Exchange Format Bibliography

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
This paper gives a brief bibliographical overview of exchange formats and related research areas. We classify exchange formats and try to give a brief assessment of the more interesting ones.

Keywords
Exchange format, classification, overview, bibliography

1 Introduction

This paper gives a brief bibliographical overview of exchange formats and related research areas. Specifically, we focus on the following areas:

- Formats to represent graphs (section 2).
- General data exchange formats (section 3).
- Persistent intermediate representations in compiler construction (section 4).
- Making data structures persistent (section 5).

Section 6 closes the paper with a brief discussion how the above approaches specify constraints on otherwise unrestricted structures (in the following called schemas).

2 Graph Formats

Graph drawing and editing tools typically have their own graph specification format. The format has typically a textual representation that is easy to read and write. The benefits of this approach are twofold. Humans can write their own graph specification directly in the graph format (without necessarily using a graph editing tool) and tools can easily generate and parse graph data. On the downside, these formats are not space efficient. Graph formats share the following characteristics: (1) Expression of arbitrary graphs, (2) distinction between nodes and edges, and (3) attribution of nodes and edges.

Dot [18] [21] and neato [20] are two graph layout tools that share an almost identical graph format. Dot makes layouts of directed graphs whereas neato is for undirected graphs. The format has three kinds of items: graphs, nodes, and edges. The top-level graph can be structured by introducing subgraphs, which define a subset of nodes and edges. Nodes are identified by their name. Edges are created when nodes are joined with an edge operator. Graphs, nodes, and edges can be attributed. Attributes are name-value pairs of character strings. It is also possible to define new default values for edge and node attributes (this can be seen as a primitive form of attribute inheritance). The format has a large number of predefined attributes (e.g., node shape, line style, color, and layout parameters such as weights). Application specific attributes, which are ignored by dot, can be attached as well. As a simple example, we give a dot graph specification that contains two nodes (called `one` and `two`) that are connected with an edge. The edge is colored red. We will use this as a running example to give an impression how different languages specify the same graph.

```
digraph G {
  one -> two [ color= red ]
}
```

The Graph Description Language (GDL) of the VCG tool [25] describes graphs that consist of subgraphs, nodes, and edges. The GDL syntax supports special kinds of edges (e.g., `nearedge`) that control the graph layout. Attributes are key-value pairs. Default attributes and redefinition of default attributes for edges and nodes are supported as well. Nodes are identified with an obligatory `label` attribute. Edges must have `sourcename` and `targetname` attributes that refer to the nodes' labels. GDL has a very rich set of predefined attributes. The core of GDL is compatible to GRL, the specification language of the Edge [24] graph editor. The following gives a code example:

```
graph: {
  node: { title: "one" }
  node: { title: "two" }
  edge: { sourcename: "one" targetname: "two" color: "red" }
}
```

The Graph Modeling Language (GML) [13] (which is used by Graphlet [15]) has a simple and orthogonal syntax. A GML file consists of key-value pairs. Values can be integers, floats, strings, and lists. Integers are restricted to 32 bit and floats are double precision. A list consists of key-value pairs. Duplicate keys within the same list are legal and it is guaranteed that the order of these duplicates is preserved during reading and writing. This makes it possible to represent array and list data structures. GML has predefined keys like `graph`, `node`, and `edge`. A graph is represented with the top-level key `graph`, whose value is a list that contains `node` and `edge` keys. Every node must be given a unique integer id. The id is then used to represent edges between two nodes. GML defines default values for every data type. These defaults are then used for missing key-value pairs. Keys unknown to GML are handled differently, depending if the key is considered safe or unsafe. Unknown keys that are safe are considered invariant to graph modification and hence are ignored (i.e., written back unmodified). In contrast, unsafe keys are deleted (i.e., not written back) if the tools cannot update them properly. Safe and unsafe attributes are distinguished with the following convention: Safe (unsafe) attributes start with a lower (an upper) case letter. The following gives a code example:

```
graph [
  node [ id 1
         label "one" ]
  node [ id 2
         label "two" ]
```

```
    edge [ source 1
           target 2
           graphics [ fill "red" ]
         ]
]
```

The daVinci system [10] uses terms to represent graphs, which distinguishes it from all other graph formats. Terms make the format not easy to read and hard to write. A term consists of a unique string for identification, a type name, a list of attributes, and a list of child nodes. Because each term is uniquely identified, other terms can reference them by name. This way it is possible to construct cyclic graphs. The type name is used to define default values for attributes. An attribute is a string pair, the first entry being the key and the second entry being the value. Unknown keys and attribute values are simply ignored. The following gives a code example (we do not claim correctness):

```
[ l("one", n("", [],
    [ l("one->two", e("", [a("COLOR", "red")],
        l("two", n("", [], [] )))) ] )) ]
```

Other—more general—exchange formats evolved from these graph formats. They retain the basic structure to represent graphs, but do not predefine graph-specific attributes. Thus they are well suited to represent arbitrary domain-specific information that can be attached to graphs. RSF and TA are examples of such formats.

The Rigi Standard Format (RSF) [32] uses sequences of triples to encode graphs. A triple either represents an edge between two nodes (e.g., "`myEdge one two`") or binds a value to a node's attribute. Furthermore, it is possible to assign types to nodes. RSF has two major dialects: unstructured and structured. The unstructured dialect does not support edge attributes. The structured dialect is used to specify schemas.

The Tuple-Attribute language (TA) [14] [5] is based on the RSF triple notation. Edges and nodes are specified identically to RSF. Nodes and types can be attributed. Attributes can be nested (i.e., attributes can have sub-attributes). Attribute values are either a string or a list. The list items are either strings or sub-lists. The following gives a code example:

```
myEdge one two
(myEdge one two) { color = red }
```

Besides the introduced ones, there are many other graph formats with minor variations in their expressiveness (and major variations in their syntax). We furthermore know of, but did not discuss, the GraphEd [12] file format (the predecessor of GML), and the Tom Sawyer Software format [26].

3 Data Exchange Formats

Data exchange formats are used to store and retrieve data in a structured and domain-independent way. Typically a binary format is used, but textual formats exist also. In contrast to graph formats, data exchange formats are not always suitable to easily represent arbitrary graphs. Furthermore, the textual representation can be hard to read (and impossible to write) by humans.

The Extensible Markup Language (XML) [23] was designed as a universal structured document format, but it is general enough to represent arbitrary structured data in a text file. The format of a XML file is given separately in a Document Type Definition (DTD). The XML syntax is quite verbose and not (necessarily) meant to be read by humans. XML imposes a hierarchical tree structure on the data. The Graph Exchange Format (GraX) [7] uses XML for data exchange. XML is a simplified subset of SGML [11].

The Annotated Terms (ATerms) format [29] has been designed to exchange tree and DAG data structures. Data is represented as typed terms. Examples of types are primitive types (integer, real, or binary large data object (BLOB)), function applications, and lists. ATerms have (optional) annotations that are identified with a string key. The user interface defines operations for making, matching, and reading/writing of ATerms. ATerms can be represented in textual or binary format.

The Graph Exchange Language (GEL) [17] is used for the language-independent exchange of graph-structured data. Graphs are constructed with a specific stack-based language, i.e., the construction of a node assumes that its child nodes are on top of the stack in appropriate order. Special stack elements are used to construct graphs that have cyclic references. Every node is identified by a unique integer number and has a type, an arity, and a number of edges. The type of a node is an uninterpreted sequence of bytes, which is used to attach application-specific data. GEL has a textual as well as a binary encoding. The encoding is the program itself that specifies the construction of the graph (thus a simple virtual machine is needed that interprets the program and builds the graph when the format is read).

The Abstract Syntax Notation One (ASN.1) [28] is a data declaration language. There are primitive pre-defined types (e.g., boolean, arbitrary length integer, floating point, and bit/byte strings) and type constructors for records, variants, one-dimensional arrays, and (un-)ordered sets. The actual encoding of the exchange format is specified by the ASN.1 transfer syntax, which converts types to unambiguous sequences of bytes.

XDR [27] is another data declaration language similar to ASN.1. XDR is simpler than ASN.1 and its type declarations have a syntax similar to C. Most binary formats (e.g., ASN.1 and ATerms) are optimized for size down to the bit level, which means that encoding and decoding is awkward and costly. In contrast, XDR's data unit is 4 bytes wide, which makes it easier and faster to read/write, but wastes space.

The Abstract Syntax Description Language (ASDL) [30] has been designed to describe abstract syntax trees, but any other tree data structure can be expressed as well. The trees are described with a simple declarative language which looks similar to an algebraic data type specification. ASDL supports a pre-defined (infinite precision) integer and a string type. A translator automatically generates readers and writers (called picklers) for a binary format. ASDL achieves interoperability between different programming languages by generating picklers for C, C++, Java, and ML. ASDL resembles a simple subset of ASN.1. In fact, ASDL's goal is to be less verbose, cryptic, and complex than SGML or ASN.1.

This paper does not discuss interface definition languages (IDLs), such as OMG CORBA, ONC RPC or Xerox's ILU, because they typically have awkward encodings for tree-like data structures [30, p. 214].

4 Intermediate Representations

Intermediate representations (IRs) for compilers often consist of tree-like data structures. Still, only a small number of IRs can be used to exchange arbitrary data because typically IRs cannot be made persistent and lack generality and extensibility. In the following, we discuss IRs that come close to both criteria.

The SUIF compiler system [1] defines its own specification language (called Hoof) to describe strongly-typed IR nodes. New IR nodes can be easily introduced by sub-classing from existing ones or a completely new IR can be designed by starting to subclass from the root of the inheritance hierarchy. IR nodes consist of fields. A field can have a primitive type (e.g., boolean, fixed-size integer, arbitrary precision integer, or string), a collection type (e.g., vector or indexed list), or a reference type. A reference type denotes an edge to another IR node. This means that edges are no first-class objects. References can be specified as being owning references. Every IR object must have precisely one owner, i.e., another object with an owning reference pointing to it. Otherwise IR nodes can reference each other arbitrarily, thus modeling a typed graph. Embedded within the graph is the ownership tree. The ownership tree is very useful, as it guarantees that all nodes are visited exactly once if in visiting a node we visit each of its owned objects. SUIF has a macro processor that generates C++ code from Hoof specifications. Essentially, every IR node is translated to a C++ class. The generated code contains picklers to read and write the graph in a binary format.

Jordan's Modula-3 environment [16] has the ability to make its abstract syntax tree (AST) persistent. The system has a pickle stub generator that generates Modula-3 code to pickle data structures in a type-safe manner. It has a mechanism to deal with external nodes (i.e., nodes that are not part of the AST).

The Slim Binary Format [8] is a target-machine–independent program representation. The format is essentially a compressed abstract syntax tree. Compression of trees is achieved with Semantic-Dictionary Encoding (SDE) [9], which contains template entries for data sequences that occur more than once in the tree.

The Architecture-Neutral Distribution Format (ANDF) [22] is a language-independent and machine-independent distribution format. Part of ANDF is a tree-structured, extensible IR. ANDF has both a textual and a compressed binary format.

5 Persistent Data Structures and Object Stores

One way to make graph structures persistent is to build them in-memory and then to make them persistent by implementing a writer that translates them to an exchange format. Another approach is to simply make the in-memory representation persistent. The transfer of data between an executing program and an external store that preserves type safety is called pickling or marshaling. Unfortunately, automatic picklers are programming language specific. This means that pickling is only useful if data is not exchanged between different languages. Modula-3 has built-in language support to pickle arbitrary graph structures. The implementation is based on run-time type identification and information obtained from the garbage collector. Objective Caml [19] [6] offers similar functionality.

Other useful concepts of persistence are persistent databases, persistent programming languages, and object stores. For example, a persistent graph structure can be easily represented in an object-oriented database system. This field is too vast (and too remote from exchange formats) to discuss it here. Instead, we give an example of a persistent object store specifically for graph structures.

The (P)Graphite tool [31] automatically generates Ada code from a strongly-typed declarative graph description language called GDL. In GDL every node has a certain kind and a node has certain attributes depending on the node kind. Attributes are typed and can have primitive types or represent references to nodes, which makes it possible to connect nodes into directed graph structures. Graphite generates Ada code for graph manipulation, whereas PGraphite additionally incorporates a persistent-object abstraction.

6 Schemas

Schemas impose certain constraints upon otherwise unrestricted structures. To specify the constraints, typically a (declarative) specification language is used. This definition of a schema is fairly general and means, for example, that every data exchange format has a schema because it imposes structure upon a raw sequence of bits (e.g., with the introduction of data types). In this section we focus on schemas that place constraints on otherwise unrestricted graphs. Graph schemas typically type the nodes and edges of a graph (i.e., the graph is "colored"). Graph constraints are then expressed in terms of types (i.e., the graph is "shaped").

The Graph eXchange format (GraX) [7] introduces a graph model called TGraphs. TGraphs are directed and ordered. Nodes and edges are attributable and typed, whereas types specify the possible attributes of nodes and edges. The type system allows multiple inheritance. Schemas are defined through extended entity relationship diagrams that can be annotated with constraints given is a specification language (called GRAL). Thus, the schema denotes a set of corresponding TGraphs that adhere to certain constraints (e.g., valid edge and node types and valid connections of edge and node types).

The TA format contains a textual schema description that has the expressiveness of binary relations (and is thus similar to GraX). TA distinguished between entity types and relation types. Essentially, entities are nodes and relations are edges. The schema is expressed with a triple notation. Triples are used to describe entity types and relationships between these types. Attributes (and default values for these attributes) can be associated with entity and relationship types. Multiple inheritance of types is supported as well. When an entity type inherits from another, the inheriting type gets all the attributes of the super type. Furthermore, any connectivity allowed for the super type is also allowed for the subtype.

Other research areas that are not directly related to exchange formats have also developed schemas to impose restrictions on graph structures. For example, PCTE [4] models software development environments with graphs. PCTE's information repository is based on the binary entity relationship model. The Virtual Tree Processor (VTP) of the Centaur system [3] has a specification language to describe schemas for trees. Finally, the Attributed Graph Specification (AGS) [2] is a declarative formalism to put restrictions on graph structures.

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] Bowen Alpern, Alan Carle, Barry Rosen, Peter Sweeney, and Kenneth Zadeck. Graph attribution as a specification paradigm. *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engi-*

neering Symposium on Practical software development environments, pages 121–129, November 1988.

[3] P. Borras, D. Clement, Th. Despeyroux, J. Incerpi, G. Kahn, B. Lang, and V. Pascual. CENTAUR: the system. *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical software development environments*, pages 14–24, November 1988.

[4] Gerard Boudier, Ferdinando Gallo, Regis Minot, and Ian Thomas. An overview of PCTE and PCTE+. *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical software development environments*, pages 248–257, November 1988.

[5] Ivan T. Bowman, M. W. Godfrey, and Richard C. Holt. Connecting software architecture recovery frameworks. *Proceedings of the First International Symposium on Constructing Software Engineering Tools (CoSET '99)*, May 1999.

[6] Objective Caml. http://caml.inria.fr/ocaml.

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

[8] Michael Franz and Thomas Kistler. Slim binaries. *Communications of the ACM*, 40(12):87–94, December 1996.

[9] Michael Steffen Oliver Franz. *Code-Generation On-the-Fly: A Key to Portable Software*. PhD thesis, Swiss Federal Institute of Technology Zurich, 1994.

[10] Michael Fröhlich and Mattias Werner. The graph visualization system daVinci — a user interface for applications. Technical Report 5/94, Department of Computer Science, University of Bremen, 1994.

[11] Charles F. Goldfarb and Yuri Rubinsky. *The SGML Handbook*. Clarendon Press, 1990.

[12] Michael Himsolt. GraphEd: A graphical platform for the implementation of graph algorithms. In R. Tamassia and I.G. Tollis, editors, *Graph Drawing*, volume 894 of *Lecture Notes in Computer Science*, pages 182–193. Springer Verlag, 1994.

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

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

[15] Graphlet Homepage. http://brahms.fmi.uni-passau.de/Graphlet.

[16] Mick Jordan. An extensible programming environment for Modula-3. *Proceedings of the fourth ACM SIGSOFT symposium on Software development environments*, pages 66–76, December 1990.

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

[18] Elefherios Koutsofios and Stephen North. *Drawing graphs with* dot. AT&T Bell Laboratories, October 1993.

[19] Xavier Leroy. *The Objective Caml System.* INRIA, 1999.

[20] Stephen C. North. *NEATO User's Guide.* AT&T Bell Laboratories, October 1992.

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

[22] Open Systems Foundation. *OSF Architecture-Neutral Distribution Format Rationale*, June 1991.

[23] W3C XML Page. http://www.w3.org/XML.

[24] F. N. Paulish and W. Tichy. Edge: An extensible graph editor. *Software—Practice and Experience*, 20(S1):S1/63–S1/88, June 1990.

[25] Georg Sander. *VCG: Visualization of Compiler Graphs.* Universität des Saarlandes, February 1995.

[26] Tom Sawyer Software. http://www.tomsawyer.com.

[27] R. Srinivasan. XDR: External data representation standard. *Network Working Group, RFC 1832*, August 1995.

[28] Andrew S. Tanenbaum. *Computer Networks.* Prentice Hall, third edition, 1996.

[29] 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.

[30] 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.

[31] 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.

[32] Kenny Wong. *RIGI User's Manual — Version 5.4.4.* Department of Computer Science, University of Victoria, June 1998.