The Fourth Symposium on Natural Language Processing 2000

AN HPSG FORMALISM FOR IMPLEMENTING A JAVA PARSER*

VLADO KEŠELJ
Department of Computer Science, University of Waterloo, Waterloo, ON N2L 3G1, Canada

We present a new HPSG formalism used to implement a Java parser for Head-Driven Phrase Structure Grammars (HPSGs), named Stefy. The parser is used in an Internet information retrieval system. The specification of the formalism is presented and the reasons for introducing this approach are explained. We also present the unification algorithm and discuss its complexity.

Key words: HPSG, unification, parsing, information retrieval.

1. INTRODUCTION

This paper describes the underlying formalism of the parser Stefy—a parser for natural languages (NL), implemented in the programming language Java, and based on the HPSG (Head-Driven Phrase Structure Grammar) model for NLs.

The parser is a part of a larger project to implement a natural language processing (NLP) system for Internet information retrieval (IR). This larger project requires Java applets capable of parsing a NL. More details regarding this application are presented in Kešelj 1999.

HPSG. The HPSG formalism for NLP is a high-level unification approach, suitable for studies in theoretical linguistics and strongly influenced by the Prolog paradigm. As any other grammar formalism, it is used to describe syntax of a NL; but, unlike other grammars, HPSG also provides a way of describing semantics simultaneously. Additionally, it is recognized that HPSGs also provide a very flexible model, in which a large number of NL phenomena can be effectively handled (Sag and Wasow 1999), and whose grammars are robust and can be relatively easily managed. For this reason, the HPSG model is attractive for lower-level, practical applications, i.e., for the area of natural language engineering. More details about the formalism and a precise definition of it, as used in Stefy, are given in Section 3.

Motivation behind Java parser. As mentioned above, this Java parser will be used in Internet IR applications. Some work on developing HPSG parsers in low-level programming languages has been done (Makino et al. 1998). However, we are not aware of any reports about systems implemented in Java.

The additional reasons for using Java for this task include:

- Java supports dynamic class loading and object serialization, which are important features necessary for our concept of *distributed NLP,*
- the similarity between hierarchical HPSG concepts and OOP (object-oriented programming) can be conveniently used in parser and grammar design, and
- Java is a good prototyping language, compared to C++ for example, and facilitates easy experimentation with various approaches, which makes this shift in programming language paradigm less drastic and easier to handle.

*This work is supported by NSERC.

2. BACKGROUND

HPSG. The HPSG model was formulated in 1986. The main books on the subject are Pollard and Sag (1994) and Sag and Wasow (1999). Carpenter (1992) gives a precise formal treatment of the formalism.

Parsers. The first HPSG chart parser was introduced by Proudian and Pollard (1985). The parser was implemented in Prolog. In 1990, Popowich and Vogel described their HPSG chart parser, also developed in Prolog, which had a better coverage of the developments in the HPSG formalism up to that time. Probably the best-known and the most widely used parser today is the system ALE (Carpenter and Penn 1999), which is developed in Prolog. The system LiLFeS (Makino et al. 1999) is an HPSG parser implemented in a low-level language. In Carpenter (1999), a chart parser for probabilistic context-free grammars, implemented in Java, is described. The LKB system (Copestake et al. 1999) is a grammar and lexicon development environment developed in Lisp. It is designed for use with constraint-based formalisms, more specifically for the use of typed feature structures, such as the HPSG formalism.

Unification. Unification is a built-in feature in the programming language Prolog, but it is not a part of Java. It is an essential notion in the HPSG model, so it is implemented within the project Stefy in form of a Java package called `avm`. The method of unification (at least in the modern sense) is introduced by Robinson in (1965). Our unification algorithm is a variation of the Huet's algorithm (Huet 1976). Knight (1989) gives a good survey on the subject of unification. Malouf (1999) treats some efficiency issues regarding graph unification for parsing constraint-based grammars, which are also relevant for the HPSG parsing.

3. HPSG FORMALISM

The HPSG formalism is a relatively complex notion (compared to context-free grammars, for example) and can be approached in various ways. One of the choices that we have to make before defining it is related to the tradeoff between implementational efficiency and efficacy of directly capturing of various linguistic rules. In this section we present our definition of the HPSGs, as implemented in Stefy, Version 0.1. This definition is implementation-oriented, so it does not necessarily reflect an ideal way of defining HPSGs in a theoretical-linguistic sense. However, as discussed in Section 5, it is more suitable for our task than the existing formalism.

Before defining the crucial notion of the HPSG formalism—the *head-driven phrase structure grammar*—we incrementally define several simpler notions.

The HPSG formalism is based on the *attribute-value matrices* (AVM), also called *(typed) feature structures.* The AVMs are recursive structures, and an important question is whether we allow cycles in those structures; e.g., can a proper part of a matrix be the matrix itself. Acyclic matrices are simpler to describe theoretically. On the other hand, allowing cycles is less expensive from the implementational point of view and they provide for a more direct representations of some NL phenomena.[1] For this reason, we choose to include cyclic matrices.

[1]E.g., see Carpenter 1992. For example, the sentence "This sentence is false." requires a cyclic structure for its semantic representation.

Example 1. Some examples of AVMs are given below:

$$\begin{bmatrix} \textit{employee} & \\ \mathsf{ID}: & \mathrm{JQP} \\ \mathsf{NAME}: & \text{John Q. Public} \\ \mathsf{DEPT}: & \begin{bmatrix} \textit{dept} & \\ \mathsf{ID}: & \mathrm{D1} \\ \mathsf{NAME}: & \text{Dept. of CS} \end{bmatrix} \end{bmatrix}$$

This is the example of an AVM containing data about an employee—a typical example from the database domain. The italic identifiers *employee* and *dept* denote *types,* the strings typeset in the sans-serif font and followed by a colon, e.g. ID:, denote *attributes,* and the other strings, e.g 'John Q. Public', are called *atoms.*

A list containing two atoms 'a' and 'b', i.e., $\langle \mathrm{a}, \mathrm{b} \rangle$, can be represented as follows:

$$\begin{bmatrix} \textit{nlist} & \\ \mathsf{HEAD}: & \mathrm{a} \\ \mathsf{TAIL}: & \begin{bmatrix} \textit{nlist} & \\ \mathsf{HEAD}: & \mathrm{b} \\ \mathsf{TAIL}: & [\textit{elist}] \end{bmatrix} \end{bmatrix}$$

The important notion of *structure sharing* can be illustrated by the following AVM:

$$\begin{bmatrix} \textit{nlist} & \\ \mathsf{HEAD}: & \boxed{1} \\ \mathsf{TAIL}: & \begin{bmatrix} \textit{nlist} & \\ \mathsf{HEAD}: & \boxed{1} \\ \mathsf{TAIL}: & [\textit{list}] \end{bmatrix} \end{bmatrix}$$

This AVM can be interpreted as a pattern that matches all arrays having at least two elements, and having the first two elements equal. The symbol $\boxed{1}$ denotes an *index,* or a *variable,* and it is used to denote "identical" entities, in a certain sense.

An example of a cyclic structure is the following:

$$\boxed{1}\begin{bmatrix} \textit{employee} & \\ \mathsf{ID}: & \mathrm{NC} \\ \mathsf{NAME}: & \text{Nick Cercone} \\ \mathsf{DEPT}: & \begin{bmatrix} \textit{dept} & \\ \mathsf{ID}: & \mathrm{CS} \\ \mathsf{NAME}: & \text{Dept. of CS} \\ \mathsf{CHAIR}: & \boxed{1} \end{bmatrix} \end{bmatrix}$$

It describes an employee, who is the chair of the department he belongs to.

Now, we can more formally introduce these notions. Let A_t denote a finite set of *attributes,* A denote an infinite enumerable set of *atoms,* and V denote an infinite enumerable set of *variables.* We assume that these sets are disjoint.

A set of types is a lattice, and it can be any finite lattice:

Definition 1 (Lattice of types) *A* lattice of types *is a non-empty finite set T with a partial ordering $\leq$, such that for every pair $t, u \in T$ there is a unique smallest element $t \vee u$*

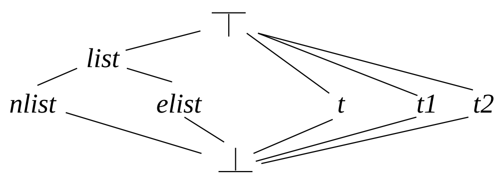
that is greater than t and u, and there is a unique largest element $t \wedge u$ that is smaller than t and u.

As a consequence, there exist the minimal and the maximal element of the set T, and they are denoted $\perp$ and $\top$.

We assume that the sets A_{t}, A, V, and T are disjoint.

The type $\perp$ is specific in our application in sense that it is not considered a "valid type;" rather, it is used to denote inconsistencies in unification operations.

Example 2. A lattice of types can be represented by a diagram, e.g.,



represents a lattice of types, where *nlist* $\vee$ *elist* $=$ *list*, *elist* $\wedge\, t = \perp$, and so on. Types *nlist* and *elist* can be used to represent non-empty and empty lists, as already illustrated in the previous example.

The first step towards defining AVMs is the definition of a simpler kind of AVM, which we call *basic AVM.*

Definition 2 (Basic AVM) *A* basic AVM *over the set of attributes A_{t}, the set of atoms A, the set of variables V, and the lattice of types T (or simply, over A_{t}, A, V, and T) is any atom, variable, or any tuple*

$$(t, (a_1, v_1), (a_2, v_2), \ldots, (a_n, v_n)),$$

where $n \geq 0$, $t \in T \setminus \{\perp\}$, $a_i \in A_{\mathrm{t}}$, $v_i \in V$, and $a_i \neq a_j$ for $i \neq j$. The set of all basic AVMs over A_{t}, A, V, and T is denoted $\overline{M}_{A_{\mathrm{t}},A,V,T}$, or simply $\overline{M}$.

We assume that the set of tuples defined in this way does not have any elements in common with the sets A_{t}, A, V, or T.

A basic AVM that is a tuple $(t, (a_1, v_1), (a_2, v_2), \ldots, (a_n, v_n))$ is denoted

$$\begin{bmatrix} t & \\ a_1 & v_1 \\ a_2 & v_2 \\ \vdots & \vdots \\ a_n & v_n \end{bmatrix}$$

If the type is not explicitly denoted, the type $\top$ is assumed. For instance, $[\,]$ represents the basic AVM: $(\top)$.

Definition 3 (System of equations) *Given a set of basic AVMs $\overline{M}$, a* system of equations *(or a* system *for short) is any finite relation $\sigma \subset V \times \overline{M}$.*

Definition 4 (Satisfiable system) *A system σ is* satisfiable *if there exists an equivalence relation $\equiv$ on $\overline{M}$, such that $\sigma \subseteq \equiv$, and for all $\overline{m} \equiv \overline{m}'$ one of the following holds:*

1. $\overline{m} \in V$ *or* $\overline{m}' \in V$,

2. $\overline{m}, \overline{m}' \in A$ *and* $\overline{m} = \overline{m}'$, *or*
3. $\overline{m} = (t, (a_1, v_1), \ldots, (a_n, v_n))$, $\overline{m}' = (t', (a'_1, v'_1), \ldots, (a'_{n'}, v'_{n'}))$, $a_i, a'_i \in A_{\mathrm{t}}$, $v_i, v'_i \in V$, $a_i = a'_j \Rightarrow v_i \equiv v'_j$; *and there is a* $t_0 \in T \setminus \{\perp\}$ *such that for all basic AVMs* $\overline{m}'' = (t'', (a''_1, v''_1), \ldots, (a''_{n''}, v''_{n''}))$:
$$\overline{m} \equiv \overline{m}'' \Rightarrow t'' \geq t_0.$$

If σ is a satisfiable system, then there exists the smallest equivalence relation $\equiv$ with the above properties, and we denote it $\equiv_\sigma$.

Example 3. A simple example of an unsatisfiable system is:

$$\{(v_1, \mathrm{a}), (v_2, \mathrm{b}), (v_1, v_2)\}$$

where a and b are two distinct atoms.

Definition 5 (AVM) *An* AVM *is any pair* (v, σ) *of a distinguished variable* v *and a satisfiable system* σ. *The set of all AVMs is denoted* $M_{A_{\mathrm{t}},A,V,T}$, *or simply* M.

Definition 6 (Domain of variables) *For any basic AVM* $\overline{m}$, *we define the* domain of variables *of* $\overline{m}$, $D_{\mathrm{v}}(\overline{m})$, *to be:*

$$D_{\mathrm{v}}(\overline{m}) = \begin{cases} \emptyset, & \text{if } \overline{m} \in A; \\ \{\overline{m}\}, & \text{if } \overline{m} \in V; \\ \{v_1, \ldots, v_n\}, & \text{if } \overline{m} = (t, (a_1, v_1), \ldots, (a_n, v_n)). \end{cases}$$

For a system σ, *the* domain of variables *of* σ *is*

$$D_{\mathrm{v}}(\sigma) = \bigcup_{(v,\overline{m}) \in \sigma} (\{v\} \cup D_{\mathrm{v}}(\overline{m})).$$

For any AVM $m = (v, \sigma)$, *we define the* domain of variables *of* m, $D_{\mathrm{v}}(m)$, *to be:*

$$D_{\mathrm{v}}(m) = \{v\} \cup D_{\mathrm{v}}(\sigma).$$

Intuition behind the domain of variables is that a domain of variables includes all variables which are part of an AVM. Two AVMs are “independent” if their domains of variables are disjoint.

Since any system σ is finite, $D_{\mathrm{v}}(m)$ is a finite set for any system or any AVM. It is also finite for any basic AVM.

Example 4. A frequently used structure in HPSGs is a list. For instance, $\langle a_1, a_2 \rangle$ is a list of two atoms $a_1, a_2 \in A$. Using the type hierarchy given in the previous example, the list $\langle a_1, a_2 \rangle$ can be represented in the standard HPSG notation as:

$$\begin{bmatrix} \mathit{nlist} & \\ \textsf{HEAD:} & a_1 \\ \textsf{TAIL:} & \begin{bmatrix} \mathit{nlist} & \\ \textsf{HEAD:} & a_2 \\ \textsf{TAIL:} & \begin{bmatrix} \mathit{elist} \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

According to Definition 5, this AVM can be represented as a pair (v_1, σ), where

$$\begin{aligned} \sigma \;=\; & \{(v_1, (\mathit{nlist}, (\textsf{HEAD}, v_2), (\textsf{TAIL}, v_3))), (v_2, a_1), \\ & (v_3, (\mathit{nlist}, (\textsf{HEAD}, v_4), (\textsf{TAIL}, v_5))), (v_4, a_2), (v_5, (\mathit{elist}))\}. \end{aligned}$$

The domain of variables of this AVM is $D_{\mathrm{v}}((v_1, \sigma)) = \{v_1, v_2, v_3, v_4, v_5\}$.

Definition 7 (HPSG rule) *If $m \in M$ and $v_1, \ldots, v_n \in V$ $(n \geq 1)$, then*

$$m \to v_1 v_2 \ldots v_n$$

is an HPSG non-lexical rule.

Let Σ be a distinguished subset of A, which we call alphabet, *and its elements are called* letters *or* terminals. *For any $m \in M$ and $a \in \Sigma$,*

$$m \to a$$

is an HPSG lexical rule.

Lexical and non-lexical HPSG rules are called HPSG rules.

We define the domain of variables *for HPSG rules in the following way:*

$$D_{\mathrm{v}}(m \to v_1 v_2 \ldots v_n) = D_{\mathrm{v}}(m) \cup \{v_1, v_2, \ldots, v_n\}$$

and

$$D_{\mathrm{v}}(m \to a) = D_{\mathrm{v}}(m).$$

If $f : V \to V$ is a bijection, then a *variable renaming* defined by f is a transformation that replaces each variable v with its image $f(v)$. Let r be a rule, let $D_{\mathrm{v}}(r) = D$, and let D_1 be a finite set of variables such that $|D| = |D_1|$. Then, we can always find a variable renaming such that after we rename all variables in r and obtain a new rule r_1, $D_{\mathrm{v}}(r_1)$ becomes D_1.

Definition 8 (HPSG) *An HPSG is any tuple $(\Sigma, A_{\mathrm{t}}, A, V, T, S, P)$, where Σ is an alphabet, A_{t} is a set of attributes, A is a set of atoms ($\Sigma \subseteq A$), V is a set of variables, T is a lattice of types, S is the initial AVM ($S \in M$), and P is a finite set of HPSG rules.*

Definition 9 (HPSG derivation) *Let $(\Sigma, A_{\mathrm{t}}, A, V, T, S, P)$ be an HPSG. The* HPSG derivations *have the form $m \Rightarrow \alpha$, where $m \in M$ and $\alpha \in (V \cup \Sigma)^*$, and they are defined recursively in the following way:*

1. *If $S = (v, \sigma)$, then $S \Rightarrow v$ is a derivation. The domain of variables of this derivation is defined to be $D_{\mathrm{v}}(S \Rightarrow v) = D_{\mathrm{v}}(S)$.*
2. *If*
   - *$S_1 \Rightarrow \alpha v \beta$ is a derivation, where $v \in V$, $S_1 = (v_1, \sigma) \in M$, and $\alpha, \beta \in (V \cup \Sigma)^*$,*
   - *$m_1 \to \gamma$ is a rule in P, or is obtained from a rule in P by variable renaming, where $m_1 = (v_2, \sigma_1)$; so that the variable domains of $S_1 \Rightarrow \alpha v \beta$ and $m_1 \to \gamma$ are disjunct, and*
   - *$S_2 = (v_1, \sigma \cup \sigma_1 \cup (v, v_2))$ is an AVM,*

   then $S_2 \Rightarrow \alpha \gamma \beta$ is a derivation. Its domain of variables is defined to be $D_{\mathrm{v}}(S_2 \Rightarrow \alpha \gamma \beta) = D_{\mathrm{v}}(S_1 \Rightarrow \alpha v \beta) \cup D_{\mathrm{v}}(m_1 \to \gamma)$.

Example 5. Let

$$\begin{bmatrix} t & \\ \mathsf{RPREF:} & \alpha \\ \mathsf{SUFF:} & \beta \end{bmatrix}$$

be a form of AVM, which we use to represent a list of atoms. The lists α and β are used to store a reversed prefix and corresponding (non-reversed) suffix of the represented list. For instance, the list $\langle a_1, a_2, a_3 \rangle$ can be represented as:

$$\begin{bmatrix} t & \\ \mathsf{RPREF:} & \langle a_2, a_1 \rangle \\ \mathsf{SUFF:} & \langle a_3 \rangle \end{bmatrix}$$

We need two rules in order to be able to derive one representation from any other equivalent representation: one rule to move an atom from a reversed prefix to the suffix, and another rule to move an atom from a suffix to the reversed prefix. We illustrate just one of the rules, since the other is analogous. First, using the standard HPSG notation, the rule can be represented as follows:

$$\boxed{1}\begin{bmatrix} t & & \\ \mathsf{RPREF}: & \boxed{2} & \begin{bmatrix} nlist & \\ \mathsf{HEAD}: & \boxed{4} \\ \mathsf{TAIL}: & \boxed{5}\,[list] \end{bmatrix} \\ \mathsf{SUFF}: & \boxed{3}\,[list] & \end{bmatrix} \rightarrow \boxed{6}\begin{bmatrix} t & & \\ \mathsf{RPREF}: & \boxed{5} & \\ \mathsf{SUFF}: & \boxed{7} & \begin{bmatrix} nlist & \\ \mathsf{HEAD}: & \boxed{4} \\ \mathsf{TAIL}: & \boxed{3} \end{bmatrix} \end{bmatrix}$$

The indices of the form $\boxed{i}$ are used to denote variables v_i. Now we can see how the rule is expressed according to Definition 7: It is a rule $m \rightarrow v_6$, where $m = (v_1, \sigma)$ is the AVM defined by:

$$\begin{aligned} \sigma = \quad \{ \quad & (v_1, (t, (\mathsf{RPREF}, v_2), (\mathsf{SUFF}, v_3))), \\ & (v_2, (nlist, (\mathsf{HEAD}, v_4), (\mathsf{TAIL}, v_5))), \\ & (v_3, (list)), (v_5, (list)), \\ & (v_6, (t, (\mathsf{RPREF}, v_5), (\mathsf{SUFF}, v_7))), \\ & (v_7, (nlist, (\mathsf{HEAD}, v_4), (\mathsf{TAIL}, v_3)))\}. \end{aligned}$$

Definition 10 (Language generated by HPSG) *Let $(\Sigma, A_t, A, V, T, S, P)$ be an HPSG. The language generated by this grammar is the set:*

$$\{w \in \Sigma^* : S \Rightarrow w \textit{ is an HPSG derivation }\}.$$

The HPSG formalism defined in this way does not seem to justify its name, since the role of the word *head* remains unclear. The word becomes relevant in certain conventions used for building rules, called *principles.* We regard principles as a part of higher level of abstraction. Within this formalism, they are part of the rules. However, the implementation provides an explicit way of specifying principles.

The HPSG formalism defined in this way (as the standard HPSG formalism) does not give us any new class of languages. It is equivalent to the Turing machine, i.e., the following theorem holds:

Theorem 1 (Equivalence to Turing machines) *The class of languages generated by HPSGs is equal to the class of recursively enumerated languages.*

Proof. First, we prove that each language generated by an HPSG is recursively enumerable. By finding all HPSG derivations consisting of no more than n steps given in Definition 9, and incrementally increasing n, we can output all words of the language using a Turing machine. The only nontrivial routine is verification that a system of equations is satisfiable. We will see in the next section how this can be done effectively.

Second, we prove that any recursively enumerable language can be generated using an HPSG. It is known that any recursively enumerable language can be generated by an unrestricted grammar in the Chomsky hierarchy (i.e., type 0 grammar; e.g., see Hopcroft and Ullman 1979). Hence, it is sufficient to show that any unrestricted grammar can be simulated by an HPSG. Using the AVMs shown in the previous examples, we simulate unrestricted grammar derivations by representing their sentential forms as lists of terminals and non-terminals. Hence, we include all terminals and non-terminals of the unrestricted grammar in the set of atoms of the HPSG. If S is the start symbol of the unrestricted grammar, then the initial AVM of the HPSG grammar is:

$$\begin{bmatrix} t \\ \mathsf{RPREF:}\ \langle\rangle \\ \mathsf{SUFF:}\ \ \langle S\rangle \end{bmatrix}$$

The symbol $\langle\rangle$ denotes the AVM $[elist]$. The rules from the previous example are included, so symbols can be moved orderly from reversed prefix to suffix and vice versa. For each unrestricted rule, for example $ABC \to XY$, we introduce an HPSG rule:

$$\begin{bmatrix} t \\ \mathsf{RPREF:}\ \boxed{1}\,[list] \\ \mathsf{SUFF:}\ \ \langle A, B, C, \boxed{2}\,[list]\rangle \end{bmatrix} \to \begin{bmatrix} t \\ \mathsf{RPREF:}\ \boxed{1} \\ \mathsf{SUFF:}\ \ \langle X, Y, \boxed{2}\rangle \end{bmatrix}$$

In order to produce a word at the end of derivation, for each terminal a of the unrestricted grammar, we introduce the following three HPSG rules:

$$\begin{bmatrix} t \\ \mathsf{RPREF:}\ \langle\rangle \\ \mathsf{SUFF:}\ \ \langle a, \boxed{1}\,[list]\rangle \end{bmatrix} \to \begin{bmatrix} t1 \\ \mathsf{TERM:}\ a \end{bmatrix} \begin{bmatrix} t \\ \mathsf{RPREF:}\ \langle\rangle \\ \mathsf{SUFF:}\ \ \boxed{1} \end{bmatrix}$$

$$\begin{bmatrix} t \\ \mathsf{RPREF:}\ \langle\rangle \\ \mathsf{SUFF:}\ \ \langle a\rangle \end{bmatrix} \to \begin{bmatrix} t1 \\ \mathsf{TERM:}\ a \end{bmatrix}$$

$$\begin{bmatrix} t1 \\ \mathsf{TERM:}\ a \end{bmatrix} \to a$$

The last rule is the lexical rule. HPSG constructed in this way generates the same language as the starting unrestricted grammar. ■

4. UNIFICATION ALGORITHM

Before presenting the unification algorithm, we introduce notions of equivalent systems, equivalent AVMs, solved systems, and compact AVMs.

4.1. Equivalent AVMs

Intuitively, the variables in AVMs are used to "connect" various parts of an AVM, and this "connecting" can be done in various ways for one conceptually unique AVM.

Example 6. The AVM

$$\begin{bmatrix} \mathsf{A:}\ \mathrm{a} \\ \mathsf{B:}\ \mathrm{b} \end{bmatrix}$$

can be represented in the following two equivalent ways:

$$(v_1, \{(v_1, (\top, (\mathsf{A}, v_2), (\mathsf{B}, v_3))), (v_2, \mathrm{a}), (v_3, \mathrm{b}), (v_4, v_1)\})$$

and

$$(v_1, \{(v_1, (\top, (\mathsf{A}, v_2))), (v_2, \mathrm{a}), (v_3, \mathrm{b}), (v_4, v_1), (v_4, (\top, (\mathsf{B}, v_3)))\})$$

Let us formally define this equivalence relation:

Definition 11 (Equivalent systems and equivalent AVMs) *Two systems σ_1 and σ_2 are equivalent if for any other system σ_3, both systems $\sigma_1 \cup \sigma_3$ and $\sigma_2 \cup \sigma_3$ are either satisfiable or not satisfiable in the same time.*

Two AVMs (v_1, σ_1) and (v_2, σ_2) are equivalent, if $v_1 \equiv_{\sigma_1} v_2$ and the systems σ_1 and σ_2 are equivalent.

We can recall that the relation $\equiv_\sigma$ is defined after definition 4. If systems σ_1 and σ_2 are equivalent, then the relations $\equiv_{\sigma_1}$ and $\equiv_{\sigma_2}$ are equal. Otherwise, if a pair $(\overline{m}_1, \overline{m}_2)$ belongs to one and not the other relation, then we can find two variables v_1 and v_2 such that $v_1 \not\equiv_{\sigma_1} v_2$ and $v_1 \not\equiv_{\sigma_2} v_2$, and two distinct atoms a and b, and define:

$$\sigma_3 = \{(v_1, a), (v_2, b), (v_1, \overline{m}_1), (v_2, \overline{m}_2)\}$$

According to the above definition, the systems σ_1 and σ_2 are not equivalent, since exactly one of the systems $\sigma_1 \cup \sigma_3$ and $\sigma_2 \cup \sigma_3$ is satisfiable. Hence, the definition of equivalent AVMs is symmetric, i.e., $v_1 \equiv_{\sigma_1} v_2$ if and only if $v_1 \equiv_{\sigma_2} v_2$, for equivalent systems σ_1 and σ_2.

This relation is an equivalence relation and it partitions the set of all systems and the set of all AVMs into equivalence classes. Note that all unsatisfiable systems form one class in the set of systems. Each other class of systems can be put in a pair with a distinguished variable, and such a pair is associated with a class of AVMs. In the previous example, we saw two equivalent AVMs. The former AVM does not "divide," the matrix, and we consider that to be a good property. The later AVM does divide the matrix. We distinguish the former AVM, and call it a *compact AVM*. In each equivalence class of satisfiable AVMs, there is at least one compact AVM.

Definition 12 (Solved system and compact AVM) *A satisfiable system σ is called a* solved system *if the following properties are satisfied:*

- *σ has the functional property, i.e., $(x, y) \in \sigma \wedge (x, z) \in \sigma \Rightarrow y = z$,*
- *σ is anti-reflexive, i.e., $(x, x) \notin \sigma$ for all x, and*
- *if σ^+ is the transitive closure of σ, then σ^+ is antisymmetric; i.e., for all variables v_1 and v_2, $(v_1, v_2) \in \sigma^+$ and $(v_2, v_1) \in \sigma^+$ implies $v_1 = v_2$.*

An AVM (v, σ) is called a compact AVM *if σ is a solved system.*

The compact AVMs are important, since they are easier to represent and more efficient to use in our implementation.

4.2. Unification Algorithm

The standard unification algorithm is reduced to the algorithm for verifying that a system is satisfiable. The problem of unifying two AVMs $m = (v, \sigma)$ and $m_1 = (v_1, \sigma_1)$ is reduced to verifying that the system $\sigma_2 = \sigma \cup \sigma_1 \cup \{(v, v_1)\}$ is satisfiable. The result of the unification is the AVM (v, σ_2). The result could be specified in other ways as well, but the resulting AVMs would be equivalent to (v, σ_2). In the implementation, we prefer that the result is in a compact form.

Algorithm IsSatisfiable is given for the problem: Is a system σ satisfiable? In addition to a Boolean output, σ_s is a solved system equivalent to σ if the system is satisfiable. The well-known UNION-FIND structure for representing disjoint sets is used (e.g., see Cormen et al. 1989). The stack operations POP and PUSH are also used on σ. The algorithm is based on the Huet's unification algorithm (Huet 1976). The basic idea is to keep all equivalent variables ($\equiv_\sigma$) in one set, and use an array f to associate each set representative, which is a variable, with a basic AVM.

Algorithm 1 (IsSatisfiable)

Input:	σ	input system
Output	**true** or **false**	Is σ satisfiable?
	σ_s	defined if σ is satisfiable; solved and equivalent system to σ

For each $v \in D_{\mathrm{v}}(\sigma)$ **do** MAKE-SET(v), $f[v] \leftarrow$ NULL
While $\sigma \neq \emptyset$ **do**
| $(v, x) \leftarrow$ POP(σ)
| **If** $x \in V$ **then**
| | $f_1 \leftarrow f$[FIND-SET(x)]
| | $f_2 \leftarrow f$[FIND-SET(v)]
| | $v \leftarrow$ UNION(v, x)
| | $f[v] \leftarrow f_2$
| **Else**
| | $f_1 \leftarrow x$
| | $v \leftarrow$ FIND-SET(v)
| **If** $f[v] = f_1$ **or** $f_1 =$ NULL **then continue**
| **Else If** $f[v] =$ NULL **then** $f[v] \leftarrow f_1$
| **Else If** $f_1 \in A$ **or** $f[v] \in A$ **then Return false**
| **Else**
| | Let $f[v] = (t, (a_1, v_1), \ldots, (a_n, v_n))$ and
| | $f_1 = (t', (a'_1, v'_1), \ldots, (a'_{n'}, v'_{n'}))$
| | $t'' \leftarrow t \wedge t'$
| | **If** $t'' = \bot$ **then Return false**
| | $\{a''_1, \ldots, a''_{n''}\} \leftarrow \{a_1, \ldots, a_n\} \cup \{a'_1, \ldots, a'_{n'}\}$
| | **For each** a''_i in $\{a''_1, \ldots, a''_{n''}\}$ **do**
| | | **If** $a''_i = a_j = a'_k$ for some a_j and a'_k **then**
| | | | PUSH$(\sigma, (v_j, v'_k))$
| | | | $v''_i \leftarrow v_j$
| | | **Else If** $a''_i = a_j$ for some a_j **then** $v''_i \leftarrow v_j$
| | | **Else** $(a''_i = a'_k$ for some $a'_k)$ $v''_i \leftarrow v'_k$
| | $f[v] \leftarrow (t'', (a''_1, v''_1), \ldots, (a''_{n''}, v''_{n''}))$
$\sigma_s \leftarrow \emptyset$
For each $v \in D_{\mathrm{v}}(\sigma)$ **do**
| **If** FIND-SET$(v) = v$ **then** PUSH$(\sigma_s, (v, f[v]))$
| **Else** PUSH$(\sigma_s, (v,$ FIND-SET$(v)))$
Return true

In order to determine the running-time complexity, we have to make some assumptions about the size of the input. The input system σ is a set of ordered pairs of a variable and a basic AVM. The size of a basic AVM is a constant if it is an atom or a variable, or it is proportional to the number of pairs in it if it is a tuple; i.e., it is $a + bn$ where a and b are constants, and n is the size of the tuple. We assume that the size of one such pair is the sum of the sizes of its components, and the size of a system is the sum of the sizes of all such pairs.

The loop 2–26 is repeated $O(n)$ times, where n is the size of the input. This is because in each iteration of the loop one pair is poped out of σ (step 3). On the other side, new pairs are pushed in step 22, but each such push reduces the total size of all basic AVMs by one, so there can be at most $O(n)$ pushes.

The loop 28–30 is repeated $O(n)$ times, as well.

There are two approaches to analysis of the loop 20–25. There is a finite number of attributes in an HPSG. In the implementation, this is used to create a hash table without collisions for keeping attribute-value pairs. Having this in mind, the number of iterations in loop 20–25 is $O(1)$ (within the outside loop). On the other hand, if we wanted to avoid this assumption we could keep the attributes in an efficient search tree and the number of iterations in the loop would be $O(\log(n))$.

The amortized complexity of the operations FIND-SET and UNION is $O(m \cdot \alpha(m))$ for m operations, where $\alpha(m)$ is the very slowly growing inverse of Ackermann's function (Cormen et al. 1989). Hence the complexity of our algorithm is $O(n \cdot \alpha(n))$. If we did not assume that the number of attributes is limited by a constant, and used search trees to represent basic-AVM tuples, then the complexity would be $O(n \log(n) \cdot \alpha(n \log(n)))$.

5. DISCUSSION

The HPSG formalism is usually based on the theory of typed feature structures described in Carpenter 1992. The definition presented here is more compact, and it provides for an easy implementation in programming languages that do not include a built-in unification mechanism. In particular, this formalism describes the Java parser implemented.

However, the formalism is not suitable for development of the grammar and lexical resources. Other systems, like ALE (Carpenter and Penn 1999) and LKB (Copestake 1999), are more suitable for this task. After a grammar or a lexicon is developed in one of those systems, it is translated into a Java description and used in our system.

6. CONCLUSION

In this paper, we have presented a new precise and compact description of the HPSG formalism. It is the underlying formalism of the parser Stefy implemented in Java, which is a part of an IR system. It is proven that the grammar defines the set of recursively enumerable languages.

ACKNOWLEDGMENTS

I wish to thank Dr. Nick Cercone for valuable discussions and comments regarding this paper. This work is supported by the Natural Sciences and Engineering Research Council of Canada (NSERC).

REFERENCES

CARPENTER, BOB. 1992. The Logic of Typed Feature Structures with Applications to Unification-based Grammars, Logic Programming and Constraint Resolution, volume 32 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, New York, 1992.

Carpenter, Bob. 1999. Probabilistic word graph parser: Java source & documentation. http://www.colloquial.com/carp/Parser/.

Carpenter, Bob, and Penn, Gerald Penn. 1999. ALE, the attribute logic engine, user's guide. WWW, May 1999. http://www.sfs.nphil.uni-tuebingen.de/~gpenn/ale.html.

Copestake, Ann. 1999. *The (new) LKB system, Version 5.2*, October 1999. http://www-csli.stanford.edu/~aac/lkb.html.

Cormen, Thomas H., Leiserson, Charles E., and Rivest, Ronald L. 1989. *Introduction to Algorithms*. The MIT Press.

Earley, J. 1970. An efficient context-free parsing algorithm. *Communications of the ACM*, 13(2):94–102, February 1970.

Hopcroft, John E., and Ullman, Jeffrey D. 1979. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley.

Huet, G. 1976. *Résolution d'equations dans les langages d'ordre 1,2, ...,ω*. Thèse de Doctorat d'Etat, Université de Paris 7 (France).

Kešelj, Vlado. 1999. Java parser for HPSGs: Why and how. In Nick Cercone, Kiyoshi Kogure, and Kanlaya Naruedomkul, editors, *Proceedings of the Conference Pacific Association for Computational Linguistics, PACLING'99*, pages 288–294, Waterloo, Ontario, Canada, August 1999. http://www.grad.math.uwaterloo.ca/~vkeselj/papers/pacling99.http.

Knight, Kevin. 1989. Unification: a multidisciplinary survey. *ACM computing surveys, March 1989*, 21(1):93–124.

Makino, Takaki, Yoshida, Minory, Torisawa, Kentaro, and Tsujii, Jun'ichi. 1998. LiLFeS—towards a practical HPSG parser. In *Proceedings of the COLING-ACL 98, Montreal*, pages 807–811.

Malouf, Rob, and Carroll, John. 1999. Efficient graph unification for parsing constraint-based grammars. Paper presented at the Computer Science Department, Duke University. http://hpsg.stanford.edu/rob/papers/duke-slides.ps.gz.

Pollard, Carl J, and Sag, Ivan A. 1994. *Head-Driven Phrase Structure Grammar*. University of Chicago Press, Chicago.

Popowich, Fred, and Vogel, Carl. 1990. Chart parsing head-driven phrase structure grammar. Technical Report CSS-IS TR 90-01, Simon Fraser University.

Proudian, Derek, and Pollard, Carl J. 1985. Parsing head-driven phrase structure grammar. In *Proceedings of the Twenty-Third Annual Meeting of the ACL*, pages 167–171, Chicago, IL. ACL.

Robinson, J. A. 1965. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, January 1965.

Sag, Ivan A., and Wasow, Thomas. 1999. *Syntactic Theory: A Formal Introduction*. CSLI Publications, Stanford.