Proceedings of the First International Workshop on PEIRCE: A Conceptual Graphs Workbench

Gerard Ellis
Robert Levinson
(Eds)

Las Cruces, New Mexico, July 10, 1992

Contents

Author Index

Abstract

PEIRCE is a project aiming to build a state-of-the-art, industrial strength conceptual graphs workbench. PEIRCE is integrating the conceptual graphs development efforts that are taking place around the world. There are already 60 researchers from 10 countries involved in the project.

PEIRCE will provide a robust, portable, freely available conceptual graphs workbench that will fast track new techniques into the community; facilitate comparison of competing techniques; help researchers cooperate in development; and speed application development.

On July 10, 1992 a workshop on PEIRCE was held as part of the Seventh Annual Workshop on Conceptual Graphs, in Las Cruces, New Mexico, July 8-10. The PEIRCE project grew from the cooperative research community that exists on the conceptual graph mailing list (`cg@umn-cs.cs.umn.edu`). Members of the conceptual graph community are co-researching and co-developing topics within the PEIRCE project. The level of co-operation within the community rose to new heights at the workshop in Las Cruces.

The first public release of the PEIRCE software will take place at the next PEIRCE Workshop in August 1993. The PEIRCE workshop will run in association with the First International Conference on Conceptual Structures in Quebec City, Canada August 4-7.

1

The Birth of PEIRCE: A Conceptual Graphs Workbench

Gerard Ellis [1]
Robert Levinson [2]

ABSTRACT PEIRCE is a project aiming to build a state-of-the-art, industrial strength conceptual graphs workbench. PEIRCE is integrating the conceptual graphs development efforts that are taking place around the world. There are already 60 researchers from 10 countries involved in the project. This paper describes the PEIRCE project and the structure for managing the distributed development of PEIRCE. PEIRCE will provide a robust, portable, freely available conceptual graphs workbench that will fast track new techniques into the community; facilitate comparison of competing techniques; help researchers cooperate in development; and speed application development.

Introduction

The PEIRCE project is named after the American philosopher Charles Sanders Peirce who in 1896 developed the logic Existential Graphs [Rob73]. Peirce described existential graphs as "the logic of the future". It is nearly a century since Peirce made that statement, and only now is his logic poised to become a reality with the widespread use of powerful graphics workstations that can run the PEIRCE workbench.

John Sowa introduced Peirce's Existential Graphs to the field of artificial intelligence, as the foundation of his Conceptual Graphs theory [Sow84]. Conceptual graphs is an order-sorted logic with a standard mapping to natural language. A diverse, active research community has grown up around Sowa's work. Each year an international conceptual graphs workshop is held, this being the seventh. Next year the first international conceptual graphs conference will be held. A survey of current research and practice in the conceptual graph community can be found in [NNGE92].

There are several very good implementations of subsets of conceptual graphs theory, and there are many people doing innovative work on particular aspects. But there is not as yet a robust, widely available set of tools for developing applications based on conceptual graphs.

The PEIRCE project is an open, nonproprietary, international, collaborative effort to build an industrial strength, portable, freely available conceptual graphs workbench (PEIRCE). The project is open to collaboration from all researchers in the conceptual graphs community and others.

PEIRCE will allow developers to write / draw / parse / learn large conceptual graphs programs / databases / ontologies. A module will be included to demonstrate PEIRCE's support for natural language processing - a major application for many conceptual graphs researchers. An adaptive, pattern-oriented chess playing system [LS91], Morph, will be used to verify and show the high-level support for learning mechanisms and databases of conceptual graphs. Morph is currently implemented in C. Much of the technology in Morph is built on top of a generic retrieval, learning and search model known as APS [GL92]. APS and Morph will form a valuable part of the PEIRCE Workbench. PEIRCE will also be used for knowledge based systems, graphical programming, computer assisted theorem proving, specification and verification.

PEIRCE will fast track the introduction of new techniques into the community. New techniques will be directly evaluated against old by using the same data, platforms and many of the same support

[1] Comp. Sc. Dept., Univ. of Queensland, Brisbane, QLD 4072, Australia. ged@cs.uq.oz.au

[2] Dept. of Comp. and Inf. Sc., Univ. of California, Santa Cruz, 229 Appl. Sc. Building, Santa Cruz, CA 95064, USA. levinson@cse.ucsc.edu

routines. The project will nurture synergism between competing mechanisms. The developers will be able to assemble hybrid tools from basic tools using well documented interfaces. Prototyping of methods will be done in PEIRCE, thus the prototypes will be available immediately for evaluation by the community.

There is a group dedicated to interfacing PEIRCE to large ontologies such as CYC. PEIRCE will accept ontologies in many formats, KIF (Knowledge Interchange Format), CYC, KL-ONE, SNePS, etc. Access to large collections of knowledge will prompt and validate new techniques.

The PEIRCE project at the time of writing involves over 60 researchers from over 10 countries each developing part of the workbench. The diversity of development environments is itself a test of portability. But, portability is not left to chance, programming standards are being set.

This paper describes the main high-level modules contained in PEIRCE, each of which is being developed by researchers spread around the world. The paper describes the organization and glue needed to successfully complete the distributed development of a large software project such as PEIRCE.

1.1 Parts of PEIRCE

The PEIRCE project is divided up into modules including

- Programming Standards
- Database Storage and Retrieval
- Linear Notation Input and Output
- Massively Parallel Hardware
- Graphical Editor and Display
- Conceptual Catalogs (Ontologies)
- Programming in Conceptual Graphs with Constraints
- Inference/Theorem-Proving Mechanisms
- Learning Mechanisms
- Natural Language Parsers and Generators
- Information Systems Engineering
- Vision System

Each of these groups is described in terms of their dependency on other groups and the sorts of functions they provide. The remaining chapters of this proceedings further details the work of each group.

1.1.1 Programming Standards (STDS)

The core implementation language of PEIRCE is C^{++}. Advanced services can be implemented in either C^{++}, Prolog or other high-level languages on top of the core. The development environment of choice is X-Windows on UNIX workstations. Development on IBM PC compatibles is underway, and there are plans for portability to MacIntosh. This group is establishing guidelines for language and environment versions, naming conventions, system calls, and software release procedures.

1.1.2 Database Storage and Retrieval (DB)

The DB group is involved in manipulating, storing and retrieving large numbers of conceptual graphs. This group is involved in implementing core operations such as graph matching (projection), unification (join), indexing, semantic distance, etc. Many of the PEIRCE modules are dependent on the DB module. The DB module is a core service of PEIRCE.

Testing the database requires large numbers of conceptual graphs, thus the database is dependent on the conceptual catalog group. In the short term the graph mechanisms are being tested on graphs (not necessarily conceptual) that are automatically generated. For example the chess graphs generated by Morph [LS91] are being used to test classification and compilation techniques in the generalization hierarchy [Ell92b].

Database operations are encapsulated in a partially ordered set of conceptual graphs ADT, CGPoset (see Section 1.2), so that implementations of each of these methods can be easily replaced. Current implementations of the generalization hierarchy (CG database) [Lev92b, Ell92b] are data structures kept in memory. This is hidden from the users of CGPoset. In the future, large, external, distributed conceptual graphs databases can be used as a replacement or alternative to current implementations of CG databases.

1.1.3 Massively Parallel Hardware (HARDWARE)

In the long term the Von Neumann machine may not be an appropriate platform for intelligent systems. At the moment users have to tell the machine where something is when they want to get it. But in a contents addressable memory, the user can describe the thing they want and the machine will retrieve it. This group will look at massively parallel hardware to support contents addressable memories as the base for an intelligent platform for PEIRCE.

1.1.4 Linear Notation Input and Output (LINEAR)

The LINEAR group is charged with standardizing the linear notation for conceptual graphs and providing parsers and generators for the notation. [EL92] contains a grammar for the standard notation. The LINEAR group records any extensions to the syntax, such as including the programming languages: Prolog, LIFE [AKP91], and CGC (Conceptual Graphs with Constraints).

1.1.5 Graphical Editor and Display (GRIP)

The GRIP group is standardizing the graphical notation and developing graphical editors and displays. The editor will look something like the palette system for XFig, MacDraw etc. GRIP will allow the user to draw assertions and queries (conceptual graphs) on the sheet of assertion. The palette will contain concepts, relations, contexts, etc. Answers to queries will give a view of the sheet assertion, that is, the query will act as a filter on the sheet of assertion. GRIP will include 3-Dimensional rendering of knowledge. There will also be a standard package for printing conceptual graphs in documents. A linear notation parser will be written in PostScript. The brackets, parenthesis, and arrows in

```
[type:ref]->(type)->[type:ref]...
```

will be implemented by PostScript macros. Thus PostScript will parse the linear form of conceptual graphs and print the intended graphical form.

1.1.6 Conceptual Catalogs (CCAT)

The CCAT group needs a database system and a linear/graphical interface for encoding ontologies in conceptual graphs. As well as developing their own ontologies and some support tools, CCAT is interfacing PEIRCE to large collections of knowledge such as CYC. CCAT is involved in designing maintenance routines for merging ontologies; building translators for knowledge representations other than conceptual graphs: CYC, KIF (Knowledge Interchange Format), KL-ONE, SNePS, etc. These data sets will be valuable in testing the other modules, as well as forming the basis for many applications.

1.1.7 Programming in Conceptual Graphs with Constraints (CGC)

The language CGC will include conceptual graphs as a subset, but add notation for specifying focus and functional constraints. CGC will have a similar control structure to Prolog but will replace Prolog's Herbrand terms and predicates with conceptual graphs. Prolog can be viewed as a restricted set of conceptual graphs theory. A compiler for CGC will also compile Prolog. CGC would eventually replace Prolog after it has bootstrapped itself from a Prolog implementation. People implementing software in conceptual graphs can be viewed as already programming in CGC. With the addition of GRIP, people will be able to truly program in conceptual *graphs*.

The philosophy behind CGC is that the tools available to the programmer should not be restricted. That is, the compiler should be clever enough to determine when the more powerful features are not being used and compile the program as efficiently as Prolog or simpler structures in these cases. We believe that strategies designed to compile and index large sets of graphs will outperform the general search technique of Prolog when applied to graphs. This is being investigated in the ongoing work on compilation of conceptual graphs in [Ell92b].

CGC will have all the powerful data structures of Prolog such as lists, trees, records etc, compiled efficiently. As well as having access to general graphs.

If PEIRCE members style their Prolog programs so that their representation of graphs can be replaced with conceptual graphs, they will be able to take advantage of the graph indexing and compilation techniques developed in the future. For example the following CGC program

```
[CAT: Tom]<-(AGNT)<-[EAT]->(OBJ)->[MOUSE].
?- [CAT:*x]<-(AGNT)<-[EAT]->(OBJ)->[MOUSE:*y],
   write('The cat '),
   write_name(*x),
   write(' is eating a mouse '),
   write_name(*y).
```

can be translated to the following Prolog program by a preprocessor

```
cg(c('CAT':'Tom'/[1], 'EAT':*/[1,2], 'MOUSE':'Jerry'/[2]),
        r('AGNT'/r(2,1), 'OBJ'(2,3))).
?- call_cg(cg(c('CAT':X/[1], 'EAT':*/[1,2], 'MOUSE':Y/[2]),
        r('AGNT'/r(2,1), 'OBJ'(2,3)))),
   write('The cat '),
   write_name(X),
   write(' is eating a mouse '),
   write_name(Y).
```

In the interim, the program will call the predicate `call_cg`

```
call_cg(QueryCG) :-
        clause(cg(Cs, Rs)),
        match(QueryCG, cg(Cs, Rs)).
match(QueryCG, DatabaseCG) :-
        ...
```

By building this level of encapsulation, calls to `call_cg` can be removed by the CGC compiler in the future and the graph will be directly compiled into C^{++}code to do unification/matching within the CGC database when such a compiler is available.

PEIRCE will be a conceptual graphical programming system that will allow developers to draw programs (using conceptual graphs) or write programs (using natural language) that use large bodies of knowledge in many different formats.

The CGC group is also charged with investigating, documenting, and standardizing calls across programming languages used in PEIRCE. The languages supported initially are C^{++}, Prolog and LIFE [AKP91]. Section 1.2 describes some of the general abstract data types needed for PEIRCE. For example a graph ADT in Prolog may call C^{++}routines to do graph matching. A C^{++}conceptual graphs ADT may call a Prolog routine for parsing conceptual graphs. The programming language of choice will eventually be CGC. The LINEAR group is working on extending the syntax of the parser to handle Prolog and LIFE programs.

1.1.8 Inference/Theorem-Proving Mechanisms (PROOF)

The PROOF group will develop proof procedures for conceptual graphs. The alpha and beta rules of Existential Graphs [Rob73] will make excellent interactive proof tools. Eventually proof techniques developed by this group will be written in CGC.

1.1.9 Learning Mechanisms (LRN)

LRN is providing mechanisms for learning search control knowledge, structural concepts from noisy examples, conceptual clustering and classification. LRN requires a database system, conceptual catalog, and programming language. The LRN group is intimately involved with the database group DB. Classification of conceptual graphs in the generalization hierarchy is the foundation of structural learning. The LRN group will provide graph generalization, construction and extraction routines.

Ultimately LRN will use CGC to encode it's learning mechanisms. By using conceptual graphs to encode learning mechanisms, eventually PEIRCE will learn meta-rules, that is, learning strategies. For example the following is an example of a rule to forget conceptual graphs.

```
IF   [CG:*x]->(DEGREE-OF-USE)->[REAL]->(LESS-THAN)->[REAL: 0.1]
THEN [*x]-><FORGET>.
```

LRN is vital to the long term research prospects of PEIRCE. LRN will supply the large numbers of graphs that are necessary to test the DB module. Computer generated graphs are currently the only source of large sets of complex graphs needed to test the DB module.

1.1.10 Natural Language Parsers and Generators (NLP)

One of the goals of PEIRCE is to support natural language processing. PEIRCE supplies the NLP group with a database system; large conceptual catalogs; matching, unification, and other graph manipulation routines; a theorem-prover; a high level programming language, CGC; linear and graphical editors; and learning mechanisms. The NLP group will specify its requirements implementing much of its own high level requirements.

1.1.11 Vision System (VISION)

The PEIRCE project hopes to get people involved in a third application, vision, to demonstrate the processing power of the system as well as to provide a useful set of library routines. Applications developers of all kinds are encouraged to get involved now, so PEIRCE can supply the tools they need.

1.2 The foundation of PEIRCE

At the heart of PEIRCE is a collection of abstract data types (ADTs). One such ADT is the conceptual graphs ADT. The conceptual graphs ADT include methods for

1. subsumption - an operation for comparing two conceptual graphs.
2. read - a conceptual graphs parser: either linear (supplied by LINEAR) or graphical (supplied by GRIP).
3. write - a conceptual graphs display system (linear or graphical).
4. decompose - decompose a conceptual graphs into a set of canonical formation rules.
5. compose - compose a set of canonical formation rules into a conceptual graph.
6. join - the canonical formation rule for joining two conceptual graphs.
7. copy - copy a conceptual graph.

8. restrict-type - restrict a concept's type.
9. restrict-referent - restrict a concept's referent.
10. simplify - remove a duplicate relation.
11. generalize - generalize a graph by removing structure or generalize concepts.
12. specialize - specialize a graph by adding structure or specializing concepts.
13. semantic-distance - calculate the semantic distance given another conceptual graph.

The CG ADT is built on a general graph ADT. General graph algorithms that are written for the graph class are inherited by the CG class. The graph class will be relatively stable, any changes to conceptual graphs theory will generally be independent of the basic graph formalism. Conceptual graphs can be seen as bipartite graphs with concept and relation labels:

```
CG             :: Bgraph<Concept, Relation>.
Concept        :: Abstraction Referent.
Relation       :: Abstraction.
Referent       :: generic | Individual | Poset<CG>.
Individual     :: int.
Abstraction    :: Type* | Lambda.
Type           :: code Approximation.
Approximation  :: int | real | string.
Lambda         :: Slist<int> CG.
```

One problem is that many of the routines work on different subsets of conceptual graphs theory. For instance many database methods do not consider nested graphs, quantifiers, type definitions. That is, they only support flat graphs with (undefined) types and individual markers. It is important that methods work for general graphs, as these can be extended to the CG subclass level by working on routines for the labels.

There is a partially ordered set ADT (poset) that is used to implement the hierarchies used in conceptual graphs such as the type hierarchy, and the generalization hierarchy. The set of methods for the Poset ADT will include

1. insert - insert an element into the poset.
2. delete - delete an element from the poset.
3. immediate-successors - return the set of immediate successors of an element.
4. immediate-predecessors - return the set of immediate predecessors of an element.
5. merge - merge two posets.
6. statistics - print statistics on cost of queries, structure of poset, etc.
7. display - display the poset in a hierarchical form.
8. encode - encode the hierarchy using the bit encoding method for posets described in [AKBLN89].
9. code - find the binary code for a element in the poset.
10. decode - find the element(s) of the poset given a binary code.
11. glb - compute the greatest lower bound (code) of two given elements of the poset.
12. lub - compute the least upper bound (code) of two given elements of the poset.

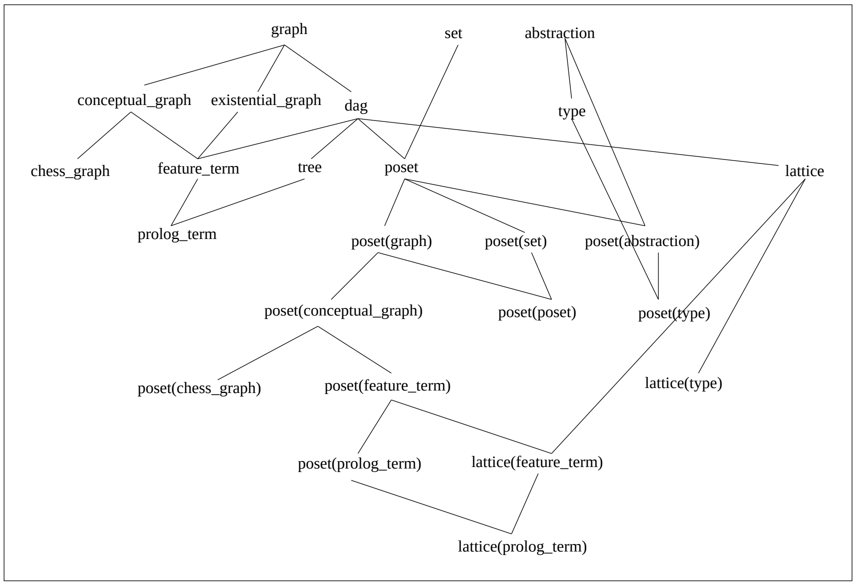

FIGURE 1.1. A Hierarchy of ADTs in PEIRCE

The Poset ADT handles the type hierarchy, the generalization hierarchy over conceptual graphs, and hence the generalization hierarchy over Prolog terms, and feature terms. Using a graph's decomposition rules, the hierarchy and the graph elements can be compiled directly into C++ code rather than a data structure that requires interpretation. Figure 1.1 illustrates the hierarchy over ADTs in PEIRCE.

The advantage of this object-oriented approach is that much of the useful work can be done in the graph, poset, poset(graph), and lattice ADTs that will remain independent of any changes to conceptual graph theory, and can be reused for other graphical languages.

The Poset ADT can be used to implement feature logics and Prolog-Like languages as well as conceptual graphs. The Poset ADT will form the basis of the database provided by the DB group. It will also form the core of the CGC language. CGC will be compiled down to C++ code using the `poset(conceptual_graph)` ADT.

This work will make use of the retrieval methods described in [Ell92a, LE91]. Some of the ideas in these papers have already been implemented and are in use in Morph [LS91]. There is also an implementation in C^{++}that will form the basis of PEIRCE. [3] Through restructuring and compilation of graph data these retrieval methods are expected to satisfy the efficiency requirements that will allow PEIRCE to be a viable tool for both research and industrial-strength application development.

1.3 Distribution and Control

The PEIRCE workbench will be stored at several FTP sites around the world: `cs.uah.edu` (United States), `cluster.cs.su.oz.au` (Australia) (see Section 2.6.3). Subgroups will send their code to a team of people for vetting before release on the FTP sites.

1.4 Important Dates

The first official meeting of the PEIRCE project participants will be the First International Workshop on PEIRCE: A Conceptual Graphs Workbench held in association with the Seventh Annual Conceptual Graphs Workshop in Las Cruces, New Mexico, in July 1992. Subgroups working on each of the PEIRCE

[3] The linear notation parser described in [Pag90] is being integrated with the database to demonstrate the database at the conceptual graph workshop and is the first test of PEIRCE's distributed development.

modules will give status reports [EL92].

The first public release of the PEIRCE Workbench is scheduled for demonstration at the International Conference on Conceptual Structures in Quebec City, Canada, in August 1993. It will include linear and graphical interfaces and a CG database with production rules designed to facilitate the integration of the other modules. Future versions will build on this core. There will also be a PEIRCE workshop at the Conceptual Graphs conference in Quebec.

Summary

PEIRCE is an exciting project with a level of cooperation not found in many research communities. PEIRCE will produce a robust, portable, freely available conceptual graphs workbench. This shared effort will allow many of the researchers in the conceptual graphs community to get on with the research they are really interested in: natural language processing, knowledge based systems, graphical programming, data modeling, vision, learning, planning, etc.

The international PEIRCE TEAM so far includes

Jerrold Aronson	SUNY Binghamton, New York	United States
Alex Bejan	IBM Kingston, New York	United States
Gary Berg-Cross	Advanced Decision Systems	United States
Keith Campbell	Stanford Univ	United States
Key-Sun Choi	Korea Adv Inst Sc Tech	Korea
Tae-Sang Choi	Univ Missouri-Kansas City	United States
Young Bae Choi	Univ Missouri-Kansas City	United States
Peter Creasy	Univ Queensland	Australia
Walling Cyre	Virginia Tech	United States
Harry Delugach	Univ Alabama-Huntsville	United States
Peter Eklund	Univ Adelaide	Australia
Gerard Ellis	Univ Queensland	Australia
John Esch	Paramax Systems	United States
Jean Fargues	IBM Paris Scientific Center	France
Norman Foo	Univ Sydney	Australia
Brian Gaines	Univ Calgary	Canada
Brian Garner	Deakin Univ	Australia
Steve Graham	Univ Missouri-Kansas City	United States
Rob Greenwood	Virginia Tech	United States
Young-Suk Han	Korea Adv Inst Sc Tech	Korea
Tim Hines	Univ Missouri-Kansas City	United States
Kees Hoede	Univ Twente	The Netherlands
Wen-Jung Hsin	Univ Missouri-Kansas City	United States
Chris Khoo	Syracuse Univ	United States
Myung-Cheol Kim	Korea Adv Inst Sc Tech	Korea
Pavel Kocura	Loughborough Univ Tech	United Kingdom
Fritz Lehmann	GRandAI Software	United States
Bob Levinson	Univ California at Santa Cruz	United States
Michel Liquiere	Reunion Univ	France
Peter Ludemann	Quintus Corp.	United States
Dickson Lukose	Deakin Univ	United States
Meenakshi Manek	Virginia Tech	United States
Graham Mann	Univ New South Wales	Australia
Illias Mavreas	Univ Missouri-Kansas City	United States
Guy Mineau	Laval Univ	Canada
Robert Muehlbacher	A C Nielsen	Austria
Sung Hyon Myaeng	Syracuse Univ	United States
Antoine Ogonowski	GSI-ERLI	France
Jonathan Oh	Univ Missouri-Kansas City	United States
Maurice Pagnucco	Univ Sydney	Australia
Heather Pfeiffer	New Mexico State Univ	United States
Simon Polovina	Loughborough Univ Tech	United Kingdom
James D Roberts	Univ California at Santa Cruz	United States
Kelvin Ross	Univ of Queensland	Australia
S Shankaranarayanan	Virginia Tech	United States
Doug Skuce	Univ Ottawa	Canada
Kwang-Jun Seo	Korea Adv Inst Sc Tech	Korea
John Sowa	SUNY Binghamton	United States
Aniruddha Thakar	Virginia Tech (US)	United States
Bill Tepfenhart	Bell Labs	United States
Bosco Tjan	Univ Minnesota	United States
Harmen van den Berg	Univ Twente	The Netherlands
Eileen Way	SUNY Binghamton	United States
Wei Wei	Univ Missouri-Kansas City	United States
Yin Min Wei	Univ Ohio	United States
Michel Wermelinger	Univ Nova de Lisboa	Portugal

Mark Willems	Univ Twente	The Netherlands
Vilas Wuwongse	Asian Inst of Tech	Thailand
Gi-Chul Yang	Univ Missouri-Kansas City	United States
Wen Zhang	Virginia Tech	United States

2

STDS Software Development Standards

Harry S. Delugach [1]

2.1 Introduction

The PEIRCE software development effort is a collaborative project that will ultimately involve many programmers in different locations. The main advantage of this approach, of course, is that the resulting workbench will have capabilities beyond what any single person could produce. The obvious pitfall to this approach is that separately-developed pieces of software must be able to operate together as a coherent whole. Experience strongly suggests that software development standards are essential to a successful collaborative effort. This paper recommends some preliminary standards for the PEIRCE project, and their rationale.

One important consideration in the selection of standard software is that it be freely available to developers around the world, particularly outside the United States, and that it have some degree of general acceptance so that some developers (but perhaps not all) will have some acquaintance or expertise with the software being considered. Each software product mentioned herein is known to be in the public domain, and can be freely obtained, copied and distributed for research (non-profit) purposes.

2.2 Purpose and Scope of Standards

A standard embodies an agreement between participants in a software development effort. A standard prescribes some structure, practice, form, behavior or procedure that everyone agrees to follow. Such an agreement must therefore be based on a belief that the standards given are reasonable, practical, and are specifically intended to support integration of the eventual product. If developers do not like the standards, or if they do not believe the standards will help the overall effort, then the standards will be ignored, and integration will be difficult.

An important purpose of any standard is to enable persons other than the original authors to understand, modify and/or incorporate someone else's work. When the work is finally completed, one or more original authors may be on vacation, no longer associated with the project, unwilling to answer questions, or otherwise unavailable. Even in those rare cases where only one author will ever be involved, adherence to standards will help that individual author in organizing his/her efforts.

Developers are encouraged to adhere to the set of standards, whatever their eventual form may be. Since participation in the PEIRCE project is voluntary, no one may require the use of standards. It is therefore even more important that everyone understand why standards are needed and why these particular standards were chosen. Responsibility for adhering to the standards rests entirely with the participants, although the caretakers or coordinators may take it upon themselves to inform developers of variations from the standards.

This chapter should therefore be considered a draft standard, to be discussed and modified as the community of developers desires. I have attempted to give a justification for each standard that is recommended, so that developers will understand the rationale behind the standard and its importance to the overall effort. No standard will be a "perfect" choice, nor will it please everyone; nevertheless, adherence to even an imperfect standard will result in a better product than if the standard is ignored.

---

[1]Department of Computer Science, University of Alabama in Huntsville Huntsville, AL 35899 U.S.A., delugach@cs.uah.edu

2.3 Definitions

The following terms are used in this chapter:

- **Caretaker** - someone performing an administrative function, such as maintaining a repository or mailing list. This person is primarily responsible for accurate and systematic carrying out of his/her duties.
- **Community** - all persons involved in the PEIRCE development effort, whether as programmers, designers, documenters, consultants, coordinators, or caretakers.
- **Coordinator** - a person with responsibility for making decisions regarding the technical aspects of PEIRCE. Module designation, naming, requirements, and integration are part of this person's duties.
- **Header** - a set of labeled information to identify a unit or a part of a unit. Described below as a unit identification header.
- **Module** - A uniquely named design/programming unit with a well-defined purpose and well-defined interface to other modules.
- **Repository** - A collection of units maintained by a caretaker on behalf of the entire development community.
- **Subsystem** - A set of modules, grouped together by purpose or function, to be specified by name and agreed upon by the development community as a whole.
- **Unit** - Any one of the following:
  a design module specification,
  an implementation module,
  a report,
  a manual,
  a test data set,
  a bug report,
  a change request.

2.4 Project Documentation Standards

Since collaboration depends upon communication between participants, documentation standards are not a luxury but a necessity. Developers must rely on written units to convey information to the community.

2.4.1 Manuals and Technical Reports

The standard text format for manuals and technical reports shall be LaTeX, as described in [Lam86]. This is a subset of TeX, as described in [Knu84]. It is available via anonymous ftp from byron.u.washington.edu (128.95.48.32) in the directory ./pub/unixtex. It is large, so please be sure to observe proper ftp etiquette when getting it. It is also available through the University of Washington, for a small fee. Contact:

```
Director
Northwest Computing Support Center
Thomson Hall, Mail Stop DR-10
University of Washington
Seattle, WA 98195 (USA)
(206)543-6259
```

or send electronic mail to Elizabeth Tachikawa at elisabet@max.u.washington.edu (note the "s"). The fee charged for getting the TeX distribution through the University of Washington helps fund the further development of Unix TeX, so it's a good idea to order it this way. This distribution compiles under Ultrix, but BSD systems have used it as well. Many support programs and utilities are available via anonymous ftp from available from ymir.claremont.edu (134.173.4.23) in [anonymous.tex] and its subdirectories.

Rationale:

- While not an ANSI standard, LaTeXseems to be the text processor of choice in many university settings, at least in the United States and Australia. At least one publisher, Springer-Verlag, accepts "camera-ready" manuscripts using its own style files. The PEIRCE project may also define its own style files, based on Springer-Verlag's, to re-format documents with a minimum of re-editing.
- LaTeXhandles bibliographies and citations with ease.
- It is freely available.
- LaTeXsource files are in ASCII text form, which is conveniently mailed electronically to other participants, transferred from machine to machine. etc.
- There are many utilities for converting to/from LaTeXdocuments and other formats, and also utilities for previewing LaTeXdocuments under X11R5.

2.4.2 Unit Identification Header

A PEIRCE unit that is submitted for integration into the PEIRCE Repository must conform to the PEIRCE development standards. Every unit submitted must have a This unit defines what belongs in a PEIRCE unit header. Submitters are urged to include every item in the header, and to keep them in the standard order given. The best approach is to consider that for some purposes, only the unit header will be distributed.

Every unit should have at least the following information embedded within its contents. If an item does not apply, show it as "n/a" for "not applicable". Items shown as *Required* must be included for any unit. A template is given in Unit-Header-Template

- **Name.** *Required.* The unit's name, unique within a subsystem. For a report, manual, bug report or change request, a longer title should be supplied, but a name is always required.
- **Title.** *Required.* A longer (one phrase) title for a report, manual, bug report or change request. For program or design units, this may given as "n/a" for "not applicable"
- **Subsystem.** *Required.* The subsystem name, as agreed upon by the development community, to which this unit belongs. Subsystems so far are shown below, abbreviation followed by group designation.
  - *CCAT* : Conceptual Catalogs and Ontologies
  - *CGC* : Programming in Conceptual Graphs With Constraints
  - *DB* : Database and Retrieval
  - *FACE* : Programming Interfaces for C, C++, Prolog and Others
  - *GRIP* : Graphical Interface
  - *ISE* : Information Systems Engineering
  - *LEARN* : Learning Mechanisms
  - *LINEAR* : Linear Notation Interface
  - *NLP* : Natural Language Interface: Understanding and Generation
  - *PROOF* : Inference and Theorem-Proving Mechanisms

  - *STDS* : Software Management and Programming Standards
  - *VISION* : Vision Systems

- **Type.** *Required.* A unit can have any one of the following types:
  - *Design Module* : a design module specification
  - *Module* : an implementation module
  - *Tech Report* : a technical report, suitable for publication.
  - *Note* : a note, less formal than a report and not for publication.
  - *Manual* : a manual, either for users or for developers.
  - *Testing* : a test data set, test plan or report.
  - *Bug* : a bug report,
  - *Change Request* : a change request.
- **Filename.** The name of the file in which the unit is stored, making it easier for others to locate a machine-readable version of the unit.
- **Purpose.** A brief description of the intent of the unit, as well as any applicable scope, or function information. This section may also contain a brief description of the unit.
- **Author.** *Required.* Includes the author's name, postal address, phone number and electronic mail address.
- **Date.** *Required.* The date on which the module was delivered to the repository. Date should be expressed in day-month-year form, preferably as DD MMM YY
- **Version Type.** A version type indicates whether the given unit is for testing, information only, offered for comments, or release for general use. Version type designations are:
  - *Preliminary* : For general comments and feedback.
  - *Release* : For general use.
  - *Urgent* : For immediate attention.
  - *Stable* : For use with caution; bugs likely with bug reports welcome.
  - *Testing* : For general testing purposes; send bug reports.
- **Version Number.** Each group may assign version numbers as they wish. In general, versions ending in .0 should represent significant differences from any previous versions.
- **Change Log.** Changes to units must be documented explicitly. The change log provides a facility for listing each change as it is made. Change log should include:
  - *Date* : Date that the change was made, in day month year form, e.g., 14 Jul 1992.
  - *Person* : Person(s) who made the change.
  - *Change and Reason* : Description of the change and the reason for the change. Provide as much detail as necessary.
- **Related Files/Units.** Any other files or units that may affect or use this one, or that are affected or used by this one.
- **Variance from Standards.** Include here a description of any variations from these standards, with an explanation of why the standard does not apply and how the variance will impact other units over the lifetime of the development effort.
- **Language.** The language in which this unit is expressed. Possible languages are: C, C++, Prolog (specify dialect), LaTeX, UNIX csh, etc.

- **Files Needed.** Show any "include" files, library files, input data files, preprocessors, etc. that must be available in order to build and use the module.
- **Build Commands.** Show the command(s) that put the unit into a usable form.
- **Input and Output.** A description of the form of all input data used and output data produced by the module. If data is in a data structure, then the data structure must be carefully and completely specified. If data is in a file, it must be described as either binary or ASCII text information, and its layout is to be specified. Examples will be helpful. *For Design/Implementation Modules.*
- **Functional Processing.** A description of what the module is intended to do. Data flow diagrams, or other descriptive forms are welcome. Be as specific as possible, particularly for implementation modules. *For Design/Implementation Modules.*
- **Algorithms Used.** A description of the algorithms used, a source for each if there is one, and a justification for why that particular algorithm was chosen. *For Design/Implementation Modules.*
- **Test Data.** Documentation must include test data sets, both for typical or usual function of the software, and for exceptional cases. *For Design/Implementation Modules.*
- **References.** Any publications, etc. relevant to the unit.

2.5 Design and Programming Standards

The following standards apply to the design and implementation of units used in the PEIRCE project. Design is the process of creating specifications for software modules. The standard unit of design will be the *module*. A design module may be implemented by one or more implementation modules/files/functions. A design module may be specified by text. An implementation module will in the form of the standard programming language.

2.5.1 Module Documentation

Both design and programming modules must contain all information specified in the unit identification header (see above). Additional information must be supplied as follows:

- *Input and Output.* A description of the form of input and output data used and produced by the module. If data is in a data structure, then the data structure must be carefully and completely specified. If data is in a file, it must be described as either binary or ASCII text information, and its layout is to be specified. Examples will be helpful.
- *Functional Processing.* A description of what the module is intended to do. Data flow diagrams, or other descriptive forms are welcome. Be as specific as possible, particularly for implementation modules.
- *Algorithms Used.* A description of the algorithms used, a source for each if there is one, and a justification for why that particular algorithm was chosen.
- *Test Data.* Documentation must include test data sets, both for typical or usual function of the software, and for exceptional cases.

2.5.2 Programming Language

The programming language for all implemented modules will be C++, version 2.1. A description of the language can be found in [Str91]. A version of C/C++ is available via anonymous ftp in /pub/gnu at prep.ai.mit.edu (the GNU archive) from the Open Software Foundation for non-profit distribution.

Rationale:

- The combined GNU C and C++ compiler adheres closely to the proposed version 2.1 standard that is being considered. There is no ANSI standard C++, but the C portion does meet the ANSI C standard.
- C++ supports an object-oriented programming style, which encourages programmers to encapsulate abstractions into well-defined objects with specific operations to manipulate them.

2.5.3 Coding Standards

Changes to source code must be identified (see above) both in the unit header and at the point in the code where a change was made.

Programmers are urged to code for generality. Arbitrary constants should not be embedded in the source code; instead, numbers and strings should be specified using the "#define" facility. Symbolic names thus defined will all be in UPPER CASE letters. Functions shall begin with upper case letters. Other identifiers will be all lower case.

2.5.4 Operating System

The operating system standard will be POSIX 1003.1-1990. It is based on UNIX, so most UNIX developers will be able to conform to it without much additional effort. Since POSIX supports the C programming language, it is the most reasonable operating system standard. Further details can be found in [Zlo91]. The standard itself [ISO90] is available from the publisher.

2.5.5 Graphic Interface

The graphic interface will be X11R5. It is a window-based interface, supported by most UNIX systems. The X code is available free via FTP from export.lcs.mit.edu.

Several tool sets are based on X11R5. Motif is available from the Open Software Foundation, under a licensing agreement. Interviews is available via FTP from interviews.stanford.edu, and has a mailing list at interviews@interviews.stanford.edu.

2.6 Project Management Standards

Since the management of the project is divided among several people, developers will need to observe certain rules for distributing their finished products. In particular, naming conventions must be observed. Each PEIRCE subgroup has been given an abbreviation, such as STDS or CCAT. Each filename will have an abbreviation as its prefix.

The repositories (see below) will contain a list of all currently-known units (whether in development or completed), their file and/or subsystem names, and the person(s) responsible for that unit.

2.6.1 Testing of Modules

Developers are responsible for testing their modules as fully as possible before offering them to the rest of the community, unless the module is released explicitly for the purpose of testing by others.

2.6.2 Integration of Modules

A set of system makefiles will be maintained in the module repository (see next section). A small group of persons will be designated as integrators; their job will be to incorporate new modules into the system, add appropriate lines to the makefile, and announce a new addition to the PEIRCE community. If problems arise during integration, the integrators will return the module to its authors for further modification. Integrators will *not* debug others' code.

2.6.3 Module Repository

A common repository of PEIRCE modules will be maintained. The repository will be accessible via FTP (a standard file transfer utility) with an anonymous login. Several sites will maintain identical repositories; their caretakers will ensure (as far as practical) that the contents are identical. Members of the community are urged to access the nearest site, to minimize data transfer costs and lessen our impact on the sites whose managers have kindly volunteered to help with PEIRCE. Two repository sites are currently set up:

- cluster.cs.su.oz.au (129.78.8.1). /pub/cg/peirce
  Caretaker: Maurice Pagnucco (morri@karl.cs.su.oz.au)
- cs.uah.edu (146.229.2.2). /pub/cg/peirce
  Caretaker: Harry Delugach (delugach@cs.uah.edu)

Adding Units To the Repository

For security reasons, it is advisable to mail submissions to one of the caretakers who will then add the unit(s) to the repository.

Changing and Updating Units

Changes to units in the repository are handled much as an original submission. The new unit's change log and header will indicate changes from the old unit. In submitting the changed module, its authors will indicate exactly which old unit(s) the new one is replacing.

Deleting Units

Deleting a unit may present problems if the unit is required for the operation of other already-distributed units. If a unit is to be deleted, the author(s) send a request to the caretaker with a reason for the deletion. If the deletion seems reasonable, then the caretaker will post a notice to that effect.

Repository Log

A repository log will be kept of all changes to the repository. It is not practical for the caretaker to note all retrievals from the repository, so it is up to users of the repository's units to periodically check the log for any changes that are relevent to their own work.

2.7 Acknowledgements

I wish to acknowledge Mike Hewett's excellent contribution to the programming standards and his list of references.

3

DB: Conceptual Database

Gerard Ellis [1]
Robert Levinson [2]
Key-Sun Choi [3]
Young-Bae Choi [4]
Peter Creasy [5]
John Esch [6]
Sung-Hyon Myaeng [7]
Heather Pfeiffer [8]
John Sowa [9]
and Gi-Chul Yang [10]

3.1 Introduction

The DB group is involved in manipulating, storing and retrieving large numbers of conceptual graphs. Fundamental operations to be implemented in the CG database include: graph matching (projection), unification (join), and indexing. Many of the PEIRCE modules are dependent on the DB module for these operations. For this reason the DB module is the core of PEIRCE.

The DB subgroup is one of the larger subgroups in the PEIRCE project, with over 10 people working on PEIRCE's conceptual graphs database. We welcome other volunteers.

Issues that this group focuses on include:

- storing and retrieving conceptual graphs
- fundamental operations on conceptual graphs
  - graph matching
  - maximal join (unification) of conceptual graphs
- close matches
- semantic distance
- classification
- generalization and learning

---

[1] Key Centre for Software Technology, Department of Computer Science, University of Queensland, Brisbane, QLD 4072 Australia, ged@cs.uq.oz.au
[2] Department of Computer and Information Sciences, University of California Santa Cruz Santa Cruz CA 95064 U.S.A, (408)459-2087, ARPANET:levinson@pal.ucsc.edu
[3] Korea Advanced Institute of Science and Technology (KAIST) kschoi@cs.kaist.ac.kr
[4] University of Missouri-Kansas City, choi@vax2.cstp.umkc.edu
[5] University of Queensland, peter@cs.uq.oz.au
[6] Paramax Systems, esch@email.sp.unisys.com
[7] Syracuse University, shmyaeng@mailbox.syr.edu
[8] New Mexico State University, hdp@nmsu.edu
[9] SUNY Binghamton, sowa@turing.pacss.binghamton.edu
[10] University of Missouri-Kansas City, yang@vax2.cstp.umkc.edu

- deductive databases
- compilation of conceptual graphs
- indexing conceptual graphs
- merging databases
- large knowledge bases
  - large numbers (e.g., 10^9) of relatively small graphs (e.g., 100 nodes)
  - large conceptual graphs (e.g., 10^6 nodes)
  - using current database technology
  - distributed database
- knowledge base management
  - external databases
  - recovery

3.2 Progress to Date

There is already a degree of collaboration in the PEIRCE DB subgroup. Levinson and Ellis have collaborated on designs for associative databases [LE91]. Levinson, Ellis, and Yang et al. are working on integrating their methods into the PEIRCE DB module implemented in C^{++}. Each of these methods works on associative retrieval of graphs from a database. There is an exciting opportunity for synergism of these methods. The techniques developed can be compared to existing techniques in PEIRCE, using the same data and platform.

A major problem (common to other subgroups)is that there will be multiple implementations of the same routine. It is necessary to design a mechanism that will allow these competing mechanisms to coexist. This is easy in the case where only one implementation exists in the executable. This can be done by each developer using the same ADT definition (header file) and supplying their own version of the back end of service routines. The version is selected when making the executable.

3.2.1 CGMA : A Novel Conceptual Graph Matching Algorithm

Yang, G-C., Choi, Y.B., and Oh, J.

A novel Conceptual Graph Matching Algorithm (CGMA) [YCO92] is proposed along with a knowledge base organization scheme. This method indexes each graph on its concepts and relations. Each concept and relation in a query graph is hashed to a set of graphs in the database. An intersection of these sets finds the set of graphs that contain all of the concepts and relations of the query graph.

CGMA allows the user to select the Degree of Matching and Degree of Inheritance, thus allowing *close matches*, rather than exact matches. A conceptual graph is represented in a special linear form (called U-Form) for CGMA. U-Form eliminates explicit arcs in the graph. The CGMA method builds on experience [YO90] in constructing an efficient dictionary management system for machine translation.

3.2.2 Canonical Databases

Esch, J., and Ellis, G.

When building conceptual graph databases, ill-formed graphs are often encountered. This occurs when separate well-formed pieces are joined in an attempt to construct a well-formed graph. This is avoided in canonical databases, which only allow use of canonical formation rules on graphs already in the canonical database. This fits in with the work of Ellis in Section 3.2.5 where graphs in the database are represented as canonical formation rules performed on immediate generalizations in the generalization hierarchy.

The PEIRCE system needs to allow ill-formed graphs, otherwise we could not reason about such graphs. We need to be able to reason that "green ideas sleeping furiously" are ill-formed, so we must still be capable of storing these graphs.

3.2.3 CONSTRUCT: A Conceptual Graph Processor

Esch, J.

A major efficiency problem in CONSTRUCT [ENYG89] is that all arcs were kept as linked lists. If type and relation definitions can be "compiled", so their instances have direct references, performance would be at least an order of magnitude better.

3.2.4 Multi-Level Hierarchical Retrieval

Levinson, R., and Ellis, G.

This method is based on classification of graphs into a hierarchy. That is the relationship between the graph to be inserted and all of the graphs in the database are found. The graph is inserted by attaching links to the closest (least general) generalizations and closest (least special) specializations in the database.

Classification is done in two phases [Lev92b, Ell92a]. Phase I is a constrained topological search of the generalization space to get the immediate generalizations of the query graph. That is a level order traversal of the hierarchy constrained to generalizations of the query. Phase II searches topologically the intersection of the subhierarchies found in Phase I.

The multi-hierarchical retrieval method [LE91] is based on hierarchies of the form

- conceptual graph hierarchy (generalization hierarchy)
- node descriptor hierarchy
- node descriptor unit hierarchy
- type hierarchy

The conceptual graph hierarchy contains terms with a rich structure, node descriptors have less structure, node descriptors still less, with types having the minimal structure necessary to differentiate terms.

A graph can be described by a set of node descriptions. A Node description contains information about the type of the node, and nodes and arcs attached to the node within a radius determined by the user. Node descriptors are like subgraphs of the original graph. By finding the set of node descriptions in the database, the conceptual graphs with those node descriptions can also be found. To compare node descriptors it may take $O(n^3)$ time in the worst case, whereas to compare conceptual graphs may take exponential time, $O(2^n)$. Finding the set of node descriptions and intersecting the sets of graphs indexed approximates computing general graph comparison over the whole database. The nodes descriptors contain information necessary for a graph to be related. However, in real applications this method has proved sufficient for returning only related graphs.

There is a partial order (hierarchy) over node descriptions. Node descriptions are stored and retrieved in the node descriptor hierarchy, which is also implemented using the general classification algorithms developed for the generalization hierarchy.

Similarly a node descriptor can be indexed by its set of node descriptor units. These are similar to single concepts and relations. There is a hierarchy of these as well. The general classification method is used to classify these patterns also.

The multi-level hierarchical method allows search in the hierarchy without doing graph comparisons. The method is designed to handle complex graphs which are computationally expensive to do normal graph comparison.

The hierarchy is a data structure kept in RAM. These techniques are all based in memory, as yet there is no external database capability. The hierarchy is written out as a text file when shutting down the database, and is loaded from this file at startup time.

3.2.5 Compiling Conceptual Graphs

Ellis, G., and Levinson, R.

In [GT87] conceptual graphs are represented as differences between a graph and its neighbours in the generalization hierarchy. In [Ell92b] these differences are in the form of instances of the canonical formation rules. These rule instances are implemented by an abstract machine for conceptual graphs where the formation rules are abstract machine instructions. Alternatively, the rule instances can be expanded into C^{++}code and optimized.

Methods have also been designed for compiling the hierarchy structure itself into C^{++}code rather than left as a data structure to be interpreted by some algorithm. Decision Graphs [KS90] have been designed for indexing Prolog terms using matching. This method is a compilation of depth-first search through the hierarchy. Ellis is adapting the method to more general terms such as conceptual graphs.

3.2.6 Adaptive Maximal Join

Creasy, P., and Ellis, G.

In the Information Systems Engineering subgroup report (this volume) Creasy and Ellis describe an interactive join technique for conceptual schema integration. The system tries to do a maximal join, when conflicts or under specification arises, the system prompts the user for more information to resolve the conflict, then continues joining.

3.2.7 Conceptual Programming Environment

Pfeiffer, H. D.

The Conceptual Programming Environment, CP, is currently implemented in LISP; but is being reimplemented in C^{++}. The database of conceptual graphs is implemented by storing the graphs into an internal hash table structure using 3-tuples from database theory. These graphs are stored as objects where the object follow a direct hierarchical structure of object types [PH91]. The CP environment handles linear and graphical input and can handle objects of multiple types, such as, *FACTS*, *DEFINITIONS* (schemas, relations, types), *PROCEDURAL* (time and space), and *CONSTRAINTS* (feasibility and runtime).

The CP environment includes the current graph operations of **join** and **project**, with the current research interest being on implementing both maximal and partial joins using time and space saving algorithms over single and multiple referent concepts.

3.2.8 Large Databases

Sowa, J.

Current prototypes do all operations in main memory and only read the CG databases in and out at the beginning and end of a session.

That strategy is fine for a prototype, but very large knowledge bases will have to be read in and out dynamically. Even more of a challenge is to support multiple access by many users on a LAN or even a world-wide distributed network.

There are systems like Andrew at Carnegie-Mellon, which support hundreds or thousands of computers with a single hierarchical file structure. Each user's personal file system is treated as a branch or perhaps several branches of the global system. For each user, the local files look like a part of the global system, but other users cannot see your files unless you grant them the authority to access them.

Many people are talking about extending such campus-wide systems to a global network with all the computers in the world connected to a single giant file system. Anyone could access one of someone's public files in another country with the same commands used for accessing files on a local disk. Such systems are still in the planning stage, but by the time the PEIRCE project has implemented all the glorious features that people are thinking about, the hardware and software systems for extending them to a global knowledge base should be available.

With such a view, the file system becomes first and foremost a naming system, and only secondarily a storage system. For systems like Andrew, the location of your data is irrelevant. You can sign on to a computer in your office or one in your home, and see exactly the same files in both. When you sign off from the computer in your office, the files you changed are automatically migrated back to the central server. Then when you go home and sign on, you find exactly the same files with all the latest updates.

In conceptual graphs, we have a hierarchical context system, which is a logical partitioning that could be associated with a hierarchical file system. We probably wouldn't want to allocate a separate file to each tiny context that was used only to negate a single relation. But the principle could be that any context that the user gives a name to would be associated with a file of the same name in the file system (perhaps with a couple of extra extenders to specify the owner, etc.). If you don't name a context, it would just be grouped in the same file as some containing context that does have a name.

For the actual I/O operations, we could let the system worry about that. On some systems, such as the IBM RS/6000 (a UNIX-based system) or the IBM mainframes (which can run several different operating systems, including IBM's version of UNIX called AIX), they have memory-mapped I/O.

What that means is that the hardware supports very large virtual addresses of 48 bits or more. Then when you say "Open file", no actual I/O occurs, but the physical address of the entire file is loaded into the page tables for the virtual memory. The programmer just pretends that the entire file is all in main memory, and the system takes care of paging it in or out as needed.

Since not all systems have such hardware, the PEIRCE DB routines should be designed to use it if available or mimic it if not. You could have commands like "find context N" or "save context N", where the name N would be associated with some file. If the system has memory-mapped I/O, the subroutine that implements the command would just do an open file and let the hardware worry about I/O. But if the system does old-fashioned I/O, the subroutine would read the whole file into main memory. The logical interfaces in either case would be identical.

When you're doing list processing in virtual memory, you have to worry about the possibility that a list or graph might have nodes that get scattered over many megabytes or gigabytes of virtual address space, and the paging demands on the system might degrade performance. After you do joins, for example, a single graph might have pieces stored in widely separated parts of the file.

To improve performance with virtual memory, many list-processing systems automatically compact the storage whenever they do garbage collection. Instead of moving data around in the old file, they open a fresh clean file and copy the accessible lists (or graphs) from the old file into the new one; then they rename the new file with the old name and discard the old file. All the lists or graphs in the new file are in contiguous storage locations, and the inaccessible fragments are thrown away.

In summary:

1. Continue to program all graph operations as if they were in main memory.
2. But divide a knowledge base into contexts according to the usual CG conventions. Named contexts would be associated with named files, and unnamed contexts would be grouped in the smallest containing named context.
3. No new CG formalism is needed to name contexts: Since a context is represented by a concept, you can just link it to a name with the NAME relation. If you want to know whether a context has a name, just check to see whether it is linked to a NAME relation.
4. However, the PEIRCE project should decide upon some conventions for assigning names in a systematic fashion. A hierarchical naming scheme that parallels the system file names seems reasonable.
5. This raises a question about coreference links between graphs in different contexts. So far, we have just been using simple variables like *x or *y to represent coreference links. We may have

to introduce names like *N.x or *N.y, where N is the name of the outermost context in which the node *x or *y occurs. If a context doesn't have a name, then N could be just some GENSYM variable that would be invisible to the human user, but it could be changed to a hierarchical name if the user later gave a name to the context.

6. Design the routines for finding and saving named contexts in such a way that they are largely independent of the operating system facilities. The only system dependencies would be localized in small subroutines that either did memory mapped I/O if available or did physical I/O for an entire file at a time.

This kind of an approach could be implemented on current systems, and it would allow a CG system to grow into a world-wide network if and when such facilities became available – all of world knowledge would then become one giant conceptual graph.

3.3 Testing the Database

Testing the database requires large numbers of conceptual graphs, thus the database is dependent on the conceptual catalog subgroup. In the short term the graph mechanisms will be tested on graphs (not necessarily conceptual) which are automatically generated. For example the chess graphs generated by the adaptive, pattern-oriented chess playing system, Morph [LS91], are being used to test classification and compilation techniques in the generalization hierarchy.

3.4 Interactions with other Subgroups: LRN, CGC

The graphs generated by the learning algorithms from the LRN subgroup will supply the necessary numbers of graphs for testing large databases. The learning algorithms themselves learn certain graphs, for example, an induction method over conceptual graphs generates generalizations of graphs in the database. These graphs are useful for representing graphs as differences in [Ell92b]. That is learning algorithms which generate new graphs have the potential to speed search in hierarchies.

The CGC subgroup is developing Conceptual Graphs with Constraints (CGC) as a programming language. The development of a compiler for CGC will be tightly integrated into storing and retrieving conceptual graphs from a database developed by the DB subgroup. Techniques being developed to index and compile graphs in a generalization hierarchy will generalize to handle production rules in the CGC language.

4

Hardware for PEIRCE

James D. Roberts [1]

4.1 Introduction

The newly formed PEIRCE hardware group is beginning to collect requirements and performance information. Only the author and Claude Nosphitz, both at UCSC, are working specifically on PEIRCE hardware issues, so volunteers are more than welcome. This paper includes the results of discussions with Robert Levinson and Gerard Ellis. The goal is to support PEIRCE with hardware platforms that can dramatically improve performance for conceptual graph research and applications. To meet this goal we will investigate hardware alternatives to workstations, particularly parallel accelerator boards and systems. We will select a kernel of abstract data types and methods, permitting PEIRCE to be easily ported a variety of platforms.

4.2 Objectives

First, the group will collect and organize performance information and requirements, including:

- which PEIRCE modules and applications need acceleration;
- how processing, memory, and disk requirements scale with problem size;
- the distribution of characteristics of graphs, hierarchies, and posets;
- what are desirable problem sizes for research and applications.

Next, we will identify viable means of accelerating PEIRCE for large applications and estimate achievable speedups. Platform options include:

- high performance RISC processors and mainframes,
- existing parallel machines,
- specialized architectures.

It is anticipated that uniprocessors, including mainframes, will be too slow for extremely large applications. Existing massively parallel systems are unlikely to prove adequate, as they are tuned to numeric scientific applications and emphasize vector processing.

Application-specific hardware clearly lacks the flexibility required for an evolving PEIRCE and existing parallel PROLOG and LISP engines do not support the high degree of parallelism necessary for dramatic speedup. Therefore, it is proposed that a versatile new parallel architecture tailored to symbolic processing will be required.

Key questions include what portions of PIERCE are parallelizable, and what grains of parallelism are efficiently realizable. Communication patterns specific to the various grains of parallelism in PEIRCE are particularly important. The distributed database issues discussed by Sowa [EL92] are also vital to the success of very large scale applications. In addition to a distributed database, distributed processing

[1] Department of Computer Engineering, University of California Santa Cruz Santa Cruz CA 95064 U.S.A, (408)459-4043, ARPANET:donrob@cse.ucsc.edu

should also be investigated. This may take the form of major modules executing at the sites of their developers. Network latency and transfer rate may be crucial for many applications.

Supporting the goals of PIERCE, we expect to select a kernel of low-level ADTs and methods which are parallelizable or can otherwise take advantage of special hardware features. In keeping with PIERCE's software goals, the kernel will be portable to different hardware platforms with the bulk of the PIERCE code unchanged. Initial work will be on the conceptual database module, PEIRCE's core, with emphasis on the conceptual graph and poset ADTs and methods presented by Ellis and Levinson in [EL92]. The following methods can particularly benefit from fine grain parallelism:

- subsumption;
- poset encoding, GLB, LUB, insert and delete;
- immediate-successors and immediate-predecessors.

The hardware group will study high performance platforms such as mainframes, parallel systems, and algorithmically specialized hardware, in addition to the workstation and personal computer portability provided by the main PEIRCE software group.

4.3 The MISC Machine

Hardware research at UCSC includes the development of a general purpose parallel symbolic architecture, the MISC Machine (Misc Is Symbolic Computing). The MISC Machine is a novel approach to massively parallel hardware for AI and cognitive modeling. Its architecture features multiple arrays of simple processing elements, each array supervised by a more complex processor. MISC emphasizes associative processing, a form of data parallelism in which memory is accessed by content rather than address, as the basis for intelligent systems. The architecture extends prior work such as SNAP [M+92] and IMX2 [H+91], providing greater flexibility and scalability. MISC will support a variety of knowledge representations and AI approaches. Target applications include high-level neural network models in addition to semantic networks and conceptual graphs.

Much of the inspiration for the MISC Machine came from work with Levinson's Morph chess program, particularly the partial order and node descriptors of Method IV [Lev92b] and Ellis' compilation techniques [Ell92b]. The Morph group has explored a distributed parallel implementation, but results have been poor due the high latency of local area networks. The MISC machine takes an alternative approach utilizing fine grain data parallelism, emphasizing content addressable memory (CAM) techniques. Much of the MISC research concerns overcoming the limitations of CAM, enabling its use for the dynamic linked data structures and data dependant control structures characteristic of AI. In addition to multiple CAM arrays, the MISC Machine features a supervisor processor for each array to address coarser grains of parallelism. The supervisors are also tailored for symbolic programs, and are necessary for control and data structures too complex for the CAM arrays. The MISC architecture is highly scalable, with potential implementations ranging from single supervisor systems for embedded intelligence (such as diagnostics and control) to workstation accelerator boards to a full scale supercomputer.

4.4 Innovative Hardware for PEIRCE

The MISC research indicates that systems based on content addressable processing can provide excellent hardware support for large high performance conceptual graph applications. This approach also appears encouraging in the light of prior results utilizing CAM in AI systems for backtracking, clause filtering, conflict resolution, demand driven reduction, garbage collection, inference, Rete, unification and variable binding [K+89, Kog91]. CAM is also well known for its benefits in symbol and pattern matching. With full data parallelism many operations that require lengthy algorithms on conventional processors can be performed in constant time with simple code. This can radically alter how one conceptualizes and codes a problem. Constant time CAM operations include:

- exact, partial, and closest match;
- set operations (intersection, union, isMember, isSetEmpty);
- maximum, minimum, mean;
- magnitude comparison;
- arithmetic and logical.

Like other parallel processors, CAM based systems provide a factor of N speedup (where N is the number of data elements) for algorithms such as shortest path and minimum spanning tree, and similar speedups for matrix and relational database operations [FKS86]. The hardware cost for such a massively parallel system need not be prohibitive, as bit-serial CAM requires only a small constant factor area increase over that required for passive data storage.

4.5 Conclusion

The hardware group will investigate platforms for accelerating PEIRCE compilation and execution, with alternatives ranging from workstations to existing high performance computers to special purpose accelerator boards and parallel systems. Emphasis will be placed on parallelizing the core ADTs and methods. Major research issues include:

- code characteristics, performance requirements, hardware requirements, and target problem sizes;
- which ADTs and methods can be efficiently parallelized, what grains of parallelism are available for exploitation;
- appropriate hardware systems and hardware characteristics, the potential for speedup.

This research will provide dramatic speedup beyond workstations. One proposed accelerator, the MISC Machine, is currently under development. A full MISC system will hold over 10 million nodes in memory and execute most primitives in constant or logarithmic time.

5

LINEAR : Linear Notation Interface

Maurice Pagnucco [1]
Young Bae Choi [2]
Heather Pfeiffer [3]
Michel Wermelinger [4]

5.1 Introduction

The linear form of a Conceptual Graph is an important form of input and output for many Conceptual Graph tools. It also serves as an important notation for presenting Conceptual Graphs to others. It is therefore an integral part of the PEIRCE project.

In this paper we shall introduce the PEIRCE linear notation interface subgroup (LINEAR) which is devoted to the development of the linear form notation and related tools. We shall outline its aims and current members, along with other issues relevant to the subgroup.

5.2 Objectives

The prime objective of the group is to produce a parser and generator for the linear form of a Conceptual Graph. At present, we have begun work on a C++/LEX[LS]/YACC[Joh78] version of the parser that will be capable of parsing a small subset of the linear form grammar (see Appendix A). This initial form of the grammar is based on John Sowa's presentation in [Sow84].

The parser will accept, as input, ASCII descriptions of Conceptual Graphs in linear form. Provided the description is syntactically correct, the relevant values will be placed in the appropriate PEIRCE core data structures. Any operations that are to be performed on graphs or that need to use the graphs (e.g., displaying graphs in graphical form, joining graphs, etc.) will then make use of these data structures. Conversely, the generator will output those data structures into an ASCII description of the equivalent linear form.

Once the initial version of the grammar is completed we intend to gradually augment it, adding not only suggested extensions to the linear form for specifying Conceptual Graphs but also other important productions that implement various aspects of Conceptual Graph theory (e.g., a production to allow users to enter concept subtype information thus specifying a concept type hierarchy). This task will be accomplished through the use of various presentations of the linear form notation, in particular [Esc89], [Sow84] and [Sow89], and through discussion with the Conceptual Graph community.

The development of this grammar will place the group in an excellent situation to define a standard for the linear form notation of a Conceptual Graph. It is expected then, that the group will coordinate any suggestions regarding the linear form notation (changes, extensions, etc.) proposed by other PEIRCE subgroups and members of the Conceptual Graph community in order to achieve this aim.

---

[1] Knowledge Systems Group, Basser Department of Computer Science, Madsen Building, F09, University of Sydney, Sydney, NSW 2006, AUSTRALIA. `email: morri@cs.su.oz.au`

[2] Computer Science Telecommunications Program, University of Missouri-Kansas City, 5100 Rockhill Road, Kansas City, Missouri 64110-2499, USA. `email: choi@vax2.cstp.umkc.edu`

[3] New Mexico State University, hdp@nmsu.edu

[4] Dep. de Informática, Univ. Nova de Lisboa, Quinta da Torre, 2825 Monte da Caparica, PORTUGAL. `email: mw@fct.unl.pt`

5.3 Members of the LINEAR Subgroup

The members of the LINEAR subgroup and their relevant research interests in this area, are:

- Young Bae Choi
  A parser for a Conceptual Graph matching algorithm called CGMA (Conceptual Graph Matching Algorithm) was developed as a submodule of the implementation of CGMA. It is written in Prolog [YCO92].
- Maurice Pagnucco (Coordinator)
  Honours degree thesis on graphical interfaces for Conceptual Graphs [Pag90]. A parser for the linear form of a Conceptual Graph which was developed as part of this project is available, in modified form, via ftp from:
  - `cluster.cs.su.oz.au (129.78.8.1):`

    `/pub/peirce/cgparser.tar.Z`
  - `crl.nmsu.edu (128.123.1.18):`

    `/pub/cg/cgparser.tar.Z`
- Michel Wermelinger
  Implemented CGT (Conceptual Graph Tools) [Wer91] as part of his B.Sc. thesis. CGT is written in Prolog and includes a DCG linear form parser and generator. CGT and related documentation is available via ftp from
  - `cluster.cs.su.oz.au (129.78.8.1):`

    `/pub/peirce/cgt.tar.Z`

5.4 Relationship to Other Subgroups

The activities of the LINEAR subgroup are likely to affect most of the subgroups involved in the PEIRCE project. This is partly due to the importance of the linear form notation as a form of input and output for many Conceptual Graph tools but also due to the fact that many of the subgroups will need to access the data structures into which the input Conceptual Graphs (in linear form) will be parsed. The former point enables the use of the linear notation as a file interchange format, thereby providing an easy way to exchange knowledge bases between the various PEIRCE modules. The latter point is very important as the various subgroups that need to use and manipulate the data structures will have to be clear on what to expect in them once a graph has been parsed.

One group that does deserve particular mention, though, is the Conceptual Catalogue (CCAT) subgroup. It is anticipated that this subgroup will be of special importance to the LINEAR subgroup in supplying and collating data sets to thoroughly test the Conceptual Graph parser to be developed by this group.

5.5 Implementation

As mentioned previously, the LINEAR subgroup is currently developing an initial version of the parser using the C^{++} programming language, the LEX lexical analyser generator and the YACC parser generator tools. This will facilitate its integration with the current PEIRCE code.

In the future there may be the possibility of developing a Prolog version of the parser along with other tools related to the linear form notation. Any tools produced will be based directly on the grammar developed by the group.

5.6 Problems in Integration

The main problem in integrating the parser to the remainder of the PEIRCE code is likely to arise in deciding how to represent the parsed information in the data structures (a problem alluded to in the previous section). The needs of each subgroup should be catered to by the appearance of the data structures and the values they expect to find contained therein. Extensions to the linear form notation may be particularly problematic as they usually represent abbreviations for more complex Conceptual Graphs. In fact, procedures for expanding these abbreviated forms into their equivalent graphs may be required.

5.7 Conclusions

The linear form is an important notation for representing Conceptual Graphs. It is the aim of the LINEAR subgroup to coordinate the development of a standard for the linear form notation. To this end, the group has begun production of a parser which it hopes to expand into a robust implementation of the linear form grammar.

Appendix A

This appendix contains a skeleton YACC grammar for the linear form of a Conceptual Graph. It represents the initial grammar to be used by the LINEAR subgroup in implementing a parser and is therefore only a small subset of the entire linear form grammar.

The grammar below can be understood as a series of production rules — the head of the rule is *recognised* once all the elements in the body have been *recognised*. Alternatives are separated by a vertical bar ('|'). Terminal symbols are represented by upper case identifiers (e.g., NAME, NUMBER) or by surrounding them with single quotes (e.g., '.', '-', '>'). C style comments are also used.

```
%%
file            :       file element sep
                |       sep
                ;

sep             :       sep '\n'
                |       /*empty rule*/
                ;

sep2            :       sep2 '\n'
                |       '\n'
                ;

element         :       cg '.'
                ;

cg              :       cgraph
                ;

cgraph          :       concept
                |       concept rlink
                |       relation conlink
                ;

rlink           :       arc relation
                |       arc relation conlink
                |       '-' rlist ','
                ;

conlink         :       arc concept
                |       arc concept rlink
```

```
                |       '-' conlist ','
                ;

rlist           :       sep2 relation
                |       sep2 relation conlink
                |       rlist sep2 relation
                |       rlist sep2 relation conlink
                ;

conlist         :       sep2 arc concept
                |       sep2 arc concept rlink
                |       conlist sep2 arc concept
                |       conlist sep2 arc concept rlink
                ;

concept         :       '[' typefield ']'
                |       '[' typefield ':' reffield ']'
                ;

relation        :       '(' typelabel ')'
                ;

arc             :       '<-'
                |       NUMBER '<-'
                |       '->'
                |       NUMBER '->'
                ;

reffield        :       '*' gmarker
                |       '#' nmarker
                ;

gmarker         :       NAME
                |       /* empty rule */
                ;

nmarker         :       NUMBER /* this number should be
                                  a positive integer */
                |       /* empty rule */
                ;

typefield       :      typelabel
                ;

typelabel       :       NAME
                ;
%%
```

6

GRIP: Graphical Interface

Peter Eklund[1]
Michel Wermelinger[2]
Keith Campbell [3]
Harry Delugach [4]
Robert Muelbacher [5]
Heather Pfeiffer [6]

6.1 Objectives

A major advantage of conceptual graphs is their ease of use particularly in the graphical form. The goals of the graphical interface subgroup is the creation of a document providing general guidelines for the implementation of such an interface and the creation of one (or more) implementations conforming to those guidelines. Furthermore, members of the group plan to make available a suité of portable universal interfaces for the representation and acquisition of conceptual graphs.

6.2 Products

There are a number of interfaces that can be considered within the boundaries of the GRIP's interests. The most obvious being a generalized interface that arranges and displays the core graph ADTs on a graphics view surface, and, conversely, translates graphs constructed with the interface into the core ADT structures. Such an interface should be customizable to user requirements, for example concept and relations of particular types may be given a specific characteristic in terms of shape, color or shading.

Such an interface needs to include simple editing facilities for text and graphical representations that allow consistency between the representations, i.e. changes to the graph knowledge base may effect changes to the graphical form and visa versa. In addition it needs to be flexible enough to cover a wide range of applications from Knowledge and Expert System based applications tools, Database Query formation, Natural Language Understanding and Text Retrieval systems. In other words the interface has to be general enough to be easily extended to accommodate the needs of the entire conceptual structure community.

The above represents the function of the core graphical interface that the GRIP subgroup hopes to develop by the time of the first International Conference on Conceptual Structures in Quebec City, Canada in August 1993. Although there are several additional areas that will be considered by the subgroup, the functionality of the core interfaces is considered fundamental to progress in interface and acquisition in conceptual graph research.

[1]Department of Computer Science, University of Adelaide, PO Box 498, Adealide 5001, AUSTRALIA
Email: `peter@cs.adelaide.edu.au`

[2]Dep. de Informática, Univ. Nova de Lisboa, Quinta da Torre, 2825 Monte da Caparica, PORTUGAL
email: `mw@fct.unl.pt`

[3]Section on Medical Informatics, Stanford University, email: `campbell@sumex-aim.stanford.edu`

[4]Computer Science Department, University of Alabama in Huntsville

[5]A.C. Nielsen, Austria, email: `rmu@cmis.co.at`

[6]Computing Research Laboratory, New Mexico State University, email: `hdp@nmsu.edu`

6.3 Platform

The tools will be written in C++ and will run in an XWindows environment. Additionally, members of the group will concentrate their attention to developing functionally equivalent tools for the Apple Macintosh.

6.4 Research Objectives

1) Temporal and Modal forms of conceptual structure representation in graphics. Since there is considerable practical interest in temporal logics for planning and representation, one of the subgroups long term objectives is to develop interfaces that reflect the chronology of actions and events. This is also true of modal and higher order variants of conceptual structures. By experimenting with explicit modal, belief, knowledge and default operators a clearer picture of the relation between and explicit nonmonotonic form of conceptual structures and a mechanism for belief revision will become apparent. Experimenting with visual representations of modal and temporal forms will motivate this study.

2) Incorporating an epistemic regime into the human-computer interaction. One of the great advantages of conceptual structures is that they are first-order equivalent. This means that ambiguous interpretations of the graphs can only occur via modeling errors or mistakes in the mind the designer. The designer of a conceptual graph may believe that the graph reads one way when its logical extension reads in another. By imposing a formal epistemic system for internalizing user input, the implications of one or more graphs can be drawn. This in practise corresponds to incremental theorem proving during the design stages of conceptual structures to avoid the even greater consequences of mistakes inferred at a later point in time.

Other areas of interest to this subgroup include knowledge acquisition aspects of conceptual structures, natural language interfaces, and graph-based output from text retrieval and database queries. The subgroup aims to attract researchers working in human-computer interaction and knowledge acquisition for knowledge based systems who have an interest in conceptual structures. In addition it will be of interest to those working with higher-order representations of conceptual structures. Since both 1) and 2) are research oriented their realization in software represent advanced facilities for future generations of the core system.

6.5 Relationship with Other Subgroups

The Graphical interface group hopes to satisfy the demands of the entire CG community but would be expected to cooporate most closely with the Database and Linear Interface Subgroups.

6.6 Existing Software

There is already considerable experience in the construction of graphical user interfaces in the group. Eklund [Ekl88] constructed a conceptual structure interface for production rule acquisition in an existing knowledge-base system shell. The results, although favorable, indicated the direction for fundamental improvements and isolated some difficulties with the graphical form [Ekl88]. This tool was constructed in the Poplog[Har92] AI environment using the now outdated Sunview graphics environment. More recently Eklund [Ekl91] has constructed a similar style of interface for semantic nets using CLOS (Common Lisp Object System), CLUE (Common Lisp User Interface) and CLX (Common Lisp Xwindows libraries).

More recent work on conceptual graphs and graphical interfaces is represented in Wermelinger [WL92, Wer91] where a Conceptual Graph Editor (CGE) for X-Windows has been constructed. CGE is coded in X-Prolog, a result of the ESPRIT project "Advanced Logic Programming Environments", extending Quintus Prolog 2.4 in order to enable the programmer to access the X Toolkit and Athena Widgets declaratively. Unfortunately, X-Prolog is no longer supported. The Italian company DS logics has ported and extended it to SICStus Prolog 2.1 and called thew result APPEAL (see the last ALP newsletter). It is planned to port CGE to APPEAL (and therefore make it widely available) and also to Quintus Prolog 3.1, which supposedly interfaces with X Windows.

In the sense that the existing software's platforms are either defunct or at odds with the General Platform requirements of PEIRCE, experience appears the only reusable resource.

6.7 Possible Problems and Challenges in Integration

The major difficulty is in providing an interface which is customizable to different user requirements. The demands of the user working in natural language understanding, text retrieval and knowledge based systems are so diverse that the interface needs to be flexible enough to accommodate these varying needs and yet powerful enough to be of benefit. Devising an interface which is sufficiently flexible and yet powerful enough to be of real value in most environments is largely a specification problem. The nature, style and complexity of the tools proposed for construction are considered by the authors to be relatively straight forward.

7

CCAT: Conceptual Catalogues

Jonathan C. Oh [1]
Steve Graham
Gi-Chul Yang
Liping Yang
Young Bae Choi
Tae-Sang Choi
Wen-Jung Hsin
Fritz Lehmann [2]
Alex Bejan [3]
Key Sun Choi [4]
Guy Mineau [5]
Sung Hyon Myaeng [6]
Doug Skuce [7]
Bill Tepfenhart [8]

7.1 Introduction

In our paper "Lexical Conceptual Grammar" which we will present to the natural language interface subgroup, we discuss the parser/interpreter in some detail. In that paper we introduce the lexicon and conceptual template as auxiliary components for the interpreter or CG compiler. The UMKC group has been working on an ontology based on Dixon's work for the purpose of developing the concept-based lexicon. Doug Skuce mentions Miller's work, which work the UMKC group intends to use as well.

7.2 Previous Works

Currently there are several other groups that have been working on conceptual catalogues: Doug Skuce's ontology now being developed, Paola Velardi's CG interpretations of Italian words, the primitives in Roger Schank's Conceptual Dependency theory, the extensive catalogue of everyday acts in G. P. Zarri's RESEDA system, the abstract CG ontology of time and space of Roger Hartley and related systems, the taxonomy of primitives used in Yorick Wilks' Preference Semantics System (based on the NUDE system of R. H. Richens in the 1950's), and, to the degree it is disclosed, MCC's Cyc catalogue.

In addition, we may consider Julius Laffal's "A Concept Dictionary of English", H.G. Burger's "Wordtree", Anna Wierzbicka's thirteen essential concepts discussed in her books, Igor Melcuk's Explanatory-Combinatory Dictionary, lexicons of a priori artificial languages like Lincos and Loglan/Lojban,

[1] Computer Science University of Missouri at Kansas City
[2] 124 Parsons, Webster Groves, MO 63119 USA. (314)968-8337 fritz@rodin.wustl.edu
[3] IBM Kingston NY, alge@kgnvmy.vnet.ibm.com
[4] Korea Adv Inst Sc & Tech (KAIST), kschoi@cs.kaist.ac.kr
[5] Laval University, mineau@lavalvm1.bitnet
[6] Syracuse University, shmyaeng@mailbox.syr.edu
[7] University of Ottawa, doug@csi.uottawa.ca
[8] Bell Labs, bill@violin.att.com

Roget's Thesaurus and the many ultra-specialized, extensive thesauri used in fields like medicine, geology etc. Other ontologies of interest include the Penman ontology developed by John Bateman and Ed Hovy based on works by M.A.K. Halliday and C. Matheissen.

Beyond this, we should keep abreast of developments in automatic formal interpretation of natural language dictionaries, e.g. the projects of Amsler, Boguraev, Slator and Litkowski.

7.3 Work Plan

Doug Skuce feels that there should be an agreement on the understanding of the term ontology first and on some basic ontological categories for knowledge sharing once the definition of the term is fixed at least within the CG/PEIRCE community, if not in a larger knowledge representation research community.

First he lists some of the uses of the term ontology as follows:

1. a knowledge base (e.g. CYC's)
2. a taxonomy of something (e.g. Roget's)
3. a set of semantic primitives (e.g. Wierzbicka's)
4. a set of very general semantic categories (e.g. Aristotle's)
5. combinations of the above
6. probably several related senses, including meaning "for a restricted domain" (e.g. "an ontology of baseball players").

He proposes to limit the use of the term in sense 4 above. We agree with Doug Skuce that important part of our work consists of finding out who has ontologies of which type and of studying them and relating them and finally converting them to some standard form. He has been developing a general ontology for some time that is intended as a first step toward that goal. He views the following to be some critical steps to take at this point:

1. to determine where PEIRCE group is going to get its shared ontology from.
2. to organize the initial development effort for the most basic (i.e. the highest several levels) set of categories and establish some guidelines or protocol for such effort.

D. Skuce proposes using for an initial knowledge interchange a format that he developed, a simple frame-like text notation with some simple syntax restrictions that is intended to be read mainly by humans, to defer any serious involvement in formal syntactic details until later.

Fritz Lehmann, however, says the first priority of this group should be to collect several conceptual catalogues from different sources; put them into a fairly normal CG format; annotate the entries with explanations, sources and examples (specify necessary and sufficient conditions where appropriate); and make these available to the CG/PEIRCE community. This would allow people to pick and choose. Deciding on a particular shared central ontology (maybe expanding and revising the Conceptual Structures appendix) would be deferred to the next phase. That task involves controversial philosophical choices, and the various sources need to be harmonized. He points out that we should have an informal taxonomy of ontologies, classified so as to give a newcomer some idea of the "level" and "subject area" addressed by a particular ontology. This initial informal taxonomy is to keep us from getting bogged down prematurely in philosophical debates. He believes this CCAT group should leave the design of an interchange format to other groups; when they decide on a format, we'll use it. He also says we should avoid the difficult issue of automatic translation from one sub-ontology (or "microtheory") to another.

Charles Petri at MCC is working on ontologies; see "Easy Programming" in June issue of IEEE Expert. He provides common semantics for words such as "selling", "feasibility", "qualifying", "ballpark estimate", and various computer terms. While the semantics are much looser than those supplied for models in SUMM, there might be something useful in this research for either the DARPA knowledge sharing project or the public clearinghouse for models and ontologies.

8

CGC: Programming in Conceptual Graphs with Constraints

Gerard Ellis [1]
Alex Bejan [2]
Peter Creasy [3]
Harry Delugach [4]

8.1 A CGC Compiler

Ellis, G.

The language CGC will include conceptual graphs as a subset, but add notation for specifying focus and functional constraints. CGC will have a similar control structure to Prolog but will replace Prolog's Herbrand terms and predicates with conceptual graphs. Prolog can be viewed as a restricted set of conceptual graphs theory. A compiler for CGC will also compile Prolog. CGC would eventually replace Prolog after it has bootstrapped itself from a Prolog implementation. People implementing software in conceptual graphs can be viewed as already programming in CGC. With the addition of the GRIP editor, people will be able to truly program in conceptual *graphs*.

The philosophy behind CGC is that the tools available to the programmer should not be restricted. That is, the compiler should be clever enough to determine when the more powerful features are not being used and compile the program as efficiently as Prolog or simpler structures in these cases. We believe that strategies designed to compile and index large sets of graphs will outperform the general search technique of Prolog when applied to graphs. This is being investigated in the ongoing work on compilation of conceptual graphs in [Ell92b].

CGC will have all the powerful data structures of Prolog such as lists, trees, records etc, compiled efficiently. As well as having access to general graphs.

If PEIRCE members style their Prolog programs so that their representation of graphs can be replaced with conceptual graphs, they will be able to take advantage of the graph indexing and compilation techniques developed in the future. For example the following CGC program

```
[CAT: Tom]<-(AGNT)<-[EAT]->(OBJ)->[MOUSE].
?- [CAT:*x]<-(AGNT)<-[EAT]->(OBJ)->[MOUSE:*y],
   write('The cat '),
   write_name(*x),
   write(' is eating a mouse '),
   write_name(*y).
```

can be translated to the following Prolog program by a preprocessor

```
cg(c('CAT':'Tom'/[1], 'EAT':*/[1,2], 'MOUSE':'Jerry'/[2]),
          r('AGNT'/r(2,1), 'OBJ'(2,3))).
?- call_cg(cg(c('CAT':X/[1], 'EAT':*/[1,2], 'MOUSE':Y/[2]),
```

[1] Key Centre for Software Technology, Department of Computer Science, University of Queensland, Brisbane, QLD 4072 Australia, ged@cs.uq.oz.au

[2] IBM Kingston NY, alge@kgnvmy.vnet.ibm.com

[3] Key Centre for Software Technology, Department of Computer Science, University of Queensland, Brisbane, QLD 4072 Australia, peter@cs.uq.oz.au

[4] Computer Science Department, Computer Science Building, University of Alabama in Huntsville, Huntsville, AL 35899 U. S. A. (205) 895-6614, delugach@cs.uah.edu

```
        r('AGNT'/r(2,1), 'OBJ'(2,3)))),
    write('The cat '),
    write_name(X),
    write(' is eating a mouse '),
    write_name(Y).
```

In the interim, the program will call the predicate `call_cg`

```
call_cg(QueryCG) :-
        clause(cg(Cs, Rs)),
        match(QueryCG, cg(Cs, Rs)).
match(QueryCG, DatabaseCG) :-
        ...
```

By building this level of encapsulation, calls to `call_cg` can be removed by the CGC compiler in the future and the graph will be directly compiled into C^{++}code to do unification/matching within the CGC database when such a compiler is available.

PEIRCE will be a conceptual graphical programming system that will allow developers to draw programs (using conceptual graphs) or write programs (using natural language) that use large bodies of knowledge in many different formats.

We hope to be able to supply a first draft CGC interpreter/compiler for the release in August 1993. The LINEAR subgroup will supply a parser that will read CGs, LIFE, and Prolog terms. This will be used as a preprocessor for translating CGC into Prolog or LIFE programs for compilation as described above.

9

LRN : Machine Learning

Robert Levinson [1]
Michel Liquiere [2]
Guy Mineau [3]

9.1 Summary of Progress

To date, we are a subgroup with many possibilities but only a few participants. We welcome other volunteers. Conveniently, each member of the current subgroup has worked on substantially different graph learning problems, and the previous work complements each other. In all cases we believe that these learning mechanisms represent advance facilities for the PEIRCE workbench, that may nonetheless provide the very foundation that is needed for ambitious applications of PEIRCE. In addition to supporting these methods we intend to support other learning mechanisms that are based on first order logic or graph matching.

Each method could benefit by access to the node descriptor generation mechanisms of Method IV [LE91] as well as the associative retrieval algorithms themselves. Further utilities for reading in and out and displaying CG graphs are also critical.

9.2 Learning Structural Concepts from Noisy Examples

Liquiere, M.

Given two sets of conceptual graphs - positive examples and negative examples construct a plausible concept that explains these in CNF. The terms of the CNF refer to the presence or absence of subgraphs of the original graphs. Such routines have wide applicability including biology, geology and chemistry. For example, with this method one may be able to predict side effects in pharmaceuticals or weaknesses in chess positions or in building designs.

The current routines that have been built work in polynomial-time by using traditional statistical classification methods. Instead for PEIRCE we would like to write a routine that works directly with the node descriptors of Method IV retrieval [LE91]. These node descriptors are analogous to the rooted trees used in Liquiere's previous work and should facilitate matching.

Liquiere's work with noise, incomplete or uncertain knowledge will be a critical addition in those real domains such as geology, biology, and vision where this is required.

9.3 Conceptual Clustering as Knowledge Base Structuring

Mineau, G.

Conceptual clustering can improve some knowledge processing tasks as it provides an index over the contents of knowledge objects. This index is built using some comparison mechanism among the knowledge objects (in our case, conceptual graphs) in order to detect similarity among their contents.

[1]Department of Computer and Information Sciences, University of California Santa Cruz Santa Cruz CA 95064 U.S.A, (408)459-2087, ARPANET:levinson@pal.ucsc.edu
[2]Reunion University France, mili@iremia.fr
[3]Laval University Quebec CANADA, mineau@vm1.ulaval.ca

There appears to be a computational problem every learning module has to face sooner or later. Learning with a lot of examples and counter-examples, or with a lot of knowledge objects, will require that the operators it needs to process the graphs are very efficiently implemented. It is vital to the learning module that the operators it uses are the fastest they can be.

9.3.1 Operators Needed by the Learning Module

Consequently, for feasibility reasons, different types of graphs will require different types of operators. The operators normally used by a learning process are the comparison operators: $<$, $>$, $\leq$, $\geq$, $=$ and $\neq$, the formation operators: join, copy, restrict, simplify, the generalize operator: $\uparrow$ and the specialize operator: $\downarrow$. The generalize operator is itself divided into three suboperators: generalize a concept or a relation, erase part of a graph, or graphs.

Since time is of the essence when dealing with learning, it is vital to make certain the implementation of all these operators is optimal. If conflicts arise with other applications or modules, we need to plan a way to determine different versions of the same set of operators.

Also, as we may have embedded graphs, different versions of the same set of these operators would be needed. As a first step, I propose to develop a first version of this set of operators, knowing that other versions will come about, but which is restricted to graphs which have a mapping to a FOL formula (no embedded subgraphs), called first-order conceptual graphs.

9.3.2 User-defined Similarity

The learning module needs to detect similarity among graphs. The similarity criteria are diverse and generally depend on the needs of the application which will use the result of the learning process.

The user should be able to define formally what these criteria are, maybe by giving some procedural attachment to a predefined function SIM(...) which returns the similarity among its parameters. The canonical specifications of this function remain to be clearly established.

Also, since SIM is at the heart of any learning module, it is vital to ensure its optimal coding. Part of its action will call for interaction with data structures encoding background knowledge, type hierarchies, examples, counter- examples, prototypes, individuals, etc. It must access everything the system knows. Submodules automatic communication is thus vital to the PIERCE project. Modules seeking information from other modules need to communicate. At this stage, the learning module links its faith to the CCAT module, where the ontology of the system will be defined and made available.

9.3.3 Learning and Knowledge Acquisition

Learning is done more easily if knowledge acquisition has applied some control over the knowledge objects it acquired. In effect, normalization of conceptual graphs improve the learning process since the SIM function can be simplified. Formal ways of linking knowledge description to knowledge acquisition and learning is needed.

Also, learning methods applied when knowledge is acquired help the knowledge acquisition process. For example, we can detect useless generalizations of graphs, some reoccurring patterns within the data (empirical prototypes), inconsistencies between the formed classes (by a learning process) from the real data, and anticipated classification (as given by external knowledge sources). Consequently, the operators available within the learning module need to be accessible from the knowledge acquisition module. In fact, the knowledge acquisition module should use learning methods to ensure a certain regularity of expression of graphs identified as related (similar), and which will be classified later on. So the learning module must also link its faith to the knowledge acquisition module.

A large body of conceptual graphs are divided into a hierarchical classification structure based on syntactic similarity. These graphs can then be used to suggest classifications of newly seen graphs or to facilitate associative retrieval. Again it will be desirable to see how the node descriptors of Method IV can be used to accelerate Mineau's current algorithms. Liquiere's work with noise may also be an interesting addition,

Related reading includes [MGG89, Min89, MGG90, Min90a, Min90b, Min92].

9.4 Experience-Based Knowledge Acquisition for State-Space Search Problems

Levinson, R.

Most artificial intelligence and machine learning research has required and in many cases encouraged the supplying of knowledge to a system in some form by a human expert or oracle. Such intervention has taken many forms including the supplying of rules, heuristics, control strategies, pre-classified examples, and domain theories. APS (adaptive-predictive search) is an architecture that attempts to minimize the need for human intervention in the creation of an intelligent agent. APS systems view the problem-solving domain within the traditional reinforcement learning framework: as a state space search in which periodic reinforcement (positive or negative) is received from the environment. The unique features of the APS architecture are its uniform pattern-weight representation of control knowledge and its integration of several learning techniques including temporal difference learning, simulated annealing, and genetic algorithms. Despite the lack of human intervention and in stark contrast to most neural net and connectionist systems, APS systems create and produce knowledge in a form (pattern-weight pairs) that is readily understood by human users. Thus the potential for further refinement by the humans remains possible.

APS is applicable to all state-space search problems with large number of states and in which periodic reinforcement is only provided occasionally. Indeed, APS has been applied to a variety of domains including chess, othello, pente, and image alignment. The major testbed is the Morph adaptive pattern-oriented chess system.

For these domains the system is supplied a pattern representation scheme and domain-dependent rules for pattern addition. In ongoing research we are attempting to produce domain-independent syntactic guidelines for pattern addition, thus giving the system further autonomy (e.g. see discussion below on constructive induction).

9.4.1 APS and Knowledge Acquisition: the Role of Human Intervention

How can such a novel architecture as APS, a "weak-method" approach address the problems of knowledge acquisition? Does this research address any of the generally recognized issues of acquiring and using expert problem-solving knowledge? For example: the design of appropriate task-specific knowledge representations, dealing with uncertain and incomplete knowledge, and acquiring knowledge for supporting functions such as explanation.

Indeed, we believe this architecture (and variants) has things to contribute to traditional knowledge acquisition research. To prepare APS for a given task domain the following steps must be taken:

- The problem must be cast as a state-space search problem. State representations, operator specifications, initial state and goals much each be defined.
- A reward structure must be developed for goals and subgoals.
- A pattern representation scheme with appropriate pattern addition rules must be added to the system. (As mentioned above, we are addressing the issue of giving the system more responsibility for this.)

Thus, although more responsibility is being taken by the learning system the expert and knowledge engineer are still very much involved in the problem formulation stage. This is as it should be: *the humans are forced to deal with the problem at the level they are most comfortable with. They are asked to specify which actions are available to the problem-solving system, what the goals are, and how performance should be evaluated.*

One positive feature of APS is that the domain-dependent knowledge is localized (and thus can be changed readily) in the pattern representation language and the pattern addition schemes. Further, the pattern representation language, being uniform, focuses attention directly on the structure of domain knowledge.

An explanation system may be developed by having patterns recall those situations that led to their creation and those instances that have most influenced their weight. Although, this does not lead to a domain-theoretic explanation it does allow the system to provide examples (cases) that have led it to believe what it believes about the current situation. Using this information, the knowledge engineer may be able to make further adjustments to the pattern representation scheme so that further distinctions can be achieved.

9.4.2 Constructive Induction and Inductive Logic Programming

To fully exploit previous experience, the method chosen to represent each experience is critical. An ideal representation is one that uses features that are general enough to occur across many experiences (domain states), but are such that their significance is invariant across these experiences. How to construct such a representation for a complex domain is not obvious. As in other areas of Peirce we wish to stress the importance of giving the system itself more responsibility for *the form* in addition to the content of the learned knowledge.

Towards System Autonomy

An important question, then, is whether an APS system could produce its own abstracted state descriptions for the purpose of enhancing learning efficiency. Here we present some preliminary ideas in this direction. The current progress in constructive induction (CI) research provides encouragement. However, nearly all CI schemes expect to be supplied a domain theory, training instances or a problem specification in some formalism. For example, suppose a system has no problem-specific knowledge beyond the specification of the search problem itself (the initial state, the goal condition and the operator definitions). Tackling exactly this problem Callan and Utgoff [CU91] have shown how useful domain features can be abstracted from a problem specification in predicate calculus. In the case of chess, analysis of the goal would likely produce useful features like whether a king is in check, the number of legal moves for the king, whether a checking piece can be captured or an interposition is possible, etc. Analysis of the operators might produce features about which legal moves each piece has and whether one piece can legally capture another, whether castling is legal and whether a pawn can promote. Barney Pell [Pel91] is applying further "general game-playing" analysis to produce features even more useful than these from the rules of the game.

Though CI schemes such as these take some of the burden off the human developer of the system, being semantic approaches they still require the type of human intervention that it is our research objective to eliminate. We believe that a set of domain-independent syntactic guidelines such as the following will produce a useful start in that direction. Assume that states are represented as conceptual graphs. Guidelines:

- Patterns should be formed for nodes that are (nearly)identical on one or more attributes.
- Patterns should be formed from edge and node sets corresponding to each individual relation. Similar to the radius 1 neighborhoods provided by MethodIV.
- Patterns should be formed from the deleted graph pieces in moving from one state to the next.
- Patterns should be formed from the added graph pieces in moving from one state to the next.
- Patterns that co-occur and intersect in the state description should be combined to form a new pattern based on their conjunction.
- Through abstraction these guidelines may be applied at various granularity levels.

Such guidelines are reminiscent of heuristics for concept discovery used in AM and Eurisko and other machine discovery systems [LBS81], but here are meant to be purely syntactic - based on the original state descriptions. Analysis shows that most of the patterns (and many others) considered by APS systems in a variety of domains would, in principle, be generated from these rules. A deletion mechanism should be able to filter out unneccesary or irrelevant patterns. Also, (for those applications where assistance is desirable), it remains easy for an outside expert to communicate to the system which types of patterns are worth considering - since the desirability of a pattern is itself a pattern.

For arbitrarily complex domains, perhaps the guidelines, themselves, may need to be supplied by a human expert. It is our hope, however, that a universal and focused set of such guidelines can be developed - perhaps with the assistance of the machine itself.

9.4.3 Inductive Logic Programming (ILP)

ILP is a method based on Horn clauses that has been developed for constructing new terms in a logic programming environment. ILP has been successful in the limited framework to which it has currently been applied.

It is our intention to build on the success of ILP by first implementing existing algorithms on Horn clauses represented as conceptual graphs. But then we believe we can take advantage of the unique structuring provided by conceptual graphs to build a more powerful analog to the ILP algorithms.

The ILP framework applies generally to learning problems when the needed features are not necessarily known in advance.

9.4.4 Knowledge Mining

An intelligent system should never be allowed to sit idly. When inactive from the user's point-of-view the system should be busy examining the contents of its knowledge for information/patterns that may be useful to the user and for its own restructuring. Thus we will divide the knowledge mining effort into two subtasks: self-organization and pattern finding.

Self-organization will attempt to improve performance through precompilation of answers to potential user queries. For example, the addition of a specific generalization graph to the database can greatly enhance retrieval time, especially when the added graph is an actual future query to the system.

Pattern-finding will attempt to find structure in the data that may previously have not been known. Such discoveries and "all employees with salaries greater than $50,000 have taken more than 10 days sick leave" and "sulfur and oxygen behave similarly with respect to reaction conditions Y" are possible. The structuring provided by the CG formalism and the Database organization should allow knowledge mining to operate at a much more powerful level than in current relational databases. Indeed, the Method IV organization itself exploits substantial structure discovered in the stored knowledge.

9.4.5 Database Operations Required

There is a tight link between the DB subgroup and the LRN subgroup. In general, the LRN subgroup requires general matching and classifying operations. The LRN subgroup has determined that the following operations are required to support the planned activities:

1. Has graph occurred previously. (Exact matching)
2. Ability to label graphs (e.g. positive example, negative example, noisy example)
3. Find all or immediate generalizations of graph in database.
4. Find all or immediate specializations of graph in database.
5. Form conjunction and negation and disjunction and XOR of database graphs.
6. Find glb in type lattice. Find lub in type lattice.
7. Ability to limit which graphs are considered in matching process (either as a pre or post-filter).
8. Fuzzy case matching - where all cases (graphs) match but to a given degree only.
9. Find the graphs which are closest to query graph using similarity metric supplied by the user. To be specified are types of partial matching:
   - Fully constrained.

- Relations only.
- Concepts only.
- Unconstrained.

10. Ability to allow subgraph for subgraph substitution in matching.
11. Given two cgs return: all maximal common subgraphs all minimal joins.
12. Ability to create additional "links" amongst database graphs (such as analogy, sibling, path)
13. Ability to create new partial orders on fly (e.g. save answer to specific queries.)
14. Given n graphs find the largest graph they all share.
15. Parse: Given graph Q find specific locations and all occurrences of graphs in database in Q. Return binding lists.
16. Production rules: Given graph and graph-to-graph rule, produce new graph.

Further reading includes [CU91, Lev92a, LG92, LBS+92, GL91b, Lev92b, LE91, LS89b, LS89a, LPS90, BC91, Pel91, LBS81].

10

NLP: Natural Language Parsers and Generators

Jonathan C. Oh [1]
Steve Graham
Wen-Jung Hsin
Gi-Chul Yang
Young Bae Choi
Key-Sun Choi [2]
Sung-Hyon Myaeng [3]

10.1 Introduction

A major application of Conceptual Graphs [Sow84] is natural language processing. The NLP subgroup consists of a number of groups. The following is a description of the work in this area by University of Missouri at Kansas City group.

10.2 Lexical Conceptual Grammar

Oh, J.C., Graham, S., Hsin, W-J., Yang, G-C., Choi, Y.B.

We introduce a Natural Language Interface based on the Lexical Conceptual Grammar. Lexical Conceptual Grammar is compared with other grammars such as Lexical Functional Grammar and Cullingford's ERKS [Cul86].

10.2.1 Introduction

While it is not always taken to be one of the major concerns in grammar theories, we feel that the balance between a grammar and its associated lexicon for a language cannot be ignored without the grammar itself rendered impracticable. The idiosyncrasies of lexical items clearly belong in the lexicon and those regularities shared by many lexical items must be formulated as rules so that unnecessary duplications may be avoided inside the lexicon. But how regular will a regularity have to be to be formulated as a rule? This will depend on the application in question. It would be nice to be able to adjust the degree of regularity (regularity threshold) desired for a given application. We could even entertain the notion of automated grammar discovery.

In most natural language based systems such as natural language interfaces, automated consultation programs, automated help systems, information retrieval systems, and question answering systems, natural language input from the user must be converted into some semantic language which can then be used by the reasoning component. The choice of the semantic language has some influence on the kind of grammar to be used. The role of a grammar should be to guide the parsing-interpretation whether in two separate steps or in a single sweeping step.

In this paper we will show how a lexical conceptual grammar will guide the parser-interpreter, how

[1]Department of Computer Science, University of Missouri at Kansas City. oh@vax2.cstp.umkc.edu
[2]Korea Adv Inst Sc & Tech (KAIST), kschoi@cs.kaist.ac.kr
[3]Syracuse University, shmyaeng@mailbox.syr.edu

a lexical conceptual grammar can be written, and then compare our grammar with Kaplan-Bresnan's lexical-functional grammar [KB82].

10.2.2 Lexical Parsing and Interpretation

First we will try to demonstrate how a NL system will parse a sentence. For definiteness, we will use Cullingford's ERKS system [Cul86] and illustrate how it works using the example sentence below:

1] Olivia punched Muhammed in the nose.

In order to parse/interpret this sentence, we need a dictionary for the content words and some directives for the so-called functor words. The concepts for the content words, punch and nose in the example sentence are given by the Lisp function 'def-wordsense' (refer to the appendix for the actual wordsenses and parse directives for the words in that example sentence). This function not only associates the surface form with its word senses or concepts for the word but also specifies the constraints for various slot-fillers and what group, if any, of slot fillers are to be identical. In the case of 'punch', the actor and to or patient are expected to be high-animate beings and the actor (Muhammed) and the owner of the hand that is used in the punching act are one and the same person. If the conjugation of the surface dictionary form is not regular, that also is specified by this function. Although the dictionary is represented procedurally by an insertion function, one can easily think of the lexicon in a declarative format and implement it accordingly.

For a functor word, directives are given as to how the word contributes to the overall conceptual structure, which constitutes the meaning of the sentence as a whole. Each word has one top request (topreq), a production for introducing into the temporary memory used for parsing/interpretation the conceptual structure corresponding to its meaning. This could be handled by 'def- wordsense' so that either the conceptual structure corresponding to the word in question or some directive as to how the use of the word contributes to the overall structure can be given in the lexicon as the meaning of the word. The parser can be written so that it can use the information in constructing the overall semantic structure for the sentence containing the word. Purely for the sake of efficiency, Cullingford [Cul86] opts for the strategy used here. So for the definite article 'the', no wordsense is provided in the lexicon, instead a directive is provided how to modify the overall semantic structure when the parser encounters the article. The directive says to look for the concept for a noun group that is associated with this article and fill in the ref (reference) slot with the value 'def.'

Every word in a sentence including content words has an 'adictdef' a directive as to what to add (topreq) to the temporary memory mentioned in the previous paragraph and how to fill in certain gaps or fillers (gapreq) under certain conditions. Cullingford's ERKS [Cul86] translates natural language expressions directly into semantic language expressions without going through a separate syntactic parsing step. Thus the conditions for filling in gaps are specified in both syntactic and semantic terms simultaneously. If one feels that utilization of any other types of information such as pragmatic cues might be helpful, this is the spot for it.

For 'punch', the first gapreq is to pick up the actor slot-filler. You look for a concept for the noun group which occupied either an active subject position or an object of 'by' in a passive sentence or any other syntactic position that is construed as 'in-act-spot'. If you find such a concept, the parser also checks its semantics as well, i.e. whether the semantic type of the concept is related to the type 'person' either as its subtype or as its supertype (could-be-person-p). Only when both this syntactic and semantic conditions are satisfied the concept is picked up and used to fill in the actor slot (gap). Regardless of whether we find such a concept we move down to the gapreq until we go through the list of gapreqs. If any of the gapreqs does not fire, it remains for further consideration until it gets killed off and deleted from the temporary memory.

The conceptual structure that is constructed by the parser is

```
(propel          actor HUM0
    obj       (bpart bptype (hand)   partform (fist)
                             partof  HUM0)
    to (physcont part (bpart  bptype (nose)  partof HUM1)))
HUM0:
(person   gender (fem)   persname (Olivia))
```

```
HUM1:
(person   gender (masc)   persname (Muhammed))
```

Notice that all the fillers (gaps) of the conceptual structure for the verb 'punch' are filled in appropriately. Also all the constraints mentioned under the lexical entry for the word are satisfied. The referential equivalencies are also ensured in this conceptualization.

The position of Cullingford's system [Cul86] is an extreme one in the sense that all syntactic information is implemented within the lexicon and there is no grammar rule. This allows for flexible treatment of sentences, even ungrammatical ones. Unlike two-step parser where all unparsable strings get discarded and never given to the semantic interpreter, one-step systems like Cullingford's attempt [Cul86] to give semantics for any word string. The drawback of the latter type systems is that they tend to be less disciplined. It is difficult to have a disciplined/principled way of dictionary writing and to avoid case by case improvisations. Also by necessity the dictionary is saturated with redundancies. Many words behave similarly syntactically or semantically. The gapreqs for filling in actor slots will be similar for verbs of the same semantic and therefore syntactic type.

One can certainly conceive of a computer program that peruse 'adicdefs' for words and extract similarities and as its pattern is seen to repeat itself so many times the pattern gets identified and named by the program. The dictionary can be made that much more readable and simpler by using the name rather than the whole pattern. This process is the categorization process in grammar writing. Other structural information that is used in syntactic conditions can be formulated as syntactic rules.

10.2.3 Lexical Functional Grammar

In this section we discuss the lexical functional grammar, for the lexical conceptual grammar shares many basic principles with it. In a lexical functional grammar, the correspondence between syntax and semantics is captured in two stages. Lexical entries specify a direct mapping between semantic arguments and surface functional configurations. Syntactic rules relate these grammatical functions to particular syntactic configurations. Every sentence is assigned two levels of syntactic description; constituent structure (c-structure) and functional structure (f-structure). The c-structure is the familiar syntactic phrase structure in a generative or transformational grammar. But it is the f-structure that we obtain from c-structure that is used as input to the semantic interpreter. The c-structures are defined in terms of syntactic categories, terminal strings, and their dominance and precedence relationships.

The f-structures are defined in terms of grammatical function names, semantic forms for content words, and feature symbols. For our example sentence used in the previous part, the following phrase structure rules will assign the given c-structure to the sentence (also see FIG. 10.1):

```
S ->      NP              Vimpact        NP            PP
        (^subject)=v                   (^object)=v    (^loc)=v

NP -> (Det)  Adj*  N
```

The f-structure for this sentence might look like FIG. 10.2. Such F-structures are derived from C-structures via functional- descriptions (f-descriptions) of the sentence.

10.2.4 Lexical Conceptual Grammar

Consider the following group of verbs: accompany, follow, guide, lead, precede, and track. Both the subjects and objects of these verbs are semantically of what we call locomotive type, type of moving objects. Both the subjects and objects move or at least intend to move toward the same goal or direction. Instead of specifying the word senses of these verbs separately, we will label the word sense template for this group of verbs *parallel-motion*, which we enter into the *template depository*. Each entry in the template depository carries not only the template for the word sense but also what we might call *meaning postulates*, i.e. common semantic features shared by the words grouped together under the label.

1. The skinny Indian tracked down the fugitive.

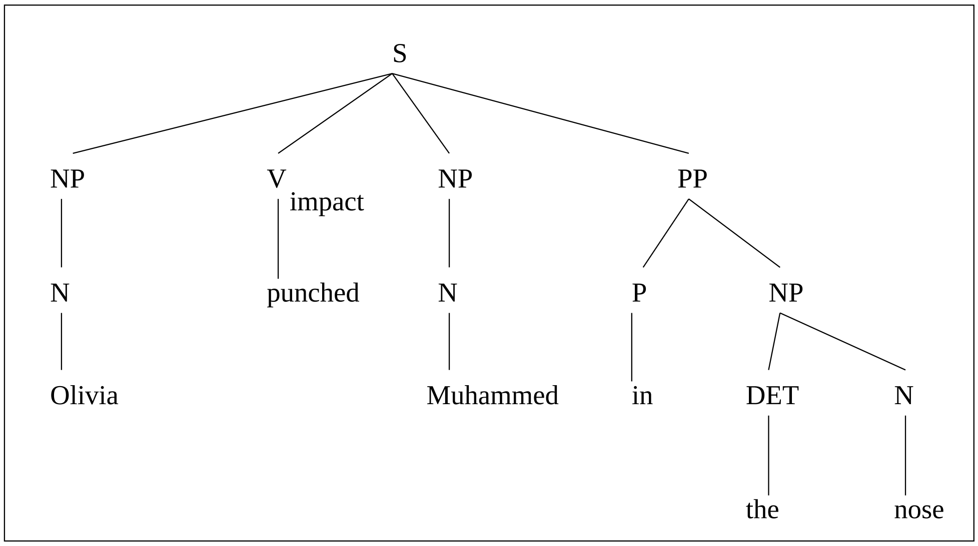

FIGURE 10.1. Phrase structure of "Olivia punched Muhammed in the nose"

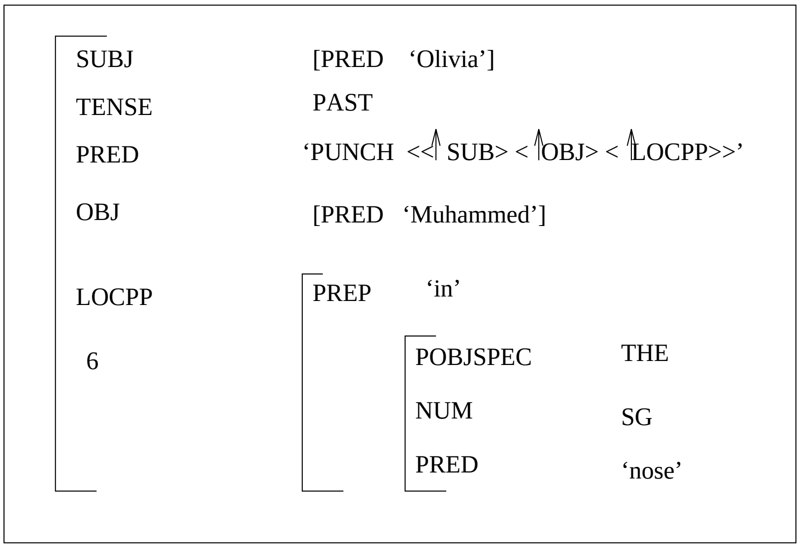

FIGURE 10.2. F-structure for "Olivia punched Muhammed in the nose"

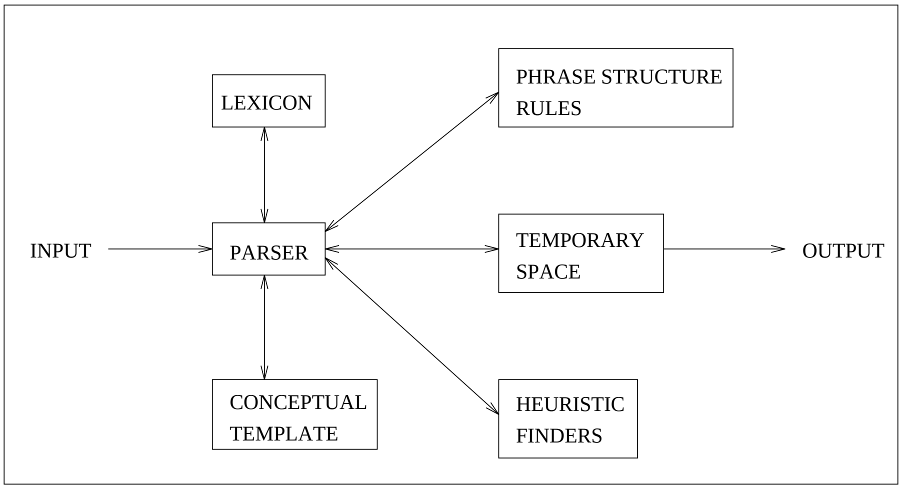

FIGURE 10.3. Overview of an NL understander

```
template  parallel-motion

  [track]-
        (agnt)->[[Indian: lambda]->(state)->[skinny]: #x]
        (obj)->[fugitive:#y]
        (dir)
        (goal)


  meaning postulate:  parallel-motion

        [move]->(agnt)->[T: #x]
              ->(dir)->[LOCN:#z]
        [move]->(obj)->[T:#x]
              ->(dir)->[LOCN:#z]


S ->    NP1     Vparallel-motion        NP2
        agnt                            obj

NP -> (DET) (ADJ)* N (REL CLAUSE)*
```

In parsing a sentence, a heuristic plan called v-finder searches through each word and proposes main verbs for the sentence. Initially, the v-finder will consider each word and looks up the category information for that word and if one of its senses has the category verb, that word is tentatively considered a candidate. After further consideration of the context, some drop out and for each of the surviving candidates, we consider both the conceptual template and phrase structure rule(s) and construct the conceptual graph in a temporary space. With the example sentence, *NG-finder* will look for NP1 and NP2 and their corresponding conceptual structure fragment. Many times such fragmental analysis fails resulting in the abandonment of the original surviving hypothesis. Eventually, if the parse succeeds, there will remain only the desired conceptual graphs representing the meanings of the sentence. These successful graphs will be output at a signal from the parser.

Schematically the NL understander looks like FIG. 10.3.

The lexicon will carry under each entry the category, semantic type for each word sense of that entry and other idiosyncratic features on all levels, morphological, lexical, syntactic, semantic, and pragmatic.

10.2.5 Generation

The Generation subgroup at UMKC has implemented the first version of language generator from CG to English in C++. This language generator closely follows [Sow84] Section 5.4. Because there are many relation types, our goal in the first version is to get a working implementation that allows for future expansion. Currently, we are working on providing a complete language generator for any conceptual graph. We hope to see a much more complete package in the next version. We are also working on designing a new algorithm for generation. The new algorithm will use U-Form [YCO92] as its internal structure. The U-Form eliminates the necessity of explicit arc and contains the natural way of extracting subgraphs. This fact will help to find the lexical items of corresponding subgraphs easily. We will compare the performance of two algorithms after implementation.

10.2.6 Conclusions

LCG depends on a secure balance between Lexicon and Phrase- Structure Rules and also between Phrase-Structure Rules and Heuristic Finders. Lexicon, Phrase-Structure Rules, and Conceptual Template are three components that help us avoid the limitations of the ERKS-like systems. LCG will be able to interpret incomplete and pseudo-grammatical sentences as well as complete/grammatical sentences

The NLP group at UMKC is working on Lexicon, Conceptual Template, Phrase-Structure Rules, Heuristic Finders, and Parser for English and Korean at this time. This is to be used for natural language interface in an automated consultation system based on a conceptual graph theory.

For further reading also see [Dix91, Sow91, Win83].

Appendix A

```
(def-wordsense wsPUNCH1
  surface-form (punch)
  ;;; punch is regular; thus no pastform/partfom otherwise those forms
  ;;; will be given here
  ws-structure
  (propel-f
        actor   (person-f)
        obj             (bpart-f        bptype (hand))
        to              (physcont-f     part (bpart-f   partof (person-f)))
        time    (times-f)
        mode    (modes-f))
  constraints
  (((actor) couldbe-hianimate-p)
   ((to val partof) couldbe-hianimate-p))
  equivs
  (((actor)(obj partof)))
  )

(adictdef punch
  pos v
  topreq
  ((test (:= topcon (add-word-con wsPUNCH1)))
   (actions
    ;add time/mode info
    (addtime topcon (make-special-structure '(:pres)))
    (addmode topcon (make-special-structure '(:t)))
    (activate
     ;;;;;;;pick up the actor
     (gapreq
      (test (:= actorcon (if-clavail topcon (g)
                              (and (in-act-spot g topcon)
                                   (semfeature g 'couldbe-person-p)))))
      (actions
       (fill-gap '(actor) topcon actorcon)))
     ;pick up the bodypart; don't lose the possessor of the bodypart
     (gapreq
      (test (:= bpartcon (if-clavail topcon (g)
                              (and (prepconst g '(in on upside))
                                   (semfeature g 'couldbe-bpart-p)))))
      (actions
       (make-unavail bpartcon)
       ;merge-con makes a merger of two "congruent" concepts: i.e., based
       ;        on same type, with no conflicting fillers
```

```
      (fill-gap '(to val) topcon (merge-con (grf '(to val) topcon)
                                            (grf '(mynom) bpartcon)))))
     ;pick up the possessor of the bodypart
     (gapreq
      (test (:= bpartowner (if-clavail topcon (g)
                               (and (in-obj-spot g topcon)
                                    (semfeature g 'couldbe-person-p)))))
      (actions
       (fill-gap '(to val partof) topcon bpartowner))))
    )))

(def-wordsense wsNOSE1
  surface-form (nose)
  ws-structure
  (bpart-f bptype (nose) partof (person-f))
  )

(adictdef Muhammed
        pos noun
        topreq
        ((test (:= topcon (add-con (person-f persname (Muhammed)
                                             gender (masc))
                                   ()())))
        (actions
          (activate
           ;treat following unknown as a surname
           (gapreq
            (test (:= unkcon (if-ngavail topcon (g)
                               (and (foll g topcon)
                                    (semfeature g 'unknown-p)))))
            (actions
             (fill-gap '(surname) topcon unkcon)))))))
 )

(adictdef in
  pos prep
  topreq
   ((test (:= topcon (add-con (prepconst-f prep (in) mynom (())
                                           canbuild (())) ()()))))
    (actions
     (activate
      ;;;;;;;;in a place
      (gapreq
       (test (:= placecon (if-prepclavail topcon (g)
                               (semfeature g 'couldbe-place-p))))
       (actions
        (kill rest-of-pool)
        (fill-gap '(mynom) topcon placecon)
        ;canbuild a p-config fragment
        (fill-gap '(canbuild) topcon (make-cd '(inside part (()))))
        (fill-gap '(canbuild part) topcon placecon)))

      ;;;;;;;;;;in April
      (gapreq
       (test (:= durcon (if-prepclavail topcon (g)
                               (semfeature g 'couldbe-timdur-p))))
       (actions
        (kill rest-of-pool)
        (fill-gap '(mynom) topcon durcon)
        ;canbuild an abstime
        (fill-gap '(canbuild) topcon (make-cd '(during part ())))
        (fill-gap '(canbuild part) topcon durcon)
         ;if topcon is still avail at clp, attach to nearest clp
        (activate
         (gapreq
          (test (null (get topcon '*avail)))
          (actions (kill rest-of-pool)))
         (gapreq
          (test (:= clpcon (if-clavail topcon (g)
                                 (and (foll g topcon)
                                      (semfeature g 'clp-p)))))
          (actions
           (kill rest-of-pool)
           (activate
            (gapreq
             (test (:= concon (if-clavail topcon (g)
                                (and (prec g clpcon)
                                     (semfeature g 'concept-p)))))
             (actions
              (make-unavail topcon)
              (fill-gap '(abstime) concon (grf '(canbuild) topcon)))))))))
```

```
      ;;;;;;;;;in the army, build an a-config fragment
      (gapreq
       (test (:= orgcon (if-prepclavail topcon (g)
                                    (semfeature g 'couldbe-org-p))))
       (actions
        (kill rest-of-pool)
        (fill-gap '(mynom) topcon orgcon)
        (fill-gap '(canbuild) topcon (make-cd '(member part (()))))
        (fill-gap '(canbuild part) topcon orgcon)))

      ;;;;;;;;in ¡nominal¿
      (gapreq
       (test (:= nomcon (if-prepclavail topcon (g)
                                              ; accept an entity or
                                              ; nominalized episode
                                              ; (e.g., "in a course")
                                    (semfeature g 'nomconcept-p))))
       (actions
        (kill rest-of-pool)
        (fill-gap '(mynom) topcon nomcon)
        ;if a nominalized concept "in a course", and in-phrase doesn't
        ;disappear, create during-con at clp
        (activate
         (gapreq
          (test (null (get topcon '*avail)))
          (actions (kill rest-of-pool)))
         (gapreq
          (test (:= clpcon (if-clavail topcon (g)
                                    (and (foll g topcon)
                                         (semfeature g 'clp-p)))))
          (actions
           (activate
            (gapreq
             (test (:= concon (if-clavail topcon (g)
                                 (and (not (eq g topcon))
                                      (semfeature g 'couldbe-episode-p)))))
             (actions
              (:= durcon (make-cd (during-f)))
              (fill-gap '(con1) durcon (copy-con concon))
              (fill-gap '(con2) durcon topcon)
              (fill-gap () concon durcon)))))))
          ))
        ))))

(adictdef nose
  pos noun
  topreq
  ((test (:= topcon (add-word-con wsNOSE1)))
   (actions
    ())))

;simple "the"
(adictdef the
  pos adj
  topreq
  ((test (:= topcon (add-con '(def) nil nil)))
   (actions
    (activate
     (gapreq
      (test (:= nomcon (if-ngavail topcon (g)
                            (and (fon 'end-ng)
                                 (foll g topcon)
                                 ;want the actual concept
                                 g))))
      (actions
       (fill-gap '(ref) nomcon topcon))))
   )))
```

11

ISE Information Systems Engineering Subgroup

Peter Creasy [1]
Alex Bejan [2]
Gary Berg-Cross [3]
Gerard Ellis [4]

11.1 Introduction

The large investment in, and complexity of, enterprise information systems compels the development of better tools to manage the design process and integration of enterprise subsystems. A number of avenues exist for conceptual graphs to play a significant role in the information systems future. For example, ANSI has chosen conceptual graphs as the normative language to integrate enterprise models central to information systems. The Information Systems Engineering subgroup intends to explore these roles.

In particular some of the immediate aims are to provide mappings between a number of schema languages (such as NIAM, ER and Express) and conceptual graphs and to describe these languages in terms of conceptual graphs. Other areas of interest include schema/model integration, the use of ontologies [BCH92] (which interfaces with the CCAT group) and incorporating conceptual graphs into CAiSE tools.

These will require developing a number of tools, some of which will be available through other groups, but the more specialised will need to be developed or be specialisations of more general tools (e.g. schema integration will use graph joining).

The following is the first of many projects in this subgroup.

11.1.1 A Conceptual Graphs approach to Conceptual Schema Integration

Creasy, P., and Ellis, G.

In order to facilitate the effective and economic production of information systems, formal modelling tools (schema or modelling languages) and associated methodologies are used. The modelling process, which aims to capture as much as possible of the Universe of Discourse (UoD), is central to the implementation of database systems.

Several of these tools (e.g. NIAM [NH89]) have been used commercially for a number of years. These languages capture the static aspects of the UoD, providing the ability to represent a few (but important) constraints, and permit a translation into a relational database schema. There have been attempts to expand these languages to capture more of the UoD, in particular to capture the dynamic aspects. However no tool has been as successful as the static languages.

There is a growing tendency and need in information systems to model applications using a number of different modelling languages, with a subsequent integration of these partial models into an enterprise

[1] Key Centre for Software Technology, Department of Computer Science, University of Queensland, Brisbane, QLD 4072 Australia, peter@cs.uq.oz.au

[2] IBM Kingston NY, algae@kgnvmy.vnet.ibm.com

[3] Advanced Decision Systems, garybc@chesapeake.ads.com

[4] University of Queensland, ged@cs.uq.oz.au

model. One possibility for achieving this integration would be to provide a mapping between each pair of these modelling languages and to map them to one language. Alternatively, a core conceptual language could be used as a standard translation intermediary, thereby reducing the total mappings required from $O(n^2)$ to $O(n)$. This language could be the centre of a CAiSE (computer aided information systems engineering) tool with a mapping provided between each modelling language and the core language. Such CAiSE tools are becoming widely used in information systems design. Each tool generally uses only a single modelling language. The integration of models from different languages is a major unsolved problem in CAiSE tools.

This idea of having a core language corresponds to the ANSI (American National Standards Institute) proposal for the IRDS (Information Resource Dictionary System) conceptual schema. A number of modelling languages have been examined as candidates for the IRDS normative language. The IRDS describes and manages an organisation's information system resources. The normative language is part of the IRDS layered framework. In this framework, each layer provides a higher level of abstraction in terms of the layer directly below it. Within the normative schema layer the normative language can define all constructs that would be used in any modelling schema language. Thus the normative language "will be rich enough to include a superset of the semantics of all existing schema languages" [ANS91].

ANSI has approved the ANSI Task Group X3H4.6 proposal to use conceptual graphs for the normative language. Consequently it has become critical to solve the problem of using conceptual graphs as a conceptual schema language. The choice was made as conceptual graphs are/have:

- highly readable graphical language
- linear form
- great expressive power
- easily mapped to and from natural language
- well-defined theoretical base
- fifteen year history of research
- world-wide user community
- commercial applications in production use
- suitable for iterative refinement and production of less detailed specifications using abstractions - achieved using lambda abstraction
- represent first and higher order statements, with or without quantifiers.

This project concentrates on the conceptual schema aspects of the normative language. As NIAM is a language which encompasses the semantics of most static conceptual schema languages we focus on it to establish the requirements of a normative language in conceptual schema terms. The techniques developed can then be extended to other modelling languages such as ER.

Conceptual graphs are much more general than NIAM. In particular it is much easier to express the dynamic aspects of a UoD with conceptual graphs. However if conceptual graphs are to be used as a normative language they must be able to express NIAM's static constraints relatively easily.

A few other papers have previously shown mappings between ER and conceptual graphs but these have avoided any examples that could not be simply represented in conceptual graphs. We have previously pointed out [Cre90, Cre88] some of the problems of using the basic form of conceptual graphs as a conceptual modelling language, due to the relatively complex nature of the specialised constraints required. This project proposes a number of abstractions to enable the constraints provided by NIAM to be expressed in a readable form using conceptual graphs.

One of the strategies for conceptual design of large systems is to separately develop a number of subschemas (or views) and integrate these into an enterprise schema. There have been long-standing problems automating this integration process (e.g. name, type and constraint conflicts). We contend that by integrating the schemas at the conceptual graphs level a number of these problems can be overcome. This will increase the automation of this process.

The project aims to demonstrate the feasibility of a number of different aspects of the CAiSE/ANSI paradigm. Firstly, we intend to show a mapping between conceptual graphs and NIAM and secondly, we intend to develop the conceptual graph join techniques to permit schema integration. These aspects are covered in the succeeding sections.

Translator: NIAM to Conceptual Graphs

A mapping from NIAM to conceptual graphs which only accounts for the UoD syntax is fairly simple. However to map NIAM's (and ER's) major constraints (e.g. mandatory role and uniqueness constraints) is somewhat more difficult. While they can be mapped to first order logic, this defeats some of the prime requirements of the normative language - readability and reverse mapping.

We have thus proposed equivalent forms in conceptual graphs with abstractions defined in terms of the graphs themselves. These also cover the standard ER constraints. In addition we have already shown that these abstractions can be used to map most of NIAM's other more obscure constraints. The development of abstractions for the remainder of NIAM's constraints is continuing. Techniques which will permit a reverse mapping are being developed.

These mappings will become the basis for a translator from NIAM to conceptual graphs and a partial translator from conceptual graphs to NIAM.

Interactive Maximal Joining of Conceptual Graphs

The schema integration is to be done at the conceptual graph's level and by utilising conceptual graph joins. Some of the theory for maximal joins between graphs has been shown in the original theory. However this only covers maximal joins for generic concepts. We need to extend the theory to cover joins using quantifier referents. The theory for this is currently being done. It is proposed to develop this theory is association with the PEIRCE DB conceptual graphs database subgroup.

Schema integration is an interactive process. It involves matching compatible structures from distinct schemas and using these as joining points. It normally is a mostly manual process. We propose to make it a semi-automatic process by using what we call adaptive maximal joins. The maximality is determined interactively.

We have proposed [MC92] a two level approach to information systems design using conceptual graphs. In this approach the informal (but semantically undefined) relationships of NIAM would be defined in terms of the semantically richer conceptual graphs. We can incorporate this two level approach into the translation process and thus more readily automate the schema integration process.

An advantage of using a normative language is that the schema integration process only has to be designed an implemented in that language.

Summary

The project is a step towards the ANSI IRDS goal of using conceptual graphs as the normative language while permitting their use in a CAiSE tool that can integrate conceptual schemas.

12

VISION: Machine Vision

Brian J. Garner [1]

12.1 Introduction

The objectives of the Vision Subgroup may be broadly definede as Object Definition (symbolic description of an object class), concept classification and image understanding.

New entrants to the field are welcome!

The research mission is intended to not only provide tools for vision research but to provide system support integration for a wide range of CG environments, based on PEIRCE standards, ranging from multimedia presentations to animation graphics. A Generalised User Interface(GUI) platform for application kernels is also in prospect.

12.2 Progress

The initial research of the Deakin CG Group has focussed on interactive classification. This work has been published [GT86, GT87, GLT87, GT88, GL91a, GTL+92] and the software (in MU Prolog) is available to PEIRCE members for unrestricted use! (contact brian@deakin.oz.au).

[1]Division of Computing and Mathematics, Deakin University, Geelong, Australia 3217. **Email:** `brian@deakin.oz.au` TEL: (6152) 271383 FAX: (6152) 272028

13

Bibliography

13.1 References

[AKBLN89] Hassan Aït-Kaci, Robert Boyer, Patrick Lincoln, and Roger Nasr. Efficient implementation of lattice operations. *ACM Transactions on Programming Languages and Systems*, 11(1):115–146, January 1989.

[AKP91] Hassan Aït-Kaci and Andreas Podelski. Towards a meaning of life. In Jan Maluszyński and Martin Wirsing, editors, *Proceedings of the 3rd International Symposium on Programming Language Implementation and Logic Programming (Passau, Germany)*, pages 255–274. Springer-Verlag, LNCS 528, August 1991.

[ANS91] ANSI. Task group x3h4.6 working paper on the conceptual schema. Technical report, ANSI, 1991.

[BC91] A. L. Birnbaum and G. C. Collins, editors. *Proceedings of the Eighth International Workshop on Machine Learning*. Morgan Kaufmann, 1991.

[BCH92] Gary Berg-Cross and John Hanna. Exploration of conceptual structures in support of data modeling, concepts, contextually distributed knowledge and the role of cognitive processes. In [Pfe92], pages 157–166, 1992.

[Cre88] Peter Creasy. An information systems view of conceptual graphs. In *Proceedings of the International Computer Symposium*, pages 833–838, Taipei, December 1988.

[Cre90] Peter Creasy. Conceptual graphs from a knowledge systems viewpoint. In *Australian Joint Artificial Intelligence Conference*, volume 406 of *Lecture Notes in Computer Science*, pages 256–271. Springer-Verlag, 1990.

[CU91] J. Callan and P. Utgoff. A transformational approach to constructive induction. In *Proceedings of the Eighth International Workshop on Machine Learning*, pages 122–126. Morgan Kaufmann, 1991.

[Cul86] R. E. Cullingford. *Natural Language Processing*. Rowman and Littlefield, 1986.

[Dix91] R. M. W. Dixon. *A New Approach to English Grammar on Semantic Principles*. Oxford, 1991.

[EG90] Peter Eklund and Laurie Gerholz, editors. *Proceedings of the Fifth Annual Workshop on Conceptual Graphs*, ISBN 91-7870-718-8, Boston&Stockholm, 1990. Linköping University.

[Ekl88] Peter Eklund. Using conceptual structures for knowledge-base systems development. Master's thesis, 1988. Master of Philosophy thesis.

[Ekl91] Peter Eklund. *An Epistemic Approach to Interactive Design in Multiple Inheritance Hierarchies*. PhD thesis, Linkoping University, Department of Computer and Information Science, 1991.

[EL92] Gerard Ellis and Robert A. Levinson, editors. *Proceedings of the First International Workshop on PEIRCE: A Conceptual Graphs Workbench*, 1992. Held in association with the Seventh Annual Workshop on Conceptual Graphs, Las Cruces, New Mexico, July 8-10.

[Ell92a] Gerard Ellis. Compiled hierarchical retrieval. In [NNGE92], pages 285–310. Ellis Horwood, 1992.

[Ell92b] Gerard Ellis. Compiling conceptual graphs. *IEEE Transactions on Knowledge and Data Engineering*, 1992. to appear.

[ENYG89] John W. Esch, Timothy E. Nagle, Morgan L. Yim, and Laurie L. Gerholz. CONSTRUCT. In [NN89], 1989.

[Esc89] John W. Esch. Conceptual graph linear forms. In [NN89], 1989.

[FKS86] Christer Fernstrom, Ivan Kruzela, and Bertil Svensson. *LUCAS Associative Array Processor*. Springer-Verlag, New York, 1986.

[GL91a] B. J. Garner and D. Lukose. Expert advisor utilising goal interpretation methods. In *Proceedings of the World Congress On Expert Systems*, Florida, USA, December 16-19 1991.

[GL91b] Jeffrey Gould and Robert A. Levinson. Method integration for experience-based learning. In R. Michalski, editor, *Proc. International Workshop on Multi-Strategy Learning*, pages 378–393, Harpers Ferry, West Virginia, November 7-9 1991. George Mason University Press. Also appears as Technical Report UCSC-CRL-91-27. Submitted to Machine Learning Journal.

[GL92] Jeffrey Gould and Robert A. Levinson. Experience-based adaptive search. In R. Michalski and G. Tecuci, editors, *Machine Learning IV: A Multi-Strategy Approach*. 1992. to appear.

[GLT87] B. J. Garner, D. Lukose, and E. Tsui. A batch processor for the knowledge base editor. Technical Report 87/4, Xerox AI Laboratory, Division of Computing and Mathematics, Deakin University, February 1987.

[GT86] B.J. Garner and E. Tsui. An extendible graph processor for knowledge engineering. In J.F. Gilmore, editor, *Applications of AI III*, pages 415–433, 1986. SPIE Proc. 635.

[GT87] Brian J. Garner and Eric Tsui. A self-organizing dictionary for conceptual structures. In J. F. Gilmore, editor, *Proceedings of the Conference on Applications of Artificial Intelligence*, SPIE Proc. 784, pages 356–363, 1987.

[GT88] B. J. Garner and E. Tsui. General purpose inference engine for canonical graph models. *Knowledge Based Systems*, 1(5):266–278, 1988.

[GTL+92] B. J. Garner, E. Tsui, D. Lui, D. Lukose, and J. Koh. Progress on an extendible graph processor for knowledge acquisition, planning and reasoning. In [NNGE92]. Ellis Horwood, 1992.

[H+91] Tetsuga Higuchi et al. IMX2: A parallel associative processor. *ACM SIGARCH*, 19(3):22–33, September 1991.

[Har92] S. Hardy. The POP programming environment. Technical Report Cognitive Science Memo, CSRP 005, University of Sussex, 1992.

[ISO90] ISO/IEC. *Portable Operating System Interface (POSIX) - Part 1: System Application Program Interface (API) [C Language]*. Publication Sales, IEEE Service Center, P.O.Box 1331, 445 Hoes Lane, Piscataway NJ 08855-1331, U.S.A., telephone (201) 981-0060., 1990. Reference number ISO/IEC 9945-1:1990(E), IEEE Std 1003.1-1990.

[Joh78] S. C. Johnson. Yacc — yet another compiler-compiler. Technical Report 32, Bell Laboratories, July 1978.

[K+89] Peter Kogge et al. Vlsi and rule-based systems. In Jose G. Delgado-Frias and Will R. Moore, editors, *VLSI for Artificial Intelligence*. Kluwer Academic Publishers, Boston, 1989.

[KB82] R. M. Kaplan and Joan Bresnan. Lexical functional grammar: A formal system for grammatical representation. In Joan Bresnan, editor, *In The Mental Representation of Grammatical Relations*. MIT Press, 1982.

[Knu84] Donald E. Knuth. *The TEXbook*. Addison-Wesley, 1984.

[Kog91] Peter M. Kogge. *The Architecture of Symbolic Computers*. McGraw-Hill, New York, 1991.

[KS90] Shmuel Kliger and Ehud Shapiro. From decision trees to decision graphs. In S. Debray and M. Hermengildo, editors, *Proceedings of the North American Conference on Logic Programming*, pages 97–116. MIT Press, 1990. Austin, Texas, 29 October-1 November.

[Lam86] Leslie Lamport. LATEX: *A Document Preparation System*. Addison-Wesley, 1986.

[LBS81] P. Langley, G. L. Bradshaw, and H. A. Simon. BACON.5:the discovery of conservation laws. In *Proceedings of Int'l Joint Conference on Artificial Intelligence*. AAAI, 1981.

[LBS+92] Robert A. Levinson, B. Beach, Richard Snyder, T. Dayan, and K. Sohn. Adaptive-predictive game-playing programs. *Journal of Experimental and Theoretical Artificial Intelligence*, 1992. to appear. Also appears as Technical Report UCSC-CRL-90-12, University of California, Computer Research Laboratory, Santa Cruz, CA 95064.

[LE91] Robert A. Levinson and Gerard Ellis. Multi-level hierarchical retrieval. In [Way91], pages 67–81, Suny-Binghamton, July 1991. Also, to appear in special issue of Knowledge-Based Systems Journal on Conceptual Graphs.

[Lev92a] Robert A. Levinson. APS: An architecture for experience-based knowledge acquisition. In *Proc. Workshop on Computational Architectures for Supporting Machine Learning and Knowledge Acquisition at Machine Learning Conference*. Morgan Kaufmann, 1992. Aberdeen, Scotland, July 4.

[Lev92b] Robert A. Levinson. Pattern associativity and the retrieval of semantic networks. *Computers & Mathematics with Applications Journal*, 23(2):573–600, 1992. Part 2 of special issue on Semantic Networks in Artificial Intelligence, Fritz Lehman, Guest Editor, Pergamon Press. Also to appear in the book "Semantic Networks for Artificial Inteligence", Pergamon Press. Also appears as Technical Report UCSC-CRL-90-30, University of California, Computer Research Laboratory, Santa Cruz, CA 95064.

[LG92] Robert A. Levinson and Jeffrey Gould. Experience-based adaptive search. In Ryzard Michalski and George Tecuci, editors, *Machine Learning 4: A Multi-Strategy Approach*. Morgan Kaufman, 1992. Presented at Workshop on Machine Learning at Ninth Biennial Conference of the Canadian Society of Computational Studies (CSCSI), Vancouver, Canada, May 12.

[LPS90] Michel Liquiere, P. Pingand, and J. Sallantin. The practical application of machine learning to the prediction of tertiary structure of proteins. In *Advances in Biomedical Computing and Human Reasoning*. 1990.

[LS] M. E. Lesk and E. Schmidt. Lex — a lexical analyzer generator. In *Unix Programmer's Manual, Tenth Edition*, Bell Laboratories, 1989.

[LS89a] Michel Liquiere and J. Sallantin. Inne: A structural learning algorithms for noisy data. In *EWSL*, 1989.

[LS89b] Michel Liquiere and J. Sallantin. Inne: Induction in networks. In *Proc. IEEE International workshop on Tools for AI*, Fairfax, USA, October 1989.

[LS91] Robert A. Levinson and Richard Snyder. Adaptive pattern oriented chess. In *Proceedings of AAAI-91*, pages 601–605. Morgan-Kaufman, 1991.

[M+92] Dan Moldovan et al. SNAP: Parallel processing applied to AI. *Computer*, 25(5):39–49, May 1992.

[MC92] Bernard Moulin and Peter Creasy. Extending the conceptual graph approach for data conceptual modelling. *Data and Knowledge Engineering*, 8(3):165–194, 1992.

[MGG89] Guy Mineau, J. Gecsei, and R. Godin. Improving consistencies within knowledge bases. In M. Shader and W. Gaul, editors, *Knowledge, Data and Computer-Assisted Decisions*, volume F 61 of *NATO ASI Series*, pages 49–65. Springer-Verlag, 1989.

[MGG90] Guy Mineau, J. Gecsei, and R. Godin. Structuring knowledge bases using automatic learning. In *Proc. of the 6th Int. Conf. on Data Engineering*, pages 274–280. IEEE Computer Society Press, 1990.

[Min89] Guy Mineau. Induction on conceptual graphs: Finding common generalizations and compatible projections. In [NN89], 1989.

[Min90a] Guy Mineau. Normalizing conceptual graphs. In [EG90], 1990.

[Min90b] Guy Mineau. *Structuration de bases de connaissances par generalisation.* PhD thesis, Universite de Montreal, June 1990. These de Doctorat (Ph.D.).

[Min92] Guy Mineau. Acquisition d'objets structures destines a la classification symbolique. In *Actes des 1eres Journees Francophones sur l'Apprentissage et l'Explicitation des Connaissances*, Dourdan, France, April 1992.

[NH89] G M Nijssen and T. Halpin. *Fact-Based Conceptual Schema and Relational Database Design Using NIAM.* Prentice Hall, 1989.

[NN89] Janice A. Nagle and Timothy E. Nagle, editors. *Proceedings of the Fourth Annual Workshop on Conceptual Graphs*, Detroit, 1989.

[NNGE92] Tim Nagle, Jan Nagle, Laurie Gerholz, and Peter Eklund, editors. *Conceptual Structures: Current Research and Practice.* Ellis Horwood, 1992.

[Pag90] Maurice Pagnucco. Cogno - a graphical user interface to conceptual graphs. Unpublished Honours Thesis, University of Sydney, November 1990.

[Pel91] Barney Pell. A computer game-learning tournament. (In Preparation), 1991.

[Pfe92] Heather Pfeiffer, editor. *Proceedings of the Seventh Annual Workshop on Conceptual Graphs.* Springer Verlag, 1992. Las Cruces, New Mexico, July 8-10.

[PH91] Heather D. Pfeiffer and Roger T. Hartley. The conceptual programming environment, cp: Time, space and heuristic constraints. In [Way91], 1991.

[Rob73] Don D. Roberts. *The Existential Graphs of Charles S. Peirce.* Mouton, The Hague, 1973.

[Sow84] John F. Sowa. *Conceptual Structures: Information Processing in Mind and Machine.* Addison-Wesley, Reading, MA, 1984.

[Sow89] John F. Sowa. Conceptual graph notation. In [NN89], 1989.

[Sow91] John F. Sowa, editor. *Principles of Semantic Networks: Explorations in the Representation of Knowledge.* Morgan Kaufmann, San Mateo, CA, 1991.

[Str91] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley, 1991. 2/e.

[Way91] Eileen Way, editor. *Proceedings of the Sixth Annual Workshop on Conceptual Graphs*, Binghamton, 1991. SUNY Binghamton.

[Wer91] Michel Wermelinger. Get: Graph editor and tools — the incomplete reference. Technical Report RT-3/91-CIUNL, Centro de Informática da Universidade Nova de Lisboa, January 1991.

[Win83] T. Winograd. *Language as a Cognitive Process.* Vol 1. Addison-Wesley, 1983.

[WL92] Michel Wermelinger and José Gabriel Lopes. An x-windows toolkit for knowledge acquisition and representation based on conceptual structures. In [Pfe92], pages 233–242, 1992.

[YCO92] Gi-Chul Yang, Young Bae Choi, and Jonathan Oh. Cgma : A novel conceptual graph matching algorithm. In [Pfe92], pages 69–78, 1992.

[YO90] Gi-Chul Yang and Jonathan Oh. An efficient dictionary management system for machine translation. In *Seoul International Conference on Natural Language Processing*, Seoul, 1990.

[Zlo91] Fred Zlotnick. *The POSIX.1 Standard, A Programmer's Guide.* Benjamin Cummings, 1991.

13.2 For further information

Please consult the detailed conceptual graphs bibliography in [NNGE92].