The API Perspective of Exchange Formats

Holger M. Kienle, Jörg Czeranski, and Thomas Eisenbarth
Bauhaus Group
University of Stuttgart, Germany
phone: +49-711-7816-355
{kienle,czeranski,eisenbarth}@informatik.uni-stuttgart.de

ABSTRACT
The discussion about a universal exchange format mainly centers around syntax and semantic issues. This paper focuses on the requirements for the API that sits on top of an exchange format.

Keywords
Exchange format, in-memory representation, requirements

1 Introduction
The discussion about a universal exchange format focuses mainly on syntactic and semantic issues. The syntactic part is concerned about how to represent the data that we want to store. The semantic part (sometimes referred to as *schema*) deals with the structure (or the interpretation) of the data itself.

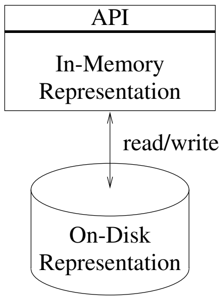

Figure 1: Exchange format architecture

We believe that another important aspect is the *in-memory representation* of the exchange format (as opposed to the *on-disk representation*, which is usually focused on) and the application programmer interface (API), which sits on top of the in-memory representation (see figure 1). Exchange formats are often introduced without discussing their in-memory representation and API. The API of an exchange format is used to (1) read and write the format and (2) inspect and modify the in-memory representation. This paper focuses on APIs by identifying important characteristics and discussing design decisions that must be made for them.

Motivating Example
When reading the exchange format from disk, a transformation into the in-memory representation takes place. The API sits on top of the in-memory representation and determines the "look and feel" of the exchange format. Hence, one can argue that the API is much more important than the

actual, underlying exchange format. The transformation from on-disk to in-memory representation and the design of the API might seem to be tightly coupled and thus trivial, but in practice the designer has quite a bit of leeway. To illustrate this, we briefly describe two exchange formats: the Abstract Syntax Description Language (ASDL) [11] and the binary format of Annotated Terms (ATerms) [10], which is called Binary ATerm Format (BAF). The rest of the paper uses both formats as a running example to illustrate some points under discussion.

ASDL and BAF have a lot in common; namely, both are binary formats, both can represent tree-like data structures (e.g., they are well suited to express abstract syntax trees), and both formats are typed. Yet, both have a totally different in-memory representation and API.

ASDL uses a typed declarative specification language to describe the schema (i.e., the tree structure). A translator generates code from the specification for the in-memory representation and reading/writing (so called *picklers* [2]). The in-memory representation is modeled differently for procedural style (e.g., C), object-oriented (e.g., C++ and Java), and functional (e.g., ML) programming languages. For example, the code generated for C++ and Java translates tree nodes to classes with strongly typed constructors and accessors. In ASDL, an `add` node that takes two integers is specified as follows:

```
add = Add(int left, int right)
```

From this specification, the in-memory and API code is generated. To construct a new `add` node in Java, the constructor of the generated class `Add` is used:

```
Add myAddNode = Add( 42, 43 );
```

In contrast to ASDL, ATerm does not have schemas. The in-memory representation is close to the machine model in a binary encoding. Every node has a type, but the API is not statically typed. To construct a new node, a string pattern is specified that contains typed placeholders. These placeholders are then filled with already existing nodes of the appropriate type. For example, the following code constructs an `add` node, which takes two integer nodes:

```
ATerm myAddNode = ATmake("add(<int>,<int>)", 42, 43);
```

The basic API consists of 13 (generic) functions for construction, matching, and reading/writing of nodes.

2 APIs

This section starts with a brief overview of existing APIs for exchange and graph formats. We then explain why we consider language-independence an important feature of an API and finally give a list of functionalities that an API should probably have or come close to.

Existing APIs

The API sits on top of the in-memory representation. Depending on the interface, it can be a thin layer tightly coupled with the in-memory representation or an entirely different abstraction. The ASDL interface is integrated with the in-memory representation. In fact, the only function of the interface is to create in-memory data structures and to provide accessors to the members of the structure. Thus, except for the static typing rules of the host language, the interface implements no sanity checks of the built structures. The ATerm interface on the other hand completely abstracts away from the in-memory representation. Run-time checks are made to ensure consistency (including proper typing). In general, the major advantage of a statically typed interface is early (i.e., compile-time) error detection. The disadvantage is less flexibility when new structures are built. Furthermore, for every type a new representation must be offered in the host language (e.g., different constructors and accessors).

In general, the flavor of the API is heavily influenced by its application domain. As already pointed out in [3], a universal exchange format has to accommodate various (re-engineering) tools and thus must be able to accommodate various levels of abstractions and different views. The same requirement holds for the API. We know of the following flavors:

- **Embedding:** Embedding the API into a high-level programming language is by far the most common approach. If the exchange format is typed, the typing is usually partially reflected in the type system of the programming language. Constraints that cannot be reflected in the programming language must the checked by the API and errors must be dealt with. As already discussed, ASDL constructs specific APIs for every target language from a common specification, whereas ATerms are widely language neutral.

  The LEDA library [7] is a collection of data types and algorithms. LEDA has a large variety of graph data structures (e.g., parameterized graphs, planar maps, node/edge arrays, and node/edge maps). The API has the functionality to parse GML [4]graph specifications. LEDA translates the GML representation to internal data types that can be conveniently manipulated by the user.

  The Graph Exchange Language (GEL) [8] has a C API that allows the user to add callback functions to perform actions when certain structures have been read in.

- **Domain-specific language:** The API can be a specifically tailored scripting language. For example, dotty is a specific scripting language for the dot graph exchange format [9]. The scripting language supports insertion and deletion of nodes, edges, and subgraphs and has functions for reading and writing graphs.

- **Data base queries:** The objects expressed by certain exchange formats can be interpreted as entries in a relational or object-oriented database. Manipulation of such entries can then be performed with database queries. The TA exchange format [6] is based on typed graphs. An edge between two nodes is specified as a triple:

  Edge SourceNode TargetNode

  As discussed in [6], this triple can be interpreted as an entry in a binary relation of a relational database. TA has a query front-end (called Grok) that can, for example, compute the union of two relations or join two relations [5]. This means that Grok's functionality is similar to SQL queries.

- **Object stores:** In order to make in-memory structures persistent, object stores can be used. For example, the PGraphite system [12] is an object store for graph data structures. The API allows to create, open, and delete repositories. For a give repository, nodes can be inserted, retrieved and removed. Nodes must be explicitly created and can be uniquely identified with a persistent identifier (PID).

Language Independence
A universal exchange format has to offer an API for a large variety of programming languages. Ideally, the interface should offer the same functionality independent of the programming language.

Care should be taken that the feature set of one language does not influence the design of the offered functionality in such a way that it becomes unreasonably hard to implement them in other languages. Still language features can be used for specific implementations of the API: the *what* of the functionality should be the same across languages, while the *how* might differ. For example, it would be sensible to use garbage collection if the language supports it, instead of forcing the programmer to handle deallocation. Similarly, exceptions can be used for error handling where they are available.

The situation is more difficult for types. Untyped interfaces can easily have the same structure and offer the same level of abstraction in different languages. For example, the ATerm interface discussed above consists of a small number of (global) functions that can be consistently offered through various languages. Typed interfaces on the other hand tend to map the type system imposed by the exchange format to the type system of the programming language. ASDL constructs typed interfaces and data types for the in-memory representation for C, C++, Java, and ML from the same specification. The interfaces and data-types for different languages (naturally) look quite different.

Thus there is a tradeoff between exploiting the type system of the language, resulting in earlier error detection and possibly better performance, and interfaces having the same structure for all languages, hence decreasing effort for programmers who migrate tools to a different language or who use multiple languages. Furthermore, the level of detail of consistency checks should be the same for all API implementations, to ensure that the same data can be represented regardless of language.

Functionality
The functionality provided by the API should be general enough to be suitable for a large range of applications. Otherwise, applications are forced to first convert the data from the API into an application-specific representation. If such a conversion has too many drawbacks (e.g., difficult to implement and inefficient to perform), tool writers will probably decide in favor of a specific exchange format whose API fits their needs.

The following are important points to consider when designing an API:

- **Introspection:** Introspection is the ability to inquire for meta-information such as the structure of an entity. (Introspection is also called reflection, e.g., in Java.) For example, the user can ask for the kind of an entity and the attributes associated with that entity. The most primitive form of introspection is a simple type check. The interface of the Graph Description Language (GDL) [12] offers these basic services. If the schema allows inheritance, subtype checks can be offered. The SUIF compiler system [1] implements an introspection framework in C++ for the Hoof specification language. The introspection interface is of special interest for tools that have to traverse the structure of the in-memory representation and access information in a uniform, abstract way (e.g., generic printers and readers/writers).

- **Navigation:** The interface should offer the ability to conveniently navigate the graph structure. Depending on the application different traversal strategies are needed. For example, some applications require a traversal of the entire graph, other applications only want to query for specific nodes or edges, even want to match subgraphs. The traversal for tree structures is easier to support (and more intuitive for the user) than traversal of arbitrary graphs. The ATerm API offers matching of an existing ATerm by comparing it with a specified pattern. ASDL has no navigation support, except for simple accessors.

- **Modification:** Modification of the in-memory representation is another important property. For example, if a tools wants to convert between two different representations, the source representation must be read in, appropriately modified, and finally written back. Replacing a structure while other structures reference it should be supported by the API. ATerms become immutable after creation. (This property allows efficient sharing of identical terms.) ASDL supports destruction of structures, but no sanity checks are performed (e.g., if destruction causes dangling references). The SUIF compiler system uses its introspection framework to implement several navigation strategies and to allow easy modification during traversal.

- **Annotations:** There should be an easy way for applications to transparently extend the main data structures with annotations to represent domain-specific information. This of course implies that the underlying format supports a general annotation mechanism. Annotations should be typed and structured. For example, ATerm annotations are ATerms themselves and thus both typed and structured. Annotations can be attached to every ATerm and distinguished with labels. ASDL offers no annotation mechanism.

- **Garbage Collection:** Changing the structure of the in-memory representation (e.g., deletion of nodes) can cause unreferenced entities. Various levels of support are possible: If the entity is a dynamically allocated object in the programming language, the user can explicitly deallocate it. This is unsafe because no sanity checks can be made before deallocation. If the interface supports deallocation, sanity checks can be made. The most convenient and least error-prone solution is automatic deallocation with a garbage collector. The ATerm implementation has its own garbage collector. SUIF allows the user to "trash" objects. Trashed objects are not deallocated right away. Instead, they are kept as annotations. The user can explicitly request garbage collection, which traverses the in-memory representation and removes the trashed objects that are unreferenced.

- **Error Handling:** If the user violates the protocol of the interface (e.g., violation of integrity constraints imposed on the graph structure), the interface must deal with the situation in a sensible manner. The easiest (and least attractive) solution is to abort with an error message. More sophisticated approaches communicate errors back to the user (e.g., with a return value or by raising an exception).

3 Afterthought

Realizing all of the outlined requirements mentioned in the previous section in a single API will be hard to achieve in practice for two reasons. First, the requirements that applications have are too diverse. For example, some applications only want to query structures in an efficient way, while others have to traverse, modify, and annotate the structures. Secondly, some requirements are contradicting or hard to unify (for examples, refer to the discussion about language-independence).

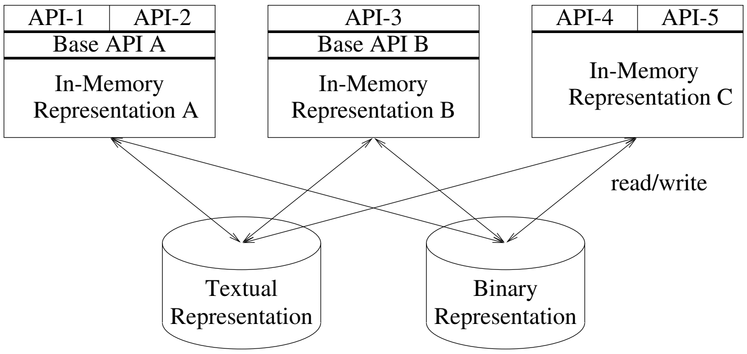

Figure 2: Proposed exchange format architecture

Indeed, in order to accommodate various requirements, an exchange format should offer several APIs, with different trade-offs and flavors. For example, some applications need a database style query language (e.g. Grok), while others need a graph API such as LEDA. Different APIs will probably need different underlying in-memory representations. Hopefully certain in-memory representations can be designed in such a way that several APIs can sit on top of them. A specific in-memory representation will usually offer a simple base API, which is then extended by domain-specific APIs. All in-memory representations use a single universal exchange format (which has a textual as well as a binary representation). Figure 2 illustrates the proposed architecture.

REFERENCES

[1] Gerald Aigner, Amer Diwan, David L. Heine, Monica S. Lam, David L. Moore, Brian R. Murphy, and Constantine Sapuntzakis. *The Basic SUIF Programming Guide*. Computer Systems Laboratory, Stanford University, November 1999.

[2] A. Birrell, M. Jones, and E. Wobber. A simple and efficient implementation of a small database. *Proceedings of the Eleventh ACM Symposium on Operating systems principles*, pages 149–154, November 1987.

[3] Jürgen Ebert, Bernt Kullbach, and Andreas Winter. GraX — an interchange format for reengineering tools. *WCRE '99*, pages 89–98, 1999.

[4] Michael Himsolt. GML: Graph modelling language. Draft, Unpublished, December 1996.

[5] R. C. Holt. Binary relational algebra applied to software architecture. Technical Report 345, CSRI, University of Toronto, 1996.

[6] Ric Holt. An introduction to TA: The tuple-attribute language. http://plg2.math.uwaterloo.-ca/~holt/papers/ta.html, 1997.

[7] LEDA Research Homepage. http://www.mpi-sb.mpg.de/LEDA.

[8] J.F.Th. Kamperman. GEL, a graph exchange language. Technical report, CWI, Amsterdam, Netherlands, 1994.

[9] Stephen C. North and Elefherios Koutsofios. Applications of graph visualization. *Graphics Interface '94*, pages 235–245, 1994.

[10] M.G.J. van den Brand, H.A. de Jong, P. Klint, and P.A. Olivier. Efficient annotated terms. August 1999. To appear in Software—Practice and Experience.

[11] Daniel C. Wang, Andrew W. Appel, Jeff L. Korn, and Christopher S. Serra. The Zephyr abstract syntax description language. *Proceedings of the Conference on Domain-Specific Languages*, pages 213–228, October 1997.

[12] Jack C. Wileden, Alexander L. Wolf, Charles D. Fisher, and Peri L. Tarr. PGRAPHITE: An experiment in persistent typed object management. *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical software development environments*, pages 130–142, November 1988.