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

Robert Levinson
Gerard Ellis (Eds.)

Laval University, Quebec, Canada, August 7, 1993.

# Contents

# Author Index

# 1

# STDS: PEIRCE Software Requirements and Development Standards

## Harry S. Delugach [1]

```
%=====================================================================
%            P E I R C E   W o r k b e n c h   U n i t
%
% NAME: stds2
% TITLE: STDS: PEIRCE Software Requirements and Development Standards
% TYPE: Chapter in PEIRCE proceedings
% SUBSYSTEM: Documentation
% FILENAME: stds2.tex
% PURPOSE: To argue for development of a set of explicit software requirements
%          for PEIRCE, and to promulgate software design and programming
%          standards.
% AUTHOR: Harry S. Delugach (delugach@cs.uah.edu)
% DATE: 2 Jul 1993
% VERSION TYPE: Release
% VERSION NUMBER: 1.1
%
% CHANGE LOG:
%   Date         Person       Change and Reason
%   -----------  ------------ ------------------------------------------
%   14 Jul 1992  Delugach     Initial version
%   07 Jul 1993  Delugach     Addition of software requirements sections.
%
% RELATED FILES/UNITS: stds.tex
% VARIANCE FROM STDS: n/a.
% LANGUAGE: LaTeX
% FILES NEEDED:
%     stds.bib   : contains references needed for bibtex
%     peirce.sty : adaptation of svma.sty specific to PEIRCE
% BUILD COMMANDS:
%     latex stds2
%     bibtex stds2
%     latex stds2
% INPUT:
% OUTPUT: stds2.dvi DVI file
% FUNCTIONAL PROCESSING: n/a
% ALGORITHMS USED: n/a
% TEST DATA: n/a
% REFERENCES: In stds.bib
=====================================================================
```

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

## 1.1 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.

## 1.2 Software Requirements For PEIRCE

Experienced practitioners of software development know that a project must have well-understood requirements. The need for software requirements has been extensively documented; e.g., [15, 31, 34, 36, 51, 5]. Requirements must have certain properties to be useful:

**Correct.** Each requirement must accurately convey the user's need. A fundamental limitation of requirements analysis is that the needs exist in the user's (or group's) mind. A requirements analysis process is undertaken in order to elicit the actual requirements and make them explicit. A secondary limitation is that users are notoriously inadequate in knowing what they want until a detailed analysis has been performed, thus altering the actual (in their mind) requirements as well as the explicit ones.

**Consistent.** A set of requirements must be internally consistent. This characteristic is hard to achieve with a requirements document written in English text. It is quite possible (and likely) for one requirement to conflict with another. Requirements should therefore be stated in some formal language that will allow internal consistency checking, similar in fashion to a programming language compiler. [2]

**Complete.** This characteristic means that all relevant requirements have been captured. Without a working prototype and plenty of feedback from an active user community, it will be difficult to achieve.

**Understandable.** The representation and formulation of the requirements must be clearly understood by all developers. The advantage of natural language is that all participants know one; the disadvantage is that natural language is notoriously ambiguous – readers can often misinterpret what other writers have said.

**Testable.** Each requirement must be stated in behavioral terms so that the developer can clearly show whether or not the resulting software meets the requirement.

For example, consider the requirement: *PEIRCE software should be easy-to-use.* This is a vague, ambiguous requirements. Whose use should be "easy"? What kinds of "use" are to be easy: graphic manipulation (interface) or integration of new modules? How does one define and measure "easy"? What user group forms the basis for determining "easy"? How can one verify that the resulting software is, in fact, "easy to use"?

### 1.2.1 The PEIRCE Community

The PEIRCE community has particular characteristics that govern the selection of requirements for its software. Members fall into two broad categories with respect to PEIRCE products: *users* and *researchers*. Users are those who are primarily operating the software with few changes. Researchers are those who adapt the software for studying particular modifications or adaptations of conceptual graphs. The category of users is relativly easy to satisfy, since their interaction with software products

[2] A natural and logical choice for such a language would be conceptual graphs themselves. This writer has spent some time exploring that possibility [16, 17]. At present, it is clear that many requirements are not easily expressed in first-order logic and hence are unable to be captured by conceptual graphs.

is essentially through their external interfaces. The category of researchers is much harder to satisfy, since any portion of code may need to modified in an efficient and understandable way.

Any member of the community will probably act in both roles. Someone studying new matching algorithms will likely be a mere user of a graphical interface. Likewise, someone studying interfaces may be a mere user of the parser and matcher.

Members of the community are much more interested in exploring new ideas and using the PEIRCE tools than in software development *per se*. This is to be expected in a vibrant, ongoing research community. As a result, the software development itself is being pursued in an *ad hoc* fashion, with only broad guidelines which are themselves being only partly followed.

Since the PEIRCE community is widely dispersed geographically, we must rely primarily upon electronic mail as our communication mechanism. Once a year we meet as a group where we can exchange ideas and get a sense of what the group wants. This "sense" is subjective and incomplete; it is far from being a sound basis from which to develop software. People commonly make the mistake that what they know and feel is what everyone knows and feels, too. PEIRCE needs a clearly stated set of goals and requirements for its software or else we will all happily pursue what we want, assuming (incorrectly) that everyone else wants the same thing.

One may argue that an explicit set of requirements is not a good thing for PEIRCE at this time, that they would unnecessarily hamper and constrain developers from implementing their ideas. It is true that the wrong set of requirements will certainly harm the project; this is true of any other project as well. We therefore consider requirements to be vital to the success of PEIRCE. Explicit requirements should lead to focused, explicit discussion about the qualities of the products we want. This chapter introduces some requirements for discussion and elaboration.

### 1.2.2 Requirements For PEIRCE

The following are a preliminary set of requirements. The software within PEIRCE will:

1. Support graphic editing of conceptual graphs (create, move, connect, print)
2. Enable conversions between display form and linear form.
3. Support the ability to organize graphs into sets based on their origination; e.g., canonical set, database set, specific domains, etc.
    (a) Display them.
    (b) Store them as a group.
    (c) Process graphs based on their inclusion in one or more groups.
    (d) Include user-defined graphs in any given set.
4. Perform automatic consistency checking among a set of conceptual graphs.
    (a) Using constraints as given in Sowa84.
    (b) Using additional user-defined constraints.
    (c) Displaying results according to user-defined forms.
5. Support automatic graph matching, including, but not limited to:
    (a) Successive application of CG primitive operations to obtain a match,
    (b) A trace of what specific primitives were applied.
    (c) Assumptions (if any) made by the matcher along the way.
6. Contain query graph capabilities: i.e., creating a query graph and seeing:
    (a) Whether it can be satisfied
    (b) What resulting graph(s) satisfy the query.

   (c) The inference path(s) from which the result graph is obtained.

7. Support the ability to convert to/from CGs and natural language, including:
   (a) Ability to center upon particular domains of discourse.
   (b) Ability to substitute other natural languages.
   (c) Ability to tailor conversion according to user preferences.

8. Support the ability to extend/augment the editing/conversion/matching/checking/query capabilities in order to try out some new or modified feature of CGs.

9. Possess sufficient documentation and explanation so that
   (a) Users will be able to achieve at least some required behaviors within one hour of installing the software, without consulting the software's authors.
   (b) Users will be able to make changes for their own purposes, achieving some useful result while consulting the software's authors no more than once.

These requirements are a minimum set. They are open to debate, discussion, modification, argument, and/or acceptance.

## 1.3 Purpose and Scope of Standards

Experience tells us that software development standards are essential to a successful collaborative effort. This chapter recommends some standards for the PEIRCE project, and their rationale. These standards are an expanded version of those given in [18].

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 many developers (but probably 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.

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 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.

## 1.4 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:<br>a design module specification,<br>an implementation module,<br>a report,<br>a manual,<br>a test data set,<br>a bug report,<br>a change request.

## 1.5 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.

### 1.5.1 Manuals and Technical Reports

The standard text format for manuals and technical reports shall be LaTeX, as described in [40]. This is a subset of TeX, as described in [39].

LaTeX 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<br>Thomson Hall, Mail Stop DR-10<br>University of Washington<br>Seattle, WA 98195 (USA)<br>(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, LaTeX seems 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 in LaTeX form via text 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.
- LaTeX handles bibliographies and citations with ease.
- It is freely available.
- LaTeX source 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 LaTeX documents and other formats, and also utilities for previewing LaTeX documents under X11R5.

### 1.5.2 Unit Header Identification Standard

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 PEIRCE unit header. 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 in some cases, only information the unit header will be distributed, so be sure to include all important information in the header.

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 Figure 1.1 and may be found in the file `stds-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

```
======================================================================
             P E I R C E   W o r k b e n c h   U n i t

NAME:
TITLE:
TYPE:
SUBSYSTEM:
FILENAME:
PURPOSE:
AUTHOR:
DATE:
VERSION TYPE:
VERSION NUMBER:
CHANGE LOG:
  Date         Person       Change and Reason
  -----------  ------------ -------------------------------------------
RELATED FILES/UNITS:
VARIANCE FROM STDS: n/a.
LANGUAGE:
FILES NEEDED:
BUILD COMMANDS:
INPUT:
OUTPUT:
FUNCTIONAL PROCESSING:
ALGORITHMS USED:
TEST DATA:
REFERENCES:
======================================================================
```

FIGURE 1.1. PEIRCE Unit Identification Header Template

  - *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, style 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.

```
======================================================================
         P E I R C E   W o r k b e n c h   B U G   R E P O R T

BUG ID: (TBA)
UNIT NAME:
UNIT TITLE:
UNIT TYPE:
SUBSYSTEM:
UNIT FILENAME:
BUG REPORT FILENAME:
UNIT DATE:
VERSION TYPE:
VERSION NUMBER:
PERSON(S) REPORTING:
DATE OF REPORT:
RELATED FILES/UNITS:
BRIEF DESCRIPTION OF BUG:
DETAILED DESCRIPTION OF BUG:
INPUT:
EXPECTED OUTPUT/BEHAVIOR:
ACTUAL OUTPUT/BEHAVIOR:
CAUSE OF BUG (if known):
SUGGESTED REPAIR STRATEGY:
WORKAROUND:
REFERENCES:
DISPOSITION: (TBA)
======================================================================
```

FIGURE 1.2. PEIRCE Bug Report Identification Header Template

### 1.5.3 Bug Report Header Information Standard

Users of the PEIRCE software are implicitly acting as testers for the community. When bugs or suspected anomalies are discovered, users are encouraged to submit a report of the phenomenon. This section defines what belongs in a PEIRCE bug report. Submitters are urged to include every item in the header, and to keep them in the standard order given. A template is given in Figure 1.2 and may be found in the file `stds-bug-header.template`.

- **Bug ID.** This will be filled in with a standard identification code by the standards/integration coordinator. Users should leave it blank. The coordinator will forward the bug to the appropriate person(s).
- **Unit Name.** *Required.* The unit's name in which the bug was found. Should correspond exactly to the **Name** item in the unit's standard header.
- **Unit Title.** *Required.* The title of the unit in which the bug was found.
- **Subsystem.** *Required.* The subsystem name, as agreed upon by the development community, to which the "offending" unit belongs. Should correspond exactly to the **Subsystem** item in the unit's standard header.
- **Type.** *Required.* Should correspond exactly to the **Type** item in the unit's standard header.
- **Unit Filename.** Should correspond exactly to the **Filename** item in the unit's standard header.
- **Bug Report Filename.** A suggested name for the filename in which the bug report appears.

- **Person(s) Reporting.** *Required.* Includes information about the person submitting the report, including the name(s), postal address, phone number and electronic mail address.
- **Date of Report.** *Required.* The date on which the bug report was submitted. Date should be expressed in day-month-year form, preferably as day-month-year; e.g.; 02 Jul 1993.
- **Version Type.** Should correspond exactly to the **Version Type** item in the unit's standard header.
- **Version Number.** Should correspond exactly to the **Version Number** item in the unit's standard header. If there is no version number in the unit, then refer to the version by the date on which it was submitted.
- **Related Files/Units.** Any other files or units that are affected by the bug/anomaly.
- **Brief Description.** A one- or two-line description concisely stating the features of the bug.
- **Detailed Description.** A thorough description of the bug, including the input and output information below.
- **Input.** A specification of the input that produced the actual behavior described. The specification should be complete enough that the bug can be reproduced by someone else using the unit.
- **Expected Output/Behavior.** A description of the output expected from the unit's operation, given the specified input.
- **Actual Output/Behavior.** A description of the actual output observed from the unit's operation.
- **Cause of Bug.** Describe the suspected source of the bug, or more than one suspected source if the reporter can isolate this information.
- **Suggested Repair Strategy.** If the reporter has suggestions on how to fix the bug, include them here.
- **Workaround.** Until the bug is repaired, what can users do to continue operating the software without incurring the bug's effects.
- **References.** Any publications, etc. relevant to the bug report. The "offending" unit and its references need not be repeated.
- **Disposition.** Leave blank. To be filled in by the coordinator.

## 1.6 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.

### 1.6.1 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.

### 1.6.2 Programming Language

The programming language for all implemented modules will be C++, version 2.1. A description of the language can be found in [56]. 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.

PEIRCE participants should note: C++ is not simply an enhanced version of C. It is an adaptation of C geared toward object-oriented programming techniques. Programmers are urged to consult a text on object-oriented programming and development (e.g., [8, 9, 45, 52]) in order to make full use of encapsulation and modularity. Mere use of C++ by itself in no way guarantees that program units will be encapsulated or modular! Certain coding standards are recommended here; others may be found in the texts mentioned.

### 1.6.3 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.

### 1.6.4 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 [62]. The standard itself [38] is available from the publisher.

### 1.6.5 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.

## 1.7 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.

### 1.7.1 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.

### 1.7.2 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.

### 1.7.3 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. Four repository sites are currently set up:

- ftp.cs.su.oz.au (129.78.8.208). /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)
- crl.nmsu.edu (128.123.1.18). /pub/cg
  Caretaker: Heather Pfeiffer (hdp@nmsu.edu)
- lut.ac.uk (131.231.16.16). (in preparation)
  Caretaker: Rob Kirkwood (R.S.M.Kirkwood@lut.ac.uk)

*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.

## 1.8 Conclusion

The PEIRCE Workbench software must exhibit certain baseline functions and behavior which are yet to be fully determined. A detailed set of requirements has not been agreed upon. More importantly, the Workbench effort requires adherence to a set of standards to accomplish its primary goal: a common environment with which to study conceptual graphs. While everyone seems to agree on the need for standards, so far no one has submitted a single module that adheres that adheres to the standards.

## 1.9 Acknowledgement

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

# 2

# LINEAR — Linear Notation Interface

**John Esch**[1]
**Maurice Pagnucco**[2]
**Michel Wermelinger**[3]

## 2.1 Introduction

The purpose of the LINEAR subgroup is to coordinate the development of a standard linear form grammar for Conceptual Graphs and to develop related tools. The main aim of this paper is to present a linear form grammar for conceptual graphs which we hope will form the basis for future standardization efforts. We also suggest some possible extensions to this grammar and contrast it with the grammar used by PEIRCE.

In the following section we present a linear form grammar based on the work of Esch [27] while section 2.3 discusses some suggested extensions to the linear form grammar. Section 2.4 presents the linear form grammar adopted by the current version of PEIRCE (version 0.1) and discusses some of the more significant differences between this grammar and that of section 2.2. In appendix 2.7 we include Sowa's comments on possible future directions with a discussion on integrating the CG linear form grammar with that of KIF[29], the Knowledge Interchange Format developed by the Interlingua Committee of the DARPA-sponsored Knowledge Sharing Effort.

## 2.2 Linear Form Grammar — Current Status

This section contains extended BNF productions defining a grammar for Conceptual Graph Linear Forms and some Extended Linear Forms. It is based on [27]. The underlying character set is ASCII[4] and the case of letters is only significant within strings (production 260). However, actual implementations may provide for user-definable options, like typelabels in uppercase and variables in lowercase. The user should also be able to choose whether the Extended or the Contracted Form (see section 2.3) is to be used on output. Furthermore, in production 356 parsers may let the arc before an *r-node* be optional, as well as the last comma before the period terminating the graph.

---

[1]Paramax Systems Inc. (A Unisys Company), P.O. Box 64525, MS U1N28, St. Paul, MN 55164, USA. email: `esch@email.sp.paramax.com`

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

[3]Departamento de Informática, Universidade Nova de Lisboa, 2825 Monte da Caparica, PORTUGAL. email: `mw@fct.unl.pt`

[4]However, for a true mapping between Natural Language and CGs the international ISO character set should be used.

### 2.2.1 Meta-Language productions

<table>
<tr><th>Number</th><th>Name</th><th>Production</th></tr>
<tr><td>100</td><td>Terminal</td><td>Terminal symbols are written between quotes, e.g. “0”.</td></tr>
<tr><td>110</td><td>Non-Terminal</td><td>Non-Terminal symbols are denoted by italics, e.g. *concept*.</td></tr>
<tr><td>120</td><td>Alternatives</td><td>Alternatives are separated by a vertical bar |, e.g.<br>“0” | “1” means a 0 or a 1.</td></tr>
<tr><td>130</td><td>Groupings</td><td>Groupings are contained in curly braces { and }, e.g.<br>$\{A\ B\ C\}$ means $A$ followed by a $B$ followed by a $C$, and<br>$\{A \mid B \mid C\}$ means either an $A$, or a $B$ or a $C$.</td></tr>
<tr><td>150</td><td>Options</td><td>Options are contained in square brackets [ and ], e.g.<br>[*rlink*] means an optional *rlink* and<br>[“–>” | “<–”] means –> or <– or neither.</td></tr>
<tr><td>160</td><td>Repetition</td><td>Repetition is denoted by ... following a symbol, group<br>or option, e.g. *cgraph*... means 1 or more<br>*cgraph*s, { }... means 1 or more of the group or<br>alternatives within a group, and [ ]... means 0 or more of<br>the option or alternatives within the option.</td></tr>
<tr><td>170</td><td>Spaces</td><td>Spaces, returns, linefeeds, tabs, etc. within a production<br>are treated as a single space except within terminals or as<br>otherwise explicitly noted by, e.g. a *newline* non-terminal.<br>Where such a space occurs, it implies zero or more spaces,<br>returns, linefeeds, tabs, etc. in the language being defined<br>which are also interpreted as a single space for parsing;<br>no space between successive terminals or non-terminals is<br>interpreted as no space in the target language.</td></tr>
<tr><td>180</td><td>Comments</td><td>Comments in the meta and target languages are denoted by a<br>double semicolon where it and any other text remaining on<br>the line are interpreted as an end of line.</td></tr>
</table>

### 2.2.2 Low Level Productions

<table>
<tr><th>Number</th><th>Non-Terminal</th><th>Production</th></tr>
<tr><td>200</td><td>*digit*</td><td>“0” | “1” | “2” | “3” | “4” | “5” | “6” | “7” | “8” | “9”</td></tr>
<tr><td>202</td><td>*number*</td><td>*digit*...</td></tr>
<tr><td>204</td><td>*letter*</td><td>“A” | “B” | “C” | “D” | “E” | “F” | “G” | “H” | “I” |<br>“J” | “K” | “L” | “M” | “N” | “O” | “P” | “Q” | “R” |<br>“S” | “T” | “U” | “V” | “W” | “X” | “Y” | “Z” |<br>“a” | “b” | “c” | “d” | “e” | “f” | “g” | “h” | “i” |<br>“j” | “k” | “l” | “m” | “n” | “o” | “p” | “q” | “r” |<br>“s” | “t” | “u” | “v” | “w” | “x” | “y” | “z”</td></tr>
<tr><td>206</td><td>*punctuation*</td><td>“[” | “]” | “(” | “)” | “{” | “}” | “-” | “<” |<br>“>” | “@” | “#” | “*” | “,” | “.” | “:” |<br>“ ” | “~” | *tab* | *lambda* | *forall*</td></tr>
<tr><td>208</td><td>*eol*</td><td>[*comment*] {*eof* | *return* | *newline*}</td></tr>
<tr><td>210</td><td>*other-char*</td><td>character-other-than-digit-letter-punctuation-or-eol</td></tr>
<tr><td>220</td><td>*name-char*</td><td>*digit* | *letter* | *other-char*</td></tr>
<tr><td>230</td><td>*text-char*</td><td>*name-char* | *punctuation*</td></tr>
<tr><td>240</td><td>*text*</td><td>*text-char*...</td></tr>
<tr><td>250</td><td>*comment*</td><td>“;;” *text*</td></tr>
<tr><td>260</td><td>*string*</td><td>“"” *text* “"”</td></tr>
<tr><td>270</td><td>*identifier*</td><td>*letter*[*name-char*]...</td></tr>
<tr><td>280</td><td>*name*</td><td>{*letter* | *digit*}[*name-char*]...</td></tr>
<tr><td>290</td><td>*long-name*</td><td>*name*{“ ”...*name*}... ;;a sequence of names</td></tr>
</table>

### 2.2.3 Conceptual Graph Productions

<table>
<tr><th>Number</th><th>Non-Terminal</th><th>Production</th></tr>
<tr><td>300</td><td>cgraph</td><td>c-node | relation conlink | actor conlink</td></tr>
<tr><td>310</td><td>c-node</td><td>concept [rlink]</td></tr>
<tr><td>312</td><td>r-node</td><td>relation [conlink]</td></tr>
<tr><td>314</td><td>a-node</td><td>actor [conlink]</td></tr>
<tr><td>320</td><td>concept</td><td>ind-concept | context | str-concept</td></tr>
<tr><td>322</td><td>ind-concept</td><td>“[” [typefield] [reffield] [annotation] “]”</td></tr>
<tr><td>324</td><td>typefield</td><td>[not]typelabel | lambda-exp</td></tr>
<tr><td>326</td><td>exist-ref</td><td>“[” [[typelabel] “:”] “*” generic-ref “]”</td></tr>
<tr><td>328</td><td>str-concept</td><td>“[” string “]” ;;short for [WORD: string]</td></tr>
<tr><td>330</td><td>relation</td><td>“(” [not]typelabel [annotation] “)”</td></tr>
<tr><td>340</td><td>actor</td><td>“<” typelabel [annotation] “>”</td></tr>
<tr><td>350</td><td>conlink</td><td>arc c-node | “-” conlist</td></tr>
<tr><td>352</td><td>rlink</td><td>arc r-node | arc a-node | “-” rlist</td></tr>
<tr><td>354</td><td>conlist</td><td>{newline arc c-node }... “,”</td></tr>
<tr><td>356</td><td>rlist</td><td>{newline {arc r-node | arc a-node} }... “,”</td></tr>
<tr><td>360</td><td>typelabel</td><td>long-name [“@” canon-name]</td></tr>
<tr><td>362</td><td>canon-name</td><td>name</td></tr>
<tr><td>364</td><td>annotation</td><td>“;” long-name</td></tr>
<tr><td>370</td><td>context</td><td>[not] “[” cgraph... “]”</td></tr>
<tr><td>374</td><td>not</td><td>“~”</td></tr>
<tr><td>376</td><td>lambda-exp</td><td>lambda “(” generic-ref “)” cgraph</td></tr>
<tr><td>380</td><td>arc</td><td>[number] { “<−” | “−>” }</td></tr>
<tr><td>390</td><td>generic-ref</td><td>identifier</td></tr>
</table>

### 2.2.4 Referent Field

<table>
<tr><th>Number</th><th>Non-Terminal</th><th>Production</th></tr>
<tr><td>400</td><td>reffield</td><td>“:” [names] [vars] [mark] [not] [referent]</td></tr>
<tr><td>402</td><td>names</td><td>name[“=” name]...</td></tr>
<tr><td>404</td><td>vars</td><td>variable[“=” variable]...</td></tr>
<tr><td>406</td><td>variable</td><td>“*”[identifier]</td></tr>
<tr><td>406</td><td>mark</td><td>“!” | “?”</td></tr>
<tr><td>410</td><td>referent</td><td>individual [measure] | generic-referent</td></tr>
<tr><td>412</td><td>individual</td><td>“#”[name]</td></tr>
<tr><td>414</td><td>measure</td><td>“@” long-name</td></tr>
<tr><td>420</td><td>generic-referent</td><td>set-ref [quantity] | cgraph |<br>literal | string | measure</td></tr>
<tr><td>430</td><td>set-ref</td><td>[“COLL”|“DISJ”|“DIST”|“RESP”]<br>“{” set-ref-list “}”</td></tr>
<tr><td>432</td><td>set-ref-list</td><td>“*” | set-ref[“,” set-ref]...</td></tr>
<tr><td>434</td><td>set-ref</td><td>name | variable | individual</td></tr>
<tr><td>440</td><td>quantity</td><td>“#” | forall | “@” long-name</td></tr>
<tr><td>450</td><td>literal</td><td>“’” long-name | “’” list</td></tr>
<tr><td>452</td><td>list</td><td>“(” [long-name | list]... “)”</td></tr>
</table>

### 2.2.5 Extended Linear Forms

| Number | Non-Terminal | Production |
|---|---|---|
| 500 | *lf-file* | {*statement* \| *comment*}... |
| 502 | *statement* | {*cgraph* \| *other-stm*} "." |
| 504 | *other-stm* | *canon-stm* \| *type-stm* \| *relation-stm* \| *actor-stm* \| *schema-stm* \| *proto-stm* \| *selecting-stm* \| *individual-stm* \| *canonical-stm* |
| 510 | *canon-stm* | "CANON" *canon-name* [ "IS" *cgraph* ] |
| 512 | *type-stm* | "TYPE" [*not*]*typelabel arg* "IS" [*exist-ref*] *cgraph* |
| 514 | *relation-stm* | "RELATION" [*not*]*typelabel args* "IS" [*exist-refs*] *cgraph* |
| 516 | *actor-stm* | "ACTOR" *typelabel args* "IS" *cgraph* |
| 520 | *schema-stm* | "SCHEMA" ["FOR"] *typelabel args* "IS" *cgraph* |
| 522 | *proto-stm* | "PROTOTYPE" *typelabel args* "IS" *cgraph* |
| 524 | *selecting-stm* | "SELECTING" "GRAPH" ["FOR"] *typelabel* "IS" *cgraph* |
| 526 | *individual-stm* | "INDIVIDUAL" *typelabel ind* "IS" *cgraph* |
| 528 | *canonical-stm* | "CANONICAL" "GRAPH" "IS" *cgraph* |
| 540 | *arg* | "(" *identifier* ")" |
| 550 | *args* | "(" *identifier* {","*identifier*}... ")" |
| 560 | *exist-refs* | *exist-ref*... |
| 570 | *ind* | "(" ["#"]*identifier* ")" |

## 2.3 Extensions to Linear Form

The grammar presented in the previous section already contains some extensions to a minimal linear form grammar. In this section we suggest some further possible extensions to this grammar.

Firstly, we consider some extensions mentioned by Sowa [54]. These extended linear forms will in fact be parsed by the grammar in section 2.2 — this table presents their equivalent linear forms.

| Name | Contracted Form | Extended Form |
|---|---|---|
| name contraction | `[C:Name]`<br>`[C:Name#1234]` | `[C]->(NAME)->[WORD:"Name"]`<br>`[C:#1234]->(NAME)->[WORD:"Name"]` |
| measure contraction | `[C:@10m]`<br>`[C:#1234@10m]` | `[C]->(MEAS)->[MEASURE:10m]`<br>`[C:#1234]->(MEAS)->[MEASURE:10m]` |
| quantity contraction | `[C:{*}@2]` | `[C:{*}]->(QTY)->[NUMBER:2]` |
| negation | `(R)->~[C]` | `(R)->[C]<-(NEG)` |
| proposition contraction | `[PROPOSITION:G]` | `[PROPOSITION]->(STMT)->[GRAPH:G]` |
| situation contraction | `[SITUATION:G]` | `[SITUATION]->(DSCR)->[PROPOSITION]->(STMT)->[GRAPH:G]` |

The next table presents further extensions, which for the time being aren't processed by the previous grammar.

| Name | Contracted Form | Extended Form |
|---|---|---|
| empty set | `[C:{}]` | `[C:{*}]->(QTY)->[NUMBER:0]` |
| cardinality restrictions | `[C:{*}@>2]`<br>`[C:{*}@2-4]` | `[C:{*}]->(QTY)->[NUMBER]->(GT)->[NUMBER:2]`<br>`[C:{*}]->(QTY)->[NUMBER:*x]-`<br>`->(GE)->[NUMBER:2]`<br>`->(LE)->[NUMBER:4]` |
| subrange types | `[NUMBER>0:1]`<br>`[SPEED<=55mph]` | `[@lambda(x) [NUMBER:*x]->(>)->[NUMBER:0] : 1]`<br>`[@lambda(x) [SPEED:*x]->(MEAS)->[MEASURE]-`<br>`->(<=)->[MEASURE:55mph]]` |

Further extensions and changes to the grammar may be considered. Most of them need further discussion among the CG community. We therefore list several topics that we would like to see discussed at the workshop.

**Special Symbols** The current grammar doesn't define the *lambda* and *forall* non-terminals. The symbols "@lambda" and "@every" have been suggested by Sowa. However, ambiguity may arise in productions 360 and 440. Another suggestion might be to use the quote character, as for lists. Or one can simply introduce the notion of reserved identifier (as in Pascal) which can't be used otherwise. In this case, no special symbol like @ would be necessary.

**Names and Identifiers** If typelabels like `PHYS-OBJ` and `>` (for the greater-than relation) and variables like "*123" and "*x_1" are to be possible[5], the definitions of *name* and *identifier* (productions 270 and 280) must be changed or even merged together. Moreover, it would be nice to allow accents for an easier mapping between Natural Language and CGs. For example, the Portuguese word *acção* (action) could be mapped into $\boxed{\text{A,CC\textasciitilde AO}}$. Notice that this isn't possible for the moment because both "," and "~" are part of *punctuation* and as such they can't be considered as *other-char*. Also, it isn't possible to define the non-member function as `ACTOR ~member(element, list) IS`... and `SCHEMA FOR ~boat(x) IS`... isn't possible either because *typelabel* (production 360) doesn't include *not* (it is defined case by case in the other productions where *typelabel* occurs).

**Other** A graph like $\neg\boxed{\text{TAUTOLOGY: }\boxed{*\text{p}}\ \neg\boxed{*\text{p}}}$ is not possible because production 370 doesn't allow typelabels and coreference markers for *negated* contexts. It might be handy to allow (nested) multiple-line comments, especially for batch-oriented CG "compilers". Furthermore, the terminal "**" has been proposed for sets, instead of simply "*".

## 2.4 Comparison with PEIRCE Linear Form

The grammar in section 2.2 represents many years of work and most of the mainstream ideas on conceptual graphs. PEIRCE on the other hand, being only in its first release, uses a much smaller grammar.

In this section we first present the linear form productions of the grammar parsed by the current PEIRCE release and then contrast it with the grammar of section 2.2 pointing out some of the more significant differences between the two.

[5] They aren't for the moment because both "-" and ">" aren't part of *other-char* and as such can't appear in names, while identifiers can't begin with digits and can't include *other-char*s.

### 2.4.1 Grammar for PEIRCE 0.1

The following table presents the linear form grammar accepted by the current version of PEIRCE [6] [24] (see pp. 11-13).

<table>
<tr><th>Non-Terminal</th><th>Production</th></tr>
<tr><td>sentence</td><td>term “.” | term “?”</td></tr>
<tr><td>term</td><td>unconnected_graph | declaration</td></tr>
<tr><td>unconnected_graph</td><td>connected_graph | connected_graph unconnected_graph</td></tr>
<tr><td>connected_graph</td><td>node link</td></tr>
<tr><td>graph</td><td>node link</td></tr>
<tr><td>node(Type)</td><td>left_delimiter(Type) node_label(Type) right_delimiter(Type)</td></tr>
<tr><td>node_label(Type)</td><td>type | type “:” referent | variable | graphs</td></tr>
<tr><td>referent</td><td>variable | individual_marker | graphs</td></tr>
<tr><td>variable</td><td>“*” variable_name | variable_name</td></tr>
<tr><td>individual_marker</td><td>string | individual | ’#’ individual</td></tr>
<tr><td>individual</td><td>individual_first_char individual_suffix</td></tr>
<tr><td>individual_first_char</td><td>upper_case | digit</td></tr>
<tr><td>individual_suffix</td><td>empty | individual_suffix_char individual_suffix</td></tr>
<tr><td>individual_suffix_char</td><td>alphanumeric | “_”</td></tr>
<tr><td>graphs</td><td>graph | graph graphs</td></tr>
<tr><td>left_delimiter(CONCEPT)</td><td>“[”</td></tr>
<tr><td>left_delimiter(RELATION)</td><td>“(”</td></tr>
<tr><td>right_delimiter(CONCEPT)</td><td>“]”</td></tr>
<tr><td>right_delimiter(RELATION)</td><td>“)”</td></tr>
<tr><td>link</td><td>empty | arc node link | “-” links “,”</td></tr>
<tr><td>links</td><td>link_one_of_many | link_one_of_many links</td></tr>
<tr><td>link_one_of_many</td><td>“\n” next_link link</td></tr>
<tr><td>next_link</td><td>arc node | node arc node</td></tr>
<tr><td>arc</td><td>“−” | “−” type_label “−” | “−>” | “−” type_label “−>” |<br>“<−” | “<−” type_label “−”</td></tr>
<tr><td>variable_name</td><td>lower_case variable_name_suffix</td></tr>
<tr><td>variable_name_suffix</td><td>variable_name_suffix_char variable_name_suffix</td></tr>
<tr><td>variable_name_suffix_char</td><td>alphanumeric | “_”</td></tr>
<tr><td>type</td><td>type_label | type_label “(” feature_list “)”</td></tr>
<tr><td>type_label</td><td>first_type_label_char type_label_suffix. | string</td></tr>
<tr><td>first_type_label_char</td><td>upper_case_char | digit | “−”</td></tr>
<tr><td>type_label_suffix</td><td>empty | type_label_suffix_char type_label_suffix</td></tr>
<tr><td>type_label_suffix_char</td><td>alphanumeric | “_”</td></tr>
<tr><td>feature_list</td><td>feature_pair | feature_pair “,” feature_list</td></tr>
<tr><td>feature_pair</td><td>type_label “=>” feature_term |<br>type_label “=>” feature_term “:” variable |<br>feature_term | feature_term “:” variable</td></tr>
<tr><td>string</td><td>““” non_double_quoted_chars “””</td></tr>
<tr><td>declaration</td><td>type_label “<” type_labels | type_label “::” individual</td></tr>
<tr><td>typelabels</td><td>type_label | type_label “,” type_labels</td></tr>
</table>

### 2.4.2 Comparison with Standard Linear Form Grammar

Firstly, there are a number of things considered in the PEIRCE grammar that are not addressed by the grammar presented in section 2.2. These include:

- Declaration of subtype relationships[7] (e.g., $\texttt{Type} < \texttt{Type}_1, \ldots, \texttt{Type}_n$)

[6] PEIRCE can in fact parse more than just conceptual graphs in linear form (see [24]) but we shall restrict ourselves to discussing the linear form portion of its grammar.

[7] Production 512 handles this partially.

- Declaration of conformity relationships[8] (e.g., `Type :: #1234`)
- Type labels on arcs denoting relations `[C1]−R−>[C2]` (e.g., see [24] p.28)
- Feature terms (e.g., [24] p.28 `[WHITE KING(RANK=>0): *x2]`)

Among the things covered by our grammar but not as yet by PEIRCE are:

- Negated contexts
- Actors
- Reasonably complex referents — PEIRCE currently allows only generic and individual referents

There are also some other differences between the two grammars. In particular:

| Feature | Grammar in Section 2.2 | PEIRCE v0.1 |
|---|---|---|
| Relation/Concept | Differentiated in grammar | Not differentiated in grammar but code checks that relations are connected to concepts and vice versa |
| Variables | Must begin with "*" followed by an identifier beginning with a letter | Must begin with lowercase letter may be preceded by "*" |
| Individuals | Must begin with "#" followed by an name beginning with a letter or digit | Must begin with uppercase letter or digit may be preceded by "#" |
| Type Label | Series of names beginning with a letter or digit. May also appear negated | Must begin with uppercase letter or be a string |

## 2.5 Conclusions

In this report we have investigated a number of issues concerning the linear form of a conceptual graph. We have presented a fairly comprehensive grammar that represents many of the mainstream developments in conceptual graphs over the years. This grammar was compared with that of PEIRCE 0.1 to give an idea of the fundamental differences between the two. Regarding the integration with KIF, the minimal change that should be incorporated in CGs is the addition of "?"*identifier* as a form of variable as well as "*"*identifier* as now allowed. The semantics would be that "?"*identifier* refers to a previously defined variable, if any, while "*"*identifier* defines a new scope for that variable.

Our agenda for future work includes:

- Clean up the grammar making a distinction between symbols and separators.
- Maybe add a general purpose macro mechanism for contracted and expanded forms.
- Augment and adapt the descriptions and the examples contained in [27] to the new grammar.
- Write some graph examples both in standard and PEIRCE notation in order to show the differences between both grammars more clearly.

[8] Production 526 and the occurrence of individuals in concepts handle this.

## 2.6 Acknowledgements

The authors would like to thank Gerard Ellis for providing a copy of the PEIRCE linear form grammar included in section 2.4 and John Sowa for letting us use his notes on interfacing CGs with KIF.

## 2.7 Future Directions — Interfacing with KIF

Last week[9], there was a meeting with Mike Genesereth, Richard Fikes, Jim Fulton, Bob Neches, and me about the relationship between KIF, CGs, and SUMM (Fulton's Semantic Unification Meta-Model). Our conclusion was that we would propose both KIF and CGs as "normative languages" for the IRDS Conceptual Schema and we would adopt the SUMM ontology for metalanguage definitions.

That means that one critical issue is to get KIF and CG syntax lined up in a one-to-one correspondence wherever possible. There will be many cases where we have to map one feature in one of the languages into an expression in the other, but we would like to minimize those occurrences. I'm all in favor of adopting ideas from LIFE and other sources, but any syntactic constructions that might create additional complexity in the KIF $\leftrightarrow$ CG mapping should be avoided. I think that we should encourage the LIFErs to map their syntax to the Peirce software, but we should avoid cluttering up the CG syntax with too many additional conventions.

The basic syntactic unit in KIF is the *word* as defined by the Common Lisp reader (i.e. the blank and the parentheses are the main delimiters). Words are classified in the following categories:

**Variables** Words whose first character is "?" or "@". There are two kinds:

1. Individual variables begin with "?": `?x ?y ?abc_123`
2. Sequence variables begin with "@": `@x @y @abc_123`

**Operators** Words that are defined as operators, but not distinguished by any special character. There are 4 kinds:

1. Term operators: `listof setof quote if cond the setofall kappa lambda`
2. Sentence operators: `= /= not and or => <= <=> forall exists`
3. Rule operators: `=>> <<= consis`
4. Definition operators: `defobject defunction defrelation := => :&`

**Constants** Any word that is neither a variable nor an operator. There are four kinds, but they are not distinguished by any special syntax: object constants, function constants, relation constants, and logical constants.

Following is some discussion about the possible ways of representing these KIF constructs in CGs:

**Variables** Since both KIF and CGs use a prefix symbol for marking variables, I would be reluctant to do anything that would make CGs diverge farther from KIF. Prolog and LIFE are much less important for the standards issues than KIF. Following are some possibilities for extensions to CG notation:

1. Keep "*" as the marker for CG variables and map it to "?" for KIF individual variables. The sequence variables in KIF occur only at the end of a term, and I think that we can represent them with a slight extension to the current CG notation for sequences: `(+ ?a ?b ?c)` becomes `[NUMBER: <*a, *b, *c>]->(+)->[NUMBER]` and `(+ ?a ?b @c)` becomes `[NUMBER: <*a, *b : *c>]->(+)->[NUMBER]`. The vertical bar could be used, as in Prolog, to show that the last variable unifies with the tail of a list. That would eliminate the need for a special marker to distinguish individual variables from sequence variables.

[9]This appendix is the (slightly edited) message sent by John Sowa to the CG mailing list on February 10, 1993.

2. Adopt the KIF "?" for an individual variable and "@" for sequence variables. The "*" could be used as the prefix for quantifiers; e.g. `*` by itself would be "exists", `*every` would be equivalent to the KIF `forall`, `*1` would mean exactly one, etc.
3. Adopt the KIF "?" for variables, the notation <?a,?b:?c> for sequences, use "*" as the quantifier prefix, and free up "@" for some other purpose.

One point that came up last week was the problem of *capturing* a CG variable when a graph that happened to contain the same variable name was copied into a context. In predicate calculus or KIF, a new occurrence of an existential quantifier introduces a new variable, even if the name happens to be the same; e.g. $(\exists x)(cat(x) \wedge (\exists x)mat(x))$. The two $x$'s in that formula would be distinct. This is bad coding practice, but it might arise in computer-generated expressions. If we had separate symbols for the quantifier and the variable, we could write `*?x` in the context where the variable was defined, and just `?x` at subsequent references. This convention would simplify the mapping from KIF to CGs, and it might be useful in CGs by themselves.

**Operators** 1. Term operators: Some of these would map to CG expressions in the referent field; `listof` becomes `<,,,>`; `setof` becomes $\{,,,\}$; `setofall` becomes `*@all`; `lambda` becomes `@lambda` or `*lambda`; `the` becomes `#`; etc. Others like `if` and `cond` would become relations.

2. Sentence operators: The quantifiers would become `*every` and `*`; `= /= or <=>` would become dyadic relations; `not` would become `NEG` or some single character to be determined; `=>` would be the current "IF-THEN" construction; and we could add a backwards "IF" for `<=`.

   Note: When an upside-down "A" is available in the character set, it may be used for the universal quantifier. If it is not available, I have been using `@every` instead of `@forall` because the usual occurrences of `@every` map more directly to the English word "every". I would prefer to use `@all` in combination with $\{*\}$ to represent the KIF `setofall`. Whether we use "@" or "*" for the quantifier prefix depends on the escape character for variables.
3. Rule operators: These would become relations that apply to contexts.
4. Definition operators: These would also become relations that link contexts, but most of them could be defined in terms of lambda.

**Constants** Since the CG referent field already contains some special symbols, I think that we would have to be more restrictive. I would suggest the following categories:

1. Simple names: Any string of alphanumeric characters plus the underscore that starts with a letter or "_": `John elephant abc_12 _123`
2. Complex names: Any string of characters delimited with single quotes: `'John' 'elephant' 'Grand Central Terminal' '+' '**'`

   We could treat `'John'` and `John` as equivalent. There is a question about whether the upper/lower case distinction should be significant in names and variables. I think that KIF and CGs should adopt the same conventions on this point.
3. Numbers: Whatever the ANSI standards people want to define as a recognizable number for CG and KIF: `23.4 -1.06e+2 0x90AB`
4. Strings: Any string of characters delimited with double quotes: `"John" """" "23.4" "This is a sentence."`

   We should consider what to use as an escape character; i.e. something like `\n` for newline, `\t` for tab, etc. Again, this should be the same for KIF and CGs.

# 3

# AHP: Advanced Hardware for PEIRCE

**James D. Roberts**[1]
**Ron Demara**[2]
**Gerard Ellis**[3]
**Richard Hughey**[4]
**Robert Levinson**[5]
**Claude Noshpitz**[6]

## 3.1 Introduction

PEIRCE is already being ported to a variety of platforms, both workstations and personal computers, by other researchers. The AHP group is exploring *advanced*, high performance hardware platforms. Our work focuses almost exclusively on parallel processing, including a system under development tailored for CG processing. The goal is to support PEIRCE with hardware platforms that can dramatically improve performance and in turn increasethe size of CG research domains and applications.

To date, we have been unsuccessful in obtaining information from applications that is necessary for evaluating processing needs and characteristics. We urge those who have gathered statistics (averages or distributions on number of concept-types, number of CGs, number of nodes and edges per graph, height and breadth of their hierarchies, etc.) for applications, or who have profiled runtimes versus problem size and characteristics to contact us. We would also like to obtain PEIRCE DBs in order to extract such information ourselves. Finally, we welcome others who would like to join this hardware group.

## 3.2 Objectives

Our objectives are to:

- Collect application characteristics and performance requirements, including: DB and individual graph characteristics, how application performance scales with problem size, and some target application sizes.
- Identify suitable hardware platforms to achieve the necessary speedups, emphasizing commercially available parallel systems.
- Develop and evaluate systems specifically tailored to CG processing.

---

[1]Dept. of Computer Engineering, Univ. of California Santa Cruz, Santa Cruz CA 95064, USA. donrob@cse.ucsc.edu
[2]Dept. of Electrical and Computer Engineering, University of Central Florida, Orlando FL 32816, USA. rfd@engr.ucf.edu
[3]Key Center for Software Technology, Dept. of Computer Science, Univ. of Queensland, Brisbane, QLD 4072 Australia. ged@cs.uq.oz.au
[4]Dept. of Computer Engineering, Univ. of California Santa Cruz, Santa Cruz CA 95064, USA. rph@cis.ucsc.edu
[5]Dept. of Computer and Information Sciences, Univ. California Santa Cruz, Santa Cruz CA 95064, USA. levinson@cis.ucsc.edu
[6]Dept. of Computer Engineering, Univ. of California Santa Cruz, Santa Cruz CA 95064, USA. claude@cse.ucsc.edu

- Provide dramatically increased performance with little or no increase in programming difficulty to the PEIRCE user and minimal complication to the PEIRCE programmer.

It is anticipated that uniprocessors, including mainframes, will be too slow for extremely large applications, and that single-purpose application-specific processor designs will lack the flexibility required by an evolving PEIRCE and the CG community. Parallel Prolog and LISP engines do not support the high degree of parallelism required, so other alternatives must be sought.

## 3.3 Results to Date

Research over the last year has shown that the low-level routines required by PEIRCE can be efficiently parallelized on a SIMD architecture (single-instruction multiple-data stream massive parallelism) with near linear speedup (doubling the number of processors approximately halves computation time). It has also been demonstrated that PEIRCE can be ported to a MIMD (multiple-instruction multiple-data stream multiprocessing) architecture. We have analyzed how processing requirement and speed scales in problem size for CG DB queries and evaluated communication requirements and processor allocation. The most crucial work remaining includes extending these results to other operations (such as maximal join and close matches) and issues, and then to remaining PEIRCE modules.

### 3.3.1 Large Systems

Roberts, J. D.

At last years workshop, the DB group indicated goals of systems as large as $10^9$ graphs averaging 100 nodes each and CGs as large as $10^6$ nodes. As we scale from current projects to these enormous sizes, Sowa noted that issues other than just processing power become critical, memory being key among these [25]. Our MSIMD parallelization work indicates that the optimal number of processors grows $O(n^2 \log N)$, where $n$ is the average number of nodes in a graph and $N$ is the number of CGs in the database. Memory scales $\Omega(nN)$ and dominates the cost of a system with a large number of CGs. As Sowa discussed, this indicates the need for *virtual memory* in which CGs are stored on disk rather than in memory (at about 1/10 the cost of memory), possibly organized by contexts. He also notes that as the required hardware becomes large and expensive, access by many users over internet will be appropriate and discusses this in detail. We also propose that large machines could be accomplished through *distributed processing*, connecting several smaller machines over a LAN or the proposed US high-speed optical network. These issues should be hidden from the PEIRCE user, and introduce minimal complications for the PEIRCE programmer.

### 3.3.2 Processor Allocation

DeMara, Ron

Research on the SNAP semantic network (SN) processor [19] has produced results which should be qualitatively, if not quantitatively, useful in PEIRCE work. Assigning nodes in a SN to a particular processor in a parallel system is analogous to assigning CGs in the partial order (generalization hierarchy) to particular processors. DeMara's results show that below a threshold of available parallelism (which unlike numerical algorithms can be quite high for AI) allocation has only a small impact on performance. Above this threshold, optimal allocation can be formulated in terms of marker propagation parallelism and the critical path through the applicable subgraph of the DB (which is analogous to predecessor and successor fanout in the generalization hierarchy). The best allocation strategy provides roughly a factor of 2 improvement in performance.

### 3.3.3 Communication Requirements

Noshpitz, Claude

A simulator has been built for evaluating communication patterns in a parallel CG system. CG reference profiles for the Morph chess program have been collected, and evaluation has begun. Additional reporting may be added to PEIRCE to collect similar profiles for other applications.

### 3.3.4 MSIMD Parallelization, Tailored Architecture

Roberts, J. D., Levinson, R., Hughey, R, and Ellis G.

The hardware group members at the University of California, Santa Cruz, are designing a parallel computer, the MISC Machine, tailored specifically to CGs as well as to a variety of other AI paradigms. Its overall architecture features multiple SIMD arrays, each with a supervising microprocessor integrated into a MIMD communication network, MSIMD (multiple SIMD). Its development also includes an associative processing model, in which collections of data are accessed and manipulated by content, to hide low-level parallelism and make programming the multiple levels of parallelism manageable. MISC is described in somewhat more detail in last years workshop proceedings [48].

Our key results are presented in a paper in the main session of this conference [49], which we summarize here. First, we have shown efficient (linear speedup) MSIMD algorithms for subgraph isomorphism (projection), DB query, and reflexive-transitive closure coding of concept-type and generalization hierarchies. Subsumption and other operations on the coded hierarchies can then be performed in constant or logarithmic time. Our current parallel DB query combines Levinson's Method III [43] for DB search pruning and Ullmann's refinement for pairwise graph comparison [59]. Method III empirically reduces the number of necessary graph comparisons to $O(\lg^2 N)$ where $N$ is the number of graphs in the DB. Ullmann's subgraph isomorphism refinement algorithm is empirically $O(n^4)$ expected case, where $n$ is the number of nodes in the larger graph. Willett et al. present a parallelization of refinement to reduce time to $O(n^2)$ using $O(n^2)$ processors [61], and we propose that $O(\lg N)$ projections can be performed in parallel without significant loss of efficiency. We are thereby able to reduce time for a DB query from $O(n^4 \lg^2 N)$ to $O(n^2 \lg N)$, using $O(n^2 \lg N)$ processors. For a DB of 64K graphs averaging 16 nodes each, this represents a parallelization speedup of three orders of magnitude, in addition to the two orders of magnitude search reduction from Levinson's algorithm[7]. Our key analytical result is that the above parallelization is faster than exhaustive search even with a processor for every graph in the KB. After investigating parallelizing the core low-level routines of Method III and hierarchy coding (generalization and concept-type), we are beginning to develop methods specific to the associative MSIMD architecture. We have also begun coding low-level DB routines for the MasPar-2, a commercial SIMD parallel machine which offers peak performance ranging from 4,250 to 68,000 MIPS.

The above asymptotics reveal excellent parallel scaling in $N$; a linear increase in the number of processors (or individual processor power) provides an exponential increase in the maximum number of graphs in the DB. Small increases in the average size of the CGs works even more strongly against us, however. *Type-definition expansion (join) versus nested comparison in projection becomes a crucial implementation issue.* Although nested comparison would clearly miss possible projections, there is an *exponential* advantage in DB size and a polynomial time advantage.

The MISC Machine is intended to be highly scalable, with implementations ranging from workstation accelerators to "personal superminis" to supercomputers with a million or more SIMD processors. Rough estimates indicate that a workstation accelerator with 1,024 processors could be sold in the neighborhood of $20 thousand, and would offer at least 2 orders of magnitude speedup over current workstations, dramatically increasing feasible DB and graph sizes.

[7]For larger DBs, search reduction would provide greater speedup than the parallelization.

## 3.4 Interaction With Other Subgroups

With our current focus on retrieval and hierarchy coding, our group's work relates closely with that of the DB subgroup. We encourage further communication with others dealing with hardware-related issues.

As our subgroup's work progresses, it will be appropriate to coordinate with languages and other aspects of the STDS subgroup. The Santa Cruz group is considering the development of an associative programming language for programming the MISC Machine. It would be developed in order to hide low-level parallelism through operations on collections (such as sets and tuples) with notations at the level of AI algorithm pseudocode, and to make the multiple levels of parallelism manageable to the programmer. Ideally it would include a cross-compiler for C++, allowing programs to run on other platforms, and the MISC Machine should also be programmable in C++ with associative and MSIMD objects and methods. We are also considering the language Sather.

> Sather is an object-oriented language that is particularly well suited for the needs of scientific research groups. It is designed to be very efficient and simple while supporting strong typing, garbage collection, object-oriented dispatch, multiple inheritance, parameterized types, and a clean syntax. It compiles into portable C code and easily links with existing C code. [47]

It also features a large class library. Most importantly, a public domain version is currently available, and both MIMD and SIMD parallel versions are under development.

# 4

# Implementing a CG Platform for Question/Answer and DataBase capabilities

**Boris Carbonneill**[1]
**Ollivier Haemmerlé** [2]

## 4.1 Objectives

We have developed a platform providing a set of classes allowing to represent and manipulate a complete CG environment. Using this platform, we are currently working in two ways:

- Question/Answer system [10] [11] providing different retrieval strategies using CGs operations such as projection, graph splitting, common generalization, type contraction and expansion...
- an implementation of the concepts introduced in the article: "Conceptual Graphs for Relational Databases" [7], which will be presented at the 1st ICCS (in this paper, we will show CG's capabilities to represent and extend Relational DataBases).

In the context of PEIRCE, we think that our work on Question/Answer system complements the work being done in the DB subgroup. This is why we will be concentrating on this part of our work in the following paper.

## 4.2 Cooperative research efforts

At the moment, we are involved in two cooperative research projects:

- The French PRC-IA project (Coordinated Research Project on Artificial Intelligence), whose objectives are CG model formalization, logical interpretation and implementation of a set of CG tools (graphical interface, development platform...)
- The European MENELAS project. The objective of MENELAS is to perform such an in-depth understanding of patient discharge summaries (PDSs) in French, English and Dutch, as to allow users to access the information they contain. This system uses CGs to represent and query this information.

## 4.3 The platform

The platform [32] was developed using the C++ language on UNIX stations.

The higher level class is the `program` class in which a set of tools for parallel programming and the body of user's application are present. A `program` object is composed of a set of `memory` objects and a

[1]LIRMM, 161 rue Ada, F-34392 Montpellier Cedex 5
Tel (33) 67 41 85 79 - Fax (33) 67 41 85 00
e-mail: bc@lirmm.fr
[2]haemmer@lirmm.fr

set of procedures manipulating these memories.

A memory object is a complete set of data, allowing one to work in a CG environment. It is composed of a support object and many hierarchical memory zones, containing CGs and lambda-abstractions.

A support object contains the lattice of the concept types, the partially ordered relation types set, and the conformity relation that describes it.

A lattice object is represented by two arrays: the first one gives the order relation between two concept types, the second one gives for each pair of concept types its upper and lower bound.

A set of relation types object is treated like a lattice object, but only the first array is defined.

A conformity relation object is an array giving for each referent identifier a pointer to the minimum concept type associated to this referent.

A conceptual graph object is defined by a set of concept nodes and a set of relation nodes. Edges are represented for each concept node and relation node objects by a list of neighbours (this list is ordered and has a fixed arity in the case of relation nodes).

A concept type object is defined by an identifier, and if necessary by lambda-abstractions which define type or graphs of constraints.

A relation type object is defined by an identifier, and by an array defining the minimal concept type allowed for each neighbour of a relation of this relation type. As for the concept type, it is possible to define a relation type by using a lambda-abstraction.

A lambda-abstraction object is characterized by a reference to a conceptual graph object, and by an array of pointers on each concept node whose referent is a variable.

A text file exchange format is used by the platform. This format allows one to represent a lattice of concept types, partially ordered set of relation types, conformity relation, and sets of CGs and lambda-abstractions. Here is an example of such a file:

```
Begin
    Support:Test;
        Lattice:
            ConceptTypes:
                Entity;
                Action;
                HumanBeing;
                Thing;
                Sleep;
            EndConceptTypes;
            Order:
                HumanBeing < Entity;
                Thing < Entity;
                Sleep < Action;
            EndOrder;
        EndLattice;
        TRelSet:
            RelationTypes:
                agt{Signature:2,HumanBeing,Action};
                obj{Signature:2,Action,Entity};
            EndRelationTypes;
```

```
        EndTRelSet;
        Conf:
            Bob,HumanBeing;
        EndConf;
    EndSupport;
    Graph:gc1;
        Nature:fact;
        Set:ens1;
        Concepts:
            c1=[HumanBeing:Bob];
            c2=[Sleep:*];
        Relations:
            r1=(agt);
        Edges:
            r1,c1,1;
            r1,c2,2;
    EndGraph;
End
```

## 4.4 The Question/Answer system

### 4.4.1 Overview of the system

From the initial question, a question tree is developed. The system prioritizes each question according to different criteria which are not presented in this paper. Questions are put into an agenda, and are selected by order of their priority. A selected question is projected onto each fact of the knowledge base. If a projection exists, it is returned as an answer to the question (this answer can be incomplete), and will be used to build the final answer. If no projection is returned, all questions which could be created from the current question are generated, and put in turn into the agenda.

We chose to work exclusively on the question graph. Indeed, generation of new facts brings about a combinatorial explosion of the knowledge base, which makes looking for answers impossible in sufficient depth.

This process is reiterated until a positive answer is obtained, or a failure is registered. A failure occurs when there are no more questions in the tree, or when a stop-condition has been satisfied (these stop-conditions are defined by the application). If a search fails, pertinent answers are returned (if there are any).

### 4.4.2 The Projection operation

This basic operation has been implemented in two different ways:

- on the one hand with a standard backtracking algorithm,
- on the other hand, with a polynomial tree-projection [2]: the query graph is transformed into a tree, projections of this tree are polynomially searched, then projections of the initial query graph are recomposed.

Note that these two algorithms are able to return all the projections of a graph (injective or not).

### 4.4.3 The Split operation

It may happen that an answer cannot be found when the information is split between several CG facts. In this case, the query graph is split in two by duplication of an individual concept node. The two resulting graphs are then viewed as two new questions. If answers to each of these new questions can be found, the answer to the initial query will be produced using a join between images of the duplicated concept.

### 4.4.4 The Concept type relaxation operation

Suppose we have the following query "Does Mr Smith have a car?", and the fact "Mr Smith has a motorbike". The projection from the query to this fact is impossible. However, if car and motorbike are subtypes of a vehicle type, it would be relevant to produce the new query "Does Mr Smith have a vehicle?" by relaxing the car concept type to vehicle. Projecting this new query could allow the system to return answers such as "No, but Mr Smith has a motorbike". Intuitively, this relaxation can be done by searching the upper-bound of car and motorbike. But from a semantic point of view, this relaxation is of no interest if the semantic distance between the two concepts is too large. In the first step, we defined more precisely the notion of semantic distance; then we implemented a heuristic limiting this distance during the relaxation stage.

### 4.4.5 The Graph truncation operation

Suppose we have the following query "Does Mr Smith have a car with an electric engine?", and the fact "Mr Smith has a car". The projection from the query to this fact is impossible. However, we can produce a generalization of the query by removing the part "with an electric engine". The projection of the new question allows the system to return answers such as "I don't know, but Mr Smith has a car".

### 4.4.6 The Expansion/Contraction operations

If there are defined concept types in the types lattice, the projection of the query may be impossible due to a different form of the information in the query and the fact. As opposed to the previous three operations which implied a difference in semantics between the initial question and the question produced, in this case only the form of the question is modified. Type expansion is done using a join between the genus concept of the type definition and the concept of the query which must be expansed. Type contraction uses Sowa's algorithm [55].

# 5

# GRIP: Toward a Standard GUI for Conceptual Structures

**P.W. Eklund**[1]
**J. Leane**[2]
**C. Nowak**[3]

ABSTRACT
A major advantage of conceptual graphs is their ease of use in the graphical form. This paper is intended to present some general guidelines for building a suitable conceptual graph interface and the details of a particular implementation currently under development. The resulting software is to be completely integrated with the existing PEIRCE code, built to conform with the PEIRCE standards of portability and robustness, and is to be used by the PEIRCE community for representation, acquisition and querying of conceptual graphs.

## 5.1 GRIP Software

The software under development at the University of Adelaide is the authors' efforts to provide a definitive prototype implementation for the drawing and editing of conceptual graphs capable of the providing graphical operations required by the Graphical Editor and Display (GRIP) research group: a subgroup of the PEIRCE project.

The software (locally referred to as *GrIT*) must conform to PEIRCE specifications [23]. It will arrange and display the core graph ADTs on a graphics view surface and conversely translate graphs constructed with the interface into the core ADT structures. The ADT structures will be derivations of those used by Ellis' (Conceptual Database Subgroup) PEIRCE database [22]. Accordingly, *GrIT* will have direct access to the functionality of the PEIRCE database and maintain complete integration and consistency with any forthcoming release of the PEIRCE code.

The "stub" code for the widgets of *GrIT* were generated using a SunSoft product, Devguide [57], [58]. The XView widget set [33], an implementation of the OPEN LOOK standard (the GUI standard supported by Sun and AT&T for X Window System V Release 4) was chosen to implement the graphics because (i) it lends itself naturally to object-oriented programming in C++, (ii) is a public domain software, (iii) is supported by Devguide, (iv) executes expectedly under different window managers.

The design was based on the guidelines for GUIs presented in [28].

Previous work (e.g. [20] and [21] — cf. [23] pages 36 – 37) gives considerable experience, although the existing software's platforms are at odds with the requirements of PEIRCE.

Conceptual Graphs are constructed by using the drawing and editing facilities of *GrIT* and entered into the PEIRCE database by "asserting" the graph. Once drawn the conceptual graphs may be reloaded, edited, etc. Graphs can also be drawn and used to "query" the database. All responses to such queries are graphically displayed. The task of entering a conceptual graph in a linear form and displaying it graphically is a non-trivial one and the problem is currently being examined. In the short term conceptual graphs may be entered in their linear form and as the linear form is parsed the user positions the constructs, which are then automatically drawn together with the appropriate arcs, labels, etc.

[1]Department of Computer Science, The University of Adelaide, Adelaide 5005, Australia, email: peter@cs.adelaide.edu.au
[2]email: josh@cs.adelaide.edu.au
[3]email: nowak@cs.adelaide.edu.au

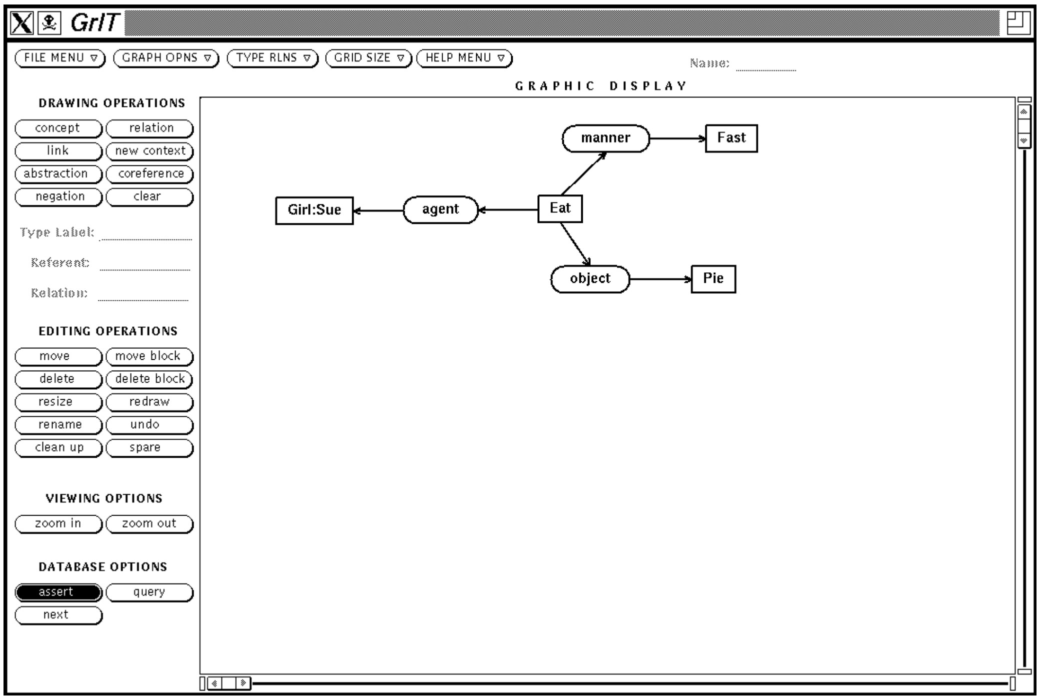


FIGURE 5.1. Screendump of *GrIT* showing an assertion

Once a conceptual graph is drawn and asserted it resides as an ADT in the PEIRCE database. The ability to display such a graph constructed with *GrIT* in linear form is then derived directly from the functionality of the PEIRCE database.

The implementation is considered to be general enough to be used for a variety of applications and sufficiently extendable to accommodate the needs of the entire conceptual structure community. However, at the current stage, it forms a testbed to experiment with possible solutions to various problems, and to make decisions about the relative importance of possible operations. Also, the necessity of ensuring a quality integration with the existing PEIRCE code requires more cautious approach towards implementation.

The picture below (Figure 1) presents the appearance of the current version of *GrIT*, with the "assert" operation selected. Specific operations will be described in Section 5.5.

## 5.2 Research efforts

In this section we describe our research efforts which extend the prototype implementation.

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 long term objectives is to develop interfaces which 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 an explicit non-monotonic 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 of 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 practice 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.

Research undertaken by the authors (Nowak, under the supervision of Eklund) is aimed at addressing (1) and (2). Realizations of such tasks in software will represent advanced facilities for future generations of the core system.

Other areas of interest include automatic node placement, knowledge acquisition aspects of conceptual structures, natural language interfaces, graph-based output from text retrieval and database queries.

## 5.3 Relationship to other subgroups

Close cooperation between the Graphical Interface and Database Subgroups is necessary to achieve a high level of integration. As discussed earlier, the data structures and retrieval mechanisms developed by the Database Subgroup are being re-used at a high level in *GrIT* (some modules in *GrIT* are specializations of classes of the PEIRCE code).

The GRIP Subgroup hopes to satisfy the demands of the entire CG community, with respect to graphical interfaces for conceptual graphs. The expansion of the GRIP implementations as a response to specific requirements from other groups is expected in the future.

## 5.4 Platform

The product is being implemented in C++ using an object oriented methodology. Although the XView Toolkit provides many high level features, all the graphics code is programmed at the *Xlib* level.

The design of the software will follow the principles and criteria for object-oriented programming [44]. Useful implementation suggestions for object-oriented GUIs can be found in [4]. In keeping with these principles the interface will remain as independent from the application code (data structures, graphic code, etc.) as is practical. All button/menu selections will invoke callbacks to a screen handler which will in turn carry out the required action. Similarly, the application code will have little knowledge of the interface, allowing either the application code or the interface to change almost independently, via the screen handler. This will facilitate the exporting of *GrIT* to different graphics environments, e.g. Apple MacIntosh.

## 5.5 Operations provided by *GrIT*

In this section we shall enumerate the operations and features to be provided by *GrIT*. As system development is to be incremental there is a need to discern between fundamental conceptual graph operations and the more advanced facilities. Furthermore some of the functionality of *GrIT* will depend directly on the PEIRCE database.

Operations considered so far as desirable include:

- **Operations on Files**: In general conceptual graphs may be represented in either graphical or linear form. Although *GrIT* deals with graphical forms it is desirable to both read in and write out linear forms. The former involves the problem of automatic node placement and is currently being examined. The latter option is available. The pull-down menu provides the load, save, new and quit operations.

- **Operations on Conceptual Graphs**: Certain graphs are declared to be canonical. The operations *copy, join, simplify* and *restrict* are formation rules which can be used to derive new canonical graphs from other canonical graphs ([53], pages 90 – 96). As these operations involve algorithms which copy graphs, graph pattern match and merge graphs at the ADT level the implementation of these operations may be done in conjunction with the Database Subgroup.

- **Type Lattice Relations**: Semantic meaning is given to conceptual graphs by the underlying type hierarchy. A type hierarchy is a lattice of type labels, and each concept type must exist in the lattice to be sensibly defined. Its meaning is essentially defined by its relative position in the lattice. The pull-down menu TYPE RLNS provides the following operations:

  - **subtypes/supertypes**: determining the subtypes/supertypes of a concept,
  - **max. com. subtyp/min. com. supertype**: determining the maximum common subtype/minimum common supertype for a given pair of concepts.

  The type lattice is automatically loaded once a *GrIT* session has been initiated, and may be displayed graphically.

- **Grid Size**: Allows the user to choose vertical and horizontal spacing between invisible grid lines simplifying the operation of placing objects manually.

- **Help Menu**: Each XView widget can be supplied with a help facility, activated by placing the mouse pointer over the widget and pressing F1 function key. The help menu describes this facility and the conceptual graph notation used.

- **Drawing Operations**: The user will be able to draw concepts, relations, links between them, coreference links, context boxes, negation of contexts and abstractions corresponding to user defined types. Naturally, the difficulty lies in maintaining consistency between the state of the data and the information displayed on the screen.

  Concepts and relations require labels to be entered by the user. The method used in *GrIT* is to provide "textfields" which in general appear in a semi-invisible state. When input is required the relevant textfield appears in bold and automatically gains keyboard focus. For example, if the user is entering a label for a concept, ending the label with a colon will cause the type label textfield to disappear and the referent textfield to appear and have the keyboard focus. This scheme has an advantage of requiring minimum user interaction and enforcing a strict sequencing of events.

- **Editing Operations**: These operations are not specific to conceptual graph interfaces and include moving objects, moving blocks, deleting blocks, resizing, redrawing, renaming, undoing previous actions and cleaning up (realigning user selected portions of the graph). The implementation of these operations follows naturally from the object oriented methodology used.

- **Viewing Options**: Navigation of large conceptual graphs is via scrollbars. Additionally, the ability to zoom in and out provides the user with an overall perspective of the graphs under consideration.

- **Database Options**: The user may assert a graph — this results in the placement of the graph in the PEIRCE database. The "query" operation in turn results in all graphs that are as specialized as the querying graph. For example, given the assertion made in Figure 1, the following Figure 2 corresponds to the query (top half of the screen) and the response (lower half of the screen). Selecting the button "next" results in displaying the next response, if one exists (cf. Prolog). It is envisaged that other subgroups may insert their own code into the 'assert' and 'query' modules, allowing interaction with their respective projects.

## 5.6 Advanced Features

Various advanced features are being considered, and may be implemented if a perceived need exists.

One such feature is the automatic placement of graphical objects, and as such might require special research effort. One possible method has already been commented on — the prompting of the user for positions whilst parsing the linear forms. A second solution might be to break the task of placing objects on the screen into two subtasks, namely placing objects within contexts and then placing whole contexts on the screen, with some degree of user interaction.

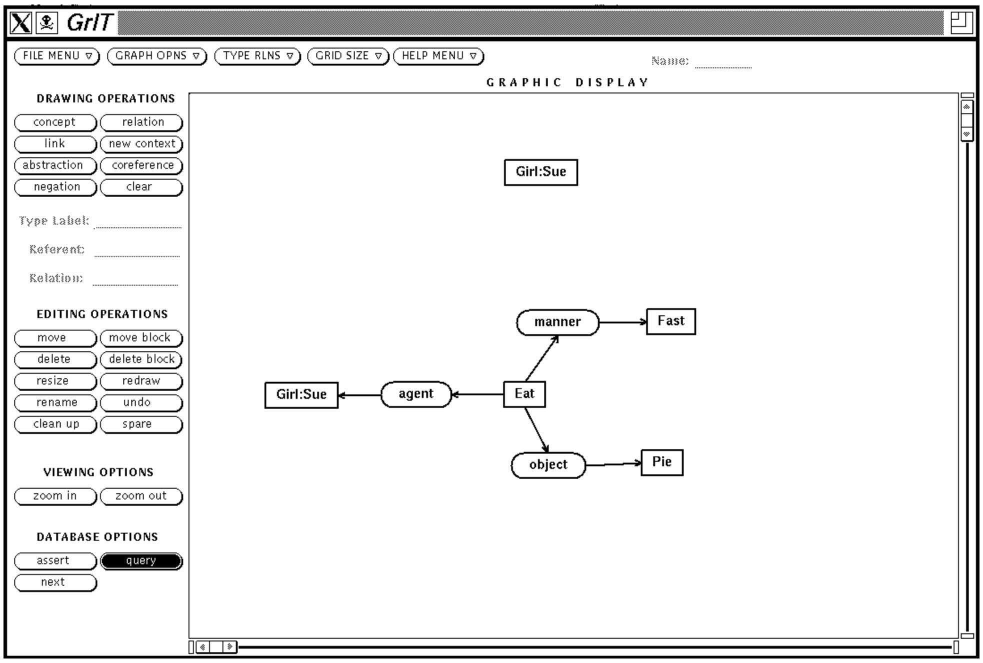


FIGURE 5.2. Screendump of *GrIT* showing a result of a query

Another extension is to add a third dimension to the graphic interface. Although drawing in three dimensions will not be supported by the current version of *GrIT* (the current release of X Window has little external support for three dimensional graphics, e.g. SunPHIGS 2.0), the idea of having several graphical planes overlapping each other, each filtering information being passed to the topmost plane, is interesting. Different planes could be raised to the top to give different perspective of the conceptual graphs.

The use of color, either by itself or combined with the idea of overlapping planes, would give the interface greater expressiveness. For example, concepts within a given type hierarchy may be given a particular shade of blue, ranging from deep blue for the most general concepts, to light blue for the more specialized concepts. The semantic correctness of relations joining such concepts could then be ascertained by inspection.

In [50] Robertson et al. discuss how color, 3D graphics and animation may lead to reducing the cost of finding, accessing and manipulating information. In moving away from the desktop Windows-Icons-Menus-Pointing (WIMP) metaphor to a system which provides a larger virtual work space by increasing the density of information via advanced graphics technology, the user is presented with a system more suited to his perceptive and cognitive capabilities, and hence overall interaction time is considerably reduced. A new version of X Window (explicitly supporting 3D graphics) together with 3D widget sets and toolkits will allow the inclusion of such features.

## 5.7 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 users 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. Devising an interface which is sufficiently flexible and powerful enough to be of real value in most environments is largely a specification problem. The nature, style and complexity of the tool construction are considered by the authors to be relatively straightforward, but satisfying the GUI requirements of the PEIRCE community may be a more demanding and challenging task.

# 6

# LEARN: Domain-Independent Experience-Based Learning Modules and Conceptual Clustering

**Robert A. Levinson**[1]
**John Amenta**[2]
**Guy Mineau**[3]
**Michel Liquiere** [4]

## 6.1 Introduction - R. Levinson

Domain-independent learning modules state-space search algorithms are being developed for Peirce and are based directly on the following tenets or guiding principles for developing machine intelligence:

1. Intelligence is optimal problem-solving in pursuit of specific goals under resource constraints. (Note that no mention is made of human cognition or consciousness).
2. Given this definition domain-independence and adaptability are fundamental aspects of intelligence. (To not adapt, if economical, is sub-optimal)
3. Single and multi-agent state space search problems are a large and important class of problems in which to develop and study machine intelligence
4. The task before us is to develop fully domain-independent methods for working in state-space search domains.
5. The knowledge required to perform well in these domains is embodied in the definition of the domain and the mathematical structure of the state space. I.e. the degree that a heuristic is good is explainable within the framework of the mathematics of the space.
6. Experience in a state space reveals mathematical structure that can be exploited by an optimal problem solver (what is not revealed, need not be exploited).
7. For many practical problems, experience alone reveals enough structure to lead to efficient problem-solving.
8. The underlying mathematical structure in these domains is independent of the labels given to the conditions and operators in the state-space definition.
9. This mathematical structure is embodied in the interactions of the conditions, defined by the operators of the domain, the relationship of this interaction complex to a given state, and the relationship of the given state to the goal state.

---

[1]Dept. of Computer Science, Univ. of California Santa Cruz, Santa Cruz CA 95064, USA. levinson@cis.ucsc.edu
[2]Dept. of Computer and Information Sciences, Univ. California Santa Cruz, Santa Cruz CA 95064, USA. wjames@cis.ucsc.edu
[3]Laval University Quebec CANADA, mineau@vm1.ulaval.ca
[4]Reunion University France, mili@iremia.fr

10. The effects of such interaction complexes is entirely domain-independent and is governed by regular laws just as such interactions in matter and energy are governed by the laws (not necessarily all known) of physics.

11. These laws once discovered and exploited by computers will make them intelligent under the definition given above.

12. Computers may be an important tool in the process of discovering these laws.

With these tenets in mind and the ultimate goal of discovering the laws of state-space search before us we are working on these separate but absolutely complementary projects:

- GENERIC-REPRESENTATION: a graph-theoretic Conceptual Graph representation of the definition of a single-agent or multi-agent state-space search problem that is independent of all domain-specific or arbitrary labels. (i.e. a program that can convert the declarative definition of a state space into a more generic, but equivalent, graph-theoretic representation).
- INFORMED-LEASRNER: A program that performs well in state-space search given this graph-theoretic definition of the state space, supplied domain-independent heuristics and experience.
- BLIND-LEARNER: A program that performs well on state-space search problems given no definition of the state space or pre-supplied heuristics - just the rewards at the end of the game, experience and legal states (as raw bit vectors) to choose from at each choice point.
- APSII: A programming environment that allows experiments to be easily designed and carried out on the above topics.

None of these projects is designed to directly produce the laws of state space search but to gradually give us an understanding that could lead to these laws and to their empirical validation. Meanwhile work on these problems increases the power of the machine. The basis for our work on each of the four subprojects is described below. The summation of this work is a fully domain-independent adaptive search and game-playing system known as MORPHII that is in the process of being implemented. MorphII is a successor of the Morph chess system [42, 30] that has achieved approximate novice strength despite using just 1-ply of search and few human-supplied heuristics. Referees have been built for Tic-Tac-Toe and backgammon in addition to chess. A super-referee is being built based on the generic representation that will allow MorphII to play and learn any game or search problems - provided the rules of the domain are supplied using a generic representation.

## 6.2 APSII: Domain-Independent Games environment - J. Amenta

The old Morph system [42, 41, 30] has now entered a new phase of development, no longer dependent on one domain for its learning and evaluating routines. Through an extensive recoding of pre-ansi C, to C++ Object Oriented code, the new Morph now has the capabilities of working in any pre-defined games environment, with any number of Agents, with any combination of Knowledge Structures.

APS II can be divided into three major subsystems: Agent, Knowledge, and Referee. Each subsystem is run as separate units with well defined interfaces that communicate an Object Oriented fashion with each other, and are managed overall by a central manager routine which activates and deactivates system objects on demand. For example, when APSII is activated in a domain where multiple agents can be elected the manager routine will instantiate agent objects by user option, and then inform the Referee subsystem of the agents which have been activated.

To start, an Agent in the system represents a player in a given game environment. For example, in Chess, two Agents must be selected, White and Black. In Bridge, four. However, in certain game environments, such as Craps, any number of Agents might participate (up to memory limitations). APSII, therefore, has been designed to manage multiple agents as required for the domain (game or other) that it is to operate in. Some Agent types have already been defined for the system, among them: Human, Pipe, Random, File, C++ Object, APS. The Human agent allows direct human input in the game environment, while the Pipe agent defines an agent which APSII can hook up with through the Unix multi-tasking Operating system as a separate process. Random represents a random moving agent, and File an agent read from an IO file, while C++ Object Agent can be an agent which can be added on to the Morph C++ system as an Object, as long as it conforms to the inheritance rules for Morph's Agent objects.

The most important Agent is the APS learning agent (Adaptive Predictive Search) which is being built based on the domain-independent methods for exploiting mathematical structure described above. Other learning Agents could be defined, but APS has been selected as the main default. The APS agent is completely domain independent, it can be operated by Morph in any selected gaming environment, and contains by default, a Search system, that allows it to select domain operators (game moves) by looking ahead, a Learning system, which can utilize any selected Knowledge objects and evaluate them independent of what they represent in actuality, and Eval objects which can evaluate any given domain for both the Search and Learn systems. The APS Agent was created to be extremely flexible in the Knowledge types it can manipulate, and the states it can actually learn from in any given domain, along with the power of search it is to use (with a trade off in time spent in search). Thus APS can be used in applications beyond the pure research framework of this paper.

The second major subsystem in Morph II, is Knowledge. This system was built using the Peirce DB routines developed by Gerard Ellis. APS II's Knowledge subsystem, creates what we refer to as Ktype objects. Each Ktype object represents a grouping of Knowledge which Morph can then assign to an APS agent for manipulation in a given environment. It is not necessary that the APS agent know what kind of Ktype object it is utilizing, nor is the APS agent limited to only one Ktype per domain, that is, several Ktypes can be activated by the Morph manager, and then assigned to an APS agent (or multiple APS agents) for any given environment. Thus, Ktypes are not limited to single domains and can be improved through use in different domain-game settings. For example, a Ktype representing "Time" knowledge might be available for activation. This Ktype could potentially be useful in several game environments, and could be assigned to APS agents working in an activated domain.

The third subdivision of Morph II, Interacting with the Agent and Knowledge subsystems, the Referee system in Morph is responsible for domain specific requirements. That is, prior to codification of data into Morph's Conceptual Graph system, the primitive level sensing of that data must be undertaken. The Referee system is responsible for defining the restraints inherit in each domain Morph operates in. It defines the primitive level States and Operators of that domain, for example in Chess, an Operator is a move (e.g. P-K4, or Q-Q2), and a State is an entire, static board configuration, and also defines the number of agents that can or must operate in a given domain, along with goal and legal states which can be achieved.

In summary, Morph II is a new evolutionary stage in our quest to uncover the fundamental laws of computational intelligence and experiential learning. Hopefully, the taxonomy of projects and generic games lays the foundation for other learning groups to directly participate in the graph-theoretic domain-independent study of state-space search as well.

## 6.3 Conceptual Clustering - G. Mineau

This year, we worked on the development of a conceptual graph algebra used when a query is sent to the knowledge base. Generalizations of the query are found by the Levinson and Ellis method. Algebraic formulation of the query is then issued using these generalizations. Algebraic reformulation of the query is performed next, according to the characteristics of the operators of this algebra, in order to speed up the evaluation of the query. This work is still too young to be presented this year, but will be the subject of a paper next year.

This year we also studied and experimented with a classification algorithm for conceptual graphs. This algorithm is presented in the Nagle, Nagle, Gerholz and Eklund book. First, we modified this algorithm so that it would be incremental (as will be published shortly), and then we studied the impact of conceptual graph normalization on the classification potential of the knowledge base. The normalization has proven to be useful at little cost (Facilitating the creation of a multiple index on graph-described documents, Mineau, G., 1993, 2nd Int. Conf. on Information and Knowledge Management, Arlington, Virginia, November 1-5, 1993).

This leads us to believe that it would be feasible and useful to extract certain generalizations from the original graphs, and to include them into the knowledge base as organized by the Levinson and Ellis DB methodology.

However, our clustering method produces a large number of useless classes, since the method is based on a syntactical approach. This year's work will mainly concern ways of pruning the classification structure in order to keep only significant (useful) classes. Statistical methods on class membership, on the importance of characteristic description pieces, methods involving user-defined priorities and background knowledge will all be contributed.

By next year, we hope to propose a generalization mechanism which would enrich the knowledge base in a useful manner by the user or the application. By these additions, certain types of queries (about classes of the original graphs) should be answered more efficiently. Furthermore, automatic reformulation and improved evaluation of queries should be possible.

# 7

# Schema Integration and Model Translation

**Peter Creasy**
**Department of Computer Science**
**The University of Queensland**
**QLD Australia 4072**

## 7.1 Introduction

Information Systems Engineering incorporates a number of ideas and projects which examine the role of conceptual graphs in information systems. Several of these projects are currently under development. One of these [12] continues the project proposed in [13], to facilitate the integration of conceptual schemas using conceptual graphs and to aid the translation between models. Further development on this project is discussed below. This problem is also addressed in the recent ANSI report of the conceptual schema [3]. Other work currently in progress includes examination of the ANSI proposals [6] and the examination of EXPRESS [37] as one of the languages to be incorporated into the schema integration/translation project.

## 7.2 Schema Integration

### 7.2.1 ER

In [12] the we reported on a project involving schema integration using conceptual graphs as an intermediary. This idea is part of ANSI's thoughts on the *Normative Language*. The report focuses on ER. The ER example used is shown in figures 7.1 and 7.2. A linear form of ER has been developed and a translation from ER to conceptual graphs is given in [12]. The form is such that a reverse translation is simple. This is primarily due to the fact that ER relationships have a single structure which describes general, non directional associations. It is non directional in terms of reading the association in natural language. NIAM does not have this single simple structure but a compound structure. There is a role for each representation of an entity type in an association. represented by roles, In conceptual graphs we use two structures to represent such meaning - one for each direction.

A schema theory analogous to the conceptual graphs theory has been developed. Generalisation involves the replacement of a type in a supertype. In the case of a relationship this is essentially a relaxation of constraints on the relationship. The constraints on the relationships are handled by an ER constraints lattice. An example in the paper shows a schema hierarchy where a mandatory N to 1 relationship is generalised to a mandatory M to N and also to an N to 1. The joins over an entity type demonstrates the integration of schemas.

Using data dictionaries, which are defined with a conceptual graphs language, have been proposed. Such dictionaries will facilitate this join process. We envisage that in defining a UoD such dictionaries would be defined. Additionally, ontologies (see CCAT section) for various domains would supplement such dictionaries.

### 7.2.2 Implementation

Recent development with four research assistants has meant that the translator, which takes the ER linear form and converts it into conceptual graphs is operational. A schema matcher (MATCHES) has been developed and considerable work done towards the implementation of a maximal join algorithm.

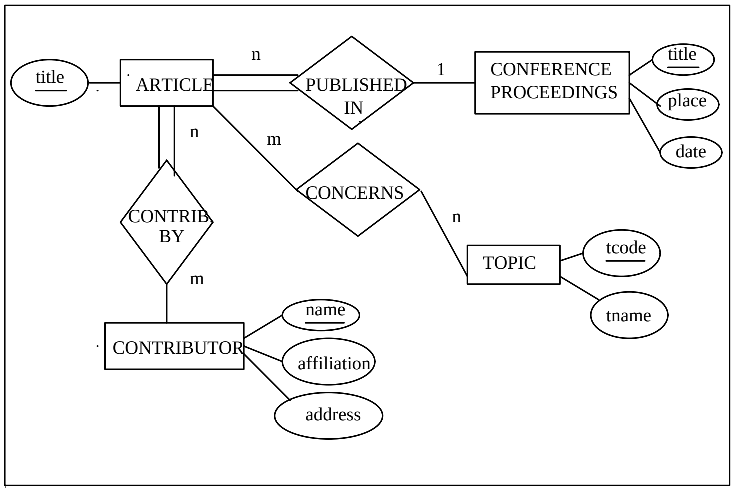


FIGURE 7.1. View 1 of the UoD as an ER diagram

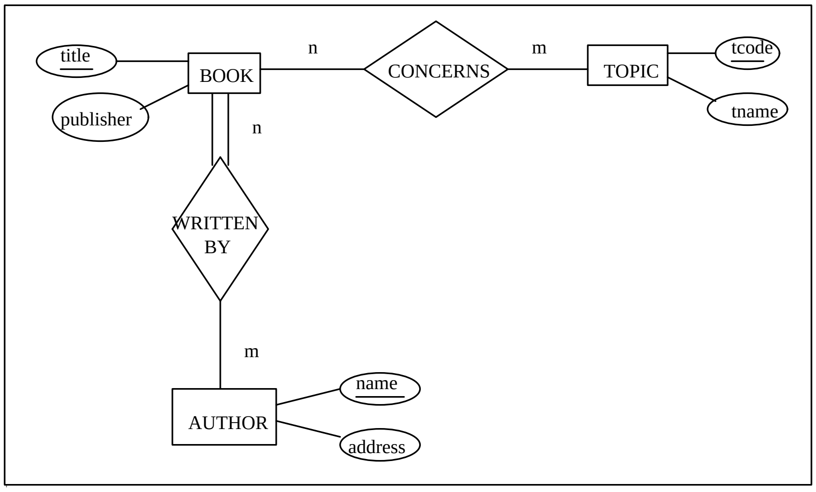


FIGURE 7.2. View 2 of the same UoD

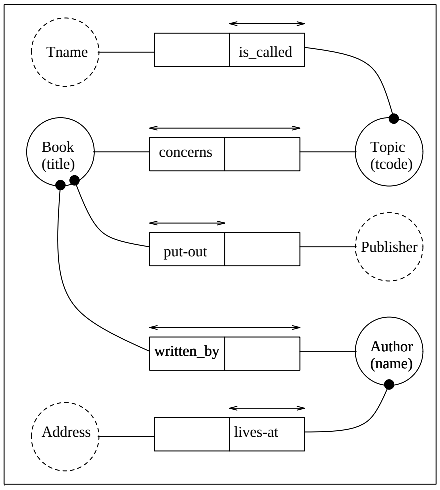


FIGURE 7.3. View 2 in NIAM

MATCHES takes two schemas and determines if one is more general than the other. The process is: Takes two graphs and tries to find a set of bindings between nodes of graph1 and nodes of graph2 for which the generalisation is true. For each node of graph 2 it first finds the set of candidate bindings from graph 1. A node in graph 1 is a candidate if the node "matches" the first. The node "matches" the first if it is a supertype of it and (for relations) if it has the appropriate arity. This list of candidates is then sorted in decreasing order of arity. The lists are then sorted in increasing order of size. If the lists are the same size, they are ranked in decreasing order of the arity of the nodes in graph 2. The candidate lists are then exhaustively searched for a complete set of bindings which produce a valid generalisation match. To be a valid match, every node in graph 1 must be a subtype (or equivalent) to its binding node in graph 2, the arities of the binding relational nodes must be equal, and every mandatory relational node in graph 2 must be bound to some node in graph 1.

### 7.2.3 NIAM

The modelling language NIAM [60] is semantically richer than ER.[1] Traditionally, NIAM (Natural language Information Analysis Methodology) is supposed to be based on natural language. Each fact type (ER relationship type) corresponds to an "elementary sentence" from natural language. For example, we could abstract from the sentence "The book with title 'The Lycian Shore' *concerns* the topic with topic code Trav" to obtain the fact type of figure 7.3. The other role name would be obtained from the sentence "the topic with topic code Trav is *found in* the book with title 'The Lycian Shore'."

The NIAM schemas for the UoD are shown in figures 7.3 and 7.4. This has been oversimplified. We have represented what are normally thought of as attributes as label types (the dashed circles). These are usually thought of as "naming" the entity types. But we have used them essentially as attributes. The identifying label type names are shown in parentheses beneath the entity type names.

A linearisation of NIAM [35] has since been developed. The linear form of figure 7.3 is:

```
(BOOK) -
       - (_title_)
       -m-[concerns][]-n-(TOPIC) -
                             - (_tcode_)
                             =n=[is-called][]-1-(.TNAME.),
       =n=[put-out][]-1-(.PUBLISHER.)
       =m=[written-by][]-n-(AUTHOR) -
```

[1]ER is, in general, considered to be a subset of NIAM. Although this view depends upon which version of ER is being considered.

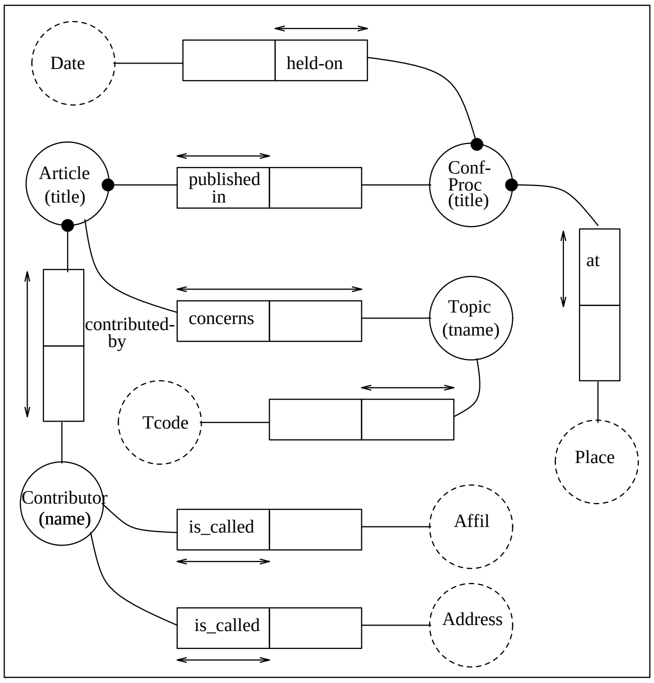


FIGURE 7.4. View 1 in NIAM

```
- (_name_)
=n=[lives-at][]-1-(.ADDRESS.),,.
```

We could translate the NIAM to conceptual graphs and get essentially the same conceptual graph schema as we did with ER. However this is a somewhat artificial as the NIAM case is not as simple as shown in the example. This is due to there only being an entity type in NIAM and not attributes. This makes it difficult to identify attributes using syntax only and to carry out both the reverse mapping and the integration with ER, although some automatic identification of attributes is possible.

However in [14] we have proposed a modification to NIAM such that it is possible to have a form which is similar to ER. ER is generally written in an "abstracted" form whereby the attributes are not shown on the ER diagram. We have argued that it is quite natural for users to distinguish entities from attributes.

We have thus distinguished major entity types and re-written them as compound entity types, wherein the "attributes" are attached to the inside of the entity type "box", as in figure 7.5 . This demonstrates the compound entity type *CONF-PROC* with its three attributes, *title, place* and *date*. In this case the identifying attribute is in parentheses as usual. This additional category of entity type we have introduced is not strange to NIAM which happily regards the fact type as an entity type.

We have drawn the entity type a slightly different shape to the usual ellipse, but this helps accommodate the internally drawn attributes and is not at variance with some forms of NIAM. We can then carry out the abstraction process as we suggested for ER and redraw the compound entity type without its attributes. The shape reminds us that there are attributes. We envisage that CAiSE tools would readily switch between these forms. This version of NIAM permits us to map to conceptual graphs in a form more readily amenable to reverse mapping and integration.

Some progress has been made on incorporating NIAM into the PEIRCE system. The translation to conceptual graphs has been partly developed. A modification of the ER parser is in progress. The feasibility of this has been shown. As mentioned the reverse translation is not as simple as the ER case. The integration of ER and NIAM is currently being investigated.

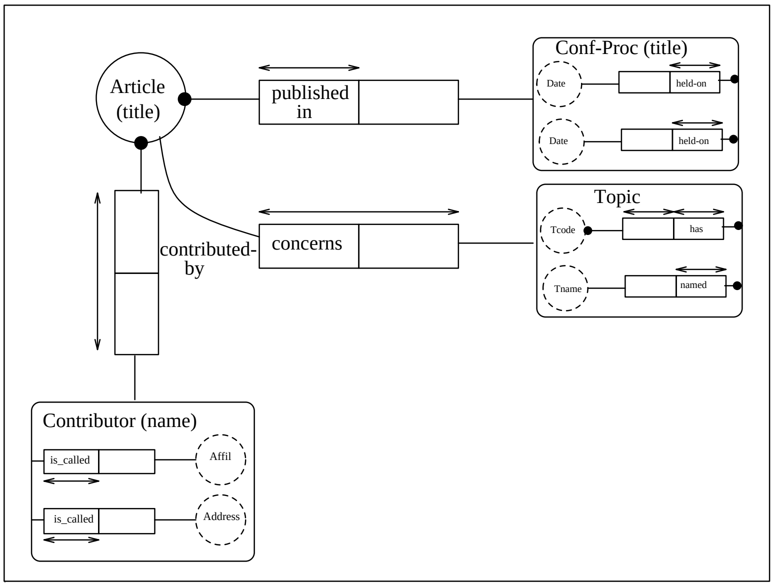


FIGURE 7.5. View 2 in modified NIAM with the attributes identified

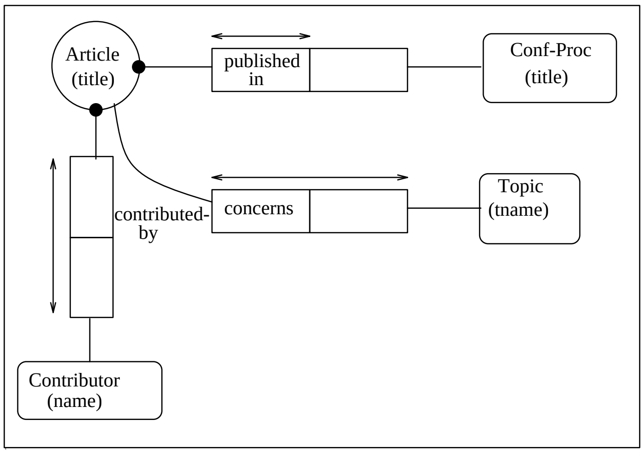


FIGURE 7.6. View 2 in modified NIAM with the attributes absorbed

## 7.3 ANSI Proposals

The ANSI X3H4.6 report [3] examines in detail possible contenders for the IRDS conceptual schema. While the IRDS is ANSI's view of it, it differs very little from the ISO view. The IRDS framework in many ways resembles FRISCO's framework [1]FRISCO93. It would be a worthwhile project exploring the differences and similarities.

# 8

# References

The following is a complete bibliography based on the citations in the previous chapters.

## 8.1 References


[1] A framework of information systems concepts. Technical Report IFIP8.1, IFIP, 1990. Interim Report.

[2] *Polynomial algorithms for projection and matching*, Las Cruces, New Mexico, 1992.

[3] ANSI. IRDS conceptual schema. Technical Report Document X3H4.6/92-001R3, 1992.

[4] P. S. Barth. An object-oriented approach to graphical interfaces. *ACM Transactions on Graphics*, 5(2):142–172, 1986.

[5] Thomas E. Bell, David C. Bixler, and Margaret E. Dyer. An extendable approach to computer-aided software requirements engineering. *IEEE Trans. Softw. Eng.*, SE-3(1):49–59, Jan. 1977.

[6] G. Berg-Cross and J. Hanna. Exploration of conceptual structures in support of data modelling, concepts, contextually distributed knowledge and the role of cognitive processes. In *Proceedings Seventh Annual Workshop on Conceptual Graphs*, Las Cruces, 1992.

[7] Claude Boksenbaum, Boris Carbonneill, Ollivier Haemmerlé, and Thérèse Libourel. Conceptual graphs for relational databases. In *1st International Conference on Conceptual Structures*, Quebec, 1993.

[8] Grady Booch. *Object Oriented Design With Applications.* Benjamin Cummings, Redwood City, CA, 1991.

[9] Timothy A. Budd. *An Introduction To Object-Oriented Programming.* Addison-Wesley, Reading, MA, 1991.

[10] Boris Carbonneill. Un module de question/réponse sur les graphes conceptuels. Technical report, Rapport de DEA de l'université des sciences et techniques du Languedoc, 1992.

[11] Boris Carbonneill and Ollivier Haemmerlé. Rock : un système de question/réponse fondé sur le formalisme des graphes conceptuels. Article submitted to the 9th Congrès Reconnaissance des Formes et Intelligence Artificielle, Paris 1994, 1993.

[12] Peter Creasy and Gerard Ellis. A conceptual graphs approach to conceptual schema integration. In [46], 1993.

[13] P.N. Creasy and G. Ellis. ISE: Information systems engineering. In [26], 1992.

[14] P.N. Creasy and W. Hesse. Simplification of NIAM, through attribute absorption. Technical Report Internal Report, University of Queensland, 1993.

[15] Alan M. Davis. *Software Requirements: Analysis and Specification.* Prentice-Hall, Englewood Cliffs, New Jersey, 1990.

[16] Harry S. Delugach. *Multiple-Viewed Approach to Software Requirements.* PhD thesis, Department of Computer Science, University of Virginia, Charlottesville, VA, May 1991.

[17] Harry S. Delugach. Analyzing multiple views of software requirements. In P. Eklund, L. Gerholz, J. Nagle, and T. Nagle, editors, *Conceptual Structures: Theory and Practice*, chapter 20, pages 391–410. Ellis-Horwood, Ltd., 1992.

[18] Harry S. Delugach. STDS: Software development standards. In *Proc. Intl. Conf. on PEIRCE: A Conceptual Graphs Workbench*, pages 13–18, Las Cruces, NM, U.S.A., July 1992. New Mexico State Univ.

[19] Ronald F. DeMara. *Parallelism, Design, and Performance of a Marker-Propagation Reasoning System*. PhD thesis, University of Southern California, 1992.

[20] P. Eklund. Using conceptual structures for knowledge-base systems development. Master's thesis, Brighton Polytechnic, 1988.

[21] P. Eklund and J. Kellet. Prospects for conceptual graphs in knowledge acquisition interfaces. In *Proceedings of the 3rd European Workshop on Knowledge Acquisition*, 1989.

[22] G. Ellis. *PEIRCE User Manual*. Department of Computer Science, University of Queensland, 1993.

[23] G. Ellis and R. Levinson, editors. *Proceedings of the First International Workshop on PEIRCE: A Conceptual Graphs Workbench*, Las Cruces, New Mexico, 1992.

[24] Gerard Ellis. *PEIRCE User Manual*, 1993.

[25] Gerard Ellis et al. Db: Conceptual database. In *Proceedings of the First International Workshop on PEIRCE*, 1992.

[26] 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. Available as University of Queensland Technical Report 241, Queensland, 4072, Australia, or anonymous ftp `ftp.cs.uq.oz.au`, `/pub/peirce/TR0241.ps.Z`.

[27] John W. Esch. Linear forms for conceptual structures. In Timothy E. Nagle, Janice A. Nagle, Laurie L. Gerholz, and Peter W. Eklund, editors, *Conceptual Structures: Current Research and Practice*, Ellis Horwood Series in Workshops, pages 595–604. Ellis Horwood, New York, 1992.

[28] M. Fitter and T. R. Green. When do diagrams make good computer languages? *International Journal of Man-Machine Studies*, 11(2):235 – 261, 1979.

[29] Michael R. Genesereth and Richard E. Fikes. Knowledge Interchange Format version 3.0 reference manual. Technical Report Logic-92-1, Computer Science Department, Stanford University, June 1992. "Living document" of the Interlingua Working Group of the DARPA Knowledge Sharing Effort.

[30] J. Gould and R. Levinson. Experience-based adaptive search. In *Machine Learning:A Multi-Strategy Approach*, volume 4. Morgan Kauffman, 1992. To appear.

[31] Sol J. Greenspan. *Requirements Modeling: A Knowledge Representation Approach to Software Requirements Definition*. PhD thesis, University of Toronto, Dept. of Computer Science, 1984.

[32] Ollivier Haemmerlé. Une plate-forme d'expérimentation de logiciels sur les graphes conceptuels. Technical report, 1992.

[33] D. Heller. *The Definitive Guide to the X Window System, XView Programming Manual*, volume 7. O'Reilly & Associates, Inc., 1992.

[34] Kathryn L. Heninger and David L. Parnas. Specifying software requirements for complex systems: New techniques and their applications. *IEEE Trans. Softw. Eng.*, SE-6(1):1–14, Jan. 1980.

[35] E. Hii. Linearisation of NIAM. Technical Report Internal Report, University of Queensland, 1993.

[36] *Software Requirements Specifications*. ANSI Std. 830-1984. Institute of Electrical and Electronic Engineers, 1984.

[37] ISO. EXPRESS-I language definition. Technical Report Report TC184/SC4/WG5.

[38] 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.

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

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

[41] R. Levinson. Experience-based creativity. In *Proceedings of Symposium on AI, Reasoning and Creativity*, Brisbane, Australia, 1991. Kluwer Academic Press. To appear in the Kluwer book AI and Creativity. Also, appears as UCSC Tech Report 91-37.

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

[43] R. A. Levinson. Pattern associativity and the retrieval of semantic networks. In Fritz Lehman, editor, *Semantic Networks in Artificial Intelligence.* Pergamon Press, 1992.

[44] B. Meyer. *Object-oriented Software Construction.* Prentice Hall International Series in Computer Science. Prentice-Hall, 1988.

[45] Bertrand Meyer. *Object-Oriented Software Construction.* Prentice-Hall Intl., Hertfordshire, England, 1988.

[46] Guy Mineau and Bernard Moulin, editors. *Proceedings of the 1st International Conference on Conceptual Structures*, Quebec City, Canada, August 4-7 1993. Springer-Verlag.

[47] Stephen Omohundro and Chu-Cheow Lim. The Sather language and libraries. Technical Report TR-92-017, ICSI, Berkeley, CA, 1992.

[48] James D. Roberts. Hardware for peirce. In *Proceedings of the First International Workshop on PEIRCE*, 1992.

[49] James D. Roberts, Robert Levinson, and Richard Hughey. Issues in parallel hardware for conceptual graph retrieval. In *Proceedings of the ICCS*, 1993.

[50] G. Robertson, S. Card, and J. Mackinlay. Information visualization using 3d interactive animation. *Communications of the ACM*, 36(4), April 1993.

[51] Gruia-Catalin Roman. A taxonomy of current issues in requirements engineering. *IEEE Computer*, 18(4):14–23, Apr. 1985.

[52] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorenson. *Object-Oriented Modeling and Design.* Prentice-Hall, Englewood Cliffs, NJ, 1991.

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

[54] John F. Sowa. *Conceptual Structures: Information Processing in Mind and Machine.* The System Programming Series. Addison-Wesley Publishing Company, 1984.

[55] John F. Sowa. *Conceptual structures: Information processing in mind and machine.* Addison-Wesley, 1984.

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

[57] SunSoft. *Open Windows Developer's Guide 3.0. User's Guide.* SunSoft, 1991.

[58] SunSoft. *Open Windows Developer's Guide: Programmer's Guide to the XView Toolkit Code Generator.* SunSoft, 1991.

[59] J. R. Ullmann. An algorithm for subgraph isomorphism. *Journal of the ACM*, 16:31–42, 1976.

[60] G. Verheijn and J. van Bekkum. NIAM: An information analysis method. In *Information Systems Design Methodologies: A Comparative Review*. North Holland, Amsterdam, 1982.

[61] Peter Willett, Terrence Wilson, and Stewart F. Reddaway. Atom-by-atom searching using massive parallelism: Implementation of the Ullmann subgraph isomorphism algorithm on the Distributed Array Processor. *Journal of Chemical Information and Computer Science*, 31:225–233, 1991.

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