The SoftBench Static Analysis Database

The static analysis database supports the SoftBench static analyzer and the C++, C, FORTRAN, Pascal, and Ada programming languages. The underlying data is isolated by a compiler interface and a tool interface.

by Robert C. Bethke

The SoftBench static analysis database, Static.sadb, is a repository for generic program semantic information. Within SoftBench the database supports the static analyzer along with graphical editing and rule-based program checking. The data model is relatively general and currently supports C++, C, FORTRAN, Pascal, and Ada.

The database also serves as a product and can be customized by the user. Its compiler interface and tool interface are documented and allow the integration of other languages and compilers and the use of custom analysis tools.

The Data Model

The underlying data is a set of persistent C++ objects. These objects serve to model the semantics of the program. The underlying persistent objects are isolated by the compiler interface and the tool interface. The isolation has important implications for allowing a variety of compiler integrations and provides flexibility in changing the underlying data management without affecting either the compilers or the tools.

Many of the persistent objects are language-generic (language-insensitive) and are intended to model all similar constructs. For example, a Struct object is used to model C structures and Pascal records. A Function object is used to model functions and procedures in all languages. In some cases, it is necessary to have language-specific objects because the semantics are too specific to apply to other languages. Examples of language-specific objects are C++ Class objects and Ada Module objects.

Each persistent object is assigned a unique object identifier known as a *handle*. Given an object's handle, it is possible to query the object by means of *methods* for relevant information such as its name, list of references, and so on. All associations among the persistent objects are maintained by these handles. For example, the association from a Variable object to its typedef object is maintained by the Variable's having the handle of its typedef as an attribute. One-to-many associations are maintained as a set of handles. For example, a File object will have a set of handles to associate all other source files included by it.

To illustrate associations, consider the following C code:

```
typedef struct S {int x; int y;} SType;
Stype var;
```

The associations among the semantic objects in this code fragment are shown in Fig. 1.

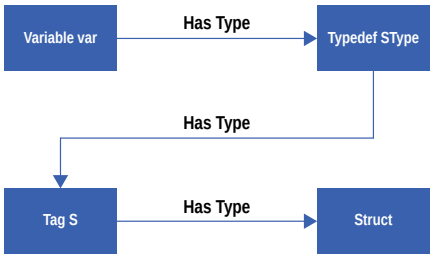

Fig. 1. *Associations among semantic objects for the C code example given in the text.*

Container objects are used to model scoping and binding and to organize the semantic objects for efficient updating and navigation. Each container has a set of handles for all objects contained in it and each object contained has the handle of its container. Examples of container objects are Files, Functions, and Classes. A File contains the program constructs defined in that source file, a Function contains its parameters and blocks, and a Class contains its members. For example, Fig. 2 shows the object containment for the following C++ class definition:

```
Class cls {
   public:
      cls (int x) {mem=x;}
   private:
      int mem;
};
```

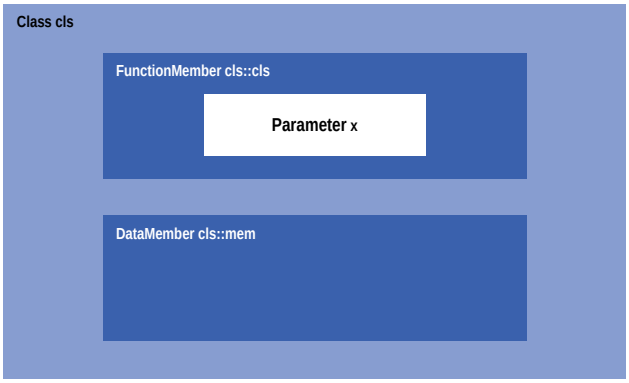

Fig. 2. *Object containment for an example C++ class definition.*

The Semantic Objects

The following is a partial list of the semantic objects stored in the database.

SymbolTable. The global SymbolTable is a container that serves as the root of navigation in the database. Its entries are all globally scoped semantic objects and Files in the database. There is only one global SymbolTable per database.

File. A File is a container that contains all semantic objects that are defined in a specific source file. Attributes of a File are its name, a language kind, and a set of *include* and *included by* associations with other Files.

Module. A Module is a container that contains all semantic objects that are defined within an Ada module. A Module must be contained within a File or within another Module. Attributes of a Module are its name and a set of imported associations with other Modules.

RefList. A RefList is an array of references that are associated with named objects in the database. Attributes of a RefList are the corresponding referent (the File in which the references originate) and the number of references in the list.

Macro. A Macro is a language-generic object for representing a preprocessor or language macro. Attributes of a Macro are its name and a set of RefLists.

Identifier. An Identifier is a language-generic object for representing a named symbol. This object is mostly used by weaker (scan-based) parsers that do not intend to distinguish certain categories of objects. Attributes of an Identifier are its name and a set of RefLists.

Label. A Label is a language-generic object for representing statement labels. Attributes of a Label are its name, an enclosing Block or Module, and a set of RefLists.

Variable. A Variable is a language-generic object for representing variables. Attributes of a Variable are its name and type, an enclosing Block or File, and a set of RefLists.

Function. A Function is a language-generic object for representing functions and procedures. Attributes of a Function are its name, a return type, a set of Parameters, an outer Block, a container (the enclosing File, Module, or Block), and a set of RefLists.

Parameter. A Parameter is a language-generic object for representing function parameters. Attributes of a Parameter are its name and type, an enclosing Function, and a set of RefLists.

Block. A Block is a container for representing a function block. Attributes of a Block are its begin and end line numbers, the File in which it is contained, and an enclosing Block or Function.

Typedef. A Typedef is a language-generic object for representing named program types. Attributes of a Typedef are its name, the type it denotes, an enclosing File or Block, and a set of RefLists.

Tag. A Tag is a language-generic object for representing aggregate (Enum, Struct, Class, and ClassTemplate) type names. Attributes of a Tag are its name, the aggregate it denotes, an enclosing File or Block, and a set of RefLists.

Enum. An Enum is a language-generic object for representing enumerated types. Attributes of an Enum are its corresponding Tag and a set of EnumMembers. RefLists to the enumeration are on the corresponding Tag.

EnumMember. An EnumMember is a language-generic object for representing enumeration constants. Attributes of an EnumMember are its name, an enclosing Enum, an ordinal value, and a set of RefLists.

Struct. A Struct is a language-generic object for representing program structures, records, and unions. Attributes of a Struct are its corresponding Tag and a set of DataMembers. RefLists to the Struct are on the corresponding Tag.

DataMember. A DataMember is a language-generic object for representing fields of a structure, union, class, or record. Attributes of a DataMember are its name and type, an enclosing Struct or Class, and a set of RefLists.

Class. A Class is a C++-specific object for representing C++ classes. Attributes of a Class are its corresponding Tag, a set of DataMembers, a set of FunctionMembers, a set of base and derived Classes, a set of friend Classes and friend Functions, a set of nested Classes within, and the ClassTemplate of which it is an instance. RefLists to the Class are on the corresponding Tag.

FunctionMember. A FunctionMember is a C++-specific object for representing C++ class member functions. Attributes of a FunctionMember are its name, a return type, a set of Parameters, an enclosing Class, the File in which it is defined, an outer Block, and a set of RefLists.

ClassTemplate. A ClassTemplate is a C++-specific object for representing class templates. Attributes of a ClassTemplate are its corresponding Tag, a set of DataMembers, a set of FunctionMembers, a set of FunctionTemplate members, a set of TemplateArguments, a set of base and derived Classes and ClassTemplates, a set of friends, and a set of Class instances. RefLists to the ClassTemplate are on the corresponding Tag.

FunctionTemplate. A FunctionTemplate is a C++-specific object for representing function templates. Attributes of a FunctionTemplate are its name, a set of TemplateArguments, and a set of Function or FunctionMember instances.

The Compiler Interface

From the compiler perspective the database can be thought of as a persistent symbol table for a set of source files such as a library or an application. The compiler sees the contents of only one compilation unit and emits information accordingly, but the database creates only objects that are not yet in the database. The database creates and merges all the program objects as the source files are compiled.

Compilation may result in objects being removed. Persistent objects are removed when they are old or are contained in objects that are old. For example, when a file has been modified and is being recompiled, the File is old and its contents are removed from the database. The compilation will proceed and instantiate the appropriate new objects contained in the File.

The database is incremental to the file level. If one source file in an application or library is changed, the compilation will result in the removal and repopulation of objects in that File. After the compilation the database is again consistent and available for queries from a reader.

The compiler interface is procedural in style and is intended to be easily added to most compilers. The interface is structured around the creation of objects and the establishment of associations and containment relationships among the objects.

The Tool Interface

From the tool perspective the database supports concurrency control to the extent of allowing multiple readers and one writer. A reader can have up to 256 databases open for reading. The reader must structure queries within a transaction and is allowed to leave the database open while it is being modified by a writer. The reader is notified of a change to the database via a callback when starting a transaction. Nested transactions are not supported.

The tool interface is a class library that reflects the underlying object model. Each persistent object is presented as a handle. Internally, each handle is mapped into a pointer to the real persistent object. All information pertaining to the object is made available via methods. Navigation among objects is supported by methods that return a handle or an iterator over a set of handles. For example, the following is a partial definition of the Symbol class.

```
class Symbol {
  public:
    Symbol(PerHandle symbolhandle);
    Symbol();
    ~Symbol();

    // Name, kind and attributes of the symbol.
    char *Name() const;
    PerKind Kind() const;
    Attributes Attrib() const;
```

```
    // Enclosing scopes of the symbol.
    DBboolean EnclosingFile(File &file) const;
    DBboolean EnclosingBlock(Block &block)
      const;

    // Iterator to all reference lists for this
    // Symbol.
    ITERATOR(RefList) RefLists() const;

  protected:
    PerHandle SymbolHandle;
};
```

The global SymbolTable is the root for all navigation. This object provides navigation and hashed searching to all globally scoped symbols. The following code segment illustrates how to access all globally scoped functions from the global SymbolTable.

```
SymbolTable symtab;
// Construct an iterator over all global
// functions.
ITERATOR(Function) functionitr =
   symtab.GlobalFunctions();
// For each function print its name and if the
// function is defined, the file in which it is
// defined.
ITERATE_BEGIN(functionitr) {
    File sourcefile;
    printf("%s", functionitr.Name());
    if (functionitr.EnclosingFile(sourcefile))
        printf(" contained in %s",
              sourcefile.Name());
    printf("\n");
} ITERATE_END(functionitr)
```

All of the relationships among the semantic objects are first-level. Hence, many of the interesting queries and rules will require a transitive closure* of the relationships. For example, consider the following function, which prints all the derived classes of a given class.

```
void derivedclasses(Class theclass) {
    // Iterate over immediate derived classes of
    // theclass.
    ATTRIBUTE_ITERATOR(Tag) tagaitr =
              cls.DerivedClasses();
    ITERATE_BEGIN(tagaitr) {
       // Print the class name.
       printf("%s\n", tagaitr.Name());
       Class dercls;
       // Navigate to the actual derived class
       // and recursively call derivedclasses to
       // print its derived classes.
       if (tagaitr.ClassType(dercls))
           derivedclasses(dercls);
    } ITERATE_END(tagaitr)
}
```

API Products

The database APIs (application programming interfaces) are available in the SoftBench 4.0 product and are used internally by the SoftBench parsers and tools. They are also used by some customers for compiler integrations. The tool interface is the fundamental component of the software developer's toolkit for user-defined rules.

* The transitive closure for a particular object under a particular transitive binary relationship is the set of objects descended from the particular object by way of the particular relationship. For example, if B is derived from A and C is derived from B, the transitive closure for the object A under the relationship "derived from" is the set of objects whose elements are B and C.

▶ Go to Next Article
▶ Go to Journal Home Page