Between Finite State and Prolog: Constraint-Based Automata and Efficient Recognition of Phrases

Klaus U. Schulz and Thomasz Mikolajewski
CIS, University of Munich
Oettingenstr. 67, D-80538 München, Germany
e-mail: schulz/thomasz@cis.uni-muenchen.de

Abstract

This paper describes a new type of automaton that has been developed at CIS Munich for efficient recognition of phrases in text files. The concept of a *constraint-based automaton* is tailored to sets of phrases where grammaticality depends on morphological agreement conditions. It incorporates features from three sides: traditional finite state techniques, methods from constraint programming, and some technology from modern large-scale electronic dictionaries. A miniature programming language with a special purpose constraint solver, tokenizing routines for input text files, and with built-in access to large electronic dictionaries has been built to support efficient implementation of constraint-based automata. In this paper we describe the formal concept of a constraint based automaton. We give the syntax of transition rules and explain the procedural behaviour. Additional information on background algorithms and on some details of the miniature programming language are also included. Some experimental results are described that compare the performance of constraint based automata with Prolog and illuminate the range of potential applications.

1 Introduction

In computational linguistics, efficient recognition of phrases is an important prerequisite for many ambitious goals, such as, e.g., automated extraction of terminology , part of speech desambiguation, and automated translation. If one wants to recognize a certain well-defined set of phrases, the question comes up which type of computational device should be optimal for this task. In many cases, finite state methods are appropriate and favourable because of their efficiency ([GrP, Ta95, Sc96]). However, often one would like to recognize sets of more complex phrases where correct resolution of grammatical structure requires morphological analysis. As an example, consider the analysis of german noun phrases.

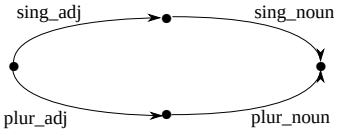
- for a correct separation between two consecutive noun phrases (as in "*der König den Tieren*"[1]) on the one hand and a single complex noun phrase (as in "*der*

[1] "*the king to the animals*"

König der Tiere"[2]) on the other hand, case dependencies have to be taken into account,

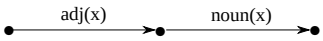
- for a correct analysis of relative clauses with extraposition (as in "*die Fußballspieler den Ball, die..*"[3]) number information is required.

Similarly, morphological information is needed for correct resolution of verb-argument structure and for other phenomena. As the impact of morphological information grows, finite state techniques become more and more inappropriate. In principle, of course, arbitrary agreement constraints can be encoded in the transitions of a finite state automaton, as long as a finite number of morphological variants is considered only. The following figure exemplifies this possibility, showing how to enforce agreement in number of an adjective and a noun in a finite state automaton.



However, if we want recognize a non-trivial subset, say, of all german noun phrases, then the size of any finite state automaton necessarily exceeds a reasonable bound because of the inevitable multiple duplication of transition rules that is caused by the large number of morphological variants. This means that the design and implementation of an appropriate finite state automaton is in practice an infeasible task.

In this situation, Prolog is a prominent alternative. In the example given above, we may just introduce a variable for the number in order to enforce agreement.



Still, one might ask if current Prolog systems offer an optimal solution. Definitely it would be inappropriate to rely on the unification mechanisms. The reason is that the morphological information that comes with a given input token after lexical analysis is disjunctive in nature: from the background dictionary we typically obtain a finite set of possible morphological values for a given word form. Since logical variables cannot represent disjunctions, we would have make extensive use of the backtracking mechanism in the recognition process, which leads to poor performance. Constraint-based extensions of Prolog, such as PROLOG III [Col90] or ECL^{i}PS^e [EC95], offer much better possibilities in this respect but their built-in constraints are of course not tuned to control of morphological agreement conditions. Another point where the possibilities offered by Prolog/CLP might be improved is the lexical look-up. Usually, the integration of dictionaries in CLP languages is made in an ad hoc way or based on traditional data base techniques. However, lexical look-up in electronic dictionaries–where words

[2] "*the king of the animals*"

[3] "*the soccers the ball, who...*".

are represented in a suitable way[4]—can be made much more efficient using transducer techniques. Clearly such techniques should be integrated in any modern system for recognition of phrases. Eventually it is an open question if the heavy machinery of a fully fledged CLP language leads to a loss of efficiency if compared with more limited concepts.

In this paper we describe the notion of a *constraint-based automaton*. The new concept incorporates characteristic features both of finite state techniques and constraint programming; on the implementation side, modern technology as it has been developed in the context of large-scale electronic dictionaries has been integrated.

- From finite state automata we inherit the general structure of rules and deterministic execution.
- A special constraint solver, based on arc-consisteny techniques ([Ma77]), facilitates the control of morphological agreement conditions. An adequate treatment of other linguistic concepts is supported through unification over rational trees ([Col84]).
- Efficient look-up methods, originally developed by the second author for the german CISLEX dictionary [GM94, Ma94], support direct access to large-scale electronic dictionaries of the DELA-type [Co90, Si93].

A miniature programming language has been developed and implemented in C for supporting efficient implementation of constraint-based automata. This language comes with a special optimization. A built-in preprocessing step for constraint-based automata extracts a finite state model for deterministic control of the automaton. Phrase recognition is then organized as a two phase process. In the first phase, lexical categories of input words are analyzed using the deterministic finite state control only. Once an admissible sequence of categories has been found by this filtering process, the relevant subsegment of the input is analyzed by the constraint-based automaton, envoking now constraint-solving and unification.

In two respects, then, the concept of a constraint-based automaton lies between finite state technology and Prolog/CLP. First, the syntactic form of transition rules is more restricted than a program clause of a constraint logic program. Second, when processing an input text file, classical finite state techniques are used as long as possible, and the real power of unification and constraint solving is only envoked once a suitable candidate sequence of input tokens has been found by the filtering process. The present work builts up on earlier theoretical investigations on automata with built-in unification [SG95].

The paper has the following structure. In Section 2 we describe how morphological information is treated in constraint-based automata. In Section 3 we specify the formal syntax of states and transition rules. We describe the control of constraint-based automata, and the two phase recognition process. Section 4 provides a formal model for the computation with constraint-based automata. Section 5 gives some empirical results

[4]E.g., using tries, lookup rates up to 13.000 words per seconds are reached on NeXT or Sun for the CISLEX system [GM94].

on the performance of constraint-based automata in our current implementation and corresponding programs written in ECL^{i}PS^e Prolog. An appendix is added where the constraint solver is described in detail.

2 Morphological variables and morphological constraints

In this section we explain which kind of constraints we use for constraint-based automata, and how we use them. Ignoring some peculiarities concerning the nature of constraints, the following techniques are well-known in the area of constraint programming. Nevertheless, a brief description seems inevitable if we want to explain the concept of a constraint-based automaton. We start with some information on the background dictionary.

Entries in electronic full form dictionaries for lexical categories like nouns, determiners, adjectives, verbs etc. are typically equipped with morphological information. This information stores which values of features like *number*, *gender* etc. can be associated with a given inflected form of a word. In general it is not possible to specify the set of possible values for each of the relevant features separately since there are many interdependencies. For example, the german determiner "*die*" can have both values *plural* and *singular* for *number*, in the singular case the gender is always *femininum*, while in the plural case all three german genders are possible. For this reason the morphological information that is attached to these entries has—modulo variants in the style of representation—the form of a finite set of tuples, each tuple representing one possible combination of values for a fixed sequence of features. E.g., the entry for the german determiner "*den*" in the CISLEX dictionary [GM94, Ma94] has the form

den *d.DET4*:**dmFy**:**dmMy**:**dmNy**:**aeMy**

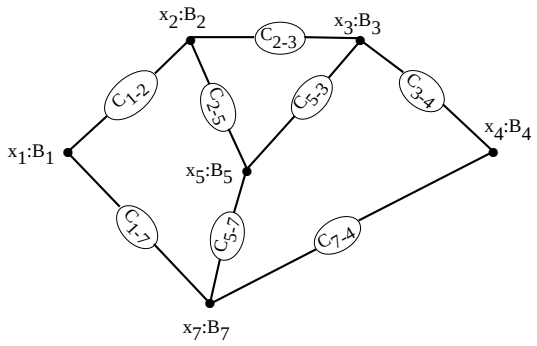
Each of the four character sequences in bold-face represents one possible quadruple of values for the features *case*, *number*, *gender*, *declination-type*, on the basis of an encoding that is not relevant here. In the sequel, such a finite sequence of tuples will be called a *box* of *morphological tuples*, for simplicity. We shall always assume[5] that all tuples of a given box have the same type, which means that their components represent values for the same sequence of features.

In the syntax for a constraint-based automata, variables may be used to refer to boxes of the form described above. These variables will be called *morphological variables*. When we use a constraint-based automaton for evaluating a given sequence of input words, the lexical look-up will bind morphological variables to the boxes that are associated with the input tokens in the dictionary. Once this instantiation has taken place, morphological variables behave essentially in the same way as the "finite domain variables" of modern CLP languages (e.g., [DS88]).

[5] In a few cases the morphological information that is attached to a given entry in the CISLEX dictionary is not of a completely homogeneous form. For the sake of clarity we shall not comment on this point which causes some additional technical difficulties in the actual handling of constraints.

In order to deal with grammatical agreement conditions, two types of constraints for morphological variables can be used in a constraint-based automaton. *Assignment constraints* have the form x: B where x is a morphological variable and B is box. Such a constraints may be used to bind x explicitly to B, without any reference to the background dictionary. *Coincidence constraints* have the form T: $x = y$. Here x and y are morphological variables and T is a sequence of features, such as, e.g., $\langle gender, number \rangle$. A constraint $\langle gender, number \rangle$: $x = y$, for example, expresses that the correct values of x and y must agree on gender and number. If new constraints are added, the assignment of boxes to morphological variables may be changed dynamically. Before we explain this mechanism we have to describe the global organization of constraints in more detail.

In a constraint-based automaton, constraints are organized in a *network*. Such a network can be represented in the form of a finite graph. Each vertice is labelled with a pair x: B where x is a morphological variable and B is a box which represents the set of possible values for x. The arcs of the graph are labelled with the coincidence constraints between these variables. The following figure gives an example. C_{i-j} stands for the set of coincidence constraints between variables x_i and x_j.



A network of constraints

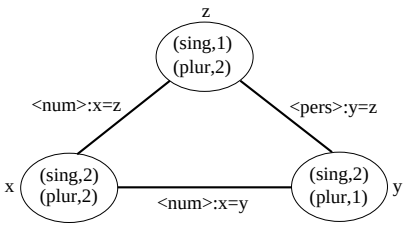
When lookin at the arcs of such a network one will often detect certain inconsistencies. Assume, for example, that box B_1 in our network is $\{\langle sing, 1\rangle, \langle plur, 1\rangle, \langle plur, 2\rangle\}$, that B_2 is $\{\langle sing, nominativ, 2\rangle, \langle plur, nominativ, 3\rangle\}$ and that C_{1-2} contains the constraint $\langle person \rangle$: $x_1 = x_2$. The constraint expresses that the correct values for x_1 and x_2 respectively must have the same value for "*person*". But then $\langle sing, 1\rangle$ cannot be the correct value for x_1 since there is no entry in B_2 where "*person*" has the value 1. For the same reason, x_1 cannot have the value $\langle plur, 1\rangle$, and x_2 cannot have the value $\langle plur, nominativ, 3\rangle$. Hence, by "firing" the constraint $\langle person \rangle$: $x_1 = x_2$ we will obtain the new assignments x_1: $\{\langle plur, 2\rangle\}$ and x_2: $\{\langle sing, nominativ, 2\rangle\}$.

With these new assignments, the subnet including the two nodes for x_1 and x_2 and the arc between these nodes is *consistent* in the following sense: for each selection of a possible value v_1 for x_1 we may choose a possible value v_2 for x_2 such that v_1 and v_2 satisfy all constraints for x_1 and x_2. A network of constraints is *arc-consistent* ([Ma77]) if each subnet with two nodes is consistent.

When we run a constraint-based automaton on a given sequence of input tokens, a

constraint solving mechanism is applied that maintains the arc-consistency of the network of morphological constraints that are accumulated during the analysis. A special-purpose constraint-solver for morphological constraints has been developed that removes inconsistencies in a given two-node subnet. In order to maintain arc-consistency of the whole network we use the algorithm AC_3 from [Ma77]. This algorithm, well-established in the constraint programming community, has worst-case complexity ed^3 where e denotes the number of variables and d is the cardinality of the largest domain (box) of a variable ([MF85]).

A net is called *globally consistent* if each selection of values $\vec{v}$ for a subset $\vec{x}$ of the set of all variables such that $\vec{v}$ respects all constraints between variables in $\vec{x}$ can be extended to a global selection of values for all variables such that all constraints in the net are satisfied. It is wellknown that arc-consistency does not necessarily imply global consistency. In particular it might be impossible in an arc-consistent net to choose even one selection of values for the variables that respects all constraints, as can be demonstrated with the following simple example.



Arc-consistency does not guarantee globally consistency

In general, NP-complete algorithms are needed to maintain global consistency in a network. For this reason most systems in constraint programming are based on the simpler notion of arc-consistency. In our practical work with constraint-based automata we did not find any situation where an inconsistency on the morphological level was not detected by the constraint-solver.

3 Syntax of Constraint-Based Automata

In this section we desribe the syntactic form of the states and transition rules of a constraint-based automaton. We begin with a definition of boxes and constraints. For the sake of transparency and generality we give generic definitions that do not depend on the morphology of a particular language.

3.1 Boxes, constraints and terms

Definition 3.1 Let $\mathcal{F}$ be a finite set of *features*, assume that each feature $f \in \mathcal{F}$ comes with a set of *possible values*, V_f. A *type* is a non-empty sequence of distinct features. Hence the set of types is finite. To each type $T = \langle f_1, \ldots, f_k \rangle$ we assign the

set $B_T := V_{f_1} \times \ldots \times V_{f_k}$ of *possible values of type* T. A *box* is a finite, *non-empty* set B of possible values of the same type T. This type T is called the type of box B.

It should be noted that the features in $\mathcal{F}$ may also characterize semantic properties, in principle. But the constraints that we introduce below are not tuned to semantic information, for two simple reasons. Still, the amount of semantic information that can be found in large-scale electronic dictionaries is rather limited. Furthermore, the integration of general techniques for the treatment of semantic information would presuppose that this kind of information is standardized at least in some way, and still it is far from clear how such a standard should look like.

Definition 3.2 A *morphological constraint* is an expression of the form $x\colon B$ or $T\colon x = y$ where B is a box and T is a type. Constraints of the form $x\colon B$ are called *assignment constraints*, constraints of the form $T\colon x = y$ are called *coincidence constraints.*[6]

Morphological onstraints can be used to formulate conditions on morphological agreement. In order to support the handling of other linguistic phenomena, such as, e.g., extraposition, general long-distance dependencies, and computation of semantic representation, the states of a constraint-based automaton are first-order terms like in Prolog. Transitions will be based both on unification and on constraint solving.

Let Σ be a first-order signature, i.e., a set of free function symbols[7], all of fixed arity, and let *Var* be a countably infinite set of variables.

Definition 3.3 The set of *terms* is the smallest set of expressions that is closed under the following construction rules.

- Every variable and every constant $a \in \Sigma$ is a term.
- If $t_1, \ldots, t_n$ are terms, and if $f \in \Sigma$ is an n-ary function symbol, then $f(t_1, \ldots, t_n)$ is a term.

Note that in the syntax of terms, no distinction is made between logical variables and morphological variables. In practice, type checking is left to the unification and constraint solving subprocedures.

Definition 3.4 A *constraint* is a morphological constraint or a *locical constraint*, i.e., an equality $t_1 = t_2$ between terms. A *constraint set* is a finite set C of constraints. If C denotes a constraint set, then $C_=$ (resp. C_a, C_c) denotes the set of all logical (assignment, coincidence) constraints in C.

[6] In our actual implementation, additional constraints of the form $x\colon T$ may be used for the convenience of the user, where T is a type. Internally, these constraints are equivalent to assignment constraints $x\colon B_T$.

[7] The set Σ, the set of features, and the set of possible values of features are assumed to be disjoint.

3.2 States and Transition Rules

A constraint-based automaton has the surface structure of an ordinary finite state automaton, which means that it has one start state, a finite number of final states, and a finite number of transition rules between states. The "states" of a constraint-based automaton, however, may be rather complex objects. We shall now specify the syntactic form of the start state, the final state and transition rules. Two points will be postponed for simplicity:

1. In practice, each state and each transition rule is equipped with some additional information that is used to control execution of the automaton.
2. Constraint-based automata possibly have a finite number of subordinate constraint-based automata, and there are special rules for calling subautomata.

Both the form of and the role of the control declaration, and the organization of subautomata will be explained below.

Start and **final states** of a constraint-based automaton are expressions of the form s, C where s is a term and C is a finite (possibly empty) set of morphological constraints.

Lexical transition rules have the form $h \stackrel{\mathrm{Cat}(\mathrm{x_1},\ldots,\mathrm{x_k})}{\longrightarrow} t, C$ where h, and t are terms, Cat is a lexical category, $x_1, \ldots, x_k \in Var$, and C is a finite set of morphological constraints.

Empty transition rules have the form $h \rightarrow t, C$ where h, and t are terms, and C is a finite set of morphological constraints.

We generally assume that a **background dictionary** $\mathcal{D}$ is given with entries of the form $w : Cat(B_1, \ldots, B_k)$ where w is a (possibly inflected form of a) word, Cat is a lexical category, and $B_1, \ldots, B_k$ is a sequence of boxes of distinct type.

Because the syntax does not take care of types, some caution is necessary when writing the transition rules of a constraint-based automaton. For example, the evaluation of a coincidence constraint T: $x = y$ will only succeed once both x and y are instantiated and bound to boxes of appropriate types T_x and T_y where T is a subtype of both types. The evaluation of an assignment constraint x: B will fail if x is bound to a box B' such that B and B' have distinct type, or if x is bound to a non-variable term.

Control declarations and deterministic finite state control. Constrained-based automata use a rigid control mechanism that makes it possible to execute the automaton temporarily exactly like an ordinary deterministic finite state automaton. In order to explain this control principle we have to describe now the full syntactic form of states and transition. The start state, each of the final states and each transition rule is labelled with a number m. In addition, start state and lexical transition rules are equipped with a control declaration of the form

$$[Cat_1 : n_1, \ldots, Cat_k : n_k]$$

where each Cat_i denotes a lexical category and each n_i is a number of a transition rules or a final state ($1 \leq i \leq k$). The last entry may also be a plain number n_k of a finite state, in which case the declaration has the form $[Cat_1 : n_1, \ldots, Cat_{k-1} : n_{k-1}, n_k]$.

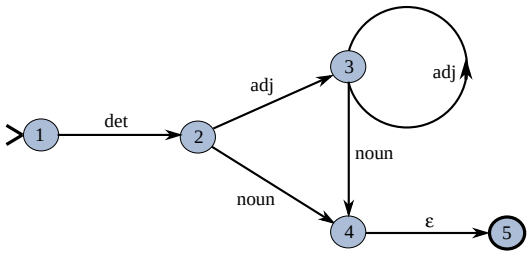
The *naive* idea of how to use this information is the following. Once the evaluation of the present (state or) transition rule is finished, the machine will check if the next input token is of category Cat_1, or Cat_2, etc. If (i is the smallest number such that) the next input token is of category Cat_i, then the automaton will use rule number n_i for evaluating this input token.[8] If the last entry is a plain number n_k, then the automaton will switch to the final state n_k in all cases where the previous entries do not match. If the last entry of the declaration has the "conditional" form $Cat_k : n_k$, and if the next input token fits none of the categories mentioned in the control declaration, then the evaluation fails.

The *real use* of the information is more powerful. Given the numbers and control declarations of a constraint-based automaton $\mathcal{A}$ we assign to $\mathcal{A}$ a **finite state control companion** $\mathcal{A}_{\text{FSC}}$ in the following way. The states of $\mathcal{A}_{\text{FSC}}$ are the (numbers of) rules of $\mathcal{A}$. Moreover, $\mathcal{A}_{\text{FSC}}$ has a transition $n \xrightarrow{\text{Cat}} m$ iff the control declaration of rule n of $\mathcal{A}$ has an entry $Cat : m$. A final entry n_k in the control declaration of rule number m is translated into an empty (vacuous, silent) transition $m \xrightarrow{\epsilon} n_k$ to the final state n_k.

Example 3.5 The following five lines represent a constraint-based automaton $\mathcal{A}$ that may be used for recognition of german noun phrases of the simple form *det adj* noun*. The start state has number 1, the final state number 5. Letters g, k, n stand for the features *gender*, *case*, *number*.

1.	S		$[det : 2]$
2.	$S \xrightarrow{\text{det}(x)} S(x)$		$[adj : 3, noun : 4]$
3.	$S(x) \xrightarrow{\text{adj}(y)} S(x)$	$\{\langle g, k, n\rangle : x = y\}$	$[adj : 3, noun : 4]$
4.	$S(x) \xrightarrow{\text{noun}(y)} S(x)$	$\{\langle g, k, n\rangle : x = y\}$	$[5]$
5.	$S(x)$		

The companion $\mathcal{A}_{\text{FSC}}$ of $\mathcal{A}$ is the following deterministic finite state automaton:



On the basis of the companion $\mathcal{A}_{\text{FSC}}$, the search for correct phrases proceedes in two steps. In the first phase, we only use $\mathcal{A}_{\text{FSC}}$ until a suitable candidate sequence of

[8] In our actual implementation, a decision to use rule n_i for the next step of the evaluation is not modified at a later point since no backtracking is used. In principle, this can lead to unwanted failure since the wrong part-of speech category might be selected first for lexically ambigous input tokens. For the analysis of german corpora this was not a problem.

input tokens $w_1 \ldots, w_k$ has been found. In the above example, every sequence of words $w_1 \ldots, w_{k-1}, w_k$ will be considered as a candidate where the dictionary $\mathcal{D}$ has a determiner of the form w_1, adjectives of the form $w_2, \ldots, w_{k-1}$, and a noun of the form w_k. Once a candidate sequence $w_1 \ldots, w_k$ has been found, $\mathcal{A}$ will be used in the second phase to process $w_1 \ldots, w_k$ using constraint-solving and unification. Procedurally, the transition rules of a constraint-based automaton $\mathcal{A}$ are treated like program clauses of a CLP program, see Section 4 and the appendix for details. Note that it is only in this second phase where morphological conditions are checked.

The advantage of the two phase recognition process is twofold. First, because of the deterministic control no backtracking mechanism is needed.[9] Second, as long as the automaton does not enter into the second phase, no renaming of transition rules is necessary. This is one important contrast to Prolog, where each rule has to be renamed, using a new set of variables, before it may be used in the derivation, and a good amount of computation time is spent just for this task. With constraint-based automata, this process starts only once we enter phase two.

Constraint-based automata with subautomata. In order to facilitate the design of large constraint-based automata, we introduced a new type of transition rule for calling a subautomaton. Each subautomaton has a unique finite state. Rules calling a subautomaton have the form $h \xrightarrow{(\mathcal{B}:s,f)} t, C$ where h, s, f, and t are terms, and C is a finite set of morphological constraints. $\mathcal{B}$ is the name of the subautomaton that is called, and s and f are Σ-terms that can be unified with the start state and the final state of $\mathcal{B}$ respectively. The idea behind these rules is the following. When $\mathcal{B}$ is called, s and the (Σ-term of the) start state of $\mathcal{B}$ are unified. Since h and s may share variables, it is possible to pass relevant information from h to the start state (term) of $\mathcal{B}$ through unification. When the call to $\mathcal{B}$ is finished, unification of the final state (term) of $\mathcal{B}$ with f may be used to lift information to the state (term) t.

Since we did not want to abandon the deterministic finite state control principle and the two phase recognition process, the set of all subautomata of a constraint-based automaton is organized in a strictly hierarchical, non-recursive way.

4 The formal model

In order to give a better picture of the evaluation with constrained based automata we shall now introduce a formal model. First, we shall define what it means that an automaton accepts a given input sequence. We shall then introduce a ground version of a constraint-based automaton that comes with its own simple notion of acceptance. It will be shown that both notions of acceptance coincide. To begin with, we have to define the solution domain for constraints. Readers that are familiar with rational trees can safely ignore the following definitions.

A *position* is a finite sequence $\langle n_1, \ldots, n_k \rangle$ of natural numbers. A *tree domain* is a set t of positions that is closed under prefixes and has the following property: if

[9] The price that has to be paid is that grammars have to be written in a rigid format.

$\langle n_1, \ldots, n_k \rangle \in t$ and $n_k > 0$, then $\langle n_1, \ldots, n_k - 1 \rangle \in t$. A leaf of a tree domain t is a maximal position, i.e., a position $\langle n_1, \ldots, n_k \rangle$ such that t does not have any element of the form $\langle n_1, \ldots, n_k, n_{k+1} \rangle$.

Definition 4.1 Let Σ be a signature, i.e., a set of function symbols. An **m-tree** over Σ is a pair $\langle t, D \rangle$ where t is a tree domain and D is a labelling function, i.e., a function with domain t such that for all $\pi \in t$

1. $D(\pi)$ is an n-ary element of Σ if π has $n > 0$ successors,
2. $D(\pi)$ is a constant of Σ or a possible value of some type (Def. 3.1) if π is a leaf.

Subtrees are defined as usual. An m-tree is **rational** if it contains only a finite number of distinct subtrees. With $\mathcal{T}$ we denote the set of all rational m-trees over the (fixed) signature Σ.

The set $\mathcal{T}$ of rational m-trees will be the solution domain for constraints. An **assignment** is a mapping $\alpha : \mathit{Var} \to \mathcal{T}$. Each assignment will be identified with its unique homomorphic extension to the set of all terms. We write t^α for the image of the term t under the assignment α.

Definition 4.2 An assignment α **satisfies** an assignment constraint x: B iff $x^\alpha \in B$. An assignment α satisfies a coincidence constraint T: $x = y$ if x^α and y^α are boxes that have the subtype T and if x^α and y^α coincide on all features in T. An assignment α satisfies a logical constraint $x = y$ if $x^\alpha = y^\alpha$. An assignment α satisfies a constraint set C if α satisfies each $c \in C$.

In order to simplify the discussion we shall restrict the following considerations to constrained based automata with (start state, final states and) lexical transition rules only. We shall also ignore matters of control. With more technical ballast, the formal model can be lifted to automata with rules that call subautomata, and to rules with control declaration.

In the sequel, let $\mathcal{A}$ denote a constrained based automaton, and let $\mathcal{D}$ denote a fixed background dictionary.

Definition 4.3 A **closed path** of $\mathcal{A}$ is given by a sequence

$$\pi = (t_0, C_0), R_1, \ldots, R_k, (h_{k+1}, C_{k+1})$$

where t_0, C_0 is the start state of $\mathcal{A}$, each R_i is a transition rule $h_i \overset{\mathrm{Cat_i(x_1^i,\ldots,x_{m_i}^i)}}{\longrightarrow} t_i, C_i$ of $\mathcal{A}$, for $i = 1, \ldots, k$, and h_{k+1}, C_{k+1} is a final state of $\mathcal{A}$. In more detail, states and transitions of π are assumed to be renamed in such a way that all members of π are variable-disjoint. The closed path π is a *candidate path* for the sequence of words $w_1 \ldots w_k$, given $\mathcal{D}$, if $\mathcal{D}$ contains entries $w_i : \mathit{Cat}_i(B_1^i, \ldots, B_{m_i}^i)$, for $i = 1, \ldots, k$.

Let π be a candidate path for $w_1 \dots w_k$. The **constraint set assigned to π and $w_1 \dots w_k$** is

$$C_\pi := \bigcup_{i=0}^{k} \{t_i = h_{i+1}\} \cup \bigcup_{i=0}^{k+1} C_i \cup \bigcup_{i=1}^{k} (\bigcup_{j=1}^{m_i} x_j^i{:}\ B_j^i).$$

If an assignment α satisfies C_π, then it unifies consecutive heads and tails of transition rules of π, it satisfies all morphological constraints occurring in π and it selects morphological values for the words $w_1 \dots w_k$ that are possible according to $\mathcal{D}$.

Definition 4.4 The input sequence $w_1, \dots, w_k$ is **accepted by** $(\mathcal{A}, \mathcal{D})$ if $\mathcal{A}$ has a closed path π that is a candidate path for $w_1, \dots, w_k$, given $\mathcal{D}$, such that the constraint set C_π assigned to π has a solution.

Given $\mathcal{D}$, each state and transition rule of $\mathcal{A}$ can be thaught of representing a possibly infinite collection of ground states and transition rules in the following sense.

Definition 4.5 The set of **admissible ground instances** of a state t, C is

$$\{t^\alpha \mid \alpha : Var \to \mathcal{T}, \alpha \text{ solves } C\} \subseteq \mathcal{T}.$$

The set of admissible ground instances of a rule $h \overset{\mathrm{Cat(x_1,\dots,x_m)}}{\longrightarrow} t, C$ is the set of all transition rules of the form $h^\alpha \overset{w}{\longrightarrow} t^\alpha$ where the assignment α solves C and where $\mathcal{D}$ contains an entry $w : Cat(B_1, \dots, B_m)$ such that $x_i^\alpha \in B_i$, for $i = 1, \dots, m$.

Definition 4.6 The **ground version** $Gr_\mathcal{D}(\mathcal{A})$ of the constraint-based automaton $\mathcal{A}$, given $\mathcal{D}$, contains all admissible ground instances of the start state and of the final states of $\mathcal{A}$, and of the transition rules of $\mathcal{A}$. The instances of the start state of $\mathcal{A}$ are the start states of $Gr_\mathcal{D}(\mathcal{A})$, the instances of the final states of $\mathcal{A}$ are the final states of $Gr_\mathcal{D}(\mathcal{A})$.

As we see in the example below, $Gr_\mathcal{D}(\mathcal{A})$ can be represented as a labelled directed graph where vertices are rational m-trees, edges represent transitions that are triggered by words (full forms) occurring in the dictionary $\mathcal{D}$.

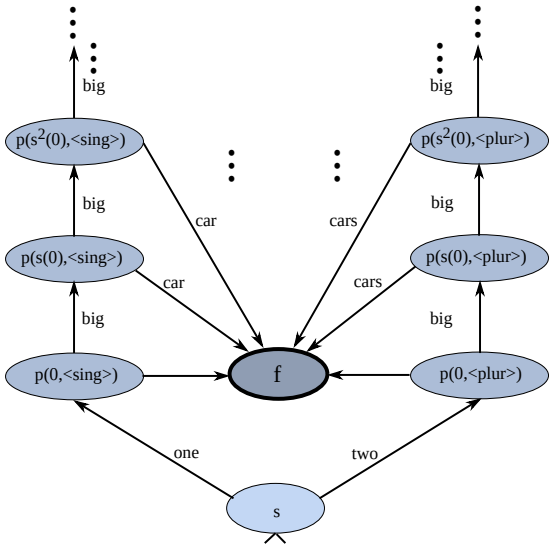
An **admissible path** of $Gr_\mathcal{D}(\mathcal{A})$ is a sequence of transitions, with labels $w_1, \dots, w_k$, say, leading from a start state of $Gr_\mathcal{D}(\mathcal{A})$ to a final state.

Definition 4.7 The sequence $w_1, \dots, w_k$ is *licensed by* $Gr_\mathcal{D}(\mathcal{A})$ iff $Gr_\mathcal{D}(\mathcal{A})$ has an admissible path with labels $w_1, \dots, w_k$.

Example 4.8 Let $\mathcal{A}$ be the automaton

1. s
2. $s \overset{\mathrm{det}(x)}{\longrightarrow} p(0, x)$ $\quad \{x{:}\ \{\langle sing\rangle, \langle plur\rangle\}\}$
3. $p(x, y) \overset{\mathrm{adj}}{\longrightarrow} p(s(x), y)$
4. $p(x, y) \overset{\mathrm{noun}(z)}{\longrightarrow} f$ $\quad \{\langle n\rangle{:}\ y = z\}$
5. f

with start s and final state f. Assume that $\mathcal{D}$ contains the entries *one:* $det(\langle sing \rangle)$, *two:* $det(\langle plur \rangle)$, *big:* adj, *car:* $noun(\langle sing \rangle)$, *cars:* $noun(\langle plur \rangle)$. Then the following figure represents $Gr_{\mathcal{D}}(\mathcal{A})$. States that cannot be reached from the start state s are omitted for simplicity.



In this example, each sequence of the form "*one (big)* car*" or "*two (big)* cars*" is licensed by $Gr_{\mathcal{D}}(\mathcal{A})$.

Theorem 4.9 *A sequence of input tokens $w_1, \ldots, w_k$ is licensed by $Gr_{\mathcal{D}}(\mathcal{A})$ iff $w_1, \ldots, w_k$ is accepted by $(\mathcal{A}, \mathcal{D})$.*

Proof. Assume that $w_1, \ldots, w_k$ is accepted by $(\mathcal{A}, \mathcal{D})$ on the closed path

$$\pi = (t_0, C_0), R_1, \ldots, R_k, (h_{k+1}, C_{k+1}).$$

Since π is a candidate path for $w_1, \ldots, w_k$, each R_i is a transition rule of the form $h_i \overset{Cat_i(x_1^i, \ldots, x_{m_i}^i)}{\longrightarrow} t_i, C_i$ where $\mathcal{D}$ has entries $w_i : Cat_i(B_1^i, \ldots, B_{m_i}^i)$, for $i = 1, \ldots, k$. By assumption, C_π has a solution α. This implies that ${x_j^i}^\alpha \in B_j^i$, for $i = 1, \ldots, k$ and $j = 1, \ldots, m_i$. It follows that

$$h_1^\alpha \overset{w_1}{\longrightarrow} t_1^\alpha, \ldots, h_k^\alpha \overset{w_1}{\longrightarrow} t_k^\alpha$$

is an admissible path of $Gr_{\mathcal{D}}(\mathcal{A})$ since $h_1^\alpha = t_0^\alpha$ (resp. $t_k^\alpha = h_{k+1}^\alpha$) is a start (resp. final) state of $Gr_{\mathcal{D}}(\mathcal{A})$ and since also $t_i^\alpha = h_{i+1}^\alpha$, for $i = 1, \ldots k-1$. Hence $w_1, \ldots, w_k$ is licensed by $Gr_{\mathcal{D}}(\mathcal{A})$.

Conversely, if $w_1, \ldots, w_k$ is licensed by $Gr_{\mathcal{D}}(\mathcal{A})$, let $R_1^0, \ldots, R_k^0$ be an admissible path of $Gr_{\mathcal{D}}(\mathcal{A})$ where the ground rule R_i^0 has the form $h_i^{\alpha_i} \overset{w_i}{\longrightarrow} t_i^{\alpha_i}$, for $i = 1, \ldots, k$. Here $\mathcal{A}$ has a corresponding rule R_i of the form $h_i \overset{Cat_i(x_1^i, \ldots, x_{m_i}^i)}{\longrightarrow} t_i, C_i$ where the assignment α_i solves C_i and where $\mathcal{D}$ has an entry $w_i : Cat_i(B_1^i, \ldots, B_{m_i}^i)$ such that ${x_j^i}^{\alpha_i} \in B_j^i$,

for $i = 1, \ldots, k$ and $j = 1, \ldots, m_i$. Moreover, if (t_0, C_0) denotes the start state of $\mathcal{A}$, then there exists an assignment α_0 that satisfies C_0 such that $t_0^{\alpha_0} = h_1^{\alpha_1}$, and there exists a final state (h_{k+1}, C_{k+1}) of $\mathcal{A}$ and an assignment α_{k+1} that satisfies C_{k+1} such that $h_{k+1}^{\alpha_{k+1}} = t_k^{\alpha_k}$. It follows that

$$\pi = (t_0, C_0), R_1, \ldots, R_k, (h_{k+1}, C_{k+1})$$

is a candidate path for $w_1 \ldots w_k$. Without loss of generality we may assume that start state, final state, and transition rules are pairwise variable disjoint. Hence we may assume that $\alpha_0 = \ldots = \alpha_{k+1} =: \alpha$. But then the assignment α satisfies the constraint set C_π assigned to π and $w_1 \ldots w_k$, given $\mathcal{D}$, which shows that $w_1, \ldots, w_k$ is accepted by $\mathcal{A}$. □

5 Preliminary empirical evaluation

In order to support the implementation of constraint-based automata, a miniature programming language has been implemented in C. The source code contains ca. 8.000 lines. The language accepts as input Prolog-style transitions rules of the form given above (rules may call a subautomaton). Given the program for a constraint-based automaton, a built-in preprocessing step computes the finite state control companion before evaluating the given text file.

The table given below summarizes the results of a preliminary empirical evaluation where we compared the computation time for extracting all phrases of a particular form from a text of 200kByte (16.500 words). The experiments run on a SUN Sparc 10, times are in seconds.

The six lines of the table describe independent experiments were we extracted a particular set of phrases in each case. In the first experiment, we searched for an empty category, in other words, for a simple non-existing phrase. The second line gives the results where we extracted words that belong to a finite set of lexical categories. The third line describes the time to extract all noun phrases of the simple form "determiner followed by a finite sequence of adjectives followed by a noun". The next experiments used constraint-based automata with subautomata, the numbers in brackets give the nesting degree. PP(1) (resp. PP(9)) stands for some automaton for recognizing prepositional phrases with one (resp. nine) level(s) of subordinate automata. NP(3) stands for an automaton for noun phrases with three levels of subordinate automata.

The first column gives the times that were needed using a grammar with constraints in ECL^{i}PS^e Prolog. One optimization has been built-in already here. Instead of using a global dictionary with alle the words of the text (which would lead to much worse results), each sentence is processed with a small dictionary that containts just all the words of the sentence in order to make the lexical look-up more efficient. The times in the first (second) column include (exclude) the computation of the local dictionaries. The third and fourth column give the processing times using constraint-based automata, counting (column 3) or disregarding (column 4) the time for printing correct phrases on the screen. The last two columns compare the times of columns 1 and 3 respectively columns 2 and 4.

	1. ECL^{i}PS^e PROLOG	2. ECL^{i}PS^e - temp. dict.	3. Constraint-B. Automaton	4. CBA - print time	1:3	2:4
Empty Cat.	775	28	1	1	775	30
Particle	780	31	5	3	150	10
det adj* n	830	83	27	17	30	5
PP(1)	815	68	10	7	80	10
NP(3)	890	140	17	10	50	14
PP(9)	990	240	16	14	60	17

6 Conclusion

We have introduced the concept of a constraint-based automaton. With this new notion, recognition of phrases is based on some form of computation that lies between finite state techniques and logic programming with constraints. Our aim was to create a tool for efficient recognition of phrases, supporting control of morphological agreement conditions. First evaluation results indicate that in fact a reasonable gain in efficiency could be obtained if compared with CLP implementations. In our actual work, we see various further possibilities for optimization, both on the conceptual level and on the level of the implementation. To some extend, the possible optimizations depend on the structure of phrases that we want to extract. Empirically, the extraction of "short" noun phrases in german corpora shows that there are some characteristic sequences of boxes such that most of the correct phrases come with one of these sequences. If we use a variant of the dictionary where boxes are replaced by numbers, and if we encode these characteristic sequences as sequences of numbers, then it is possible to "built-in" these sequences and to avoid constraint solving in most of the cases. On the implementation side, the internal renaming (copying) of rules—which is actually necessary during phase two—can probably be reduced to a minimum, since in general each rule is only used a restricted number of times (typically once, or twice) during the analysis of a given sequence of input tokens.

References

[Col84] A. Colmerauer, "Equations and inequations on finite and infinite trees," in: *Proc. 2nd Int. Conf. on Fifth Generation Computer Systems*, 1984, pp. 85-99.

[Col90] A. Colmerauer, "An introduction to PROLOG III," *C. ACM* **33**, 1990, pp.69–90.

[Co90] B. Coutois, "Un système de dictionaires èlectroniques pour les mots simples du francais." in *Language Francais* **78**, 1990.

[DS88] M. Dincbas, P. Van Hentenryck, H. Simonis, A. Aggoun, "The Constraint Logic Programming Language CHIP." in *Proc. of the 2nd International Conference on Fifth Generation Computer Systems*, 1988, pp. 249-264.

[EC95] "ECL^{i}PS^e 3.5 User Manual", ECRC Munich, 1995.

[GrP] M. Gross, D. Perrin (Eds.), "Electronic Dictionaries and Automata in Computational Linguistics," Springer LNCS 377, 1989.

[GM94] F. Guenthner and P. Maier: *Überblick zum CISLEX-Wörterbuch-System*, CIS-Bericht, 1994, *Lexikographica*, to appear.

[Ma77] A.K. Mackworth, "Consistency in Networks of Relations," *AI Journal* **8** (1), 1977, pp. 99-118.

[MF85] A.K. Mackworth, E.C. Freuder, "The Complexity of some Polynomial Network Consistency Algorithms for Constraint Satisfaction Problems," *Artificial Intelligence* **25**, 1985, pp. 65-74.

[Ma94] P. Maier: *Lexikon und Lemmatisierung*, CIS-Bericht-95-84.

[Sc96] A. Schiller, "Multilingual Finite-State Noun Phrase Extraction," In: *Proceedings of the ECAI 96 Workshop "Extended Finite State Models of Language"*, A. Kornai (Ed.), pp. 65-69.

[SG95] K.U. Schulz, D.M. Gabbay, "Logic Finite Automata," In: *Applied Logic: How, What and Why,* L. Pólos, M. Masuch (Eds.), Kluwer Academic Publishers, 1995, pp. 237-285.

[Si93] M. Silberztein, "Dictionaires électroniques et analyse automatique de textes," Masson, Paris, 1993.

[Ta95] P. Tapanainen, "RXRC Finite-State Compiler," Technical Report MLTT-020, Rank Xerox Research Center, Meylan, France, 1995.

Appendix: The Constraint Solver

In the actual implementation, the finite state control companion $\mathcal{A}_{\text{FSC}}$ of a constraint-based automaton $\mathcal{A}$ acts as a transducer. Recall that the sequence of states of $\mathcal{A}_{\text{FSC}}$ that are visited in a successful run of $\mathcal{A}_{\text{FSC}}$, for input $w_1 \dots w_k$, say, represents a closed path π of $\mathcal{A}$ which is a candidate path for $w_1 \dots w_k$, given the background dictionary $\mathcal{D}$. In our implementation, the output of $\mathcal{A}_{\text{FSC}}$ is just the constraint set C_π associated with π (Definition 4.3). In order to check if $w_1 \dots w_k$ is accepted on π it remains to decide solvability of C_π. In this appendix we shall describe the treatment of C_π.

In the sequel, constraint sets C are described as triples $(C_=, C_a, C_c)$ where $C_=$ denotes the set of logical constraints (equations) of C, C_a denotes the assignment constraints of C and C_c denotes the coincidence constraints of C. Empty components are omitted.

Definition 6.1 A constraint set $(C_=, C_a)$ is an *assignment table*, if the following conditions are satisfied:

1. $C_=$ contains only equations $x = t$, where x is a variable and t is an arbitrary term, and equations $t_1 = t_2$, where t_1 and t_2 are *non-variable* terms. All left-hand sides of equations in $C_=$ are distinct.
2. $C_=$ does not have a cyclic subsystem, i.e., a subsystem of the form $t_1 = t_2, \dots, t_{n-1} = t_n, t_n = t_1$,
3. a variable that occurs as a left-hand side of an equation in $C_=$ does not occur in C_a and vice versa. A variable has at most one occurrence in C_a.

Definition 6.2 A constraint set $(C_=, C_a, C_c)$ is in *solved form* if $(C_=, C_a)$ is an assignment table and if the following conditions hold:

4. each variable x occurring in C_c also occurs in C_a.
5. whenever C_c contains a constraint T: $x = y$, then T is a subtype of the boxes B_x, B_y, where x: B_x and y: B_y are the unique assignment constraint for x and y in C_a (compare 3, 4) respectively.
6. (C_a, C_c) represents a consistent network.

If conditions 1-5 are satisfied and (C_a, C_c) is an arc-consistent network we say that $(C_=, C_a, C_c)$ is in *arc solved form*.

We distinguish three types of variables in C: variables occurring in C_a are called morphological variables. The left-hand sides of the equations in $C_=$ are called dependent variables, and the remaining variables of C (which can only occur in the right-hand sides of the equations of $C_=$) are called parameter variables.

Theorem 6.3 *Let C be a constraint system in solved form. Then there exists an assignment α that satisfies C.*

Proof. (Sketch) Since (C_a, C_c) represents a consistent net, we can choose an appropriate value v_x for all morphological variables x such that all constraints in (C_a, C_c) are satisfied when we assign v_x to x. Next, parameter variables are mapped to arbitrary m-trees. It remains to assign appropriate values to the dependent variables such that the equations in $C_=$ are satisfied. We iterate the process where we replace the variables occurring in the right-hand sides of the equations in $C_=$ by their chosen values (for morphological variables and parameter variables) or by the right-hand sides of their defining equations (for dependent variables). Infinite iteration of this process leads to m-trees on the right-hand sides. The limit equations can be considered as an assignment of m-trees to dependent variables, and it is easily seen that the resulting assignment on all variables of C represents a solution of $C_=$ and hence of C. □

Lemma 6.4 *Let $(C_=, C_a, C_c)$ be a constraint system in arc solved form. If (C_a, C_c) is solvable, then also $(C_=, C_a, C_c)$ is solvable.*

We shall now describe a procedure that transfers a given constraint set C into an equivalent constraint set in arc solved form. The procedure may fail, indicating (type inconsistencies or) unsolvability of C. With a straightforward modification of the last step we could turn this into a solved form procedure, which would yield a decision procedure for solvability. It is just for efficiency reasons that we compute arc solved forms.

The *representant* of a term t with respect an assignment table $(C_=, C_a)$ is either t itself if t does not occur as the left-hand side of an equation in $C_=$, or it is the representant of t' if $C_=$ contains an equation $t = t'$. Conditions 1 and 2 of Definition 6.1 ensure that each term has a unique representant. Note that, in general, subterms of representants are not themselves representants.

Arc-Solved Form Procedure

The input is a constraint set $(C_a, C_=, C_c)$ such that every variable occurring in C_C also occurs in C_a. The procedure has two phases. In the first phase, $(C_a, C_=)$ is transformed into an assignment table $(C_a^1, C_=^1)$ such that $(C_a, C_=)$ and $(C_a^1, C_=^1)$ have the same solutions. As a side effect, new constraints may be added to C_c which means that we have may obtain an extension C_c^1 of C_c. The process may also fail, indicating unsolvability. In the second phase, $(C_a^1, C_=^1, C_c^1)$ is transformed into an equivalent arc solved form. Again this step may fail.

First Phase. Assume that we have reached a system $(S_=, S_a, D_a, D_=, D_c)$, starting from $(\emptyset, \emptyset, C_a, C_=, C_c)$, such that $(S_=, S_a)$ is an assignment table. If both D_a and $D_=$ are empty, then $(C_a^1, C_=^1, C_c^1) := (S_=, S_a, D_c)$ is the output of phase 1. In the other case,

I. Let x: $B \in D_a$. We compute the representant t of x with respect to $(S_=, S_a)$.

1. If t is a non-variable term, then we stop with failure ("box-term clash").
2. If $t := y$ is a morphological variable of $(S_=, S_a)$, let y: $B' \in S_a$.
   (a) If B and B' have distinct domain type, then we stop with failure.
   (b) If B and B' have the same domain type, then we compute $B'' := B \cap B'$.
      i. If $B'' = \emptyset$, then we stop with failure.
      ii. If $B'' \neq \emptyset$ we replace y: B' in S_a by y: B''.
3. If $t = z$ is a variable, but not a morphological variable, then we add z: B to S_a.

II. Assume now that $D_a = \emptyset \neq D_=$. Let $t_1 = t_2 \in D_=$. Both sides are replaced by their representants with respect to $D_=$. Let $t_1' = t_2'$ be the equation that is obtained. In the following cases, we stop with failure:

4. If t_1' and t_2' are morphological variables of $(S_=, S_a)$, with boxes of distinct type,
5. If t_1' is a morphological variable and t_2' is a non-variable term, or vice versa.
6. If t_1' and t_2' are non-variable terms with distinct topmost function symbols,

In the remaining case, we proceed as follows.

7. If t_1' and t_2' are morphological variables of $(S_=, S_a)$, with boxes of the same type T, then we add T: $t_1' = t_2'$ to D_c.
8. If $t_1' = y$ is a morphological variable and $t_2' = x$ is a variable, but not a morphological variable, then we add $x = y$ to $S_=$.
9. Similarly, if $t_2' = y$ is a morphological variable and $t_1' = x$ is a variable, but not a morphological variable, then we add $x = y$ to $S_=$.
10. If $t_1' = x$ is a variable, but not a morphological variable, and t_2' is a term, then we add $x = t_2'$ to $S_=$.
11. If t_1' is a non-variable term and $t_2' = x$ is a variable, but not a morphological variable, then we add $x = t_1'$ to $S_=$.
12. If t_1' has the form $f(r_1, \ldots, r_n)$ and t_2' has the form $f(s_1, \ldots, s_n)$, then we add the equation $f(r_1, \ldots, r_n) = f(s_1, \ldots, s_n)$ to $S_=$, and we add the equations $r_1 = s_1, \ldots, r_n = s_n$ to $D_=$.

Since new equations may be created during the first phase (see 12), termination is perhaps not obvious. But only subterms of input terms may occur in equations. In situation 12, two different subterms are identified (i.e., they receive the same representant) according to $S_=$. Hence the number of different subterms in the input system puts a bound on the maximal number of applications of rule 12.

Proposition 6.5 *The first phase terminates. If* $(C_a, C_=, C_C)$ *is solvable, the first phase does not fail. If* $(C_a^1, C_=^1, C_c^1)$ *is the output, then* $(C_a^1, C_=^1)$ *is an assignment table.*

Second Phase. Let $(C_a^1, C_=^1, C_c^1)$ be the output of the first phase. We update each coincidence constraint of C_c^1, replacing each variable by its representant with respect to $C_=^1$. Let D_c^1 denote the new set of constraints.

I. For each T: $x = y \in D_c^1$ we consider the assignments x: B_x and y: B_y in C_a^1. If T is not a subtype *both* of the type of B_x and B_y, then we stop with failure.

II. If part I succeeds, we check arc-consistency of the network $(C_a^1, C_=^1, D_c^1)$, removing arc-inconsistent values in the domains (boxes) associated with morphological variables, as indicated in Section 2. This step may fail if an empty domain is found. In the other case, we obtain the arc-consistent net (C_a^2, C_c^1). Then $(C_a^2, C_=^1, C_c^1)$ is the output of the algorithm.

Theorem 6.6 *If the arc solved form procedure fails for input* C*, then* C *is unsolvable. If the arc solved form procedure does not fail for input* C*, the output* $(C_a^2, C_=^1, C_c^1)$ *is in arc-solved form. If the net* (C_a^2, C_c^1) *is consistent, then* C *is solvable.*