3721 Executive Center Drive
Suite 100
Austin, TX 78731-1615
www.cyc.com
(512) 342-4000
info@cyc.com

The Cyc® System: Notes on Architecture

Nick Siegel, Keith Goolsbey, Robert Kahlert, and Gavin Matthews

November 2004

Cycorp, Inc.
3721 Executive Center Drive
Suite 100
Austin, Texas 78731-1615

{nsiegel, goolsbey, rck, gmatthew}@cyc.com

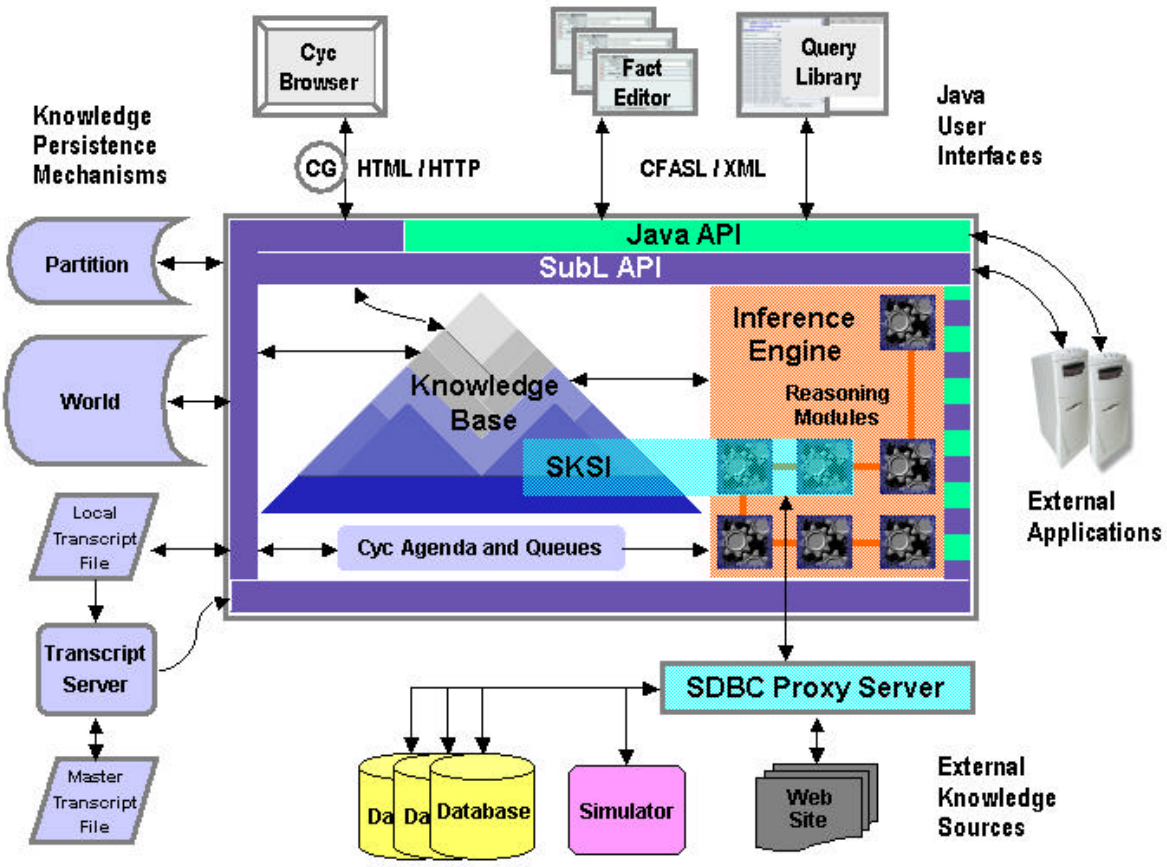

Figure 1: The Cyc System

Introduction: Cyc and the Cyc System

This document describes the architecture of the Cyc System, which includes a knowledge-based reasoning program, Cyc, along with Cyc's supporting knowledge persistence mechanisms, user interfaces, and application programming interfaces (APIs). Particular attention is given to those mechanisms that enable the transfer of knowledge (assertions) to and from Cyc's knowledge base (KB), and that allow communication between components of the Cyc system and other running programs.

In its compiled form, the core of the Cyc System -- the Cyc knowledge-based reasoning program (usually called just "Cyc") -- is contained in two files: the world and the Cyc executable. The world file contains efficiently stored declarative knowledge (facts and rules). The executable file consists mostly of compiled procedures and their supporting data structures.

The world file contains a copy of the knowledge in the KB that has been translated into a compact, efficiently loaded binary format called CFASL.

The Cyc executable file contains the compiled object (machine-level) code for the inference engine and for the Cyc agenda. The inference engine allows a running Cyc image to derive new conclusions from the facts and rules stored in the KB. The agenda drives the processing of queued KB update operations (resulting, for example, from edits submitted by a user). The executable file also contains the compiled code for the functional interface (FI) and network socket connections that support the Java API, and for the HTML generation procedures that implement Cyc's CGI-based web browser user interface.

When the executable file is invoked to start a running process, part of the information that must be provided (*e.g.*, via a command line argument or in a configuration file) is the name of the desired world file. The newly started process will try to locate the specified world file and, if successful, will try to load the entire contents of the file into memory (RAM plus virtual memory).[1]

The procedural core of Cyc, including the inference engine, is developed and implemented in SubL, a dialect of Common Lisp. SubL, in turn, is implemented both in Common Lisp and in ANSI C. To generate a new C executable, SubL source code files are translated into C source code files, which are then compiled and linked. Because every C executable implements a complete SubL interpreter, SubL is the language of choice for server-side scripting.

The Cyc System includes the following components and mechanisms, each of which will be explained below:

- Knowledge Base
- Worlds
- Inference Engine
- User Interfaces

[1] At the time of writing, the Cyc executable is just under 100Mb in size, and the world file is just under 1Gb. A typical memory configuration for Cyc running on Linux includes 1.5Gb of RAM and 2Gb of swap space (virtual memory).

- Transcripts and the Transcript Server
- Partitions
- Semantic Knowledge Source Integration (SKSI) Facility
- Application Programming Interfaces (APIs)

Knowledge Base

At the time of writing, the full version of the KB contains over 2.5 million assertions (facts and rules) interrelating more than 155,000 concepts. Most of the assertions in the KB are intended to capture "commonsense" knowledge pertaining to the objects and events of everyday human life, such as buying and selling, kinship relations, household appliances, eating, office buildings, vehicles, time, and space. The KB also contains highly specialized, "expert" knowledge in domains such as chemistry, biology, military organizations, diseases, and weapon systems, as well as the grammatical and lexical knowledge that enables the natural language processing (parsing and generation) capabilities incorporated into Cyc's user interfaces (see below).

Worlds

A world file is a snapshot of a portion of some Cyc image's virtual memory that has been "dumped", or saved, in CFASL form. Because every assertion encoded in a world file was already present in the knowledge base of some previously running Cyc image and is therefore guaranteed to be in canonical form, a newly started Cyc image can load the contents directly into main memory without performing any integrity checks. This is vastly faster than augmenting the knowledge base via a method that requires each input expression to be translated into canonical form, such as loading a transcript file (see below).

Inference Engine

Cyc's ability to reason is provided by an inference engine that employs hundreds of pattern-specific heuristic modules, as well as general, resolution-based theorem proving, to derive new conclusions (deduction) or introduce new hypotheses (abduction) from the assertions in the KB. Cyc's inference engine is multi-threaded, which means that it is able to work on multiple tasks (such as question answering or problem solving) at the same time. Intermediate results are saved in ephemeral (in-memory) data structures called "problem stores". This allows the inference engine to resume work on a problem -- as, for example, when a user decides to allow Cyc more time to find an answer – without having to repeat steps.

The inference engine is capable of deriving new conclusions via both forward (opportunistic, update driven) and backward (query driven) reasoning. It is also able to provide complete explanations for its answers, including the names of the sources (*e.g.*, people, published works, web sites) from which information was obtained. It can even alert the user in cases where both pro and con arguments can be constructed for particular conclusions, perhaps due to differing circumstances or changes in context. Users can modify dozens of parameters to achieve very fine-grained control of inference, if desired. A subcomponent of the inference engine – the Experience Based Tactician – learns to favor productive search paths, so that inference performance automatically improves over time.

User Interfaces

Operations on the KB (additions, modifications, deletions, and queries) are stated using Cyc's formal, declarative representation language, CycL, which is based on second order predicate calculus. Cyc supports several user interfaces, for skill levels ranging from expert to novice.

The Cyc Browser consists of dynamic (CGI-generated) HTML pages that allow experienced users to query, browse, edit, and add to the contents of the knowledge base. Use of the Cyc Browser requires that some web server be installed in order to invoke cg, the small CGI program that processes httpd requests for designated Cyc images. Cyc comes with its own simple HTTP server installed, but many Cyc users prefer to use an industrial-strength HTTP server, such as Apache.

The Fact Editor is a template-based Java interface that allows even relatively inexpert users to add new facts (ground assertions) to the knowledge base, and edit existing facts. The Query Library is a Java interface that allows users to pose pre-formulated queries (such as the libraries of queries already defined for particular domains), and to compose new, arbitrarily complex queries by assembling and editing pre-existing query fragments and templates. All of these interfaces employ Cyc's natural language generation capabilities to render assertions, queries, and query answers in English, shielding users from the underlying, and sometimes dauntingly complex, CycL formalisms.

Both the Fact Editor and the Query Library can be run as applets in the Cyc Browser. Both are implemented with the Cyc Java API, which provides a framework for an external Java program to communicate with a Cyc image.

Transcripts and the Transcript Server

Every running Cyc image has a separate copy of the knowledge base. This raises the question of how, at a site where several people are using Cyc and the KBs of several images are frequently modified, the different KBs are kept in sync. This is accomplished by having each image write its own operations (KB modifications) to a sequence of transcript files. As operations are added to the image's local transcript file, they are also periodically transmitted to the Transcript Server, a small Java program that maintains a master file of all changes received from any image. If an image is configured to receive dynamic KB updates from other images (this is optional), it periodically connects to the Transcript Server, downloads any new operations to a dedicated queue, and runs the operations.

Cyc's transcript file mechanism works well as a way of keeping separate copies of the knowledge base in sync for days, or even weeks, at a time. But as the number of operations in the master transcript file increases, it becomes increasingly time consuming for a newly started Cyc image to process all of the operations. Each operation is checked to make sure that its encapsulated assertion is in the correct (canonical) form, and the time required to perform this check adds up. Eventually, a Cyc image that is completely up-to-date (*i.e.*, that has processed all of the operations contained in the master transcript file) should be used to write out a new world file. This world then becomes the new foundational knowledge base that all Cyc images load when they start up, and to which new operations are applied. The transcript files are reset, so that only the changes to the new foundational knowledge base will be recorded.

Partitions

Transcript files are a safe, effective way of transferring assertions between different Cyc knowledge bases, because the integrity checks that are performed when a transcript file is loaded prevent non-canonical assertions from being added. Another way to transfer assertions from one instance of the knowledge base to another is to use Cyc's partition mechanism. The mechanism uses a configuration file to allow the user to declaratively specify the kinds of assertions to be selected. A procedure then uses the information contained in the configuration file to write CFASL files containing the matching assertions and, if desired, to delete the assertions from the source knowledge base. The CFASL files can then be loaded into a target knowledge base, ideally bringing it into alignment with the source knowledge base. Since all assertions in the CFASL files are assumed to be formally correct (since they were in the knowledge base from which they originated), this process works best when the target knowledge base is closely related to the source knowledge base.

Semantic Knowledge Source Integration (SKSI) Facility

SKSI provides a means for Cyc to seamless access external structured data. The Cyc System includes specialized CycL vocabulary, inference modules and supporting connection management code that together constitute a facility called *Semantic Knowledge Source Integration* (SKSI). SKSI's CycL vocabulary supports detailed semantic descriptions of external information sources, such as databases and web sites. These semantic descriptions render explicit the entities, concepts, and relations that often are only implicit in a source's implementation data model (e.g., in DBMS metadata), and that link these explicitly represented items into the vast amount of knowledge already in the KB. The inference engine can use this explicitly represented knowledge about external data sources, along with SKSI's specialized inference modules, to access the sources and reason with their data as if they were part of the KB. When presented with a CycL query, Cyc's inference engine is able to dispatch SQL SELECT statements to one or several relevant databases, send form submissions to relevant web sites, search the information contained in the Cyc knowledge base, and combine results obtained from all of these types of sources to provide an answer (a set of bindings).

SKSI also allows the inference engine to update external knowledge sources. This capability makes possible complex transformations in which a Cyc image retrieves data from one or more sources, combines or filters it, and writes the results to data tables.

The dispatch of appropriate query and update expressions to different types of external sources is managed by the Semantic Data Base Connectivity (SDBC) proxy server. This is a Java program that accepts requests from a Cyc image, converts them to the correct format (SQL, HTTP GET, CORBA, SOAP, *etc.*), dispatches the requests to the correct location, receives the results, and then sends them back to the Cyc image.

Application Programming Interfaces (APIs)

The Cyc System provides two API layers: SubL and Java. The SubL API exposes methods that allow external programs to access the inference engine and, via it, to perform operations (queries, updates, deletions) on the knowledge base. External programs can access the SubL API of a Cyc image via TCP/IP, and can exchange messages with it in either plain text (ASCII) or CFASL format, as appropriate. The SubL API provides no support for callbacks. Invocations of SubL API methods to query or update the knowledge base can block to wait for a response,

which may be unacceptable for many applications. A suboptimal but sometimes adequate solution is to check progress via polling.

The Java API is built on top of, and greatly improves, the SubL API. It provides support for event listeners and KB event notification, whereby observers can register their interest in certain types of changes in the contents of the knowledge base, and also for iterative answer/data retrieval from a running Cyc image. Cycorp is moving toward greater use of XML as an application interchange and messaging format, in part because many tools for parsing and manipulating XML now exist in Java. Cyc supports the CycML language, which is a simple translation of the CycL representation language into XML syntax.

There are two main choices to consider when designing an external application that will communicate with Cyc: whether to use the SubL API as opposed to the Java API, and whether to be client- or server-centric. In general, direct use of the SubL API that bypasses the Java API, while possible, is not recommended. A client-centric approach pushes more of the responsibility for data processing, state maintenance, and application flow-of-control into the client (*i.e.*, the connecting external application), while a server-centric approach pushes more of this activity into Cyc. As previously mentioned, Cyc's procedural core is implemented in SubL, and while the Cyc executable itself is compiled into C code, it contains a full SubL interpreter. This allows Cyc to be dynamically extended and scripted by external applications. An external application can call public SubL functions that have been added to the system and define new such functions, a likely occurrence within any collaborative project. This sort of approach is harder to code than a "pure Java" approach that requires no knowledge of SubL, but it is also much more flexible.

Use of the Java API to connect to a Cyc image requires the creation of an instance of the CycAccess class. The Java API provides a set of classes that serve as wrappers for Cyc's native (SubL and CycL) data objects and data values. Each of these client-side wrappers knows how to associate itself with the corresponding server-side object. For knowledge base objects that use GUIDs or other such unique identifiers, the Java API uses these identifiers. Objects such as SubL strings and numbers are represented by Java objects of the (usually obvious) corresponding type. Figure 2 depicts a subset of the class hierarchy for the Java API. Note that these classes support many of the types that define, or are important for, CycL.

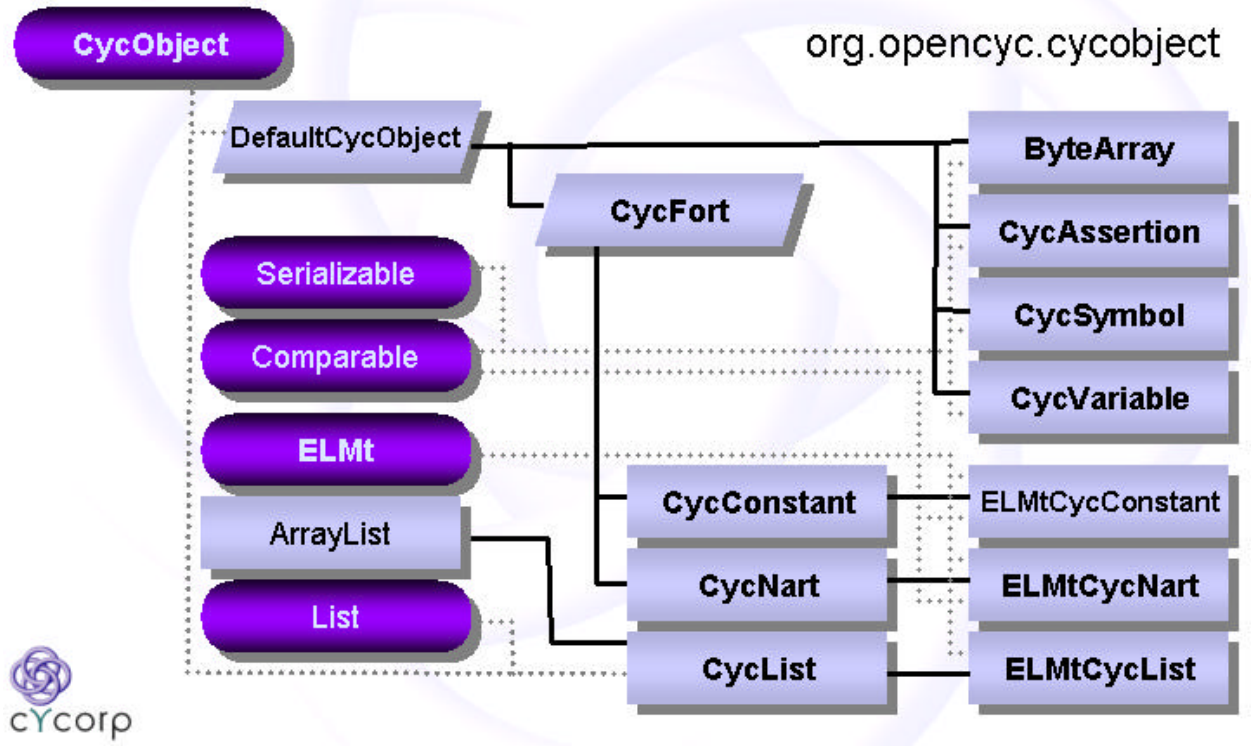

Figure 2: Java API Data Type Classes

One problem with the "obvious" way to build an external application that uses Cyc, such as a new type of agent, is that calls to Cyc (*i.e.*, to the underlying SubL methods) will block the current thread until the Cyc server completes the request. In a real application this would be unacceptable. For example, when updating the knowledge base, the agent might not care exactly when the assertions are processed, but when asking a query, it might need to have some sort of immediate, partial response followed by a stream of answers as they are computed (*i.e.*, an iterative response). This is especially important in a GUI application. In general, blocking methods should never be called within a display management thread, since this may cause UI responsiveness to plummet. The Java API defines special SubL Worker classes to circumvent this problem and allow external applications to capitalize on the benefits of multi-threading. Figure 3 depicts the class hierarchy for the Java API's SubL Worker classes.

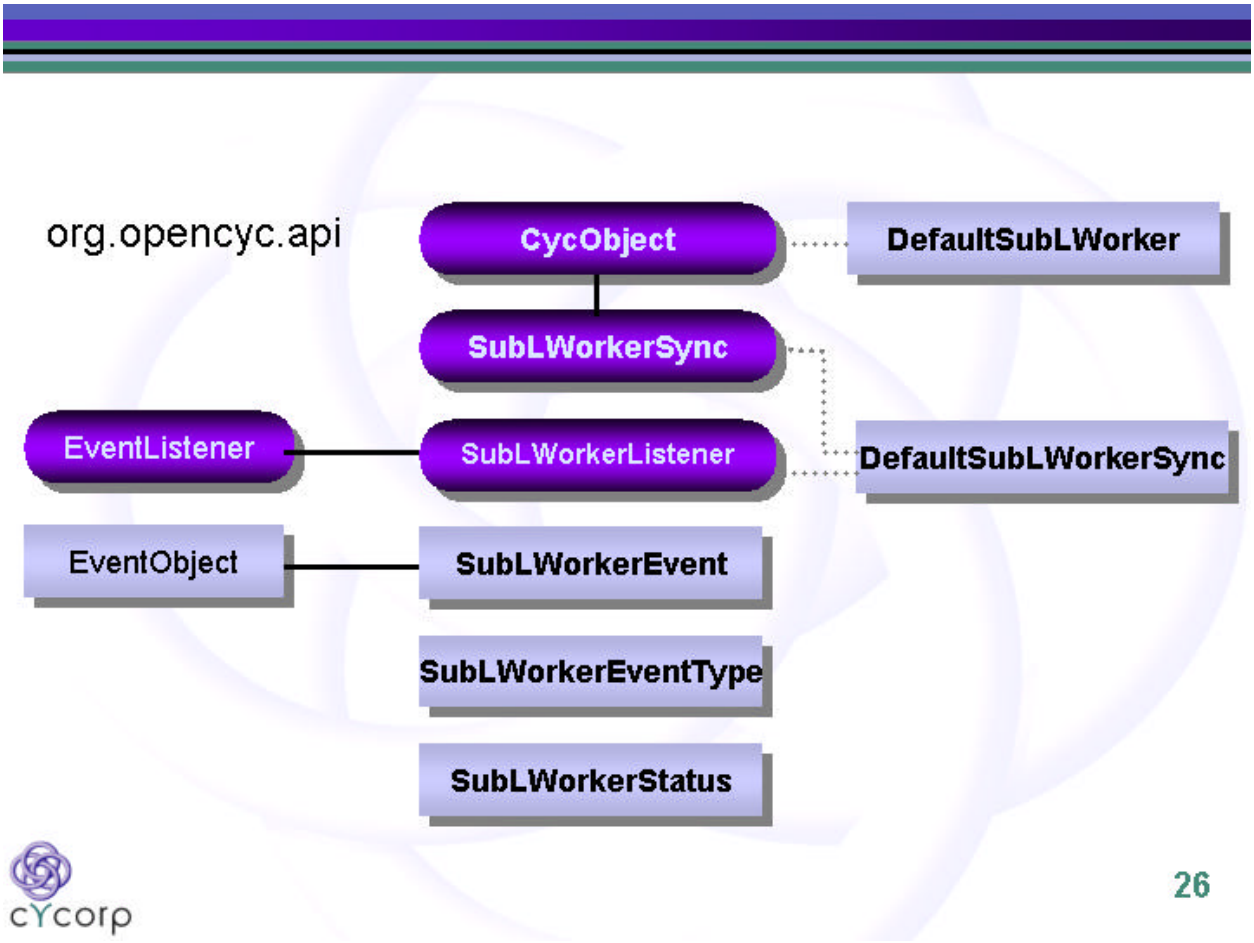

Figure 3: Java API SubL Worker Classes

When using the Java API to connect to Cyc, five pieces of connection configuration information must be provided, either by the connecting program, or in a Java properties file:

1. Location (hostname) - The Cyc server (*i.e.*, the running Cyc executable) could be running on the same machine as the connecting application or somewhere else.
2. Base port number – A Cyc image can listen on several different ports, for different protocols. These ports are found at specific offsets from the base port. For this reason, different Cyc images should have base numbers at least 20 apart. By convention, the base port is normally assigned to 3600. The base port number of a Cyc image is normally set when the image is first started.
3. Protocol – As already mentioned, the Java API can use two different protocols, plain text and CFASL. These are the default protocols for port offsets 1 and 14, respectively. Generally, users of the Java API will prefer the CFASL (binary) protocol, which is more compressed. The ASCII interface is provided for legacy applications.
4. Connection persistence – Keeping a persistent connection open generally results in better performance than frequently opening intermittent connections.
5. Aggressive loading – Some object properties and other commonly used data can be

fetched before use and cached, which usually results in better performance.

The first thing an external application must do to use Cyc is establish a connection with the Cyc server. This is done by instantiating the CycAccess class. The following code snippet shows how this is done. Note that in this example, the hostname and base port are explicitly referenced in the constructor invocation. This example also illustrates the three types of exceptions the constructor can throw. Two pertain to network problems, and one is a Cyc-specific indicator of internal server problems or protocol errors.

```
try {
  CycAccess access = new
    CycAccess("myhost.domain.com", 3640);
  System.out.println(
    "Successfully established CYC access " +
     access.connectionInfo());

  // do stuff with the connection here

  access.close();
} catch (IOException io) {
  // if a data communication error occurs
} catch (UnknownHostException nohost) {
  // if cyc server host not found on the network
} catch (CycApiException cyc_e) {
  // if the api request results in a cyc server error
  // example: cannot launch servicing thread;
  // protocol errors, etc.
}
```

Cycorp's developers offer the following recommendations for enabling external applications to communicate with, and effectively use, Cyc.

1. Let Cyc store the information. By all means cache information on the application (client) side, but be prepared for Cyc to update it. Design your application so that it registers the appropriate listeners, and so can be informed of the changes occurring on the Cyc server.
2. Cyc should be responsible for ensuring the integrity of the data. Often, Cyc can suggest potential values that the application can offer to the user, but the application does not necessarily have the full picture about what changes are and are not valid.
3. Cyc should not be involved in the details of rendering (GUI display). It is up to the application how the information should be laid out, and what widgets should be used. Cyc can and should provide information that can be used to decide between specific interface metaphors, and Cyc can be used as a repository for storing rendering information and user preference models (*e.g.*, Cyc knows that some of its regular users are color-blind), but the external application should have ultimate control of presentation.

A full description of programming with the Cyc SubL and Java APIs now warrants a book-length treatment, but the notes above should provide prospective developers with a sense of the possibilities.