V: A Visual Query Language for Multimodal Interface

Robert Nilsson
robnil@sics.se

Kent Saxin Hammarström
kent@csd.uu.se

June 1994

Natural Language Processing Group
Swedish Institute of Computer Science
Box 1263, S-164 28 KISTA
SWEDEN

Abstract:
This report proposes a two-dimensional, visual, direct manipulation query language intended to be used as an alternate modality to natural language in a multimodal interface. The language focuses on the visualisation of the logic of queries, and is intended to be flexible, extensible, and to have at least the expressive power of first order predicate logic with constraints. The language provides a basis for future inclusion of higher order logic (and thus a framework for database navigation within the language itself), and generalised quantifiers. As far as we have been able to judge, no such visual language of equal expressive power already exists. A "proof of concept" prototype has been implemented that illustrates most of the key concepts of the language.

Keywords: Multimodal interface. Database query. Visual language

Also published: This report is also available as Uppsala Master's Theses in Computing Science 60, ISSN 1100-1836.
# Table of Contents

1 Introduction ................................................................................. 1  
   1.1 Interactions between the modalities of the interfaces .................. 3  
2 The Language ............................................................................... 7  
   2.1 Base Language ........................................................................ 7  
      2.1.1 Entities ........................................................................... 8  
      2.1.2 Relations ........................................................................ 9  
      2.1.3 Queries .......................................................................... 10  
   2.2 Full Language ........................................................................ 10  
      2.2.1 Quantifiers ...................................................................... 11  
      2.2.1.1 Syntax ....................................................................... 11  
      2.2.1.2 Semantics ................................................................... 12  
      2.2.2 Composite entities ............................................................... 12  
      2.2.2.1 Constraints .................................................................. 12  
      2.2.2.2 Set operations ................................................................. 13  
      2.2.3 Logical connectives ............................................................... 13  
      2.2.3.1 Negation ..................................................................... 13  
      2.2.3.2 Conjunction and disjunction ........................................ 14  
   2.3 Simplified language ................................................................. 14  
      2.3.1 Quantifiers ...................................................................... 15  
      2.3.2 Composite entities ............................................................... 16  
   2.4 Extensions ............................................................................. 16  
      2.4.1 Higher-order relations .......................................................... 16  
      2.4.2 Generalised quantifiers ....................................................... 18  
      2.4.3 Adding declarative statements ........................................... 19  
3 The user interface ......................................................................... 21  
   3.1 Design Overview ..................................................................... 21  
   3.2 Functionality ........................................................................... 23  
      3.2.1 Static vs dynamic syntax ................................................... 23  
      3.2.2 The query board ................................................................. 24  
      3.2.3 An informal description of dynamic syntax and semantics 24  
         3.2.3.1 Creating entities ......................................................... 24  
         3.2.3.2 Creating relations ....................................................... 24  
         3.2.3.3 Changing entities and relations ................................... 25  
         3.2.3.4 Different box drawing modes - the toolbox .............. 25  
         3.2.3.5 Selecting objects ........................................................ 26  
         3.2.3.6 Cutting and pasting ...................................................... 26  
         3.2.3.7 Creating macros - saving queries .............................. 26  
         3.2.3.8 Quantifying entities .................................................... 26  
         3.2.3.9 Quantifying in the full language ............................... 26  
         3.2.3.10 Quantifying in the simplified language .................. 27  
         3.2.3.11 Composite entities ................................................... 27  
         3.2.3.12 Negation ................................................................. 28  
         3.2.3.13 Reducing and enlarging icons and relations ........... 28  
         3.2.3.14 Hierarchical history tree of queries ....................... 30  
         3.2.3.15 Inheritance hierarchy for objects ......................... 31  
   3.3 The Implementation .............................................................. 32
3.3.1 Design Realisation ......................................................... 32
3.3.2 The interface ................................................................. 34
  3.3.2.1 The menu bar ....................................................... 34
  3.3.2.2 The button tile .................................................... 34
  3.3.2.3 The realisation of the query board ......................... 34
  3.3.2.4 The text tile ....................................................... 35
3.3.3 Translation to TRL ....................................................... 35
3.4 Further work on the implementation ................................. 36

References ............................................................................. 41
Appendices .............................................................................. 43
  A Additional query samples ............................................. 43
  B Formal syntax of the full and simplified language ............ 44
    B1 Syntax for the full language .................................... 44
    B2 A BNF for the simplified language ......................... 47
  C Syntax of TRL ................................................................. 48
  D Description of Resource Files ....................................... 49
    D1 V Resources .............................................................. 49
    D2 Application Resource .............................................. 49
    D3 Application Definitions Resource ............................ 51
    D4 Requirements for the application(s) ....................... 51
  E "User Manual" ................................................................. 51
1 Introduction

The original goal of the project described in this thesis was to implement a bimodal interface, using an existing natural language interface (based on the Core Language Engine, CLE: Alshawi et al., 1992; Rayner, 1993; SRI, 1992) and a visual language. However, as we soon discovered that we could find no language both visual enough and expressive enough to be interesting as an alternate modality to natural language, the emphasis of the project shifted to one of language design. The remainder of this report therefore describes the first draft of such a language, $V$, and a "proof of concept" prototype that implements some aspects of it.

As the term "visual language" can have a number of different interpretations, depending on the reader, in this report we use the definition by Catarci, Massari and Santucci (1991) that a language being visual "... implies the systematic use of visual expressions (such as icons, drawings or gestures) to convey meaning in a formal way." They further subdivide visual languages into iconic ones (those using images and icons extensively) and graphical ones (those primarily using drawings, such as graphs, flow charts, block diagrams, etc.). According to this classification, $V$ is primarily graphical, though with iconic elements.

Another way to classify visual languages (Epstein, 1991) is to describe them along the dimensions of visual metaphors used, expressive power, and the underlying computational mechanisms. According to this scheme, $V$ can be described as primarily using the metaphors of the Entity-Relationship model (i.e. boxes representing objects, connected by named arcs representing the relations between them), and having the expressive power of at least full first order predicate logic (FOPL) with constraints (i.e. the objects need not be predefined, but can be described by subqueries). The underlying computational mechanism is irrelevant, as $V$ is a purely declarative language.

To complement these definitions, it is necessary to add that $V$ is a direct manipulation language, by which we mean that the components of an expression in the language can be manipulated (moved, modified, etc.) directly on-screen with a pointing device. Furthermore, it is primarily intended to visualise the structure of queries/problems, rather than the atomic concepts involved, as this often seems to be the major stumbling block in formulating queries.

As mentioned, $V$ is intended to be used in a bimodal interface. Modality, according to Nigay (1993) "... refers to the type of communication channel used to convey or acquire information. It also covers the way an idea is expressed or perceived, or the manner an action is performed." Bimodality is a special case of multimodality, which (ibid.) is "... the capacity of the system to communicate with a user along different types of communication channels and to extract and convey meaning automatically." The key concept here is "... to extract and convey meaning automatically." A voice mail system, for example, is not a multimodal system, but multimedia, as it does not extract meaning from the data it transfers.

Most of the existing visual query languages initially surveyed (Epstein, 1991; Haddock, 1992; Staes et al., 1991; Catarci, Massari & Santucci, 1990; Infotool AB) were either not visual enough, in that they do not support visualisation of the structure of queries (as opposed to visualisation of atomic concepts), or insufficiently expressive, using only a subset of FOPL. In our opinion, to be interesting as an alternate modality to natural language, a visual language would need at least the power of first order predicate logic.

However, there are three languages that approached our ideal, and that have influenced the design of our language. These are:
• Intuitive (SISU, 1993):
The query language used in Intuitive satisfies our requirements as to degree of visuality, but is not expressive enough. Its power is approximately equal to the base level of our language, i.e. conjunctions of existentially quantified variables constrained by a single unary predicate.

• The language of Whang et al. (1992):
This language also satisfies our requirements as to degree of visuality, and is more powerful than the query language of Intuitive. It does not, however, have the full power of FOPL. For example, the query “List the divisions such that for each blue item there is a department in the division that sells the item” can be expressed, but “List the divisions owning a department that sells all the blue items” cannot. In addition, the reading direction of queries is unclear.

• The language of Bretan (1990):
This language has both sufficient visuality and sufficient expressiveness. However, we dislike its use of explicit variables, as they make connections between the different views of the same variable instances difficult to perceive. Furthermore, its structure seems too closely tied to that of natural language to provide the maximal benefit of having a second modality, and it cannot handle user defined entities (what we call composite constraint entities), except as material implications.

One other language that is quite similar to V, but that has had no direct influence on it, is that of Conceptual Graphs (Sowa, 1984). This language has an expressiveness and syntax reminiscent of that of Intuitive and the base level of V, but a different emphasis, as it is mainly geared towards representing propositions in semantic nets. However, as the formal semantics of conceptual graphs have been worked out to a much larger degree than for any other visual language that we have looked at, it would seem a good starting point for the formal semantics of V.

V is a direct manipulation language, by which we mean that the elements (words) of the language can be directly manipulated on the screen, using a pointing device such as a mouse. Using a direct manipulation interface (DMI), as opposed to a graphical interface in general, has a few implications in itself, the major one being the immediacy of query construction.

This has as a result that it is very easy to construct queries incrementally (Staes et al., 1991; Bretan & Karlsgren, 1993), especially if some form of “macro” capacity and/or history recording mechanism is included (i.e. the capability to store and retrieve (sub)queries and/or composite entities and/or templates for such). Stored (sub)queries/entities can easily be retrieved and used as components in a new query by connecting them in a suitable way. Furthermore, queries can easily be refined in a stepwise way based on feedback, either by evaluating the intermediate queries and modifying them until the sought answer is obtained and/or through the use of paraphrases (see section 1.1).

One factor that we consider of paramount importance in constructing visual languages in general, and direct manipulation languages in particular, is consistency. According to Payne and Green (1986) (speaking in particular of action languages, i.e. languages for describing the actions that can be performed in an interface, but the principles would seem to be of general applicability) consistency of a language can be subdivided into at least four different factors, viz:

• Syntactic consistency: Constructs of a language should, as far as possible, belong to a limited set of hierarchically related syntactic families. Loosely
speaking, there should be no "irregular words" in the language.

- **Lexical consistency**: The lexemes of an artificial language should be as closely tied to their meaning as possible, and the lexical relations should be congruent to the semantic relations of the lexemes. The obvious way of achieving one form of lexical consistency in V is to make the meaning of the atomic icons as intuitive as possible.

- **Semantic-syntactic alignment**: Congruence should be displayed not only between lexemes and their semantics, but also in the structure of composites (s.l.). Conversely, the semantics of constructs in the language should not be context dependent, i.e. the language should be compositional. A simple example of this would be that quantification of a composite entity should have the same syntax as quantification of an atomic entity, ceteris paribus.

- **Semantic consistency**: Simply expressed, semantic consistency means that semantic language constructs should have as few arbitrary restrictions on possible subcomponents as possible. For example, given that the language has quantification and the possibility to refer to objects through description (subqueries, or what we call “constraints”), then it should be possible to use anything that can be quantified as a component of a constraint and vice versa. Similarly, if an atomic semantic entity can be quantified then so should a composite entity. Semantic consistency, together with semantic-syntactic alignment, is sometimes referred to as orthogonality of a language.

One very important part of the language and interface that we have not treated very deeply, partly due to the limited scope of the project and partly because it is not within our area of competence, is their cognitive aspects. Before this language and interface could become a practically useful tool, we therefore consider the input of a specialist in user interface design to be critical. As an example, the language would probably be much easier to use if it were possible to use colour as an additional syntactic dimension. It is, however, important not to require colour use, as the availability of colour monitors is not universal.

One major aspect of the visual language that is missing in the current report is a full formal semantics for at least the base language and the full language. We have made a start on such a description, but did not deem it complete enough to be interesting to include in this report.

Further, no formal syntax and semantics description is made for the action language of the interface, since we consider it not within the scope of this paper.

### 1.1 Interactions between the modalities of the interfaces

Having both a natural language modality and a visual language modality in an interface is advantageous in that these modalities tend to have complementary expressiveness.

Natural language, as supported by natural language interface, tends to be verbose, ambiguous and vague, and syntactic errors are easy to make, but it is usually more flexible and natural (cf. Bretan, 1990; Bretan & Karlsgren, 1993; Cohen et al., 1989) than visual languages. The "naturalness" means that it is easier to learn than other languages, and the flexibility comes from its capability to express semantically complex expressions and the capability to express meaning through contextual anchoring. The ease of learning, however, is offset by the opacity of the linguistic and conceptual coverage of natural language systems, as this coverage presumably never will even approximate that of a human in the same situation. The user does not have full freedom of expression in an NLI, as both the syntactic constructions and the underlying knowledge structure is lim-
ited, but what is and what is not allowed is not immediately obvious.

As to verbosity, however, it is worth noting that it is not automatically true that
the natural language formulation of a query is more verbose than the visual lan-
guage formulation. Consider for example the query "Who sold more than Mols?", which is quite succinct in NL, expressing much information through contextual
anchoring, as compared to its complex visual counterpart (see figure 27 in appen-
dix A).

Visual languages, on the other hand, can be succinct and precise, and syntactic
mistakes can be prevented, as the system can be made so as to only provide "cor-
correct" actions in each situation. On the other hand, the traditional weaknesses of
visual languages (and direct manipulation interfaces in particular) are the diffi-
culties of expressing semantically complex information and contextual informa-
tion. We believe that V circumvents some of these difficulties, in that some forms
of complex information (object reference through description, see 2.2.2 on page 12
and 2.3.2 on page 16) and quantification information (See 2.2.1 on page 11 and
2.3.1 on page 15) are reasonably easy to express.

Among the specific advantages of a bimodal interface, as opposed to having two
separate interfaces, are the possibilities of improved learning and incremental
query construction through the cross-modal paraphrasing of (possibly partial)
queries, and user control of discourse structure through reification and mainte-
nance of discourse objects and deixis (cf. Bretan, 1990; Bretan & Karlsgren, 1993;
Cinque et al., 1991). The degree to which it is possible to realise each of these is
dependent on the degree of integration between the interfaces, paraphrasing
requiring the lowest and discourse management and deixis the highest.

• Paraphrases
In this context, "paraphrasing" means to render a natural language query in
the visual language and vice versa. For this to be possible, both analysis and
generation components are required in both modalities.

• Improved learning
Learning the syntax of the visual language is simplified, as the beginning
user can formulate queries in natural language and have them para-
phrased in visual language, thus getting immediate feedback on the syntax
of the visual language.
Analogously, learning the syntactic and conceptual coverage of the natural
language interface is simplified, as queries can be formulated visually and
then paraphrased in natural language.

• Incremental query construction
In this context (see also above), incremental query construction means that
when the formulation of a query is not immediately evident in one modali-
ity, one can make an initial attempt in that modality, paraphrase it and
continue the construction in the other one by means of direct inspection
and modification. Similarly, queries in NL that turn out have been not
quite correctly formulated can be paraphrased in VL and modified. Incre-
mentality is also achieved directly in the visual interface due to its interac-
tivity.

• Deixis and discourse management
Deixis is here taken to be the ability to make intermodal cross-references, i.e. to "point at" expressions in the alternate modality. One would for example be
able to pose the query "How many $\star$ are there?", where "$\star$" would be a refer-
tence to a visual (sub)query.
In this way the discourse objects can be reified, making them explicit and per-
sistent, thus putting the management of referents in the NLI under explicit user control. Following the terminology of Luperfoy (1992), the V entities would seem to be a realisation of the conflation of her second and third tier discourse objects.

- Cross-modal natural language analysis recovery (Bretan & Karlsgren, 1993)
  A rather elegant way of recovering from analysis failures in the NL component would be to ask the VLI to show for example the "maximal entity descriptions" (recognized noun phrases) and let the user add the missing components to the visual expression.

An example (ibid.) is the sentence "What colleges are pretty much within walking distance of the centre of the city?", which a particular NLI (the CLARE system (Rayner, 1993; SRI, 1992)) cannot parse completely. When asked to display the corresponding maximal entities of the query, CLARE responds with "what colleges" and "the centre of the city". Now, if this were sent to the V VLI, it could be paraphrased as shown in figure 1, and it would

![Diagram](image)

**Figure 1.** Visual output of the failed analysis of the NL query "What colleges are pretty much within walking distance of the centre of the city?"

be a simple matter for the user to connect the college entity to the centre entity with a suitable relation meaning "close to". Possible relations could even be suggested by the system, for instance as described in 2.4.1.
2 The Language

The visual language described in this report is loosely based on the syntax for the Entity-Relationship model (Chen, 1976), in that it uses the model of unique entities (objects belonging to subsets of the universe that satisfy a particular unary predicate) and relations between these entities that further restrict the entities, but is to a large extent influenced by the languages of Bretan (1990) and Whang et al. (1992).

An important consequence of this model is that we do not use explicit variables in our language. These are really only necessary in linear languages to express non-linear connections between entities (e.g. pronouns in natural language, variables in standard FOPL, and the so-called “event variables” and “state variables” commonly used in NLP).

The overriding concerns in designing this language have been to provide:

- Not a monolithic language that the user has to adapt to, but instead a coherent system of languages, such that the user can choose the level of complexity that she needs.

- An extensible language, i.e. a language that can be used as a base for even more powerful languages, that has at least the power of first order predicate logic.

- A language that is consistent at as many levels as possible (cf. “Introduction” on page 1), though lexical consistency (i.e. making the meaning of the entity pictures intuitively obvious) has been of low priority during the project.

- A language for which it would be fairly easy to describe the syntax and semantics formally. This includes both what we call the “static” syntax and semantics (i.e. the visual statements themselves), and what we call the “dynamic” syntax and semantics (i.e. how one static statement is transformed into another one).

- A language that is as visual as possible, at least in the sense of visualising query/problem structure (i.e. the logic of the statements).

One consequence of these design principles is that our language seems to differ from many other visual languages in that we have neither been concerned about the execution of the visual statements, as that belongs to the domain of the application(s) and interface modules, nor about diverging from the structure of natural language, as we consider the main point of providing an alternate modality to not mirror the other modality too closely. As an example, the use of intramodal cross-reference and anaphora is no problem in V, due to the uniqueness of the entity symbols.

In the following we have divided the description of V into three separate subsections: “The base language”, “The full language,” and “The simplified language” respectively. This division does not mirror any absolute difference between the variants of the language, but is rather a pragmatic classification scheme to highlight different aspects of the language. In reality, our intention is that the user would “mix and match” from these levels, using a consistent dynamic syntax to express himself, while the interface itself would take care of expressing the statements in the syntax of an appropriate level.

2.1 Base Language

The base language is simply the least common denominator for the V languages, containing nothing but atomic entities and relations. This level has the minimum level of expressiveness for V, handling conjunctions of relations between existen-
tially quantified entities, and is thus quite similar in expressiveness to the query language of Intuitive (SISU, 1993). Our base language, however, can also handle negation.

2.1.1 Entities

An (atomic) entity is one of the two kinds of atoms (words) of the base language, the other one being relations (cf. 2.1.2 on page 9).

Syntactically an entity is a box with a name (the name of the class of objects that it represents), a bitmap (that should mirror its intended lexical semantics), and (optionally) the name of an instantiation, which is a particular object of that class, (see fig. 2).

![Figure 2. An atomic entity, the set of kings. \( \exists X. \text{king}(X) \)]

Semantically an entity box represents a subset of the universe, viz. those objects for which the unary relation identified by the entity name holds true, i.e. a constraint. All entities are implicitly existentially quantified.

In addition to these standard entities (that the "user" can specify), the language provides for system entities. Currently there is only one such, the date entity, used for specifying time (see fig. 3).

![Figure 3. The date entity. The arrows of each attribute are used to increase/decrease their respective values (all attributes need not be specified).]

Entities can be negated, meaning the set of objects that do not belong to the indicated class, by a cross over the box. The implicit quantification is outside of a negation, such that the semantics of figure 4 is \( \exists X. \neg \text{king}(X) \), rather than \( \neg \exists X. \text{king}(X) \).

![Figure 4. Entity negation. \( \exists X. \neg \text{king}(X) \)]
2.1.2 Relations

The second atom of the base language is the relation, which syntactically is a diamond, containing the relation name, connected by directed or undirected arrows to the participating entities, which in turn can be other relation diamonds. In principle relations can be n-ary, where the number of arcs indicate arity, and arrows show direction (e.g. fig. 5 on page 9). If higher arities than three

![Diagram of relations](image)

**Figure 5.** The most common types of relation symbols.

are needed, one would have to resort to for example numbering the arcs.

Relations form the body of visual queries, i.e. given that the participating entities belong to the specified sets, do any tuples of entities from these sets exist such that the (implicit) conjunction of the given relations is true. Linguistically relation symbols usually correspond to verbs, but can also be, for example, adjectives (unary relations on entities) or other modifiers of nouns, or adverbs or other modifiers of verbs (relations in which at least one participating object is a relation tuple).

The semantics of a relation diamond participating in another relation, is that tuples of the first relation participate in the second one. We consider such constellations to be decompositions of higher arity relations, e.g. the semantics of figure 7 would be more or less equivalent to that of figure 6, except as noted below.

Such decompositions of higher arity relations would seem to increase the ease of use of the language by separating common modifications to relations, such as the time at which something occurred or where it occurred, from the kernel relation. In addition, they allow the possibility of not having to specify all possible modifications. Without this capability, the language would probably be unusable, as it each relation would have to include each possible modification, which would form an infinite set of relation variants. The formal expressiveness, however, would be the same.

However, these “hierarchical relations” introduce a new problem. Due to the extensionality of mathematical sets, there is no way of distinguishing between a tuple as member of different relations. An example of the kind of problems this
Figure 6. "Did any persons go to Sweden during 1993?"

\[ \exists x \in \{ y \mid \text{person}(y) \}; \]
\[ \exists z \in \{ t \mid \text{country}(t) \land t = \text{Sweden} \}; \]
\[ \exists s \in \{ r \mid \text{time}(r) \land r.year = 1993 \}; \]
\[ \text{go_to_at_time}(x, s, z) \]

gives rise to, is the following scenario: John went to Sweden at some unspecified time, and he also lived in Sweden during 1992. The tuple \(<\text{John}, \text{Sweden}>\) thus belongs to both the relation \(\text{go_to}\) and the relation \(\text{live_in}\), and the tuple \(<\text{John}, \text{Sweden}>, 1992>\) is a member of the relation \(\text{at_time}\). If one were to pose the query "Who went to Sweden during 1992?", one of the answers would, erroneously, be John. To avoid this kind of problem, relation tuples have to be tagged in some way that indicates as members of which set they are referred to. This could presumably be achieved in a number of ways, but we have chosen what seems to be the standard technique in computational linguistics, namely "event variables" (cf. Hobbs, 1985).

An event variable is an extra argument added to relations, and it is this extra entity that actually participates in the hierarchical relations, thus tagging the participating tuples. In the visual queries, this event variable is implicit, but in the textual representations (cf. fig. 7) it needs to be made explicit by inserting an extra argument in the relations.

Negation of relations is consistent with negation of entities, in that it is shown by a cross over the relation diamond. The semantics is quite simple: If an undotted relation stands for \(\varphi(X)\), then the same relation with a cross over the diamond means \(\neg \varphi(X)\).

2.1.3 Queries

In itself, a visual statement in \(\text{V}\) represents an existentially quantified conjunction of predicates. Interpreted as a query, this represents a yes/no question ("Do any such tuples exist?"). To represent a WH question ("Which such tuples exist?"), the interest marker is introduced. In database terminology this is the projection of those entities: Those entities one wishes to list are marked with a heavy border (see example in figure 7).

2.2 Full Language

The full language is a superset of the base language with explicit quantification, both universal and existential, and thereby the full power of first order predicate logic, and constraints (composite entities).
Figure 7. "Which persons went to Sweden during 1993?"

\[ \{ P \mid \exists M \in \{ X \mid \text{person}(X) \} \}
\exists C \in \{ Y \mid \text{country}(Y) \land Y=\text{Sweden} \}.
\exists E \in \{ Z \mid \text{event}(Z) \}.
\exists T \in \{ R \mid \text{time}(R) \land R.\text{year} = 1993 \}.
go_{to}(M, C, E) \land \text{at_time}(E, T) \land P=M \}

The notation we have chosen for these concepts is to a large degree influenced by the boundary notation of James & Bricken (1992) and by the visual query languages of Whang et al. (1992) and of Bretan (1990), in that we use enclosing boxes to indicate such information as scope of quantifiers and extent of composition. We use two basic kinds of boxes\(^1\): One which relations can cross, which we call a transparent box or frame, and one that they cannot penetrate, which we call an opaque box or simply box. The frames are used to indicate the scope and binding of quantifiers, and the boxes to show the borders of non-atomic entities.

A general principle in reading statements in the full language is that they are read from the outside in (with respect to the various boxes described below), where the order between components at the same level of nesting is immaterial.

2.2.1 Quantifiers

Quantifiers have the usual meaning of first order predicate logic (FOPL), and can be implicit or explicit in a V statement. Only existential quantification can be implicit. Any entity that is not explicitly quantified is implicitly existentially quantified at the outermost level.

V currently supports the full range of quantification of FOPL, i.e. arbitrarily deeply nested universal and existential quantification with scope.

2.2.1.1 Syntax

The notation for explicit quantification is to surround the quantified entity or entities with a named (e.g. \( \forall \) and \( \exists \), or forall and exists) transparent frame (for an example, see fig. 8). A differently quantified set of entities within the scope of a separate quantifier is notated by putting the frame representing the innermost

---

1. Neither of these have any real meaning in themselves, only together with the other components of the box or frame.
quantification visually inside the outer one.

As we did not realise the necessity of event variables until quite late in the project, the language currently does not support the quantification of events. One possible syntax would be to simply treat the relation diamonds in the same way as the entity boxes, but we suspect that this would magnify the cognitive complexity of the language to a very large degree.

2.2.1.2 Semantics

Entities are quantified by the closest surrounding quantification frame, and within the scope of any frames that surround the innermost one.

Composite entities (section 2.2.2) are opaque to quantifiers, i.e. only the projected subset is quantified, not any part inside the entity box.

2.2.2 Composite entities

In general, a composite entity is a subset of the universe of discourse that cannot be defined simply as the objects satisfying a single unary predicate. There are two types of such composite entities:

- Members of an entity class that satisfy a set of relations involving other entities
- The result of a set operation on a number of atomic entities

In both cases these composite entities can be seen equally as a way of formulating subqueries, a way of enabling the definition of macros to build a library of commonly used (sub)queries, or as a way of specifying complex constraints on the participating entities.

2.2.2.1 Constraints

The V syntax for constraints is to surround the subquery that specifies the constraint with an unnamed opaque box, where the box has the same role in the surrounding query as an atomic entity box has.

We have as yet not decided on a definitive syntax for which set of constituent entities are principal, i.e. projected to the surrounding query. Ideally the syntax would be consistent with the one used for marking objects of interest (which indicates projection to the top query level), but we have not been able to find such a convention that would extend to an arbitrary number of nested compositions; pro tempore we have chosen to notate one-level projection with a line connecting the principal to the surrounding box (for an example, see fig. 9).

An interesting alternative, that probably would be visually clearer, would be to use a pseudo-three-dimensional perspective in the queries, where composite enti-
Figure 9. A query with a composite constraint entity: “List the queens, such that all the persons they have been married to are sons of queens.”
This query also illustrates the repeated use of the same entity icon, which means that the corresponding instances belong to the same set, but are not necessarily the same.
\[
\{Q | \forall T \in \{Q', P\}, \exists P'' \in \{P'' | \text{person}(P'')\},
\exists Q'' \in \{Q'' | \text{queen}(Q'')\},
\text{married}(Q'', P'') \land Q''=Q'' \land P''=P''\}.
\exists D \in \{D' | \text{queen}(D')\},
\text{son_of}(T[2], D) \land Q=T[1]\}
\]

Entities are shown on a plane different from the main query, and their respective principals connected to boxes, which represent the result of the subquery, in the parent query. This would also have the added advantage of providing a perspicuous notation for multiple separate projections from the composite entity, allowing for example queries such as “List all queens, such that all the persons they have been married to are related to them.”

The interpretation of these constraint based composite entities is the set of entities, specified by one-level projection markers, that satisfy the statement contained in the box.

2.2.2.2 Set operations
The second way of specifying composite entities is to perform set operations on a set of preexisting entities. Due to the semantics of entities, this is essential to be able to form conjunctions and disjunctions of atomic constraints, and useful, though not necessary, for composite entities. V therefore contains the basic binary set operations, intersection and union. The syntax for these operations (cf. fig. 10) is to surround the participating entities with named opaque boxes (e.g. ∩ and ∪, or entity-and and entity-or).

2.2.3 Logical connectives
The logical connectives are the ways of composing the body of the statements, viz. negation, conjunction, and disjunction.

2.2.3.1 Negation
The full language uses the same syntax for negating entities and relations as does the base language:

- If an entity (atomic or composite) symbol stands for the constraint \( \varphi(X) \), then the crossed out symbol stands for the constraint \( \neg \varphi(X) \).
- If a relation symbol stands for \( \varphi(x_1, \ldots, x_n) \), then the crossed out symbol means \( \neg \varphi(x_1, \ldots, x_n) \).

The notation for the negation of a quantifier, intended to be consistent with the
Figure 10. Set intersection: "List those that are both kings and employees"
\[ K \subseteq \{ X \text{king}(X) \} \cap \{ Y \text{employee}(Y) \} \]
or equivalently
\[ K \text{king}(K) \land \text{employee}(K) \]

notation for other negations, is a cross over the quantifier frame (cf. fig. 11).

Figure 11. Negation of quantifier: "Are there no kings?"
\[ \neg \exists K e \{ K' \text{king}(K') \} \]

2.2.3.2 Conjunction and disjunction

Conjunction is implicit in queries, but when necessary (i.e. within a disjunction or when negated), it is notated with a named (e.g. \& or and) transparent frame (cf. fig. 12). The same syntax is used for disjunction, though with a different name (e.g. \lor or or). The composition principles for these frames are the same as for all other transparent frames (cf. paragraph 2 of section 2.2 on page 10). As usual, conjunctions and disjunctions act only on the "visible" relations of its contents, not on the internals of opaque boxes, and is opaque to other connectives.

2.3 Simplified language

The simplified language originated in the full language. The reason for its existence is that the hierarchical box notation of the full language can become cognitively very complex, and thus difficult to read, although this would normally only reflect the complexity of the query's intended meaning. Because of this, we decided to introduce two simplifications that do away with most of the boxes, so that simpler queries need not be so complex. However, as a consequence of our choice of simplifications, the power of the simplified language is less than that of the full language, so the former cannot completely replace the latter, as certain statements cannot be made in the simplified language (vide infra). This has been a deliberate design decision, as we found it impossible to keep the full power with any form of simplified syntax we could devise. N.B. that the simplifications are intended to be independent of each other; the language does not require the use of
both together.

2.3.1 Quantifiers

The main simplification consists of removing the quantifier frames. Instead of surrounding the quantified entities with a named frame, in the simplified language a quantifier is placed inside the quantified entity, attached to a relation arc. The subtree originating at this attachment point forms the scope of the quantification (cf. fig. 13). As before, any entities that are not explicitly quantified are implicitly existentially quantified. Negation of the quantifier is notated as in figure 11.

In the simplified language, making a quantification explicit has the additional role of precedence resolution: An explicit quantification is always outside any implicit quantifications. In addition, an explicit existential quantification raises that quantifier above the closest universal quantification.

The main consequence of this simplification is that the power of the language, as compared to the full language, is reduced in that arbitrarily complex quantifications no longer are possible. As an example, $\exists X \forall Y \exists Z. q(X,Z) \land p(X,Y)$ can be
expressed (see fig. 15), but $\exists X \forall Y \forall Z. q(X,Z) \land p(X,Y) \land r(W,Y)$ cannot.

### 2.3.2 Composite entities

The second simplification consists of removing the boxes for composite constraint entities. The main use of composite constraint entities is when they are explicitly quantified (especially universally), so only those composite constraints are retained. The syntax is the same as for simplified quantification, but the quantifier is placed outside the principal atomic entity of the composition (for an example, see fig. 15).

The interpretation of this notation is that the entity to which the quantifier is attached is the principal entity of a constraint composition, the constituents of which are the remaining subgraphs connected to the principal entity.

### 2.4 Extensions

This section contains some possible extensions to the language that we have considered, but not investigated closely. These extensions are higher-order relations and generalised quantifiers.

#### 2.4.1 Higher-order relations

Higher-order relations are necessary to express such concepts as the number of objects in an entity class and quantification over predicates and entity properties.

In a practical application it would probably be difficult to do without the ability to count the number of objects that satisfy the constraints of a query (e.g. "How many objects has X sold?")}, so we have designed a preliminary notation for this:

In the full language a higher-order relation is notated by connecting the appro-
Figure 16. Simplified composite constraint entities: "List the queens, such that all the persons they have been married to are sons of queens."

\[ \{ Q \mid \forall T \in \{ <Q', P'> \mid \exists P'' \in \{ P'' \mid \text{person}(P'') \} \}.
\exists Q'' \in \{ Q'' \mid \text{queen}(Q'') \}.
\exists Q^* \in \{ Q'' \mid \text{married}(Q^*, P') \land Q=Q'' \land P'=P'' \}.
\exists D \in \{ D' \mid \text{queen}(D') \}.
\text{son_of}(T[2], D) \land Q=T[1]\} \]

appropriate relation arc(s) not directly to the entity, but to a circle surrounding it. This circle would mean that the relation is not to the members of the entity set, but rather to the set itself.

In the simplified language the notation is functional, in that the corresponding function name is syntactically used in the same way as a quantifier. Quantification of an atomic entity corresponds to application of the function to that atomic entity set, and the value of the function application participates in the relation to which the name of the function is attached. One drawback of the functional notation, is that the result of a function application cannot itself be quantified.

In the same way, quantification of a composite entity corresponds to application of the function to the projected set, and the value of the application participates in the relation attached to the function name. If an entity to which a function is applied has an interest marker, then it would be the result of the application that is of interest, and not the entity itself.

Quantifications over predicates and entity properties would be easily incorpo-rated into this framework: One would simply omit the relation or entity name. However, there are two problems with this notation:

- The location of the relation symbols relative to the quantifier borders would now matter in the full language, which might result in a language that is both visually more complex and in which it is more complicated to construct statements.
- Expressing that two entities are related to (possibly) different instances of the same unknown entity is problematical, and a question we have not as yet solved. An example of this kind of query would be "To what entities are both John and Tom related?"

The introduction of higher-order quantification also has an interesting pragmatic consequence: It provides a framework within the language itself for handling browsing/navigation of the database and/or database schema of the application. To find out, for example, in what relations a particular entity or entity instance participates one would pose a query like that of figure 17, which would be interpreted as \[ \{ x \mid \exists X \in \{ U \mid x(X) \land X=U \}. \exists z. \exists y. (x, y) \land z = \xi \}. \]
2.4.2 Generalised quantifiers

A generalised quantifier is (Alshawi et al., 1992) "... a relation \( Q \) between two sets \( A \) and \( B \) (where \( A \) is called the restriction set and \( B \) the body set) that satisfies some specific requirements." These requirements are that \( Q \) is "... insensitive to anything but the cardinalities of the sets \( A \) (the restriction set) and the set \( A \cap B \) (henceforth: the intersection set)." As an example, universal quantification corresponds to the generalised quantifier \( \lambda n. \lambda m. n=m \), where \( n \) is the cardinality of the quantified set, and \( m \) is the cardinality of the set of entities belonging to the quantified set that satisfy the remainder of the formula. Generalised quantifiers can be classified as absolute (those that only depend on the cardinality of the intersection set, e.g. "at least three") or relative (those that depend on both cardinalities).

Absolute generalised quantifiers would probably be easy to incorporate in the full language, as the restriction set corresponds to our entities and the intersection set to those of an entity that satisfy the surrounding query. All that would be necessary, would be to allow more than two kinds of quantifier names in the frames, e.g. also ranges such as \([1,7]\), meaning that there are at least one but at most seven members of the given entity. Relative quantifiers, however, (apart from universal quantification) are more problematical, as they require a way of referring to the cardinalities of the respective sets.

A consistent way of providing relative generalised quantifiers (if higher-order relations are available) would be to notate them through higher-order relations (a combination of the cardinality relation and arithmetic relations, or a simpler synonym) between the principal and the surrounding entity box.

Another possibility would be to provide conventional names for the respective cardinalities for use in composing the names of the quantifier frames. For example, the cardinality of the restriction set might be named \( R \), and that of the intersection set \( I \). Using this naming scheme, universal quantification would be named \( N=M \). This would probably be a more user friendly approach, but would complicate the processing of quantifiers substantially.

A third possibility may be to extrapolate from the universal and existential quantifiers, and simply provide a set of standard quantifiers with conventional names, such as forall, exists, some, most, etc., although this would severely curtail the power of the quantifiers.
In the simplified language incorporation of generalised quantifiers would be more difficult, as it would require some form of linear order on the relative precedence of the various quantifiers (which may be an infinitely large set). Some way of reversing the default precedence, that is more general than the current one, would probably also be needed. However, as the primary purpose of the simplified language is to provide a language for less complex statements and that is easy to read, it may not be necessary to include generalised quantifiers within it.

2.4.3 Adding declarative statements

It would be fairly easy, at least as far as the language itself is concerned, to add some features of a DBMS. By adding more modes than the two existing ones to the language, one could not only query the database, but also perform such operations as modifying the database or the database schema (imperative mode) by stating that what a visual statement means should hold. The main problem in practice would probably be that the expressiveness of V by far exceeds that of most databases.
3

4 The user interface

This section contains a descriptions of what we would like the actual bimodal interface to be ("Design Overview" on page 21 and "Functionality" on page 23) and of the actual "proof of concept" prototype ("The Implementation" on page 32).

4.1 Design Overview

In this section we outline the fundamentals behind the internal design of the system with which the user interacts, e.g. process communication. The main goal was to create a system that is as modular and as flexible as possible. The user should not be tied to one sort of configuration, but rather be able to set up his own system configuration through different resource files.

We designed the system as several separate programs: The user interface, a database and a natural language engine, i.e. the user interface and the applications. These programs may be very resource consuming, so some sort of distributed system is necessary to spread out the load to different machines available on the computer network. The communication between the programs (processes) occurs through separate channels for natural language, for expressions in an interface language, and for control instructions (e.g. a control instruction may be an instruction to switch the application between executing queries and echoing paraphrases in the alternate modality).

The information that the interface needs to communicate with the applications, such as which applications to start and on which machines, application specific interface components, and the mapping between $V$ objects and application objects, is intended to be specified in resource files that $V$ reads when started.

The translation between the command language of $V$ (i.e. the language that the user uses to communicate with the user interface program) and that of the application, and between the interface language of $V$ and that of the application, respectively, would preferably be achieved through separate, application specific translation modules (especially for application programs where the source code is not available to be modified). The interface language of $V$ is the logic representation of the visual query.

When evaluating a query, be it a natural language query or a visual query, the query is first translated to a logical form representation. This logical form may differ between the different applications so a translation between them is necessary, i.e. when paraphrasing a natural language query as a visual query or evaluating a visual query as a database query. This translation, however, should be quite simple, as it is between formal languages having well defined semantics. The only real problems would appear in translating concepts that do not exist in the target language; in these cases, a design decision would have to be made as to what translation would introduce the least semantic distortion. The different translation directions and translation states are visualised in figure 18.

There are three ways of displaying the answer from the database: As tables, paraphrasing it as a natural language sentence, or as a visual answer. Which of these ways of displaying answers to use should probably be decided by the user; but how to handle visual answers has not been considered within this project.

The discourse memory shown in figure 18 is shared between the NLE and the DMI, the reason being to make cross-modal references possible. With this kind of cross-reference, as described by Cohen et al. (1989), we could refer to graphical objects within a natural language query by pointing at that object with a pointing
Figure 18. Overview of the bimodal interface.

device.

This high-level design as a distributed and modular system would make it fairly easy to change components of the system. Presumably one would mostly want to
change the back-end component (here the database), but changing the NLP component would be equally easy.

4.2 Functionality

A direct manipulation language will only be interesting if it is easy to handle and consistent. This is of paramount importance, and further explained by Payne and Green (1986) (see also "Introduction", page 2).

One of the main advantages of the DMI described here is that when creating queries, all the objects of the domain are given by the interface, e.g. if the back-end application does not know anything about cars, the interface makes it impossible for the user to ask about cars, since no icon by that name exists. A specific problem which this approach causes, is navigation among a large set of icons and relations, but for a partial solution see below. Which objects exist in the DMI should be governed by the natural language engine (NLE) and the database, as we have no framework for handling hypothetical objects.

It is possible to have different domains for different applications, for example:

- \text{domain(NLE)} = \text{domain(DMI)} = \text{domain(database)}

  this is the ideal case, when the domains of the three applications are the same. We can only ask about the things that really are in the database, no more, no less.

- \text{domain(NLE)} = \text{domain(DMI)} \supset \text{domain(database)}

  possible when having the same domains for several database applications. Makes it possible to ask for things not stored in the current database, in which case the user should be informed.

- \text{domain(NLE)} = \text{domain(DMI)} \subset \text{domain(database)}

  not acceptable, we are not able to ask about all the objects handled by the database.

These examples are not all the examples possible, but they illustrate the importance of choosing and constructing the domain of the DMI.

There must be a way for the DMI to get the information about the domains of the NLE and the database. In our system this is done by resource files constructed by an administrator, containing the relation names used by each application together with the corresponding DMI name and default arguments, and the application object names and their corresponding DMI names. See "Description of Resource Files" on page 49 for resource syntax. The best solution, however, would be to use the respective conceptual schema directly, if possible. This would eliminate the problem of constructing proper domains.

From each resource object description file the interface creates a tree of the database objects, which is a visualisation of the object hierarchy, i.e. an object closer to the root is more general than an object deeper in the tree. In other words, the most specific objects are in the leaves of the tree (see fig. 19). This gives us a suitable structure for handling the objects and their menus when the size of the menus grows. The tree constructed is a visualisation of part of the conceptual schema of the domain and is explained in more detail by Bretan (1990).

Each relation description in the resource file has an entry for specifying default arguments, e.g. the relation married is always a relation between two persons, thus two person objects are default objects to the relation married.

Before we start describing the pragmatic aspects of the language, we have to introduce a few concepts used later.
4.2.1 Static vs dynamic syntax

In a DMI there are two different kinds of syntax, static and dynamic, where static syntax can be regarded as the state of the query and the dynamic syntax is the transition between two of these states (describing the "task language" of Payne & Green, 1986). In other words, the static syntax describes what a query looks like, and the dynamic syntax describes how queries may be modified.

The static and dynamic syntax are always coupled, but they can be tightly or loosely coupled. The difference between these two is quite important. In a language with tightly coupled syntax, all the individual transitions correspond directly to mouse events, since the query composition reflects the static syntax of the query. In a loosely coupled system a mouse event may move the visual query through several states, i.e. we have the freedom to let the dynamic syntax reflect the semantics of the language in a more conceptual way.

4.2.2 The query board

The query board, or drawing board, is a window or part of a window on which the user is supposed to compose a query. The query board can be compared to a drawing application, the user is 'drawing' a query with drawing tools, though the drawing tools of the query board are more complex than the tools of a typical drawing application.

4.2.3 An informal description of dynamic syntax and semantics

A query consists of entities (icons) and relations between entities. One of the most important practical questions is how to create icons and relations in the simplest and most efficient way.

4.2.3.1 Creating entities

Since the interface is a direct manipulation interface, creation is done by selecting entities and relations from nested menus, one for entities and one for relations. To create an entity, the user presses a mouse button where the object is to appear, selects 'Create entity' from the popup menu that appears, and chooses the desired entity by selecting it from the entity submenu.

The number of objects the interface must be able to handle may be large and therefore we have to have a way to support efficient user navigation among the objects. This is done as described above, by designing the menus to mirror the object hierarchy and by letting the user traverse the menu tree when creating the
queries.

4.2.3.2 Creating relations

When creating a relation by using the relation menu, three objects will appear on the board (for binary relations): Two icons (representing the default argument entities) and one diamond. The diamond is placed between the two icons and is connected to them by two arcs. The relations are selected from the relation menu; if the married relation is selected, a married diamond will appear. The entities of the relation are the default entities described in the resource files.

The second way to create a relation is when there already are two entities on the screen. By dragging the mouse cursor (with some combination of mouse buttons pressed) between two entities X and Y, the generic relation r(X, Y) is created. The diamond created has no name and needs to be changed.

4.2.3.3 Changing entities and relations

Changing the name/type of an entity or relation is done by first selecting the object with a mouse click. Then the “change object” menu is opened by pressing the proper mouse button, (The menu should look the same as the “create object” menu.) After selecting the required object from the menu, the new object will replace the old one.

4.2.3.4 Different box drawing modes - the toolbox

When dragging the mouse cursor over the drawing board with the proper button pressed, a box will appear. This box means different things at different times, sometimes we want to create a universal quantifier box and at other times we want to select items. Thus we need some way to differentiate between the different actions available. One way to make it possible to change the interpretation of mouse events is to use a toolbox (see fig. 20) to indicate different modes.

The toolbox is a window containing buttons, one button for each mode accessible. The selected button indicates the active mode of the drawing board, i.e. how mouse events are interpreted as actions.

To minimise the size of the toolbox, the number of buttons should be as low as possible. One way to achieve this is to have one button type for several functions. An example: AND and OR are two different buttons, but they share the same position in the toolbox. The AND button is shown in the toolbox. Selecting a button is done with the left mouse button and changing a toolbox button is done with the right mouse button. If the user wants the OR button he first has to change the button in the AND button position with the right mouse button. After this he can select the OR button from the tool box which now is visible.

4.2.3.5 Selecting objects

There are two ways of selecting objects, by clicking with the mouse on an object or by surrounding several objects with a box in select mode (the select button in the toolbox is selected). After selecting one or more objects, the user may apply actions to them, such as renaming or deleting.

4.2.3.6 Cutting and pasting

After having selected the objects, the user may want to invoke the editing actions, e.g. cutting or pasting. This is done by choosing such an operation from the edit menu. If an entity inside a relation is cut, the relation will be removed too, since a relation must not lack any arguments. The relation will not be moved to the clipboard, however. A selected and cut object will be moved to the clipboard, and can be viewed by invoking a ‘view clipboard’ command from the macro menu,
which will open a new window on the screen containing the last objects moved from the drawing board. Pasting is a bit more difficult to handle, since the spatial information on how to connect the removed objects to the rest of the query will be lost. The simplest solution is to paste the objects to the drawing board and then let the user rearrange their positions.

4.2.3.7 Creating macros - saving queries

Creating macros is handled much the same way as cutting objects: A query or subquery is selected and a 'cut to macro' or 'copy to macro' function is invoked. The macro is saved with a name specified by the user to a user specific macro resource file, and can subsequently be accessed by the user through the macro menu under the given name. The user defined name of the macro is also used when using the macro as a supericon (see “Composite entities” on page 27 and “Reducing and enlarging icons and relations” on page 28). The user should be able to select a bitmap for the supericon too. This could be done by selecting a pre-defined bitmap from a bitmap menu specified by the V administrator or by using an integrated bitmap editor (See also “Further work on the implementation” on page 36). A 'delete-macro' function (also found in the macro menu), removes the macro from the menu and the user specific macro resource file.

4.2.3.8 Quantifying entities

When quantifying entities we have to distinguish between quantifying in the simplified language and in the full language, as the syntax differs between them.

4.2.3.9 Quantifying in the full language

In the full language quantifying is done in the same way as selecting, but with the drawing board in quantifying mode, i.e. a quantifier button in the toolbox is selected (cf. fig. 20). There are (at least) two quantifying modes, existential and universal, to choose between. It would also be possible to have a mode for generalised quantifiers, (see “Generalised quantifiers” on page 18). After selecting the
objects to be quantified, a box is created, surrounding the objects and indicating both the objects quantified and the scope of the quantifier (being those entities within the box that were not the ones to be quantified. Objects in the same proximate quantifier box are on the same quantifying level, i.e. no quantified object is in the scope of the other.

4.2.3.10 Quantifying in the simplified language

Quantifying in this level can be done in two different ways, depending on whether we have a loosely or tightly coupled static and dynamic syntax.

- Quantifying with loosely coupled syntax
  In a loosely coupled system, the actions reflect the semantics of the language rather than the static syntax of the query (see “Static vs dynamic syntax” on page 23). Since an action with loosely coupled syntax is governed by the semantics, the action may have indirect consequences on other objects not directly subjected to the invoked action. The dynamic syntax is as follows: First, the type of quantifier is chosen from the quantifier popup menu of the icon to be quantified. Then the entities that will be in the scope of the new quantifier are selected. The event is terminated by pressing the right mouse button. The system will automatically place any necessary quantifiers in the entities that are not within the scope of the new quantifier, ‘lifting’ them above that quantifier.

- Quantifying with tightly coupled syntax
  Since the loosely coupled syntax is more complex to implement, we give an example of quantifying with tightly coupled syntax. Here the user has to explicitly manage the details of scope resolution herself, placing any explicit quantifiers necessary to make exceptions to the default scoping. This places a higher cognitive load on the user, as he has to make inferences about the consequences of the scoping rules before being able to decide where to place these quantifiers. The user chooses a quantifier from the quantifier popup menu of the icon. The relation attachment point of the entity closest to the pointer determines the direction of the quantifier, i.e. if we open the quantifier menu with the mouse pointer on the left side of an entity, the quantifier will be placed on the left side of the entity with scoping direction to the left (cf. 2.3.1 on page 15).

4.2.3.11 Composite entities

There are many possible ways to handle composite constraint entities in an actual interface. We present two different ways of creating subqueries, or composite constraint entities (or simply composite entities) as we call them. As in quantifying, we have to distinguish between creating composite entities in the full and in the simplified language.

- Composite entities with opaque boxes (full syntax).
  Composite constraint entities with opaque boxes are subqueries separated from the rest of the query by a solid box, i.e. a relation connection can not cross the border of the box. First the principal entity is chosen by getting the “composite entities” item from the popup menu of the entity. An indication, (see “Constraints” on page 12), will appear on the icon to show that this entity will be projected to the rest of the query and the drawing board automatically switches to “composite entity” mode. The subquery is constructed by drawing a box around the relations (with entities) to be in the composite entity (the subquery). To quantify the projected entities, a quantifier box has to be created, enclosing the projection, as described in “Selecting objects” on page 26.
• Composite entities without boxes (simplified syntax)

In the simplified language, composition is very easy to specify. The user chooses the projection item from the popup menu of the icon and selects a quantifier. The selected quantifier will appear on the relation connection between the icon and the relation symbol (the diamond).

These are just two possible simplifications, and are not intended preclude any others being introduced. It should also be possible to mix simplifications according to the needs of each user.

4.2.3.12 Negation

Negation as an action is a relatively straight-forward operation. Syntactically there are three things that can be negated: Entities, relations and quantifiers. Negation of entities and relations is handled the same way in both the full and simplified language. The relation is negated by choosing the negation item from the relation popup menu and the cross appears to indicate the action done (and of course that the relation is negated). When negating an entity the entity popup menu is opened by pressing a mouse button, the “negate” menu item is selected. A cross covers the bitmap to indicate a negation.

The negation of the quantifiers differs a bit between the full and simplified language, and therefore a separate description is given for them.

• Negation of quantifier in the simplified language

Negating the quantifier is similar to negate entity action. By positioning the mouse pointer over the quantifier to be negated and pressing the mouse button, the quantifier menu shows up. From the quantifier menu the ‘negate quantifier’ is chosen. The cross covers the quantifier to indicate the result and semantics.

• Negation of quantifier in the full language

Negating a quantifier placed in a frame is done in the same way as negating a quantifier placed on an icon, but with the slight difference that the action of negating is performed on the frame rather than the entity. The quantifier is negated by a single click (with the mouse button) on the quantifier name in the frame.

4.2.3.13 Reducing and enlarging icons and relations

Icons and relations are fairly large objects. They contain a lot of information and are thus quite complex, especially the entity icon. When a query grows, the drawing board gets too small to contain the whole query. This implies that a reduction function is needed. By reduction function we mean a function (or functions) that reduces the size of the objects on the query board. An enlarge function is thus the reverse function, a function that magnifies the objects.

Reduction could be performed in many different ways and ought to be specified, not by the constructor of the interface, but by the user. The interface should provide several reduction methods and let the user decide what he finds the best way to reduce the size of icons and relations. The user selects the objects to be reduced by enclosing them with a box while the drawing board is in reduction mode. The user should also be able to directly reduce the size of an object by using the popup menu of that object. The specific form of reduction is determined by the user and could be one of the following:

• Only show the name of the object.

Perfect to use on relations, the font size should be changeable. See figure 21(ii).
- Only show the bitmap of the icon. There should be different sizes of the bitmap too, but the user has to be able to distinguish it from other entities. See figure 21(iii).
- Only show text and bitmap. When this reduction function is invoked only the name and bitmap of the icon will be shown, not the instantiation and the quantifier items. See figure 21(iv)
- Reduces the size of the whole icon. The icon in miniature, a small bitmap and small font size. The icon may become too minute, difficult for the user to perceive. See figure 21(v).

![Diagram](image)

**Figure 21.** (i) is the original query, (ii) - (v) are different reduction methods applied to the original query.

Another important reduction is reduction of the composite entities to supericons. To be able to do this the user must have access to a large library of icons and/or rather, an icon editor, because this feature would be pointless if the icon representing the composite entity so badly matched the essence of the subquery that no one would understand it.
Figure 22. This figure shows how the subquery of figure 9 on page 13 can be reduced to a supericon. The person icon is put outermost to indicate that it is the principal entity and projected to the rest of the query.

As shown in figure 22, it may be possible to generate the supericons from the subquery itself by putting the icons from the composite entity together into the super icon. How this should be performed is not obvious, but the interface should provide the user with a tool to do this “super icon composition”.

4.2.3.14 Hierarchical history tree of queries

When querying the database, one often wants to reuse an earlier query or modify an earlier query. As described in Cohen et al. (1989), we introduce a hierarchical history tree of queries. Every time the user evaluates a query, the query will be saved as a node in the tree. If the query has been constructed from an empty query board, the node containing the query becomes the root of a new tree. On the other hand, if it is a modification of a previous query, the node becomes a child of the original query node. By selecting the “history tree” menu item from the edit menu a hierarchical history window opens containing the latest modified state and a tree (cf. fig. 23). The window consists mainly of two items, a tree window and a query window. The tree window contains a visualisation of the hierarchy tree with the current node of the history tree indicated. By using the mouse to select nodes in the tree or keyboard arrows, the user easily steps through the history tree. The query window to the right of the tree window contains the selected query. When she finds a suitable starting position for making a new query, she pushes the insert button and the query from the hierarchy tree will replace the query on the query board. Evaluating this query after modification, creates a new subtree in the hierarchy tree, with the old node as parent and the new node as child.
Figure 23. The hierarchical history tree. To the left the visualisation of the tree is shown. To the right is the query of the node indicated in the tree. The user selects a node with the mouse by clicking a mouse button on a node.

4.2.3.15 Inheritance hierarchy for objects

According Cohen et al. (1989), a major disadvantage of DMI's is that their hierarchical menus get unwieldy to navigate when large. In part due to this argument, the interface would, in addition to such menus, have a visual representation of the inheritance hierarchy of objects (cf. fig. 24), as follows:

By showing the visualisation of the object type hierarchy tree in a window, it is possible for the user to select the desired entity directly from this window. The visualisation of the inheritance hierarchy thus is an alternative to the "create entity" menu. An interface should provide support for both the menu and the window.
Figure 24. An example of a visual representation of the inheritance hierarchy of some objects. In this tree, both the name and a bitmap constitutes the node. This should be an option, since when the tree grows, it becomes important to reduce the size of the tree. The selected entity is indicated with a thick border. By pressing the "insert" button (or the "replace" button when changing an entity), the entity is inserted into the query at the requested position.

4.3 The Implementation

As implementation languages we chose SICStus Prolog (Andersson et al., 1993), and XPCE (Wiemaker & Anjewierden, 1992a & 1992b, Wielemaker, 1992) which is an object-oriented language running under Prolog (or LISP) that has predefined classes for graphics handling, process handling, chains, text editors, etc. During the implementation process we ran into several problems since the Public Domain version of XPCE we had access to contained several minor bugs and it was poorly documented. The biggest problem, though, was that XPCE is not logic programming. The variables of XPCE and Prolog look the same but are very different. There are no logic variables in XPCE, they are all global pointers to (graphical) objects, etc.

One of the ideas behind this project was to use the CLARE system (SRI, 1992; Rayner, 1993) as the natural language engine component. Unfortunately we did not get access to the CLARE system in time for this report, thus only simulations of the real system have been used.

4.3.1 Design Realisation

We will here describe some of the realisation problems that occurred during the implementation, the implementation decisions made, and the features missing from the interface.

One of the main mistakes was to use TRL as an interface language for V, instead
of using a separate interface language to represent the visual query. The CLARE system has two different interface languages, TRL and LF. We chose TRL since we thought the semantics of the two were equivalent, but the TRL had nicer syntax. The choice of TRL as interface language was a mistake since it does not have a natural way to express constraints, i.e. our entities. (For an overview of the translations required of the system, see fig. 25).

![Diagram of the CLARE system]

**Figure 25.** Overview of the translation in the implemented interface.

The fact that the database is built into the CLARE system eliminated the communication between our program and the database. The CLARE system contains a TRL to SQL application which we intended to use when querying the database. The graphical interface, the CLARE system and the database are all computing intensive processes, so to be able to run the system with tolerable response times,
a kind of distributed system was necessary. The communication between the
processes is done by using sockets. At first we used SICStus sockets, but since
they interfered with XPCE we changed to XPCE sockets. The late discovery of the
incompatibilities between SICStus and XPCE forced us to use a single bidirec-
tional channel for all communication.

Another problem was the realisation of the icons. In the simplified language we
place the quantifiers in the icons. The problem includes finding a good placement
within the icons for these quantifiers, how small can we make them, etc. This is
an optimization problem for which the solution is not obvious; the prototype
shows one solution of this problem. In the prototype it is only possible to connect
relations to the icon at two points, one on the left side and one on the right side. A
box on either side indicates where to place the quantifier and its direction.

The quantifiers available are universal and existential quantifiers, but adding
more should not cause any trouble.

System entities, such as date and time entities, are not implemented, and nei-
ther are super icons and macros. Only one level of composite entities is imple-
mented.

4.3.2 The interface

The graphical part of the interface consists of four different tiles, each with a
different function (see appendix, fig 30). Uppermost is a tile containing the menu
bar. Under that there is the button tile and the query board. The button tile is
placed to the left of the query board. Beneath both these tiles there is a text tile
containing an emacs editor.

4.3.2.1 The menu bar

The menu bar is the Macintosh-like menu bar found in many direct manipula-
tion languages.

| File | Edit | Macro |

Figure 26. The main menu bar

The file menu currently contains an information item and a close item. The edit
menu holds items for cutting and pasting. The macro menu is supposed to contain
menu items for macro handling, such as creating macros and listing available
macros, but these are not implemented yet. Each entry in the menus has a key
command (short cut) for fast access to an operation.

4.3.2.2 The button tile

To the left of the query board, under the menu bar, is the button window. This
tile contains the user defined buttons, specified in a resource file (see “Application
Resource” on page 49). The user defined buttons are intended to open user
defined windows to control applications, such as the database or the NLE. It is
also possible to bind user defined actions to these buttons.

The tile also contains three default buttons, one clear button to clear the query
board from relations and entities, one tool button to open the toolbox window, and
one evaluate button to evaluate the visual query.

4.3.2.3 The realisation of the query board

As the implementation is merely a proof of concept prototype, the interface is
not fully implemented. Many of the features described in “An informal description of dynamic syntax and semantics” on page 24 are missing, such as the hierarchy tree and the ability to reduce and enlarge icons and relations. The possibility to store queries as macros and using the macros as super icons is not implemented either and no part of the macro handling is implemented.

4.3.2.4 The text tile

At the bottom of the interface window, we have put an Emacs compatible text editor for natural language queries. In this editor it is possible to enter and evaluate natural language queries.

The query history is automatically given by the editor history and a query can be re-evaluated by placing the cursor on the same line as a previous query (it is also possible to edit a previously queried query) and pressing return. The query is copied and placed last in the text buffer and then evaluated.

4.3.3 Translation to TRL

We will not delve into implementation details, just outline some basic implementation concepts and give some illustrative examples of translation. As mentioned in “Design Realisation” on page 32, we use TRL as the interface language to represent the visual queries.

When translating the visual query, there are three different stages: (1) identifying the relations and entities, (2) identifying the composite entities (if any), and finally (3) resolving the scope of the quantifiers present. Stage (1) is trivial. Stages (2) and (3) are more complex since the problem involves processing general graphs.

The basic idea behind stage (2) is to scan the graph to identify the composite entities, constructing a forest of the relations of the query. We define the forest as a set of triples, \( \text{Forest} = \{<\text{Node}_1, \text{Child}_{11}, \text{Child}_{12}>, ..., <\text{Node}_n, \text{Child}_{n1}, \text{Child}_{n2}>\} \), where \( \text{Child}_j \) is a Forest. This means that Forest is a set of binary trees, where the nodes \( \text{Node}_1, ..., \text{Node}_n \) represent the relations that are at the same “level” of the query, i.e., those elements contained within the same proximate composition box. \( \text{Child}_j \) of \( \text{Node}_i \) represents a composite entity and is itself a Forest. For example, if \( \text{Child}_{12} \) is a non-empty set, this means that the second argument of the relation represented by \( \text{Node}_1 \) is a composite entity.

The third stage, resolving the scope of the quantifiers, creates a set of pairs, \( \{<X_1, Y_2>, ..., <X_n, Y_P>\} \). Where \( X \) is the quantified entity and \( Y \) is a set of entities such that they are in the scope of \( X \). From this set we reconstruct the scoping hierarchy of the variables, needed to construct the interface language expression corresponding to the visual query.

To illustrate, we give an example in the simplified language:

We have the following visual query, “Are all countries ruled by a king?”,
database query. First we have to translate the visual query to the interface language. This is done by evaluating the three steps, formulated above.

Step (1), identifying the entities and relations, gives the following result:

1. entities = \{king(X), country(Y)\}
   relations = \{rule\}

   Identifying composite constraint entities, step (2), results in:

2. tree = \{<rule, {}, {}>\}

   Since there are no composite entities in the query, the children of rule are the empty sets.

   Resolving the scope of the quantifiers, step (3):

3. Scope = \{<king(X), {}>, <country(Y), {}>, <king(X), {}>\}

   The universal quantifier of country covers the implicit existential quantifier of king.

   By processing the result of step (2) and (3) we can construct the resulting TRL:

4. \forall X, \forall Y, impl(country(X), \exists Y, and(king(Y), rule(Y, X)))

   When the visual query has been translated to the interface language, which in our case is the same as the interface language of the CLARE system, we send this expression to the CLARE system with either a paraphrase command (if we want to paraphrase the query as a natural language query) or an evaluate command (if we want to evaluate it as a database query).

4.4 Further work on the implementation

As the current implementation is merely a “proof of concept” prototype, there is quite a lot of work to be done on the actual program. The prototype could probably be used as a basis for at least a full prototype, but some major parts are missing. The main parts that need to be completed are:

- Interface language
  As mentioned previously, our choice of TRL as the interface language was not a felicitous choice, as its semantics do not correspond completely to those of V. It would probably be preferable to design our own interface language, corresponding more closely to the intended semantics of the language, thus localising any distortion of meaning that may occur to a separate translation module (between our interface language and that of the application(s)).

- Language constructions not implemented
  - Negation.
    No form of negation is implemented. Neither negation of entities, negation of relations, nor negation of quantifiers is implemented.
  - Interest marker
    The interest marker is not implemented, which means that there is no way of specifying wh-queries.
  - Generation of interface language representations of the visual statements and of visual statements from interface language representations. Generation of composite entities from expressions in the interface language is missing and some corrections to the translation of visual expressions need to be made.
  - Interface facilities not implemented
  - Macros
    What we call “macro handling” needs to be implemented. It seems to us
that the ability to save complex (sub)queries for reuse is of critical importance to the usability of the interface.

- History tree
  The evaluated queries are currently not saved and therefore it is impossible to retrieve an old query.

- Reducing and enlarging objects
  The feature of reducing the size of one or more objects, to be able to visualise large queries, is not implemented.

- Dynamic syntax
  The use of a dynamic syntax that is separate from the static one, as we have argued, is probably of great importance to the "user friendliness" of the language and would definitely make the process of transforming the visual statements of the simplified language into the corresponding statements in the interface language simpler.

- Process communication
  Due to the late discovery of severe incompatibilities between XPCE and SICStus Prolog socket handling, the communication model we had intended to use between V and the applications, where the different modalities (interface language, natural language, and control statements) use different channels, has not been implemented. As is, the communication is primitive, consisting of a single channel, but there may be better solutions than the one we designed, as we are not specialists in process communication.
Acknowledgements

This project has been financed by the Swedish Institute of Computer Science (SICS).

We would like to thank the Natural Language Processing group at SICS for a very congenial and supportive working environment, and especially our advisor Ivan Bretan, without whose support and penetrating discussions the language would have been a much inferior product.
References


A Additional query samples

The bitmaps are not included in these samples, as we have not had access to an extensive bitmap library.

Figure 27. "Who sold more than Mols?" One possible interpretation.

Figure 28. "What are the capitals of the countries which do not import any Volvo cars?"
B Formal syntax of the full and simplified language

The syntax of the languages is described using the relation grammar of Crimi et al. (1991). The production rules are a valiant on BNF notation, where each rule generates a set of new nodes of the parse tree and (possibly) a set of relations that holds between these nodes, i.e. inherited attributes. The evaluation rules use the relations/attributes generated by the production rules to synthesise attributes that are not simply inherited.

B1 Syntax for the full language

The grammar of V is a 6-tuple (V_{node}, V_{terminal}, V_{relation}, Q, P, R), where V_{node} = {Q, B, X, Y, G, C, D, E}, and V_{terminal} = {forall, exists, text, bitmap}, V_{relation} = {start, end, connected}, Q is the start node, P is the set of production rules and R is the set of evaluation rules.

The terminals that we use are abbreviations for graphical objects of the language; for example, the forall terminal together with the box terminal stands for the following graphical object

```
\forall
```

The bitmap terminal represents the graphical picture inside the icon.

B1.1 Production rules

\[ X ::= \{Y_1, \ldots, Y_n\} \{p_1(Y_1, \ldots, Y_i), \ldots, p_n(Y_k, \ldots, Y_1)\} \text{ is to be read as "the non-terminal } X \text{ generates the nodes } Y_1 \text{ to } Y_n \text{ and the relations } p_1 \text{ to } p_n \text{ between these."} \]

\[ Q ::= \{\forall\} \]

\[ Q ::= \{B, Y\} \text{ (connected}(B, Y)\} \]

\[ B ::= \{\text{QuantBox, Q}\} \]

\[ Y ::= \{\text{Connection, B, Y}\} \text{ (connected}(B, Y), \text{ connected}(B, \text{ Connection})) \]
\[ Y ::= \{ B, Y_1 \} \text{ connected}(B, Y_1) \]  
\[ Y ::= \{ X \} \text{ | } \{ \} \]  
\[ X ::= \{ \text{Connection}, G, X_1 \} \text{ end}(\text{Connection}, G) \lor \text{start}(\text{Connection}, G) \]  
\[ X ::= \{ \text{Connection}, G \} \text{ end}(\text{Connection}, G) \lor \text{start}(\text{Connection}, G) \]  
\[ G ::= \{ \text{Entity} \} \]  
\[ G ::= \{ \text{Entity}, \text{Connection}, G \} \text{ start}(\text{Connection}, \text{Entity}) \]  
\[ G ::= \{ E, G \} \text{ start}(E, G) \]  
\[ G ::= \{ C, G \} \text{ start}(C, G), \text{end}(C, G) \]  
\[ G ::= \{ D, G \} \text{ end}(D, G) \]  
\[ E ::= \{ \text{Connection}, \text{Entity} \} \text{ end}(\text{Connection}, \text{Entity}) \]  
\[ E ::= \{ C, \text{Entity}, E \} \text{ start}(E, \text{Entity}), \text{end}(C, \text{Entity}) \]  
\[ E ::= \{ E_1, E_2 \} \text{ start}(E_2, E_1) \]  
\[ C ::= \{ \text{Connection} \} \]  
\[ C ::= \{ C, \text{Entity}, \text{Connection} \} \text{ start}(\text{Connection}, \text{Entity}), \text{end}(C, \text{Entity}) \]  
\[ D ::= \{ G, \text{Connection} \} \text{ start}(\text{Connection}, G) \]  
\[ \text{QuantBox} ::= \{ \text{Box}, \text{Quantier} \} \]  
\[ \text{Box} ::= \text{box} \]  
\[ \text{Quantifier} ::= \text{forall} \text{ | } \text{exists} \]  
\[ \text{Entity} ::= \{ \text{Name}, \text{ID}, \text{Bitmap} \} \text{ | } \text{Projection} \]  
\[ \text{Projection} ::= \{ \text{Box}, Q \} \]  
\[ \text{Connection} ::= \{ \text{Arc}_1, \text{Diamond}, \text{Arc}_2 \} \text{ same_side}(\text{Arc}_1, \text{Arc}_2) \]  
\[ \text{Arc} ::= \{ \text{Arrow}, \text{Line} \} \]  
\[ \text{Arc} ::= \{ \text{Arrow}, \text{Line} \} \]  
\[ \text{Name} ::= \text{text} \]  
\[ \text{ID} ::= \text{text} \]  
\[ \text{Bitmap} ::= \text{bitmap} \]  
\[ \text{Arrow} ::= \text{left} \text{ | } \text{right} \]  

**B1.2 Evaluation rules**  
\[ P ::= (X \Rightarrow Y), Q_1, ..., Q_n \text{ means that if the node } X \text{ has generated the set of children } Y \text{ and the relations } Q_1, ..., Q_n \text{ hold true then } P \text{ is true.} \]  
\[ \text{end}(\text{Connection}, G) ::= (G \Rightarrow \{ C, G_1 \}), \text{end}(\text{Connection}, G) \]  
\[ \text{end}(\text{Connection}, G) ::= (G \Rightarrow \{ \text{Entity}, \text{Connection}_1, G_1 \}), \text{end}(\text{Connection}, \text{Entity}) \]  
\[ \text{end}(C, G) ::= (G \Rightarrow \{ \text{Entity} \}), \text{end}(C, \text{Entity}) \]  
\[ \text{end}(C, G) ::= (G \Rightarrow \{ C_1, G_1 \}), \text{end}(C, G_1) \]  
\[ \text{end}(C, G) ::= (G \Rightarrow \{ \text{Entity}, \text{Connection}, G_1 \}), \text{end}(C, G_1) \]  
\[ \text{end}(C, \text{Entity}) ::= (C \Rightarrow \{ \text{Connection} \}), \text{end}(\text{Connection}, \text{Entity}) \]  
\[ \text{end}(C, \text{Entity}) ::= (C \Rightarrow \{ C_1, \text{Entity}_1, \text{Connection} \}), \text{end}(\text{Connection}, \text{Entity}) \]  
\[ \text{start}(C, G) ::= (G \Rightarrow \{ \text{Entity}, \text{Connection}, G_1 \}), \text{start}(C, \text{Entity}) \]  
\[ \text{start}(C, G) ::= (G \Rightarrow \{ C_1, G_1 \}), \text{start}(C, G_1) \]  
\[ \text{start}(E, G) ::= (G \Rightarrow \{ \text{Entity}, \text{Connection}, G_1 \}), \text{start}(E, \text{Entity}) \]  
\[ \text{start}(E, G) ::= (G \Rightarrow \{ C, G_1 \}), \text{start}(E, G_1) \]  
\[ \text{start}(E_1, E_2) ::= (E_2 \Rightarrow \{ C, \text{Entity}, E_3 \}), \text{start}(E_1, \text{Entity}) \]  
\[ \text{start}(E_1, E_2) ::= (E_2 \Rightarrow \{ E_3, E_4 \}), \text{start}(E_1, E_3) \]  
\[ \text{start}(\text{Connection}, G) ::= (G \Rightarrow \{ \text{Entity} \}), \text{start}(\text{Connection}, \text{Entity}) \]
start(Connection, G) ::= (G ⇒ {Entity, Connection, G_1}), start(Connection, G_1)
start(Connection, G) ::= (G ⇒ {E, G_1}), start(Connection, G_1)
start(Connection, G) ::= (G ⇒ {C, G_1}), start(Connection, G_1)
start(E, Entity) ::= (E ⇒ {Connection, Entity}), start(Connection, Entity)
start(E, Entity) ::= (E ⇒ {E_1, E_2}), start(E_1, Entity)
start(E, Entity) ::= (E ⇒ {C, Entity_1, E_1}), start(C, Entity)
start(C, Entity) ::= (C ⇒ {Connection}), start(Connection, Entity)
start(C, Entity) ::= (C ⇒ {C_1, Entity, Connection}), start(C_1, Entity)

connected(B, Y) ::= (Y ⇒ {})
connected(B, Y) ::= (Y ⇒ {X}), connected(B, X)
connected(B, Y) ::= (Y ⇒ {Connection, B_1, Y_1}), (connected(B, Connection) ∧
connected(B_1, Connection) ∧ (connected(B, Y_1) ∨ connected(B_1, Y_1)))
connected(B, Y) ::= (Y ⇒ {B_1, Y_1}), (connected(B, Y_1) ∧ connected(B_1, Y_1))
connected(B, X) ::= (X ⇒ {Connection, G, X_1}), (connection(B, Connection) ∧
connected(B, X_1))
connected(B, X) ::= (X ⇒ {Connection, G}), connected(B, Connection)
connected(B, Connection) ::= (B ⇒ {Box, G}), connected(B, Connection)
connected(Q, Connection) ::= (Q ⇒ {G}), connected(G, Connection)
connected(Q, Connection) ::= (Q ⇒ {B, Y}), (connected(B, Connection) ∨ connected(Y, Connection))
connected(Y, Connection) ::= (Y ⇒ {X}), connected(X, Connection)
connected(Y, Connection) ::= (Y ⇒ {Connection1, B, Y_1}), (connected(B, Connection) ∨ connected(Y_1, Connection))
connected(Y, Connection) ::= (Y ⇒ {B, Y_1}), (connected(B, Connection) ∨ connected(Y_1, Connection))
connected(X, Connection) ::= (Y ⇒ {Connection1, G, X_1}), (connected(G, Connection) ∨ connected(X_1, Connection))
connected(X, Connection) ::= (Y ⇒ {Connection1, G}), connected(G, Connection)
connected(G, Connection) ::= (G ⇒ {Entity}), connected(Entity, Connection)
connected(G, Connection) ::= (G ⇒ {Entity, Connection1, G_1}), (connected(Entity, Connection) ∨ connected(G_1, Connection))
connected(G, Connection) ::= (G ⇒ {E, G_1}), (connected(E, Connection) ∨ connected(G_1, Connection))
connected(G, Connection) ::= (G ⇒ {C, G_1}), (connected(C, Connection) ∨ connected(G_1, Connection))
connected(G, Connection) ::= (G ⇒ {D, G_1}), (connected(D, Connection) ∨ connected(G_1, Connection))
connected(E, Connection) ::= (E ⇒ {Connection1, Entity}), connected(Entity, Connection)
connected(E, Connection) ::= (E ⇒ {C, Entity, E_1}), (connected(C, Connection) ∨ connected(Entity, Connection) ∨ connected(E_1, Connection))
connected(E, Connection) ::= (E ⇒ {E_1, E_2}), (connected(E_1, Connection) ∨ connected(E_2, Connection))
connected(C, Connection) ::= (C ⇒ {C_1, Entity, Connection1}), (connected(Entity, Connection) ∨ connected(C_1, Connection))
connected(D, Connection) ::= (D ⇒ {G, Connection1}), connected(G, Connection)
connected(Entity, Connection) ::= (end(Connection, Entity) ∨ start(Connection,
B2 A BNF for the simplified language

The relation grammar, as described by Crimi et al. (1991), for the simplified language, is a stripped down version of the grammar for the full language, since the spatial syntax of the simplified language is simpler to express formally.

The grammar of the simplified version of V is a 6-tuple \( (V_{\text{node}}, V_{\text{terminator}}, V_{\text{relation}}, G, P, R) \), where \( V_{\text{node}} = \{G, C, D, E\} \), \( V_{\text{terminator}} = \{\text{forall}, \text{exists}, \text{text}, \text{left, right, bitmap, none}\} \), \( V_{\text{relation}} = \{\text{start, end, same_quant, same_side, above}\} \).

- \( G \) is the start node, \( P \) is the set of production rules, and \( R \) is the set of evaluation rules.

B2.1 Production rules

\[
\begin{align*}
G &::= [\text{Entity}] \\
G &::= [\text{Entity, Connection, G}] \text{ (start(Connection, Entity))} \\
G &::= [E, G] \text{ (start}(E, G)) \\
G &::= [C, G] \text{ (start}(C, G), \text{ end}(C, G)) \\
G &::= [D, G] \text{ (end}(D, G)) \\
E &::= [\text{Connection, Entity}] \text{ (end}(\text{Connection, Entity})) \\
E &::= [C, \text{ Entity}, E] \text{ (start}(E, \text{ Entity}), \text{ end}(C, \text{ Entity})) \\
E &::= [E_1, E_2] \text{ (start}(E_2, E_1)) \\
C &::= [\text{Connection}] \\
C &::= [C, \text{ Entity, Connection}] \text{ (start}(\text{Connection, Entity}), \text{ end}(C, \text{ Entity})) \\
D &::= [\text{G}, \text{ Connection}] \text{ (start}(\text{Connection, G})) \\
\text{Entity} &::= [\text{None, Name, ID, Bitmap, None}] \\
\text{Entity} &::= [\text{None, Name, ID, Bitmap, QuantBoxRight}] \\
\text{Entity} &::= [\text{QuantBoxLeft, Name, ID, Bitmap, None}] \\
\text{Entity} &::= [\text{QuantBoxLeft, Name, ID, Bitmap, QuantBoxRight}] \text{ (same_quant(QuantBoxLeft, QuantBoxRight))} \\
\text{QuantBoxLeft} &::= [\text{Box, Quantifier}] \\
\text{QuantBoxRight} &::= [\text{Box, Quantifier}] \\
\text{Quantifier} &::= \text{forall} \mid \text{exists} \\
\text{Connection} &::= [\text{Arc}_1, \text{ Diamond, Arc}_2] \text{ (same_side(Arc}_1, \text{ Arc}_2)} \\
\text{Arc} &::= [\text{Arrow, Line, Projection}] \text{ (above(Projection, Line))} \\
\text{Arc} &::= [\text{Arrow, Line, Projection}] \text{ (above(Projection, Line))} \\
\text{Name} &::= \text{text} \\
\text{ID} &::= \text{text} \\
\text{Bitmap} &::= \text{bitmap} \\
\text{Projection} &::= \text{forall} \mid \text{none} \\
\text{Arrow} &::= \text{left} \mid \text{right}
\end{align*}
\]

B2.2 Evaluation rules

\[
\begin{align*}
\text{end(}\text{Connection, G}) &::= (G \Rightarrow \{C, G_1\}), \text{ end(}\text{Connection, G_1}) \\
\text{end(}\text{Connection, G}) &::= (G \Rightarrow \{\text{Entity, Connection1, G_1}\}), \text{ end(}\text{Connection, Entity}) \\
\text{end(}\text{C, G}) &::= (G \Rightarrow \{\text{Entity}\}), \text{ end(}\text{C, Entity}) \\
\text{end(}\text{C, G}) &::= (G \Rightarrow \{C_1, G_1\}), \text{ end(}\text{C, G_1}) \\
\text{end(}\text{C, G}) &::= (G \Rightarrow \{\text{Entity, Connection, G_1}\}), \text{ end(}\text{C, G_1})
\end{align*}
\]
end(C, Entity) :- (C \rightarrow (\text{Connection})), \text{end}(\text{Connection}, \text{Entity})
end(C, Entity) :- (C \rightarrow (C_1, \text{Entity}_1, \text{Connection})), \text{end}(\text{Connection}, \text{Entity})
start(C, G) :- (G \rightarrow (\text{Entity}, \text{Connection}, G_1)), \text{start}(C, \text{Entity})
start(C, G) :- (G \rightarrow (C_1, G_1)), \text{start}(C, G_1)
start(E, G) :- (G \rightarrow (\text{Entity}, \text{Connection}, G_1)), \text{start}(E, \text{Entity})
start(E, G) :- (G \rightarrow (C, G_1)), \text{start}(E, G_1)
start(E_1, E_2) :- (E_2 \rightarrow (C, \text{Entity}, E_3)), \text{start}(E_1, \text{Entity})
start(E_1, E_2) :- (E_2 \rightarrow (E_3, E_4)), \text{start}(E_1, E_3)
start(\text{Connection}, G) :- (G \rightarrow \{\text{Entity}\}), \text{start}(\text{Connection}, \text{Entity})
start(\text{Connection}, G) :- (G \rightarrow (\text{Entity}, \text{Connection}, G_1)), \text{start}(\text{Connection}, G_1)
start(\text{Connection}, G) :- (G \rightarrow (E, G_1)), \text{start}(\text{Connection}, G_1)
start(\text{Connection}, G) :- (G \rightarrow (C, G_1)), \text{start}(\text{Connection}, G_1)
start(E, \text{Entity}) :- (E \rightarrow \{\text{Connection}, \text{Entity}\}), \text{start}(\text{Connection}, \text{Entity})
start(E, \text{Entity}) :- (E \rightarrow \{E_1, E_2\}), \text{start}(E_1, \text{Entity})
start(E, \text{Entity}) :- (E \rightarrow \{C, \text{Entity}_1, E_1\}), \text{start}(C, \text{Entity})
start(C, \text{Entity}) :- (C \rightarrow (\text{Connection})), \text{start}(\text{Connection}, \text{Entity})
start(C, \text{Entity}) :- (C \rightarrow (C_1, \text{Entity}, \text{Connection})), \text{start}(C_1, \text{Entity})

\text{same\_quant}(\text{QuantBoxLeft}, \text{QuantBoxRight}) :- (\text{QuantBoxLeft} \rightarrow (\text{Quantifier}_1), \\
\text{QuantBoxRight} \rightarrow (\text{Quantifier}_2)), \text{Quantifier}_1 \equiv \text{Quantifier}_2

\text{same\_side}(\text{Arc}_1, \text{Arc}_2) :- (\text{Arc}_1 \rightarrow (\text{Arrow}_1, \text{Line}_1, \text{Projection}_2), \text{Arc}_2 \rightarrow (\text{Arrow}_2, \text{Line}_2, \text{Projection}_2)), \text{Arrow}_1 \equiv \text{Arrow}_2

\textbf{C Syntax of TRL}

\text{TRL\_formula} ::= \text{<term>} \mid \text{<abstract>} \mid \text{<form>}

\text{term} ::= \text{<variable>} \mid \text{<constant>} \mid \text{<function\_application>} \mid \text{<type>\#$<id>}

\text{type} ::= \text{<constant>}

\text{id} ::= \text{term}

\text{abstract} ::= \text{<variable>}'\land'\text{<abstract>}

\quad \text{between}

\quad \mid \text{<variable>}'\land'\text{<form>}

\quad \text{between}

\quad \text{mula}

\quad \text{in}

\text{form} ::= \text{and}(\text{<form>}, \text{<form>}) \mid \text{or}(\text{<form>}, \text{<form>}) \mid \text{not}(\text{<form>}) \mid \text{impl}(\text{<form>}, \text{<form>}) \mid \text{<form>} \leftarrow \text{<form>} \mid \text{<form>} \leftrightarrow \text{<form>}

\mid \text{exists}(\text{<variable>}, ..., \text{<variable>}, \text{<form>}) \mid \text{forall}(\text{<variable>}, ..., \text{<variable>}, \text{<form>})

A Prolog variable
A Prolog atom
A Prolog term
The object of type "type", whose database identifier is "id"
Denotes an n-ary relation R objects, such that R holds the objects A_1, ..., A_n iff the for-
resulting from substituting A_i for X_i
P holds
Conjunction
Disjunction
Negation
Implication
Equivalence
Existential quantification
Universal quantification

K. Saxin Hammarström & R. Nilsson
| cardinality(<term>, <variable>^<form>) Holds iff there are precisely <term> instantiations of <variable> such that <form> holds |
| sum(<term>, <variable>^<form>) Holds iff all objects A, such that P holds when A is substituted for X, are summable quantities, and <term> is their sum |
| order(<term>, <variable>^<variable>^<form>, <term>) Holds iff the second term represents an ordering relation, \( D_{\text{Max}} \) is the maximal \( D_1 \) under the relation represented by the second term, such that the form when \( D_1 \) is substituted for the second variable and some \( X_1 \) is substituted for the first variable, and the first term is such an \( X_1 \) |
| kw(<form>) Metaknowledge: Holds iff there is an effective translation of the form |
| def(<form>) Necessity: Holds iff the form is implied by the background theory without the database |

D Description of Resource Files

The resource files of V determine the user configurable aspects of the interface, such as what application(s) to run, which bitmaps to use for the icons, etc. All of them are plain text files that have to conform to certain grammars, described below. The three main ones are the V resource file, the application resource file, and the application definitions resource file. All of these must exist for the program to run.

D1 V Resources

This resource file describes which application to run and the name of its resource file. It must be named either "/.vresources" or its name given as the first command line argument when starting XPCE.

\(<v\_resource> ::= \{<application> <application\_resource\_file> <machine\_name>\} <application> ::= \{<name>\} <names ::= <name> \mid <name> <names> <name> ::= <string> <application\_resource\_file> ::= <string> <machine\_name> ::= <string> Provided\ string\ isn't\ \{' or '\}''

D2 Application Resource

This file describes the application specific components of the visual interface and the names of the macro file, the application definitions file, and the initialisation commands to send to the application.
To be able to refer to an item in several places:

(resource_items ::= resource_item resource_items | ε)

/resource_item ::= window <name> { <block> }

/block ::= graphical_group { resource_contents } <block> | ε

/resource_contents ::= <menubar> <items>
| <window_item> <resource_contents>
| ε

/items ::= <window_item> <items> | ε

/menubar ::= menubar { <pulldown_items> }

/window_item ::= <text_item>
| <check>
| <radio>
| <item>

/pulldown_items ::= <pulldown_item> <pulldown_items> | ε

/pulldown_item ::= <name> { <menu_items> }

/menu_items ::= <item> <menu_items>
| <pulldown_item>
| ε

/item ::= <command_spec>
| <window_switch> 

/text_item ::= text <name> <default_text> <command> <terminator>

/default_text ::= <string>

/command ::= <string>

/terminator ::= <string>

/check ::= <name> { <check_box> }

/check_box ::= <on_command> <off_command> <default_check_state>
| <check_box> | ε

/on_command ::= <command>

/command ::= <string>

/off_command ::= <command>

/default_check_state ::= <boolean>

/boolean ::= on | off

/radio ::= radio <name> { <radio_items> } <default_radio_state>

/radio_items ::= <command_spec> <radio_items> | ε

/command_spec ::= command <name> <command_list>
| <window_item> <resource_name>

/command_list ::= { <commands> }

/commands ::= ε | <command> <commands>

/resource_name ::= <name>

/default_radio_state ::= <name>
<r_file3> ::= <menubar> <r_file4> | <r_file4>  
<r_file4> ::= buttons { <items> }  
The buttons that will appear in the main window

D3 Application Definitions Resource

Describes the mapping between names in V and in the application, and the bit-maps to use for entities.

<definitions> ::= relations { <relations> } entities { <entities> }  
<relations> ::= <relation> <relations>  
 | <relation>  
<relation> ::= <V_name> <application_name> { <default_entities> }  
<V_name> ::= <non_reserved_name>  
<non_reserved_name> ::= <name>  
If not '{', '}', '/', 'relations', or 'entities'  
<application_name> ::= <non_reserved_name>  
<default_entities> ::= <V_name> <default_entities> | ε  
<entities> ::= <entity> | <entity> <entities>  
<entity> ::= <V_name_of_entity> <V_name_of_superentity> <application_name> <icon_filename>  
<V_name_of_entity> ::= <V_name>  
<V_name_of_superentity> ::= <V_name>

D4 Requirements for the application(s)

At present, due to the simplified communication model, the NLE application is required to be a Prolog program having the following four predicates defined:

• v_trl_paraphrase(TRL)
  True if TRL is a TRL. Prints the NL paraphrase of the TRL to standard output as a side effect.

• v_trl_answer(TRL)
  True if TRL is a TRL. Prints the answer, to the query represented by the TRL, to standard output as a side effect.

• v_nl_to_trl(NL)
  True if NL is a natural language expression that the application can parse. As a side effect the TRL representation of the statement is printed to standard output.

• v_evaluate(NL)
  True if NL is a natural language expression that the application can answer. As a side effect the answer to the query is printed to standard output.

In the real system, the intention is that V and the application(s) would communicate through a number of separate channels (e.g. sockets/pipes on a Unix system), one for natural language, one for the interface language, and one for commands.

A special "feature" of the Emacs tile in the main interface window, is that any line that starts with an exclamation point ('!') in the first position, is sent unmodified directly to the application(s).

E "User Manual"

Before starting V, make sure that the resource files (See "Description of Resource Files" on page 49) exist and are correctly specified.

Start XPCE, possibly with the name of the V resource file (if it is different from
~/.Vresources. Change directory to that containing the V files and either load v.ql (Prolog directive load('v.ql')). or consult v.pl (Prolog directive [v]).. Finally, start V by entering the goal run. The main V window and the output window should appear, looking something like figure 30.

![V: A Visual Query Language for Multimodal Interfaces](image)

**Figure 30.** The V main window and output window just after starting the interface.

The leftmost rectangle (tile), second from the top, contains the standard buttons (Tools, Clear, and Evaluate), and the application specific buttons (defined in the application resource file. Tools opens the tool window, Clear clears the query board, and Evaluate evaluates the visual query.

The largest rectangle is the query board, where visual queries are composed. Queries are posed by clicking the Evaluate button, and answers to the queries will appear in the output window.

How to compose a query? By pressing the right mouse button with the mouse
cursor on the query board, a create relation/entity menu is opened. This is a pull-
right menu and by dragging the mouse to the right with the mouse button still
pressed, either the create relation menu or the create icon menu opens. From this
menu the user the selects an item to be included in the query. If a relation item
was chosen, the relation with its default entities (defined in a resource file) will be
placed on the query board. Another way to create a relation is by dragging the
mouse, with the left mouse button and shift button simultaneously pressed, from
the left entity to the right entity of the relation and then release the buttons. A
default relation appears between the left and right entity.

The user selects an object by a single click with the left mouse button. By press-
ing the right mouse button on the selected object, a the entity menu opens. From
this menu the user can change the entity.

If the user selects the cut item (or ^x) from the edit menu, the selected items are
cut to the clipboard. From the clipboard it is possible to paste back the cut objects
with the “paste” item (^p) from the edit menu.

When pressing the right mouse button in one of the boxes inside an entity, a
quantifier menu opens. From this menu the user can select a quantifier to be
placed inside the box from which the menu was opened.

A composite entity is constructed by pressing the right mouse button inside one
of the boxes placed above a relation arc. The entity next to the quantifier becomes
the principal entity of the subquery. The composite entity consists of all the rela-
tions and entities connected to this principal entity, but not including the relation
with the composite entity quantifier though.

The leftmost rectangle, immediately below the query board is the natural lan-
guage input area. This tile is in most respects an Emacs editor, and thus all the
normal Emacs commands can be used (e.g. ^k to cut a line, ^space and M-w to
copy, ^y to paste, ^p to go to the previous line, ^n to go to the next line, etc.) When
carriage return is pressed, the current line is sent to the NLE, and answers will
appear in the output window. One special “feature” of this Emacs tile is that if the
first character of a line is an exclamation point (!!), the remainder of the line will
be sent uninterpreted directly to the NLE. The tile immediately to the right of the
Emacs tile is intended to contain buttons for Emacs specific commands. Currently
it only contains a Clear button for erasing the Emacs buffer.

The lowermost tile contains radio buttons that govern what to do when evaluat-
ing a query. In the Evaluate state, both VL and NL queries are sent to the applica-
tion for evaluation, in the Echo Natural Language as Visual Language state NL
queries are paraphrased as VL queries, and in the Paraphrase Visual Language as
Natural Language state, finally, VL queries are paraphrased as NL queries.