Notio - A Java API for developing CG tools

Finnegan Southey[1] and James G. Linders[2]

[1]University of Waterloo, Dept. of Systems Design Engineering, Waterloo, Ontario, Canada, N2L 3G1
finnegan@pami.uwaterloo.ca
[2]University of Guelph, Department of Computing and Information Science, Guelph, Ontario, Canada, N1H 2G1
jgl@snowhite.cis.uoguelph.ca

Abstract. Notio [1] is a Java API for constructing conceptual graph tools and systems. Rather than attempting to provide a comprehensive toolset, Notio attempts to address the widely varying needs of the CG community by providing a platform for the development of tools and applications. It is, first and foremost, an API specification for which different underlying implementations may be constructed. A pure Java reference implementation is currently available for development and evaluation of the API, and to guide future implementations. An overview of the motivation, design, and features of Notio is provided.

Introduction

Since their introduction by John Sowa in 1984 [2] researchers have sought to create tools and applications for working with and harnessing conceptual graphs (CG). Many systems for representing, storing, retrieving, acquiring, and editing CGs have been created over the years, some highly focussed on particular knowledge domains or applications, and others of more general utility. We present Notio, a new application programming interface (API) which presents a set of primitives for CG construction and manipulation, providing a platform for the development of CG tools and applications.

Motivation

Most of the general-purpose software developed for CGs has been in the form of Atools@, software which provides a user interface or specialized language for manipulating graphs. Some of these interfaces have been graphical while others have employed special languages, command shells, or simple natural language interfaces to access the underlying graph implementation (e.g. Deakin Toolset [3] and CGKEE [4]). The goal has often been to provide a general-purpose conceptual graph

environment for knowledge engineers and researchers. Such systems are frequently described as *workbenches* (e.g. Peirce [5], CoGITo [6]).

The chief problem encountered with workbenches is a lack of consensus on the form they should take. No unifying, formal model for CG management currently exists. Thus, many different systems have been developed, each providing a set of tools for various tasks. These systems have been developed in many different languages and for many different platforms. The result is a considerable duplication of effort since most systems do not, or can not, use components from existing systems. In some cases this is inevitable, since researchers may wish to try new approaches and interfaces, but it is rare that the entire system is novel. At the same time, there are many differences in internal representations and algorithms for manipulating graphs. It is, therefore, no wonder that no one workbench satisfies everyone.

We decided to address the issue of generality, alternative implementations, and avoiding duplication of effort by adopting the approach used for the development of many major computing standards. We have elected to define an API that describes a number of operations for the construction and manipulation of CGs without dictating the nature of the underlying implementation, or the applications and interfaces used to invoke the operations. While other CG programming libraries exist (e.g. CoGITo), we believe Notio is the only CG API offered first and foremost as a specification (although some documents on Peirce discuss a similar use of Abstract Data Types).

Design Goals for Notio

The Notio API was constructed with the following design goals in mind:

- Emphasis on ease-of-use for application authors
- Intuitive representation of commonly accepted CG abstractions
- API independent from underlying implementations internal representation
- Portability
- Extensibility by providing formal extension facilities
- Generality by minimizing specification of the structure of an application or knowledge base
- Flexibility by allowing varying levels of compliance and making various constraints optional
- Robust by providing an extensive set of exceptions for error handling

Notio: A Java Package

Notio[1] is a class library written in Java[2] using version 1.1 of the Java platform. Such

[1] ANotio@ is the Latin word for Aconcept@ (pronounced “no-tee-oh”).

libraries are known as *packages* in Java parlance. Java was selected as the development language for a variety of reasons:

- Platform independence
- Object-oriented language
- Robust environment (no pointers, automatic garbage-collection)
- Several free and commercial development environments available
- Formal source-based documentation facilities (javadocs)

An object-oriented programming language was selected after a couple of prototypes using declarative languages (Prolog and CLIPS). While declarative languages have many attractive features and have been used to implement CG systems in the past (eg. CGPro [7]), it was decided that the object-oriented approach would find a broad acceptance both inside and outside of the knowledge representation community.

The Architecture of a Notio-based System

Notio is first and foremost an API specification, a document that specified a collection of Java class definitions and the behaviour expected of the methods in those classes. From this document a developer should be able to create an entire set of classes that implement the specified behaviour. Such an implementation is called an *implementation layer*. The internals of an implementation layer are entirely at the discretion of the developer. They can be a pure Java implementation of the API:s functionality, links through the Java Native Interface (JNI) to an existing CG-system written in another language, a client for a knowledge base system, or anything else the developer desires.

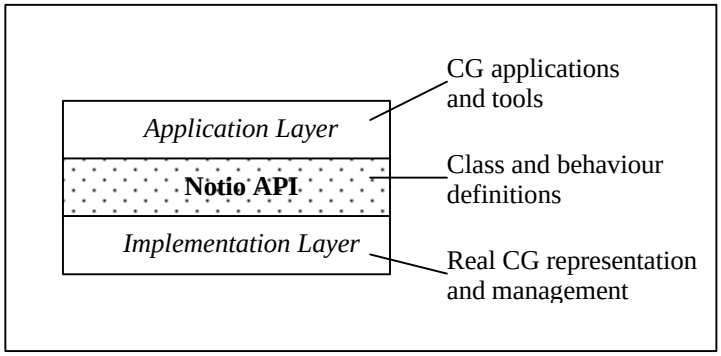

Figure 1: The structure of a Notio-based system.

Applications written to use the Notio API form what is known as an *application layer*. An application layer instantiates API classes and calls methods in them to accomplish

[2] Java **J** and JavaBeans **J** are trademarks of Sun Microsystems, Inc.

its goals, the implementation layer responds by changing whatever internal structures it holds and presenting them when requested in the form of API classes. Thus, the API classes acts as representatives for graph structures in the implementation layer and need not comprise the actual representation. Only requested structures need be instantiated by the implementation layer. A diagrammatic view of the architecture of a Notio-based system is shown in **Figure 1**.

Our Approach to the Notio API

Having stated that an obstacle to the wide acceptance of any particular workbench or tool is the difficulty of satisfying enough of the CG community, it is fair to ask why we believe that a general-purpose API can fulfill a similar role. We do not claim that this API will satisfy all needs, nor that it will suit the development of any one application, but we do believe that this is correct level to generalize and standardize.

CG Applications and tools are often very specific to a knowledge domain or real-world application and thus are virtually impossible to generalize. Similarly, efficient CG representations and operations still form an area of active research in which a wide variety of approaches are taken and different aspects of performance emphasized. However, an interface between these two layers, based on the abstractions which the CG paradigm suggests, allows for either to be replaced or reused as necessary. A substantial level of general agreement on this set of abstractions does exist in the CG community.

We base this opinion on the recent efforts towards the establishment of the ANSI standard (dpANS) for CGs [8], in which the CG community has worked together to reach a consensus. The standard seems to embody a set of properties and operations that are widely considered as integral to CGs. The Notio APIs development began as the first publically available versions of the draft standard were circulating and it was decided to design the API under the dictates of the emerging standard. Notio is, therefore, an attempt to realize the abstractions presented in the dpANS. However, it is understood that this standard could never embrace all of the various extensions and subtleties related to CGs, and so the API was also designed to allow extensions and varying levels of compliance.

While several CG tools have provided essentially the same set of abstractions, Notio provides them as an API specification. We believe an API to be an excellent generic method for presenting the abstractions and operations since it relies on a widely established language and programming paradigm, rather than custom languages or interfaces. This same principle guides the design of many development environments such as APIs for networking, graphical interfaces, and databases.

Design of the Notio API

The Basic Structures

Since the classes in the API are intended to provide an intuitive representation for working with CGs we elected to create a fairly direct correspondence between the classes and the abstractions defined in the dpANS. Thus, we have a Graph class which is composed of instances of the Concept and Relation classes. The Concept class is in turn composed of instances of the ConceptType and Referent classes. Referents are composed of Quantifiers and Designators. Relation instances have a RelationType and a list of arguments which serve as the arcs in the graph.

Notio directly supports compound graphs. Concepts may have a Referent with a DescriptorDesignator, which points to a nested graph. The CoreferenceSet class works with the Concept class to provide full support for coreferences, including the ability to enforce the scoping rules specified in the dpANS. The API allows for the traversal of compound graphs in either direction (top-down or bottom-up).

Types are handled via a ConceptTypeHierarchy instance or RelationTypeHierarchy instance as appropriate. They allow for labelled and unlabelled types, with or without signatures and type definitions. RelationTypes may optionally supply a valence (number of arguments) in the absence of a type definition. Changes to types are immediately reflected throughout existing graphs.

Knowledge Bases

Many tools require the use of a Aknowledge base@ construct that stores graphs and types. In developing the Notio API we wished to avoid dictating the overall structure of applications as much as possible and so the KnowledgeBase class, which corresponds to the abstraction specified in the dpANS, is entirely optional. While one must create type hierarchies in order to use types, there is no need to store them in a KnowledgeBase instance, and graphs may be created freely and tracked in any manner the developer sees fit. It is possible to create any number of entirely distinct type hierarchies, graphs, and knowledge bases, allowing for multiple, distinct Notio-based applications running concurrently and using one or more knowledge bases, either shared or unshared.

Operations

Several operations are defined in the Notio API. Operations are not described in terms of algorithm, but rather, in terms of the effect they produce. Thus, the focus is on defining a variety of operations rather than providing specific implementations. Currently defined operations include construction and deconstruction operations that

combine components together to form graphs and take them apart respectively. The basic canonical operations are present: copy, restrict, simplify, and join. Other operations such as type expansion, path-finding, and subgraph extraction have been included or are under consideration. Many operations can be invoked in a few different ways. For example, simplification can be performed on a specific type of relation between specific concepts, on all instances of a given relation type within a graph, and on all relations in a graph, regardless of type. Joins can be single-point, multi-point, or maximal (join criteria can be controlled using the matching schemes described below). Restrictions can be performed on type, or referent, or both.

Copying deserves special notice since this apparently simple operation can actually become quite complex when applied to compound graphs, graphs with coreference sets, or to some subset of the components in a graph. In Notio, copy operations can be performed not only on graphs, but on concepts, relations, referents, and other components. The extent and behaviour of the copying operation is dictated by a *copying scheme*, an instance of the CopyingScheme class which contains flags that direct the copying process. For maximum control and flexibility, copying schemes can be nested within each other. Nested schemes are applied to correspondingly nested graphs, allowing for different copying techniques depending on context.

The Notio API also defines methods for graph matching. This is arguably one of the most complex operations commonly performed on graphs and it takes many different forms. Notio takes an approach to matching similar to that used for copying. Graphs and other components can be matched according to a *matching scheme*, a set of flags that determine how matching should be performed. The classification of matching operations was based on the approach presented by Nagel et al. [9]. That work has been extended to better describe the matching of compound graphs and coreference sets. The flags allow control over the comparison of elements such as types, referents, designators, markers, and graph structure. Refinement of the matching scheme structure is still ongoing.

Errors and Exceptions

Notio reports errors via Java's exception mechanism. There are two basic exception types, OperationErrors and OperationExceptions. OperationErrors need not be explicitly handled in an application and are used to report problems which a Notio application may either confidently ignore or cannot handle. OperationExceptions must be explicitly handled, and are used chiefly to report problems in complex operations that are likely to throw exceptions in spite of the best planning by the developer. Using this mechanism means that Notio applications can handle errors at any level they choose but still keep their source code relatively simple. Implementation layers can also embed their own error information within Notio's exception classes, thus providing a means for passing implementation-specific error messages through the API cleanly.

Translators

In addition to the basic conceptual graph structures and operations, Notio offers a standard interface for translators which parse or generate various CG formats such as the Conceptual Graph Interchange Format (CGIF) and the Linera Format (LF). This common interface allows pluggable translation modules so that new translators can be added without altering the API itself. Applications can use Java's dynamic class loading capabilities to load new translators without recompilation. Translators can act on strings or streams, allowing use of files or network connections for translation.

Extensions

Naturally, researchers are going to want features not offered by the Notio API, either because they are too specific or are the result of new research. Instead of creating a separate library or modifying the API in incompatible ways, developers can add features using the formal extension mechanism. This allows new features to added to the API in such a way that they can be discovered, activated, and configured by applications in a programmatic fashion. It is hoped that commonly desired extensions will be standardized, and that multiple implementations of the same extension will be usable by applications in much the same manner as the core Notio API.

Omissions and Variable Compliance

While the Notio API tries in some ways to act as a lowest common denominator for CG operations, it was decided that requiring complete compliance from implementation layers would result in overly-demanding implementation projects or an excessively restricted API. It is not reasonable to expect all implementation layers to offer all of the operations defined in the Notio API. The most notable of these is graph matching which is extremely flexible and general in the API specification. A particular implementation layer may only be able to perform a subset of the possible matching operations. Thus, implementation layers may omit some operations and raise an exception indicating that the desired operation is not available. Applications should also be able to programmatically determine whether a given feature is available or not. These mechanisms have not been completely defined as we wish to relate them to the *conformance pairs* discussed but not defined in the currently available dpANS.

The Reference Implementation Layer

Along with the Notio API itself, we have developed an implementation layer that serves as the prototype and reference implementation for the Notio API. The *reference implementation layer* is written completely in Java and is correspondingly both portable and capable of running in web browsers. In developing the reference implementation, the emphasis has been on simplicity and clarity rather than

performance. Efficiency concerns will be addressed once the API itself has achieved a high level of stability but it is hoped that alternate implementations will more effectively fill this need since it is desirable to keep the reference implementation fairly simple.

Another role for the reference implementation will be as an example and starting point for new implementation layers. One way to rapidly develop a new implementation layer would be to gradually replace functionality in the reference implementation. As far as possible, the reference implementation was designed with this possibility in mind, and uses standard API calls internally where possible rather than implementation specific calls.

At the time of writing, the reference implementation's source code is not publically available but the compiled package is. This is because we wish to focus public attention on the API specification rather than the underlying implementation. The present plan is to release the reference implementation under GNU's Library General Public License [3], which allows the use of the library in commercial applications. However, the best means for making the source available are still under consideration. Our intention is that the API and reference implementation should be as freely available as possible without compromising the goal of achieving a level of standardization. We anticipate releasing the source during 1999.
The API specification is generated directly from source of the reference implementation using the Java Development Kit's (JDK) javadoc tool. This helps ensure that the API specification and reference implementation are synchronized and up to date. Bugs, ideas, and other details of the development process are handled in a similar manner.

The reference implementation is bundled with other packages which include translator implementations for CGIF and LF, a testing suite, a set of compiled examples, and a demo suite. The documentation consists of the API specification in HTML format, automatically generated lists of bugs and ideas, and source code for the examples.

Both the API documentation and the reference implementation are available from:
http://dorian.cis.uoguelph.ca/CG/projects/notio/index.html

Using the API

To give an example of the kind of work that may be done with the API, consider a simple tool we have created as a testbed for Notio. This tool provides a GUI interface that manages a set of graphs. All graph display is accomplished using CGIF. Operations such as copy, join, and simplify are provided through lists of nodes and pull-down menus. Graphs can be entered in a window, or read from files and URL's,

[3] Information on GNU's LGPL is available at http://www.gnu.org.

all in CGIF format. They can also be saved in CGIF or LF files. The greatest effort in developing this tool was in building the user interface. The actual code involved in implementing operations and managing the objects was very small and quickly written.

Future Work

The specification of the Notio API is not complete although the essential structures are unlikely to change and we are currently engaged in a process of enhancement and refinement. The reference implementation is at the Aalpha@ stage and implements most of the specified features. It is already in use for three different projects at the University of Guelph including a database for storing and searching CG-s, a knowledge browser and summariser, and our own Ossa project, a conceptual modelling tool. Other institutions are also considering or using the Notio API in their own projects.

The present API is focussed on operations involving one or two graphs. We hope to add interfaces for working with larger sets of graphs. We are very interested to hear from the CG community about their needs and views on such an API and solicit any comments or criticism. We hope to start creating alternate implementation layers soon and encourage other groups to approach us about building implementations or an interface to existing systems. We are also starting work on a set of reusable JavaBean components based on the Notio API to handle common needs like graph editing and layout. These components will further our aim of helping to prevent unnecessary repetition of work in the CG community.

In conclusion, we encourage members of the CG community to examine, evaluate, criticize, contribute to, and above all, use the Notio API.

References

1. Finnegan Southey, AOssa: A Modelling System for Virtual Realities Based on Conceptual Graphs and Production Systems@, MSc. thesis, supervisor Dr. James G. Linders, University of Guelph, Canada, 1998.
2. John F. Sowa, AConceptual Structures: Information Processing in Mind and Machine@, Addison-Wesley, Reading, MA, 1984.
3. Brian Garner, Eric Tsui, and Dickson Lukose, ADeakin Toolset:Conceptual Graph Based Knowledge Acquisition, Management, and Processing Tools@, in *Proceedings of the Fifth International Conference on Conceptual Structures*, ICCS-97, pages 589-593, Springer, 1997.
4. Dickson Lukose, ACGKEE: Conceptual Graph Knowledge Enginnering Environment@, in *Proceedings of the Fifth International Conference on Conceptual Structures*, ICCS-97, pages 589-593, Springer, 1997.
5. G. Ellis and R. Levinson, AThe Birth of Peirce@, in *Proceedings of the Seventh Annual Workshop on Conceptual Structures*, pages 219-228, Springer-Verlag, 1993.

6. Michael Chein, AThe CORALI Project: From Conceptual Graphs to Conceptual Graphs via Labelled Graphs@, in *Proceedings of the Fifth International Conference on Conceptual Structures*, ICCS-97, pages 65-77, Springer, 1997.
7. H. Petermann, L.Euler, K.Bontcheva, ACGPro - a PROLOG implementation of Conceptual Graphs@, Technical Report FBI-HH-M-251/95, University of Hamburg, October 1995, 35 pages.
8. John Sowa et. al., ADraft Proposed American National Standard (dpANS) for Conceptual Graphs@, unpublished, February, 1999.
9. Timothy E. Nagle, John W. Esch, & Guy Mineau, AA notation for conceptual structure graph matchers@, in AConceptual Structures: Current Research and Practice@, Timothy E. Nagle, Janice A. Nagle, Laurie L. Gerholz and Peter W. Eklund (eds.), Ellis Horwood, 1992.