Resolution for Feature Logic

Bernd Fischer *
AG Softwaretechnologie
TU Braunschweig
fisch@ips.cs.tu-bs.de

June 15, 1993

Abstract: A common approach to combine the object-oriented and logic programming paradigms is to formulate a set of inference rules for an object logic. We show how resolution is expressed using the full feature logic. In contrast to similar approaches we do not only exchange the underlying term universe but discard the predicate calculus completely. We demonstrate that an untyped resolution violates a closed world assumption and introduce a type discipline to solve this problem. To integrate inheritance into this framework we introduce polymorphic types and rules.

1 Introduction

Object-oriented programming (OOP) and logic programming (LP) are two well-known programming paradigms. In the last few years, a number of approaches emerged which try to integrate them (see [Alex93] or [McCa92] for a survey.) These approaches vary greatly in scope and strategy: implementing LP in OOP and vice versa, constraint LP, or modelling state by modal logic for example.

A couple of approaches—including our own—follows a very general, common integration scheme. They formulate a set of inference rules for an object description language or *object logic.* Depending on the applied rules and logic this may result in a deductive object-oriented database (e. g. [KLW90]) or in a LP language in the Prolog tradition (e. g. Login [AKN86], LIFE [AKP91].)

This report shows how a resolution procedure for the full feature logic including disjunction and negation ([Smol92]) works. We then show some preliminary concepts for a LP language à la Prolog based on this resolution procedure.

The difference to other approaches which follow the same integration scheme results from the applied object logic. The terms of the feature logic constitute a Boolean algebra, the

*This work was supported by the DFG, grant Sn 11/1–2.

feature term algebra. Since clauses may be represented as implications in feature logic they are feature terms as well. Hence, object level (i. e. terms) and meta level (i. e. formulas) coincide. In contrast to other approaches which just exchange the underlying term universe we reformulate the resolution algorithm within the object logic and discard the predicate calculus completely.

A couple of advantages results from this method. First, an object logic as expressive as feature logic eases the description of complex objects. Second, feature resolution answers not only the questions explicitly posed by a query. It also yields information which is not directly related to the query but also relevant to the user. Third, the equivalence of object and meta level results in a more natural formulation of meta-information. Consequently, such generic information may easily be used during deduction.

In the sequel we introduce feature logic by means of examples and sketch its application to software engineering within the NORA project. For a formal treatment see [Fisc93]. Section 4 deals with a modified resolution algorithm. We then take care of the problem that this "naïve" modification does not preserve the closed world assumption and show finally how inheritance fits into this framework.

2 Feature Logic by Examples

Feature logic is an extension of predicate calculus. Its roots lie in computational linguistics where it is applied in unification-based grammar formalisms (see [Kay79]), knowledge representation (frame-based languages), and logic programming.

Features are (slot, value)-pairs which are considered to be partial object descriptions. A term

```
[os:msdos]
```

consisting of a single feature is thus interpreted as an object whose `os`-property has the value `msdos`. No other properties of this object are known. Using the operators of propositional calculus we can combine features and simple values to complex object descriptions. A *conjunction* of feature terms yields a list of properties which hold simultaneously. Thus

```
[os:msdos, arch:486]
```

describes an object whose `os`- and `arch`-properties are known. Similarly, a *disjunction* describes a list of alternative properties or values where at least one of the alternatives holds. So

```
{arch:386, arch:486}
```

describes all `386`-based and `486`-based systems. *Variables* denote sets of ground instances. As usual, they are capitalized.

There exist two special feature terms, *top*, represented as $\top$ or `[]`, and *bottom*, represented as $\bot$ or {}. Top denotes no specific or *void* information. It may be interpreted as an empty conjunction.The bottom element indicates *inconsistent* information. Inconsistency results from a property of features called *functionality*. It roughly says that each property of an object can have just one, unique value.

Let us for example assume that `msdos` and `bsd` are different atomic values. Then the term

```
[os:msdos, os:bsd]
```

is inconsistent. In other words, there exists no object which matches this partial description, no matter which other properties it might have.

Exploiting a close resemblance to Boolean algebra we also use $\sqcap$ for conjunction, $\sqcup$ for disjunction, and $\neg$ for negation. We then inductively define the set $F_{\Sigma(X)}$ of all feature terms over a fixed set F of feature symbols, a fixed set Ω of operator symbols, and a set X of variables as follows:

- $T_{\Omega(X)} \subset F_{\Sigma(X)}$, *(ordinary terms)*
- $p : t \in F_{\Sigma(X)} \ \forall p \in F, t \in F_{\Sigma(X)}$, *(features)*
- $t_1 \sqcap t_2, t_1 \sqcup t_2, \neg t \in F_{\Sigma(X)} \ \forall t, t_1, t_2 \in F_{\Sigma(X)}$ *(logical connectives)*
- $\bot, \top \in F_{\Sigma(X)}$ *(bottom, top)*

The feature terms do not constitute a flat universe but are ordered by *subsumption*. Subsumption (denoted by $\sqsubseteq$) combines the notions of instantiation and extension and orders terms by the amount of information they represent. A more general term subsumes a more specific. That is, t_1 subsumes t_2 either if t_2 is an instance of t_1 or if t_2 "has more" slots than t_1. We have for example:

```
[os:msdos, arch:486] ⊑ [os:msdos]  ⊑ [os:X]
```

Subsumption is defined by the following relation:

- $\sigma(t) \sqsubseteq t$ iff $t \in T_{\Omega(X)} - X$
- $t \sqsubseteq x$ iff $t \in F_{\Sigma(X)}, x \in X$
- $p : t_1 \sqsubseteq p : t_2$ iff $t_1 \sqsubseteq t_2$
- $t_1 \sqcap t_2 \sqsubseteq t_1 \sqsubseteq t_1 \sqcup t_2$
- $\neg t_2 \sqsubseteq \neg t_1$ iff $t_1 \sqsubseteq t_2$
- $\bot \sqsubseteq t \sqsubseteq \top$

Let $\sqsubseteq^*$ be the transitive, reflexive closure of $\sqsubseteq$. We say that t_1 *subsumes* t_2 iff $t_2 \sqsubseteq^* t_1$. For the following, we will abbreviate $\sqsubseteq^*$ by $\sqsubseteq$.

Subsumption may be used to explain the meaning of a feature term more formally. Let F_Σ^C denote the set of conjunctive ground terms, i. e. the set of all terms which contain neither disjunction nor negation nor variables. A feature term t then represents all terms in F_Σ^C which are subsumed by t, i. e.

$$[\![t]\!] = \{t' \in F_\Sigma^C \mid t' \sqsubseteq t\}$$

This interpretation reflects the idea that feature terms are *partial* object descriptions. Since we do not know anything about the aspects not mentioned explicitly we consider *all* possible extensions via subsumption. Thus the interpretation of a term

```
[os:msdos, arch:486]
```

which may denote a compiler

```
[os:msdos, arch:486, language:modula2]
```

as well as an actual computer

```
[os:msdos, arch:486, graphic:multicolor]
```

contains both extensions, compiler and computer.

The central operation on feature terms is the *unification* of two terms. Its purpose is to combine the partial descriptions given by the operands into a new, more specific description. E. g. the unification of two component descriptions

```
[arch:386, os:{bsd,sysv,aix}, lang:modula2]
```

and

```
[arch:X, os:{msdos,mvs,aix}, target_arch:X]
```

yields a description of the resulting system architecture:

```
[arch:386, os:aix, lang:modula2, target_arch:386].
```

Unification may also be explained in terms of subsumption. A unifier is a term which is subsumed by both operands. Since $\bot \sqsubseteq t$ for all t the unifier always exists but we will call two terms unifiable if their unifier is not $\bot$. A term t is called *most general unifier* of t_1 and t_2 or $\mathrm{mgu}(t_1, t_2)$ if it subsumes any other unifier of t_1 and t_2. Apart from variable renaming and simplifications, feature unification is unitary. I. e. for each pair of feature terms just one mgu exists.

We now have introduced all concepts which are necessary to formulate a resolution algorithm for feature logic.

3 An overview of NORA

The work described here originated in the context of software engineering and is part of the NORA[1] project. Its goal is the development of inference-based tools for software-engineering. We thus try to apply techniques from automated deduction to "real world" problems.

NORA consists[2] of a network of independent agents which are placed around a library of reusable components. Agents are specialists for tasks like

- interface control and checking [GS93],
- component retrieval, or
- configuration management.

[1] NORA is NO Real Acronym.

[2] See [SZ93] for a detailed description of NORA's architecture, [SGS91] for a discussion of the general ideas.

The current implementation focusses on Modula-2 but NORA is intended to be language independent. Language-specific information is just an additional parameter for agents.

Throughout this project we apply feature logic as a means for knowledge representation and inter-agent communication as the following scenario illustrates.

- The interface agent infers an interface description for component that is used but not declared, for example

```
[push:proc(X,int),
 top :{proc(X,int), proc(Y,Z)}].
```

- This description is extended by control information

```
[search-for:[interface:[...]]]
```

  and—via a central dispatcher—sent to the component retrieval.

- The retrieval agent checks the library against the interface description and finds a list of matching components. This list is sent to an interactive variant editor.

- Finally the user may select one implementation, based on the provided interface and additional component properties.

The coinciding representation of data and meta-data (i. e. control information) facilitates meta-reasoning. Thus, even control tasks like message dispatching can be done unification-based, taking into account an arbitrary network of agents. This in turn allows a dynamic reorganization and extension of the agent network. Hence, NORA can easily be adopted to new tasks.

4 Feature Resolution

We now show how resolution translates into feature logic. The algorithm may at first remain basically unchanged. Only some minor changes are necessary to cope with the new term structure. Of course, the unification procedure is exchanged.

As already mentioned, implications in feature logic, e. g.

```
[mood:happy] :- [likes:X, got:X].
```

serve as clauses. The meaning of such an implication is given by subsumption. Any term which is subsumed by the premise is also subsumed by the conclusion. We may thus infer from

```
[name:"Peter", likes:ice, got:ice]
```

that Peter is happy, i. e.:

```
[name:"Peter", mood:happy, likes:ice, got:ice].
```

A program PRG is considered to be the disjunction of all its clauses, i. e. $PRG = \bigsqcup C_i$. Obviously, PRG is also a feature term. A goal may consist of several subgoals which are

connected by conjunctions, i. e. $G = \bigsqcap G_i$. As usual we then try to show that the program and the negated goal are inconsistent, $PRG \sqcap \neg G \doteq \bot$. That is, we try to derive the bottom element which clearly is the equivalent of the empty clause.

Each deduction step results in a state $(\neg[G_1, \ldots, G_n], R)$, a feature term tuple. Its first component is a conjunction of the remaining intermediate goals G_i. The second component is the (intermediate) result R. Since feature unifiers are terms and cannot be replaced by substitutions, we have to resort to terms to collect the result. The initial state of the algorithm is $(\neg G, \top)$.

A resolution step takes as input a state

$$(\neg[G_1, \ldots, G_n], R)$$

and an appropriate clause

$$P \text{ :- } Q_1 \sqcap \ldots \sqcap Q_m.$$

We assume that P is unifiable with some subgoal, say $P \sqcap G_1 = T \neq \bot$. As usual, G_1 is replaced by the body of the clause. The result and each remaining goal are then unified with T.[3] Thus, the next state is given by

$$(\neg[T \sqcap Q_1, \ldots, T \sqcap Q_m, T \sqcap G_2, \ldots, T \sqcap G_n], T \sqcap R).$$

As usual, we have to backtrack if there is no appropriate clause. Additionally, a proof attempt fails prematurely if the state is of the form $(\neg[G_1, \ldots, G_k], \bot)$. That is, we still have goals to resolve but the intermediate result is already inconsistent. This situation arises when the program contains inconsistent information.

The proof is completed if we succeed in deriving the bottom element, that is to reach a state $(\neg\bot, R)$ where $\bot$ is the single remaining goal. Then R is a consequence of the program. Since unification is monotone with respect to subsumption, R is also subsumed by the initial goal, $R \sqsubseteq G$.

Figure 1 shows an example program in feature logic. The numbers in parentheses are only for reference. It consists solely of facts,[4] rules, and (type) declarations like

```
grade = {a,b,c,d,e}.
```

We temporarily consider them as mere constant declarations. Thus, each occurrence of the identifier `grade` will be replaced by the term `{a,b,c,d,e}`. Their real purpose will be shown in section 5. Any "real program" would of course contain some of the usual non-logical constructs as the *cut* or expression evaluation.[5] We will however concentrate on the logical constructs.

We may now, for example, ask for the names of happy people and submit the query

```
?  [name:X, mood:happy].
```

to an interpreter. Applying the above abbreviations we have

G_1 = `[name:X]`

[3] Of course, the unifications need not to take place immediately but may be deferred until we try to resolve that goal or complete the deduction.

[4] We adopt the usual convention that facts are considered to be implications with an empty premise.

[5] Our experimental implementation FRoM supports cut and arithmetics.

```
(1)  grade     = {a,b,c,d,e}.
(2)  goodgrade = {a,b}.
(3)  badgrade  = {c,d,e}.
(4)  goodthing = goodgrade.

(5)  [name: "Peter", got: c, likes: [name: "Mary"]].
(6)  [name: "Paul",  got: e].
(7)  [name: "Mary",  got: a].

(8)  [mood: happy] :- [likes: X, got: X].
(9)  [mood: happy] :- [likes: [got: goodthing]].

(10) [likes: goodthing].
```

Figure 1: A simple program

and

G_2 = `[mood:happy]`.

To prove G_1, the interpreter scans the program top-down. Fact (5) is suitable and we get

$P = R =$ `[name:"Peter", got:c, likes:[name:"Mary"]]`

and thus Peter's existence as first intermediate result. It remains to show that he is happy. Rule (8) is appropriate but fails in the next step since the interpreter cannot verify that Peter got what he likes. Thus, we have to backtrack and check out rule (9). We replace $G_1 \sqcap P$ by the term

```
[name:"Peter", mood:happy, got:c,
 likes:[name:"Mary", got:goodthing]].
```

The single subterm not yet verified is

```
[name:"Mary", got:goodthing]
```

which yields our next subgoal. We have to demonstrate that Mary got a `goodthing`. Since `goodthing` via `goodgrade` is an abbreviation for {`a, b`}, fact (7) suffices. Our first result is thus

```
[name:"Peter", mood:happy, got:c, likes:[name:"Mary", got:a]].
```

The interpreter then backtracks and tries to prove G_1 applying fact (6). But neither rule (8) nor rule (9) apply for Paul. So the interpreter backtracks again and uses the next fact (7) to show G_1. By rules (8) and (10) it then infers that Mary is happy, too, because she got a `goodthing` which everyone likes:

```
[name:"Mary", mood:happy, got:a, likes:a].
```

The query is then answered since the example program contains no other appropriate clause.

5 On the closed world assumption

In this section we are going to settle the question when a clause is suitable to resolve a goal. According to Prolog we might expect that unifiability of the goal and the clause head suffices. However, due to the accumulating behavior of feature unification, this yields some at least unexpected results.

Suppose a program consisting of the single fact

```
[name:"John", age:25].
```

The query

```
?  [name:X].
```

then yields

```
[name:"John", age:25]
```

as expected whereas we might be astonished about the result of

```
?  [name:X, mood:happy].
```

Instead of returning with no answer the interpreter readily infers that John is happy:

```
[name:"John", age:25, mood:happy].
```

This obviously violates a kind of *closed world assumption*. The program itself gives no particular reason to assume that John is happy. In general, it gives no hint about John's mood at all. This behavior results from a property of feature unification. The second query introduces a "prejudice" or assumption into the deduction process, namely the `mood:happy`-feature. It is no consequence[6] of the program but neither it is a contradiction. Since feature unification accumulates information until a contradiction appears, the resolution fails to detect that `mood:happy` is an assumption and just echoes it back.

In a third query,

```
?  [name:"Joe", mood:happy].
```

however, the contradiction occurs early enough and thus inhibits the assumption from propagating to the answer. We thus get the expected result.

We inhibit the introduction of assumptions by imposing a type discipline on the applicable rules. Suppose a (negated) goal G and a clause head P. P is applicable if its type is a subtype of G's type. This is a suitable generalization of the usual claim that goal and clause head are unifiable and complementary literals. In our setting, we demand that P and G are complementary terms, i. e.

$$\begin{aligned} & P \sqcap \neg G = \bot \\ \Rightarrow \quad & [\![P]\!] \cap [\![\neg G]\!] = \emptyset \\ \Rightarrow \quad & [\![P]\!] \cap (F_\Sigma^C - [\![G]\!]) = \emptyset \\ \Rightarrow \quad & [\![P]\!] \subseteq [\![G]\!] \\ \Rightarrow \quad & P \sqsubseteq G \end{aligned}$$

[6] Consequence in an informal meaning.

and thus get the above type restriction.

To incorporate such a type discipline into programs we need the type definitions. Types are represented by type terms. They closely resemble feature terms but are built on top of another set of operator symbols called *sorts* or *basic types*. Replacing each sort by the set of its instances yields all instances of a type.

Thus, the typed equivalent of the above example is

```
person = [name:string, age:int].
person[name:"John", age:25].
```

Since subtyping is subsumption of type terms we can easily see that

$$\texttt{[name:string, age:int]} \not\sqsubseteq \texttt{[name:string, mood:moodtype]}$$

and thus both, the second and third query produce no result.

6 Inheritance

We are now going to deal with inheritance which—besides objects—is usually considered to be one of the key concepts of object-oriented programming. Despite its importance, inheritance has no definition generally agreed upon. In our opinion, inheritance results from two simpler concepts, *classification* and *delegation*.

Classification orders objects into a usually hierarchical structure. All equivalent (w. r. t. classification) objects form a *class*. It thus covers the taxonomic aspect of inheritance. Delegation denotes the mechanism to pass a method invocation to another object or class. It thus allows different classes to share method implementations and facilitates reuse.

The combination of both concepts into a new one, inheritance, ensures a consistent and clean system structure. Delegation may only take place along the lines given by classification: only more specific objects may delegate method calls to objects of more general classes. Because classification takes the implemented methods into account success of delegation may be checked statically.

In our setting a class comprises a type definition, all known facts of that type (*instances*) and all rules whose head is of that type (*methods*), e. g.

```
class     person = [name:string, age:int]
instances [name:"John", age:25].
```

Since feature logic originates in knowledge representation, the taxonomic aspect of inheritance naturally translates into our framework. Single inheritance is just subtyping as for example in

```
class student = person ⊓ [subject:string].
```

This definition extends `person` by a `subject`-feature and thus defines `student` as a subtype of `person`. This mechanism naturally extends to multiple inheritance if applied to different named types:

```
class foreigner = person ⊓ [nation:string]
class foreign_student = foreigner ⊓ student
```

The delegation behavior of inheritance is much harder to model. We have seen in the last section that a rule (i. e. method) is applicable only if it is subsumed by the goal to be resolved. Thus, delegation would proceed in the wrong direction, that is upwards. A more special method is inherited by a more general object.

Consequently, we need another concept to model inheritance right. This concept must respect the closed world assumption and thus be compatible with our type discipline.

Our solution is inspired by the polymorphic types and functions of functional languages. Similarly, we use *type schemes*. A type scheme is a type definition which contains a free variable or *type parameter*. A scheme like

```
class moody(X ⊑ person) = X ⊓ [mood:{happy,angry,sad}]
```

may be considered to be a polymorphic class definition. The type parameter is instantiated along the lines of the taxonomic hierarchy. This scheme is thus expanded to e. g.

```
class moody_student = [name:string, age:int, subject:string,
                       mood:{happy,angry,sad}]
```

Since the instantiation of the type variables may be restricted to a part of the hierarchy (in this example to `person` and its subtypes) we actually have some kind of bounded polymorphism.

A *method scheme* is a method for a type scheme. That is, its clause head has the scheme type as for example in

```
moody(X ⊑ person) ⊓ [mood:happy] :- X ⊓ [likes:[got:goodthing]].
```

Method schemes are expanded in the same way as type schemes. Thus, after expansion this rule reads as

```
moody_student ⊓ [mood:happy] :- student ⊓ [likes:[got:goodthing]].
```

This ensures that an appropriate rule exists for each type which specializes the type parameter, i. e. `person`. Hence, the method is correctly inherited.

7 Conclusions

We have shown how resolution translates into feature logic, an expressive object logic. Since its terms constitute a Boolean algebra, predicate calculus becomes redundant and is discarded. Slots then take the role of the predicate symbols. We have demonstrated that a straightforward translation of the resolution algorithm violates a closed world assumption and we have introduced a type discipline which solves this problem. Unifiability and type constraints together guide the selection of applicable rules. Finally, we have integrated inheritance into that framework.

Our efforts yield a language which integrates logic programming and object-oriented programming. To summarize (and to contribute a new slogan), it can be characterized by

$$\text{inheritance} = \text{subtyping} + \text{bounded polymorphism}.$$

A prototypical implementation called FRoM is currently in progress [Brau93].

Our first intended application of the feature resolution is an enhanced `make` facility for maintaining large software systems. It will be able to cope with incomplete and ambiguous information. This will ease the construction of multi-version systems.

We currently investigate the relations between our type structure and that of Haskell [HJW92, NS91]. In Haskell, polymorphism may be combined with overloading. The functions introduce equivalences on types called type classes. Type classes and type templates over disjunctive types seem to be closely related.

Up to now, we have just transplanted the syntax-driven resolution algorithm into a new logical structure. Another crucial point is thus an appropriate definition of entailment. This also includes further investigations about the relation between inheritance and the existential queries. The work of Smolka and Treinen [ST92] seems to be a good starting point.

Acknowledgements

M. Kievernagel served as guinea pig for most of the ideas and did a tough job in proofreading. M. Brauer implemented a language prototype. F.-J. Grosch, C. Lindig, G. Snelting, H. Uphoff and A. Zeller contributed additional valuable discussions.

References

[AKN86] H. Aït-Kaci and R. Nasr. Login: A Logic Programming Language with Built-in Inheritance. *Journal of Logic Programming*, **1986**(3):186–215, 1986.

[AKP91] H. Aït-Kaci and A. Podelski. Towards a Meaning of LIFE. In *Proc. 3rd. International Symposium on Programming Language Implementation and Logic Programming*, pp. 255–274, 1991.

[Alex93] V. Alexiev. A (Not Very Much) Annoted Bibliography on Integrating Object-Oriented and Logic Programming. Technical report, University of Alberta, 1993. Available per ftp from `menaik.cs.ualberta.ca:pub/oolog`.

[Brau93] M. Brauer. Ein Interpreter für Feature-Logik. Master's thesis, Technische Universität Braunschweig, (in preparation) 1993.

[Fisc93] B. Fischer. A New Feature-Unification Algorithm. Technical Report 93-01, Inst. f. Programmiersprachen und Informationssyteme, Technische Universität Braunschweig, to appear 1993.

[GS93] F.-J. Grosch and G. Snelting. Polymorphic Components for Monomorphic Languages. In Ruben Prieto-Diaz and William B. Frakes, (eds.), *Advances in Software Reuse*, pp. 56–65, Lucca, Italy, March 24–26 1993. IEEE Computer Society Press.

[HJW92] P. Hudak, S. P. Jones, P. Wadler. Report on the programming language Haskell: A non-strict, purely functional language. *ACM SIGPLAN Notices* **27**(5), May 1992.

[Kay79] M. Kay. Functional Grammar. In *Proceedings of the Fifth Annual Meeting of the Berkeley Linguistics Society*, 1979.

[KLW90] M. Kifer, G. Lausen, and J. Wu. Logical Foundations of Object-Oriented and Frame-Based Languages. Technical Report 90/14 (2nd. revision), Department of Computer Science, SUNY at Stony Brook, Stony Brook, NY 11794, August 1990.

[McCa92] F. G. McCabe. *Logic and Objects.* International Series in Computer Science. Prentice-Hall International, Englewood Cliffs, 1992.

[NS91] T. Nipkow and G. Snelting. Type classes and overloading resolution via order-sorted unification. *Proc. Functional Programming Lanugages and Computer Architecture*, *LNCS* **523**, pp. 1–14, 1991.

[SGS91] G. Snelting, F.-J. Grosch, and U. Schröder. Inference-Based Support for Programming in the Large. In A. van Lamsweerde and A. Fugetta, (eds.), *Proc. 3rd. European Software Engineering Conference*, *Lecture Notes in Computer Science* **550**, pp. 396–408, Milan, October 1991. Springer Verlag.

[Smol92] G. Smolka. Feature-Constraint Logics for Unification Grammars. *Journal of Logic Programming*, **1992**(12):51–87, 1992.

[ST92] G. Smolka and R. Treinen. Records for Logic Programming. In *Proc. 1992 Joint Intl. Conf. Symp. Logic Programming*, to appear 1992.

[SZ93] G. Snelting and A. Zeller. Inferenzbasierte Werkzeuge in NORA. Technical Report 93-03, Inst. f. Programmiersprachen und Informationssyteme, Technische Universität Braunschweig, April 1993.