The Birth of PEIRCE: A Conceptual Graphs Workbench

Gerard Ellis[1] and Robert Levinson[2]

[1] Comp. Sc. Dept., Univ. of Queensland, Brisbane, QLD 4072, Australia. ged@cs.uq.oz.au
[2] Dept. of Comp. and Inf. Sc., Univ. of California, Santa Cruz, 229 Appl. Sc. Building, Santa Cruz, CA 95064, USA. levinson@cse.ucsc.edu

Abstract. PEIRCE is a project aiming to build a state-of-the-art, industrial strength conceptual graphs workbench. PEIRCE is integrating the conceptual graphs development efforts that are taking place around the world. There are already over 40 researchers from over 8 countries involved in the project. This paper describes the PEIRCE project and the structure for managing the distributed development of PEIRCE. PEIRCE will provide a robust, portable, freely available conceptual graphs workbench that will fast track new techniques into the community; facilitate comparison of competing techniques; help researchers cooperate in development; and speed application development.

Introduction

The PEIRCE project is named after the American philosopher Charles Sanders Peirce who in 1896 developed the logic Existential Graphs [11]. Peirce described existential graphs as "the logic of the future". It is nearly a century since Peirce made that statement, and only now is his logic poised to become a reality with the widespread use of powerful graphics workstations that can run the PEIRCE workbench.

John Sowa introduced Peirce's Existential Graphs to the field of artificial intelligence, as the foundation of his Conceptual Graphs theory [12]. Conceptual graphs is an order-sorted logic with a standard mapping to natural language. A diverse, active research community has grown up around Sowa's work. Each year an international conceptual graphs workshop is held, this being the seventh. Next year the first international conceptual graphs conference will be held. A survey of current research and practice in the conceptual graph community can be found in [9].

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

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

PEIRCE will allow developers to write/draw/parse/learn large conceptual graphs programs/databases/ontologies. A module will be included to demonstrate PEIRCE's support for natural language processing - a major application for many conceptual graphs researchers. An adaptive, pattern-oriented chess playing system [7], Morph, will be used to verify and show the high-level support for learning mechanisms and databases of conceptual graphs. Morph is currently implemented in C. Much of the technology in Morph is built on top of a generic retrieval, learning and search model known as APS [2]. APS and Morph will form a valuable part of the PEIRCE Workbench. PEIRCE will also be used for knowledge based systems, graphical programming, computer assisted theorem proving, specification and verification.

PEIRCE will fast track the introduction of new techniques into the community. New techniques will be directly evaluated against old by using the same data, platforms and many of the same support routines. The project will nurture synergism between competing mechanisms. The developers will be able to assemble hybrid tools from basic tools using well documented interfaces. Prototyping of methods will be done in PEIRCE, thus the prototypes will be available immediately for evaluation by the community.

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

The PEIRCE project at the time of writing involves over 40 researchers from over 8 countries each developing part of the workbench. The diversity of development environments is itself a test of portability. But, portability is not left to chance, programming standards are being set.

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

1 Parts of PEIRCE

The PEIRCE project is divided up into modules including

- Programming Standards
- Programming Interfaces for C, C++, Prolog, and Other Languages
- Database Storage and Retrieval
- Linear Notation Input and Output
- Graphical Editor and Display
- Conceptual Catalogs (Ontologies)
- Programming in Conceptual Graphs with Constraints
- Inference/Theorem-Proving Mechanisms
- Learning Mechanisms
- Natural Language Parsers and Generators

- Information Systems Engineering
- Vision System

Each of these groups is described in terms of their dependency on other groups and the sorts of functions they provide. A working document on PEIRCE [5] further details the work of each group.

1.1 Programming Standards (STDS)

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

1.2 Programming Interfaces for C, C++, Prolog, and Other Languages (FACE)

The FACE group is charged with investigating, documenting, and standardizing calls across programming languages used in PEIRCE. The languages supported initially are C++and Prolog. Section 2 describes some of the general abstract data types needed for PEIRCE. For example a graph ADT in Prolog may call C++routines to do graph matching. A C++conceptual graphs ADT may call a Prolog routine for parsing conceptual graphs.

1.3 Database Storage and Retrieval (DB)

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

Testing the database requires large numbers of conceptual graphs, thus the database is dependent on the conceptual catalog group. In the short term the graph mechanisms are being tested on graphs (not necessarily conceptual) that are automatically generated. For example the chess graphs generated by Morph [7] are being used to test classification and compilation techniques in the generalization hierarchy [4].

Database operations are encapsulated in a partially ordered set of conceptual graphs ADT, CGPoset (see Section 2), so that implementations of each of these methods can be easily replaced. Current implementations of the generalization hierarchy (CG database) [6, 4] are data structures kept in memory. This is hidden from the users of CGPoset. In the future, large, external, distributed conceptual graphs databases can be used as a replacement or alternative to current implementations of CG databases.

1.4 Linear Notation Input and Output (LINEAR)

The LINEAR group is charged with standardizing the linear notation for conceptual graphs and providing parsers and generators for the notation. [5] contains a grammar for the standard notation. The LINEAR group records any extensions to the syntax, such as including the programming languages: Prolog, and CGC (Conceptual Graphs with Constraints).

1.5 Graphical Editor and Display (CGED)

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

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

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

1.6 Conceptual Catalogs (CCAT)

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

1.7 Programming in Conceptual Graphs with Constraints (CGC)

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

The philosophy behind CGC is that the tools available to the programmer should not be restricted. That is, the compiler should be clever enough to determine when the more powerful features are not being used and compile the program as efficiently as Prolog or simpler structures in these cases. We believe that strategies designed to compile and index large sets of graphs will outperform the general search technique of Prolog when applied to graphs. This is being investigated in the ongoing work on compilation of conceptual graphs in [4].

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

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

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

can be translated to the following Prolog program by a preprocessor

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

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

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

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

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

1.8 Inference/Theorem-Proving Mechanisms (PROOF)

The PROOF group will develop proof procedures for conceptual graphs. The alpha and beta rules of Existential Graphs [11] will make excellent interactive proof tools. Eventually proof techniques developed by this group will be written in CGC.

1.9 Learning Mechanisms (LEARN)

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

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

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

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

1.10 Natural Language Parsers and Generators (NLP)

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

1.11 Vision System (VISION)

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

2 The foundation of PEIRCE

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

1. subsumption - an operation for comparing two conceptual graphs.
2. read - a conceptual graphs parser (either linear (supplied by LINEAR) or graphical (supplied by CGED).
3. write - a conceptual graphs display system (linear or graphical).
4. decompose - decompose a conceptual graphs into a set of canonical formation rules.
5. compose - compose a set of canonical formation rules into a conceptual graph.
6. join - the canonical formation rule for joining two conceptual graphs.
7. copy - copy a conceptual graph.
8. restrict-type - restrict a concept's type.
9. restrict-referent - restrict a concept's referent.
10. simplify - remove a duplicate relation.
11. generalize - generalize a graph by removing structure or generalize concepts.
12. specialize - specialize a graph by adding structure or specializing concepts.
13. semantic-distance - calculate the semantic distance given another conceptual graph.

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

```
Label        :: C.
C            :: Concept | Relation.
Concept      :: `[` Abstraction `:` Referent `]`.
Relation     :: `(` Abstraction `)`.
Abstraction  :: Type | Lambda.
Referent     :: Coref | Individual | CGList.
Coref        :: '*' Var.
Individual   :: '#' Int.
Lambda       :: `lambda` CorefList CGList.
CorefList    :: `[` Corefs `]`.
Corefs       :: Coref | Coref `,` Corefs.
CGList       :: `[` CGs `]`.
CGs          :: CG | CG CGs.
```

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

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

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

The Poset ADT handles the type hierarchy, the generalization hierarchy over conceptual graphs, and hence the generalization hierarchy over Prolog terms, and feature terms. Using a graph's decomposition rules, the hierarchy and the graph elements can be compiled directly into C++ code rather than a data structure that requires interpretation. Figure 1 illustrates the hierarchy over ADTs in PEIRCE.

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

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

This work will make use of the retrieval methods described in [3, 8]. Some of the ideas in these papers have already been implemented and are in use in Morph [7]. There is also an implementation in C^{++} that will form the basis of PEIRCE. [3] Through restructuring and compilation of graph data these retrieval methods are expected to satisfy the efficiency requirements that will allow PEIRCE to be a viable tool for both research and industrial-strength application development.

[3] The linear notation parser described in [10] is being integrated with the database to demonstrate the database at the conceptual graph workshop and is the first test of PEIRCE's distributed development.

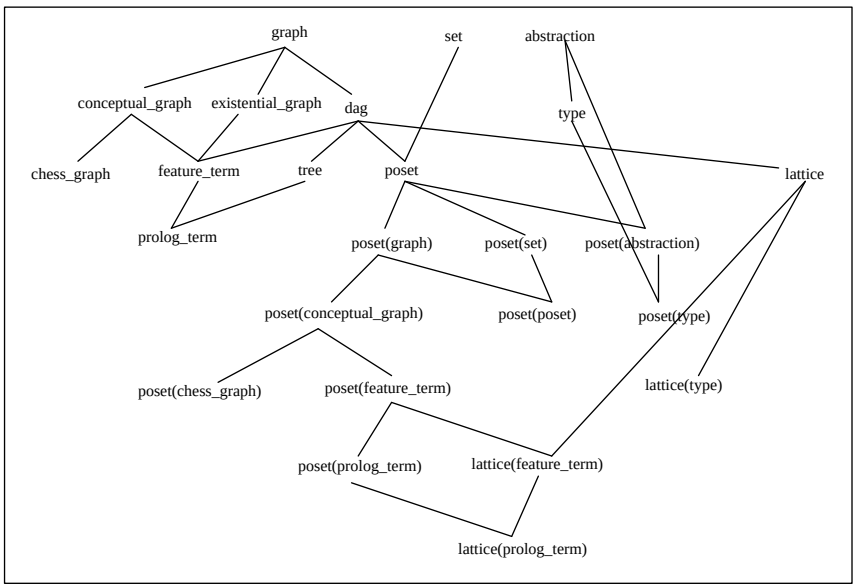

Fig. 1. A Hierarchy of ADTs in PEIRCE

3 Distribution and Control

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

4 Important Dates

The first official meeting of the PEIRCE project participants will be the First International Workshop on PEIRCE: A Conceptual Graphs Workbench held in association with the Seventh Annual Conceptual Graphs Workshop in Las Cruces, New Mexico, in July 1992. Subgroups working on each of the PEIRCE modules will give status reports [5].

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

Summary

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

The international PEIRCE TEAM so far includes

Alex Bejan	IBM Kingston, New York	United States
Keith Campbell	Stanford Univ	United States
Key-Sun Choi	Korea Adv Inst Sc Tech	Korea
Tae-Sang Choi	Univ Missouri-Kansas City	United States
Young Bae Choi	Univ Missouri-Kansas City	United States
Peter Creasy	Univ Queensland	Australia
Walling Cyre	Virginia Tech	United States
Harry Delugach	Univ Alabama-Huntsville	United States
Peter Eklund	Univ Adelaide	Australia
Gerard Ellis	Univ Queensland	Australia
John Esch	Paramax Systems	United States
Norman Foo	Univ Sydney	Australia
Brian Gaines	Univ Calgary	Canada
Steve Graham	Univ Missouri-Kansas City	United States
Rob Greenwood	Virginia Tech	United States
Young-Suk Han	Korea Adv Inst Sc Tech	Korea
Tim Hines	Univ Missouri-Kansas City	United States
Kees Hoede	Univ Twente	The Netherlands
Wen-Jung Hsin	Univ Missouri-Kansas City	United States
Myung-Cheol Kim	Korea Adv Inst Sc Tech	Korea
Pavel Kocura	Loughborough Univ Tech	United Kingdom
Fritz Lehmann	GRandAI Software	United States
Bob Levinson	Univ California at Santa Cruz	United States
Michel Liquiere	Reunion Univ	France
Meenakshi Manek	Virginia Tech	United States
Illias Mavreas	Univ Missouri-Kansas City	United States
Guy Mineau	Laval Univ	Canada
Robert Muehlbacher	A C Nielsen	Austria
Sung Hyon Myaeng	Syracuse Univ	United States
Antoine Ogonowski	GSI-ERLI	France
Jonathan Oh	Univ Missouri-Kansas City	United States
Maurice Pagnucco	Univ Sydney	Australia
Heather Pfeiffer	New Mexico State Univ	United States
Simon Polovina	Loughborough Univ Tech	United Kingdom
S Shankaranarayanan	Virginia Tech	United States
Kwang-Jun Seo	Korea Adv Inst Sc Tech	Korea
John Sowa	IBM Thornwood	United States
Aniruddha Thakar	Virginia Tech (US)	United States
Bill Tepfenhart	Bell Labs	United States
Bosco Tjan	Univ Minnesota	United States
Harmen van den Berg	Univ Twente	The Netherlands
Wei Wei	Univ Missouri-Kansas City	United States
Yin Min Wei	Univ Ohio	United States
Michel Wermelinger	Univ Nova de Lisboa	Portugal
Mark Willems	Univ Twente	The Netherlands
Gi-Chul Yang	Univ Missouri-Kansas City	United States
Wen Zhang	Virginia Tech	United States

References

1. Aït-Kaci, H., Boyer, R., Lincoln, P., Nasr, R.: Efficient Implementation of Lattice Operations. ACM Trans. Prog. Lang. Sys. **11**(1) (1989) 115–146.
2. Gould, J. and Levinson, R. A.: Experience-Based Adaptive Search to appear in Machine Learning IV: A Multi-Strategy Approach, R. Michalski and G. Tecuci editors, 1992.
3. Ellis, G.: Compiled hierarchical retrieval. In *Current Directions in Conceptual Structures Research* Nagle, T., Nagle, J., Gerholz, L., Eklund, P. (Eds.), Ellis Horwood (1992) 285–310
4. Ellis, G.: Compiling Conceptual Graphs. IEEE Trans. Know. Data Eng. (to appear)
5. Ellis, G., Levinson, R. A.: Proceedings of the First International Workshop on PEIRCE: A Conceptual Graphs Workbench. Held in association with the Seventh Annual Conceptual Graphs Workshop, Las Cruces, New Mexico, July 8-10, (1992)
6. Levinson, R. A.: Pattern Associativity and the Retrieval of Semantic Networks. J. Comp. Math. with Appl. **23** (part 2, pp. 573-600) (1992)
7. Levinson, R. A., Snyder, R.: Adaptive pattern oriented chess. Proc. AAAI-91. Morgan-Kaufman (1991) 601–605
8. Levinson, R. A., Ellis, G.: Multi-Level Hierarchical Retrieval. Know. Based Sys. J. (1992)
9. Nagle, T., Nagle, J., Gerholtz, L., and Eklund, P.: Conceptual Structures: Current Research and Practice. Ellis Horwood (1992)
10. Pagnucco, M.: CoGNO - A Graphical User Interface to Conceptual Graphs. Honours project, Sydney University, Nov (1990)
11. Roberts, D. D.: The Existential Graphs of Charles S. Peirce. Mouton, The Hague (1973)
12. Sowa, J. F.: Conceptual Structures: Information in Mind and Machine. Addison-Wesley, Reading, MA (1984)

This article was processed using the LaTeX macro package with LLNCS style