An Exportable CGIF Module from the CP Environment: A Pragmatic Approach

Heather D. Pfeiffer

Department of Computer Science
New Mexico State University
Las Cruces, New Mexico 88003-8001 USA
hdp@cs.nmsu.edu

Abstract We have upgraded the Conceptual Programming Environment (CP) from a single standalone application to a set of component modules to increase flexibility of the environment and to allow any one of the modules to be used by applications outside of the environment. This allows the CP modules, in particular CGIF, to be tested pragmaticly in real applications. The CGIF module is encapsulated as a component and has been given an API specification. This module implements the NMSU modified ICCS2001 version of the CGIF interchange format for Conceptual Graphs (CGs) that is part of the dpANS and is setup to link with applications written in C, C++, Java™and Visual Basic 6.0. Communication is possible to all these languages by designing the API of the module so that it can accept either standard C string or Unicode string within the function calls. Since modules are flexible units of code, the components of the CP Environment have been tested for use under most versions of Microsoft Windows and different flavors of Linux.

1 Introduction

John Sowa developed Conceptual Structures (CS) [35,36] as a form of semantic modelling using first-order logic. These models are based on the works of C.S. Peirce [24] and represent declarative knowledge. This knowledge is implemented using connected multi-labeled bipartite oriented graphs that Sowa refers to as Conceptual Graphs (CGs). Each graph consists of nodes (with types - concept, relation, actor, specialcontext, comment) and edges (arcs connecting the nodes). There exists a mapping from each of these graphs to formulae in first-order logic.

In order for CGs to be transferred between working tools, either all the tools must be implemented within the same environment as a single application or there must be an interchange format. When tools are developed through a single system, the same data structure (or model) can be shared among all the tools so that data can be stored and retrieved. However, when tools are not part of the same system, they do not necessarily share the same internal data structure (or design model). To support interoperability for the CG-based applications [29], an interchange format referred to as "Conceptual Graph Interchange Format (CGIF)" has been proposed [37].

This interchange format must be agreed upon by the whole working community. All though the CGIF document has been submitted as a standard, the final agreement for this format has not been reached within the community. Therefore, we present a pragmatic approach for enhancing and improving this format by offering a mechanism for interchange that can be placed in a testbed of applications. This is in response to the theoretical problems that are preventing the final agreement of this interchange format. We hope that as the community exchanges and uses CG s developed by different application tools, that the CGIF format will be enhanced, modified, and more clearly and precisely defined.

In this paper, we use the NMSU modified ICCS2001 version of CGIF[1]. At the "Workshop on Conceptual Graphs Tools (CGTools)" [28] at *ICCS2001*, at least part of the community agreed on this form of the interchange format. The major differences between this specification and the standard dpANS [40] are 1) CGStream has been added; 2) Qualifiers have zero or more quoted strings instead of arcs*; 3) Special Contexts must have (no longer optional) a colon after the SpecialContextLabel before the definition; 4) Comments are just comments (no special types); 5) Referents begin with a colon and the colon is removed from the Concept definition; 6) All Type Definitions must have a colon between the type and label (same is true of Relation Definitions); and 7) The knowledge base is called a KB as oppose to a Module (this matches the original proposed standard [37]).

There are now several tools available within the CS community for processing Conceptual Graphs (CGs) and Concept Graphs (FCAs) for use with different applications. However, it is obvious that not all these applications use the same data structure for representing CGs and FCAs. Some of the data structures represent CGs as graphs with nodes and edges as in a semantic network structure [21]. Some examples are the Conceptual Programming Environment (CP) [25,26]; CGWorld [15,14]; GoGITaNT [3]; CGKEE [22]; and editors, such as, CharGer [11,10,1] and ARCEdit [27]. Other applications use concept lattices to define conceptual relationships as defined through Formal Concept Analysis (FCA) [17]. Some examples of these are ToscanaJ [18,6,5]; Docco [2]; and WebKB [23]. Other applications do not actually display CGs, but use the Conceptual Structure syntactic data and semantics to process information and define new languages; for example, pCG [8,7] and Prolog+CG [20]. Each of these applications has its own internal representation for CS and, therefore, they all use different data structures. One would not want to use the same data structure for all these systems as they have been optimized for either the application that is being implemented and/or the new languages that are being designed.

2 CP Environment

The Conceptual Programming Environment (CP) has been completely redesigned since originally presented by NMSU in 1992 [25,26]. Recalling that CP is a knowledge representation development environment with a graphical visualization framework, this set of tools uses graph structures and operations over those structures to do knowledge reasoning.

[1] http://www.cs.nmsu.edu/~hdp/CGTools/cgstand/cgstandnmsu.html#Header_44 location

All knowledge within the system is stored and operated on as a graph. These graphs are implementations of Sowa's Conceptual Graphs [35], but also retain many of the features of graph theory [19]. Although there exists a mapping from CGs to formulae in first-order predicate calculus (FOPC), the operations used in the CP system take advantage of the graphical representation; therefore, the data structures and operations over the graphs use graph theory [19] instead of FOPC.

The original system was a single application written in Lisp and ran only on a Symbolics machine. The data structures were CGs defined using link lists of structure elements where the structures held the node information and the links were the edges of the graphs. All graphs had to be entered directly into the environment's editor, and each graph was stored into the environment's knowledge base. The CP inference engine would then operate over these data structures; sometimes creating new graphs or partial models of conceptual graphs and storing them into the environment's knowledge base. In the old environment, there was no way to import or export any of the graphs or models.

2.1 Motivation for Change

Over the last decade, we at NMSU have discovered that creating flexible, modularized environment is a much better design than a single application. Harry Delugach's invited talk at ICCS2003 [12] outlined a framework for building *active knowledge systems*. We agree with this view point and have designed the CP Environment to be the "heaven" displayed in his framework.

John Sowa, in a paper published in 2002 as part of a "Special Issue on Artificial Intelligence" of the *IBM Systems Journal* [38], proposed a modular framework as an architecture for intelligent systems because of the flexibility in communication and interoperability it provides. This flexible modular framework (FMF) allows different applications in different memory space to communicate using a blackboard architecture of message passing between applications. FMF would be very useful in implementing the reference framework discussed in Aldo de Moor's RENISYS specification methodology [13] because FMF handles interprocess communication across computers as well as processes, and it would also be useful in developing the intelligent agent operations from Delugach's framework [12]. However, the modularization of CP is at a module component level, rather than the FMF process communication level, so that the module can be directly "tied-in" to another application. The modular design at the component level also allows modules to be interchanged as units, as in modular furniture, to get the most flexibility from the environment.

The modularization of the CP Environment allows parts of the environment, the actual modules, to be both interfaced and interacted with by outside systems or applications. It also has a specific module, CGIF, that creates a mechanism to import and export CGs created from execution of the environment's inference engine modules and storage in the environment's knowledge base. This mechanism can be "plugged-in" to other applications by using the CGIF module's API specification to call the module's implementation code level [34]. Therefore, if an application currently available within the community does not have the ability to read and write CGIF format, this module could be "plugged-in" to the existing application to give it that functionality.

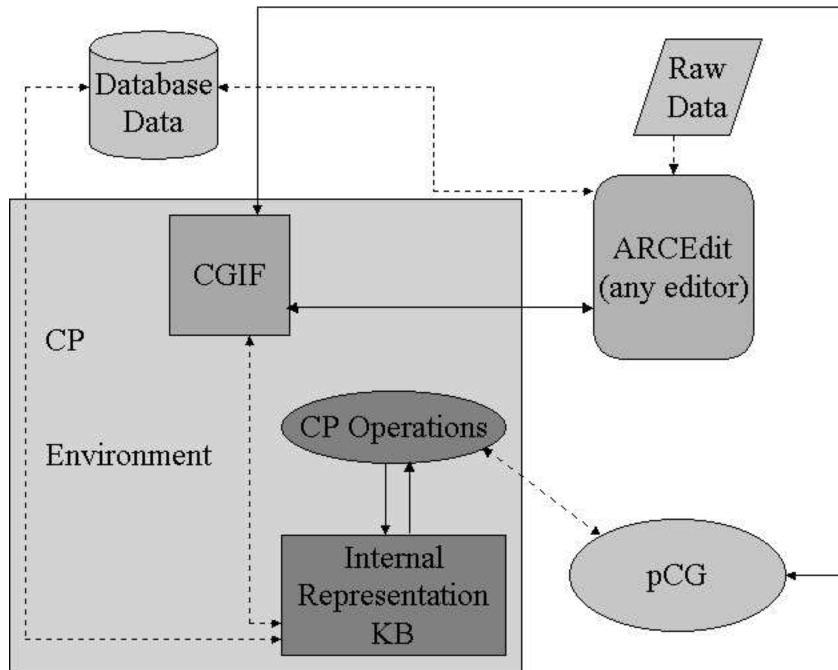

Figure 1. Current CP Environment

2.2 Layout of Environment

Figure 1 depicts the new directionality of the CP Environment. The very light gray background area indicates what is actually part of the environment. The light gray oval depicts applications, i.e. the pCG reasoning and language system. The medium gray rounded-corner-square represents editors that are available for CGs, i.e. ARCEdit; these editors should be able to import/export CGIF formated files. The light gray trapezoid and drum shapes indicate data that is not necessarily graphical in nature, but may be part of a domain of information that a user wishes to process (note: the data in the database need not necessarily be textual and may be graphical or any visual form). The very dark gray shapes are modules that are part of the CP Environment and use the environment's internal data structures. All solid arrowed lines in the figure indicate data or processing that is currently available; dashed arrowed lines indicate where an interface, connection, interaction, and/or translation should be available between these elements, but is not currently present.

This newly designed environment has several areas of research that should be investigated. Some of these would be: data structures for syntactic data processing; database record processing; raw data processing; storage and retrieval of internal KB data processing; and reasoning and query processing. These areas should handle not only declarative and assumption knowledge, but also procedural knowledge containing time, space and other constraint information.

However, in this paper, we would like to look more closely at data structures for syntactic data processing. In particular, efficient and flexible ways of dealing with moving syntactic data information around (such as CGIF formatted graphs), both inside and outside of the CP environment.

3 Implementation Language Evaluation

Each of the applications and semantic languages defined in Section 1 may or may not share the same implementation language. So an added complication, besides different internal data structures, is that an application may not be able to communicate at a function call level with another application because they are not written in the same implementation language.

When we began to redesign the CP Environment, the question arose what implementation language should be used with the new modules. We examined first the CS editors that were currently available. These editors, for editing CGs and FCAs, have different implementation languages. CharGer is based on the API/Implementation code of Notio [34,33] which is written in Java™. ARCEdit is a plug-in to PowerPoint and is written in Visual Basic 6.0. ToscanaJ has an editor as part of its suite of tools written in Java™. While Docco is actually based on a Conceptual Email Manager [9], written in C++/QT, the commercial version of the manager [16] is a plug-in for Microsoft Outlook.

Since Notio, written in Java™, is already defined as an API/Implementation code level and is available with extendable class definitions, we considered using the Notio interface for the CGIF module. However, there are some drawbacks in communications to Java™(see Section 3.2), and Notio is in hiatus and is not currently being enhanced or developed [32].

All of the applications discussed in Section 1 employ different implementation languages, such as Prolog, XML, Schema, RDF, etc., which make it difficult to pass even simple syntactic representation data by linking languages in modules. Files, streams, pipes, blackboards, etc. can be used to pass data information without passing the actual data structures, but these mechanisms can be slow if there are a large number of graphs or the graphs are extremely complex. Every time one application process needs to talk to another, these mechanisms require multiple file descriptors to be opened. If the applications or systems execute on different machines, the FMF architecture is a flexible way of passing the syntactic data representation; but, if the applications and systems are able to be executed on the same machine configuration, a good API/Implementation design would be more advisable because the module can be linked directly into the existing application. Communication by files and other stream devices may require a locking mechanism to be setup, so that one application can know when it is safe to read the input graphs from another application. The locking of records can cause a problem when two applications communicate by way of shared databases or message passing systems, such as MPI. If on the other hand, an application/system can call another application/system directly (or can link to it), processing can go more quickly.

However, connecting systems when the implementation languages are not the same is more difficult, because a straight forward ”call” to the other system’s functions is not always possible. Each language implementation has its own calling specifications. In order to know which language would be best for implementation of the new environment’s modular components, so that they could possibly be used directly by other applications, we performed an evaluation of how the C++ language interacts, interfaces and communicates with other languages.

3.1 C++ to C

Interfacing implementation code between languages that are somewhat similar, .i.e. C++ and C, is not as difficult as other communications between languages. However, this connection may not be bidirectional. The calling sequence for the C language is simpler than for the C++ language, because C++ does name mangling with the name of the function, the types of the arguments and the return type of the function. C does not do the same name mangling and uses only a modified form of the actual name of the function.

Therefore when designing an API in C++, the interface routines should be exported as ”C” functions as opposed to being methods for a class in C++; this will prevent the routines from being name mangled by the C++ compiler. Wrapping C++ with standard C routines allows the internal implementation of the module to remain C++ and use the classes and methods functionality from C++, while at the same time using the simpler formulation of the name of the calling routine provided by C.

3.2 C++ to Java™

Connecting C++ to Java™is also possible, but is more difficult than communicating with C. This connection is also not bidirectional, but for different reasons. Java™is a simpler language than C++ [30], but it is an interpreted language. This means that Java™can be byte-compiled, creating a smaller file to be moved across the web, but it is not compiled to machine code. This allows Java™to be platform independent; C++ is a compiled language and is both platform dependent and operating system (OS) dependent. However, because Java™is interpreted instead of compiled, it can not be linked and called directly by a system (application) that is not written in Java™. Java™must start the process, and then can call compiled code in some of the other compiled languages (i.e. C/C++). Therefore, Java™can call interface functions written in C or C++, but C++ can not call Java™directly.

3.3 C++ to Visual Basic 6.0

The connection or interface from C++ to Visual Basic 6.0 is the most difficult connection among the four languages discussed in this paper. One reason is that Visual Basic 6.0 is a two part language; an event driven module part and a class module part.

The class module part is very similar to C++ and holds the characteristics that are available in object-oriented languages. Class modules can also be compiled just like C++ to native code for the machine. However, the event driven part executes Basic code in response to an event. These event driven procedures (routines) are triggered by a form or control which is hooked into the visual part of the language. The triggering of a routine by an event is similar to the interpretation of a function call in languages like Java^{TM}. Because of the event driven part of the language, Visual Basic 6.0 can call C++ or C API/Interface routines, but C++/C cannot trigger an event within the Visual Basic code, so the event procedures (routines) are not executed outside of Visual Basic code.

A second reason Visual Basic 6.0 is difficult to connect, is that it has different encoding of some of its data types than C, C++, or Java^{TM}[31]. Character data is stored in more bits by Visual Basic than by C. Therefore, to pass a character string as a parameter from Visual Basic to C or visa versa, the character string must be converted to Unicode first, that is passed as a parameter, and then decoded from Unicode at the other end. This makes passing character data much more cumbersome. Also, Visual Basic defines different boolean values than C; the ”false” value is 0 in both languages, but the ”true” value for Visual Basic is -1 (negative) where in C it is 1 (positive). Therefore, in passing boolean values, the user must be careful when working with conditionals.

3.4 Visual Basic .Net

If we were willing to have the CP Environment component modules available only for execution under the Microsoft Windows OS, Visual Basic .Net does not have any of the connection or interface problems discussed above. However, this would not allow the components implementation to be moved off the Microsoft Windows OS. If the CP modules are not implemented in Visual Basic .Net, than they can be made more widely available under Linux operating systems and eventually under other operating systems such as Unix.

4 API for CGIF Module

The CGIF module within the CP Environment is shown as the medium gray box labelled ”CGIF” in Figure 1. The box is colored medium gray to depict that the functionality of this module was originally developed as part of the ARCEdit editor[27]. We decided that because the ARCEdit editor provides a way for graphs to be imported and exported to the CP Environment with a visual interface and able to import and export CGs in the CGIF format, we would remove the CGIF functions from ARCEdit and make a callable module that would become part of the environment. Weighing all learned during the evaluation of implementation languages discussed in Section 3, and wanting the new module to be portable to different operating systems, we selected the language C++ for implementing the new modules in the CP Environment.

By using C++ for the implementation of the CGIF component and defining an API to make it truly modular, we produced a DLL in the Microsoft Windows environment and later a C++ library in the Linux environment for the implementation of the module.

Definition 1. *CG*

A conceptual graph is a list of zero or more concepts, conceptual relations, actors, special contexts, or comments.

CG ::= (Concept | Relation | Actor | SpecialContext | Comment)*

The alternatives may occur in any order provided that any bound coreference label must occur later in the CGIF stream and must be within the scope of the defining label that has an identical identifier. The definition permits an empty CG, which contains nothing. An empty CG, which says nothing, is always true.

The API to the CGIF module was designed to allow: the creation of CGs as defined by the standard (see Definition 1 - from the CGIF standard document), the storage and retrieval of each of the CG elements (as in Definition 1), and the reading and writing of CGs (as in Definition 1) into a file in CGIF format for sharing among the user community. However, while the module's API was being designed, considerations were made to allow implementation languages beyond C and C++ to interface and interact with the module's implementation routines.

The API routines are defined to be standard C-named functions as wrappers around the C++ internal code. This is important to avoid any name mangling at compilation time and to allow other languages such as Java^{TM}to call the functions. The routines are also not attached to an event so they can be called by implementation languages other than Visual Basic 6.0. Using C++ instead of Visual Basic .Net allows the implementation of the module to be ported easily to operating systems other than Microsoft Windows. This was an added reason why the CGIF routines were not originally left in Visual Basic 6.0 and just made into a project by themselves (outside of ARCEdit). If this had been done, the CGIF module could have been made into a DDL for use on other Microsoft Windows machines, but could not be called by applications written in languages other than Visual Basic and could not have been ported to Linux.

As one can see in the Appendix, there are duplicates of some of the routines; this provides an interface for implementation languages, such as, Visual Basic 6.0 to pass character strings as arguments.

Example 1. Parse Concept (node) Routines

```
CGIF_API BOOLEAN STDCALL CGIF_ParseConcept( char *, BOOLEAN );
CGIF_API BOOLEAN STDCALL CGIF_ParseConceptW( BSTR, BOOLEAN);
```

An example of one of these paired routines can be seen in Example 1. In this example, the routines are almost exactly the same except that the second routine carries a 'W' after the routine name to indicate that the character string parameters or return arguments are in Unicode format, not standard C character strings.

Therefore when an implementation language needs to handle character strings as Unicode strings, they would call the routine from the API with the appended 'W'; C and C++ implementation routines can just call CGIF_ParseConcept directly.

Definition 2. *Concept*

A concept begins with a left bracket "[" and an optional monadic type followed by optional coreference links and an optional referent in either order. It ends with an optional comment and a required "]".

Concept ::= "[" Type(1)? {CorefLinks?, Referent?} Comment? "]"

If the type is omitted, the default type is Entity. This rule permits the coreference labels to come before or after the referent. If the referent is a CG that contains bound labels that match a defining label on the current concept, the defining label must precede the referent.

The code within the two API routines is identical and both functions correctly implement the definition of a Concept as defined from the standard in Definition 2.

An example of using the paired routines seen in Example 1 would be if one parsed the phrase given in Example 2 using tools written in different languages.

Example 2. CG Concept Phrase
[CAT:'Felix']

If the ARCEdit editor was to parse the concept in Example 2, the second routine of the pair (CGIF_ParseConceptW) would need to be called, so that the string parameter would be sent in Unicode form. However, if the reasoning program pCG was to parse the Example 2 phrase, it would call the first routine (CGIF_ParseConcept) and use the C string type for the parameter. Both routines would create a concept node with the type: CAT and referent: 'Felix'.

5 Conclusion

Defining the CP environment to be both flexible and modular, makes it fit into the framework for beginning to build an *active knowledge system*. The modular design is at a component/unit level - rather than at the process/machine level - as seen in the FMF architecture. However, a module from the CP Environment allows other applications or systems to use a single module or a small group of modules from CP without requiring all of it.

The component module includes an API, so that other applications can directly link to it and need not incur the overhead of files, etc. The CP Environment design creates a way for a single module, i.e. CGIF, to be made available to other applications and systems written in different implementation languages, so that they can quickly incorporate it into their systems for handling the CGIF format. The CGIF module is available for download from the CP Environment's web page [2]. As other modules become available they can be downloaded, with source implementation code and complete documentation.

[2] http://port.semanticweb.org/CP/index.html

As the community begins to use the CGIF format to transfer Conceptual Graphs from tool to tool, benchmarks of graphs can begin to be built and used in testbeds, such as PORT [4]. This will allow the user community to evaluate the features of each tool and how it might contribute to their research. Evaluation of the CGIF format would give different tool developers the opportunity to identify shortcomings in the current specifications and suggest possible theoretical design changes. The community's tool developers can then coorporate together to define more precisely the interchange format. By use of this pragmatic approach, we hope the CGIF format can be improved, and agreement might be reached within the community.

Within the CP Environment application, new modules can be added to translate CGIF format to other formats, such as the CLCE being developed currently by John Sowa [39]. This environment can be easily modified to use the FMF architecture by adding new component modules to communicate with the blackboard.

6 Future Research

Current research efforts for the CP Environment are to make the "CP Operations" module available to be linked by applications and systems outside the environment. This module has been encapsulated and an API is being defined. We will test the API/Implementation module by setting up communication between the CP Operations module and the pCG reasoning system. Once testing is complete, this module will be placed on the CP Environment web site [2] and made available to outside applications and systems. It should be noted, that the pCG reasoning system is implemented in Java™, but will be able to interface and interact directly with the CP Operations module, written in C++. The CP Operations module has the basic operations: maximal-join, project, generalize, and specialize. In the not so distant future, we hope that operations will be added to this module for performing operations over time, space and constraint processing.

Acknowledgements

Thanks to the many reviewers that took the time to help me improve and revise this paper. In particular, I would like to thank Mary Keeler for all her great comments and observations.

References

1. *CharGer - A Conceptual Graph Editor*. University of Alabama in Huntsville, Alabama, USA. http://www.cs.uah.edu/ delugach/CharGer.
2. *Docco*. University of Queensland, Australia. http://tockit.sourceforge.net/docco/index.html.
3. *GoCITaNT*. LIRMM - Montpellier, France. http://cogitant.sourceforge.net/index.html.
4. PORT: The Peirce On-line Resource Testbed Project. http://port.semanticweb.org/.
5. Peter Becker. *ToscanaJ*. Technical University of Darmstadt, Germany. http://toscanaj.sourceforge.net/.
6. Peter Becker and Joachim Hereth Correia. The ToscanaJ suite for implementing Conceptual Information Systems. In G. Stumme, editor, *Formal Concept Analysis – State of the Art*, Berlin – Heidelberg – New York, 2004. Springer. To appear.

7. D. Benn and D. Corbett. An application of the process mechanism to a room allocation problem using the pCG language. In H.S. Delugach and G. Stumme, editors, *Conceptual Structures: Broadening the Base*, Springer-Verlag Lecture Notes in Computer Science 2120, pages 360–374, 2001.
8. D.J. Benn and D. Corbett. pCG: An implementation of the process mechanism and an extensible CG programming language. In *CGTools Workshop Proceedings in connection with ICCS 2001*, Stanford, CA, 2001. [Online Access: July 2001] URL:http://www.cs.nmsu.edu/ hdp/CGTOOLS/proceedings/index.html.
9. R.J. Cole, Peter Eklund, and Gerd Stumme. Document retrieval for email search and discovery using Formal Concept Analysis. *Applied Artificial Intelligence*, 17(3), 2003.
10. H. Delugach. CharGer: Some lessons learned and new directions. In G. Stumme, editor, *Working with Conceptual Structures - Contributions to ICCS 2000*, pages 306–309, 2000. Shaker-Verlag.
11. H. Delugach. CharGer: A graphical Conceptual Graph editor. In *CGTools Workshop Proceedings in connection with ICCS 2001*, Stanford, CA, 2001. [Online Access: July 2001] URL:http://www.cs.nmsu.edu/ hdp/CGTOOLS/proceedings/index.html.
12. H.S. Delugach. Towards building active knowledge systems with conceptual graphs. In A. de Moor, Wilfried Lex, and Bernhard Ganter, editors, *Conceptual Structures for Knowledge Creation and Communications*, pages 296–308, Heidelberg, 2003. Springer-Verlag. Lecture Notes in Artificial Intelligence, LNAI 2745.
13. A. deMoor. Applying conceptual graph theory to the user-driven specification of network information systems. In D. Lukose, H.S. Delugach, M. Keeler, L. Searle, and J.F. Sowa, editors, *Conceptual Structures: Fulfilling Peirce's Dream*, Springer-Verlag Lecture Notes in Artificial Intelligence 1257, pages 536–550. ICCS, Springer, August 1997.
14. P. Dobrev, A. Strupchaska, and K. Toutanova. CGWorld-2001: New features and new directions. In *CGTools Workshop Proceedings in connection with ICCS 2001*, Stanford, CA, 2001. [Online Access: July 2001] URL:http://www.cs.nmsu.edu/ hdp/CGTOOLS/proceedings/index.html.
15. P. Dobrev and K. Toutanova. CGWorld - a web based workbench for conceptual graphs management and applications. In G. Stumme, editor, *Working with Conceptual Structures - Contributions to ICCS 2000*, Shaker-Verlag, pages 243–256, 2000.
16. Email Analysis Pty Ltd, Australia. *Mail-Sleuth*. http://www.mail-sleuth.com/.
17. B. Ganter and R.Wille. *Formal Concept Analysis: Mathematical Foundations*. Springer-Verlag, Berlin Heildelberg New York, 1999.
18. B. Groh, S. Strahringer, and R. Wille. TOSCANA-systems based on Thesauri. In Marie-Laure Mugnier and Michel Chein, editors, *Conceptual Structures: Theory, Tools, and Applications*, Springer-Verlag Lecture Notes in Artificial Intelligence 1453, pages 127–141, Heidelberg, August 1998. ICCS, Springer-Verlag.
19. F. Harary. *Graph Theory*. Addison-Wesley, Reading, MA, 1969.
20. A. Kabbaj and M. Janta-Polcynzki. From Prolog++ to Prolog+CG: A CG object-oriented logic programming language. In B. Ganter and G. Mineau, editors, *Conceptual Structures: Logical, Linguistic, and Computional Issues*, pages 540–554, Berlin, 2000. Lecture Notes in Artificial Intelligence, vol. 1867, Springer-Verlag.
21. F. Lehmann, editor. *Semantics Networks*. Pergamon Press, Oxford, ENGLAND, 1992.
22. D. Lukose. CGKEE: Conceptual graph knowledge engineering environment. In Conceptual Structures: Fulfilling Peirce's Dream, editor, *Conceptual Structures: Fulfilling Peirce's Dream*, Springer-Verlag Lecture Notes in Artificial Intelligence 1257, pages 589–593. ICCS, Springer, August 1997.
23. P. Martin. The webKB set of tools: A common scheme for shared www, annotations, shared knowledge bases and information retrieval. In D. Lukose, H.S. Delugach, M. Keeler,

L. Searle, and J.F. Sowa, editors, *Conceptual Structures: Fulfilling Peirce's Dream*, Springer-Verlag Lecture Notes in Artificial Intelligence 1257, pages 585–588. ICCS, Springer, August 1997.
24. C.S. Peirce. Manuscripts on existential graphs. *Peirce*, 4:320–410, 1960.
25. H.D. Pfeiffer and R.T. Hartley. The Conceptual Programming Environment, CP. In T.E. Nagle, J.A. Nagle, L.L. Gerholz, and P.W. Ekland, editors, *Conceptual Structures: Current Research and Practice*, Ellis Horwood Workshops. Ellis Horwood, 1992.
26. H.D. Pfeiffer and R.T. Hartley. Temporal, spatial, and constraint handling in the conceptual programming environment, cp. *Journal for Experimental and Theoretical AI*, 4(2):167–182, 1992.
27. H.D. Pfeiffer and R.T. Hartley. ARCEdit - CG editor. In *CGTools Workshop Proceedings in connection with ICCS 2001*, Stanford, CA, 2001. [Online Access: July 2001] URL:http://www.cs.nmsu.edu/ hdp/CGTOOLS/proceedings/index.html.
28. H.D. Pfeiffer and R.T. Hartley, editors. *CGTools Workshop Proceedings in connection with ICCS 2001*, Stanford, CA, 2001. [Online Access: July 2001] URL:http://www.cs.nmsu.edu/ hdp/CGTOOLS/proceedings/index.html.
29. A. Puder. Mapping of CGIF to operational interfaces. In Marie-Laure Mugnier and Michel Chein, editors, *Conceptual Structures: Theory, Tools, and Applications*, Springer-Verlag Lecture Notes in Artificial Intelligence 1453, pages 119–126, Heidelberg, August 1998. ICCS, Springer-Verlag.
30. Kirk Radeck. C# and Java: Comparing programming languages. October 2003. http://www.windowsfordevices.com/articles/AT2128742838.html.
31. Steven Roman. *Win32 API Programming with Visual Basic*. O'Reilly, first edition, 1999.
32. F. Southey. *NOTIO*. http://notio.lucubratio.org/index.html.
33. F. Southey. Notio and Ossa. In *CGTools Workshop Proceedings in connection with ICCS 2001*, Stanford, CA, 2001. [Online Access: July 2001] URL:http://www.cs.nmsu.edu/ hdp/CGTOOLS/proceedings/index.html.
34. F. Southey and J.G. Linders. NOTIO - a Java API for developing CG tools. In W. Tepfenhart and W. Cyre, editors, *Conceptual Structures: Standards and Practices*, pages 262–271, Berlin, 1999. Springer-Verlag. Lecture Notes in Artificial Intelligence, LNAI 1640.
35. J.F. Sowa. *Conceptual Structures: Information Processing in Mind and Machine*. Addison-Wesley, Reading, MA, 1984.
36. J.F. Sowa. Conceptual graphs as a universal knowledge representation. In F. Lehmann, editor, *Semantics Networks*, Oxford, ENGLAND, 1992.
37. J.F. Sowa. Conceptual graphs: Draft proposed american national standard. In Conceptual Structures: Standards and Practices, editors, *Conceptual Structures: Standards and Practices*, pages 1–65, Berlin, 1999. Springer-Verlag. Lecture Notes in Artificial Intelligence, LNAI 1640.
38. J.F. Sowa. Architectures for intelligent systems. In *Special Issue on Artificial Intelligence*, volume 41 of *IBM Systems Journal*, pages 331–349. 2002.
39. J.F. Sowa. Common Logic Controlled English. http://www.jfsowa.com/clce/specs.htm, February 2004.
40. J.F. Sowa and et al. *Conceptual Graph Standard, American National Standard NCITS*, T2/ISO/JTC1/SC32 WG2 M 00 edition. [Access Online:April 2001], URL: http://www.bestweb.net/ sowa/cg/cgstand.htm.

Appendix: Example Documentation

Main Page | Modules | File List | Globals

CGIF Parse Functions

CGIF Parse Functions

These functions can parse CGIF format for each node type in CG.

CGIF_API BOOLEAN STDCALL	**CGIF_ParseConcept** (char *conceptstr, BOOLEAN opt)
	Parse the concept string that is in the form "[" type(1)? {coreflink? referent?} comment? "]".
CGIF_API BOOLEAN STDCALL	**CGIF_ParseConceptW** (BSTR bsconcept, BOOLEAN opt)
	Parse the concept string that is in the form "[" type(1)? {coreflink? referent?} comment? "]".
CGIF_API BOOLEAN STDCALL	**CGIF_ParseRelation** (char *relstr, BOOLEAN opt)
	Parse the relation string that is in the form "(" type(N) arc* comment? ")".
CGIF_API BOOLEAN STDCALL	**CGIF_ParseRelationW** (BSTR bsrelation, BOOLEAN opt)
	Parse the relation string that is in the form "(" type(N) arc* comment? ")".
CGIF_API BOOLEAN STDCALL	**CGIF_ParseActor** (char *actstr, BOOLEAN opt)
	Parse the actor string that is in the form "<" type(N) arc* "|" arc* comment? ">".
CGIF_API BOOLEAN STDCALL	**CGIF_ParseActorW** (BSTR bsactor, BOOLEAN opt)
	Parse the actor string that is in the form "<" type(N) arc* "|" arc* comment? ">".
CGIF_API BOOLEAN STDCALL	**CGIF_ParseSpecCxt** (char *scstr, BOOLEAN opt)
	Will parse the special concept strings (Note: not implemented yet).
CGIF_API BOOLEAN STDCALL	**CGIF_ParseSpecCxtW** (BSTR bsspecxt, BOOLEAN opt)
	Will parse the special concept strings (Note: not implemented yet).

Detailed Description

These functions can parse CGIF format for each node type in the Conceptual Graphs:
Concept, Relation, Actor, Special Context.

Function Documentation

CGIF_API BOOLEAN STDCALL CGIF_ParseConcept (	char *	*conceptstr*,
	BOOLEAN	*opt*
)		

Parse the concept string that is in the form "[" type(1)? {coreflink? referent?} comment? "]".

Parameters:

conceptstr C string representation of the concept string to be parsed

opt optional indicator if parsing is successful - TRUE if not interested; FALSE if need correct indicator

Returns:

BOOLEAN indicates if the string was successfully parsed

See also:

CGIF_ParseRelation

CGIF_ParseActor

CGIF_ParseSpecCxt

CGIF_API BOOLEAN STDCALL CGIF_ParseConceptW (	BSTR	*bsconcept*,
	BOOLEAN	*opt*
)		

Parse the concept string that is in the form "[" type(1)? {coreflink? referent?} comment? "]".

Parameters:

bsconcept Unicode string representation of the concept string to be parsed

opt optional indicator if parsing is successful - TRUE if not interested; FALSE if need correct indicator

Returns:

BOOLEAN indicates if the string was successfully parsed

See also:

CGIF_ParseConcept

Note: File Extended on Web.