Union Schemas as a Basis for a C++ Extractor

Thomas R. Dean
Queen's University
thomas.dean@ece.queensu.ca

Andrew J. Malton
University of Waterloo
malton@uwaterloo.ca

Ric Holt
University of Waterloo
holt@uwaterloo.ca

Abstract

An extractor is a program which processes source code and outputs facts about the code in a software exchange format (SEF). An SEF can be further specified by a schema, analogous to a schema for a data base. This paper explains how two such schemas can be combined into a union schema as the basis for creating an extractor. We describe the CPPX extractor, which transforms a schema designed as the internals of a compiler (GCC) to a schema designed for software exchange (Datrix). CPPX performs this transformation as a pipelined sequence of sub-transformations. At each stage in the pipeline, the intermediate data conforms to the union of the two schemas.

1. Introduction

In the field of software reverse engineering, it is common to analyze a source program in order to determine certain properties about it. This analysis often begins with parsing the program to determine "facts" about it. These facts are conveniently modeled as the abstract syntax tree (AST) of the source program, with additional edges to indicate semantic information such as typing or the resolution of scoped names. There are several proposed standard *software exchange formats* [1][2], such as GXL [3][4], which standardize the output streams from such parsers. These parsers are called fact extractors or, simply, extractors.

Many legacy programs are written in the C++ language. The Datrix [7] and Columbia [14] schemas propose two different standards for representing facts extracted from C++ programs. Up to this time, there does not exist an open source C++ extractor satisfying such a proposed standard schema. The job of producing such an extractor is far from trivial, and is not rewarding to researchers or developers whose primary interest is in developing or using reverse engineering tools. Even if such an extractor existed, it would require a significant ongoing maintenance effort. The CPPX project, whose present members are the authors of this paper, undertook to create such an extractor.

The goals of this project included:

1. *Open source.* The extractor and the source code of implementation should be freely available.
2. *Minimal resources and early delivery.* We had roughly eight person months to work on this project. Dean and Malton were each available for roughly four months. Holt played a supervisory role. The target delivery date, May 2001, was four months after the initiation of the project.
3. *Minimal maintenance.* The extractor should be designed so maintenance could reasonably be done by a distributed team of unpaid volunteers, in the spirit of the open software community.
4. *Standard Software Exchange Format*. The output should be in the form that meets a visible standard or specification.
5. *Deliver all semantic facts.* The extracted data should sufficient to represent the complete semantics of the source program, down to and including expressions, and thereby suffice for the needs of a broad class of reverse engineering tools, such as tools to compute metrics, to produce visualizations, to detect clones, etc.
6. *Source complete*. It should be possible to recover the source program from the extracted facts. (This goal was significantly compromised.)
7. *Production quality.* The extractor should support commercial work.
8. *Performance.* The extractor should run no slower than a compiler.

The team took a pragmatic, engineering approach, knowing that it would probably not be able to attain all project goals. The overriding goal was to produce an initial, usable version of the extractor by the target delivery date. This led to two early design decisions:

1. The output format would satisfy GXL and the Datrix C++ schema. GXL [3][4] is a standard form of XML, specified by DTD [5], which is designed for exchanging graph data. Bell Canada's Datrix project [6] designed a model [7][8] which specifies a standard AST of a C++

program. We decided to have CPPX emit GXL streams representing C++ ASTs in the Datrix model.

2. Use GCC as a front-end. We decided that CPPX should be based the GNU C++ front-end, since that front-end already does parsing and semantic analysis of C++ programs, and since it already has a team of maintainers. We decided to minimize our changes to GCC and to minimize the size of interface between GCC and our to-be-written code for CPPX.

We needed a basis upon which two developers could proceed rapidly in parallel. We decided to view the existing GCC data structures as simply another graph, analogous to our target Datrix graph. Then we could conceive of CPPX's job as transforming one graph to another one [9][10]. Since we had considerable background in the theory and practice of graph transformations, we were quite comfortable with this approach. We then designed a common graph format to represent both the GCC internal "graph" structure and the Datrix graph.

The key idea of this paper is that a back-end extractor such as CPPX can be designed as a succession of relatively simple and mutually independent graph transformations. For example, "templates" are the C++ facility for polymorphic type construction. One task is to transform the subgraph representing a template in GCC to the corresponding subgraph in the Datrix model.

To do this, we formed a "union" of the two schemas. The union is a generalized schema having all the node types and edges of both the GCC and Datrix schemas. At each stage, after some but not all of the transformations have been performed, the intermediate graph satisfies the union schema. The intermediate graphs preserve "semantics", that is, they continue to be representations of the input C++ program.

This view of the process of a sequence of sub-transformations led to a software architecture as a pipeline, in which each transformer accepts an intermediate graph and produces one. We implemented the transformers as small C programs which operate on a common data structure which represents the evolving graph.

2. Schemas and Their Goals

For any data modeling problem there may be more than one schema to represent the data. Which schema is the most appropriate depends on the task for which the data model will be used. Some schemas are very similar except for the number and types of attributes, but the details of the schemas will differ from task to task.

A programming language is a data modeling problem. Example entity sets in the schema may be variables, procedures, types and statements. The type of a variable, the nesting of a statement within a function and the declaration of a field in a record are all relationships that might appear in the schema.

Programming language models may be used for many purposes. Perhaps the oldest class of such models are those used by compilers. A compiler builds a set of internal data structures (*e.g.* parse tree, symbol table, type table) representing the program being compiled. The name 'compiler' comes from the task of compiling tables.

Other tools such as syntax-based editors [11], code development environments, and revision control systems[12][13] also use models of software source code. Not surprisingly, the data model of each tool is specific to its task. Reverse engineering tools process software artifacts (including source code), and extract a model whose purpose is program comprehension. The use of such models varies from general technology surveys (*e.g.* embedded SQL, windowing systems, transaction environments) to impact analysis (*e.g.* impact of changing the data representation of a record), through design recovery and redocumentation.

The model used by a compiler serves as an intermediary between high level source code and low level machine code, and the schema will be tailored for this purpose. A typical compiler schema includes details of no interest for program comprehension, but necessary for semantic validation and code generation. For example, a compiler will artificially insert expressions to represent implicit type coercions, and may model assignment differently for each of the primitive types. The compiler may also represent information at a finer granularity, choosing to duplicate information throughout the model in order to ease the task of code generation.

Information of interest for program comprehension may not exist in the compiler's model, and may be difficult to infer. For example, the original lexical form of a literal will not be retained, but only its value. For another example, some compilers do not retain source formatting beyond file and line number.

2.1 C++ models

The reverse engineering community has proposed several schemas for C++ SEFs. Two of them are the Datrix Schema [7][8] and the Columbia Schema [14]. Both schemas are general schemas. That is, they are are intended to be used for a broad category of program comprehension tasks and, in particular, serve as an exchange medium between reverse engineering toolsets.

In this section we use a concrete example in C++ to

illustrate some of the differences between a reverse engineering schema and the schema used internally by the GCC compiler.

The Datrix schema is aimed at C-styled programming languages: C, C++ and Java. With some minor modifications, this is the model that we chose to use as the target of the CPPX extractor. Figure 1 shows a simple C++ class declaration which we will use to illustrate the difference between the Datrix schema and the schema used by the GCC compiler.

```
class A {
   private:
      int b;
   public:
      int get() { return b; };
}
```

Figure 1. Simple C++ Class Declaration

Figure 2 shows the Datrix model of the source in Figure 1, as produced by CPPX. The class A is represented by node 2, which has two children (connected by the ArcSon edge) corresponding to the two members of A: the field `b` (node 4) and the method `get` (node 5). Since the members are both integer-valued, their nodes are connected to the built-in type `int` (node 3) by edges of type Instance. The function contains a block, which in turn contains a return statement, which refers to the field `b`. The numbers beside ArcSon edges are the ordinal of the edge and models the syntactic order of elements such as class members or statements within a block.

The example shows that a Datrix graph is structurally close to the source code, and is independent of any particular compiler or target representation. For example, the graph does not represent the size of the C++ type `int` (it might be 16, 32, 64 or some other number of bits). The model is also independent of the implementation of the "this" pointer and the implementation of method dispatching.

Figure 3 shows a small excerpt of the graph produced by the GCC compiler from the same source code. The graph is based on data available from a maintainer's dump facility in the compiler. Some nodes and attributes not needed for the translation to the Datrix schema have been suppressed. Also the figure has been pruned to exclude unused built-in types. Even after pruning, the GCC model for the example program has 87 nodes of 22 different types. For this reason parts of the model have been elided (by "..."). Cross-references are indicated by a small circle containing a node identifier.

Note several compiler-specific details in Figure 3, in which some interesting nodes have been highlighted. The class `A` is represented by node 5 and the field `b` by the node 9. Both have size attributes that are integer literals, which in turn have a type integer_type (name: bit_size_type) given by node 15.

The chain of function_decl nodes from 9 to 148 represent the methods of the structure. The last one in the list, 148 is the user defined function `get`. The first one in the chain (node 9) is the default copy constructor. The methods in the middle (elided) are the default constructors and deconstructors. Only the `get` function has a body (also elided). The rest of the graph represents the signatures of functions that must be generated since the definition is not provided in the example.

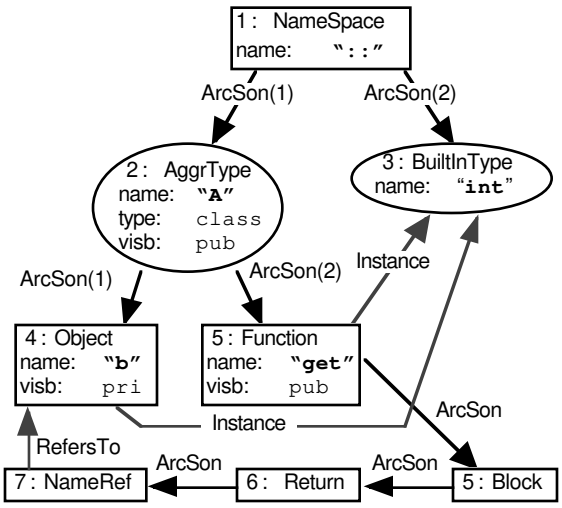

Figure 2. Datrix Model of Example Source Code

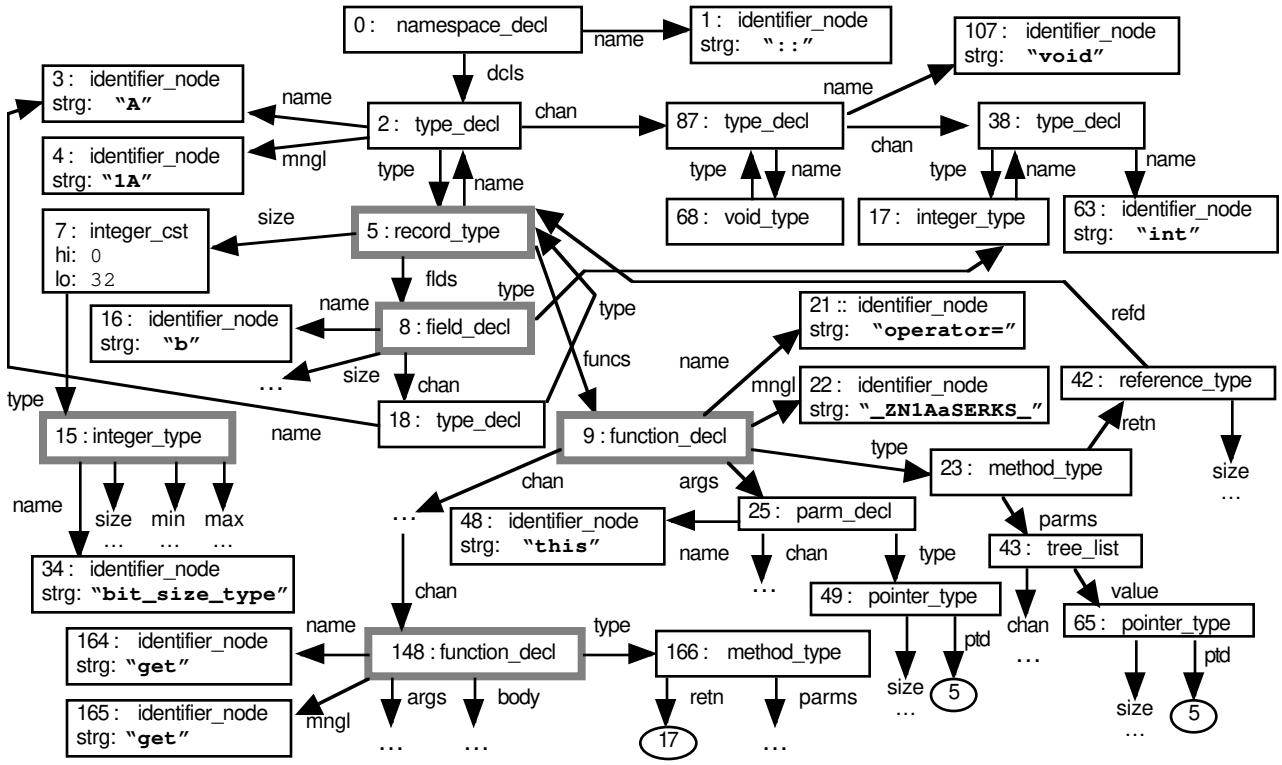

Figure 3. Reduced GCC Model of Example Source Code

3 Transformation Using Union Schemas

This section discusses how CPPX transforms the initial GCC graph to a Datrix graph.

3.1 Schema Transformation in Stages

CPPX's translation task is a *restructuring*: that is, it is supposed to be "semantics preserving" at approximately the same level of abstraction. (The GCC model is at a slightly lower level because of its goal of compilation.)

In general, a "semantics preserving" transformation between schemas is a function which maps (instances of) one schema to (corresponding instances of) the other schema, such that the "semantics" is held fixed.

The semantics is held fixed in two senses. First, a graph schema for representing facts about a program defines a "view" of the program, in which facts of interest are encoded and some uninteresting facts are ignored or abstracted away. If P is the set of programs and S the set of graphs, the view may be considered as an abstraction function c : P -> S. In our case there are two schemas $S_{initial}$ and S_{final}, and correspondingly two views $c_{initial}$ and c_{final}, The required "semantics preserving transformation" ios a function $t : S_{initial} \rightarrow S_{final}$ for which $c_{final}(p) = t(c_{initial}(p))$ for any program p. This just says that the initial view, transformed, must be the final view.

The second sense of "fixed semantics", when transforming program schemas, comes from the source completeness goal mentioned in the Introduction. From any graph of a program we must be able to reconstruct source, if not perfectly, then at least to the extent that the reconstructed program has the same (operational) semantics.

The transformation t is quite complicated to describe all at once, and we have found it useful to factor it into stages t_i, pipelined together, such that $t = t_{n-1} \circ \ldots \circ t_1 \circ t_0$. Doing so suggests that we need (at least conceptually) many different intermediate schemas, one for each stage. In practice we form instead a *union schema* : a single schema S capable of representing the views $c_{final}(p)$, $c_{initial}(p)$, and and required intermediate transformations as well. Figure 4 shows the pipeline and requirements for the intermediate transformations.

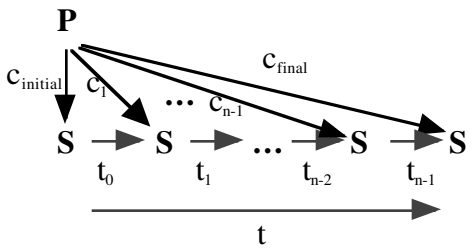

Figure 4. Union Schema Pipeline

3.2 Constructing the Union Schema

In constructing the union schema we have adapted a technique [15] familiar to us from programming in the transformational programming language TXL [10].

The purpose of the union schema is to support the initial and final encoding, and to allow (and to a certain extent direct) the construction of intermediate encodings, so that the individual steps t_i can be transformations from S to S determined by $c_i = t_i \circ c_{i-1}$. The first step in constructing the union schema is to cast the initial and final schemas into a common conceptual model. Next, corresponding structures in each are identified, and the resulting union schema is implemented as a data structure (for CPPX, in C). Lastly, translation functions and exchange-format programs are written to apply to instances of the union data structure. Similar steps were originally identified by Cordy for the closely related "union grammar" technique [15]. We now examine each of the steps in more detail.

3.2.1 Choose a common conceptual model. As described above, for CPPX we had to do with:

1. The initial model, of GCC, implemented as C structures and pointers, extended by wonderfully intricate uses of the C macro facility, and documented in English, and
2. the final model, the "Datrix" model, described by a technical report [7], and partly formalized in TA [16].

The conceptual model of the GCC schema requires entities (represented as `structs`) with entity attributes (`struct` members), direct references to other entities (pointers), and union types. Union types are implemented both by systematic use of union types in C, and also in other cases by means of the macro facility and naming conventions. The use of pointers implies that relationships are all cardinality "zero-or-one": multi-occurring relationships are encoded in the GCC model using various kinds of linked lists. Figure 5 shows that part of the GCC data structuring scheme which justifies the graph in Figure 3. In this diagram, from which much has been omitted, white-headed arrows indicate union types (the arrow points *from* the union *to* the member), black-headed arrow indicate pointers, and italic names indicate simple attributes, nesting indicates structural containment. Because this is all implemented in C, necessarily the schema has been *inferred*, by hand, from the code.

The conceptual model of the Datrix schema is that of TA [16], requiring typed entities (represented as labeled nodes in a graph) with simple-valued attributes, and typed binary relationships (represented as labeled edges in a graph), also allowed to have attributes. Relationships can be constrained with respect to the type (label) of their domain and codomain (source and target). (In the informal model, but not in the schema, there are also cardinality constraints, *i.e.* specifications of the *allowed number* of instances of a relationship involving a particular entity.) Figure 6 shows that part of the Datrix schema which justifies the graph in Figure 2. In Figure 6 white-headed arrows indicate inheritance (the arrow points at the superclass) and black-headed arrows indicate possibly-attributed relationships. Our Datrix schema was transcribed almost directly from its highly-structured documentation [7][8].

3.2.2 Identify corresponding structures. It is apparent that the two conceptual models are similar in spirit but by no means identical. For our purposes it proved practical to use the TA concepts extended by attribute values which are direct references to other entities. Thus in the union model there are typically two ways to represent a relationship: either with an explicit TA-style edge, or with a reference-valued attribute.

The task of identifying corresponding structures is fairly straightforward, but is delicate because it governs the whole transformation process which follows. Essentially the idea is to represent partial or incremental results of the whole transformation by allowing relationships in both "sides" of the union to involve entities from either "side".

For simple examples: the GCC entity function_decl corresponds to the Datrix entity Function; the GCC entity field_decl corresponds to the Datrix entity Object. In these and similar cases the correspondence holds simply because they are representing the same syntactic structure in the source language C++.

For more complex examples of correspondence, consider the relationship chan (="chain") in the GCC schema, which serves as a general-purpose link between tree nodes. With this relationship, any other relationship can be made multi-occurring by chaining multiple occurrences together. A typical example is the

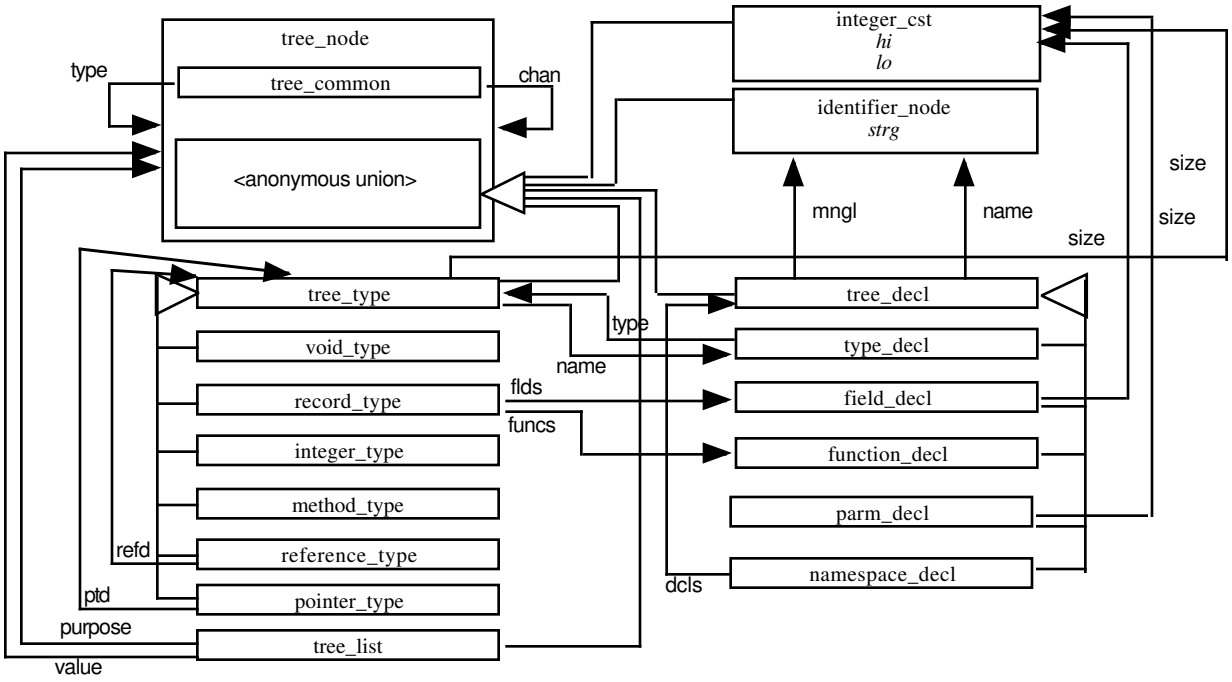

Figure 5. Partial GCC Schema

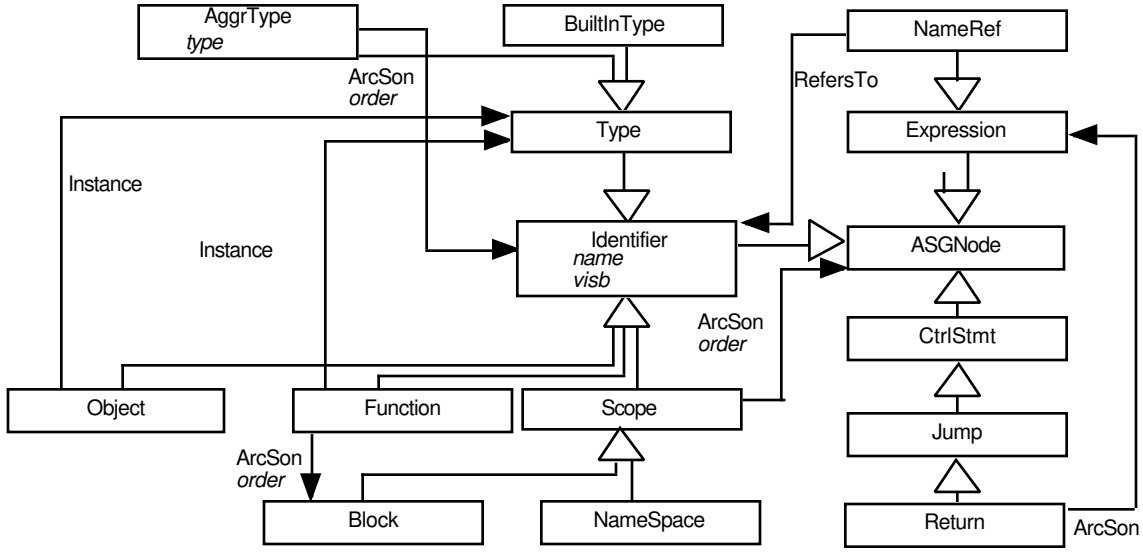

Figure 6. Partial Datrix Schema

relationship between an aggregate or record type and its members. In the GCC schema this is indicated by pointers “flds” and “funcs” from a record_type to the (first) field_decl and to the (first) function_decl, which then are linked to the remaining similar members by “chan” pointers. This is convenient for the goal of the GCC schema, namely processing during compilation. By contrast, in the Datrix schema there is an ArcSon relationship from AggrType to Identifier: and the ArcSon relationship has an order attribute indicating sequence within the aggregate. This is in keeping with the goal of the Datrix schema, which is to show a structural view

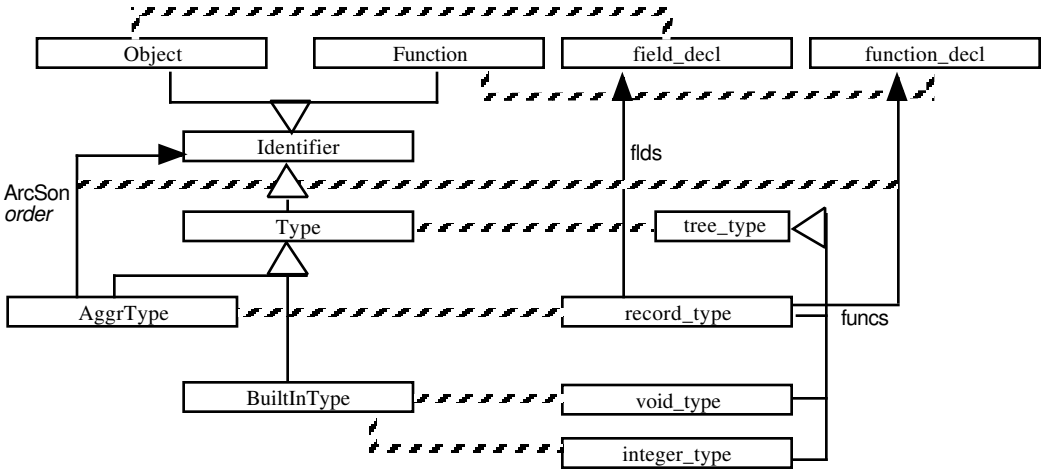

Figure 7. Partial Union Schema

suitable for high level understanding.

Figure 7 shows a portion of the union schema, in which corresponding entity types and edge types are indicated by thick lines. The meaning of the combined schema is to relax the type constraints so that entities and relationships can play a role specified for their corresponding entities or relationships.

Figure 8 shows a portion of the example "in the middle" of transformation. Here it is shown how the field b of the class A has been converted to the Datrix schema, and the corresponding edges have been converted from "flds" and "funcs" to "ArcSon". The class A itself and the function members are still in the GCC form. The root of the graph (the global namespace) has also been converted to the Datrix schema as have the built in types (elliptical nodes 68 and 17) and the edges connecting them. This example therefore satisfies the union schema, but neither the initial schema nor the final schema.

3.2.3 Build translation and output functions. We discovered that there were three kinds of transformation step for instances of the union schema:

1. simple replacement transforms
2. construction of parallel corresponding structures
3. filtering (garbage collection)

During transformation of a particular correspondence, we found that in simple cases (*e.g.* Function = function_decl) it is convenient to "replace" a node entry of GCC type with the corresponding entry of Datrix type, reusing the node directly. The advantage of doing so was that all references to that entity elsewhere in the graph would continue correct in the union schema, for they would continue to refer to the corresponding structure.

The disadvantage of this replacement approach was that it requires all the information in a node to be "converted" at once, since it is otherwise lost when the node is replaced. For more complex situations where it was not practical to do this, we created a parallel structure instead. A typical example was the list of actual parameters of a call, which are linked together using "tree_nodes" in the GCC model, but which must be formed into a subgraph called an "actual function parameter list" in the Datrix model. In the process the GCC nodes would be detached from the root of the tree, and the graph would be filtered afterwards, to remove nodes from the graph which are no longer accessible from the root of the compilation unit.

4. The CPPX Implementation

The project ran for four months in the beginning of 2001. Of this, the first month was spent planning the transformation, modifying the GCC compiler dumping routines and implementing the final output routines. The remaining two and a half months were spent building and testing the transformation routines.

The project has the following components:

1. A header file defining the binary union schema format and utilities to read and write files containing individual models.
2. Two replacement graph dumping files for the GCC compiler and changes to the makefile for the compiler.
3. Two filters used for initial input conditioning and garbage collection.
4. A number of individual transforms that change the GCC schema to the Datrix schema.
5. Output generators that translate the binary

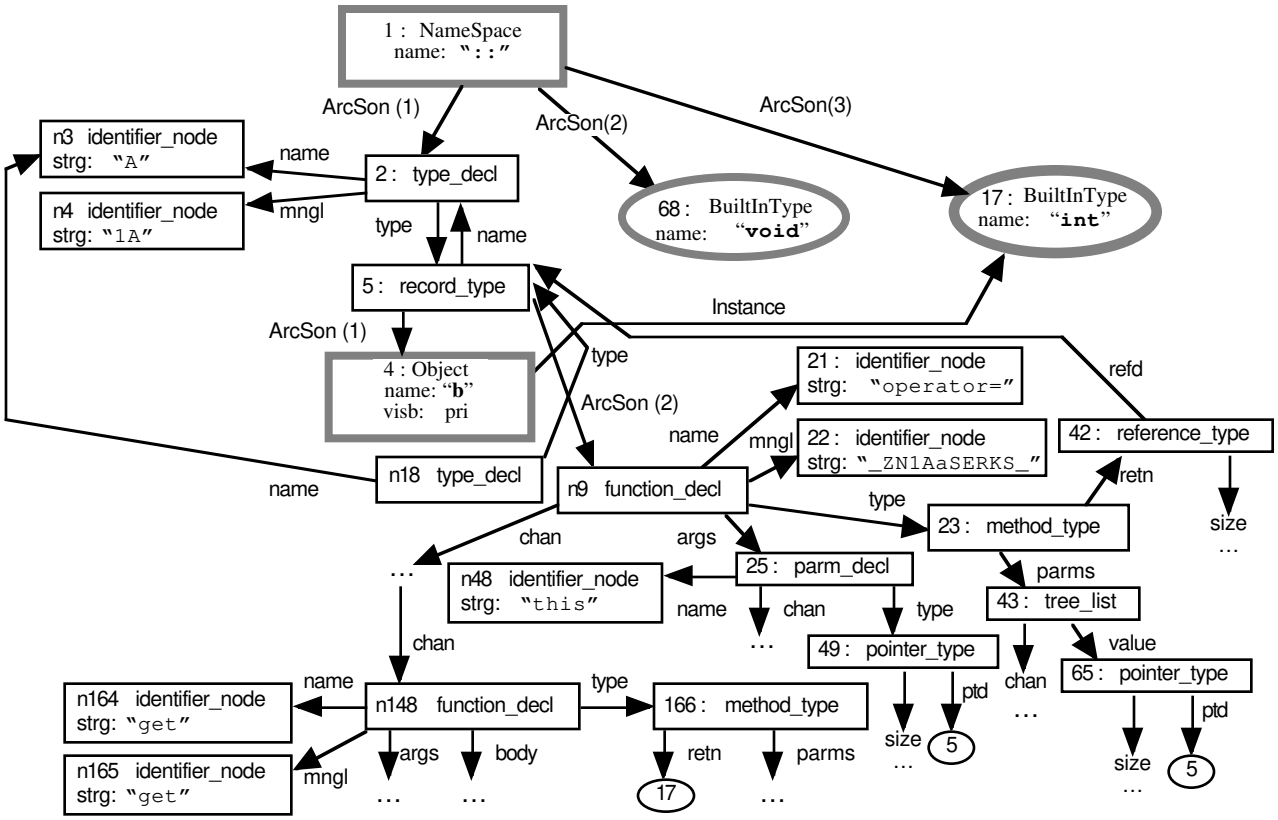

Figure 8. Partial Result in Union Schema

representation to one of three text representations: TA [16], VCG [17] and GXL.

6. Utility functions used by all of the above.
7. A query program used to examine individual elements of the binary files.
8. A driver program that coordinates most of the above elements.

The output generators can output a TA, VCG or GXL version of the final graph (in the Datrix schema) or the output of any of the intermediate results (in the union schema). Thus the CPPX generators can be used to produce interchange formats that allow tools to operate on graphs expressed in the GCC.

GXL output from the transformer has been successfully checked against the GXL DTD definition using rxp [18]. We have checked that the output of the transformer matches the modified Datrix Schema using a schema checker written in grok[19].

6. Conclusions

During the last roughly ten years, a number of efforts have produced reverse engineering systems for languages such as C++. Among the best known of these are CIAO [20], Refine [21], Rigi [2], TXL [10][22], PBX [23] and RMTool [24]. As well, large commercial organizations such as IBM, Sun and Microsoft are increasing providing open interfaces to, or open source for their compilers and their IDEs. There are a number of ongoing efforts to provide standard APIs or standard SEFs for such tools; see the WoSEF for a survey of many of these efforts [25].

In our introduction we listed 8 goals. We examine at each individually to see how close we came.

1. *Open source*. The source code can be freely downloaded from `swag.uwaterloo.ca/~cppx`.
2. *Minimal resources and early delivery*. The first version of CPPX was available from mid-May 2001.
3. *Minimal maintenance*. The multiple transform approach permitted the two developers to work quite independently. This same architecture will permit on-going distributed maintenance.
4. *Standard Software Exchange Format*. The output of the extractor is in a slightly modified version of the the Datrix model encoded in GXL. This permits other program comprehension tools to process C++ code without having to provide their own extractors.

5. *Deliver all semantic facts*. The first version omits a small set of features, but there is no obstacle to including them over the following couple of months.
6. *Source complete*. Most of the source program (after preprocessing) can be recovered from the CPPX output. However, some source features are absent from the GCC schema (e.g. lexical representation of literals), and some are not represented in the Datrix schema (e.g. preprocessor directives).
7. *Production quality*. Parsing and semantic analysis is done by the GCC compiler, and hence CPPX accepts whatever GCC accepts. Since GCC is used for production development, this goal is met to the same extent. However, there are dialects of C++ which GCC doesn't compile. Furthermore, dependence on the internal graph of GCC has meant extra maintenance as new versions of GCC were released.
8. *Performance*. Initial measurements suggest that CPPX runs at the same speed, or somewhat slower than, the production compiler.

Transformations on graphs are common in software tools. For example, graph transforms may be used to raise the level of source facts to an architectural level[7]. In many cases, translation between schemas is one goal of the transform. Thus the union schema approach has a great deal of potential for general use.

Acknowledgements.

The Consortium for Software Engineering Research (CSER), which provided funding for much of the project.

References

[1] Ivan T. Bowman, Michael W. Godfrey, Ric Holt, "Connecting Architecture Reconstruction Frameworks", Journal of Information and Software Technology, vol. 42, no.2, pp. 93–104, 1999

[2] Hausi Müller, Mehmet A. Orgun, Scott R. Tilley, James S. Uhl, "A Reverse Engineering Approach to Subsystem Structure Identification", Journal of Software Maintenance: Research and Practice, 5(4), pages 181-204, December 1993.

[3] GXL Home page, http://www.gnupro.de/GXL/

[4] Ric Holt, Andreas Winter, Andy Schür, "GXL: Towards a Standard Exchange Format", Proceedings WCRE 2000 Working Conference on Reverse Engineering, Brisbane, Australia, November 2000, pp. 162–171.

[5] GXL DTD available at http://www.gnupro.de/GXL.

[6] Bell Canada DATRIX™ Home Page, www.iro.umontreal.ca/labs/gelo/Datrix

[7] Bell Canada, DATRIX™ Abstract Semantic Graph: Reference Manual, version 1.4", Bell Canada Inc., Montréal, Canada, May 01, 2000. http://www.casi.polymtl.ca/casibell/Datrix/refmanuals/asgmodel-1.4.pdf

[8] A. Hassan, R. Holt, B. Laguë, S. Lapierre, C. Leduc, "E/R Schema for the Datrix C/C++/Java Exchange Format", WCRE 2000: Working Conference on Reverse Engineering, Brisbane, Australia, Nov 6, 2000.

[9] H. Fahmy, R.C. Holt, "Using Graph Rewriting to Specify Software Architectural Transformations", *ASE 2000 Proceedings of Automated Software Engineering*, Grenoble, France, Sept 2000.

[10] J.R. Cordy, I.H. Carmichael, R. Halliday, "The TXL Programming Language – Version 10", Legasys Corporation, Kingston, Canada, January 2000, 65 pp.

[11] T. W. Reps, T. Teitelbaum, *The Synthesizer Generator*, Springer Verlag, 1989, pp. 317.

[12] L. Cooprider, *The Representation of Families Software Systems*, Ph. D. Thesis, Carnegie–Mellon University, Computer Science Department, 1979

[13] W. Tichy. *Software Development Control Based on System Structure Description*, Ph.D. thesis, Carnegie–Mellon University, Computer Science Department, 1980.

[14] R. Ferenc, S. E. Sim, R. C. Holt, R. Koschke, T. Gyimothy, "Towards a Standard Schema for C/C++", 8th Working Conference on Reverse Engineering, Stuttgart, October, 2001.

[15] Cordy, J, R, "Cross Language Transformations in TXL", TXL Working Paper 4, 1995.

[16] Ric Holt, "An Introduction to TA", University of Toronto, Toronto, March 1997, available from: http://www-turing.cs.toronto.edu/pbs/papers/ta.html

[17] G. Sander, "Graph Layout Through the VCG Tool," *Graph Drawing, DIMACS International Workshop GD '94 Proceedings*, R. Tamassia and G. Tollis, editors, Lecture Notes in Computer Science 894, Springer Verlag, pages 194 – 205, 1995.

[18] RXP - an XMLParser available under the GPL, http://www.cogsci.ed.ac.uk/~richard/rxp.html

[19] Ric Holt, "Structural Manipulation of Software Architecture using Tarski Relational Algebra," Working Conference on Reverse Engineering, Honolulu, October, 1998

[20] Judith Grass and Yih-Farn Chen, The C++ Information Abstractor", *The Second USENIX C++ Conference*, April 1990.

[21] Reasoning Systems Inc., *Refine User's Guide*, 1992

[22] J.R. Cordy, C.D. Halpern and E. Promislow, "TXL: A Rapid Prototyping System for Programming Language Dialects", Computer Langauges, V16, N1, January 1991, pp. 97-107.

[23] Ric Holt, *Software Bookshelf: Overview and Construction*, Available at http://www-turing.cs.toronto.edu/pbs/paper/bsbuild.html

[24] G. Murphy, D. Notkin, K. Sullivan, "Software Reflexion Models: Bridging the Gap Between Source and High-level Models", Proceedings of SIGSOFT '95, pages 18-28, October 1995.

[25] WoSEF Home Page, http://www.cs.toronto.edu/~simsuz/wosef/