the reader to automaton-based descriptions of constraint predicates using predicate automata with accumulators. In this section we outline other approaches to describing constraints of membership of a variable sequence in a formal language.

The REGULAR Constraint Predicate

The REGULAR constraint predicate is discussed at length in Section 2.2. We widen our approach to automata with accumulators because they allow the capture of non-regular languages and yield (even for regular languages) automata that are often much smaller and, more importantly, instance-independent.

The CONTEXTFREE Constraint Predicate

A *context-free grammar* (CFG) encompasses a set of production rules that describe all possible strings in a given formal language. The production rules are replacements. For example, the production rule $E \Rightarrow E + E$ states that an expression (denoted by the non-terminal E) can be formed by taking any two expressions and connecting them by a plus sign. Given a context-free grammar $G = \langle \Gamma, N, P, S_0 \rangle$ with alphabet Γ, non-terminals N, production rules P, start

symbol $S_0 \in N$, and a word $w \in \Gamma^*$, the problem is to answer the question whether w is in the language of G or not.

In [44, 52, 34] the constraint predicate CONTEXTFREE is introduced, together with a propagator that maintains HAC.

6.2 Other Types of Automata

In Chapter 2 we present automata with accumulators. Here we review other extensions to classical automata.

Weighted Automata

Weighted finite automata [40] are classical possibly non-deterministic finite automata in which the transitions carry weights [28]. These weights may model, for example, the cost of executing a transition, the amount of resources needed for this, or the probability of its successful execution. The behaviour of a weighted finite automaton can then be considered as the function associating with each word the weight of its execution. For instance, if an automaton with accumulators $\mathcal{A} = \langle Q, \Gamma, \delta, \rho_0, C_0, Q_a, \alpha \rangle$ has a single accumulator, which is initialised to zero (hence $C_0 = 0$) at the start state ρ_0 and increased by a non-negative quantity at each transition, and if the acceptance function α returns that accumulator increased by a non-negative quantity, then $\mathcal{A}$ is a weighted automaton over the tropical semiring over the integers, and the algorithms implemented in [41] can be used. Among the 45 of 56 constraint predicates of the Global Constraint Catalogue [10] covered in Paper V, there are 16 described by weighted automata, such as GROUPG and GROUPV, but not GROUPH and GROUPL, whose accumulator updates use the 'max' and 'min' operators. Recall that in Paper V we automatically derive glue constraints for this particular class of automata.

The reverse of a weighted automaton is a weighted *non*-deterministic finite automaton (NFA), possibly with ε-transitions; if (but not only if) a weighted NFA satisfies the twins property [2], then it can be finitely determinised into a weighted automaton (the algorithms implemented in [41] can be used for reversal, ε-removal, determinisation, and minimisation). The twins property can be checked in time polynomial in the size of the weighted NFA [2], but at present no characterisation is known for a weighted automaton to have a reverse satisfying the twins property; this is not an issue though, as the decomposition of the AUTOMATON constraint predicate works unchanged for NFAs with accumulators [13].

Register Automata

Register finite automata [18, 37] are classical possibly non-deterministic finite automata in which the input alphabet is provided as a set of parameterised actions, states have registers that can store data values, and transitions are labelled with parameterised actions, guards over parameters and registers, and assignments to the registers. Register automata are similar to automata with accumulators except that the registers can only be assigned data values or other registers, while accumulators can be assigned the result of arithmetic operations performed on other accumulators. Moreover, register automata also restrict possible relations between data values, for example, only equality comparisons. Automata with accumulators do not have such restrictions.

6.3 Quantitative Properties of Data Streams

Quantitative regular expressions (QREs) [3, 4] are a high-level programming abstraction for specifying numerical queries over data streams. Like automata with accumulators, QREs allow complex numerical updates using operations such as min, max, sum, difference, and averaging. Unlike our focus on the practical aspects of using automata with accumulators to describe constraint predicates, the key technical challenge in [3] is defining an update language that is expressive but also theoretically guarantees well-typed programs. They prove that the expressiveness of QREs is the same as that of streaming composition of regular functions, that is monadic second-order definable string-to-term transformations, leading to a robust foundation for understanding the expressiveness of QREs.

6.4 Improving Propagation of Automaton-Induced Constraint Decompositions

In Chapter 4 we present two approaches for deriving implied constraints for automaton-induced constraint decompositions, namely linear implied constraints (Section 4.2) and glue constraints (Section 4.3). In this section we describe other approaches aimed at improving the propagation of automaton-induced constraint decompositions.

Structure of an Automaton-Induced Constraint Decomposition

The structure of a constraint problem can determine how easy it is to solve it [32]. In [29, 30] we review some of these results and apply them to automaton-induced constraint decompositions.

Manually Derived Decompositions of Constraints

There is also a large body of related work (e.g., [19, 20, 21, 45]) on decomposing global constraints manually in order to maintain HAC on the whole decomposition. Papers II, III, and V can be seen as a more systematic approach towards improving propagation, and possibly maintaining HAC, of automaton-induced decompositions.

Graph Invariants

There is some related work using graph invariants to systematically derive implied constraints in [14] in order to improve efficiency.

For example, the GROUP(X,W,G,V,H,L) constraint [10] has graph invariants, which can be seen as implied constraints. In particular, consider the following bounds on V:

$$\max(G-1,0)\cdot L+H \leq V \leq \max(G-1,0)\cdot H+L$$

Intuitively, the lower bound corresponds to having one group of H elements while all the other groups are as small as possible, that is they have L elements. The upper bound is justified in a similar way. There are 90 graph invariants in [14] for the GROUP constraint predicate: the pruning upon adding all the corresponding implied constraints is compared with the glue constraints in Paper V.

Papers II–VI explore different approaches, which are capable of deriving other implied constraints and do not require a database of invariants.

Automated Reasoning

Initial experiments with using automated reasoning systems towards inferring implied algebraic constraints from a constraint problem are reported in [31, 35], both using an extension [35] of the PRESS equation solver. In contrast, the work in Paper II uses a tool for deriving loop-invariant, while in Papers III and IV we develop a specialised tool for deriving implied-constraint for automaton-induced decompositions. Our work is aimed at a specialised class of constraint problems (namely automaton-induced decompositions) and at a specialised class of implied constraints (on the induced accumulator variables), and is therefore more successful.

6.5 Time-Series Constraints

In Chapter 3 we present time-series constraint predicates and explain how to use our transducer generator in Paper I together with the automaton synthesiser in [11] to synthesise automaton-based constraint predicate descriptions for time-series constraint predicates.

Our approach in Paper I to generating transducers from patterns differs from that of [49], which formulates the semantics of regular-expression matching as a non-deterministic transducer, while we generate minimal deterministic transducers for a specific output alphabet in Paper I. Note that neither determinising nor minimising transducers is in general a trivial task. Most general approaches have underlying assumptions, for example that the transducers are acyclic [39].

Moreover, in Papers IV and VI we enhance the propagation for time-series constraint predicates. Rather than improving the automaton-induced decomposition for each time-series constraint predicate independently, we modify the synthesiser in [11] in order to improve the synthesised automata, that is to generate automata with fewer accumulators and simplified accumulator updates. Moreover, we automatically derive linear implied constraints and introduce the concepts of *parametric bounds* and *parametric glue constraints*. Our approach differs from existing ones, which design dedicated propagation algorithms [46, 8] and reformulations [19, 20] for specific constraints, or propose generic approaches [56, 42] that do not focus on the combinatorial aspect of a constraint predicate.

7. Conclusion

> "Would you tell me, please, which way I ought to go from here?" "That depends a good deal on where you want to get to," said the Cat. "I don't much care where–" said Alice. "Then it doesn't matter which way you go," said the Cat. "–so long as I get *somewhere*," Alice added as an explanation. "Oh, you're sure to do that," said the Cat, "if you only walk long enough."
>
> *Alice's Adventures in Wonderland*
> LEWIS CARROLL

Automata with accumulators provide a uniform description format for many constraint predicates. We believe that automatically synthesising automaton-based descriptions of time-series constraint predicates, as well as automatically deriving implied constraints that improve propagation on automaton-induced constraint decompositions help extending CP solvers with new constraint predicates.

We first summarise the contributions of this dissertation in Section 7.1. We then describe in Section 7.2 some lines of future work in the areas of the synthesis of automaton-based descriptions of time-series constraints and the derivation of implied constraints for automaton-induced constraint decompositions.

7.1 Contributions

First, we have formalised the class of time-series patterns whose transducers can be handled by the automaton synthesiser in [11] and we have described a fully automated parametric tool that generates, in an off-line process, a transducer from such a pattern. Note that our tool generates exactly the handcrafted transducers of [11]. By proving that our tool only generates well-formed transducers, we have also proved that the transducers in [11] are well-formed.

This work contributes, in the context of time-series constraint predicates, to the systematic reconstruction of the Global Constraint Catalogue [10], advocated in [15]. These constraint predicates are useful not only to describe

properties of known time series, but can also be used in a wide range of applications to constrain (parts of) new time series based on previously observed samples.

Second, we have described a fully automated parametric tool that selects, in an off-line process, a set of non-redundant linear constraints that are implied by the automaton-induced decomposition in [13] of a constraint on a sequence of variables, the constraint predicate being described by a checker provided as an automaton with accumulators. We have shown that a suitable choice, by the user, among the tool-selected implied constraints can considerably improve solving time and propagation, both on a decomposition in isolation and on entire constraint models containing the decomposition. With the extra propagators for the implied constraints, it potentially takes more time to compute the fixpoint of the propagators at each node of the search tree: this may backfire on the decomposition alone, but usually pays off on entire constraint models containing the decomposition, due to the extra propagation. Within the context of automaton-based descriptions of constraint predicates in general, and time-series constraint predicates in particular, the results of this work have been shown to improve significantly both constraint programming (CP) and mixed-integer programming (MIP) models.

Finally, for a total-function constraint on a sequence of variables whose result does not change under sequence reversal, we have shown how to derive, from a description of the constraint predicate by an automaton with accumulators, an implied constraint, called *glue constraint*, between the result variables for a sequence of variables, a prefix thereof, and the corresponding suffix. Such total-function constraints have proved very useful, for instance in production sequencing and staff rostering. We have shown that the glue constraint is unique and always exists. We have also shown the usefulness of the glue constraint in solving, both by local search, where the glue constraint enables constant-time move probing, and by propagation-based systematic CP search, where the glue constraint improves propagation: our concept is thus not oriented toward a specific solving technology. Within the context of time-series constraints in particular, we have further enhanced the propagation of time-series constraints by a systematic derivation of glue constraints. Rather than deriving glue constraints for each time-series constraint predicate independently, we have introduced the concepts of parametric glue constraints.

We hope our work motivates the quest for other general results that have a positive impact on several solving technologies, such as CP, MIP, local search, and Boolean satisfiability (SAT).

7.2 Future Work

There are some directions of future work that we would like to explore, and we outline them here separated by area: generation of time-series constraint de-

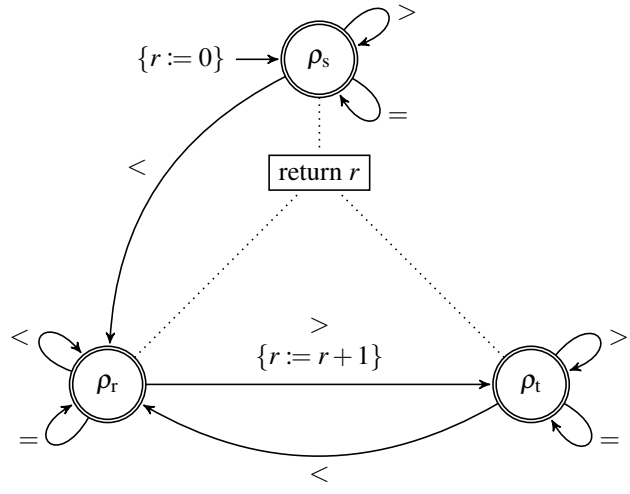


Figure 7.1. Automatically simplified automaton for the SUMONEPEAK constraint predicate in the Time-Series Constraint Catalogue [5].

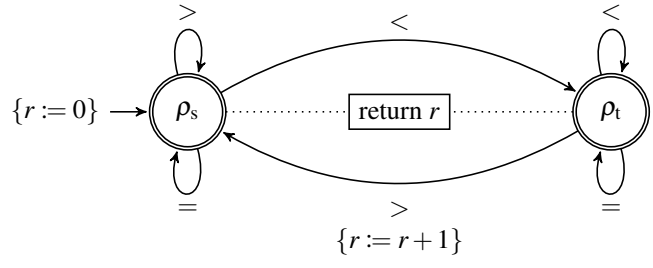


Figure 7.2. Handcrafted automaton for the SUMONEPEAK constraint predicate in the Global Constraint Catalogue [10].

scriptions, and implied-constraints for automaton-induced constraint decompositions.

Generation of Time-Series Constraint Predicate Descriptions

It would be interesting to extend the phase alphabet to be able to automatically generate seed transducers for a larger class of patterns. This would require modifying both the algorithm in Paper I and the synthesiser in [11].

Another research direction would be to design a minimisation algorithm for the synthesised automata. For example, as can be seen in Figure 7.1, the simplified automaton synthesised in [11] for the SUMONEPEAK[1] constraint predicate has three states. The SUMONEPEAK constraint predicate can also

[1]The SUMONEPEAK constraint predicate is called NBPEAK in Paper IV and [5], and simply PEAK in [10].

be described using the handcrafted automaton in [10], which, as can be seen in Figure 7.2, has only two states. Minimising automata with accumulators is not a trivial task, as it would require, in the general case, moving accumulator updates to earlier (or later) transitions, as well as determining whether or not two accumulator updates are equivalent. Another example is the GROUP constraint predicate. As seen in Section 4.2, a $\text{GROUP}(X,W,G,V,H,L)$ constraint can be decomposed into the conjunction $\text{GROUPG}(X,W,G) \wedge \text{GROUPV}(X,W,V) \wedge \text{GROUPH}(X,W,H) \wedge \text{GROUPL}(X,W,L)$, and each of the involved four constraint predicates can be described using an automaton [13]. Moreover, each of those four constraint predicates can be described as a time-series constraint, that is, by a pattern, a feature and an aggregator, and thus it is possible to use the algorithm in Paper I to generate a transducer from the pattern, and then use the generated transducer, together with the feature and the aggregator as inputs to the synthesiser in [11]. The problem is again that all four automata would have the same number of states, namely two, while some of the handcrafted automata in [10] have only one state.

Implied Constraints for Automaton-Induced Constraint Decompositions

It would be interesting to extend our ImpGen tool to use a richer template than linear inequalities for implied constraints. For instance, a non-linear template can be used by exploiting Gröbner bases. Also, disjunction enables the generation of implicative implied constraints. A motivating example is our manual derivation in Section III of Paper II of the disjunctive non-linear implied constraint

$$(S_i = 0 \vee j_{i-1} \neq J-1) \Leftrightarrow p_{i+1} \neq i$$

for $0 < i < n$, for a $\text{JTHNONZEROPOS}(\mathcal{V},J,P)$ constraint, whose automaton is in Figure 2.5.

Moreover, we mention in Paper III that an implied equality constraint of the form $C_{i-1} = C_i$ can be inferred from the implied constraints $C_{i-1} \leq C_i$ and $C_{i-1} \geq C_i$. It would be interesting to process the linear implied constraints in order to derive stronger and fewer implied constraints like linear equalities, or for even richer templates including the 'max', 'min', and 'abs' operators, such as the implied constraints $\text{abs}(C_i - C_{i-1}) \leq 1$ and $C_i \geq \max(D_i, C_{i-1})$.

8. Glossary

> "Speak English!" said the Eaglet. "I don't know the meaning of half those long words, and I don't believe you do either!"
>
> *Alice's Adventures in Wonderland*
> LEWIS CARROLL

To make this dissertation self-contained, we define the remaining used concepts, namely extensionally-defined constraint, support, hyper-arc consistency, checker, and reification.

A constraint $\gamma(\mathcal{V})$ on a sequence of decision variables $\mathcal{V} = \langle V_1, \ldots, V_n \rangle$ can be defined in a *extensional* fashion by listing all the combinations of domain-compatible assignments to $\mathcal{V}$ that satisfy γ.

The assignment $V_k = d_k$ is *supported* by a constraint (problem) if there is a solution to that constraint (problem) where $V_k = d_k$ and all decision variables take values in their current domains.

There is *hyper-arc consistency* (*HAC*) *on a constraint* $\gamma(\mathcal{V})$ if every domain value of every decision variable of $\mathcal{V}$ is supported by $\gamma(\mathcal{V})$; we also say that $\gamma(\mathcal{V})$ is HAC. Hyper-arc consistency is also known as *generalised arc consistency* and *domain consistency*.

There is *HAC on a constraint problem* if every current domain value of every decision variable of the problem is supported by the problem; we also say that the problem is HAC.

A *checker* is an algorithm that returns true if and only if a given assignment is a solution to a given constraint. For example, consider the constraint $\textsc{Exactly}(N, \mathcal{V}, P)$, which holds if and only if the sequence $\mathcal{V}$ of decision variables contains exactly N elements taking the given value P. Parameters N and P must be constants, under the restriction $0 \leq N \leq |\mathcal{V}|$. For instance,

Algorithm 3: Checker for an $\textsc{Exactly}(N, \mathcal{V}, P)$ constraint

```
i := 1
c := 0
while i ≤ |V| do
    if V[i] = P then c := c + 1;
    i := i + 1
return N = c
```

EXACTLY$(2, \langle 4,2,4,5 \rangle, 4)$ holds since exactly 2 elements of the sequence $\langle 4,2,4,5 \rangle$ take the value 4. A checker for the EXACTLY constraint is given in Algorithm 3.

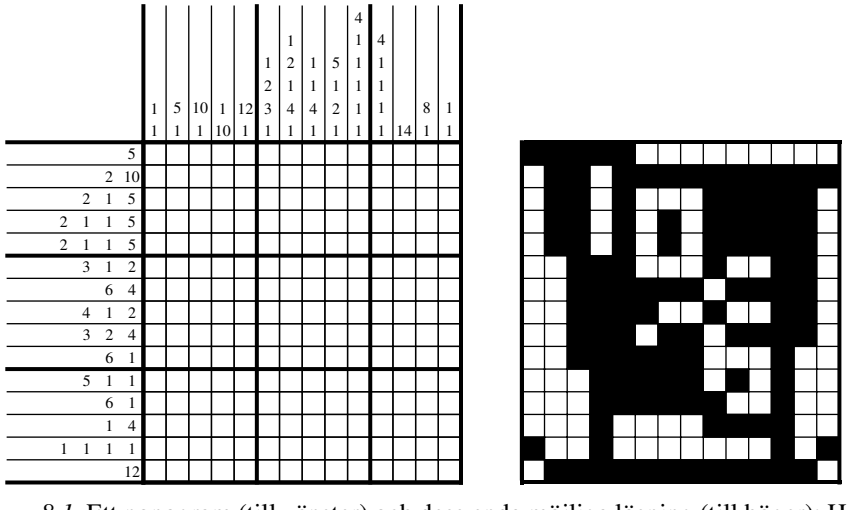
A *reified constraint* [51] $\gamma(\mathcal{V}) \Leftrightarrow B$ associates the truth value of the constraint $\gamma(\mathcal{V})$ with a $0/1$ decision variable B. If $\gamma(\mathcal{V})$ is entailed by the domains of the decision variables $\mathcal{V}$, then B is constrained to be 1. Conversely, if $\gamma(\mathcal{V})$ is disentailed by the domains of the decision variables $\mathcal{V}$, then B is constrained to be 0. Moreover, if $B = 1$ holds, then $\gamma(\mathcal{V})$ must hold, and if $B = 0$ holds, then $\neg\gamma(\mathcal{V})$ must hold.

Sammanfattning på svenska

> "Var behagar Ers Majestät att jag börjar?" frågade han. "Börja med början," sa kungen högst allvarligt, "och läs tills du kommer till slutet. Sluta där."
>
> *Alice i Underlandet*
> LEWIS CARROLL
> Översättning: ÅKE RUNNQUIST

Figur 8.1 visar ett nonogram. Ett nonogram är en gåta i form av ett rutnät, där varje ruta måste vara antingen tom (vit) eller fylld (svart). Om alla rutorna är korrekt ifyllda (enligt givna ledtrådar, d.v.s. siffrorna till vänster och ovanför rutnätet) avslöjas en bild. Nonogrammet i figur 8.1, till exempel, gömmer en bild av en hatt. Varje ledtråd indikerar hur många rutor som ska vara svarta i en given rad eller kolumn och hur dessa är fördelade. Till exempel betyder ledtråden '4 8 3' att det ska finnas tre sekvenser med svarta rutor med längden fyra, åtta respektive tre rutor, och med minst en vit ruta emellan.

För att lösa ett nonogram inom rimlig tid, är det inte möjligt att bara använda sig av trial-and-error, eftersom det helt enkelt finns alldeles för många



Figur 8.1. Ett nonogram (till vänster) och dess enda möjliga lösning (till höger): Hattmakarens hatt.

sätt att fylla i rutorna. Detsamma gäller för de flesta andra intressanta gåtor och verkliga kombinatoriska problem. Ett sätt att hantera sådana problem är istället att beskriva dem som *villkorsproblem*, och lösa dem med hjälp av *villkorsprogrammering* (eng. "constraint programming"). Villkorsprogrammering är ett sätt att programmera deklarativt i syfte att modellera och lösa kombinatoriska problem med hjälp av villkorslösare (eng. "constraint solvers"). Det finns många tillämpningsområden, t.ex. inom schemaläggning, där man använder villkorsprogrammering med goda resultat.

Villkorsproblem

Ett villkorsproblem specificeras över en mängd variabler (som kan anta olika värden) och en mängd villkor (som anger vilka värden de olika variablerna får anta samtidigt). I exemplet med nonogrammet kan varje ruta i rutnätet representeras av en variabel. Variablerna kan anta värden från en given domän. I ett nonogram kan rutorna vara svarta eller vita, så domänen för variablerna kan uttryckas som $\{\text{svart}, \text{vit}\}$. Villkoren som måste uppfyllas representeras här av ledtrådarna: varje ledtråd kan sägas vara ett villkor över en viss rad eller kolumn, och tillsammans specificerar alla ledtrådarna villkoren för nonogrammet som helhet.

Formellt sett är ett villkorsproblem en konjunktion bestående av ett villkor över en mängd variabler och dessa variablers domän(er), till exempel

$$\text{ALLDIFFERENT}(V_1, V_2, V_3) \wedge V_1 + V_3 = 4 \tag{8.1}$$

där $\text{ALLDIFFERENT}(V_1, V_2, V_3)$ anger att variablerna V_1, V_2 och V_3 alla ska anta olika värden, och där $V_1 + V_3 = 4$ anger att värdet på V_1 och V_3 tillsammans ska bli 4. Domänen för alla variablerna kan skrivas $\text{dom}(V_i) = \{1, 2, 3, 4\}$.

Medan ett nonogram ofta är gjort så att det bara har en enda möjlig lösning, kan villkorsproblem i allmänhet ha mer än en lösning (eller ingen alls). Villkorsproblemet i (8.1) ovan, till exempel, har mer än en lösning. Ibland kan vissa lösningar också vara mätbart bättre än andra, och då kan målet vara att hitta den bästa lösningen. I sådana fall kallas problemet ett *villkorsoptimeringsproblem* (eng. "constraint optimisation problem"). Om vi till exempel är intresserade av lösningar på problemet i (8.1) där värdet på V_2 är så stort som möjligt, är $V_1 = 1$, $V_2 = 4$, $V_3 = 3$ ett optimalt val.

Villkorspredikat

En viktig komponent i moderna villkorslösare är *villkorspredikat*. ALLDIFFERENT är exempel på ett villkorspredikat. Ur ett modelleringsperspektiv gör villkorspredikat det enklare att modellera vanligt återkommande delar av villkorsproblem, till exempel att en viss mängd variabelvärden

ska vara skilda från varandra genom predikatet ALLDIFFERENT. Ur ett lösningsperspektiv gör villkorspredikat att omöjliga värden på variabler kan elimineras direkt. En algoritm för beräkning av omöjliga värden på variabler kallas här *propagator*.

Ett problem med villkorspredikat är att de ofta inte existerar. Det gäller till exempel sådana som kan beskriva *tidsserier*, d.v.s. sekvenser av heltal som representerar mätningar över olika tidsintervall. Tidsserier förekommer i många olika tillämpningsområden, såsom elproduktion, personalbehov i ett call center eller patientkapaciteten per dag på ett sjukhus. Ett sätt att lösa detta problem är att dela upp villkorspredikaten i mindre delar. Då kan man utnyttja propagatorer som redan finns för de ingående delarna. Ett annat sätt är att använda sig av finita automater eller reguljära uttryck för att beskriva ett villkorspredikat. Till exempel kan uttrycket $\text{vit}^*\text{svart}^4\text{vit}^+\text{svart}^3\text{vit}^*$ användas för att beskriva att en rad i ett nonogram ska innehålla exakt två sekvenser med svarta rutor, den första fyra rutor lång och den andra tre rutor lång. Mellan sekvenserna ska finnas minst en vit ruta, och före samt efter sekvenserna av svarta rutor ett godtyckligt antal vita rutor. En finit automat kan sedan kontrollera om det reguljära uttrycket är uppfyllt eller ej på en viss rad i ett nonogram.

Forskningsproblem

I allmänhet är det inte möjligt att eliminera alla omöjliga värden på variabler, även med användning av propagatorer och uppdelning av villkorspredikat. Det är ett känt problem, och i denna avhandling tacklas det på två sätt: dels med fokus på hur villkorspredikat för tidsserier kan beskrivas med hjälp av finita automater, och dels med fokus på förbättring av propagatorer.

I Paper I visas hur villkorspredikat för tidsserier kan representeras med hjälp av finita automater och hur dessa automater kan genereras direkt från reguljära uttryck. Vi beskriver för vilka typer av reguljära uttryck som det är möjligt att generera automater, och ger en algoritm för att generera dem. Tillsammans med tidigare arbeten inom området ([11, 13]) kan våra resultat användas för att automatisera t.ex. uppdelning av villkorspredikat.

I Paper II–VI visar vi hur vi kan få fram implicita villkor från uppdelade villkorspredikat. Ett implicit villkor är en logisk konsekvens av andra villkor. Implicita villkor påverkar inte mängden möjliga lösningar på ett problem, men genom att lägga till dem i en modell kan man i vissa fall göra en tidsvinst under lösningen, eftersom man kan använda ytterligare propagatorer för dessa villkor. Våra resultat kan ses som en del i att förbättra möjligheten till propagering på automatisk väg.

References

[1] Abderrahmane Aggoun and Nicolas Beldiceanu. Extending CHIP in order to solve complex scheduling and placement problems. *Mathematical and Computer Modelling*, 17(7):57–73, 1993.

[2] Cyril Allauzen and Mehryar Mohri. Efficient algorithms for testing the twins property. *Journal of Automata, Languages and Combinatorics*, 8(2):117–144, 2003.

[3] Rajeev Alur, Dana Fisman, and Mukund Raghothaman. Regular programming for quantitative properties of data streams. In Peter Thiemann, editor, *ESOP 2016*, volume 9632 of *LNCS*, pages 15–40. Springer, 2016.

[4] Rajeev Alur, Adam Freilich, and Mukund Raghothaman. Regular combinators for string transformations. In Thomas A. Henzinger and Dale Miller, editors, *CSL-LICS 2014*, pages 9:1–9:10. ACM, 2014.

[5] Ekaterina Arafailova, Nicolas Beldiceanu, Rémi Douence, Mats Carlsson, Pierre Flener, María Andreína Francisco Rodríguez, Justin Pearson, and Helmut Simonis. Global Constraint Catalog, Volume II, Time-Series Constraints. *arXiv:1609.08925*, 2016. Available at `https://arxiv.org/abs/1609.08925`.

[6] Ekaterina Arafailova, Nicolas Beldiceanu, Rémi Douence, Pierre Flener, María Andreína Francisco Rodríguez, Justin Pearson, and Helmut Simonis. Time-series constraints: Improvements and application in CP and MIP contexts. In Claude-Guy Quimper, editor, *CP-AI-OR 2016*, volume 9676 of *LNCS*, pages 18–34. Springer, 2016.

[7] Philippe Baptiste, Claude Le Pape, and Wim Nuijten. *Constraint-Based Scheduling: Applying Constraint Programming to Scheduling Problems*. Kluwer Academic Publishers, 2001.

[8] Nicolas Beldiceanu and Mats Carlsson. Sweep as a generic pruning technique applied to the non-overlapping rectangles constraints. In Toby Walsh, editor, *CP 2001*, volume 2239 of *LNCS*, pages 377–391. Springer, 2001.

[9] Nicolas Beldiceanu, Mats Carlsson, Romuald Debruyne, and Thierry Petit. Reformulation of global constraints based on constraints checkers. *Constraints*, 10(4):339–362, 2005.

[10] Nicolas Beldiceanu, Mats Carlsson, Sophie Demassey, and Thierry Petit. Global constraint catalogue: Past, present, and future. *Constraints*, 12(1):21–62, March 2007. Catalogue at `www.emn.fr/x-info/sdemasse/gccat`.

[11] Nicolas Beldiceanu, Mats Carlsson, Rémi Douence, and Helmut Simonis. Using finite transducers for describing and synthesising structural time-series constraints. *Constraints*, 21(1):22–40, January 2016.

[12] Nicolas Beldiceanu, Mats Carlsson, Pierre Flener, and Justin Pearson. On the reification of global constraints. *Constraints*, 18(1):1–6, January 2013.

[13] Nicolas Beldiceanu, Mats Carlsson, and Thierry Petit. Deriving filtering algorithms from constraint checkers. In Mark Wallace, editor, *CP 2004*, volume 3258 of *LNCS*, pages 107–122. Springer, 2004.

[14] Nicolas Beldiceanu, Mats Carlsson, Jean-Xavier Rampon, and Charlotte Truchet. Graph invariants as necessary conditions for global constraints. In Peter van Beek, editor, *CP 2005*, volume 3709 of *LNCS*, pages 92–106. Springer, 2005.

[15] Nicolas Beldiceanu, Pierre Flener, Jean-Noël Monette, Justin Pearson, and Helmut Simonis. Toward sustainable development in constraint programming. *Constraints*, 19(2):139–149, 2014.

[16] Nicolas Beldiceanu, Pierre Flener, Justin Pearson, and Pascal Van Hentenryck. Propagating regular counting constraints. In Carla E. Brodley and Peter Stone, editors, *AAAI 2014*, pages 2616–2622. AAAI Press, 2014.

[17] Nicolas Beldiceanu, Georgiana Ifrim, Arnaud Lenoir, and Helmut Simonis. Describing and generating solutions for the EDF unit commitment problem with the ModelSeeker. In Christian Schulte, editor, *CP 2013*, volume 8124 of *LNCS*, pages 733–748. Springer, 2013.

[18] Michael Benedikt, Clemens Ley, and Gabriele Puppis. What you must remember when processing data words. In *AMW 2010*, volume 619. CEUR-WS. org, 2010.

[19] Christian Bessière, Emmanuel Hebrard, Brahim Hnich, Zeynep Kiziltan, Claude-Guy Quimper, and Toby Walsh. Reformulating Global Constraints: The *slide* and *regular* Constraints. In Ian Miguel and Wheeler Ruml, editors, *SARA 2007*, volume 4612 of *LNAI*, pages 80–92. Springer, 2007.

[20] Christian Bessière, George Katsirelos, Nina Narodytska, Claude-Guy Quimper, and Toby Walsh. Decomposition of the NValue constraint. In David Cohen, editor, *CP 2010*, volume 6308 of *LNCS*, pages 114–128. Springer, 2010.

[21] Christian Bessière, George Katsirelos, Nina Narodytska, and Toby Walsh. Circuit complexity and decompositions of global constraints. In Craig Boutilier, editor, *IJCAI 2009*, pages 412–418. AAAI Press, 2009.

[22] Stéphane Bourdais, Philippe Galinier, and Gilles Pesant. HIBISCUS: A constraint programming application to staff scheduling in health care. In Francesca Rossi, editor, *CP 2003*, volume 2833 of *LNCS*, pages 153–167. Springer, 2003.

[23] Stephen P. Boyd and Lieven Vandenberghe. *Convex Optimization*. Cambridge University Press, 2004.

[24] Aaron R. Bradley and Zohar Manna. Property-directed incremental invariant generation. *Formal Aspects of Computing*, 20(4–5):379–405, 2008.

[25] Mats Carlsson, Nicolas Beldiceanu, and Julien Martin. A geometric constraint over k-dimensional objects and shapes subject to business rules. In Peter J. Stuckey, editor, *CP 2008*, volume 5202 of *LNCS*, pages 220–234. Springer, 2008.

[26] Mats Carlsson, Greger Ottosson, and Björn Carlson. An open-ended finite domain constraint solver. In Hugh Glaser, Pieter Hartel, and Herbert Kuchen, editors, *PLILP 1997*, volume 1292 of *LNCS*, pages 191–206. Springer, 1997.

[27] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 3rd edition, 2009.

[28] Manfred Droste, Werner Kuich, and Heiko Vogler, editors. *Handbook of*

Weighted Automata. Monographs in Theoretical Computer Science. Springer, 2009.

[29] María Andreína Francisco Rodríguez. Consistency of constraint networks induced by automaton-based constraint specifications. Master's thesis, Department of Information Technology, Uppsala University, Sweden, 2011. Available at `http://urn.kb.se/resolve?urn=urn:nbn:se:uu:diva-156441`.

[30] María Andreína Francisco Rodríguez, Pierre Flener, and Justin Pearson. Consistency of constraint networks induced by automaton-based constraint specifications. In Andrea Rendl and J. Christopher Beck, editors, *ModRef 2011*, pages 117–131, 2011. Available at `http://www-users.cs.york.ac.uk/~frisch/ModRef/11`.

[31] Alan M. Frisch, Ian Miguel, and Toby Walsh. Extensions to proof planning for generating implied constraints. In *Calculemus 2001*, pages 130–141, 2001. Available at `http://www.cs.york.ac.uk/aig/projects/implied/docs/FrischMiguelWalsh.ps`.

[32] Georg Gottlob, Nicola Leone, and Francesco Scarcello. A comparison of structural CSP decomposition methods. *Artificial Intelligence*, 124(2):243–282, 2000.

[33] Ashutosh Gupta and Andrey Rybalchenko. InvGen: An Efficient Invariant Generator. In Ahmed Bouajjani and Oded Maler, editors, *CAV 2009*, volume 5643 of *LNCS*, pages 634–640. Springer, 2009.

[34] Jun He, Pierre Flener, and Justin Pearson. Underestimating the cost of a soft constraint is dangerous: Revisiting the edit-distance based SoftRegular constraint. *Journal of Heuristics*, 19(5):729–756, October 2013.

[35] Brahim Hnich, Julian Richardson, and Pierre Flener. Towards automatic generation and evaluation of implied constraints. Technical Report 2003-014, Department of Information Technology, Uppsala University, Sweden, originally written in August 2000.

[36] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 3rd edition, 2007.

[37] Michael Kaminski and Nissim Francez. Finite-memory automata. *Theoretical Computer Science*, 134(2):329–363, 1994.

[38] Michel Minoux. Personal communication to Nicolas Beldiceanu, July 2015.

[39] Mehryar Mohri. Minimization algorithms for sequential transducers. *Theoretical Computer Science*, 234(1-2):177–201, 2000.

[40] Mehryar Mohri. Weighted automata algorithms. In Manfred Droste, Werner Kuich, and Heiko Vogler, editors, *Handbook of Weighted Automata*, Monographs in Theoretical Computer Science, pages 213–254. Springer, 2009.

[41] Mehryar Mohri, Fernando C. N. Pereira, and Michael Riley. The design principles of a weighted finite-state transducer library. *Theoretical Computer Science*, 231(1):17–32, 2000. The *OpenFst Library* is available at `http://www.openfst.org`.

[42] Jean-Noël Monette, Pierre Flener, and Justin Pearson. Towards solver-independent propagators. In Michela Milano, editor, *CP 2012*, volume 7514 of *LNCS*, pages 544–560. Springer, 2012.

[43] Gilles Pesant. A regular language membership constraint for finite sequences

of variables. In Mark Wallace, editor, *CP 2004*, volume 3258 of *LNCS*, pages 482–495. Springer, 2004.

[44] Claude-Guy Quimper and Toby Walsh. Global grammar constraints. In Frédéric Benhamou, editor, *CP 2006*, volume 4204 of *LNCS*, pages 751–755. Springer, 2006.

[45] Claude-Guy Quimper and Toby Walsh. Decomposing global grammar constraints. In Christian Bessière, editor, *CP 2007*, volume 4741 of *LNCS*, pages 590–604. Springer, 2007.

[46] Jean-Charles Régin. A filtering algorithm for constraints of difference in CSPs. In Barbara Hayes-Roth and Richard E. Korf, editors, *AAAI 1994*, pages 362–367. AAAI Press, 1994.

[47] Francesca Rossi, Peter van Beek, and Toby Walsh, editors. *Handbook of Constraint Programming*. Elsevier, 2006.

[48] Jacques Sakarovitch. *Elements of Automata Theory*. Cambridge University Press, 2009.

[49] Yuto Sakuma, Yasuhiko Minamide, and Andrei Voronkov. Translating regular expression matching into transducers. *Journal of Applied Logic*, 10(1):32 – 51, 2012.

[50] Sriram Sankaranarayanan, Henny B. Sipma, and Zohar Manna. Constraint-based linear-relations analysis. In Roberto Giacobazzi, editor, *SAS 2004*, volume 3148 of *LNCS*, pages 53–68. Springer, 2004.

[51] Christian Schulte and Mats Carlsson. Finite domain constraint programming systems. In Francesca Rossi, Peter van Beek, and Toby Walsh, editors, *Handbook of Constraint Programming*, chapter 14, pages 495–526. Elsevier, 2006.

[52] Meinolf Sellmann. The theory of grammar constraints. In Frédéric Benhamou, editor, *CP 2006*, volume 4204 of *LNCS*, pages 530–544. Springer, 2006.

[53] Rahul Sharma, Işıl Dillig, Thomas Dillig, and Alex Aiken. Simplifying loop invariant generation using splitter predicates. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *CAV 2011*, volume 6806 of *LNCS*, pages 703–719. Springer, 2011.

[54] Barbara M. Smith. Modelling. In Francesca Rossi, Peter van Beek, and Toby Walsh, editors, *Handbook of Constraint Programming*, chapter 11, pages 377–406. Elsevier, 2006.

[55] Pascal Van Hentenryck and Laurent Michel. *Constraint-Based Local Search*. The MIT Press, 2005.

[56] Pascal Van Hentenryck, Vijay Saraswat, and Yves Deville. Design, implementation, and evaluation of the constraint language cc(FD). Technical Report CS-93-02, Brown University, Providence, USA, January 1993. Revised version in *Journal of Logic Programming* 37(1–3):293–316, 1998. Based on the unpublished manuscript *Constraint Processing in cc(FD)*, 1991.

Acta Universitatis Upsaliensis

Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology 1591

Editor: The Dean of the Faculty of Science and Technology

A doctoral dissertation from the Faculty of Science and Technology, Uppsala University, is usually a summary of a number of papers. A few copies of the complete dissertation are kept at major Swedish research libraries, while the summary alone is distributed internationally through the series Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology. (Prior to January, 2005, the series was published under the title "Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology".)

Distribution: publications.uu.se
urn:nbn:se:uu:diva-332149

ACTA
UNIVERSITATIS
UPSALIENSIS
UPPSALA
2017