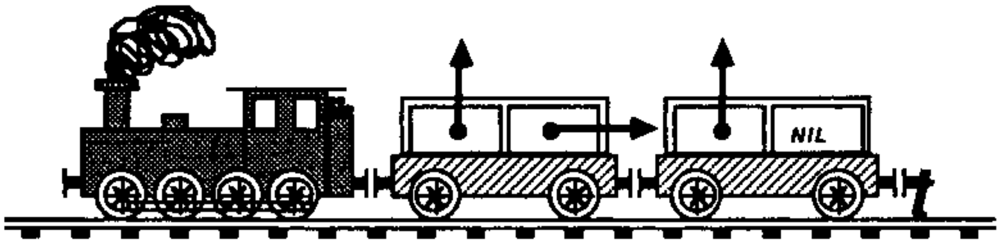
A Universal



1

Programming Language

Hans Werner Guesgen[2]
International Computer Science Institute
1947 Center Street, Suite 600
Berkeley, California 94704-1105

Abstract

In this paper, we describe a programming language based on constraints. Unlike former approaches, its interpreter propagates sets of possible values (rather than single values) through a network of constraints. During the propagation process, the value sets are filtered to obtain consistent subsets, or new values are computed for the undetermined variables from the set of given variables.

Constraint relations can be described in several ways. Finite relations can be represented straightforwardly by enumerating their extensions. Infinite relations can be described intensionally by patterns. More complex relations can be represented by constraint networks and hierarchies of constraint networks. Several control strategics are provided which compute local consistency as well as globally consistent solutions.

It is sketched how to realize a compiler for the language, which optimizes constraint descriptions at definition time. As a result, combinatorical explosion can be reduced, depending on the number of variables used in a constraint description.

[1]CONS-train-t

[2]The author previously worked at the Gesellschaft fuer Mathematik und Datenverarbeitung, Schloss Birlinghoven, 5205 Sankt Augustin, West Germany. There, his work was supported in part by the German Federal Ministry for Research and Technology within the joint project TEX-B, contract number ITW-8506-D.

1 Introduction

The problem: on the one hand, there are—let us call them—systems of class alpha and on the other, there are systems of class beta, but what we really need is a kind of alphabeta system. What we mean by class alpha and class beta can be characterized by two historical representatives.

An alpha system: in 1980, G.L. Steele published the description of a computer programming language based on constraints [Steele 1980J. It is—as far as we know—the first approach to a constrained-based programming language which is independent of a special application. The interpreter of the language is based on a computational model that is comparable to data driven machines: undetermined variables are computed from a set of given variables.

A beta system: D.L. Waltz [1972] suggested a filtering technique that enables us to solve certain constraint problems more efficiently. The idea is to avoid combinatorial explosion by propagating value sets without computing the Cartesian product between these sets.

Our goal is a kind of alphabeta system, i.e. to combine the aspects of a domain-independent constraint language with an efficient problem solving strategy. The result is the constraint satisfaction system CONSAT which is introduced in this paper:

CONSAT can be applied to arbitrary domains, i.e. the constraint variables may be associated with any kind of values.

- Single values or sets of possible values can be propagated through a constraint network. Thus, CONSAT can be used to compute values for the undetermined variables from the set of given variables, to test consistency of values, or to filter the sets of possible values, i.e. to remove inconsistent values from these sets.

Several ways to represent relations are provided. Finite relations can be represented straightforwardly by enumerating the extension of the relation. Infinite relations can be described intensionally by patterns. More complex relations can be represented by constraint networks and hierarchies of constraint networks.

Besides a standard control strategy based on local propagation, which computes local consistency, two further strategies are provided for obtaining global consistency: one of them is a combination of local propagation and backtracking, the other uses (recursive) indices in order to maintain relationships between values.

CONSAT has been implemented in Common Lisp and has been integrated into the hybrid knowledge representation system BABYLON [Guesgen et al. 1987].

2 Underlying Concepts

First, we will give an informal definition of constraints (cf. [Montanari 1974]): a constraint consists of a set of variables and a relation on these variables.

In order to maintain transparency, a constraint should describe only partial aspects of the task to be programmed. The whole task can then be described by a network of constraints: a constraint network consists of a set of constraints with shared variables.

A technique often used to satisfy networks of constraints is local propagation: the results computed by a constraint, i.e. the restrictions to the coverings of its variables, are forwarded to its neighboring constraints in the network. The result of local propagation is local consistency, i.e. sets of possible values for the variables. Local consistency (or arc consistency as defined in [Mackworth 1977] for binary constraints) means: if you pick up a constraint from the network and the covering of one of its variables, you can find values in the other variable coverings, such that the tuple of these values satisfies the constraint.

A globally consistent solution is a value assignment to the variables that satisfies all constraints in the network. If all variables in a constraint network are uniquely determined (cf. [Steele 1980]), or if there are no cycles in the network (cf. [Freuder 1982]), the locally consistent variable covering is identical with the globally consistent solution of the constraint network. Thus, computing local consistency is sufficient in many applications of constraints. Nevertheless, some mechanisms for obtaining global consistency are necessary in some cases such as in the example of this paper.

3 Describing Constraints

In CONSAT, a constraint relation is represented by a finite set of tuples and/or patterns. For short, we will refer to such a tuple or pattern as a constraint element. A tuple represents exactly one possible value combination for the constraint variables, whereas a pattern in general represents a class of value combinations, using functional expressions with references to constraint variables.

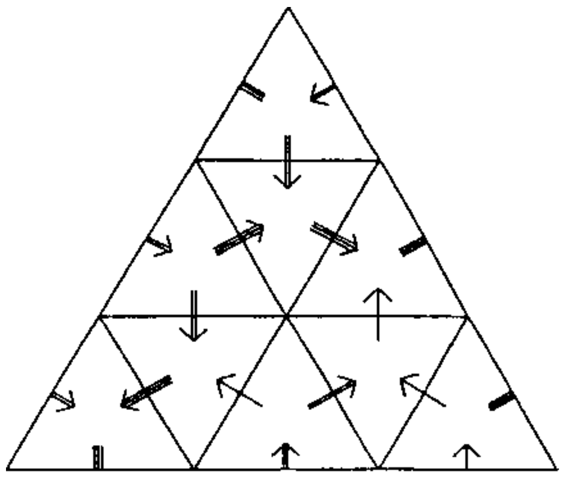


Fig.1: A puzzle with nine triangles.

In the following, the syntax and semantics of constraint elements will be explained by an example from the game world: a puzzle consisting of nine triangles. Figure 1 shows a sketch of the solved puzzle. Every triangle of the puzzle is labeled with a part of a figure. Altogether, there are three different types of figures which are represented in figure 1 by a single, double and triple arrow, respectively.

Since there are nine fields where a triangle may be put, and since every triangle has three possible orientations with respect to the field, solving the puzzle means to struggle with $9!\text{-}3^9$ possible combinations. Only a few solutions exist for the puzzle. The condition for a solution is obvious: the parts of two arrows at adjacent edges must fit, i.e. they both must be parts either of a single, double, or triple arrow.

3.1 Extensional Constraints

The edge labeling of a triangle can be represented in CONSAT by a pair consisting of a key S,D or T for a single, double and triple arrow, respectively, and a reference which determines the part of the arrow (either HEAD or TAIL). Thus, the possible labelings are:

```
(SHEAD)  (STAIL)
(D HEAD) (D TAIL)
(T HEAD) (T TAIL)
```

A triangle with its edge labelings as well as its orientation is described as follows. Looking at a triangle with one of its vertices upwards, we describe the triangle by a tuple, the first element representing the labeling of the left edge, the second one the labeling at the bottom, and the third element the right edge labeling (see figure 2).

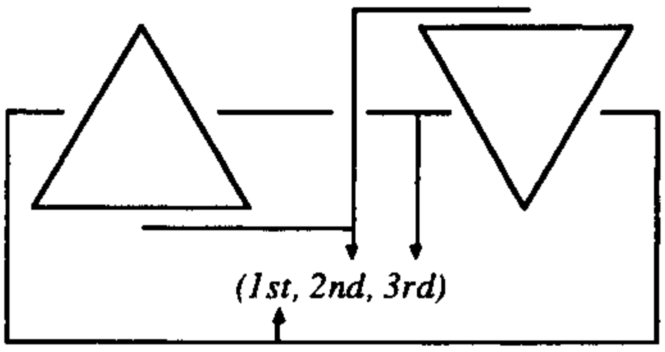


Fig.2: Representation of the triangles.

If we look at a triangle with a vertex downwards, the left edge labeling is represented by the third element of the tuple and the right one by the first element. The second element corresponds to the upper edge labeling (see again figure 2).

With these conventions we can uniquely describe a triangle and the aspects of its orientation relevant for the puzzle. For example, the tuple

```
((D HEAD)  (D TAIL)  (S TAIL))
```

represents a triangle with the head and the tail of a double arrow and the tail of a single arrow as labeling. The orientation in the puzzle may be either with a vertex upwards and the head of the double arrow at the left edge, the tail of the double arrow at the bottom, and the tail of a single arrow at the right, or with a vertex downwards and the single arrow at the left, the tail of the double arrow at the top, and the head of the double arrow at the right.

In order to express the fact that a variable is restricted to one of the above described tuples, the following, extensional constraint can be defined in CONSAT. It describes the admissible tuples by enumerating them:

```
(DEFCONSTRAINT
    (:NAME TRIANGLE)
    (:TYPE PRIMITIVE)
    (: INTERFACE TRIANGLE)
    (:RELATION
        (:TUPLE (((D HEAD) (D TAIL) (S TAIL))))
        (:TUPLE (((S TAIL) (D HEAD) (D TAIL))))
        (:TUPLE (((D TAIL) (S TAIL) (D HEAD))))

        (:TUPLE (((D HEAD) (S HEAD) (S TAIL))))
        (:TUPLE (((S TAIL) (D HEAD) (S HEAD))))
        (:TUPLE (((S HEAD) (S TAIL) (D HEAD))))

        ... further triangles ...))
```

(We abbreviate most of the constraint definitions for better transparency.)

3.2 Intensional Constraints

The representation of relations by tuples, where each tuple specifies an admissible value combination, is impossible if the relation is infinite (and inadequate for extensive relations). Thus, another way of defining constraints must be provided, which—in our opinion—should be compatible to that described in the previous section. In CONSAT, patterns can be used as constraint elements to define constraints intensionally. A pattern consists of expressions with references to the constraint variables.

For example, we can define intensional constraints with two variables (each variable representing a triangle) which are satisfied if the triangles may be placed side by side in the puzzle, i.e. if the labelings of the adjacent edges are complementary. In the puzzle, a triangle with vertex upwards can only be placed beside a triangle with vertex downwards, and vice versa. Thus, we need three constraints, declaring that a triangle is connected to another triangle at the left, bottom or right edge, respectively, where left, bottom and right refers to the triangle with vertex upwards (this convention is arbitrary). For example, the constraint defined by

```
(DEFCONSTRAINT
    (:NAME CONNECTED-LEFT)
    (:TYPE PRIMITIVE)
    (:INTERFACE TRIANGLE-1 TRIANGLE-2)
    (:RELATION
        (:PATTERN (TRIANGLE-1 TRIANGLE-2)
            :IF (COMPLEMENT-P
                    (FIRST TRIANGLE-1)
                    (FIRST TRIANGLE-2))))
    (:CONDITION
        (CONSTRAINED-P
            TRIANGLE-1 TRIANGLE-2)))
```

is satisfied, if the left edge labeling of the triangle with vertex upwards matches the right edge labeling of the triangle with vertex downwards.

The notation of intensional constraints is analogous to the representation by tuples: The ith expression of a pattern describes the value of the ith constraint variable, which is trivial in the case of the CONNECTED-LEFT constraint since they are equal. In general, it is possible to specify arbitrary Lisp forms in a pattern (instead of just referring to variables). In the example, the power of the pattern is grounded in its additional condition (the expression following :IF) which determines whether the pattern may be applied or not.

In order to avoid errors during the pattern matching process, the evaluation of a constraint can be suppressed depending on the actual values of the variables. For example, the CONNECTED-LEFT constraint must not be evaluated if TRIANGLE-1 or TRIANGLE-2 is equal to the keyword UNCONSTRAINED, which represents the (possibly infinite) set of all domain values. This restriction can be declared by a global condition associated with the constraint, for instance:

```
(:CONDITION (CONSTRAINED-P
                    TRIANGLE-1 TRIANGLE-2)).
```

4 Evaluating Constraints

When a constraint is activated by the propagation control unit, by another subsystem, or by the user (via a SATISFY expression), some of its variables may already be associated with sets of possible values. The system first computes all combinations from these lists, setting the unconstrained variables to UNCONSTRAINED.

Let the variables of the CONNECTED-LEFT constraint be associated with the following variable coverings:

```
TRIANGLE-1:   (((D TAIL)  (S TAIL)  (D HEAD)))

TRIANGLE-2:   (((D HEAD)  (S HEAD)  (S TAIL))
               ((S TAIL)  (D HEAD)  (S HEAD)))
```

then two combinations are computed:

```
(((D TAIL) (S TAIL) (D HEAD))
((D HEAD) (S HEAD) (S TAIL)))

(((D TAIL) (S TAIL) (D HEAD))
((S TAIL) (D HEAD) (S HEAD)))
```

Each combination of values is matched with the constraint elements in the following way. If the constraint element under consideration is a tuple, its values are compared with the corresponding values in the given combination. If it is a pattern, its condition (if provided) is evaluated first. If the condition evaluates to T or if no condition is provided, then the rest of the pattern is evaluated as follows: all atoms and expressions in the pattern are evaluated and the computed values are compared with the corresponding values in the given value combination.

In both cases, the matching process succeeds if for every value in the given combination the following holds:

- it is equal to UNCONSTRAINED, or

  it is equal to the corresponding value of the constraint element.

Whenever the matching process succeeds, the computed values are added to the resulting variable coverings. They are returned after all value combinations have been matched with all constraint elements, for example:

```
(SATISFY CONNECTED-LEFT :WITH
    TRIANGLE-1 = ((D TAIL) (S TAIL) (D HEAD))
    TRIANGLE-2 = (:ONE-OF
                    ((D HEAD) (S HEAD) (S TAIL))
                    ((S TAIL) (D HEAD) (S HEAD)))

———>

((TRIANGLE-1   ((D TAIL) (S TAIL) (D HEAD)))
(TRIANGLE-2   ((D HEAD) (S HEAD) (S TAIL))))
```

Formally, the result of evaluating a constraint C can be denoted as follows. Let $v_1,\ldots,v_n$ be the input variable coverings for C and R be the set of constraint elements (tuples or patterns) of C Then, the evaluation of C results in

$$pr_{1,\ldots,n}((v_1 \times \ldots \times v_n) \cap R_{inst})$$

where

$pr_{1,\ldots n}$ is used as a short form for the tuple of projection functions $(pr_1,\ldots,pr_n)$, for example $pr_{1,2}((a,b), (c, d)\} = ((a,c), \{b,d\})$,and

R_{inst} is the instantiated relation of C which is obtained from R by evaluating the patterns with the variables successively bound to elements from $v_1 \times \ldots \times v_n$.

5 Compiling Constraints

First, we will revisit the above evaluation algorithm, transforming the term which describes the result computed by the algorithm.

Let R_{inst} be equal to $\{r_1,\ldots,r_m\}$ with $r_i = (r_{i1},\ldots,r_{in})$. Then:

$$pr_{1,\ldots,n}((v_1 \times \ldots \times v_n) \cap R_{inst}) =$$

$$pr_{1,\ldots,n}((v_1 \times \ldots \times v_n) \cap \{r_1\} \cup \ldots \cup (v_1 \times \ldots \times v_n) \cap \{r_m\}) =$$

$$pr_{1,\ldots,n}((v_1 \times \ldots \times v_n) \cap \{(r_{11},\ldots,r_{1n})\} \cup \ldots \cup (v_1 \times \ldots \times v_n) \cap \{(r_{m1},\ldots,r_{mn})\}) =$$

$$pr_{1,\ldots,n}((v_1 \times \ldots \times v_n) \cap \{(r_{11},\ldots,r_{1n})\}) \cup \ldots \cup pr_{1,\ldots,n}((v_1 \times \ldots \times v_n) \cap \{(r_{m1},\ldots,r_{mn})\})$$

To determine $pr_{1,\ldots,n}((v_1 \times \ldots \times v_n) \cap \{(r_{i1},\ldots,r_{in})\})$, it is not necessary to compute the Cartesian product between $v_1,\ldots,v_n$, as the computation can be reduced to maximal n member operations:

$$pr_{1,\ldots,n}((v_1 \times \ldots \times v_n) \cap \{(r_{i1},\ldots,r_{in})\}) =$$

$$\begin{cases} (\{r_{i1}\},\ldots,\{r_{in}\}) & \text{if } r_{i1} \in v_1 \wedge \ldots \wedge r_{in} \in v_n \\ (\{\},\ldots,\{\}) & \text{else} \end{cases}$$

This means that a constraint can be evaluated more efficiently as it is sketched in the previous section.

Such an improvement presupposes that the constraint relation is already instantiated. In general, this is not the case, since patterns may occur as constraint elements. If all constraint variables are used within at least one of these patterns, one has to compute the Cartesian product between the variables to obtain an instantiated relation, i.e. the improvements concerning evaluation are lost.

Our idea to resolve this dilemma can be summarized as follows: instead of answering the question "How can we implement a program that evaluates an arbitrary CONSAT constraint?", we are guided by the question "How can we implement a program whose input is a constraint and which produces a program that evaluates this particular constraint optimally?".

In particular, a constraint is analyzed when it is defined to determine whether or to which extension the Cartesian product has to be computed during evaluation. As a result, we obtain a procedure which minimizes the costs for evaluating the constraint. For example, consider a constraint with three variables and constraint elements that use only two of them. Then, the costs for the evaluation of the constraint can be reduced up to a factor k, where k is the number of values in the covering of the non-used variable.

6 Constraint Networks

Constraint networks consist of a set of primitive constraints or lower level constraint networks which are connected by shared variables. They have the same i/o behavior as primitive constraints, i.e. a constraint network is evaluated with a variable coverings as input, and the result is set of filtered variable coverings.

In particular, a constraint network contains constraints, global variables, variable coverings, mappings between the local constraint variables and the global network variables, and an interface. The interface is a subset of the global variables which may be associated with initial values.

For the definition of constraint networks, the same constructs can be used as for primitive constraints. Consider, e.g., the puzzle with the variables P1, P2,..., P9, each variable representing a position where a triangle may be placed, then the following definition specifies the constraint network which describes the relationships that must be maintained in order to solve the puzzle:

```
(DEFCONSTRAINT
    (:NAME NINE-PUZZLE)
    (:TYPE COMPOUND)
    (:INTERFACE P1 P2 P3 ...)
    (:CONSTRAINT-EXPRESSIONS

        (TRIANGLE P1) (TRIANGLE P2)...

        (CONNECTED-MIDDLE P1 P3)
        (CONNECTED-RIGHT P2 P3)...

        (DIFFERENT-TRIANGLES P1 P2)
        (DIFFERENT-TRIANGLES P1 P3) ...))
```

The mapping of global network variables to local constraint variables is given implicitly in the above definition. Similar to a function call in Lisp, the expression (CONNECTED-MIDDLE P1 P3), e.g., means that P1 and P3 are mapped to the local variables TRIANGLE-1 and TRIANGLE-2 of the CONNECTED-MIDDLE constraint, respectively.

Since constraint networks work just like primitive constraints (viewing them just from their i/o behavior), hierarchies of constraints and constraint networks can be built straightforwardly by referring to constraint networks rather than to primitive constraints within another constraint network. A consequence of this method is that recursive constraint networks can be defined, e.g. by referring to a network within the network itself.

7 Satisfying Constraint Networks

The standard control strategy of CONSAT is based on local propagation: when a constraint network is activated (via the above introduced SATISFY) with initial values for some interface variables, the constraints which are imposed on these variables are evaluated. The result of the evaluation is propagated to their neighboring constraints, and so on. The activation process terminates when no further changes of the variables are carried out. Since the variables arc filtered by the constraints, the propagation process always stops in the case of non-recursive constraint networks (cf. [Guesgen, Hertzberg 1988]). The values of the interface variables are returned as a result of the local propagation process.

Local propagation computes local consistency which is insufficient in some cases (as, e.g., for the above introduced puzzle). Thus, CONSAT provides further control strategies for obtaining global consistency, one of them is a combination of local propagation and backtracking (LPB). If the result of local propagation is not unique, LPB sets a choice point, i.e. a value is selected from a variable covering, guided by some simple heuristics. Then, local propagation is started again, followed by setting another choice point, and so on. If an inconsistency has been detected during local propagation, or if a unique solution has been found but further solutions are requested (the user can limit the number of solutions to be computed), a choice point is backtracked and another choice point is set. The algorithm breaks off when a sufficient number of solutions has been computed or no further choice points can be set.

The application of the LPB algorithm to the NINE-PUZZLE constraint network yields the following solution:

```
(SATISFY NINE-PUZZLE :GLOBALLY 1)


(((P1  ((T TAIL) (D TAIL) (D HEAD)))
(P2  ((D HEAD) (D TAIL) (T TAIL)))
(P3  ((T TAIL) (D HEAD) (T HEAD)))
(P4  ((T HEAD) (S HEAD) (T TAIL)))
(P5  ((D HEAD) (T TAIL) (T HEAD)))
(P6  ((S HEAD) (D HEAD) (T TAIL)))
(P7  ((S TAIL) (D HEAD) (D TAIL)))
(P8  ((S HEAD) (S TAIL) (D HEAD)))
(P9  ((S TAIL) (S HEAD) (T TAIL)))))
```

A (straightforward) Prolog program seems also to be adequate to model the puzzle. However, the search space of such a program is orders of magnitude larger than the search space CONSAT actually needs. The reason is that local propagation (applied after setting a choice point) can reduce the depth of the search tree by a factor 2.

Besides LPB, another control strategy is provided by CONSAT. This strategy is based on local propagation and uses indices to maintain relationships between values during the propagation process. It is beyond the scope of this paper to describe this method in detail. A more detailed description can be found in [Guesgen 1989].

8 Conclusion

The purpose of this paper was to introduce the constraint system CONSAT, which combines the advantages of a domain-independent programming language a la [Steele 1980] and the efficient problem solving control strategy from [Waltz 1972].

In particular, CONSAT can handle arbitrary domains, i.e. the constraint variables may be associated with any kind of values. Sets of possible values rather than single values are propagated through constraint networks. Thus, CONSAT can be applied

- to compute values for the undetermined variables from the set of given variables,
- to test consistency of values, or
- to filter the sets of possible values, i.e. to remove inconsistent values from these sets.

Several ways to represent relations are provided. Finite relations can be represented by enumerating the extension of the relation. Infinite relations can be described intensionally by patterns. More complex relations can be represented by constraint networks and hierarchies of constraint networks. Besides local propagation as control strategy, CONSAT provides two mechanisms for obtaining globally consistent solutions.

CONSAT is used in several applications, for example, in a system for process diagnosis [Voss 1988], where constraints arc applied to maintain functional relationships in physical units, and in the musical domain [Lischka, Guesgen 1986], where the harmonization of chorals is supported by constraint propagation.

Beyond that, CONSAT was incorporated into the hybrid knowledge representation system BABYLON in order to use constraints together with rules, frames, and Prolog [Guesgen el al. 1987] which turned out to be most useful.

Acknowledgements

Thanks to the members of the Expert System Research Group in GMD who contributed to this paper, especially thanks to Ursula Bernhard and Marc Linster who read former versions. Ulrich Junker modelled the puzzle and helped to implement and to test CONSAT. The partners of the TEX-B project discussed many of the ideas introduced in this paper.

References

[Freuder 1982]
E. C. Freuder: A Sufficient Condition for Backtrack-Free Search. Journal of the ACM 29 (1982) 24-32.

[Guesgen 1989]
H.W. Guesgen: CONSAT — A System for Constraint Satisfaction. Research Notes in Artificial Intelligence, London: Pitman, 1989 (to appear).

[Guesgen etal. 1987]
H.W. Guesgen, U. Junker, A. Voss: Constraints in a Hybrid Knowledge Representation System. In: Proceedings of the IJCAI87, Milan, Italy, 1987, 30-33.

[Guesgen, Hertzberg 1988]
H.W. Guesgen, J. Hertzberg: Some Fundamental Properties of Local Constraint Propagation. Artificial Intelligence 36 (1988) 237-247.

[Lischka, Guesgen 1986]
C. Lischka, H.W. Guesgen: M v S I C — A Constrained-Based Approach to Musical Knowledge Representation. In: Proceedings of the International Computer Music Conference 86, The Hague, The Netherlands, 1986, 227-229.

[Mackworth 1977]
A. K. Mackworth: Consistency in Networks of Relations. Artificial Intelligence 8 (1977) 99-118.

[Steele 1980]
G.L. Steele: The Definition and Implementation of a Computer Programming Language Based on Constraints. Technical Report AI-TR-595, Massachusetts Institute of Technology, Cambridge, Massachusetts, 1980.

[Voss 1988]
H. Voss: Architectural Issues for Expert Systems in Real-Time Control. To appear in: Proceedings of the 1st IF AC Workshop Artificial Intelligence in Real-Time Control, 1988.

[Waltz 1972]
D.L. Waltz: Generating Semantic Descriptions from Drawings of Scenes with Shadows. Technical Report AI-TR-271, Massachusetts Institute of Technology, Cambridge, Massachusetts, 1972.