The Architecture of Montana: An Open and Extensible Programming Environment with an Incremental C++ Compiler

Michael Karasick
Programming Environments and Compilation
IBM T.J. Watson Research Center
PO 704, Yorktown Heights NY, 10598

msk@us.ibm.com

ABSTRACT

Montana is an open, extensible integrated programming environment for C++ that supports incremental compilation and linking, a persistent code cache called a CodeStore, and a set of programming interfaces to the CodeStore for tool writers. CodeStore serves as a central source of information for compiling, browsing, and debugging. CodeStore contains information about both the static and dynamic structure of the compiled program. This information spans files, macros, declarations, function bodies, templates and their instantiations, program fragment dependencies, linker relocation information, and debugging information.

Montana allows the compilation process to be extended and modified [11]. Montana has been used as the basis of a number of tools [1,7], and is also used as the infrastructure of a production compiler, IBM's Visual Age C++ 4.0 [8].

Keywords

programming environments, compilation, frameworks, extensible systems, incremental development environments, incremental compilation, C++,

1. INTRODUCTION

In order to program with a computer language effectively, it is often necessary to use extra-lingual tools that either extend the build process (*e.g.*, make, yacc, cpp) or extract static and dynamic information from the program. Examples of those that extend the build process are automated testing tools, stub generators, metadata catalog processors, GUI builders, and program transformers. These kinds of tools either transform the source language (in our case, C++) or use scripting languages specific to the individual tool. Tool invocation is typically incorporated into the build process with coarse-grained facilities, like make. Montana is designed and implemented to address the tool integration problem in three ways:

Open: Tools often need to access information about the static or dynamic structure of a compiled program. For example, analysis tools might need control-flow information or structural information about objects. Without this kind of information, tool writers need to reimplement parts of the compiler (a daunting prospect for C++), or modify the compiler itself to obtain the needed information. Both are difficult, and not possible in many cases. Both are also maintenance nightmares. Montana provides a set of programming interfaces that give access to the program at different levels of abstraction, from source files and macros, to abstract syntax trees for function bodies, to object-code relocations.

Extensible: Even though a program representation might be available, it is often necessary, or desirable to gather information during the compilation process, or modify the compilation process. It is unlikely that compiler writers can predict the different kinds of information that a tool writer might need. Furthermore, some tools, like style checkers and stub generators and naturally invoked during the compilation itself. Montana supplies a set of extension interfaces that allow tool writers to get fragments of their tools inserted directly into the compilation stream, without modification of the compiler itself [11].

Incremental: Montana updates its program representation in response to source code changes. This update is incremental, and so results in better performance for the compiler and tools. For example, an automatic testing tool need only test for function bodies that have changed since the last compilation.

The way in which a system like Montana is architected directly affects how well these three goals are achieved. One example of this is the tradeoff between performance and extensibility. Because Montana serves as an incremental C++ compiler as well as a tool framework, it is important that the "inner loop" run efficiently. This means that any extension mechanism which affects this inner loop must be carefully crafted. There is also a tradeoff between performance and openness. Providing programming

SIGSOFT '98 11/98 Florida, USA

interfaces that access different aspects of the compiled program can cause performance problems, because these interfaces might not be the most convenient or efficient for a compiler. To ameliorate this Montana separates data from function. There are no "privileged tools," including the compiler. Tools use the programming interfaces to extract data and construct appropriate data structures for a given task. These data structures are either hidden or exposed by the tool, as necessary.

The lack of a system like Montana means that tools must either provide their own parser and semantic analyzers, or must be inherently imprecise by approximating syntactic or semantic information. There are several offerings currently in the marketplace that attempt to provide parsing and analysis tools. Sniff [12] uses approximate parsing and serves as a reasonably powerful cross-referencer, but is not precise enough for tools like style checkers, program transformers, or code wizards. Edison Design Group [3] has written an a C++ front end that is designed to be used as the basis of tools. It is quite ambitious, with a plethora of options that implement "bugs" in versions of different vendor compilers. This front end suffers from the problem that tools can only manipulate information that is obtained from their abstract syntax trees. The ubiquitous GNU C++ compiler has been lovingly tailored by legions of graduate students in order to extract information of interest. Unfortunately, the language recognized by this compiler bears only a passing resemblance to the C++ recognized by most other C++ compilers. Consequently, the GNU compiler servers more as a research vehicle than the basis of any standardized tool suite. Ensemble [2] is a vendor-neutral tool for analyzing C programs. It provides programming representations for C program fragments, along with a number of tools that use these representations. ASIS [13] is a vendor-neutral program representation for Ada that allows access to information about compiled Ada programs. The Synthesizer Generator [10] provides ways to specify the syntax and semantics of target languages and then "build" an environment from that specification. Gandalf [5] additionally allows these specifications to describe how externally written programs (e.g., configuration management systems) can be invoked as part of the compilation. All of the systems described above are open, in that they allow access to program information.

Montana is most similar to either EDG or ASIS, but is more ambitious in several ways. ASIS and EDG provide non-extensible programming interfaces to data structures like Abstract Syntax Trees, so the available information is limited. In contrast, Montana provides similar information, but additionally provides ways in which tool writers can augment the information stored away (data extension), can add additional processing phases to the compilation process (tool extension), and provide event handlers for distinguished events during the compilation process (event extension).

Montana works by constructing and modifying a program representation, called CodeStore, stored in a persistent or transient cache. The programming interfaces to CodeStore are open, and available to tool writers. In order to access the program representation the tool writer writes a small C++ program fragment that constructs and then queries (or modifies) the CodeStore. It is also possible to extend the compilation process itself, by registering code fragments that are triggered during the compilation. These fragments can collect information, generate error messages, or modify the CodeStore directly [11]. As an example of this, consider a style checker might be as an extension to Montana: (1) the error messages produced are integrated with those produced by the compilation itself; (2) the style checker is run automatically as part of the compilation process, so the abstract syntax trees used by the compiler itself are available for diagnosis; (3) because the compilation is incremental, only program fragments that have changed are diagnosed, making the style checker inexpensive to use; and (4) the style checker written in this way can be extended and customized to check for those idioms of interest. The Montana architecture is flexible enough to allow any number of style checkers, program analyzers, and transformers to be registered as extensions.

Like any system that works with a persistent program representation, Montana is necessarily incremental. The compilation process (which we call *incorporation*), works by taking changed source files, and "incorporating" changes into the CodeStore. A side effect of this process is an incrementally compiled and linked program, if desired.

The remainder of this paper describes the architecture of Montana. Aspects of the architecture discussed are the compilation model, and the design of the CodeStore and its programming interfaces. (See [11] for more details of the CodeStore extension mechanism.) This paper focuses on the engineering tradeoffs inherent in an open, extensible system like Montana.

2. THE COMPILATION MODEL

In order that the compiler be incremental, the compilation model is quite different from that of traditional compilers. Just like conventional compilers, program text is stored in files, and the CodeStore is updated to reflect changes in the source files. However, the grain of incrementality is much smaller -- top-level declarations instead of files. This

provides for a much more efficient compilation, but has some consequences.

The compilation model is not that different from that used by program construction tools like Make [4]. In particular, the only kind of dependency that makefiles capture is a source/target dependency, *e.g.*, reprocess a source file (.cpp) if it, or header-files or libraries on which it depends change. The simplest of these kinds of dependencies are often created automatically. For example, Microsoft Visual C++ constructs stylized makefiles for predefined kinds of applications. These makefiles are then automatically maintained by the Microsoft development environment. These makefiles can be subsequently edited if the conventions that the development environment requires are maintained, otherwise the development environment will not "understand" the semantics of the makefile.

In Montana, source files listed in the *configuration* file (including header files) are processed only once, regardless of include directives. This makes incremental update of the CodeStore very fast. In order to make this work, global declarations declared in one file become automatically visible in all of the other files. Include directives are ignored for files named in the configuration file, and forward declarations are unnecessary. Finally, declarations are not assumed to be topologically ordered within a file. This means that function bodies and global initializers cannot be processed until all of the top-level type and function signatures have been processed. These processing rules work together to relieve the programmer of much of the drudgery associated with the maintenance of complex systems of header and source files. As an example of this, consider the simple Montana configuration of Figure 1, which consists of three files and a configuration file.

Contrast Figure 1 with the same program, as shown in Figure 2, written for a traditional compiler. Note that both classes must be complete before either of the inline function bodies can be compiled. Note that Montana will happily compile the program of Figure 2 with the trivial configuration file

```
target "main.exe" { "main.C" }.
```

This is because by default Montana defines macros to be local to the file in which they are defined. Because neither "A.h" nor "B.h" are listed in the configuration file, the #include directives are processed normally, and all of the macros become local to the span off "main.C". Macros that must be global to the program can either be included in every source file, or can be written in special source files that are annotated as "macro" files in the configuration file. These files are preprocessed in the order in which they are written in the configuration file unless they appear in include directives, in which case the order dictated by the include file graph is used. An example of this is shown in Figure 3. Because the `PI` macro is included in "formulas.h", `PI` is guaranteed to be defined before macro `area`. This discrimination between local and global macros is necessary to obtain reasonable semantics for the compilation model. because macro processing is inherently a file- and not program-scoped operation.

2.1 Semantic Differences Between ANSI C++ and "Montana" C++

Because top level declarations are automatically exported to all files in the C++ program in Montana, certain order-specific intra-file idioms are explicitly disallowed in order to obtain the same semantics as the inter-file equivalents. This means that when files are considered in isolation, the language that Montana recognizes differs from ANSI C++ in certain specific ways. Consider the following example:

```
const char* f(double)
   {
   return "double";
   }

const char* g()
   {
   // in Montana this is f(int);
   // in ANSI C++, f(double)
   //
   return f(5);
   }

const char* f(int)
   {
   return "int";
   }
```

Using a traditional compiler, the result of calling g() is "double", and in Montana, "int". As a second example, consider the following code fragment, written at global scope:

```
class C;

// In Montana the following is an error.
// The programmer must write "class C* x;"
//
C* x;

int C;
```

In Montana, if these declarations are at global scope, then this is a syntax error, because the "C" in "`C* x`" refers to the non-type. In both of these examples, the Montana semantics are roughly the following:

1. Treat the declarations as if each is written in a separate file.
2. Compile all top-level signatures first, assuming that each non-static declaration is automatically visible to every other file in the program.

main.C:
```
int
main()
   {
   B thing;
   thing.b();
   thing.a);
   }
```

A.h:
```
struct A
   {
   B* b()
      {
      return new B;
      }
   };
```

B.h:
```
struct B: A
   {
   A* a()
      {
      return new A;
      }
   };
```

main.icc:
```
target "main.exe"
   {
   source "main.C"
   source "A.h"
   source "B.h"
   }
```

Figure 1. Montana source files do not need header guards, not do they need explicit header files. If a file is listed in the Montana configurationfile, then non-static declarations are automatically visible across the program.

3. Compile all of the function bodies and variable initializers.

(Most of) the designers of the system consider the behavior of Montana for these kinds of examples as a feature, and not a bug. By automatically making declarations visible across the program without using header files, we obtain efficient incremental recompilation, because there is no need to recompile entire files if a file has changed. Only those portions of the file that correspond to changed declarations need to be recompiled. The compilation model is both clean and simple, and name resolution is not affected by lexical position within a file. In particular, Montana has strict enforcement of the ANSI one-definition rule, which says that class declarations are global across a program, not just a translation unit. There is also an additional benefit to tool writers because the lack of order dependencies inherently simplifies the language compiled. Some complexities vanish, *e.g.*, in Montana the point of template instantiation is simply a namespace, and not a lexical location in a translation unit.

2.2 Compiling Declaration Signatures and Function Bodies

Montana provides several ways to extend the compilation process [11]. For example, there are a number of distinguished points at which tool writers can insert code. By understanding the compilation model, tool writers can effectively extend the compilation by adding code in just the right places. The incremental compilation process, called incorporation, is now briefly described.

Recall that top-level declarations are unordered, and so their signatures must be completely processed before function bodies and variable initializers. The unordered processing of top-level declaration signatures is done by building a dependency graph, containing entities like files, macros, regions of source files corresponding to top-level declarations, declarations, and function bodies. This dependency graph is used to guarantee that changes to source files cause correct changes to be made to the CodeStore. Compilation works by adding work items to a priority queue, and using the dependency graph to create work items to be scheduled.

main.C:
```
#include "B.h"
int
main()
   {
   B thing;
   thing.b();
   thing.a();
   }
```

A.h:
```
#ifndef A_h
#define A_h
struct B;
struct A
   {
   B* b();
   };
#include "B.h"
#endif
```

A.inl:
```
inline B* A::b()
   {
   return new B;
   }
```

B.h:
```
#ifndef B_h
#define B_h
#include "A.h"
struct B: A
   {
   A* a()
      {
      return new A;
      }
   };
#include "A.inl"
#endif
```

Figure 2. The traditional compilation model (header files + declaration ordering) requires that programmers use preprocessor directives to control the order in which declarations are processed by the compiler. For complex class hierarchies it is also necessary to split inline function bodies into separate files.

pi.h:
```
#define PI 3.14159
```

area.h:
```
#include "pi.h"
#define area(r) (PI*r*r)
```

main.C:
```
int main()
    {
    cerr << area(5) << endl;
    }
```

main.icc:
```
target "main.exe"
    {
    options macros(global)
        {
        source "iostream.h"
        source "pi.h", "area.h"
        }
    source "main.C"
    }
```

Figure 3. Macros that are global to the entire program are listed in files that are listed in special "global macro" files in the configuration file.

Incorporation begins by processing source files whose time stamps have changed, subdividing the token stream into regions corresponding to top-level declarations. This subdivision is done by looking for fiducial symbols like braces, semicolons, and parentheses. For each of these changed files, its token stream is compared against the one obtained from the last incorporation, looking for source region differences. Only those regions that have changed must be processed. Processing of a source region is done by parsing and then *reconciling* the parsed declaration. Reconciling means finding the declaration, if any, from the CodeStore that matches. A change-analysis is done, and *dependent* declarations that need to be recompiled are scheduled as work items on a priority queue. Processing for function bodies and variables initializers is quite traditional. They are parsed, semantically analyzed, and then code is generated for them. In order to extend this, tool writers can replace any of these phases or add new ones.

In order to allow for incremental recompilation, dependency arcs are added to *antecedent* declarations, ones referenced inside declaration signatures, function bodies or variable initializers. Note that tool writers might have to add arcs as well in order to guarantee correctness of their extensions.

Parsed function bodies are not kept in the CodeStore. Instead, CodeStore provides facilities for function bodies to be processed on the fly. For example, the CodeStore method

```
bool
FunctionBody::build
    (
    LexicalBlockStatement*&,
    Phase,
    Storage, ...
    )
```

has several parameters, three of which are described here. This method returns a boolean that denotes success -- the resulting abstract syntax tree is returned as the first parameter. The second parameter denotes the last compilation phase (from parsing through code generation) that is to be done. The third parameter is an object that denotes where the storage for the compiled function body is to be allocated. By convention, this storage is "owned" by the caller of this method. In this way, a caching strategy is the responsibility of the caller of the `build` method. For example, non-local flow analysis that uses a subset of the entire program call graph can be written, in which the writer of the analysis has complete control over the lifetimes of the abstract syntax trees used by the analysis. This behavior, giving the tool writer control over the lifetime of the objects created by the CodeStore, makes the system both more open and more extensible.

2.3 Extending Montana

Extension writers use the CodeStore programming interfaces to modify or extend the CodeStore contents and the compilation process. Extensions are supplied in dynamically loaded libraries, which are automatically loaded as part of the configuration file processing. Extensions are defined in a special ".ice" file, which names each extension, names the library that implements it, describes the options (if any) that can be specified in a configuration file. *Incorporation* extensions modify the function body and variable initializer processing, by adding additional phases to the process. For example, the incorporation extension mechanism is used to add style checkers to the compilation process. Event-based notification, or *observer* extensions are used to gather information about the compilation process. For example, the Montana user interface is notified whenever a declaration is removed from the CodeStore. *Dependency graph* extensions are parameterized by source files. Instances of dependency graph extensions are created by the configuration file processor when it encounters source files whose suffixes are registered with the extension. Each such instance induces a node in the CodeStore dependency graph corresponding to the extension source file. In this way, when the time stamp of the extension source file changes, extension-specific processing is automatically triggered to run against the file. As an example of this, it is easy to specify a yacc extension that process ".yy" files, and constructs a C++ source file that is then compiled. Certain predefined "meta" extensions are defined as part of the system. The filter extension runs an

external command, *e.g.*, `yacc`, whenever the time stamp of an input file changes.. These filters are easily daisy-chained by listing the output of the filter as a source in the configuration file. The different extension types are often mixed together to extend the compilation process in significant ways. A single "compound" extension may add dependency nodes to the graph, modify the way functions and variables are compiled, and gather information about other parts of the incorporation process itself.

Of relevance to this paper are the design choices that led to the kinds of extensions that the system provides. A much more complete description of the Montana extension mechanism is given in by Soroker et. al. [11], including comparisons to other well-known extension techniques, like meta-object protocols.

There is a tension in the design of the extension mechanisms between providing convenient hooks for tool writers and not compromising the compilation time. As an example of this, it is useful to provide a hook so that a tool writer can obtain control every time a token is scanned, but without slowing down the compilation if the hook is not used. Different kinds of extension require different kinds of hooks in the incorporation process, and these hooks must be implemented as efficiently as possible. Thus dependency graph extensions are attached to the dependency graph with dependency arcs, incorporation extensions are attached to lists of compilation phases, and observer extensions are added to context-specific lists.

The three extension mechanisms appear arbitrary, but were chosen to provide for different styles of extensions.

- Incorporation extensions arise naturally from the need to add new kinds of algorithms that operate on function bodies and variable initializers. Most optimizers are designed to modify abstract syntax trees produced by either semantic analysis or by a previous optimization phase.
- The observer extension arises from the need to gather information during the compilation process, typically for analysis. Aside from efficiency considerations (observation is used extensively to implement the preprocessor and so must be very efficient), observers gather data at a number of different phases in the incorporation process, and it is necessary to provide a uniform interface for observation.
- Dependency graph extensions also fulfill a very-specific need, to extend the compilation processing by adding specific processing for new kinds of source files, and to provide a way for extension-specific data to persist as part of the dependency graph in the CodeStore.

Each extension mechanism is registered with the CodeStore in a different, but appropriate way. Incorporation extensions are added by registering a procedure to be called after a given (named phase). Observer extensions are registered by adding a procedure to a list associated with a particular event. Dependency-graph extensions are registered by creating dependency arcs that join them to other dependency nodes in the CodeStore.

2.4 Trade-Offs

Because of the need to temper extensibility with efficiency, there are some drawbacks to the extension mechanism. Each of the three mechanisms has its problems. Dpendency-graph extensions are the most powerful, but also the most difficult to use. Tool writers must understand the dependency-graph processing deeply in order to implement sophisticated extensions. Additionally, these extensions are not easy to debug, as they can cause the compiler to stop working. Montana explicitly chooses to give the programmer power at the expense of safety.

Incorporation extensions provide for a single tool writer to extend the system by adding extra compilation phases, but provide only limited ways to sequence the phases. In particular, if several phases of the same type are added to the compilation, *e.g.*, several optimization phases, the only way to sequence them is to manually add them one at a time. This makes it difficult to use third-party incorporation extensions, because the extensions themselves are not named. This can be fixed by adding some complexity to the registration mechanism, either by naming extensions or phases.

Finally, observer extensions are the most problematic. The designers of Montana have to predetermine all of those events for which a tool might want notification. A given event might notify its observers from many different points in the compilation, so that adding new types of observers requires (some) knowledge of the compilation process. The preprocessor, the only completely instrumented part of the system, has about 50 different events. Depending on the desired event granularity, there might be thousands of different events of interest to tool writers if the system were completely instrumented, which it currently is not. Furthermore, each event requires some custom code because different kinds of event notifications have different parameters to them. The number of events has been reduced by making them less specific. For example, observers are notified when processing of a dependency node is completed. The observer is responsible for checking if this dependency node is of interest.

3. THE CODESTORE PROGRAMMING INTERFACE

CodeStore is a C++ class library for describing the different static and dynamic aspects of a C++ program. There are class hierarchies for describing static aspects of the program like tokens, source files, macros, source regions, declarations, abstract syntax trees, expressions, types, statements, compilation targets, linker relocations. Dynamic aspects like machine registers, storage, call stacks, and breakpoints are available as well. The programming interface provides basic, but complete information. Rather than attempt to provide complete representations for all aspects of compilation, the idea is to provide a repository whose contents can be used to construct other data structures, *e.g.*, control flow graphs or SSA. In order to provide for this kind of extensibility, we have tried to minimize the number of ways in which the different parts of the CodeStore interact. The themes that underly the design, like CodeStore size, consistency, simplicity, and openness are now described.

The Montana system is also extensible, as we have described, and one way that object-oriented systems are often extended is using inheritance as an extension framework. Not surprisingly, Montana is implemented in C++, and the CodeStore is implemented using classes and methods, but we do not use inheritance as an extension to implement an extension framework.

- The use of inheritance or other C++-specific features like virtual or multiple inheritance is kept to a minimum in order to keep the size of the constructed CodeStore objects themselves small, and to keep the runtime overhead of using the system low -- recall that CodeStore methods are used in the inner loops of an incremental compiler, and so efficiency is important.
- Rather than trying to provide all of the methods that tool writers will need, a choice was made to export the CodeStore structure, and let tool writers have access to the same interfaces as the CodeStore implementors. Thus the CodeStore is an information repository, not an extension framework. The extension facilities, previously described, are cleanly separated from the information access facilities.
- Maintenance of tools is always an issue. In order to make tools based on CodeStore readable as well as writeable, we have tried to give the tool writer access to the control flow as much as possible by minimizing access to the information via call-backs and instead providing methods for type-case statements and downcasts. This makes the program control-flow much easier to understand and maintain.

3.1 Keep it Small and Simple

In order to reduce the size of a CodeStore instance and the cost of using one, complex C++ inheritance mechanisms are not used. The CodeStore classes use inheritance to provide implementations, and not as part of an extension framework. There is no virtual inheritance, because the storage layouts for classes with virtual inheritance tend to be larger than those without. This means that there are a number of methods in the CodeStore interface that are used by CodeStore itself, that should not be used by users of CodeStore. Instead of separating these using language features or as separate interfaces, they are separated as different parts of the particular CodeStore interface class, in much the same way that methods in Smalltalk classes are grouping together into public and implementations parts by convention. We document a "line" in each CodeStore interface class that separates out end-user methods from internal CodeStore methods. By doing this, we have traded off speed and interface simplicity against safety. We have further traded off simplicity against extensibility by not defining an extension framework. It was decided that the compilation algorithms and internal data structures are sufficiently complicated that providing extension through interface inheritance was only asking for trouble. Instead, we studied the compilation process, and tailored specific kinds of extensions facilities.

In order to reduce the size of a given CodeStore object, each interface object has a number of implementations. As an example of this the CodeStore interface object `FunctionParameter` has a number of sub-classes corresponding to different implementations, as shown in Figure 4. For example, most function parameters do not have default arguments, so we provide two different implementations of the function parameter interface, and use an optional return argument of type `Expression` to denote a default argument in the instance. Having multiple implementation classes trades off code size against data size. There are two implementations of the `isRegister` method instead of one, but no data to represent this in instances of any of the four classes. Default return values are provided in base classes, and are overridden in the mixins. Thus we are

```
template<bool reg>
class FunctionParameterTemplate: public FunctionParameter
   {
public:
   FunctionParameterTemplate(const Atom&, TypeDescriptor&);
   virtual const Atom& identifier()       { return _id; }
   virtual bool        isRegister()       { return reg; }
   virtual Expression* defaultArgument()  { return 0;   }
   ...
   };

template<bool reg>
class FunctionParameterWithDefaultTemplate: public FunctionParameterTemplate<reg>
   {
public:
   DefaultedFunctionParameterTemplate(const Atom&, TypeDescriptor&, Expression&);
   ...
   };

typedef FunctionParameterTemplate<false>          FunctionParameterImpl;
typedef FunctionParameterTemplate<true>           RegisterFunctionParameterImpl;
typedef DefaultedFunctionParameterTemplate<false> DefaultedFunctionParameterImpl;
typedef DefaultedFunctionParameterTemplate<true>  DefaultedRegisterFunctionParameterImpl;
```

Figure 4. Implementation classes for CodeStore objects are specialized to reduce object size.

able to provide very simple interfaces for the tool writer, and continue to keep the runtime size of CodeStore objects down.

There are three consequences of using multiple implementation classes. First, object construction is complicated, because the proper implementation class must be chosen. This is mitigated by providing factories to construct instances of CodeStore objects. For example, there might be 60 classes that implement different flavors of functions, but there is only one method for creating function declarations in the CodeStore factory class that creates declarations. Because inheritance is not used for extension, factories are not overridden by tool writers, and so the details of the different implementation classes can be hidden. This fosters open tool-writing at the expense of extensibility because one common kind of extension framework is to allow programmers to store data nuggets with the repository objects themselves.

The second consequence of using multiple implementation classes is more subtle, and deals with object mutability. As a consequence of incorporation processing, we might determine that, for example, a function that used to be a static member function is now a regular member function. Because we happen to have chosen to represent the linkage of a function indirectly using the template-mixin mechanism, the identity (address) of the represented function in the CodeStore can not be maintained. This causes change propagation to be done, and function bodies that invoke this function must then be recompiled. For this particular example, it is not serious, because the invocation code changes anyway. The trick is to use the mixin technique for aspects of the CodeStore objects that we expect to be invariant, or for aspects, which, if changed, necessitate recompilation anyway. Thus we must trade off CodeStore runtime size against compilation incrementality.

The third consequence of using multiple implementation classes is that of indirection, which impacts the runtime cost of using these CodeStore objects. Whether these multiple implementations are implemented by delegation, or by use of an indirect (virtual) function call, the overhead of obtaining data from a CodeStore object is increased. This cost can be reduced by pushing common data up the hierarchy tree, so that access does not require any function-call overhead whatsoever. In tuning the system, this proved to be a dramatic win for certain often-accessed data, like the name or enclosing scope of a declaration.

CodeStore size is dramatically lessened by not explicitly storing abstract syntax trees of function bodies and nonlocal variable initializers. These are computed on demand. Because all of the declarations to which they refer are stored away in the CodeStore, construction of an abstract syntax tree (AST) for a function body is very fast -- it consists of parsing the preprocessed token stream for the function body, and semantically analyzing the AST. This computation on-demand paradigm is used throughout the Montana system. For example, when a user of the Montana user interface sets a breakpoint at a file/line/column source location, the source region is parsed and analyzed, then the source location is correlated against the AST in order to obtain a program counter for the breakpoint. In practice, the response time is instantaneous.

3.2 Export the CodeStore Functionality

The key to making the CodeStore functionality available for programmers has been to separate out representation from behavior, so that programmers have access to the functionality that CodeStore provides, like lexing, parsing, type analysis, diagnosis, etc. By separating these capabilities out from the CodeStore representation itself, tool writers are able to use them in tools. Rather than try to predict the kinds of things that users of the CodeStore programming interfaces will wish to do, we provide set/get methods and general extension mechanisms as previously described. The model that we adopted treats the CodeStore as a repository, with each function that operates on the CodeStore implemented by separate, "helper" classes. This opens up the CodeStore to tool writers, because there are no restrictions on who implements these helper classes. Furthermore, it provides a single uniform tool extension model, that is shared by both the CodeStore implementor and the tool writer. As an example of this, it is not possible to change what "parse" means for a function body, but it is possible for programmers to define their own parser implementations which might, for example, construct abstract syntax trees directly.

By making the CodeStore just an information repository, and separating behavior from implementation, we make it possible for programmers to use the helper classes directly in ways not anticipated by the CodeStore designers. There are many examples of this. We provide a method for a parser to produce an abstract syntax tree from a character string. Thus CodeStore users can construct strings directly and process them using the same code as does CodeStore. Two examples of this are expression evaluation in the CodeStore debugger, and searches for declarations in the CodeStore. The first works by using the CodeStore parser, type analyzer, diagnostician, and transformer to process a character string representing the expression. The second example, searching for declarations in the CodeStore, does not seem to require any sophisticated support from the CodeStore until one considers how to search for

```
A<int>::B<(T&)foo>::C<X+Y+Z>
```

in the program. By providing a way to parse this into an abstract syntax tree for a name, and then type-analyze this, the writer of the declaration searcher just has to write a "one-liner" that is guaranteed to obtain the same result as a CodeStore compilation. By separating out the operations that produce and process abstract syntax trees from the CodeStore itself, we are able to export powerful CodeStore functionality to the tool writer very easily.

Another way in which the CodeStore functionality is exported is the way in which the incorporation is invoked. For incorporation either of function bodies or of an entire program, the user supplies an `IncorporationController` to CodeStore. This object provides a number of services. Among them is a place in which to deposit error messages generated as a result of incorporation. These error messages are actually n-ary trees, structured so that children provide more refined information about their parents. An example of this is a type analysis error. The top-level message may say something like "No best conversion between types `A` and `B`". Other parts of the message give details about what the choices were, and why one choice is not better than the others. These error messages can be either graphically presented as they are in the CodeStore UI, with a "more detail" button, or they can be selectively displayed, as they might be by the expression evaluator in the debugger.

3.3 CodeStore Programs must be Readable as well as Writeable

One of the ways in which CodeStore programs have been made readable is counter to the object-oriented way of "doing things." We have already said that in CodeStore, data and behavior are largely separated, so as to necessitate only one set of programming interfaces for both the system and its users. By separating out data and behavior, algorithms that use the CodeStore programming interfaces become easier to understand -- the control flow is directly manifested by the code itself. By uniformly exploiting programming conventions there becomes still less need for programmers to deal with large numbers of classes in order to understand CodeStore program fragments. Another way in which the data/behavior split simplifies tools is the way in which class hierarchy navigation is implemented. Programmers can of course use features provided by the language itself, like C++ runtime type identification (RTTI):

```
ClassDeclaration* c;
EnumDeclaration* v;
if (c = dynamic_cast<ClassDeclaration>(d))
   // ;;
else if (e = dynamic_cast<EnumDeclaration(d))
   // ;
```

Code like this is easy to write, easy to read, and more importantly, gives the programmer control over the control flow. There are instances where is useful to provide more structured access to CodeStore data structures, using *visitor* classes. While these are not useful in general, there are certainly many situations where it is desirable to visit an abstract syntax tree, performing certain actions at each of the different subtrees. CodeStore provides classes for traversals, and tool-writers can easily plug in actions to be performed, for example, for each cast expression. Because the implementation of these traversal classes does not use any specialized methods that are not part of the CodeStore

programming interfaces, tool writers can implement specialized traversal classes tailored for specific uses. It is also worth noting that a very (very) early version of CodeStore using only this style of navigation for both data structure and class hierarchy traversal. It became clear very early that this was inadequate for a general framework such as CodeStore. Imagine a mechanism like this used to compare a function parameter against an argument list. There is also a granularity issue. Using such a scheme it would have been very difficult to differentiate between different kinds of functions and variables.

There are two class-hierarchy navigation primitives provided by CodeStore that are used to implement data structure navigation like that of Figure 4. The first is a type-case, and the second is a single-level downcast, very similar to C++ RTTI. For each class A, with subclasses A_0, A_1, ..., A_k, the methods look like the following:

```
class A
   {
public:
   enum Kind
      {
      IsA0,
      IsA1,
      ...
      IsAk,
      numKinds
      };
   virtual Kind aKind() = 0;
   A0* asA0();
   A1* asA1();
   ...
   Ak* asAk();
   ...
   };
```

Using these two mechanisms, it becomes very easy to write code against the CodeStore interfaces. In particular, the code is easy to write, understand, and debug. These are all desirable characteristics of maintainable code. The downcast and type methods are uniformly used throughout the CodeStore classes. Each base class has a "Kind" enumeration, and a set of downcasters to subclasses. By requiring (by convention) that programmers use these facilities instead of C++ RTTI, readability and flexibility is obtained while maintaining structure. Additionally, the programs tend to have a great deal of uniformity, and CodeStore idioms like a type case become very easy to identify. Common wisdom is that this kind of mechanism causes type-errors to become runtime errors instead of compile-time errors. We have found that not to be the case. CodeStore is a large (750KLOC) program that is written in this way, and the advantage of providing obvious control flow swamps any disadvantages.

A programming convention that has proved very useful for readability as well as writeability of CodeStore programs is one that we call the pointer/reference convention. Simply stated, pointers are used in CodeStore methods to denote optional objects. Thus if a CodeStore method returns a pointer, then that return value might be null, and should be checked. (Recall the default function argument example from the previous section). We ask that our tool writers obey the same rule. Trivial as this convention sounds, it has dramatically simplified the CodeStore programming interface, and increased the readability of code written against it.

One final aspect of readability is the notion of uniform navigation through collections of CodeStore objects. For most objects we can define a primary collection, *i.e.*, the collection primarily through which the object will be obtained. For declarations, this primary collection is the scope containing the declaration, for statements, the containing block, for types, the type declarator list. Navigation through this primary collection is made deliberately easier. Code that navigates through these collections looks very familiar to C/C++ programmers. As a simple example of this, consider a function that determines whether a class has any constructors defined. The code searches through the members of a class (a mandatory part of the CodeStore representation of a C++ class is a scope for the members of the class). The code tests to see if each declaration is a function, and if so, if that function is a constructor (see Figure 5).

The largest client of the CodeStore interfaces so far is Montana itself, which is approximately 0.75M lines of code, and most of the system consists of code like this. The code is very writeable and is certainly readable. This example does not use the type-case idiom, but it does use a single-level downcast, the pointer/reference convention, and collection iteration.

```
bool
hasConstructor(ClassDeclaration& c)
   {
   MemberDeclarationStore& m = c.memberDeclarationStore();
   for(Declaration* d = m.firstDeclaration(); d; d = d->next())
      {
      FunctionDeclaration* f = d->asFunctionDeclaration();
      if (f && f->isConstructor())
         return true;
      }
   return false;
   }
```

Figure 5. Determine if a class has a constructor by examining each declaration, in order, and determining if that declaration is a function and a constructor.

4. CONCLUSIONS

We have had a large amount of experience with the CodeStore programming interfaces. A large number of tools have been written against CodeStore. Some published examples of these are: Rapid Type Analysis [1], a simple analysis for determining when dispatched functions can be called directly; a correct implementation of reference-counted pointers [7]; and the most complicated of the CodeStore tools, an implementation of subject-oriented composition [9]. Other tools written have been: a semantic definition/use view for the Montana user interface; a diagnosis tool, that identifies erroneous C++ idioms; a code generator that compiles C++ into SUIF [6]; and an extension to the system that analyzes macro expansion during compilation. Many of these tools were used to drive the development of the CodeStore programming and extension interfaces, and so consequently the writers of these tools did not have as positive experience as the CodeStore developers would have liked. To summarize their experiences, the following lessons have been learned:

1. Regardless of how good the programming interfaces are, CodeStore is a big system, and there is a steep learning curve. The programming interfaces are easy to learn, but we need to publish a book of idioms, a programmers guide. Once the idioms have been learned, tool writers can be very productive.

2. We have made the system very powerful, at the expense of safety. Not surprisingly, integration of complex tools can cause the compiler to crash during compilation, and debugging of integrated tools is not necessarily easy. Not surprisingly, storage management is a perennial problem. It is unclear whether garbage collection could help, because the issue is memory *staging*. Even if we supplied a garbage collector, persistent objects (stored in the persistent CodeStore in a file) are different from non-persistent ones.

3. The separation of data and function has been a huge win. Tool writers have become very acclimated to the power at their disposal. Post-hoc parsing, analysis and tool-traversal of function bodies has been very useful.

4. The number of tools written against the different extension mechanisms is inversely correlated with their power. The addition of analysis and diagnosis phases to function-body and variable-initializer analysis have been widely used, perhaps because this is a very familiar paradigm for compiler writers. The observer mechanism has also proved to be useful. In fact we have used user requirements to drive the addition of observers. Finally, the dependency-graph extensions have been the least used. They are by far the most powerful, but the most complicated. Tool writers need to have a fairly deep understanding of the incorporation process to use them.

5. The use of consistent programming conventions has been a real aid to learning the interfaces. Even though there are a large number of classes and methods in the CodeStore programming interface, uniformity conventions like the pointer/reference, downcast, and collection iteration have dramatically reduced the number of concepts that tool writers need to have.

6. The system is incremental, and so tools must also be aware of this incrementality. This means that all tools must to some extent be aware of the incorporation model. In some cases, tools that assume views of the entire program, like Rapid Type Analysis, have had to be rethought. Other kinds of tools, like diagnosis tools, fit very naturally into an incremental model.

The architecture of the system has proved amenable for C++ tools. Although the programming and extension interfaces are necessarily complex, matching the semantics of the language itself, the system has been able to show that this is a very viable architecture for a compiler and programming environment. CodeStore as a single source of information has proven itself to be a very useful information source for a

tool writer, and the extension mechanisms have also proven themselves, although to a lesser extent.

5. ACKNOWLEDGEMENTS

A large number of people have worked on the design and implementation of the system over the last four years at IBM research and development sites. In no particular order, other contributors to the system design are John Barton, Lee Nackman, Derek Lieber, Yi-Min Chee, Danny Soroker, Robert Bowdidge, David Olshefski, Derek Inglis, David Streeter, Mark Mendell, Ian Carmichael, David Brolley, Ed Merks, Michael Wulkan, Jamie Schmeiser. Many, many people have contributed to the implementation of the graphical programming environment called Montana. Many others have used the system, albeit with pain, during its gestation. Finally, thanks to Peri Tarr who read this paper very carefully and helped to fix it.

6. REFERENCES

[1] Bacon D.F., Sweeney P.F., "Fast Static Analysis of C++ Virtual Function Calls" *OOPSLA* 1996, pp. 324-341.

[2] Cayenne Software Inc., *Simplifying the Maintenance of Your C Code Using Software Reverse Engineering Technology*, http://www.cayennesoft.com.

[3] Edison Design Group, *Compiler Front Ends for the OEM* Market, http://www.edg.com.

[4] Feldman S.I., "Make -- a computer program for maintaining computer programs," *Software Practice and* Experience, 9(4), pp. 255-265.

[5] Haberman N.A., Notkin D., "Gandalf: Software Development Environments," *Transactihons of Software Engineering, 12(12), pp. 1117-1127.*

[6] Hall M.W., Anderson S.P., Amarasinghe S.P., Murphy B.R., Liao S.-W., Bugnion E., Lam M.S., "Maximizing Multiprocessor Performance with the SUIF Compiler," *IEEE Computer*, Dec. 1996.

[7] Hamilton, J., "Montana Smart Pointers: They're Smart, and They're Pointers", *Proceedings of the Conference on Object-Oriented Technology*, Portland, OR., 1977.

[8] Nackman, L.R., "CodeStore and Incremental C++," *Dr. Dobbs Journal,* Dec. 1997, pp. 92.

[9] Ossher H., Kaplan M., Katz A., Harrison W., Kruskal V., "Specifying Subject-Oriented Composition," *TAPOS Special Issue on Subjectivity in Object-Oriented* Systems, 1996, pp. 179-202

[10] Reps T.W., Teitlebaum T., *The Synthesizer Generator: A Language for Constructing Language-Based* Editors, Pringer Verlag, 1988.

[11] Soroker D., Karasick M., Barton J., and Streeter D., "Extension Mechanisms in Montana," *Proceedings of the 8th IEEE Israeli Conference on Computer Systems and Software Engineering*, Herzliya, Israel, June 1997, pp. 119-128.

[12] Take-Five Software Corporation, *Sniff+ for C/C++*, http://www.takefive.com.

[13] Bladen J.B., Blake S.J., Spenhoff D.,, "Ada Semantic Interface", *Proceedings Ada '91,* San Jose, CA, Oct 1991, pp. 6-15.