FORK: A System for Object- and Rule-Oriented Programming

C. Beckstein, G. Görz and M. Tielemann
University of Erlangen-Nürnberg

Zusammenfassung. Das Ziel des FORK-Projekts besteht in der Implementierung eines primär objekt-orientierten Wissensreprsentationssystems und seiner Anwendung auf den Entwurf und die Fehlerdiagnose technischer Systeme. Während der Kern des Repräsentationssystems FORK vollständig objekt-orientiert ist, soll das System als Ganzes eine Vielfalt von Programmierstilen unterstützen. Im folgenden beschreiben wir eine Erweiterung zur regel-orientierten Programmierung, die die sprachliche Ausdruckskraft von FORK über diejenige von LOOPS hinaushebt. Als eine Anwendung der regel-orientierten Komponente wurde eine Constraint-Sprache (relations-orienterter Programmierstil) implementiert, die ein wichtiges Werkzeug im Rahmen unseres Ansatzes zum Entwurf und zur Fehlerdiagnose technischer Systeme darstellt.

Der nächste Schritt im FORK-Projekt schließt die Entwicklung eines allgemeinen logischen Rahmensystems ein, wozu eine logische Rekonstruktion objekt-zentrierter Repräsentationen, Zugriff auf komplexe Beschreibungen mittels Unifikation und Deduktionen über strukturierte Objekte gehören. Das Problem der Nicht-Monotonität wird durch einen DeKleers ATMS ähnlichen Modul behandelt werden. Weiterhin erhoffen wir Fortschritte durch einen neuen allgemeinen Ansatz zur Darstellung zeitlicher Verhältnisse bei der Modellierung technischer Systeme, was u.E. eines der wichtigsten Themen in diesem Bereich ist.

Abstract. We describe progress made within the FORK project, whose goals are the implementation of a primarily object-oriented knowledge representation system and its application to the design and fault diagnosis of technical systems. Whereas the kernel of the FORK representation system is completely object-oriented, the system as a whole is supposed to integrate a variety of different programming styles. In the following, an extension for rule-oriented programming is described, which raises the descriptive power of the FORK system beyond that of LOOPS. As an application of the rule-oriented component, a constraint language has been implemented which plays an important rule in our approach to the design and fault diagnosis of technical systems.

The next steps in the FORK project will include the development of a general logical framework, comprising a logical reconstruction of object-centered representations, retrieval of complex descriptions by unification, and deductions on structured objects. The problem of non-monotonicity will be dealt with on the meta level by a module similar to DeKleer's ATMS. Further progress shall be achieved by concentrating on a general treatment of the problem of time in modelling technical systems which is to our opinion one of the most important issues.

1 FORK: A Flavor-Based Object-Oriented Knowledge Representation System

1.1 Knowledge Representation and Object-Oriented Programming

Knowledge-based systems differ from other software systems in that they contain an explicit encoding of the respective domain knowledge. There are at least two kinds of prerequisites for knowledge representation: methodological (i.e. epistemological and logical) and technical (in a sense linguistic) ones. In the following we concentrate on the second aspect in presenting a knowledge representation framework which is easy to use, extensible, and in particular suitable to represent the kinds of knowledge required for designing and diagnosing technical systems.

Under the technical aspect, representing knowledge in a computational system is nothing else than a special kind of programming. For programming seen as a linguistic activity, questions of expressiveness of the programming language and of adequacy and suitability of its means with respect to the field(s) of application are of immediate importance. Within the last twenty years, a broad variety of knowledge representation languages has been proposed ranging from more or less direct derivatives of first-order logic to schemata based on cognitive psychology or applied computer science like associative networks, production systems, or procedural languages. One of the most advanced approaches along this line were Minsky's [14] frames. Minsky tried to find a synthesis between declarative and procedural systems with an emphasis on object-centered representations. With a few exceptions, most of these systems were very experimental in character and could not achieve wide usage.

In the meantime in the field of programming languages a new programming "paradigm" emerged: the *object-oriented* style. The ideas of object-oriented programming were realized in various ways, either as programming languages in their own right, like Smalltalk-80, or as extensions to already existing languages.

Which are the salient features of the object-oriented programming style? Object-oriented systems offer an integrated view of the concepts of *abstract data types* and *generic functions* (see [17]). The underlying processing model is characterized as a system of communicating *objects* which *pass messages* among each other. Each object has a set of acquaintances, which are denotations of objects it "knows of", i.e. it can send messages to. Messages themselves are also objects; each message contains (a reference to) the addressee, (a denotation of) an operation, and — optionally — argument objects. Each object has a *protocol* which is a set of *methods;* these are the procedures or operations contained in messages it can process. The internal state of an object — a set of attributes — as well as its internal processing cannot be inspected from the outside. An object may be in an active or passive state, and its activities consist in sending, receiving, and processing of messages. Processing a message can cause the sending of other messages. Furthermore, most object- oriented systems offer means for structuring the object world through a *class system* by accumulating objects with the same protocol in one class. There are distinct class objects which generate instance objects as the result of processing a particular message (`"instantiate"`). Classes can usually be ordered in generalization hierarchies, along which inheritance relations with respect to their attributes — declarative or procedural, the latter being methods — hold.

So object-oriented systems offer a lot of features which are desirable for the purpose of knowledge representation. Even some features like the distinction between classes and instances and the inheritance mechanism are introduced in a methodologically cleaner and clearer way than in most knowledge representation systems. For these reasons, the FORK system [1] has been based on an object-oriented framework.

Primarily for its flexibility and extensibility LISP was chosen as the host language. Its essential advantage is that it does not enforce a particular programming style, but instead allows various programming styles based on different processing models, e.g. the imperative, functional and object-oriented styles. Because there is no fundamental distinction between program and data in LISP, the integration of object-oriented programming is facilitated by employing the dualism between "passive" data and "active" procedures.

One of the best known object-oriented extensions of LISP is the so called *Flavor* system [19], a portable reconstruction of which was the starting point for the FORK system. For the Flavor system, there are two kinds of objects: *Classes*, which are also called *"Flavors"*, and their *instances*. Classes represent generic objects; they describe instances by specifying sets of declarative (variables) and procedural (methods) attributes and inheritance relations:

- *local variables:* the so called *instance variables,*
- *class variables:* variables, which are owned by the class, but can be referred to by its instances; this is an extension to the original Flavor proposal,
- *component classes*, which themselves provide variables and methods through inheritance mechanisms,
- *methods:* procedures to process *messages:* the *protocol.*

In FORK classes as well as instances are able to process messages: Whereas classes can process messages immediately, instances pass messages to their class they have been instantiated from.

With classes, a *generalization hierarchy* can be built, which is also called the *flavor graph.* By referring to other classes ("superflavors") within the definition of a new class, attributes of the superclasses are inherited. The root of this — in general directed — graph is denoted by the most general class `VANILLA` which owns those methods which are valid for all classes. For the construction of the protocol of a new class, inherited methods can be combined in predefined ways (for "primary" methods and "before/after-demons"). So, starting from `VANILLA` successively more specialized object classes can be defined with the possibility of *multiple inheritance* of attributes.

The Portable Flavor system does not require — in contrast to the original — any modifications to the LISP interpreter. The only requirement to the underlying LISP system is full functionality of the closure mechanism. Presently versions for InterLISP and CommonLISP exist.

1.2 FORK as an Extension of Flavors for Knowledge Representation

Although there is a close resemblance between an object-oriented system like Flavors and object-centered knowledge representation languages like Frames, the latter ones provide a repertoire of specialized constructs which are particularly useful for knowledge representation. Therefore the Portable Flavor system has been extended with the following features to constitute the FORK kernel:

- A *type concept* has been introduced such that restricting ranges of values and the definition of modalities for attributes like optional or obligatory are possible.
- FORK automatically supervises *structural relations* (integrity constraints) over whole objects as defined by the user.
- *Set-valued variables* are supported such that besides type constraints for their elements also cardinality is checked automatically. Special methods to handle sets are provided.
- In addition to instance variables there are also *class variables*, as mentioned above.
- FORK has a versatile interface to the *inheritance mechanism* which allows different ways to control and influence inheritance. With respect to the descriptors of variables it is possible to differentiate inheritance according to specific roles, to refuse inherited attributes, or to transform inherited attributes (e.g. to rename variables).
- FORK allows the expression of *multiple perspectives*.

The following example, defining a class MOVING-OBJECT and a method SPEED for it, may illustrate some of these features:

```
(DEFFLAVOR MOVING-OBJECT
           (X-POS Y-POS X-SPEED Y-SPEED MASS)
           :gettable-instance-variables
           (:settable-instance-variables
              X-POS Y-POS X-SPEED Y-SPEED)
           (:initable-instance-variables MASS)

(DEFMETHOD (MOVING-OBJECT SPEED) ()
           (sqrt (+ (square X-SPEED)
                    (square Y-SPEED))))
```

Now we define a CAR as a particular MOVING-OBJECT:

```
(DEFFLAVOR CAR
           (FRAME-NUMBER
           (NO-WHEELS :MOD OBL
                      :DEFAULT 4)
```

```
          (AGGREGATES :MOD OBL
                      :DEFAULT 'OK)
          (FUEL :MOD OPT
                :RESTR (ONE-OF EMPTY FULL)
                :DEFAULT 'FULL)
          (OIL :DEFAULT 'MAX)
          (BATTERY :RESTR (AN ACCU)
                   :DEFAULT BAT-45AH)
          (TYRE-PRESSURE :MODE OBL
                         :DEFAULT 'HIGH)
          (DRIVER :RESTR (A PERSON)
                  :DEFAULT DUMMY))
          (MOVING-OBJECT)
          (:settable-instance-variables NO-WHEELS AGGREGATES
             FUEL OIL BATTERY TYRE-PRESSURE DRIVER)
          (:gettable-instance-variables FRAME-NUMBER)
          (:initable-instance-variables FRAME-NUMBER)
          (:required-flavors ACCU PERSON)
          (:documentation The flavors ACCU and PERSON should
             also be defined at time of first instantiation))
```

The instance variable FUEL is restricted by ONE-OF to the values EMPTY and FULL, which is the default. The variable BATTERY does only accept instances of the class ACCU as values. Defining CAR as a subclass of MOVING-OBJECT enables access to MOVING-OB JECT's instance variables (with the aception of MASS) and methods, i.e., in addition to the instance variables defined in CAR, each instance of CAR has also the inherited instance variables X-POS, Y-POS, X-SPEED, Y-SPEED. Furthermore a class variable SELLING-COM PANY is defined, which participates with the instance variable OWNER (an instance of PERSON) in an integrity constraint.
Volkswagens are a brand of CARs:

```
(DEFFLAVOR VW     ; instance variables:
          ((CAR-TYPE :RESTR (ONE-OF PASSAT POLO GOLF)
           (OWNER :MOD OBL
                  :DEFAULT (SEND-SELF 'GET 'SELLING-COMPANY))
           (ID :MOD OBL))
         (CAR)    ; superclass
         (:class-variables (SELLING-COMPANY
                            :DEFAULT VOLKSWAGEN-AG
                            :RESTR (A COMPANY)))
         (:initable-instance-variables CAR-TYPE)
         (:settable-instance-variables OWNER ID)
         (:gettable-instance-variables CAR-TYPE OWNER ID))
```

Now we create an instance of the VW class:

```
(SETQ MY-CAR (SEND VW 'CREATE-INSTANCE
'(CAR-TYPE  POLO
  OWNER     BECKSTEIN
  DRIVER    TIELEMANN
  ID        ERH-E-536
  BATTERY   BAT-45AH)))
```

Of course, the implementation of FORK includes tools for debugging, editing, and handling objects (e.g. an interface to the file package).

2 Rule-Oriented Programming in FORK

2.1 Rulesets, Rules, and Rule Interpreters

The first extension to the kernel of the FORK system introduces a new kind of methods: Besides methods wr¡itten in the traditional procedural or functional style, they can also be defined in the form of rule systems. To be precise, a method may be given as a forward-chaining, i.e. data-driven production system. In its present form, the rule-oriented component of FORK [18] is more powerful then that of LOOPS [4], because it offers additional means for *conflict resolution* and for processing *vague information.*

Each method written in the rule-oriented style is a *ruleset*, which consists of rules of the form

```
{rule-name} IF premise THEN action {! meta-info}
```

Each ruleset has its own rule-interpreter associated with it which executes the rules under a forward chaining control regime, which in turn may use different control structures varying among rulesets for the selection and processing of rules.

Because rulesets are ordinary methods of objects, they can be inherited (also as "before-" and "after-methods"), so that large rulesets can be structured according to an object hierarchy. As ordinary methods, rulesets are activated by message passing. Since the control structure (meta-knowledge) for processing rules is associated to a ruleset, a clear separation between control and domain knowledge can be achieved. Each rule interpreter has the following structure:

1. Preselection of a subset of rules within the ruleset through comparison of rule specific scores with a threshold value which is local with respect to the ruleset. This mechanism allows prescreening of the ruleset at runtime.

2. The preselected rules are checked for applicability and then one applicable rule is selected. For this phase three control strategies are possible:

    - FIRST or ALL: In this case the ruleset is assumed to be ordered and the check for applicability is performed in the given order. With FIRST the first applicable rule is selected, with ALL each applicable rule.
    - PRIO: The value calculated by the evaluation of the premise at runtime is used to rank the rules, and the one with the highest value is selected.

    The FIRST and ALL modes correspond to DO1 and DOALL in LOOPS. With each control strategy, rulesets can also be processed iteratively by means of a WHILE option.

The following method CONTROL for CAR is defined as a ruleset of three rules:

```
(DEFRULESET (CAR CONTROL) ()      ; no local variables
            ((C-FUEL              ; rule set
                IF (EMPTY FUEL-LEVEL)
                    THEN (SIGNAL "no fuel"))
             (C-OIL
                IF (EQ OIL-LEVEL 'MIN)
                    THEN (SIGNAL "no oil")
                         (SEND ME 'STOP-ENGINE)
                         (STOP! 'EMERGENCY-OFF))
             (C-BATTERY
                IF (LESSP (SEND BATTERY 'VOLTAGE) 6)
                    THEN (SIGNAL "low voltage")
                      !  (:USAGE ONE-SHOT-BANG)) )
             (:DOC ruleset for controlling ...)
             (:CHECK-MODE ALL)
             (:WHILE (ON IGNITION)))
```

As soon as this method is activated, the rule C-BATTERY is checked for applicability only once (ONE-SHOT-BANG), and never checked again during eventually subsequent iterations, i.e. as long as the ignition is turned on.

In contrast to the FIRST and ALL modes, for PRIO no static order criterion holds. Instead the ordering of rules is determined dynamically by priorities which are determined by the rule premises. To achieve that, first of all a transition from qualities to quantities has to be made by which a measure of evidence (Certainty Factors CF) is determined from the values of premises:

```
(not NIL) & (not (NUMBERP X))  -->  CF := MAXCF
                      X = NIL  -->  CF := 0
                  (NUMBERP X)  -->  CF := X
```

with CF $\in$ [0,MAXCF] $\subset \mathbb{N}_0$.

Boolean combinations of premises are calculated by special methods ($AND, $OR, $NOT).

The priority calculated in this way is used to schedule the respective rule in a multilevel agenda, from which afterwards the first rule from the level with highest priority is selected.

3. After selection, the action part of the respective rule is executed by the rule interpreter. The working memory (context) within the execution is performed is the FORK object to which the ruleset method belongs.

It should be noted that using the PRIO control strategy MYCIN-like rules can be expressed imediately with FORK's rule formalism.

2.2 Implementation of the Rule-Oriented Component

For the implementation of rulesets and rules, the kernel language of FORK has been used as a metalanguage, i.e., the whole rule component of FORK itself is expressed by means provided by the FORK kernel. Rulesets as well as rules are represented as FORK objects with appropriate methods. Rulesets are instances of a class `RULESET` which includes parameters and local variables as static components (instance variables), methods with defaults for conflict resolution and control- and meta-information (the rule interpreter) as dynamic attributes and rules as component flavors. In the same way rules are instances of a class `RULE` with appropriate methods. The programming environment of FORK has been augmented for rule-oriented programming with methods for debugging, editing, and handling rulesets, and rules which interface to particular attributes of the `RULESET` and `RULE` classes.

2.3 A Rule-Based Implementation of a Constraint Language

To demonstrate the versatility of rule-oriented programming in FORK, it has been used to implement a constraint language (after [16]. Constraint languages are a tool for relation-oriented programming and, therefore, play an important role in our approach to the diagnosis problem (see section 3). What a constraint language has at least to offer are means to represent objects which express *elementary relations* ("constraints") and *connections* between these objects. So, e.g. a constraint representing an `ADDER` has two input connectors `A` and `B` and an output connector `S` such that the following relations hold: `S = A + B`, and, at the same time `B = S - A` and `A = S - B`. Connecting objects of this kind leads to the construction of *constraint networks* in which computations are performed by *propagation* of values. In constraint languages, the term propagation covers local computations satisfying local dependences (as expressed by the equations for `ADDER`) as well as spreading values through connectors within a network.[1] In general, there is no preferred direction for spreading values.

So a minimal constraint language has at least to provide constructs to express

- *definitions* of constraints,
- *construction* of constraint networks,
- *integration* of new constraints into an already existing network,
- *communication* with the network interface (input and output).

With FORK, there is a straightforward approach to implement that by expressing constraints and connectors as objects and networks as aggregates of those. The propagation of values, which is locally restricted, is specified by rules belonging to constraint objects. For the case of electronic circuits, there are prototypic building blocks, like `ADDER`, of which instances are generated, which in turn are then combined to descriptions of network by attaching their connectors with each other. As a benefit of the object-oriented representation, such a description of a complex network is nothing else than *one* object on a higher descriptive level.

The following piece of code defines an `ADDER`:

[1]Spreadsheet programs are a special kind of constraint systems.

```
(DEFCONSTRAINT ADDER (A1 A2 SUM)
               NIL                        ; local variables
               (A1-A2                     ; rule set
                  IF (ALL-KNOWN? A1 A2)
                     THEN  (SET-VALUE! SUM
                              (PLUS (GET-VALUE! A1)
                                    (GET-VALUE! A2))
                              &ME))
               (A1-SUM
                  IF (ALL-KNOWN? A1 SUM)
                     THEN (SET-VALUE! A2
                             (DIFFERENCE (GET-VALUE! SUM)
                                         (GET-VALUE! A1))
                             &ME))
               (A2-SUM
                  IF (ALL-KNOWN? A2 SUM)
                     THEN (SET-VALUE! A2
                             (DIFFERENCE (GET-VALUE! SUM)
                                         (GET-VALUE! A2))
                             &ME)))
```

A constraint network then can easily be constructed in the following way:

```
(DE CONSTRAINT-NET ()
  ...
  (SETQ A (MAKE-CONNECTOR))     ; generate new instances of
  (SETQ B (MAKE-CONNECTOR))     ; CONNECTOR
  ...
  (LET ((X (MAKE-CONNECTOR))
        ...)                    ; now generate new instances of
       ...                      ; constraint ADDER
       (MAKE-CONSTRAINT 'ADDER 'A1 A 'A2 B 'SUM X)
       ... ))
```

3 Future Work

In parallel to the implementation of the FORK system, a first study in the field of diagnosis has been conducted, aiming at a clarification of the basic problems and representational needs (cf. [3,13,10]). After considering more traditional rule-based approaches to the diagnosis problem, we concentrated on an approach known as "based on *structure* and *behavior*" (cf. [6,9]). Starting with an algorithm to diagnose multiple failures in electronic circuits, considerable extensions had to be made for the more complicated case of electromechanical systems. The kernel of the resulting diagnosis system, DIAGTECH, has been implemented in the object-oriented style. In fact, DIAGTECH is a hybrid system, because it also supports the rule-based style of diagnosis, for which our logic-based "expert system shell" DUCKITO [12] is used as a subsystem.

In DIAGTECH, the treatment of conflicts, or inconsistency, as well as particular diagnostic heuristics, were embedded into one algorithm. To guarantee the usefulness of the chosen approach in the long run, it has to be generalized in a way that the recording and processing of inconsistency is performed by a separate general module. Indeed, as DeKleer [7] points out, most problem solvers search; if otherwise, a direct algorithm would solve the task. Then, two important problems are to be solved, namely, how the spaces of alternatives can be searched efficiently, and how the problem solver should be organized in general. DeKleer's solution is convincing: On the one hand it is a consequent continuation of the "Truth Maintenance Systems" (TMS) line, which forces a clean division within a problem solver between a module solely concerned with rules of the domain and another module concerned with recording the current state of the search. While the first module draws inferences, the second, the TMS, records inferences ("justifications"). So, the TMS serves three roles:

1. It serves as a "cache memory" for all inferences in that inferences, once made, need not be repeated. Inconsistencies, once detected, are avoided in the future.

2. It allows the problem solver to draw non-monotonic inferences. If non-monotonic justifications are present, the TMS has to use a constraint satisfation procedure to determine what data are assumed to be valid.

3. It assures that the data base is contradiction-free. The procedure of *dependency-directed backtracking* identifies and adds justifications to remove inconsistencies.

DeKleer's ATMS (Assumption Based TMS) [7] is a very efficient TMS module. In particular from our experience with DIAGTECH, but also from general considerations about a logical extension to our object-oriented representation system, we decided to realize such a module within our framework as the next step of the FORK project. We believe this will be a mandatory prerequisite to address the perspective of logic programming, namely (predominantly descriptive) representation and processing of relations (constraints) and implications in the object domain, and, in particular, the representation and treatment of time in a more general way. The gap between a logical reconstruction of object-centered representations (cf. [15,11]) and logic-based representation systems with their inferential properties is still to be closed. Retrieval of complex descriptions requires a powerful extension to the well-known unifcation algorithms. The direction of this research is also influenced by our previous experience with DUCKITO, which contains a truth maintenance module and an explanation component on the basis of data dependences.

As far as the problem of representing time and temporal relations is concerned, we are currently investigating approaches which reach beyond the one used within DIAGTECH. The latter one has been constructed in the spirit of Doyle's [8] JACK system. The main problem we encountered with it is not a lack of expressive power, but a fundamental discrepancy between constraint-based representations on the one hand and the directionality introduced by temporal expressions on the other. Constraint systems assume simultaneous propagation of values in all possible directions of a constraint network, i.e. non-directionality of components and multiple values. A temporal order does not allow to consider all "possible worlds" at once, but enforces an order on propagation. We hope to find a synthesis of the advantages of both by means of a modal logic approach.

References

[1] Beckstein, C.: *Integration objekt-orientierter Sprachmittel zur Wissensrepräsentation in LISP.* Diplomarbeit, IMMD und RRZE, Universität Erlangen-Nürnberg. RRZE-IAB-219, 1985.

[2] Beckstein C., Görz, G., Tielemann, M.: *FORK: Ein System zur objekt- und regelorientierten Programmierung.* In: Rollinger, C., Horn, W. (Hg.): GWAI-86. 10th German Workshop on Artificial Intelligence und 2. Österreichische Artificial Intelligence Tagung. Berlin: Springer IFB 124, 312-317, 1986.

[3] Beckstein C., Görz, G., Hernàndez, D., Tielemann, M.: *An Integration of Object-Oriented Knowledge Representation and Rule-Oriented Programming as a Basis for Design and Diagnosis of Technical Systems.* To appear in: Annals of Operations Research. Also: RRZE-IAB-248, Univ. Erlangen-Nürnberg, 1986.

[4] Bobrow, D., Stefik, M.: *LOOPS Manual & Rule Oriented Programming in LOOPS.* Xerox PARC Report, Palo Alto, 1983.

[5] Bobrow, D. (Ed.): *Qualitative Reasoning about Physical Systems.* Amsterdam: North-Holland, 1984.

[6] Davis, R.: *Diagnostic Reasoning Based on Structure and Behavior.* 1984, In: Bobrow 347–410, 1984.

[7] DeKleer, J.: *(a) An Assumption-Based TMS. (b) Extending the AMTS. (c) Problem Solving with the ATMS.* AI Journal 28, 127-163-197-224, 1986.

[8] Doyle, R.: *Hypothesizing and Refining Causal Models.* MIT-AI-Memo 811, Dec. 1984.

[9] Genesereth, M.: *The Use of Design Descriptions in Automated Diagnosis.* In: Bobrow (1984), 411–436, 1984.

[10] Görz, G., Hernández, D.: *Knowledge-Based Fault Diagnosis of Technical Systems.* In: This Conference Proceedings.

[11] Hayes, P.: *The Logic of Frames.* In: Metzing, D. (Ed.): *Frame Conceptions and Text Understanding.* Berlin: DeGruyter, 1980.

[12] Hernández, D.: *Modulare Softwarebausteine zur Wissensrepräsentation.* Studienarbeit, IMMD IV und RRZE, Universität Erlangen-Nürnberg, 1984.

[13] Hernández, D.: *Wissensbasierte Diagnose technischer Systeme.* Diplomarbeit, IMMD und RRZE, Universität Erlangen-Nürnberg. RRZE Mitteilungsblatt Nr. 44, 1986.

[14] Minsky, M.: *A Framework for Representing Knowledge.* In: Winston, P.H. (Ed.): *The Psychology of Computer Vision.* New York: McGraw Hill, 211–277, 1975.

[15] Nilsson, N.: *Principles of Artificial Intelligence.* Berlin: Springer, 1982.

[16] Steele, G.L.: *The Definition and Implementation of a Computer Language Based on CONSTRAINTS.* MIT-AI-TR-595, Aug. 1980.

[17] Stoyan, H., Görz, G.: *Was ist objekt-orientierte Programmierung?* In: Stoyan, H., Wedekind, H. (Hg.): *Objekt-Orientierte Software- und Hardware-Architekturen.* Stuttgart: Teubner, 1984.

[18] Tielemann, M.: *Eine regelorientierte Erweiterung des Repräsentationssystems FORK.* Diplomarbeit, IMMD und RRZE, Universität Erlangen-Nürnberg. RRZE-IAB-236, 1986.

[19] Weinreb, D., Moon, D.: *Flavors: Message-Passing in the LISP Machine.* MIT-AI-Memo, 1981.

C. Beckstein
University of Erlangen-Nürnberg, IMMD VI
Martensstr. 3, D-8520 Erlangen
Network: `unido!fauern!faui70!beck`

G. Görz
University of Erlangen-Nürnberg, RRZE
Martensstr. 1, D-8520 Erlangen
Network: `Goerz@SUMEX.ARPA, GOERZ@DERRZE1.BITNET`

M. Tielemann
University of Erlangen-Nürnberg, IMMD VI
Martensstr. 3, D-8520 Erlangen