

CONCURRENT, OBJECT-ORIENTED NATURAL LANGUAGE PARSING: THE *ParseTalk* MODEL

Udo Hahn, Susanne Schacht & Norbert Bröker

CLIF - Arbeitsgruppe Linguistische Informatik/Computerlinguistik
Albert-Ludwigs-Universität Freiburg
Werthmannplatz
D-79085 Freiburg, Germany

email: {hahn, sue, nobi}@coling.uni-freiburg.de

Abstract

The *ParseTalk* model of concurrent, object-oriented natural language parsing is introduced. It builds upon the complete lexical distribution of grammatical knowledge and incorporates inheritance mechanisms in order to express generalizations over sets of lexical items. The grammar model integrates declarative well-formedness criteria constraining linguistic relations between heads and modifiers, and procedural specifications of the communication protocol for establishing these relations. The parser's computation model relies upon the actor paradigm, with concurrency entering through asynchronous message passing. We consider various extensions of the basic actor model as required for distributed natural language understanding and elaborate on the semantics of the actor computation model in terms of event type networks (a graph representation for actor grammar specifications) and event networks (graphs which represent the actor parser's behavior). Besides theoretical claims, we present an interactive grammar/parser workbench, a graphical development environment with various types of browsers, tracers, inspectors and debuggers that has been adapted to the requirements of large-scale grammar engineering in a distributed, object-oriented specification and programming framework.

1 INTRODUCTION

In this article, we present *ParseTalk*, a grammar model for natural language analysis that combines lexical organization of grammatical knowledge with lexicalized control of the corresponding parser in an object-oriented specification framework. This research takes recent developments in the field of linguistic grammar theory into account that have yielded a fine-grained lexical decomposition of linguistic knowledge. We here consider the rigid form of *lexical modularization* that has already been achieved as a starting point for the incorporation of lexicalized control at the grammar level. This local, distributed perspective on the *behavior* of the grammar system contrasts with previous designs of lexicalized grammars (for instance, Head-Driven Phrase Structure Grammar: Pollard and Sag, 1987; Categorical Grammar: Hepple, 1992; Lexicalized Tree Adjoining Grammar: Schabes, Abeille and Joshi, 1988), where lexical items still are considered passive data containers whose contents is uniformly interpreted by globally defined operations (unification, functional composition, tree adjunction, etc.). Diverging from these premises, we assign full procedural autonomy to lexical units and treat them as *active lexical processes* communicating with each other by message passing. Thus, they dynamically establish heterogeneous communication lines in order to determine each lexical item's functional role in the course of the parsing process. While the issue of lexicalized control has early been investigated in the paradigm of conceptual parsing (Riesbeck and Schank, 1978), and word expert parsing in particular (Small and Rieger, 1982), these approaches are limited in several ways. First, they do not provide any means for the systematic incorporation of grammatical knowledge and, furthermore, lack an appropriate grammar theory background on which linguistic descriptions could be based on. This, often, leads to *ad hoc*, unwieldy descriptions of linguistic phenomena lacking any deeper explanatory value. Second, they do not supply any organizing facility to formulate generalizations over sets of lexical items. This causes the danger of an enormous proliferation of grammar specifications whose consistency and completeness are hard to check and to maintain, mainly due to many interdependencies left unexplicated (this can directly be attributed to the apparent grammar theory deficits). Third, lexical communication is based on an entirely informal protocol that lacks any grounding in principles of distributed computing, thus precluding any attempts at reasoning about formal properties of the underlying grammar or parser.

We intend to remedy these methodological shortcomings by designing a radically lexicalized grammar on the basis of *valency* and *dependency* (these head-oriented notions already figure in different shapes in many modern linguistic theories, e.g., as subcategorizations, case frames, theta roles), by introducing *inheritance* as a major organizational mechanism (for a survey of applying inheritance in modern grammar theory, cf. Daelemans, De Smedt and Gazdar, 1992), and by specifying a *message passing* protocol for

local and directed communication between lexical objects that is grounded in the actor computation model (Agha and Hewitt, 1987). As this protocol allows for asynchronous message passing, *concurrency* enters as a theoretical notion at the level of grammar specification, not only as an implementational feature. The *ParseTalk* model introduced in this article can therefore be considered as an attempt to replace the static, global-control paradigm of natural language processing by a dynamic, local-control model. Given these requirements, the appeal of an object-oriented approach (for a survey, cf. Nierstrasz, 1989) to natural language processing becomes evident. It provides powerful mechanisms for distributing and encapsulating knowledge (based on the definition of objects and classes), for structuring declarative knowledge (in terms of inheritance hierarchies), and for specifying procedural knowledge (via message passing protocols). It is particularly the balanced treatment of declarative and procedural constructs within a single formal framework that makes the object-oriented approach a seemingly qualified candidate for the proper specification of "active", lexically distributed grammars.

The design of such a grammar and its associated parser responds to the demands of complex *language performance* problems. By this, we mean natural language understanding tasks, such as large-scale text or speech understanding, which not only require considerable portions of grammatical knowledge but also a vast amount of so-called non-linguistic, e.g., domain and discourse knowledge. A major problem then relates to the interaction of the different knowledge sources involved, an issue that is not so pressing when monolithic grammar knowledge essentially boils down to syntactic regularities. Instead of subscribing to any serial model of control, we build upon evidences from computational text understanding studies (Granger, Eiselt and Holbrook, 1986; Costantini, Fum, Guida, Montanari and Tasso, 1987; Yu and Simmons, 1990) as well as psycholinguistic experiments, in particular those worked out for the class of interactive language processing models (Marslen-Wilson and Tyler, 1980; Thibadeau, Just and Carpenter, 1982). They reveal that various knowledge sources are accessed in an *a priori* unpredictable order and that a significant amount of *parallel* processing occurs at various stages of the (human) language processor. Therefore, computationally and cognitively plausible models of natural language understanding should account for parallelism at the *theoretical* level of language description (for a recent survey of parallel computation models for natural language processing, cf. Hahn and Adriaens, 1994). Currently, *ParseTalk* provides a specification platform primarily for the computational aspects of language performance modeling.¹ In the future, however, we also envisage this vehicle as a testbed for the configuration and simulation of cognitively adequate parsers.

Moving language performance considerations to the level of grammar design is thus in strong contrast to competence-based approaches which assign structural well-formedness

conditions to the grammar level and leave their computation to (general-purpose) parsing algorithms, often at the cost of excessive amounts of ambiguous structural descriptions. In contradistinction, we aim at an integrated model of object-oriented grammar design and parser specification that incorporates a linguistically plausible model of lexicalization, a theory of lexical inheritance and a formal specification of the behavior of a lexically distributed parser. We thus purposively extend the realm of grammatical descriptions by the incorporation of control, i.e., procedural knowledge in terms of the formal specification of a message protocol that has been adapted to linguistic processing.

The article is organized as follows: In the next section, we briefly describe the information system background of the *ParseTalk* approach. Section 3 describes the theoretical framework of the *ParseTalk* model. First, we examine its underlying grammatical constructs with emphasis on valency constraints and dependency relations (Section 3.1), including associated grammatical and conceptual hierarchies. Then, we turn to the formal syntactic and semantic specification of the actor computation model, distinguishing between the common, so-called basic model (Section 3.2) and various extensions necessary for natural language parsing (Section 3.3). The issue of how dependency relations are actually established is treated in Section 3.4. In Section 4, we shift our attention from grammar theory to human-computer interaction aspects of grammar engineering by introducing various interactive tools (editors, browsers, inspectors, and debuggers) that have been assembled in *ParseTalk*'s grammar engineering workbench. This section also contains an elaborated sample parse intended to illustrate the theoretical principles discussed in the previous section. In Section 5, an event (type) network representation of the sample parse from Section 4 is given, recasting and generalizing the behavior of the parser in descriptive terms of the actor computation model as introduced in Section 3.2. One of the touchstones of grammar design is the way ambiguity is accounted for. Section 6 covers that issue by introducing the mechanisms with which lexical and structural ambiguity are dealt with. Drawing on the sample parse from Section 4, we illustrate how structural ambiguity is treated by an appropriate message passing protocol in Section 6.3. We conclude with a brief statement in Section 6.4 of how psycholinguistic performance models relate to the *ParseTalk* model discussed so far. In Section 7, we discuss the relationships the *ParseTalk* model has with previous research and how it differs from competing proposals.

¹ We only mention that performance issues become even more pressing when natural language understanding tasks are placed in real-world environments and thus additional complexity is added by ungrammatical natural language input, noisy data, as well as lexical, grammatical, and conceptual specification gaps. In these cases, not only multiple knowledge sources have to be balanced but additional processing strategies must be supplied to cope with these disturbing phenomena in a robust way. This puts extra requirements on the integration of procedural linguistic knowledge within a performance-oriented language analysis framework, viz. strategic knowledge how to handle incomplete or faulty language data and grammar specifications. However, these challenges are currently not met by the *ParseTalk* system.

2 *ParseTalk*'s INFORMATION SYSTEM CONTEXT

The development of the *ParseTalk* model and its implementation as a prototype system forms part of a larger project, viz. the development of a text understanding system that is geared towards knowledge assimilation from journal articles covering the domain of computer technology. Its long-term goal is the continuous augmentation of the underlying background knowledge by new concepts as they are acquired from text analysis (cf. Hahn (1989) for a consideration of the corresponding research framework). In particular, *ParseTalk* can be considered the language processing kernel of such an advanced information system. This environment places severe demands on the parser's linguistic and conceptual coverage as well as on suitable strategies guaranteeing *robust* natural language processing. In particular, it raises strong requirements as far as the system's capability is concerned to recognize and properly acquire knowledge about *new* concepts yet to be learned.

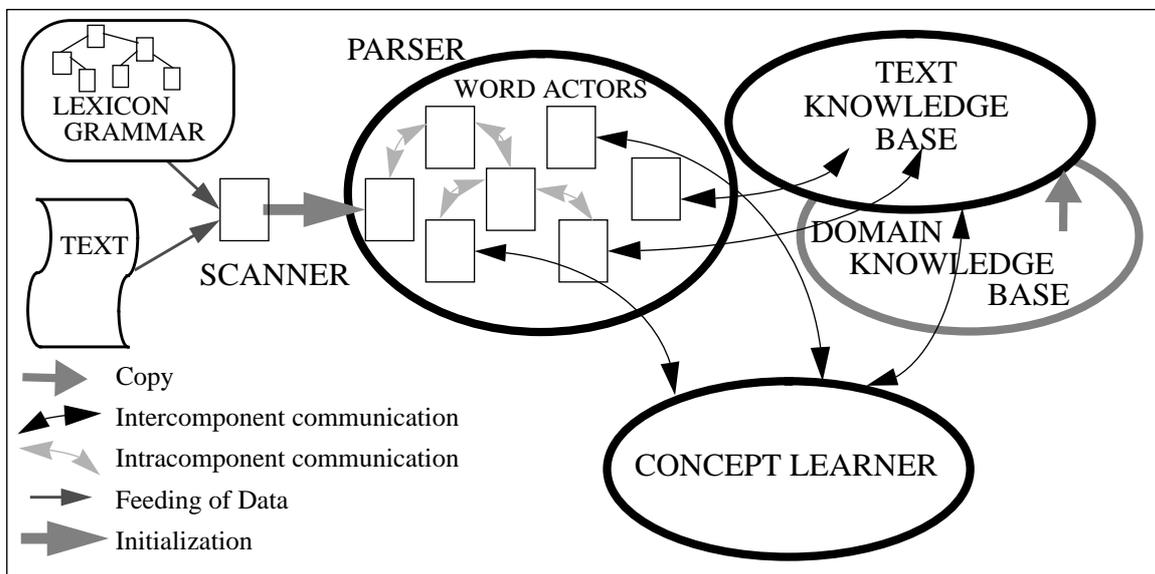


FIGURE 1. Sketch of the *ParseTalk* architecture

Fig. 1 provides an architectural overview of the language processing kernel. Three main components are distinguished: the parser, domain and text knowledge bases, and the concept learner. Any time a new *text* is entered into the system, a copy of the *domain knowledge base* is made, subsequently referred to as *text knowledge base*. The text knowledge base is incrementally modified as the text understanding process proceeds. Both these knowledge bases are implemented in LOOM, a hybrid, classification-based knowledge representation language (MacGregor and Bates, 1987). The text knowledge base is continuously queried and updated by the two remaining components, the parser and the concept learner. The *parser* is composed of a set of *word actors*, each representing (one reading of) a single word form from the input text (we currently use a full-form lexicon

that will soon be replaced by a morphological processor operating on a morpheme lexicon; cf. Bröker, Hanneforth and Hahn (1994) for a description of that prototype). The word actors themselves are created by a *scanner* module which reads the input text word by word, locates each word at the leaves of the *lexicon grammar*, and instantiates the corresponding object class description, thus incorporating information available from the text only (e.g., linear precedence of lexical items). After initialization, each word actor starts communicating with other word actors (intracomponent communication) and system components, such as the text knowledge and the concept learner (intercomponent communication). The *concept learner* is in charge of recognizing and assimilating new concepts into the text knowledge base (Klenner and Hahn, 1994), a process that requires intensive interaction, both with the parser and the text knowledge base, in order to properly balance linguistic and conceptual indicators for concept learning.

3 *ParseTalk's* CONCEPTUAL FRAMEWORK

The *ParseTalk* model is based on a fully lexicalized grammar. Grammatical specifications are given in the format of *valency constraints* attached to each lexical unit, on which the computation of concrete *dependency relations* is based (cf. Section 3.1). By way of inheritance the entire collection of lexical items is organized into *lexical hierarchies* (these constitute the *lexicon grammar*, cf. Fig. 1), the lexical items forming their leaves and the intermediary nodes representing grammatical generalizations in terms of word classes. This way of specification is similar to various proposals currently investigated within the unification grammar community (Evans and Gazdar, 1990). *ParseTalk's* concurrent computation model builds upon and extends the formal foundations of the *actor model*, a theory of object-oriented computation that is based on asynchronous message passing. We will elaborate on the semantics of actor computations (cf. Section 3.2) and the enhancements of the basic actor model to account for the specific requirements imposed by natural language processing (cf. Section 3.3). An application of these principles will be discussed considering a protocol for establishing concrete dependency relations (cf. Section 3.4).

3.1 The Grammar Model

The grammar model underlying the *ParseTalk* approach considers dependency relations between words as the fundamental notion of linguistic analysis. A *modifier* is said to depend on its *head* if the modifier's occurrence is allowed by the head but not *vice versa*².

² We extend this definition to incorporate the notion of phrases as well. Although phrases are not explicitly represented (e.g., by non-lexical categories), we consider each complete subtree of the dependency tree a phrase (this convention also includes discontinuous phrases). A dependency is not a relation between words (as in Word Grammar (Hudson, 1990, p.117), but between a word and a dependent phrase (as in Dependency Unification Grammar (Hellwig, 1988)). The root of a phrase is taken to be the representative of the whole phrase.

Dependencies, tagged by dependency relation names from the set $\mathcal{D} = \{\text{spec, subj, ppatt, ...}\}^3$, are thus asymmetric binary relations that can be established by local computations involving only two lexical items -- the prospective head-modifier pair. Co-occurrence restrictions between lexical items are specified as sets of *valencies* that express the various constraints a head places on permitted modifiers. These constraints incorporate the following descriptive dimensions:

1. **categorial:** $\mathcal{C} = \{\text{WordActor, Noun, Substantive, Preposition, ...}\}$ denotes the set of word classes, and $isa_{\mathcal{C}} = \{(\text{Noun, WordActor}), (\text{Substantive, Noun}), (\text{Preposition, WordActor}), \dots\} \subset \mathcal{C} \times \mathcal{C}$ denotes the subclass relation yielding a hierarchical ordering in \mathcal{C} (cf. Fig. 2).
2. **morphosyntactic:** A unification formalism (similar in spirit to Shieber, 1986) is used to represent morphosyntactic regularities. It includes atomic terms from the set $\mathcal{T} = \{\text{nom, acc, ..., sg, pl, ...}\}$, complex terms associating labels from the set $\mathcal{L} = \{\text{case, num, agr, ...}\} \cup \mathcal{D}$ with embedded terms, value disjunction (in curly braces), and coreferences (numbers in angle brackets). \mathcal{U} denotes the set of allowed feature structures, ∇ the unification operation, \perp the inconsistent element. Given $u \in \mathcal{U}$ and $l \in \mathcal{L}$, the *expansion* $[l : u]$ denotes the complex term containing only one label, l , with value u . If v is a complex term containing l at top level, the *extraction* $v \setminus l$ is defined to be the value of l in v . By definition, $v \setminus l$ yields \perp in all other cases. Therefore, if $v = [l : u]$ then $v \setminus l = u$.
3. **conceptual:** The concept hierarchy consists of a set of concept names $\mathcal{F} = \{\text{Hardware, Computer, Notebook, HardDisk, ...}\}$ and a subclass relation $isa_{\mathcal{F}} = \{(\text{Computer, Hardware}), (\text{Notebook, Computer}), \dots\} \subset \mathcal{F} \times \mathcal{F}$. Furthermore, the set of conceptual role names $\mathcal{R} = \{\text{HasCPU, HasHardDisk, ...}\}$ contains labels of conceptual relations (a frame-style, classification-based knowledge representation model in the spirit of MacGregor (1991) is assumed; cf. Fig. 3). The relation $cic \subseteq \mathcal{F} \times \mathcal{R} \times \mathcal{F}$ implements conceptual integrity constraints for permitted relations among concepts: $(f, r, g) \in cic$ iff any concept subsumed by $f \in \mathcal{F}$ may be related via $r \in \mathcal{R}$ to any concept subsumed by $g \in \mathcal{F}$, e.g., $(\text{Computer, HasHardDisk, HardDisk}) \in cic$. From cic the relation *permit* can be derived which explicitly states the range of concepts that can actually be related:
$$permit := \{(\chi, r, y) \in \mathcal{F} \times \mathcal{R} \times \mathcal{F} \mid \exists f, g \in \mathcal{F} : (f, r, g) \in cic \wedge \chi isa_{\mathcal{F}}^* f \wedge y isa_{\mathcal{F}}^* g\}$$
For brevity, we restrict this exposition to simple forms of taxonomic reasoning and do not consider assertional reasoning including quantification, etc. (cf., e.g., Creary and Pollard, 1985).

³ Additionally, \mathcal{D} contains the auxiliary symbol *self* which denotes the currently considered lexical item. This symbol occurs in feature structures (see 2. below) and in the ordering relations *order* and *occurs* (4. below).

4. **word order:** The (word class-specific) set $order \subset \mathcal{D}^n$ contains n-tuples which express ordering constraints on the valencies of each word class. Legal orders of modifiers must correspond to an element of $order$. For established dependencies, the (word-specific) function $occurs : \mathcal{D} \rightarrow \mathcal{N}_0$ associates dependency relation names with the modifier's (and self's) text position (the value "0" indicates dependencies not yet occupied). Both specifications appear at the lexical head only, since they refer to the head and all of its modifiers.

With these definitions, a valency can be characterized as an element of the set $\mathcal{V} \subset \mathcal{D} \times \mathcal{C} \times \mathcal{U} \times \mathcal{R}$. An illustration of the above criteria in Table 1 focuses on one specific dependency relation, the attachment of the prepositional phrase to the head noun, from the example sentence "*Compaq entwickelt einen Notebook mit einer 120-MByte-Harddisk*" [*"Compaq develops a notebook with a 120MByte hard disk"*], a complete parse of which is given in Section 4.

| | Attributes | lexical items (head underlined) | prior to dependency establishment | after dependency establishment |
|---|--|---|--|---|
| Possible Modifier | class $\in \mathcal{C}$ features $\in \mathcal{U}$ concept $\in \mathcal{F}$ position $\in \mathcal{N}$ | <i>mit einer <u>120-MByte- Harddisk</u></i> | Preposition $\left[\begin{array}{c} \text{self} \quad \left[\text{form mit} \right] \\ \text{pobj} \quad \left[\text{agr} \quad \left[\begin{array}{cc} \text{case} & \text{dat} \\ \text{gen} & \text{fem} \\ \text{num} & \text{sg} \end{array} \right] \right] \end{array} \right]$ 120MB-HARDDISK-00004 5 | Preposition $\left[\begin{array}{c} \text{self} \quad \left[\text{form mit} \right] \\ \text{pobj} \quad \left[\text{agr} \quad \left[\begin{array}{cc} \text{case} & \text{dat} \\ \text{gen} & \text{fem} \\ \text{num} & \text{sg} \end{array} \right] \right] \end{array} \right]$ 120MB-HARDDISK-00004 5 |
| Possible Head | features $\in \mathcal{U}$ concept $\in \mathcal{F}$ $order \subset \mathcal{D}^n$ $occurs : \mathcal{D} \rightarrow \mathcal{N}_0$ | <i>einen <u>Notebook</u></i> | $\left[\begin{array}{c} \text{self} \quad \left[\text{agr} <1> = \left[\begin{array}{cc} \text{case} & \text{acc} \\ \text{gen} & \text{mas} \\ \text{num} & \text{sg} \end{array} \right] \right] \\ \text{spec} \quad \left[\text{agr} <1> \right] \end{array} \right]$ NOTEBOOK-00003 {<spec, attr, self, ppatt>} {(spec,3),(attr,0),(self,4),(ppatt,0)} | $\left[\begin{array}{c} \text{self} \quad \left[\text{agr} <1> = \left[\begin{array}{cc} \text{case} & \text{acc} \\ \text{gen} & \text{mas} \\ \text{num} & \text{sg} \end{array} \right] \right] \\ \text{spec} \quad \left[\text{agr} <1> \right] \\ \text{ppatt} \quad \left[\text{form mit} \right] \end{array} \right]$ NOTEBOOK-00003 {<spec, attr, self, ppatt>} {(spec,3),(attr,0),(self,4),(ppatt,5)} |
| valencies (usually a set, here only a single valency is considered) | name $\in \mathcal{D}$ class $\in \mathcal{C}$ features $\in \mathcal{U}$ domain $\subseteq \mathcal{R}$ | | ppatt Preposition $\left[\text{ppatt} \quad \left[\text{form mit} \right] \right]$ {HasHardDisk, HasPrice, ...} | not applicable |

TABLE 1. An illustration of grammatical specifications in the *ParseTalk* model

The *feature* structure of the two heads, "*mit*" and "*Notebook*", is given prior to and after the establishment of the dependency relation. The relevant *concepts* of each of the phrases, 120MB-HARDDISK-00004 and NOTEBOOK-00003 (the extensions are irrelevant here,

but cf. Section 4), are stated. The *order* constraint for "Notebook" says that it may be preceded by a specifier (spec) and an attributive adjective (attr), while it may be followed by a prepositional phrase (ppatt)⁴. Considering a single valency for a prepositional phrase, the criteria described in the lower row state which *class*, *feature*, and *domain* constraints must be fulfilled by candidate modifiers.

We may now formulate the predicate SATISFIES which determines whether a candidate modifier fulfills the constraints stated in a specified valency at a candidate head (cf. Table 2). If SATISFIES evaluates to true, a dependency valency.name is established (object.attribute denotes the value of the property attribute at object). As can easily be verified, SATISFIES is fulfilled for the combination of "Notebook", the prepositional valency ppatt, and "mit" from Table 1.

| |
|---|
| <p>SATISFIES (modifier, valency, head) :\Leftrightarrow modifier.class <i>isa</i>_C* valency.class \wedge (([valency.name : (modifier.features \ self)] ∇ valency.features) ∇ head.features) $\neq \perp$ \wedge \exists role \in valency.domain : (head.concept, role, modifier.concept) \in <i>permit</i> \wedge \exists $\langle d_1, \dots, d_n \rangle \in$ head.order : \exists $k \in \{1, \dots, n\}$: (valency.name = d_k \wedge (\forall $1 \leq i < k$: (head.occurs (d_i) < modifier.position)) \wedge (\forall $k < i \leq n$: (head.occurs (d_i) = 0 \vee head.occurs (d_i) > modifier.position)))</p> |
|---|

TABLE 2. The SATISFIES predicate for valency checking

Note that unlike most previous dependency grammar formalisms (Starosta and Nomura, 1986; Hellwig, 1988; Jäppinen, Lassila and Lehtola, 1988; Fraser and Hudson, 1992) this criterion assigns equal opportunities to syntactic as well as conceptual conditions for computing dependency relations. Information on word classes, morphosyntactic features, and order constraints is purely syntactic, while conceptual compatibility introduces an additional descriptive layer to be satisfied before a grammatical relation may actually be established (cf. Muraki, Ichiyama and Fukumochi (1985) as well as Lesmo and Lombardo (1992) for similar approaches; cf. also the discussion of Fig. 12 and Fig. 13 in Section 4). Note also that we considerably restrict the scope of the unification operation in our framework, as only morphosyntactic features are subjected to this subformalism (this also explains why we implement unification in terms of the actor model, while Shapiro and Takeuchi (1983) propose the reverse direction, i.e., implementation of the actor model in terms of unification mechanisms). Our design, therefore, contrasts sharply with standard

⁴ Since valencies of lexical items do not only contain categorial constraints but simultaneously incorporate conceptual restrictions, they represent the range of possible arguments of the corresponding conceptual entity. We here assume that each argument position can be occupied only once, and consequently a valency can be filled by only one modifier. The example of the prepositional valency has thus been simplified in that a number of different more fine-grained valencies can be defined (such as instrument, part-of, manufacturer-of), each of which is distinguished by additional conceptual restrictions. This diversity is fused in terms of the single ppatt valency in our example.

unification grammars (and with approaches to dependency parsing as advocated by Hellwig (1988) and Lombardo (1992)), where virtually all information is encoded in terms of the unification formalism. Typed unification formalisms (cf., e.g., Emele and Zajac, 1990) would easily allow for the integration of word class information. Word order constraints and conceptual restrictions (such as value range restrictions or elaborated conceptual integrity constraints), however, are not so easily transferable, since they go far beyond the level of atomic semantic features still prevailing in unification formalisms. Kasper (1989) as well as Backofen, Trost and Uszkoreit (1991) consider similar approaches which relate LOOM-style classification-based knowledge representation systems to grammar formalisms for natural language processing based entirely on the unification paradigm.

3.1.1 A Look at Grammatical and Conceptual Hierarchies

The grammatical specification of a lexical entry incorporates structural constraints (valencies) and behavioral descriptions (communication protocols). In order to capture relevant generalizations and to support easy maintenance of grammar specifications, both are represented in hierarchies (cf. Genthial, Courtin and Kowarski (1990), Fraser and Hudson (1992) for inheritance within the dependency grammar paradigm that is restricted to structural criteria only). The *valency hierarchy* assigns valencies to lexemes. We will not consider it in depth here, since it captures traditional grammatical notions, like transitivity or reflexivity. The organizing principle is the subset relation on valency sets. The *word class hierarchy* contains word class specifications that cover distributional and behavioral properties. Fig. 2 illustrates both criteria.

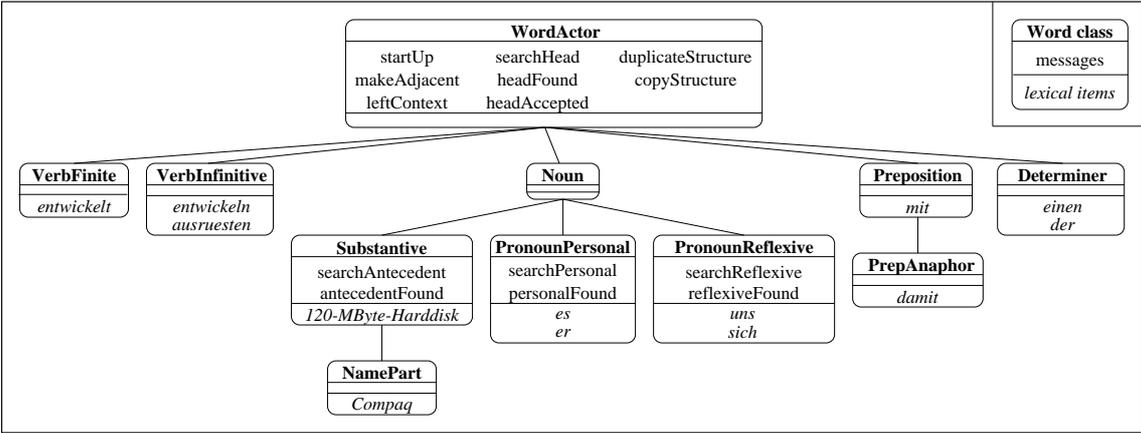


FIGURE 2. Fragment of the word class hierarchy

Finite verbs, infinitives, nouns, prepositions, and determiners are distinguished according to their different distribution (i.e., different valencies they can occupy). These distributional classes also determine which grammatical features can be specified for an instance of this class (e.g., person and number for finite verbs, case, gender, and number for nouns). The behavioral criterion is addressed by defining different message types for each class

(several messages for WordActor will be discussed in Section 3.4, 4 and 6.3). Within the Noun part of the word class hierarchy, there are different methods for anaphora resolution reflecting various structural constraints on possible antecedents for nominal anaphora, reflexives and personal pronouns. The word class hierarchy cannot be generated automatically, since classification of program specifications (communication protocols, in our case) falls out of the scope of state-of-the-art classifier algorithms.

On the other hand, the *concept hierarchy* contained in the text and domain knowledge bases is based on the subsumption relation holding between concepts, which is computed by a standard terminological classifier (MacGregor, 1991). Fig. 3 contains a small part of the concept hierarchy from the hardware domain. Most lexicon entries refer to a corresponding concept to allow conceptual restrictions occurring in valency constraints to be checked.

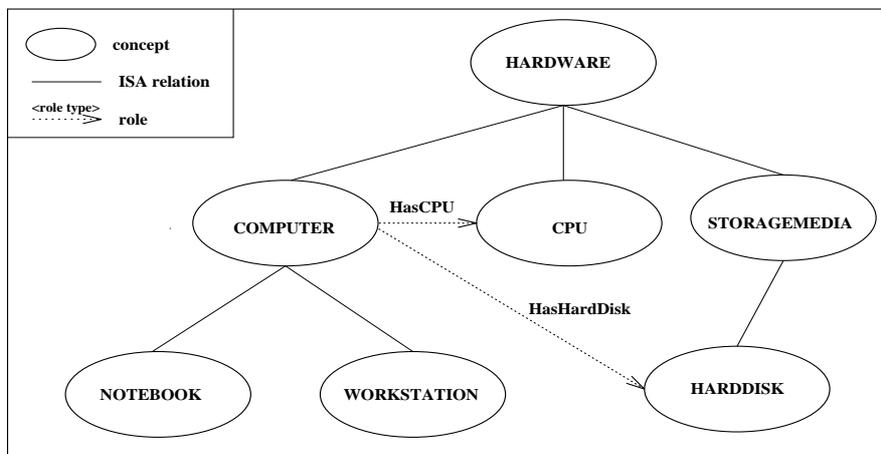


FIGURE 3. Fragment of the concept hierarchy

3.2 The Basic Actor Computation Model

Although the object-oriented paradigm features the distribution of data through encapsulation and the distribution of control via message passing, most object-based calculi rely on synchronous messages and do not encourage any use of concurrency. One of the few exceptions that aim at the methodologically clean combination of object-oriented features (encapsulation and inheritance-based sharing and re-use of structure and behavior) with concurrency and distribution is the actor model of computation (Agha and Hewitt, 1987). Unlike many other approaches which merely *add* concurrency to serial computation models by introducing explicit concurrency constructs into the definition of basically serial programming languages, the actor model is *designed* for concurrency, combining the well-understood foundations of functional programming with CSP-style process calculi (Hoare, 1985). Various formalisms for actor semantics have been proposed, ranging from purely denotational (Hewitt and Baker, 1978; Clinger, 1981) over behavioral (Greif, 1975; Yonezawa, 1977) to operational (Agha, 1986) specifications. The denotational approach is par-

ticularly adequate from a proof-theoretic point of view, while it lacks necessary abstractions for the specification of complex real-world domains. On the other hand, the operational approach is severely limited in its potential for formal reasoning about the behavioral properties of actor systems due to its lack of abstraction from concrete (actor) programming languages. As a compromise, we here favor a variant of a behavioral characterization of actor systems, since this format provides a solid platform for the formal characterization of actor programs, while at the same time it supports the description of complex, large-scale applications in a flexible way.

What we here call the *basic actor model* assumes a collection of independent objects, the *actors*, communicating via asynchronous message passing. An actor can send messages only to other actors it knows about, its *acquaintances*. A message arrival (called an *event*) can trigger the creation of new actors, the sending of further messages or changes of the receiving actor's acquaintances. All messages are guaranteed to be delivered and processed, but in an unpredictable order and indeterminate time. Each actor has an *identity* (its mail address), a *state* (the identities of its acquaintances) and a *behavior* (its scope of possible reactions to incoming messages determined by a collection of *methods*). Since new actors can be created and the communication topology between actors is reconfigurable as well, an actor system is inherently dynamic.

The basic actor model contains no synchronization primitives, but we assume *one-at-a-time* serialized actors for our specifications, i.e., actors that process only one message at a time and that process each message step by step (cf. Hewitt and Atkinson, 1979; Hewitt, Attardi and Lieberman, 1979). Consequently, the distribution of computations among the collection of actors is the only source of parallelism. Thus, depending on the chosen level of granularity in the design phase, an actor system exhibits a smaller or larger degree of parallelism according to the grain size of individual actors.

3.2.1 Syntax of the Basic Model

A *program* (in our application: a lexicon grammar) is given by a set of *actor definitions*. As part of such definitions, formal references to other actors, the *acquaintances*, are declared. An *actor* is an instantiation of a definition; the definition thus characterizes the type of an actor. An actor's behavior is determined by associated *method definitions* whose execution is activated by the arrival of a message at an actor. Messages themselves are composed of a *message key* and various *parameters*. The composite *action* underlying a particular method definition consists of simple actions, either a conditional action depending on local conditions (*if ... else*), the creation of a new actor (*create*), the sending of a message to an acquainted or a newly created actor (*send*), or the change of the actor's internal state (i.e., the specification of new acquaintances for itself, technically realized via

a become message). We may summarize these conventions in the (abstract) syntax $G1^5$ in Table 3.

| | | |
|------------|------------------|---|
| G1: | program | ::= actorDef* |
| | actorDef | ::= defActor actorType (acquaintance*) methodDef* |
| | methodDef | ::= <u>meth</u> messageKey (param*) action |
| | action | ::= action; action |
| | | <u>if</u> condition (action) [<u>else</u> (action)] |
| | | <u>send</u> actor messageKey (actor*) |
| | | <u>become</u> (actor*) |
| | actorType | ::= identifier |
| | messageKey | ::= identifier |
| | acquaintance | ::= identifier |
| | param | ::= identifier |
| | actor | ::= <u>self</u> <u>nil</u> <u>create</u> actorType (actor*) identifier |

TABLE 3. Abstract syntax of the basic actor model

We stipulate that actor definitions explicitly denote special actors that respond only to create messages supplying the required acquaintances as parameters for initialization. Note that messages are actors as well, as they have an identity and acquaintances (their parameters). In the basic model, however, they cannot receive messages themselves, since they lack any specification of behavior.

3.2.2 Semantics of the Basic Model: Event Type Networks and Event Networks

We here define the semantics of an actor program in terms of two kinds of networks. First, we consider *event types* which refer to message keys and can easily be determined from a given actor program. Next, we turn to actual events that involve instantiated actors and messages. Both, event types and events, are partially ordered by the transitive closures of two relations among them, *causes^t* and *causes*, respectively, that determine *event type networks* and *event networks*.

Given an actor program, *event types*, written as [$* \leftarrow \text{key}$], can be syntactically determined by the keys of messages that are sent. Let an actor type *aName* be defined by:

```

defActor aName (acquaintance1 ... acquaintancek)
  meth key1(param1 ... paramm) action1
  ...
  meth keyn (param1 ... paraml) actionn

```

In order to relate a given source message key name from one of the methods of an actor type to all those key names occurring in the method definitions of the considered actor type, we map message keys to sets of message keys by defining the function $script_{aName}$ as follows:

⁵ Let actor stand for an instantiated actor (a token) and **actorType** stand for a definition to be instantiated (a type, always in bold face); self in a method definition is used to express self-referentiality of the actor that is processing the method, nil denotes the undefined value.

$script_{a\mathcal{N}(ame)}: \mathcal{Keys} \rightarrow 2^{\mathcal{Keys}}$ such that

$script_{a\mathcal{N}(ame)}(key_i) = send(action_i)$ with

$$send(action) := \begin{cases} \{msgKey\} & \text{if action} = \underline{send} \text{ actor msgKey (actor}_1, \dots) \\ send(a_1) \cup send(a_2) & \text{if action} = \underline{if} \text{ condition } a_1 \underline{else} a_2 \\ send(a_1) & \text{if action} = \underline{if} \text{ condition } a_1 \\ send(a_1) \cup send(a_2) & \text{if action} = a_1; a_2 \\ \emptyset & \text{else} \end{cases}$$

For an actor program $P \in L(G1)$ $causes^t$ can be defined, a relation between event types that determines the send expressions that can be provoked by the message m with key $mKey$:

$$([* \leftarrow mKey], [* \leftarrow nKey]) \in causes^t : \Leftrightarrow \exists a : (\underline{defActor} a \dots \in P \wedge nKey \in script_a(mKey))$$

Turning to actual events now, we define an actor χ as being composed of an *identity* n (taken from the set of natural numbers, \mathcal{N}), a *state* $\in \mathcal{S}$ and a *behavior* $\in \mathcal{B}$. Hence, \mathcal{A} , the set of actors, is a subset of $\mathcal{N} \times \mathcal{S} \times \mathcal{B}$.

Let us further define the set of states $\mathcal{S} := 2^{\{(y:z) \mid y \text{ is an identifier, } z \in \mathcal{A}\}}$, with an element of \mathcal{S} associating acquaintance names and acquaintances values, the latter being actors. Since actors change the values of their attributes, their state is valid *in time*, i.e., at a particular event. The state of an actor a receiving a message m will be written as $s_{a,[a \leftarrow m]}$. State changes caused by the message (via a become action) apply at the end of the event $[a \leftarrow m]$ only.

\mathcal{B} is a set of functions capturing the behavior of actor systems and is defined as follows: Each state $s_{\chi,e}$ of an actor χ at an event e (the reception of a message m) is determined by χ 's initial state given after its creation event, and the repeated application of its state transition function, $transit_\chi$, which (implemented via become actions) maps pairs of states ($s \in \mathcal{S}$) and messages ($m \in \mathcal{M} \subset \mathcal{A}$) to new states:

$$transit_\chi : (\mathcal{S} \times \mathcal{M}) \rightarrow \mathcal{S}.$$

The send actions which an actor χ performs at a particular event are expressed as pairs of target actors and messages to be sent; the target actors are either acquaintances of the sending actor or supplied as message parameters. They are determined by the function

$$task_\chi : (\mathcal{S} \times \mathcal{M}) \rightarrow 2^{(\pi_1(\mathcal{A}) \times \mathcal{M})},$$

where $\pi_1(\mathcal{A})$ denotes the projection onto the first component of \mathcal{A} , viz. \mathcal{N} .

The *behavior* of an actor χ can then be stated by the function $behave_\chi \in \mathcal{B}$ that combines $transit_\chi$ and $task_\chi$ in that it maps pairs of states and messages to pairs consisting of the new state of the actor and a set of pairs of target actor identities and messages, viz.,

$$behave_\chi : (\mathcal{S} \times \mathcal{M}) \rightarrow (\mathcal{S} \times 2^{(\pi_1(\mathcal{A}) \times \mathcal{M})}).$$

Abstracting from a local actor perspective, the behavior of an entire actor system (in our application: the lexical parser composed of a collection of word actors) is determined by the way multiple events are related under the *causes* relation (though events are written as [actor \leftarrow message], only the message key will later be used, if no confusion can arise from that shorthand notation):

$$([a \leftarrow m], [b \leftarrow n]) \in \text{causes} : \Leftrightarrow (\pi_1(b), n) \in \text{task}_a(s_{a,[a \leftarrow m]}, m).$$

Events that are not ordered by the transitive closure of *causes* can take place in parallel or, if they refer to the same actor, in an arbitrary order. Though *event type networks* provide a global view on the behavioral aspects of our grammar specification, the current formalism still lacks the ability to support formal reasoning about general computational properties of distributed systems, such as deadlock freeness, fairness, termination, etc. Similarly, *event networks* comprise the computations performed and logical dependencies occurring during real parses, but they do not allow predictions in the general case. Providing an explicit type discipline for actor definitions might be a reasonable approach to fill the methodological gap between both layers of description.

3.3 Elements of the *ParseTalk* Extension

The *ParseTalk* model extends the formal foundations of the basic actor computation model, as discussed in the previous section, according to the requirements set up by the natural language processing application. The extensions, however, are expressible by the primitives of the basic actor model. In this sense, we here introduce *word actors*, *relations* between word actors and a special set of *messages* word actors may exchange; some remarks to synchronous local computations are also included.

3.3.1 Informal Description of the *ParseTalk* Extensions

The natural language parsing task constitutes a particularly challenging application of the actor model, since it shares all required attributes of the *open systems* metaphor, viz., it is open-ended, it is inherently dynamic, heterogeneous sources of knowledge must be combined, incomplete specifications must be accounted for, etc. (cf. Hewitt and de Jong, 1983). The grammatical knowledge associated with each lexical item is represented in a word actor definition. The acquaintances of its instances are actors which stand for the lexical item's morphosyntactic features, its conceptual representation, valency constraints and, after instantiation and subsequent parsing events, governed lexical items and further grammatical relations (e.g., adjacency, textual relations). A word actor system thus extends classical actor systems by the actors' rich internal structure and their heterogeneous communication requirements that exceed those of most sample actor collections discussed so far (cf., e.g., models of post offices (Yonezawa, 1977), flight reservation sys-

tems (Yonezawa and Hewitt, 1979), and printer queues (Hewitt, Attardi and Lieberman, 1979)). The extensions, in particular, encompass:

- **word actor relations:** Acquaintances of word actors are considered to be *tagged* according to linguistic criteria in order to serve as navigation aids in linguistic structures (the message distribution mechanism described below), thus significantly reducing the parser's search space. Textual relations, for example, are distinguished from linear adjacency and hierarchical dependency relations. Tagging imposes a kind of "typing" onto acquaintances that is missing in any actor system previously described.
- **word actor messages:** In contrast to simple messages which unconditionally trigger the execution of the corresponding method at the receiving actor, we define *complex word actor messages* as full-fledged actors with independent computational capabilities. Departure and arrival of complex messages are actions which are performed by the message itself, taking the sender and the target actors as parameters. Upon arrival, a complex message determines whether a copy is forwarded to selected acquaintances of its receiver and whether the receiver may process the message on its own. Hence, we redefine an arrival event to be an uninterruptable sequence of a computation event and distribution events. The *computation event* corresponds to an arrival of a simple message at the receiving word actor, i.e., an event in the basic model; it consists of the execution of an actor's method that may change the actor's state and trigger additional messages. The *distribution events* provide for the forwarding of the message and are realized by creating new complex messages. They depend on the (unchanged) state of the receiving actor or on the result of the computation event and take place before and after the computation event. This extension accounts for the complexity of interactions between knowledge sources one usually encounters in natural language understanding environments, an issue already addressed in the introductory section.
- **local computations:** To compute complex, but well understood and locally determined linguistic conditions and functions, such as unification of feature structures and queries sent to a (conceptual) knowledge base, we establish a synchronous request-reply protocol (cf. Lieberman (1987) for the specification of synchronous message passing in terms of asynchronous message passing for the actor language Act1). It is used for delivering an actor address to the sender of a request, in syntactic terms:

actor ::= ... | ask actor messageKey (actor*)

Furthermore, the selection of an acquaintance of an actor can be expressed as:

actor ::= ... | actor.acquaintance

Although, strictly speaking, there is no assignment to variables in the actor model, we introduce a loop expression, enumerating each element of a collection and binding it to an identifier:

action ::= ... | for var in actor : (action)

In the following subsection, we give a formal syntactic account of these extensions concentrating on complex word actor messages. Their consideration will also lead to the replacement of G1 by an updated abstract syntax, G2.

3.3.2 Syntax for the *ParseTalk* Extensions

In Table 4, we introduce syntactic conventions for ease of specification and enhanced readability of the *ParseTalk* specifications.

| | |
|--|---|
| defMsg messageType (param ₁ ... param _n) | |
| (if distributeCondition <u>distributeTo</u> distTag) | # if distributeCondition is true, distribution to |
| ... | # tagged acquaintances takes place, before any |
| | # computation process is performed |
| (if computeCondition <u>compute</u>) | # if computeCondition is true the message is sent |
| | # to the target (one of the parameters) |
| (if distributeCondition <u>distributeTo</u> distTag) | # as above, after computation processes have |
| ... | # been performed |

TABLE 4. Schema for the definition of complex message types

Unlike simple messages, the creation (and instantiation) of word actor messages is separated from their sending. This allows actors to store messages for later use and to explicitly define their reaction to an initial simple start-up message, viz. depart.

The definition of the above message form can be considered a macro which abbreviates the following low-level specifications referring to the constructs already available from the basic actor model as shown in Table 5.

| | |
|---|---|
| defActor actorMessageKey (sender target relationTag param*) | |
| <u>meth</u> depart () | # the message starts running |
| if not (target = nil) | # a direct target is specified |
| (send self arriveAt (target)) | # sending to itself, cf. arriveAt method below |
| else (if not (relationTag = nil)) | # a relation tag is supplied, so the message is |
| | # distributed to tagged acquaintances (next line) |
| (for x in sender.relationTag: | |
| send (create actorMessageKey (sender x relationTag param*)) depart)) | |
| <u>meth</u> arriveAt (actor) | # here the distribution and computation events from |
| | # the above abbreviation are modelled |
| if distributeCondition | # condition may refer to self and target |
| (send (create actorMessageKey (actor nil distTag param*)) depart); | # self-forwarding along relation tagged by distTag |
| if computeCondition | # condition may refer to self and target |
| (send actor actorMessageKey (sender target nil param*)) | # execute corresponding method at target |
| if distributeCondition | # condition may refer to self and target |
| (send (create actorMessageKey (actor nil distTag param*)) depart); | # self-forwarding along relation tagged by distTag |

TABLE 5. Message definition according to the basic actor model

Summarizing these conventions, we replace the original syntax G1 by the grammar G2 contained in Table 6 ('×' indicates the extensions relative to G1).

| | | |
|------------|--------------------|---|
| G2: | program | ::= { actorDef msgDef }* |
| × | msgDef | ::= <u>defMsg</u> messageType (param*) {(if distributeCondition <u>distributeTo</u> distTag)}* (if computeCondition <u>compute</u>) {(if distributeCondition <u>distributeTo</u> distTag)}* |
| | actorDef | ::= <u>defActor</u> actorType (acquaintance*) methodDef* |
| | methodDef | ::= <u>meth</u> messageKey (param*) action |
| | action | ::= action; action <u>if</u> condition (action) [<u>else</u> (action)] <u>send</u> actor messageKey (actor*) <u>become</u> (actor*) <u>for var in</u> actor : (action) <u>send</u> actorMessage depart |
| × | messageType | ::= identifier |
| | actorType | ::= identifier |
| | messageKey | ::= identifier |
| | acquaintance | ::= identifier |
| | param | ::= identifier |
| | actor | ::= <u>self</u> <u>nil</u> <u>create</u> actorType (actor*) identifier |
| × | actorMessage | ::= <u>ask</u> actor messageKey (actor*) actor.acquaintance |
| × | actorMessage | ::= <u>create</u> messageType (actor*) |

TABLE 6. Abstract syntax for the *ParseTalk* actor model

Due to the inclusion of the synchronous message passing construct ask, the function $script_{a\mathcal{N}(ame)}$ must be extended accordingly:

$script_{a\mathcal{N}(ame)}: \mathcal{Keys} \rightarrow 2^{\mathcal{Keys}}$ such that

$script_{a\mathcal{N}(ame)}(key_i) = send(action_i)$ with

$$send(action) := \begin{cases} \{msgKey\} \cup send(actor) \cup \cup_i send(actor_i) & \text{if action} = \underline{send} \text{ actor msgKey (actor}_1, \dots, \text{actor}_n) \\ \cup_i send(actor_i) & \text{if action} = \underline{become} \text{ (actor}_1, \dots, \text{actor}_n) \\ send(actorMsg) & \text{if action} = \underline{send} \text{ actorMsg depart} \\ send(a_1) \cup send(a_2) & \text{if action} = \underline{if} \text{ condition } a_1 \underline{else} a_2 \\ send(a_1) & \text{if action} = \underline{if} \text{ condition } a_1 \\ send(a_1) \cup send(a_2) & \text{if action} = a_1; a_2 \\ \emptyset & \text{else} \end{cases}$$

and, for actor expressions,

$$send(actor) := \begin{cases} \{msgKey\} \cup \cup_i send(actor_i) & \text{if actor} = \underline{ask} \text{ actor}_1 \text{ msgKey (actor}_2, \dots, \text{actor}_n) \\ \cup_i send(actor_i) & \text{if actor} = \underline{create} \text{ actorType (actor}_1, \dots, \text{actor}_n) \\ \emptyset & \text{else .} \end{cases}$$

3.3.3 An Example: the Scanner Actor

The following example introduces a simple service actor for the *ParseTalk* system, the Scanner (cf. Table 7), whose instances accept two kinds of messages: one for its initialization (**scan**⁶) and another one for scanning the next token and starting up the appropriate word actors (**scanNext**). The following acquaintances are supplied: the text under analysis, the current word considered, the lexicon used, the current position in the text, the first lexical item of the text, and the last lexical item read before the current one.

```

defActor Scanner (text word lexicon position first last)
  meth scan (aText aLexicon)
    send self scanNext ();
    become (aText (ask lexicon lookUp ((ask aText token) position)) aLexicon 1 nil nil)
    # the message token extracts the next word from a text
    # lexicon lookUp creates an initialized word actor with
    # this word's surface and the current position. This actor
    # takes the place of word.

  meth scanNext ()
    if (not (word = nil)) (send word startUp (last));
    # word is informed to start computing, supplied with the
    # last initialized actor as its left neighbor

    if (first = nil) (become
      (text (ask lexicon lookUp ((ask aText token) position)) lexicon (position+1) word word))
    else (become
      (text (ask lexicon lookUp ((ask aText token) position)) lexicon (position+1) first word))
    # the scanner changes its state, preparing to accept the
    # next scanNext message sent by the instantiated actor

```

TABLE 7. Definition of the Scanner actor type

Note that the **scanNext** message sent in **scan** cannot be received until **scan** is completely processed. Since the **become** action has already occurred, the acquaintance **word** has already changed when **scanNext** is executed. The fragment of an event type network incorporates all event types an instance of Scanner might take part in (Fig. 4).

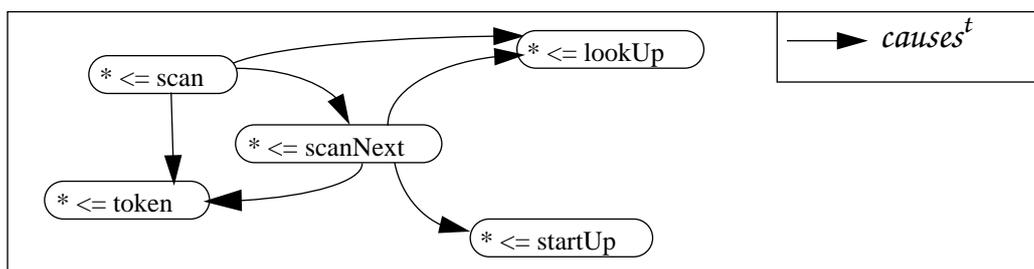


FIGURE 4. Event type network for scanning

Additionally, let us consider the corresponding network of events illustrating the scanning of two text words taken from our parsing example "*Compaq entwickelt...*" (cf. Sec-

⁶ In the following, all messages are written with their keys in bold face.

tion 4 for a thorough discussion of the associated dependency parse). Sending **scan** -- with the parameters `aText` and `aLexicon` left out in Fig. 5 -- to an instance of `Scanner`, viz. `scanner`, initializes the receiver via `become` (not represented as an event), binding the parameters to the appropriate acquaintances, and, especially, setting the acquaintance word to the actor that is returned by the expression `ask aText token`. Then, **scanNext** is sent to `self`. **scanNext** triggers sending **lookUp** to `lexicon` and **startUp** to the returned actor. Also, the reception of the next **scanNext** message is prepared by getting the next token from `text`. Note the double occurrence of `[...<= token]`. Two *different* messages are sent to the *same* actor in *different* states (indicated by the sliding position marker "`▲`").

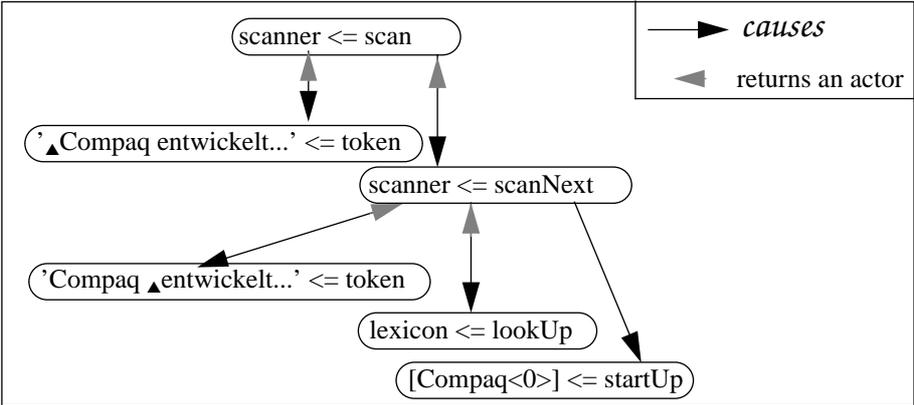


FIGURE 5. Event network for scanning "Compaq entwickelt ..." ["Compaq develops ..."]

3.4 A Simplified Protocol for Establishing Dependency Relations

The protocol described in this section allows word actors to establish dependency relations between each other according to the valency constraints as considered in Section 3.1. Additionally, structural restrictions on the resulting dependency trees are incorporated in the protocol (Section 3.4.2). To provide for domesticated concurrency as required for adequate linguistic and cognitive processing (Clark and Gibson, 1988), a receipt protocol allows synchronization between word actor instantiation (incremental reading of lexical items) and parsing activities, both taking place at a particular text position (Section 3.4.1). Concurrency then results from checking and negotiating alternative dependency structures in parallel (Section 3.4.3).

3.4.1 Synchronizing Actors: the Receipt Protocol

The receipt protocol enables an actor to determine when all events (transitively) caused by a message have terminated. This is done by sending replies back to the initiator of the computation. Since complex messages can be quasi-recursively forwarded, the number of replies cannot be determined in advance. In addition, each actor receiving such a message may need an arbitrary amount of processing time to terminate the actions caused by the message (e.g., the establishment of a dependency relation requires communication via

messages that takes indeterminate time). Therefore, each actor receiving the message must reply to the initiator once it has terminated processing, informing the initiator to which actor(s) the message has been forwarded.

A message is a *reception message* if (1) the receiver is required to (asynchronously) reply to the initiator with a receipt message, and (2) the initiator queues a reception task. An (*explicit*) *receipt message* is a direct message containing a set of actor identities as a parameter. This set indicates to which actors the reception message has been distributed. The enclosed set enables the receiver (which is the initiator of the reception message) to wait until all receipt messages have arrived.⁷ In addition to explicit receipts, which are messages solely used for termination detection, there are regular messages that serve a similar receipt purpose besides their primary function within the parsing process. They are called *implicit receipt messages* (an example of which is **headAccepted** described in Section 3.4.3)⁸. Summing up, a *reception task* consists of a set of partial descriptions of the (explicit as well as implicit) messages that must be received, and an action to be executed after all receipts have arrived (usually, sending a message).

3.4.2 Encoding Structural Restrictions

As already mentioned, structural restrictions on dependency trees are encoded in the message protocol. Word actors conduct a bottom-up search for possible heads; the principle of non-crossing arcs (i.e., *projectivity* of the dependency tree) is guaranteed by the following forwarding mechanism. Consider the case of a newly instantiated word actor w_n searching for its head to the left (the opposite direction is handled in a similar way). In order to guarantee projectivity one has to ensure that only word actors occupying the outer fringe of the dependency structure (between the current absolute head w_j and the rightmost element w_{n-1}) receive the search message of w_n (these are circled in Fig. 6).⁹ Since no structural restrictions are specified in the predicate SATISFIES (cf. Section 3.1), only word actors that are structurally legal heads must be addressed. This is reflected in the simplified message definition in Table 8.

⁷ This, of course, only happens, if the distribution is limited: The **searchHead** message discussed below is only distributed to the head of each receiver, which must occur in the same sentence. This ensures a finite actor collection to distribute the message to, and thus guarantees that the reception task is eventually triggered.

⁸ The use of regular messages as implicit receipts allows to reduce the number of messages being sent, but requires that the termination detection scheme must be redesigned for each new task. We plan to implement a generalized synchronization scheme (according to Shavit and Francez, 1986) for all protocols, moving this capability to our extensions of the actor model.

⁹ Additionally, w_n may be governed by any word actor governing w_j , but due to the synchronization implemented by the receipt protocol, each head of w_j must be located to the right of w_n .

```

defMsg searchHead (sender target initiator)
  (if GOVERNED (target) distributeTo head) # forward a copy to head, identified by head ∈  $\mathcal{D}$ , if
                                             # the targeted word actor is already governed by it
  (if true compute)                        # the message is always processed at the target;
                                             # the computation event is concretized in the word
                                             # actor specification below (see Section 3.4.3)

```

TABLE 8. Definition of the **searchHead** message type

Thus, a message searching for a head of its initiator is always locally processed at each actor receiving it, and is forwarded to the head of each receiver, if one already exists.

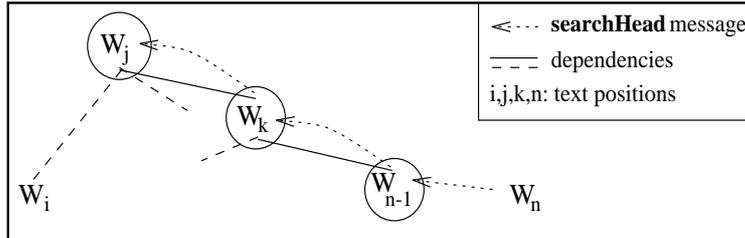


FIGURE 6. Forwarding a search message

Additionally, *direct messages* are used to establish a dependency relation. They involve no forwarding and may be specified as indicated by the template in Table 9. A number of direct messages are used for negotiating concrete dependencies, e.g., **headFound**, **headAccepted**, **receipt** (each with different parameters, as represented by "... above).

```

defMsg <directMessage> (sender target ...)
  (if true compute)
  # a direct message is always processed at the target, no distribution condition can apply

```

TABLE 9. Template for the definition of direct message types

3.4.3 An Excerpt from the Word Actor Script

The word actor protocol for bottom-up establishment of dependencies consists of three steps (cf. also Fig. 2): The search for a head (**searchHead**), the reply of a head, which satisfies the valency constraints, to the initiator of the search (**headFound**), and the acceptance by the initiator (**headAccepted**), thereby actually becoming a modifier of the determined head. Referring to the sample sentence in Section 4, "*Compaq entwickelt einen Notebook mit einer 120-MByte-Harddisk*" [*"Compaq develops a notebook with a 120MByte hard disk"*], consider the configuration in Fig. 7. The word actor which stands for the lexical item "*mit*"¹⁰ will send **searchHead** to [Notebook]. This message will immediately be forwarded to [entwickelt], since [Notebook] is governed by [entwickelt]. Nevertheless, only [Notebook] can accept the application (cf. Section 4 for considering the rea-

¹⁰ Actors representing a lexical item "*x*" will subsequently be written as [x].

sons which exclude [entwickelt] as a legal head), and thus replies by sending **headFound** back to [mit]. [mit] accepts [Notebook] as head and acknowledges by replying with **headAccepted**. This three-step protocol is necessary for the proper handling of ambiguities, which will be considered in Section 6.

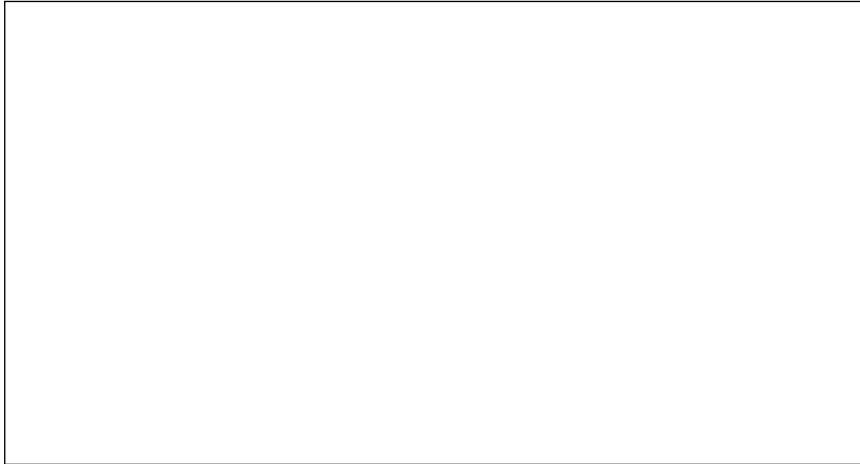


FIGURE 7. Snapshot when [mit] starts searching for its head

The corresponding method definitions are given in Table 10. The protocol allows alternative attachments to be checked concurrently, since each actor receiving a **searchHead** message may process it locally, while that message is simultaneously distributed to its head.

| | | | |
|-----------------------|----------------------------|---|--|
| <code>defActor</code> | <code>wordActor</code> | (head deps vals feats ...) | # head, dependencies, valencies, and features # are the acquaintances of the wordActor |
| <code>meth</code> | <code>searchHead</code> | (sender target init) | # processed at candidate heads, init denotes the initiator of the message # (<code>compute</code> part from the formal message definition) |
| | <code>for val in</code> | <code>vals:</code> | # check all valencies of the possible head |
| | <code>(if</code> | <code>SATISFIES</code> | (init val self) # valency check adapted from Table 2 |
| | <code>(send</code> | <code>(create</code> | <code>headFound</code> |
| | <code>(self</code> | <code>init</code> | <code>val.name</code> |
| | <code>feats</code> | <code>\val.name</code>) | <code>depart;</code> |
| | | | # reply to initiator init, restricting its morphosyntactic features |
| | <code>become</code> | (head deps vals (feats ∇ init.feats) ...) | # expand grammatical description of head |
| | <code>else</code> | <code>(send</code> | <code>(create</code> |
| | <code>receipt</code> | <code>(self</code> | <code>init</code> |
| | <code>{head}</code>) | <code>depart)</code> | # send a receipt with the head the message was forwarded to, depart realizes the departure of a complex message |
| <code>meth</code> | <code>headFound</code> | (sender target name headFeats) | # processed at the initiator of a searchHead message |
| | <code>send</code> | <code>(create</code> | <code>headAccepted</code> |
| | <code>(self</code> | <code>sender</code> | <code>name)</code> |
| | <code>depart;</code> | | # reply to head |
| | <code>become</code> | (sender deps vals (feats ∇ headFeats) ...) | # store sender as head of <code>self</code> , restrict <code>self</code> 's features through unification |
| <code>meth</code> | <code>headAccepted</code> | (modifier target name) | # processed at the head only |
| | <code>for</code> | <code>dep</code> | <code>in</code> |
| | <code>deps:</code> | | # check all dependencies |
| | <code>(if</code> | <code>(name =</code> | <code>dep.name)</code> |
| | <code>(send</code> | <code>dep</code> | <code>store</code> |
| | <code>(modifier))</code> ; | | # send the dependency the message store in order to record the modifier |
| | <code>send</code> | <code>(create</code> | <code>receipt</code> |
| | <code>(self</code> | <code>modifier</code> | <code>{head}</code>) |
| | <code>depart</code> | | # send a receipt with the head the message was forwarded to |

TABLE 10. Method definitions for `searchHead`, `headFound`, `headAccepted`

4 THE INTERACTIVE DEVELOPMENT ENVIRONMENT

This section considers the human-computer interaction issues arising in a natural language processing application which is based on the principles of concurrent, object-oriented parsing. The technical infrastructure we consider, on the one hand, reflects the demand for a grammar development and parser engineering *workbench* dedicated to the particular challenges of the object-oriented specification framework (distribution of knowledge and control, reusability of code, etc.). On the other hand, the tools we discuss allow the developers to *monitor* the distributed computations of a concurrent parsing engine in a comprehensible way, thus easing the testing, maintenance and debugging of grammar specifications and program code.

The availability of a proper development environment for grammars and parsers is of major relevance for projects dealing with non-trivial subsets of natural languages and aiming at a sufficient linguistic coverage of the associated grammar specifications (cf., e.g., Haugeneder and Gehrke, 1986; White, 1987; Boguraev, Carroll, Briscoe and Grover, 1988; Nakamura, Tsujii and Nagao, 1988). These systems provide a coherent framework for entering and updating linguistic specifications via rule, lexicon, and constraint *editors*. Furthermore, they allow for the checking of consistency of specifications and the testing of the validity of these specifications when confronted with language data on the basis of *tracers*, *steppers*, and *debuggers*. In addition, grammar *compilers* are supplied in order to achieve performance speed-ups.

Though grammar/parser engineering environments are vital for any natural language specification "in-the-large", tools for tracing the behavior of a distributed, concurrent computing system, such as the *ParseTalk* parser, are particularly needed, since their threads of control are often intertwined and hard to follow, even for experienced programmers. In particular, graphical tools have been proposed (Roman and Cox, 1989) and the *ParseTalk* environment has been designed on the basis of such suggestions focusing on lucid forms of control flow monitoring of concurrent distributed systems. The choice of a graphical toolbox is also perfectly in line with the underlying programming paradigm, since graphical interaction facilities have been provided in the object-oriented programming world from its inception (Goldberg, 1984).

Summarizing the remarks from above, major design decisions of the *ParseTalk* grammar engineering workbench were derived from software engineering as well as applicational considerations. The provision of that workbench does not constitute an original research contribution, but should be considered as part of a necessary technical infrastructure which will be introduced step by step in the following subsections. For illustrative purposes, a series of snapshots of the parse of the sentence "*Compaq entwickelt einen*

*Notebook mit einer 120-MByte-Harddisk*¹¹ is given. In Section 5, the information contained in the snapshots will be aggregated in terms of a partial event network representation, and thus it will be related to the theoretical framework worked out in the previous section.

4.1 Lexicon Tool: Browsing Lexical Specifications

Shifting our attention from the abstract hierarchy of word classes (Fig. 2) to concrete lexical entries, Fig. 8 introduces the *LexiconBrowser* which allows the addition and modification of lexical items. The entry for the surface form "*Compaq*" is selected. It is categorized as a NamePart (a subclass of Substantive, cf. Fig. 2) and refers to COMPAQ, a manufacturer, in the domain knowledge base. The morphological features of "*Compaq*" are ambiguous, since there is no explicit case marking. The highlighted portion of the grammar view contains the possible values of the case feature, viz. nom(inative), gen(itive), dat(ive), and acc(usative).

FIGURE 8. The LexiconBrowser

To guarantee consistent specifications across the different hierarchies that are used (cf. Section 3.1.1), several checks must be performed. After the edit cycle, the LexiconBrowser tests the unifiability of the features of a lexical entry with features concerning its valencies which are provided by its word class. The result of this unification is shown in the lower subview which cannot be edited but contains derived data representing the entry's morphosyntactic valency constraints. For example, in order for word actors to ful-

¹¹ A rough English translation of this reads as "*Compaq develops a notebook with a 120MByte hard disk*". Notice that from a *syntactic* perspective either the verb "*entwickelt*" or the noun "*Notebook*" may take a prepositional phrase with "*mit*" specifying, respectively, either an instrument or a part (cf. Section 6.3.1 for the treatment of this source of potential structural ambiguity).

fill a NamePart's valencies, specifier and attributive adjective, the prospective modifiers and their head must agree in case and number. Since the number of "Compaq" is restricted to singular, it must be singular for the modifiers; as its case is not restricted, it is also unconstrained for the prospective modifiers.

Consistency checking facilities will also be added concerning the availability of the concept identifier in the concept hierarchy and of the conceptual relations that are given within the valency specifications (cf. the conceptual restrictions mentioned in Section 3.1).

4.2 Parsing Tools: Monitoring the Parsing Process

FIGURE 9. The parsing environment

Fig. 9 introduces the basic utilities for the consideration of the parsing process as provided by the *ParseTalk* workbench:

- the *GraphicalActorBrowser* (left) displays the gradually emerging dependency parse tree for the sentence under consideration;
- the *ActorBrowser* (in front of the *GraphicalActorBrowser*) contains the actors currently instantiated, if any. It also allows the user to start an analysis (**Analyze** button, which displays a dialog window as shown), print the analysis result in different formats (**Bulletin**, **Dependency Tree** buttons), trace actors, and determine whether the domain knowledge base will be queried (**Connect to KB** button; this switch turned off will become relevant for the discussion of ambiguity in Section 6.3.1).
- the *MessageBrowser* (top right) displays the message traffic which occurs among the word actors;

- the *LOOMTranscript* (bottom right) contains information about query and update operations in the text knowledge base as resulting from the interactions of lexical items. In this view, one of the concepts needed later in the sample parse, DEVELOPS, is described in the LOOM-specific format (any DEVELOPMENT action requires a developing ACTOR, an OBJECT under development, and can be supplemented by an INSTRUMENT by which the OBJECT is developed).

In the following session transcript, word actors will be identified by a string **surface@pos<aspect>** with **surface** being the textual surface form, **pos** indicating the numerical text position, and **aspect** holding an aspect index. Aspects are created for the representation of ambiguities (cf. Section 6) and will not be considered any deeper in this section. Only the distinction between a lexical aspect (referencing the original lexical entry from the lexicon grammar, identified by aspect number 0) and syntactic aspects (referencing syntactically related and possibly more restricted copies of such a lexical entry, identified by aspect number 1) is relevant here. A message in the MessageBrowser will be identified by a string **type:sender=>receiver**, if it is a direct message, or **type:initiator==sender=>receiver**, if it is a distributed one. The type determines which parameters a message needs and which method is executed upon its arrival.

FIGURE 10. Actor instantiation and associated operations in the domain knowledge base

After instantiating the first word [Compaq] (first **startUp** message in the MessageBrowser), its corresponding concept is instantiated (COMPAQ-00001 on the LOOMTranscript, the numeric extension ensures uniqueness). The *WordActorInspector* shows some important parameters of [Compaq], viz. its word class (NamePart), corresponding instance in the domain knowledge base (COMPAQ-00001), morphosyntactic features (still ambig-

uous, highlighted here), acquaintances (a right neighbor, [entwickelt], and the scanner), and the prescribed relative order of modifiers (specifier and attribute preceding [Compaq] and the prepositional phrase following it). [Compaq] has already returned control to the scanner (**scanNext** message) which instantiates the second word from the sentence, viz. [entwickelt] (second **startUp** message). [entwickelt] instantiates a concept (DEVELOPS-00002 on the LOOMTranscript) and makes itself known to [Compaq] as new right neighbor (those messages are not shown here). The messages once stored for [entwickelt], representing information about the immediate left context, are now delivered (**searchHead** message highlighted in MessageBrowser) and processed. From a syntactic point of view, [Compaq] may instantiate two alternative dependencies of [entwickelt], namely subject and (direct) object. For both dependencies, a knowledge base request is generated, checking the domain restrictions for the conceptual relations that correspond to the grammatical dependencies. The highlighted request in the LOOMTranscript in Fig. 10 checks whether (the domain object corresponding to) [Compaq] may be an ACTOR¹² of (the domain object corresponding to) [entwickelt], and returns the name of a legal conceptual relation (ACTOR), while the second request checking for a direct OBJECT fails (its result is NIL).

FIGURE 11. Establishing a dependency relation

¹² Note that ACTOR and OBJECT here refer to relations between conceptual entities, not to dependencies, i.e., syntactic relations. There is only an approximate mapping between syntactic and conceptual relations (the domain restrictions in the valency definition, cf. Table 1 in Section 3.1), from which the concrete conceptual relation is determined (cf. the discussion of Fig. 12 below).

Since all restrictions on a possible modifier in the subject dependency are satisfied, [entwickelt] replies by sending a **headFound** message (indicating the dependency name, subj:). This message contains restrictions that the modifier must fulfill; in this case, morphosyntactic features are included. Upon arrival of the **headFound** message, a new aspect of [Compaq] is created that incorporates these restrictions (indicated by the <1> extension). The WordActorInspector operating on the new aspect reveals that the morphosyntactic features have been reduced to one reading, namely nominative (highlighted portion). The newly created aspect replies by sending a **headAccepted** message to [entwickelt] (highlighted in the MessageBrowser), thereby establishing the dependency relation. Note that an inverse relation from the modifier to the head is established, too (head relation in the WordActorInspector on [Compaq]). Upon arrival of the **headAccepted** message at [entwickelt], its acquaintances are expanded accordingly (not shown here). In addition, the conceptual relation corresponding to the dependency relation is established by issuing the request highlighted in the LOOMTranscript. In this way, syntactic and conceptual descriptions are generated incrementally and in parallel.

FIGURE 12. The use of the domain knowledge to restrict ambiguity

After reading the complete sentence, the configuration shown in Fig. 12 has been reached. The preposition [mit] is not yet integrated due to a mandatory valency that must be filled to make conceptual restrictions available. Upon establishment of a corresponding dependency between [mit] and [120-MByte-Harddisk] (**headAccepted** message just above the highlighted message in the MessageBrowser), [mit] starts the search for its head. Its **searchHead** message is directed to [120-MByte-Harddisk]¹³ and [Notebook] and further

¹³ [120-MByte-Harddisk] will not process this message, since it is already part of the phrase governed by [mit]. The dependency structure here simply serves as a data channel to direct the **searchHead** message to a potential head of [mit] possibly occurring to the right.

forwarded to its head [entwickelt] as shown in the MessageBrowser. From a syntactic point of view, both, [Notebook] and [entwickelt], may govern a prepositional phrase, and both generate a request to the domain knowledge base. The first one (highlighted in the LOOMTranscript) checks whether hard disk may be a property of a personal computer, i.e. 120MBHARDDISK-00004 is conceptually related to NOTEBOOK-00003 (the expression '(()) allows all conceptual roles of NOTEBOOK-00003 to be considered). This request returns the name of the sole legal conceptual relation, HASHARDDISK. The second request is restricted to the INSTRUMENT relation (trying to interpret 120MBHARDDISK-00004 as an instrument for the development action) and fails.

The integration of conceptual knowledge into the parsing process not only restricts syntactic ambiguity (preventing the attachment of [mit] to [entwickelt]; cf., however, Section 6 on that issue, again), but also leads to an analysis containing conceptual relations that cannot be inferred from syntax alone (identifying HASHARDDISK as the sole legal relation).

FIGURE 13. Syntactic and conceptual relations being established

The search of [mit] for its head, indicated by the four **searchHead** messages in the MessageBrowser, results in only one reply: a **headFound** message sent by [Notebook]. [mit] replies with a **headAccepted** message (highlighted in the MessageBrowser), thereby completing the negotiation. (One more message appears: An **updateFeatures** message is sent from [mit] to its modifier to propagate changes due to [mit] itself becoming a modifier.) The GraphicalActorBrowser now depicts the complete dependency tree (see also Table 1 in Section 3.1; both descriptions are intended to complement each other). The conceptual relations established are shown in Fig. 14.

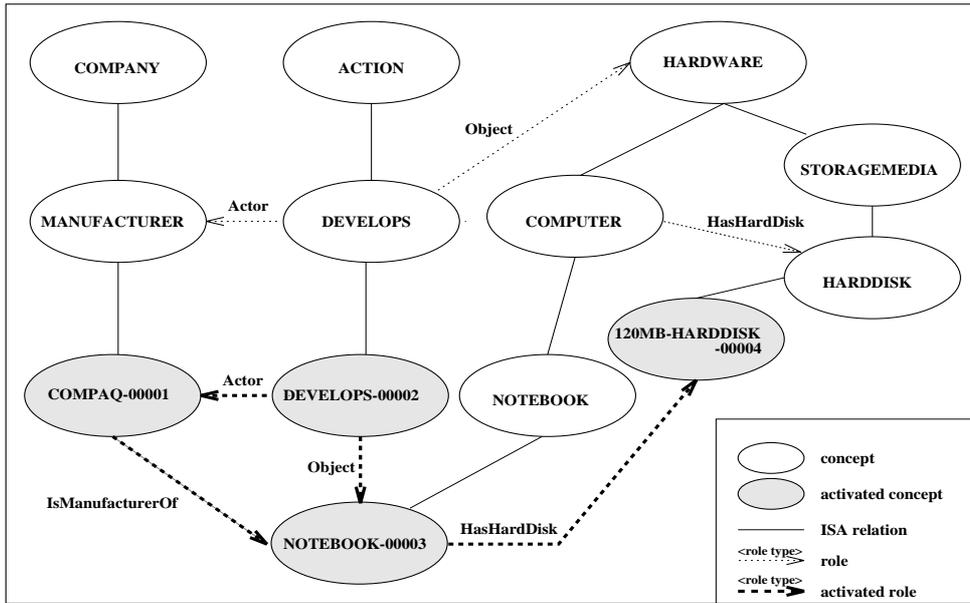


FIGURE 14. Conceptual relations established during the dependency parsing process

5 Event Network Specification of the Sample Parse

We may now abstract from the example in Section 4 and consider the parse in terms of event types and events (as introduced in Section 3.2.2). Fig. 15 depicts the event type network for the messages according to the grammatical specifications (the **copyStructure** and **duplicateStructure** messages are needed for ambiguity handling and will be referred to in more depth in Section 6.3). Note that this graph only contains possibilities -- the vertices can only be crossed if the associated conditions are met.

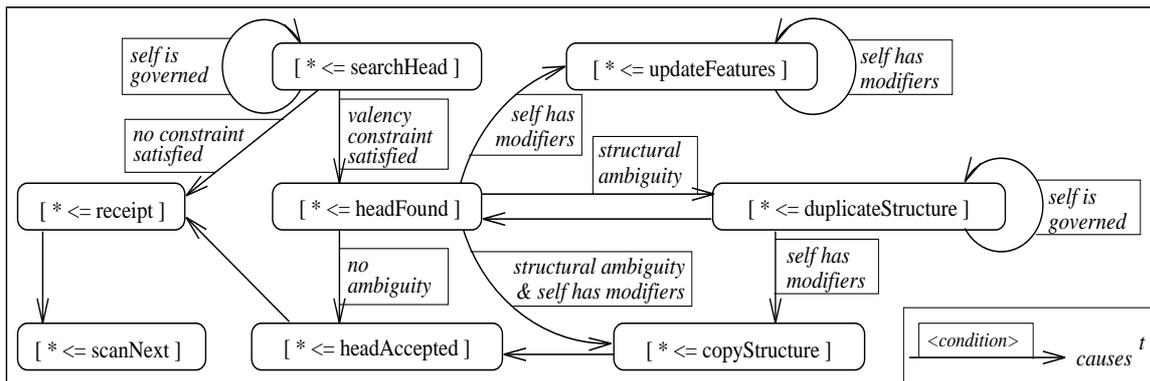


FIGURE 15. Event type network

Considering actual events, Fig. 16 gives the events caused by the satisfaction of the mandatory valency of [mit] as contained in the MessageBrowser in Fig. 13 (plus the events caused by receipt messages that are suppressed in the MessageBrowser protocol).

The two dotted lines indicate two additional possibilities for how the **scanNext** event could have been triggered. Of the three receipt events, the last one taking place triggers the **scanNext** event (note that all three events involve the same actor, [mit<1>], so that they must be ordered, even in a distributed system without global time).

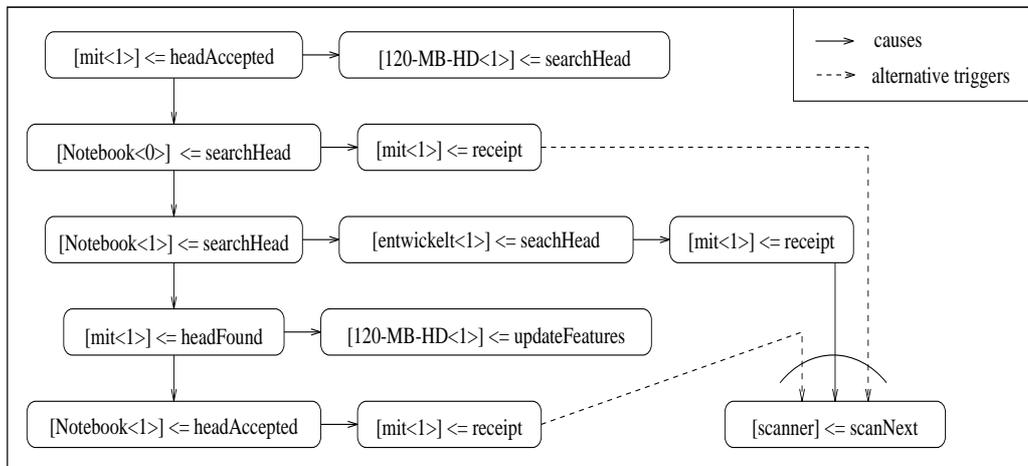


FIGURE 16. Fragment of an event network (attachment of the prepositional phrase)

These network representations visualize the grammar logic (event type networks) and the overall behavior of the parsing system (event networks). Thus, they provide an additional layer of abstraction for grammar specification and support a coherent view on how a distributed system actually performs.

6 Ambiguity Handling

Ambiguity is an inherent feature of natural languages. Thus, the ways ambiguous structures are dealt with by a natural language parsing model are a serious touchstone for the evaluation of its explanatory adequacy. Our approach to ambiguity reduction builds upon the simultaneous integration of several knowledge sources (grammatical, conceptual, and discourse knowledge) into the incremental parsing process, with alternative readings being processed in parallel (but cf. Section 6.4 for the consideration of a serial alternative to the basic parallel approach).

6.1 Packing Ambiguities

Usually, a packed representation of ambiguous structures is preferred in the parsing literature (Tamura, Bos, Murakami, Nishida, Yoshimi and Jelinek, 1991). This is entirely feasible when syntactic analysis is the only determining factor for the occurrence of partial structures. But if conceptual knowledge is taken into account, the conditions of the usage of a phrase are no longer fully determined by its syntactic structure; possible conceptual relations might equally have an influence on them. Additionally, the inclusion of an

ambiguous phrase in a larger syntactic context requires the modification of the conceptual counterparts. In a packed representation, there would necessarily have to be several conceptual counterparts, i.e., only the syntactic representation could be packed (and it might even be necessary to unpack it on-the-fly). Consequently, whenever conceptual analysis is tightly integrated into the parsing process (as opposed to semantic interpretations at a later stage, thereby producing numerous ambiguities during the syntactic analysis), structure sharing is impossible, since different syntactic attachments result in different conceptual analyses¹⁴, and no common structure is accessible that can be shared (cf. Akasaka (1991) for a similar argument). We expect the overhead of the multiplicative proliferation of parse structures to be compensated for by the ambiguity-reducing effects of integrating several knowledge sources in parallel.

6.2 Representation of Ambiguity: Shadows and Aspects

There are two kinds of ambiguities that are usually distinguished, viz. lexical and structural. Their representation is only slightly different in our approach.

A *lexical* ambiguity is detected during lexicon look-up, if there are several readings for a single surface form. One can further distinguish categorial (word class: relative pronoun vs. interrogative pronoun, e.g., "*who*"), conceptual (homonyms, e.g., "*river bank*" vs. "*finance bank*"), and morphosyntactic ambiguities (a proper noun not being marked for its case, cf. "*Compaq*" in Fig. 8 as an example). The first two kinds require separate actors to properly represent the ambiguity (since the categorial ambiguity may require a different behavior for each reading and the conceptual ambiguity may lead to different domain objects later to be referenced), while the third one can be expressed as a feature disjunction within one actor. For the case of several word actors representing ambiguous lexical items, one of them is arbitrarily chosen to represent the ambiguity, called the *original*. All other actors are acquaintances of the original referred to as *shadows*. This representation scheme allows the ambiguity to be hidden to the textual neighbors.

A *structural* ambiguity is detected during analysis, if there is more than one possible head for some word actor. Consequently, different states (i.e., different acquaintances) must be represented (including a different conceptual entity corresponding to each word actor). Each attachment alternative is therefore represented by a corresponding actor. For easy reference, there is also a "prototypical" actor representing the original lexical entry from the lexicon grammar. It is called the *lexical aspect*, while the actors that are related (and constrained in their interpretation) are called *syntactic aspects* (this has already been demonstrated in Section 4).

¹⁴ For example, a concept such as DEVELOP-1 which has COMPAQ as its actor and CAD/CAM TECHNOLOGY as its instrument role is clearly distinguished at the conceptual level from the entity DEVELOP-2 which has only COMPAQ as its actor, other roles being left uninstantiated.

Lexical and structural ambiguity result in a two-dimensional representation scheme for word forms (see left side of Fig. 17): There is an original entity plus a number of shadows (lexical ambiguity). Each of them (original entity and shadows) is represented at least by a lexical aspect (instantiated lexicon entry) plus a number of syntactic aspects representing attachment alternatives (modified lexicon entries). The right side of Fig. 17 gives an illustration of the representation of the (lexical and structural) ambiguity encountered in the discussion in Section 6.3.1 for the word "mit" (a verbal prefix being a separable part of some verbs in German).

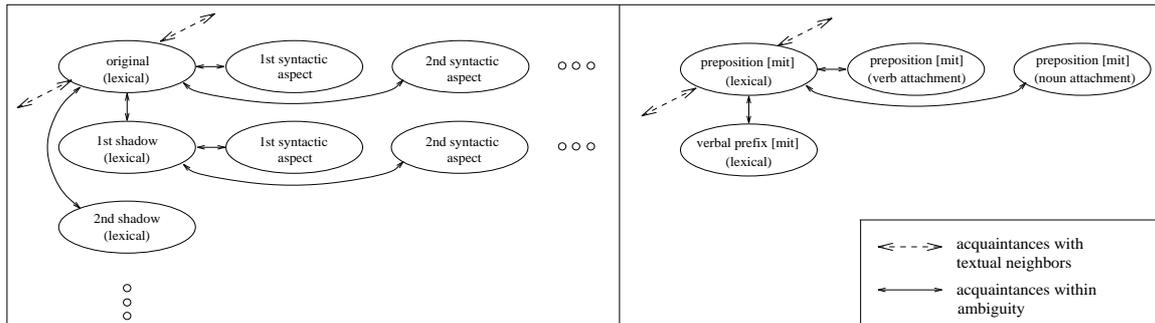


FIGURE 17. Two-dimensional representation scheme for lexical and structural ambiguity

Duplicating the actors also provides for concurrency without further costs. The administrative overhead is small, since only distribution from the original to shadows and aspects must be realized. This is done by the designated original, and thus not visible to other actors. In particular, this scheme requires no change of the underlying protocol apart from the extensions to be discussed in Section 6.3.

6.3 Protocol for the Instantiation of Structural Ambiguities

The principle behind the instantiation of structural ambiguities is quite simple. Since the standard negotiation process consists of a three-step protocol (cf. Section 3.4.3), it is possible to smoothly integrate the ambiguity handling mechanism without further changing the basic protocol. A structural ambiguity is caused by the existence of two possible heads for some word actor w . This is detected by w , if it receives more than one **headFound** message as replies to its own **searchHead** message. If this happens, one attachment has already been established (or is under construction and cannot be blocked). For the second attachment to be established, first, all actors that will represent this reading must be created (unpacked representation, cf. Section 6.1). w therefore copies itself (giving \overline{w}), requests that all modifiers copy themselves, and asks the prospective head to copy itself. The copies of the modifiers accept \overline{w} as their new head using the familiar **headAccepted** message, while the new copy of the head retries to govern \overline{w} by sending **headFound**. Thus, there are only two messages to consider (cf. Fig. 15 for the relation to other event types):

- **copyStructure** messages are sent to word actors that consequently duplicate themselves and their modifiers and are required to reply with a **headAccepted** message, if completed. The duplication of all (transitive) modifiers is achieved by recursively sending **copyStructure** messages to each of them.
- **duplicateStructure** messages are sent to word actors that consequently duplicate themselves, their modifiers, and their heads. A **headFound** message is expected from the copy of each receiving actor. The duplication of the modifiers is achieved by **copyStructure** messages, the duplication of all (transitive) heads results from recursively sending **duplicateStructure** messages to the head (until an ungoverned actor is addressed).

6.3.1 Example for Structural Ambiguity

To illustrate the protocol extension just described, we will examine the structural ambiguity that arises in the example from Section 4, if the domain knowledge base is not consulted. (The incorporation of the knowledge base into the parsing process is controlled by a button on the ActorBrowser, cf. Fig. 9.) Remember that the syntactic attachment of the prepositional phrase *"mit einer 120-MByte-Harddisk"* ["with a 120MByte hard disk"] to the verb *"entwickelt"* ["develops"] was rejected on the basis of conceptual criteria (a hard disk not being allowed as an instrument of computer development). The following example then will also illustrate the ambiguity-reducing effect of (simultaneously) combining several knowledge sources.

Reconsidering Fig. 12, the preposition [mit] has just sent its **searchHead** message to the left, and that message has been distributed to several word actors. [Notebook] as well as [entwickelt] may now accept the application of [mit], since there are no conceptual constraints that can be tested in order to preclude a dependency relation. Instead of the message traffic shown in the MessageBrowser in Fig. 12, the messages as depicted in Fig. 18 are sent. Above the highlighted message we find the negotiation of the first attachment alternative, interpreting the prepositional phrase as an instrument for [entwickelt] (upper tree in the GraphicalActorBrowser), while below that message the alternative already discussed in Section 4 appears (lower tree in the GraphicalActorBrowser), with a lot of extra message traffic owing to the duplication of tree structures required for the second structural reading.

Note that in this case, there are no structural criteria (such as completeness) that allow us to prefer one analysis over the other. Global ambiguities of this sort can only be resolved by considerations at the conceptual level of analysis.

FIGURE 18. Ambiguities due to restriction of the dependency parse to syntactic knowledge

6.4 Relation to Psycholinguistic Performance Models

The handling of ambiguities presented above reveals our preference for parallel evaluation of all alternatives. It has been argued instead that human language understanding proceeds in a deterministic fashion, choosing one alternative and backtracking if that path fails (Hemforth, Konieczny and Strube, 1993). This approach requires to rank all alternatives according to criteria referring to syntactic complexity, frequency of occurrence of syntactic patterns or lexical items, etc. The protocol outlined so far (Section 3.4) could easily be accommodated to this processing strategy: All **headFound** messages must be collected, and the corresponding attachments be ranked according to the above-mentioned criteria (possibly requiring the message to carry more information about the potential head). The "best" attachment is selected, and only one **headAccepted** message is sent. In case the analysis fails, the next-best attachment would be tried, until an analysis had been found or no alternatives were left. Additionally, the dependencies established during a failed path must be released.¹⁵

¹⁵ Note that all psycholinguistic studies we know of are referring to a constituency-based grammar model. Since our grammar is based on dependency relations, principles such as Minimal Attachment cannot be simply transferred, as the number of nodes is identical for all readings. Similar principles referring to the structural properties of dependency trees would be needed for any preferential ranking.

7 Comparison to Related Work

The issue of object-oriented natural language parsing and concurrency has long been considered from a purely *implementational* perspective. Message passing as an explicit control mechanism is inherent to various object-oriented implementations of standard rule-based parsers (cf. Yonezawa and Ohsawa (1988) for context-free and Phillips (1984) for Augmented Phrase Structure Grammar as well as Nishida and Doshita (1984) for Case Grammar). Actor-based implementations are provided by Uehara, Ochitani, Mikami and Toyoda (1985) for Lexical Functional Grammar as well as Abney and Cole (1986) for Government-Binding grammar. A parallel implementation of a rule-based, syntax-oriented dependency parser has been described by Akasaka (1991). The consideration of concurrency at the grammar *specification* level has recently been investigated by Milward (1992) who properly relates notions from categorial and dependency grammar with a state-logic approach, a formal alternative to the event-algebraic formalization underlying the *ParseTalk* model.

Almost all of these proposals lack serious accounts of the integration of syntactic knowledge with conceptual knowledge (cf. the end of Section 3.1 for similar considerations related to various models of dependency grammars). The development of conceptual parsers (Riesbeck and Schank, 1978; Gershman, 1982; Lebowitz, 1983; Dyer, 1989), however, was entirely dominated by conceptual expectations driving the parsing process, with word expert parsing (Small and Rieger, 1982) adding elaborated communication devices to autonomous lexical processes. Unfortunately, this approach did not specifically provide any mechanisms to integrate linguistic knowledge into such a lexical parser in a systematic and reliable way. The pseudo-parallelism inherent in these early proposals, word expert parsing in particular, has in the meantime been replaced by true parallelism, either using parallel logic programming environments (Devos, Adriaens and Willems, 1988), actor specifications (Hahn, 1989) or a connectionist methodology (Riesbeck and Martin, 1986) that has recently been implemented on massively parallel hardware (Kitano and Moldovan, 1992), while the lack of linguistic sophistication has more or less remained.

A word of caution should be expressed regarding the superficial similarity between object-oriented and connectionist models. Connectionist methodology (cf. a survey by Selman (1989) of some now classical connectionist natural language parsing systems) is restricted in two ways compared with object-oriented computing. First, its communication patterns are determined by the hard-wired topology of connectionist networks, whereas in object-oriented systems, and actor systems in particular, the topology is flexible and reconfigurable. Second, the type and amount of data that can be exchanged in a connectionist network is restricted to *marker* and *value passing*, together with severely limited com-

putation logic (and-ing, or-ing of Boolean bit markers, determining maximum/minimum values, etc.), while none of these restrictions apply to *message passing* models. These considerations equally extend to *spreading activation* models of natural language parsing (Charniak, 1986; Hirst, 1987) which are not as constrained as connectionist models but less expressive than general message passing models underlying the object-oriented and actor paradigm. As should be evident from the preceding exposition of the *ParseTalk* model, the complexity of the data exchanged and computations performed, in our case, require a full-fledged message-passing model.

We consider the *ParseTalk* model a proposal for a proper formal foundation of concurrent computing within the actor paradigm, where complex specification tasks are addressed. Many attempts so far were not able to balance the need for high-level language constructs (providing an appropriate level of abstraction) and formal precision. For instance, Clinger (1981) supplies a rigid denotational theory of actor computing that is nevertheless inapplicable to the specification of real-world problems. Another approach is Nierstrasz's object calculus (1992) which combines process calculi (Milner, Parrow and Walker, 1992) and object-oriented features within a model that supports concurrent functional agents with states. Yonezawa's specification language (1977) and Agha's Simple Actor Language (1986), on the other hand, provide a high-level specification environment that lacks sufficient formal rigor. The experience with different languages from the POOL family also shows the apparent dilemma -- the higher the degree of abstraction, the less tractable the semantics (America, de Bakker, Kok and Rutten, 1989; America and van der Linden, 1990). Nevertheless, the need for high-level actor languages dealing with sophisticated applications such as natural language understanding, robot control, etc. is evident, e.g., in the work of Shoham (1993) who proposes language constructs that incorporate notions from speech act theory within the paradigm of agent-oriented programming.

8 Conclusions

The *ParseTalk* model of natural language understanding aims at the integration of a lexically distributed, dependency-based grammar specification with a solid formal foundation for concurrent, object-oriented parsing. It conceives communication among and within different knowledge sources (grammar, domain and discourse knowledge) as the backbone for complex language understanding tasks. The main specification elements of the grammar model consist of categorial, morphosyntactic, conceptual, and word order constraints in terms of valency specifications attached to single lexical items. The associated concurrent computation model is based on the actor paradigm of object-oriented programming, with several extensions relating to special requirements of natural language processing. These cover mechanisms for complex message distribution, synchronization in terms

of request-reply protocols, the distinction of distribution and computation events, and the provision of "compiled" inheritance hierarchies.

The major strengths we attribute to our approach -- robust processing of real-world text in the presence of incomplete specification (requiring partial parsing) and the capability to process local and global text structures (leading to genuine text parsing performance) -- have already been shown to be a general feature of lexically distributed, message-passing-based approaches to natural language parsing (cf. Hahn, 1989; 1992). The current work extends these earlier studies by the linguistic sophistication of the current grammar model, mainly due to the incorporation of valency constraints and their realization in terms of dependency relations.

The *ParseTalk* model has been experimentally validated by a prototype system, a parser for German. The current full-form lexicon contains a hierarchy of 54 word class specifications and nearly 1000 lexical entries; a module for morphological analysis that will replace the use of full-form entries is under development (Bröker, Hanneforth and Hahn, 1994). The parser's coverage is currently restricted to the analysis of assertional sentences with focus on complex noun and prepositional phrases. The parser is implemented in Smalltalk with extensions that allow for coarse-grained parallelism through physical *distribution* in a workstation cluster (Xu, 1993) and *asynchronous* message passing. The *ParseTalk* system is coupled with the LOOM knowledge representation system (MacGregor and Bates, 1987) via a set of request and modify instructions. We currently supply a knowledge base with approximately 120 complex concepts covering the domain of information technology products.

Acknowledgments

The work reported in this paper is funded by grants from DFG (grants no. Ha 2097/1-1, Ha 2097/1-2) within a special research programme on cognitive linguistics. We like to thank our colleagues, P. Neuhaus, M. Klenner, K. Schnattinger, Th. Hanneforth, M. Kubierschky and M. Reiter, for valuable comments and implementational support.

This article combines and significantly extends two previous papers (Bröker, Hahn and Schacht, 1994; Schacht, Hahn and Bröker, 1994) that have already appeared in the *Proceedings of the 15th International Conference on Computational Linguistics (COLING'94)*.

References

- ABNEY, S. & COLE, J. (1986). A government-binding parser. In *Proc. of the North-Eastern Linguistic Society (NELS-16)*. pp.1-17. Ed. by S. Berman et al. Univ. of Massachusetts at Amherst.
- AGHA, G. (1986). *Actors: a Model of Concurrent Computation in Distributed Systems*. Cambridge, MA: MIT Press.
- AGHA, G. & HEWITT, C. (1987). Concurrent programming using actors. In A.YONEZAWA & M. TOKORO, Eds. *Object-Oriented Concurrent Programming*. pp.37-53. Cambridge, MA: MIT Press.
- AKASAKA, K. (1991). Parallel parsing system based on dependency grammar. In C.G. BROWN & G. KOCH, Eds. *Natural Language Understanding and Logic Programming, III. Proc. of the 3rd Intl. Workshop*. pp.147-157. Stockholm, Sweden, 23-25 Jan. 1991. Amsterdam: North-Holland.
- AMERICA, P.; BAKKER, J.; KOK, J. & RUTTEN, J. (1989). Denotational semantics of a parallel object-oriented language. *Information and Computation*, 83 (2), 152-205.
- AMERICA, P. & LINDEN, F. (1990). A parallel object-oriented language with inheritance and subtyping. *ACM SIGPLAN Notices*, 25 (10), 161-168 (= *Proceedings OOPSLA/ECOOP '90*).
- BACKOFEN, R.; TROST, H. & USZKOREIT, H. (1991). Linking typed feature formalisms and terminological knowledge representation languages in natural language front-ends. In W. BRAUER & D. HERNANDEZ, Eds. *Verteilte Künstliche Intelligenz und kooperatives Arbeiten. Proc. 4. Intl. GI-Kongreß "Wissensbasierte Systeme"*. pp.375-383. München, Okt. 1991. Berlin: Springer.
- BOGURAEV, B.; CARROLL, J.; BRISCOE, T. & GROVER, C. (1988). Software support for practical grammar development. In *COLING Budapest: Proc. of the 12th Intl. Conf. on Computational Linguistics [COLING '88]*. Vol.1. pp.54-58. Budapest, 22-27 August, 1988. Budapest: John von Neumann Society for Computing Sciences.
- BRÖKER, N.; HAHN, U. & SCHACHT, S. (1994). Concurrent lexicalized dependency parsing: the *Parse-Talk* model. In *COLING'94. Proc. of the 15th Intl. Conf. on Computational Linguistics*. Kyoto, Japan, Aug. 5-9, 1994.
- BRÖKER, N.; HANNEFORTH, Th. & HAHN, U. (1994). *Parsing mit objektorientierten Lexikogrammatiken*. CLIF - Arbeitsgruppe Linguistische Informatik/Computerlinguistik, Universität Freiburg (*CLIF-Report 11/94*)
- CHARNIAK, E. (1986). A neat theory of marker passing. In *AAAI-86: Proc. of the 5th National Conf. on Artificial Intelligence*. Vol.1: Science. pp.584-588. Philadelphia, PA, Aug. 11-15, 1986. Los Altos, CA: Morgan Kaufmann.
- CLARK, R. & GIBSON, E. (1988). A parallel model for adult sentence processing. In *Proc. of the 10th Annual Conf. of the Cognitive Science Society*. pp.270-276. Montreal, Quebec, Canada, 17-19 Aug. 1988. Hillsdale, NJ: L. Erlbaum.
- CLINGER, W. (1981). *Foundations of Actor Semantics*. Cambridge, MA: MIT, Dept. of Mathematics (Ph.D.Thesis).
- COSTANTINI, C.; FUM, D.; GUIDA, G.; MONTANARI, A. & TASSO, C. (1987). Text understanding with multiple knowledge sources: an experiment in distributed parsing. In *Proc. of the 3rd Conf. of the European Chapter of the Association for Computational Linguistics*. pp.75-79. Copenhagen, Denmark, 1-3 April, 1987.
- CREARY, L. & POLLARD, C. (1985). A computational semantics for natural language. In *Proc. of the 23rd Annual Meeting of the Association for Computational Linguistics*. pp.172-179. Chicago, Ill., USA, 8-12 July 1985.
- DAELEMANS, W.; De SMEDT, K. & GAZDAR, G. (1992). Inheritance in natural language processing. *Computational Linguistics*, 18 (2), 205-218.
- DEVOS, M.; ADRIAENS, G. & WILLEMS, Y. (1988). The parallel expert parser (PEP): a thoroughly revised descendent of the word expert parser (WEP). In *COLING Budapest: Proc. of the 12th Intl. Conf. on Computational Linguistics [COLING '88]*. Vol.1. pp.142-147. Budapest, 22-27 August, 1988. Budapest: John von Neumann Society for Computing Sciences.
- DYER, M.G. (1989). Knowledge interactions and integrated parsing for narrative comprehension. In D.L. WALTZ, Ed. *Semantic Structures. Advances in Natural Language Processing*. pp.1-55. Hillsdale, NJ: L. Erlbaum.
- EMELE, M. & ZAJAC, R. (1990). Typed unification grammars. In *COLING'90: Proc. of the 13th Intl. Conf. on Computational Linguistics*. Vol.3. pp.293-298. Helsinki, 1990.
- EVANS, R. & GAZDAR, G. (1990). *The DATR Papers, Vol. I*. University of Sussex, Brighton, 1990 (*Cognitive Science Research Paper CSRP 139*).

- FRASER, N. & HUDSON, R. (1992). Inheritance in word grammar. *Computational Linguistics*, 18 (2), 133-158.
- GENTHIAL, D.; COURTIN, J. & KOWARSKI, I. (1990). Contribution of a category hierarchy to the robustness of syntactic parsing. In *COLING'90: Proc. of the 13th Intl. Conf. on Computational Linguistics*. Vol.2. pp.139-144. Helsinki, 1990.
- GERSHMAN, A.V. (1982). A framework for conceptual analyzers. In W.G. LEHNERT & M.H. RINGLE, Eds. *Strategies for Natural Language Processing*. pp.177-201. Hillsdale, NJ: L. Erlbaum.
- GOLDBERG, A. (1984). *Smalltalk-80: The Interactive Programming Environment*. Reading, MA: Addison-Wesley.
- GRANGER, R.; EISELT, K. & HOLBROOK, J. (1986). Parsing with parallelism: a spreading-activation model of inference processing during text understanding. In J.L. KOLODNER & C.K. RIESBECK, Eds. *Experience, Memory, and Reasoning*. pp.227-246. Hillsdale, NJ: L. Erlbaum.
- GREIF, I. (1975). *Semantics of Communicating Parallel Processes*. Cambridge, MA: MIT, Dept. of Electrical Engineering and Computer Science (Ph.D.Thesis).
- HAHN, U. (1989). Making understanders out of parsers: semantically driven parsing as a key concept for realistic text understanding applications. *International Journal of Intelligent Systems*, 4 (3), 345-393.
- HAHN, U. (1992). On text coherence parsing. In *COLING '92: Proc. of the 15th Intl. Conf. on Computational Linguistics*. Vol. 1: Topical Papers. pp.25-31. Nantes, 23-28 Aug. 1992.
- HAHN, U. & ADRIAENS, G. (1994). Parallel natural language processing: background and overview. In G. ADRIAENS & U. HAHN, Eds. *Parallel Natural Language Processing*. pp.1-134. Norwood, NJ: Ablex.
- HAUGENEDER, H. & GEHRKE, M. (1986). A user friendly ATN programming environment (APE). In *COLING '86: Proc. of the 11th Intl. Conf. on Computational Linguistics*. pp.399-401. Bonn, August 25-29, 1986. Bonn: Institut für angewandte Kommunikations- und Sprachforschung (IKS).
- HELLWIG, P. (1988). Chart parsing according to the slot and filler principle. In *COLING Budapest: Proc. of the 12th Intl. Conf. on Computational Linguistics [COLING '88]*. Vol.2, pp.242-244. Budapest, 22-27 August, 1988. Budapest: John von Neumann Society for Computing Sciences.
- HEMFORTH, B.; KONIECZNY, L. & STRUBE, G. (1993). Incremental syntax processing and parsing strategies. In *Proc. of the 15th Annual Conf. of the Cognitive Science Society*. pp.539-545. Boulder, Col. 1993. Hillsdale, NJ: L. Erlbaum.
- HEPPLE, M. (1992). Chart parsing Lambek grammars: modal extensions and incrementality. In *COLING '92: Proc. of the 15th Intl. Conf. on Computational Linguistics*. Vol. 1: Topical Papers. pp.134-140. Nantes, 23-28 Aug. 1992.
- HEWITT, C. & ATKINSON, R. (1979). Specification and proof techniques for serializers. *IEEE Transactions on Software Engineering*, SE-5 (1), 10-23.
- HEWITT, C.; ATTARDI, G. & LIEBERMAN, H. (1979). Specifying and proving properties of guardians for distributed systems. In G. KAHN, Ed. *Semantics of Concurrent Computation. Proc. of the Intl. Symposium*. pp.316-336. Evian, France, July 2-4, 1979. Berlin: Springer.
- HEWITT, C. & BAKER, H. (1978). Actors and continuous functionals. In E.J. NEUHOLD, Ed. *Formal Description of Programming Concepts. Proc. of the IFIP Working Conf. on Formal Description of Programming Concepts*. pp.367-390. St. Andrews, N.B., Canada, Aug. 1-5, 1977. Amsterdam: North-Holland.
- HEWITT, C. & de JONG, P. (1983). Analyzing the roles of descriptions and actions in open systems. In *Proc. of the National Conf. on Artificial Intelligence [AAAI-83]*. pp.162-167. Washington, D.C., Aug. 22-26 1983.
- HIRST, G. (1987). *Semantic Interpretation and the Resolution of Ambiguity*. Cambridge: Cambridge University Press.
- HOARE, C.A.R. (1985). *Communicating Sequential Processes*. Englewood Cliffs, NJ: Prentice Hall.
- HUDSON, R. (1990). *English Word Grammar*. Oxford: Basil Blackwell.
- JÄPPINEN, H.; LASSILA, E. & LEHTOLA, A. (1988). Locally governed trees and dependency parsing. In *COLING Budapest: Proc. of the 12th Intl. Conf. on Computational Linguistics (COLING'88)*. Vol.1. pp.275-277. Budapest, 22-27 Aug. 1988. Budapest: John von Neumann Society for Computing Sciences.
- KASPER, R. (1989). Unification and classification: an experiment in information-based parsing. In *Proc. of the Intl. Workshop on Parsing Technologies*. pp.1-7. Pittsburgh, PA, August 1989.
- KITANO, H. & MODOVAN, D. (1992). Semantic Network Array Processor as a massively parallel computing platform for high performance and large-scale natural language processing. In *COLING '92: Proc.*

- of the 15th Intl. Conf. on Computational Linguistics. Vol. 2: Topical Papers. pp.813-819. Nantes, 23-28 Aug. 1992.
- KLENNER, M. & HAHN, U. (1994). Concept versioning: A methodology for tracking evolutionary concept drift in dynamic concept systems. In *ECAI 94: Proc. of the 11th European Conf. on Artificial Intelligence*. pp.473-477. Ed. by A. Cohn. Amsterdam, The Netherlands, Aug. 8-12, 1994. London: J. Wiley.
- LEBOWITZ, M. (1983). Memory-based parsing. *Artificial Intelligence*, 21 (4), 363-404.
- LESMO, L. & LOMBARDO, V. (1992). The assignment of grammatical relations in natural language processing. In *COLING '92: Proc. of the 15th Intl. Conf. on Computational Linguistics*. Vol.4: Project Notes without Demonstrations. pp.1090-1094. Nantes, 23-28 Aug. 1992.
- LIEBERMAN, H. (1987). Concurrent object-oriented programming in Act 1. In A.YONEZAWA & M. TOKORO, Eds. *Object-Oriented Concurrent Programming*. pp.9-36. Cambridge, MA: MIT Press.
- LOMBARDO, V. (1992). Incremental dependency parsing. In *Proc. of the 30th Annual Meeting of the Association for Computational Linguistics*. pp.291-293. Newark, Delaware, U.S.A., 28 June-2 July 1992.
- MacGREGOR, R. (1991). The evolving technology of classification-based knowledge representation systems. In J. SOWA, Ed. *Principles of Semantic Networks. Explorations in the Representation of Knowledge*. pp.385-400. San Mateo, CA: Morgan Kaufmann.
- MacGREGOR, R. & BATES, R. (1987). *The LOOM Knowledge Representation Language*. ISI, University of Southern California (*ISI Reprint Series, ISIRS-87-188*).
- MARSLEN-WILSON, W. & TYLER, L. (1980). The temporal structure of spoken language understanding: the perception of sentences and words in sentences. *Cognition*, 8, 1-71.
- MILNER, R.; PARROW, J. & WALKER, D. (1992). A calculus of mobile processes, III. *Information and Computation*, 100 (1), 1-77.
- MILWARD, D. (1992). Dynamics, dependency grammar and incremental interpretation. In *COLING '92: Proc. of the 15th Intl. Conf. on Computational Linguistics*. Vol.4: Project Notes without Demonstrations. pp.1095-1099. Nantes, 23-28 Aug. 1992.
- MURAKI, K.; ICHIYAMA, S. & FUKUMOCHI, Y. (1985). Augmented dependency grammar: a simple interface between the grammar rule and the knowledge. In *Proc. of the 2nd Conf. of the European Chapter of the Association for Computational Linguistics*. pp.198-204. Geneva, Switzerland, 27-29 March 1985.
- NAKAMURA, J.; TSUJII, J. & NAGAO, M. (1988). Grade: A software environment for machine translation. *Computers and Translation*, 3, 69-82.
- NIERSTRASZ, O. (1989). A survey of object-oriented concepts. In W. KIM & F.H. LOCHOVSKY, Eds. *Object-Oriented Concepts, Databases, and Applications*. pp.3-21. Reading, MA: Addison-Wesley.
- NIERSTRASZ, O. (1992). Towards an object calculus. In *ECOOP'91: Proc. of the European Workshop on Object-Based Concurrent Computing*. Geneva, Switzerland, July 15-16, 1991. Ed. by M. Tokoro, O. Nierstrasz, P. Wegner & A. Yonezawa. Berlin: Springer.
- NISHIDA, T. & DOSHITA, S. (1984). Combining functionality and object-orientedness for natural language processing. In *COLING'84: Proc of the 10th Intl. Conf. on Computational Linguistics & 22nd Annual Meeting of the Association for Computational Linguistics*. pp.218-221. Stanford, Cal., 2-6 July, 1984.
- PHILLIPS, B. (1984). An object-oriented parser. In B.G. BARA & G. GUIDA, Eds. *Computational Models of Natural Language Processing*. pp.297-321. Amsterdam: North-Holland.
- POLLARD, C. & SAG, I. (1987). *Information-Based Syntax and Semantics. Vol.1: Fundamentals*. Chicago, IL: Chicago University Press (*C.S.L.L. Lecture Notes, 13*).
- RIESBECK, C. & MARTIN, C. (1986). Direct memory access parsing. In J.L. KOLODNER & C.K. RIESBECK, Eds. *Experience, Memory and Reasoning*. pp.209-226. Hillsdale, NJ: L.Erlbaum.
- RIESBECK, C. & SCHANK, R. (1978). Comprehension by computer: expectation-based analysis of sentences in context. In W.J.M. LEVELT & G.B. FLORES d'ARCAIS, Eds. *Studies in the Perception of Language*. pp 247-293. Chichester: J. Wiley.
- ROMAN, G. & COX, K. (1989). A declarative approach to visualizing concurrent computations. *Computer*, 22 (10), 25-36.
- SCHABES, Y.; ABEILLE, A. & JOSHI, A. (1988). Parsing strategies with 'lexicalized' grammars: application to tree adjoining grammars. In *COLING Budapest: Proc. of the 12th Intl. Conf. on Computational Linguistics (COLING '88)*. Vol. 2. pp.578-583. Budapest, 22-27 Aug. 1988. Budapest: John von Neumann Society for Computing Sciences.
- SCHACHT, S.; HAHN, U. & BRÖKER, N. (1994). Concurrent lexicalized dependency parsing: a behavioral view on ParseTalk events. In *COLING'94: Proc. of the 15th Intl. Conf. on Computational Linguistics*. Kyoto, Japan, Aug. 5-9, 1994.

- SELMAN, B. (1989). Connectionist systems for natural language understanding. *Artificial Intelligence Review*, 3, 23-31.
- SHAPIRO, E. & TAKEUCHI, A. (1983). Object oriented programming in Concurrent Prolog. *New Generation Computing*, 1 (1), 25-48.
- SHAVIT, N & FRANCEZ, N. (1986). A new approach to detection of locally indicative stability. In L. KOTT, Ed. *Automata, Languages and Programming. Proc. of the 13th ICALP 1986*. pp.344-358. Berlin: Springer.
- SHIEBER, S. (1986). *An Introduction to Unification-based Approaches to Grammar*. Stanford, CA: Stanford Univ., Center for the Study of Language and Information (CSLI) (*C.S.L.I. Lecture Notes*, 4).
- SHOHAM, Y. (1993). Agent-oriented programming. *Artificial Intelligence*, 60 (1), 51-92.
- SMALL, S & RIEGER, C. (1982). Parsing and comprehending with word experts (a theory and its realization). In W.G. LEHNERT & M.H. RINGLE, Eds. *Strategies for Natural Language Processing*. pp.89-147 Hillsdale, NJ: L. Erlbaum.
- STAROSTA, S. & NOMURA, H. (1986). Lexicase parsing: a lexicon-driven approach to syntactic analysis. In *COLING '86: Proc. of the 11th Intl. Conf. on Computational Linguistics*. pp.127-132. Bonn, August 25-29, 1986. Bonn: Institut f. angewandte Kommunikations- und Sprachforschung (IKS).
- TAMURA, N.; BOS, M.; MURAKAMI, H.; NISHIDA, O.; YOSHIMI, T. & JELINEK, J. (1991). Lazy evaluation of preference on a packed shared forest without unpacking. In C.G. BROWN & G. KOCH, Eds. *Natural Language Understanding and Logic Programming, III. Proc. of the 3rd Intl. Workshop*. pp.13-26. Stockholm, Sweden, 23-25 Jan. 1991. Amsterdam: North-Holland.
- THIBADEAU, R.; JUST, M. & CARPENTER, P. (1982). A model of the time course and content of reading. *Cognitive Science*, 6, 157-203.
- UEHARA, K.; OCHITANI, R.; MIKAMI, O. & TOYODA, J. (1985). An integrated parser for text understanding: viewing parsing as passing messages among actors. In V. DAHL & P. SAINT-DIZIER, Eds. *Natural Language Understanding and Logic Programming. Proc. of the 1st Intl. Workshop*. pp.79-95. Rennes, France, 18-20 Sept., 1984. Amsterdam: North-Holland.
- WHITE, J. (1987). The research environment in the METAL project. In S. NIRENBURG, Ed. *Machine Translation. Theoretical and Methodological Issues*. pp.225-246. Cambridge: Cambridge University Press.
- XU, W. (1993). *Distributed, Shared and Persistent Objects. A Model for Distributed Object-Oriented Programming*. London University, Dept. of Computer Science (Ph.D.Diss.).
- YONEZAWA, A. (1977). *Specification and Verification Techniques for Parallel Programs Based on Message Passing Semantics*. Massachusetts Institute of Technology, Dept. of Electrical Engineering and Computer Science (Ph.D.Thesis).
- YONEZAWA, A. & HEWITT, C. (1979). Modelling distributed systems. In D.E. HAYES, D. MICHIE & L.I. MIKULICH, Eds. *Machine Intelligence. Vol. 9*. pp41-50. Chichester: Horwood.
- YONEZAWA, A. & OHSAWA, I. (1988). Object-oriented parallel parsing for context-free grammars. In *COLING Budapest: Proc. of the 12th Intl. Conf. on Computational Linguistics [COLING '88]*. Vol.2. pp.773-778. Budapest, 22-27 August, 1988. Budapest: John von Neumann Society for Computing Sciences.
- YU, Y. & SIMMONS, R. (1990). Truly parallel understanding of text. In *AAAI-90: Proc. of the 8th National Conf. on Artificial Intelligence*. Vol.2. pp.996-1001. Boston, Mass., July 29-August 2, 1990. Menlo Park: AAAI Press/MIT Press.