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.

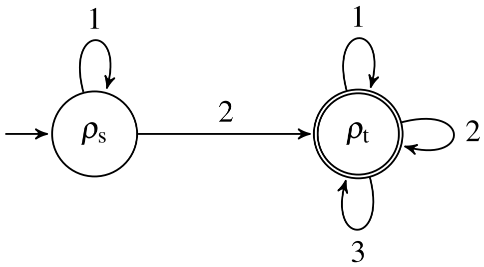

Figure 2.1. DFA for the regular expression $1^*2(1|2|3)^*$.

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

One often uses pictures to define finite automata. For example, in Figure 2.1, we define an automaton with two states, $Q = \{\rho_\mathrm{s}, \rho_\mathrm{t}\}$, represented by circles, and an alphabet of three symbols, $\Gamma = \{1, 2, 3\}$, on the transitions. The initial state $\rho_0 = \rho_\mathrm{s}$ is indicated by an arrow coming from nowhere, and an accepting state is represented by a double circle, and so $Q_\mathrm{a} = \{\rho_\mathrm{t}\}$. 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 $\langle Q, \Gamma, \Gamma', \delta, \rho_0, Q_\mathrm{a} \rangle$, 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 $Q_\mathrm{a} \subseteq Q$ 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 $\mathcal{A} = \langle Q,\Gamma,\delta,\rho_0,Q_\mathrm{a}\rangle$, 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*:

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

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.

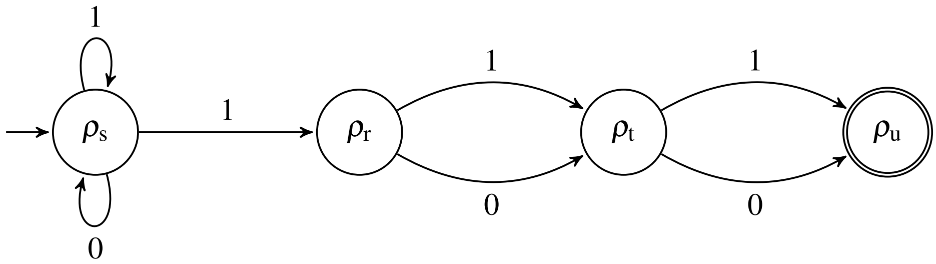

Figure 2.2. NFA for the regular expression $(0|1)^*1(0|1)^2$: all 0/1 sequences that have a 1 two characters from the end of the sequence.

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

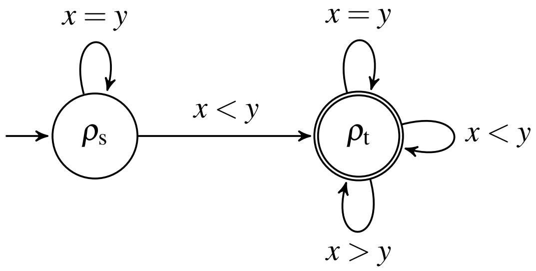

Figure 2.3. A k-ary predicate automaton with $k = 2$ describing the $<_{\text{lex}}$ constraint predicate.

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

$$\rho_s \xrightarrow{1=1} \rho_s \xrightarrow{2<3} \rho_t \xrightarrow{5>4} \rho_t \xrightarrow{6<7} \rho_t$$

ends in the accepting state ρ_t.

Given a predicate automaton $\langle Q, \Gamma, \delta, \phi, \rho_0, Q_a \rangle$, the automaton $\langle Q, \Gamma, \delta, \rho_0, Q_a \rangle$ 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:

$$q_0 = \rho_0 \wedge \mathrm{T}(q_0, S_1, q_1) \wedge \cdots \wedge \mathrm{T}(q_{n-1}, S_n, q_n) \wedge q_n \in Q_a \qquad (2.3)$$

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*:

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

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.

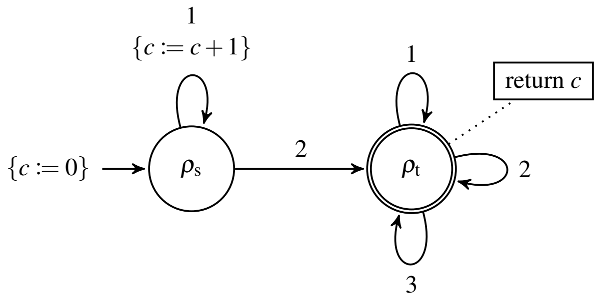

Figure 2.4. Automaton with $\ell = 1$ accumulator for the regular expression $1^*2(1|2|3)^*$. Accumulator c maintains the length of the longest prefix matching the regular expression 1^* of the sequence of symbols consumed so far.

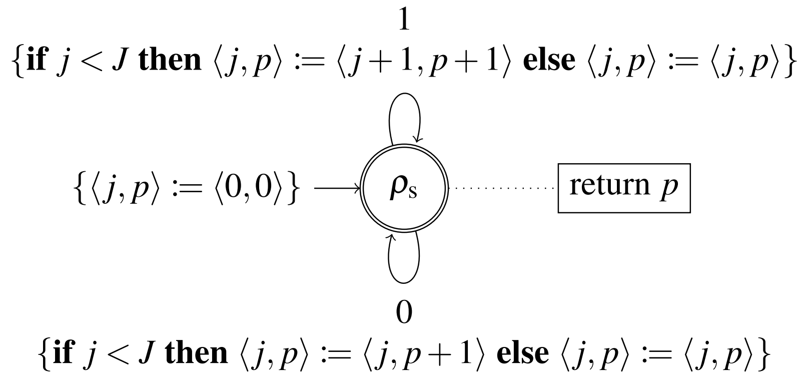

Figure 2.5. An ℓ-ary automaton with accumulators with $\ell = 2$ accumulators describing the JTHNONZEROPOS$(\mathcal{V}, J, P)$ constraint [10], which holds if and only if P is the position (counting from 1) of the J^{th} non-zero element of the sequence $\mathcal{V} = \langle V_1, \ldots, V_n \rangle$. 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 $\delta(\rho_{\text{s}}, 1) = (\rho_{\text{s}}, \langle c+1 \rangle)$ 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 $\langle c \rangle := \langle c \rangle$. If an update involves only one accumulator, then we omit the angled brackets; for example, the self-loop on ρ_{s} has $c := c+1$ instead of $\langle c \rangle := \langle c+1 \rangle$. 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:

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

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,

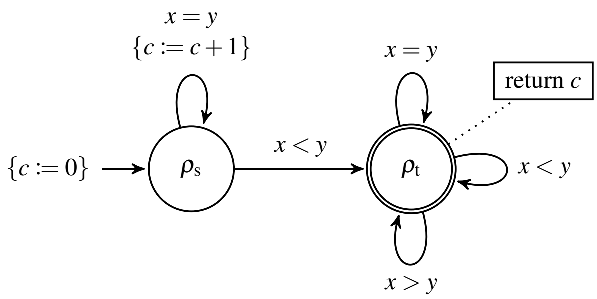

Figure 2.6. A $\langle 2,1 \rangle$-ary predicate automaton with accumulators describing a constraint predicate on two sequences of decision variables $\mathcal{V}$ and $\mathcal{W}$ which holds if and only if $\mathcal{V} <_{\text{lex}} \mathcal{W}$ holds and accumulator c denotes the length of the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

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

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

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.

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