Flexible Semantics Communication in Integrated Speech/Language Architectures

Abdel Kader Diagne and John Nerbonne*
Deutsches Forschungszentrum für Künstliche Intelligenz
Stuhlsatzenhausweg 3
6600 Saarbrücken 11, Germany
{ diagne, nerbonne } @dfki.uni-sb.de

Abstract: We consider communication between modules in an integrated architecture for Speech and Natural Language (NL), in particular the communication with the semantics module. In an integrated Speech/Language system several components—phonology (intonation), syntax, context model—may express meaning constraints, which the semantics module must flexibly manage and evaluate, in order to enable semantic inference. This paper describes an implemented approach in the ASL Project in which nonsemantic modules provide feature-based contraints that are then translated into a meaning representation language. We realize these TRANSLATOR FUNCTIONS in the spirit of federated agents' architectures (Genesreth); this functionality is required in heterogenous integrated architectures, and is implemented here using compiler technology.
Keywords: Software Interoperability, Translator Functions, Agents' (Federation) Architectures, Interfaces
Area: Natural Language and Speech Understanding.
Deutsche Zusammenfassung Wir betrachten die Kommunikation zwischen Modulen in einem System zur integrierten Analyse von Sprachlauten und Sprachstrukturen (Speech and Language) und speziell die Kommunikation mit dem Semantikmodul in so einem System. Sowohl die Syntax als auch u.a. die Phonologie (Intonation) und das Kontextmodell dürfen Einschränkungen über Semantik ausdrücken. Das Semantikmodul muß diese Constraints verwerten und darauf basierend Inferenzen vollziehen. Diese Arbeit beschreibt einen im ASL-Projekt bereits implementierten Ansatz, in dem nicht-semantische Module Einschränkungen über die Semantik in Form von Merkmalsstrukturen an den Semantikmodul liefern, die dann in die semantische Repräsentationssprache übersetzt werden. Wir realisieren diese ÜBERSETZUNGSFUNKTIONEN im Sinne der "'föderativen Agentenarchitek-

*This work was supported by research grant ITV 9102 from the German Bundesministerium für Forschung und Technologie (BMFT) to the DFKI ASL project. We thank Joachim Laubsch for valuable discussions. Diagne is the principal author and technical contributor for the work presented here.

tur"' (Genesreth); Übersetzungsfunktionalität ist in solchen heterogenen Agentenarchitekturen erforderlich und wird hier mittels Kompilertechnologie implementiert.

1 Introduction and Motivation

The technical kernel of the present paper is the description of an implemented general translation facility for a semantics module. This facility translates constraints from nonsemantic modules into a meaning representation language. It may be seen as a generalization of the syntax/semantics interface customary in NLP systems, but it is superior first in allowing multiple determination of semantics (important in speech/language applications), and second in enabling independent engineering for a semantics module—allowing it to function in various systems with minimal system prerequisites. The translation facility is best seen as the instantiation of (one part of) the FACILITATOR in federated agents' architectures [4].

This introduction discusses both motivational points and the sketches the place of translator functions in integrated agents' architectures. The next section gives a brief overview on the meaning representation formalism $\mathcal{NLL}$ and describes its interfacing facilities. In Section 3 we describe our design for translating feature based formalisms into $\mathcal{NLL}$, emphasizing its debt to compiler technology. We discuss problems that arise, our present solutions, and implementation issues. Examples are provided. A final section attempts to evaluate this work and give perspectives for interfacing the semantics module with further components in integrated speech/language architectures.

1.1 Semantics in Speech/Language Systems

The need for a general translation facility arises particularly in speech/language systems, in which input to the semantics module comes not only from syntax and lexicon, but also from suprasegmental phonology (intonation and focus accent) among perhaps others. In order to deal with these multiple sources of semantic content, we abandon the idea of there being a functional, homomorphic relation between a single (syntactic) source and a semantic representation, and instead conceive of semantics construction as a constraint-satisfaction process ([10, 11], in which syntax, suprasegmental phonology (and eventually contextual pragmatics) provide constraints on logical form which the semantics module evaluates, reasons from, and applies appropriately (our present applications are calendar management and train schedule information). The constraint-satisfaction view of semantics construction is furthermore useful in speech/language systems because it allows one to deal with partial information incrementally [5]. A first motivation for our translation facility is that it enables the constraint-satisfaction view of semantics construction. Figure 1 sketches the place of the semantics module in the communication model proposed for the ASL-Nord-Project [12].

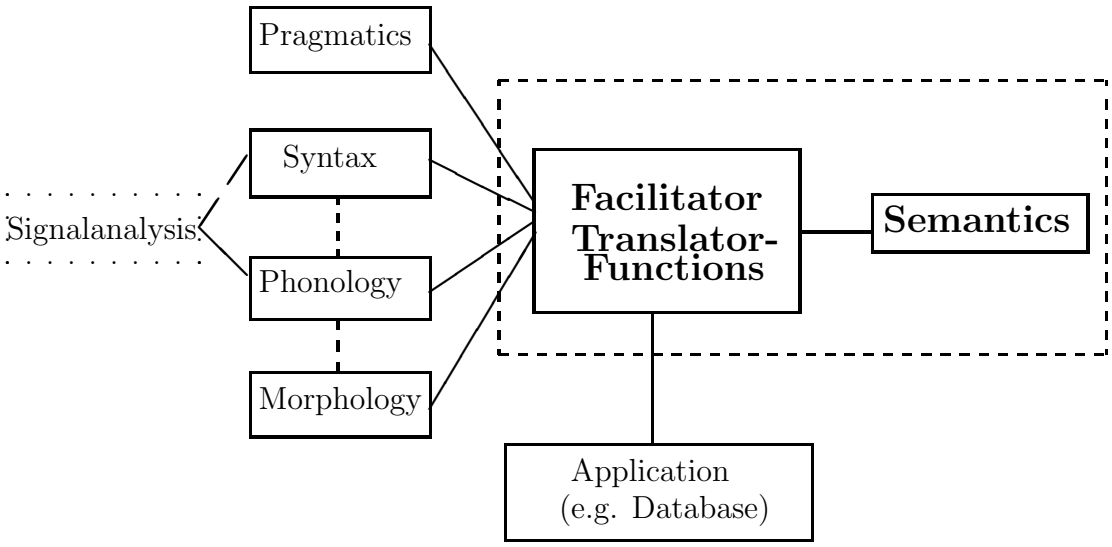

Figure 1: **Information Flow to Semantics in a Speech System:** The semantics interface contains translator-functions (one for each incoming specification language), which render incoming data and queries to the module in the semantics' internal representation language. The translators' design is based on compiler technologies, which improves the generality of the approach.

1.2 Engineering for Computational Semantics

But the facility is also very useful in systems where semantics remains entirely syntax-driven. The language in which constraints are formulated is generally that of typed feature logic ([2], [3]), and the semantic representation language is generally a variant of a higher-order predicate logic, e.g., generalized quantifier language or the language of discourse representation theory, (here it is $\mathcal{NLL}$, described in Section 2 below). The modules thus do not use the same language, and indeed could not, under normal assumptions about the expressive capabilities required in phonology and syntax on the one hand, and semantics on the other. This justifies the need for translator functions (even if more traditional syntax/semantics interfaces could suffice here).

This motivation is strengthened if one considers the state of natural language engineering for large NL systems. These are presently too complex for any single small group to develop. As a result, today's systems increasingly involve some incorporation of heterogeneous modules taken from different development efforts. This tactic of eclectically combining modules has great potential but harbors serious system architecture (communication) problems. It is easy to find components which perform well in specific systems, but which are difficult to use elsewhere because of idiosyncratic input and interface requirements. This suggests that the design of contemporary components should foresee the needs of software interoperability—the deployment of components in heterogeneous systems—from the beginning, insisting on modularity, flexibility (ease of experiment), and tolerance (minimal interface requirements) [9]. The translator facilities we provide in the

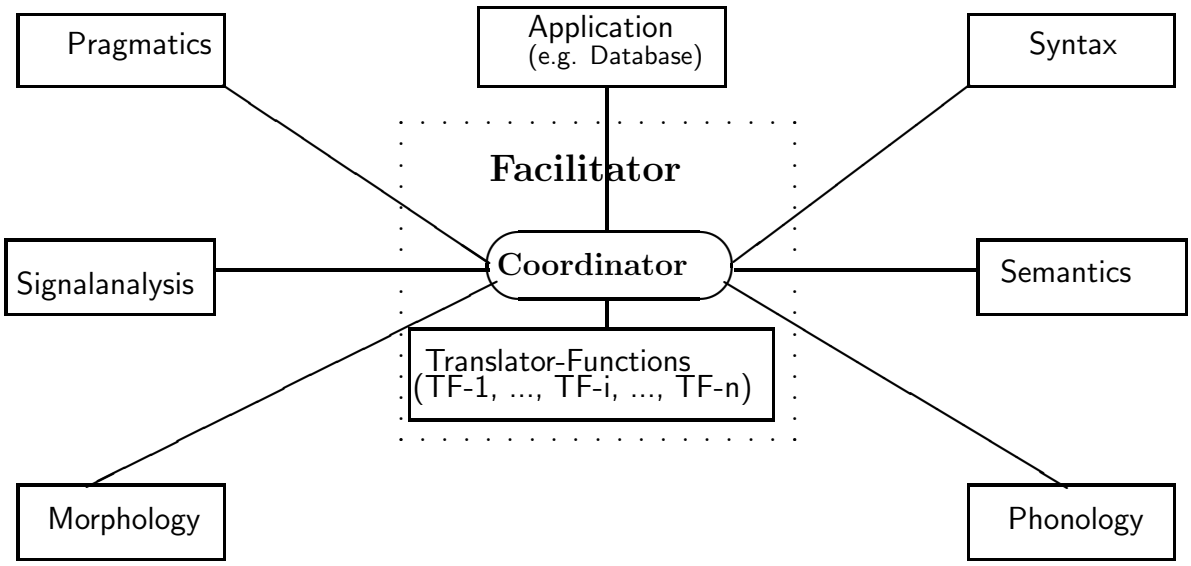

Figure 2: Communication in an agents' architecture with a global facilitator. Each module (agent) is related to a global *facilitator* which manages the communication. The facilitator consists of a *coordinator*, responsible for brokering and scheduling, and a set of translator-functions. Each module may use a different input/output data format. The task perfomed by each translator-function is to translate incoming data for further processing in a particular module. The coordinator determines the further destination of outgoing data, and invokes translators where needed.

work described serve especially the last purpose—they allow a very general use of the semantics module they provide interface to, without prerequisites about the theories, designs or even implementation languages of other system components.

1.3 Agents' Architectures

Even if we claim a general usefulness for the translator functions described below, they are quite specifically required in contemporary designs for federated agents' architectures [4]. These systems are designed to allow for heterogeneous agents (modules), and foresee that their federated actions may be coordinated by a FACILITATOR, whose tasks include brokering (deciding on responsibilities), scheduling (allocating resources and priorities), and translating (one agent's representations into another's). The last function is required wherever agents may be heterogeneous. Figure 2 sketches a federated agents' architecture.

The work we present here is compatible with the view of the semantics module as an agent whose language is $\mathcal{NLL}$, which cooperating agents need not know. The translation into $\mathcal{NLL}$ is accomplished by the functions we detail here.

2 Overview of NLL

$\mathcal{NLL}$ is a logical language for representing the meaning of natural language expressions on the computer[8].[1] Its design has five goals:

1. to provide an independent semantics module for NLP systems;
2. to support some semantic inference;
3. to provide a base for disambiguation (like SRI's QLF, quasi-logical form, [1]) and domain-specific interpretation (e.g., transduction into a database query language, such as SQL, or a language for controlling an application, such as the AKA-MOD agents' system);
4. to support experimentation at the level of semantic representation; and
5. to facilitate the semantic representation of common grammatical constructs (lexical and syntactic).

$\mathcal{NLL}$ is based on predicate logic with keyworded arguments; it contains several further expressive devices developed for the representation of NL, including plurals terms, λ abstracts, generalized quantifiers, comparative predicates and the measure phrases they employ ('60 meters'), complex determiners, and variable-binding terms. In several instances inference rules associated with the defined devices are implemented, and in general inference is supported to the extent that is required in interface applications [6]. For example, there are implemented rules for implied conjunct removal, subsumtion of a literal by specialization, distributive inferences for plural terms, simplification rules for locative terms, rewriting rules for disambiguation, and substitution [7].

3 Semantics Communication Facilities

Since $\mathcal{NLL}$ is designed to function as an independent module, a number of interfaces are available. $\mathcal{NLL}$ can be created using (i) any of numerous constructor functions; (ii) through the invocation of a reader which parses the logical syntax of $\mathcal{NLL}$: or (iii) through the specification of $\mathcal{NLL}$ structures in a feature description language. In this fashion, $\mathcal{NLL}$ attempts to allow the NL system developer with a number of choices. We should note that both interfaces (ii) and (iii) require only that strings be provided to the $\mathcal{NLL}$ modules. There is no requirement that the syntax be based on a particular theory, programming language or even machine architecture.

The interface (iii), specifying $\mathcal{NLL}$ via typed feature structures, is of central interest here, because it allows the integration of multiple modules simultaneously constraining semantics—the speech situation. In the case where multiple modules constrain semantics,

[1] $\mathcal{NLL}$ was first developed by Joachim Laubsch and John Nerbonne Hewlett-Packard Laboratories. A. Kader Diagne has since joined the effort at the German Research Center for Artificial Intelligence (DFKI).

the constraints are first unified before being passed to the interface. The interface itself is effected by a translation function which takes as input a string representation of a typed feature structure and outputs an $\mathcal{NLL}$ expression, suitable for processing by the $\mathcal{NLL}$ inference engine. The function is structured as a compiler—it reads the feature expression, parses it into an abstract syntax tree, performs tree-transformations to eliminate structural divergences, and substitutes corresponding $\mathcal{NLL}$ atoms to obtain a final translation.

Compiling expressions from a source into a target language is a widely applicable technology which we apply in NLP architectures. Here we compile feature structure descriptions into meaning representations, realizing the translator function of the facilitator in the federated agents' archictecture (above).

3.1 Parsing Feature Descriptions

The feature-based semantics interface had to meet the following requirements: (i) applicability to different formalisms implemented in different programming languages; (ii) minimal requirements for input/output data; (iii) satisfactory compilation time. The particular feature formalism is variable in that the feature structure description language is specified declaratively (cf. Appendix A), and is irrelevant to the core translation procedures. The implementation lanaguage may likewise be ignored, since we use the string representation of the feature module—which minimizes our input requirements. The employment of compiler-compiler technology provides satisfactory compilation time.

The interface inputs a string representing a feature structure description, e.g., in the language specified by the BNF presented below in Appendix A (only appropriately typed slots are considered by the compiler). This CONCRETE SYNTAX is parsed into an ABSTRACT SYNTAX TREE, which is in turn input to the transformation routines. The distinction between concrete and abstract syntax allows us to use a the same compilation routines for superficially different feature formalisms (and we have tested this against two formalisms). For example, the feature structure descriptions below result in the same abstract syntax tree illustrated in Figure 3:

Expression	**BNF Specification**
[(pred talk) (agent Jones) (theme NLP)]	conj ::= "[" av-term $*$ "]" av-term ::= "(" attribute value ")" . . .
[pred=talk, agent=Jones, theme=NLP]	conj ::= "[" av-term { "," av-term $*$ } "]" av-term ::= attribute=value . . .
[pred:talk agent:Jones theme:NLP]	conj ::= "[" av-term $*$ "]" av-term ::= attribute ":" value . . .

The implementation of the parser is carried out using the ZEBU compiler-compiler, a Lisp-Version of YACC, under an Allegro Common Lisp environment on a SUN Worksation. A prototypical version was first implemented under a REFINE programming environment, because REFINE provides high-level programming constructs to express syntactic transformations and allows the use and generation of Common Lisp programs.

3.2 Transforming Feature Specifications

The next compilation step is a tree transformation task, illustrated in Figure 4.

Different tasks have to be solved by the transformation submodule: (i) resolving structural divergences between the feature language on the one hand, and the semantic representation on the other; (ii) performing substitutions; (iii) filtering inappropriate features; and (iv) managing coreferences in the feature language; and (v) multiplying disjunctions out (not yet implemented). The use of compiler technologies accomplishes these tasks. Transformations are used to handle (i) divergences and (ii) substitutions; selective parsing (iii) filters; and a symbol table manages coreferences (iv). Thus compiler technology solves significant problems efficiently and generally.

The role of tree transformations in handling structural divergences is worth special mention. The output from the feature description parser is an abstract syntax tree which serves as the input to the transformation submodule. The transformation submodule consists of a *translator-function*, which combines calls to various auxiliary tree-traversal (and tree-transformation) functions. Two tree-traversal functions are used by the translator function: (1) a so-called PREORDER-TRANSFORMATION function for top-down traversing; and (2) a POSTORDER-TRANSFORMATION function for bottom-up traversing of the abstract syntax tree. Each of these functions takes as arguments not only the (abstract) tree specified by the root, but also a set of transformations that is applied to the nodes of the given tree. A pre/postorder-transformation function operates on the tree in several passes; only one rule is used in each pass. Each rule in the rule set may be used, therefore the choice may be nondeterministic. The algorithms terminate when no rule can perform any further change on the tree. Given this control, it is sometimes necessary to split the rule set into subsets that are used in different calls to the tree-traversal functions.

Example of transformation rules are:

- *fs-variable-to-nll-variable* which transforms a variable from the feature description (%INTEGER) into an $\mathcal{NLL}$ variable (?SYMBOL). Due to the presence of coreferences

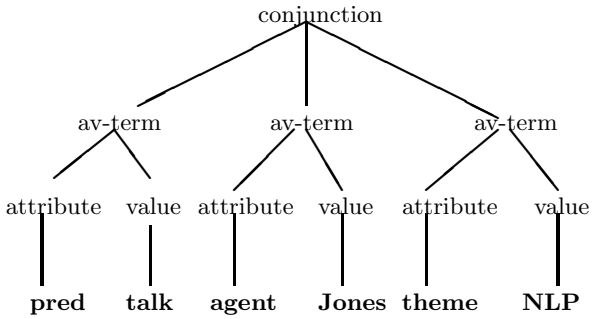

Figure 3: An abstract syntax tree for either " [(pred talk) (agent Jones) (theme NLP)] " or " [pred=talk, agent=Jones, theme=NLP] " or " [pred:talk agent:Jones theme:NLP] ".

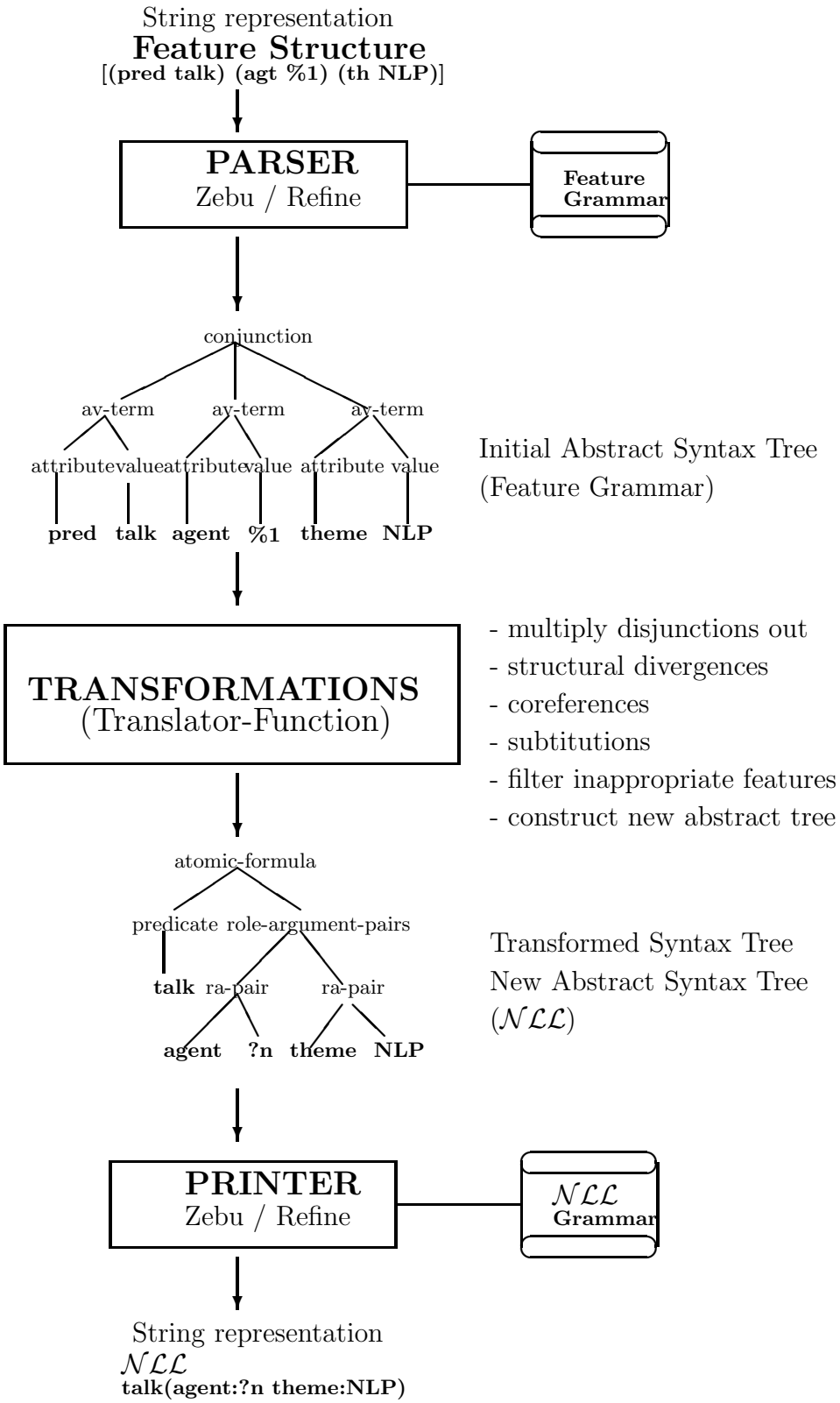

Figure 4: A feature-based semantics interface. Translating a feature structure description into $\mathcal{NLL}$. Partial specification of the sentence *Someone talks about NLP*.

this transformation must be done by generating symbols for new occurences of feature variables and keeping track of correspondences in a symbol table (usual in compiler technologies);

- *attribute-value-term-to-role-argument-pair* generates a $\mathcal{NLL}$ role-argument pair from an attribute-value-pair description, where the given attribute corresponds to a $\mathcal{NLL}$ role; etc.

The results, $\mathcal{NLL}$ expression(s) representing the possible meaning(s) of the given sentence, are likewise available in different formats, including strings.

4 Conclusion and Prospectus

The purpose of this paper has been to present a suitable and general approach for developing interfaces in an heterogeneous architecture for speech and language. We focused on the communication with the semantics module in particular the semantics interface, but a similar approach is realizable in other cases. We are working on other interfaces to the semantics module: (1) Discourse Memory (Attentional State) (2) Dialogue Management (Intentional State); (3) Knowledge Representation; (4) Phonology (esp. Intonation); and (4) Applications. $\mathcal{NLL}$ interfaces to applications like databases can be easily implemented by using translator-functions based on compilation (e.g. $\mathcal{NLL}$ to SQL interface [7] [6]).

References

[1] H. Alshawi et al. Research programme in natural language processing. Final report, Alvey Project No. ALV/PRJ/IKBS/105, SRI Cambridge Research Centre, July 1989.

[2] Bob Carpenter. *The Logic of Typed Feature Structures.* Number 32 in Tracts in Theoretical Computer Science. Cambridge University Press, Cambridge, 1992.

[3] Lutz Euler. Syntax des asl-attributenterm-formalismus. Technical Report 45-92/UHH, Verbundprojekt ASL, Fachbereich Informatik, Uni Hamburg, 1992.

[4] M.R. Genesreth. An agent-based framework for software interability. Technical report, Dept. of Computer Science, Stanford Univ, 1992.

[5] Per-Kristian Halvorsen. Situation semantics and semantic interpretation in constraint-based grammars. In *Proc. of the International Conference on Fifth Generation Computer Systems*, volume 2, pages 471–478, Tokyo, 1988. Institute for New Generation Systems.

[6] Joachim Laubsch. The semantics application interface. In Hans Haugeneder, editor, *Applied Natural Language Processing.* ??publisher, ??address, 1992.

[7] Joachim Laubsch. Logical form simplification. STL report, Hewlett-Packard, December 1989.

[8] Joachim Laubsch and John Nerbonne. An overview of \↕↕. Technical report, Hewlett-Packard Laboratories, Palo Alto, July 1991.

[9] R.E. Fikes M.R. Genesreth. Knowledge interchange format version 2 reference manual. Technical report, Dept. of Computer Science, Stanford Univ, 1991.

[10] John Nerbonne. Constraint-based semantics. In Paul Dekker and Martin Stokhof, editors, *Proc. of the 8th Amsterdam Colloquium*, pages 425–444. Institute for Logic, Language and Computation, 1992. also DFKI RR-92-18.

[11] John Nerbonne. Representing grammar, meaning and knowledge. In Susanne Preuss and Birte Schmitz, editors, *Proc. of the Berlin Workshop on Text Representation and Domain Modeling: Ideas from Linguistics and AI*, pages 130–147, Technische Universität Berlin, 1992. KIT FAST 97. also DFKI RR-92-20.

[12] Claudius Pyka. Architektur von asl-nord. Technical Report ASL-TR-4-91/UHH, Universität Hamburg, Fachbereich Informatik, Arbeitsbereich 'Natürlichsprachliche Systeme', 1991.

A BNF of Feature Description Grammar

The following BNF specifies the feature description grammar [3] used in the ASL Project.

feat-term	::=	variable
		| untagged-term
		| tagged-term
tagged-term	::=	variable "=" untagged-term
untagged-term	::=	atom
		| sort-expression
		| type conj
		| conj
		| disj
conj	::=	"[" attribute-value-term∗ "]"
disj	::=	"{" feat-term (","feat-term)∗ "}"
attribute-value-term	::=	"(" attribute feat-term ")"
sort-expression	::=	simple-sort
		| "@" sort
simple-sort	::=	primitive-sort
		| defined-sort
sort	::=	simple-sort
		| ∼ sort
		| "(" sort ("&" sort)+ ")"

		| “(” sort (“!” sort)+ “)”
attribute	::=	IDENTIFIER
atom	::=	IDENTIFIER
		| predefined-atom
predefined-atom	::=	INTEGER
primitive-sort	::=	IDENTIFIER
defined-sort	::=	IDENTIFIER
type	::=	IDENTIFIER
variable	::=	“%” POSITIVE-INTEGER