RC 20770 (92025) 3/14/97
Computer Science/Mathematics 14 pages

Research Report

Extension Mechanisms in Montana

Danny Soroker, Michael Karasick, John Barton

IBM Research Division
T.J. Watson Research Center
Yorktown Heights, New York

David Streeter

IBM Software Solutions Division
Toronto, Ontario, Canada

IBM Research Division
Almaden · T.J. Watson · Tokyo · Zurich · Austin

Extension Mechanisms in Montana

Danny Soroker Michael Karasick John Barton
IBM Research Division, Hawthorne, New York

David Streeter
IBM Software Solutions Division, Toronto, Ontario

Abstract

Montana is a new C++ programming environment being developed at IBM. Montana aims to improve the software development experience by providing rapid incremental builds, rapid complete builds, and a tightly integrated user interface driven from a single source of information. Montana uses a program representation (*CodeStore*) to achieve these aims, with compilation proceeding directly from the database. APIs for accessing information about programs from the CodeStore are also made available for tool builders. In this paper we focus on the extension mechanism in Montana, which enables integration of existing tools and creation of new sophisticated, tightly integrated tools.

1 Introduction

To write language-knowledgeable tools, the tool writer must have access to components that manipulate the language, i.e., parser, semantic analyzer, etc. These tools are often written by taking an existing compiler and modifying it, adding tool-specific code to some parts of the compiler, and removing non-specific code from other parts. The *Montana* programming environment provides an alternative approach: programming interfaces and services for inserting tool-specific code into different parts of the compiler, fundamentally changing the compilation process.

Montana extensions have been used to implement a variety of tools that would be difficult or impossible to implement with existing compilers. One example is an extension of the preprocessor that provides macro expansion views in the Montana user interface. Another extension implements subject-oriented programming[11], a technique for constructing programs by composing entire class hierarchies.

Montana provides an incremental compiler, a graphical user interface, and a set of programming interfaces that access a program representation used by both the compiler and the GUI. The program representation, or *CodeStore*, contains information like a global symbol table, preprocessed source, compiler options, declarations, etc. An early version of this program representation was described in [1]. In particular, much of the CodeStore is organized as nodes in a dependency graph. This graph is used to drive the incremental compilation process.

Because Montana is incremental, its compilation process does more than just produce object code. Instead compilation is a process of updating the CodeStore to reflect changes in the program source files. The CodeStore is updated by propagating changes through the dependency graph, updating nodes representing source files, regions within them, declarations, function bodies, and so on. We call this process *incorporation*, because the processing takes the source files and "incorporates" program changes into the CodeStore.

The extension writer can customize the incorporation process by adding three kinds of *extensions* (or *tools*):

- *Observer Extensions:* Observers [5] are notified of events at distinguished points in the incorporation process. Observers are lightweight and largely insensitive to the details of the incorporation process. They implement an event-based message service. One use of observers is to implement tools that gather statistics about the incorporation process. To illustrate this, Section 4 describes an observer extension that counts the number of function bodies processed during the incorporation.

- *Incorporation Extensions:* Function body processing is done by a fixed sequence of phases, from parsing to code generation. Incorporation extensions allow extension writers to insert extra phases, like new optimization algorithms, into the sequence. Section 5 describes an incorporation extension that adds a simple style-checker that flags excessive nesting depth of function bodies.

- *Dependency Graph Extensions:* Custom nodes and arcs are added to the dependency graph. These new nodes participate in the change propagation that is intrinsic to the incorporation process itself. Section 6 describes such an extension for resource compilation on a PC platform, which adds nodes corresponding to source files for the resource compiler.

Because the three extension mechanisms allow the incorporation process to be changed quite significantly, the extension mechanisms are both powerful and dangerous. Extensions are free to examine and modify the CodeStore. The design point is that extension writers "know what they are doing." If not, they would not be tinkering with the compilation process.

The rest of the paper contains the following sections. Section 2 describes the parts of the incorporation process relevant to extensions. Section 3 describe how extensions are specified to the system. Sections 4, 5 and 6 detail the different extension mechanisms. Finally, Section 7 provides a discussion and summary.

1.1 Related Work

Creating extensible systems is not a new idea. Emacs[13] was written in 1979. There are many extensible programming environments that allow externally written programs to be added to the environment. One such example is Gandalf[7]. Treating compilation as dependency-graph propagation is also not a new idea (e.g. [2, 12, 14]). The most ubiquitous of such schemes is the Unix make utility [3], which provides a way to obtain certain dependency graph extensions at a coarse (file) level. Montana is the first system to combine these two ideas of extensibility and dependency-graph-based compilation.

There has also been work on limited kinds of compiler extension. Meta object protocol [9], first used in LISP systems, lets extension writers define specialized behavior for actions in the LISP compiler like compiling variable assignment. Compiler vendors like Edison Design Group[6] provide compiler front-ends (parsers and semantic analyzers) that can be used to write compilers and tools although the front-ends are not designed to be extensible. The Sage++ system[16] supplies a parser/analyzer specialized for source-to-source translation tools. Unlike Montana, the kinds of extensions possible in these systems are limited.

Montana provides a variety of ways to extend itself. Furthermore, these extensions are added to the system using the powerful notion of dynamic linking [4]. As an example of the power this gives, an extension writer can use Montana to develop, say, a style-checker tool, and once the style-checker library is generated, to add it as an extension and test it on its own source code!

2 Incorporation Phases Related to Extensions

For the purposes of this paper it is not necessary to describe the details of C++-specific incorporation processing. Instead, we focus on the processing that is specific to adding extensions to an incorporation, and invoking the extensions.

The Montana extension mechanisms are used in the context of an incorporation (incremental compilation) process. The incorporation operates by propagating changes through a dependency graph that represents the compiled program. When the state information in a node is determined to be stale, the node is *invalidated*, and the node is placed on a (priority) work queue. Each step of the incorporation works by removing the node at the head of the queue and invoking its node-specific update processing.

A Montana CodeStore is associated with a C++ project. Information describing each of the extensions to which the project has access is contained in special text files. There are two such files – for Montana-provided and user-written extensions. The first step in the incorporation process updates the CodeStore's extension information with that obtained from the two files (see Section 3.2).

The next main incorporation step is processing the *configuration file*, which describes the project itself: C++ sources, linker targets, extension sources, tools, options, and environment variables that affect incorporation. Section 3.3 describes how processing of the configuration activates specified extensions (by dynamically loading their associated shared libraries) via a tool directive or a source directive.

Once the configuration file has been processed, the CodeStore contains a list of all the source files that contribute to the incorporation, and a dependency node has been created for each of them. In addition, all the extensions that actively participate in the project have been activated. Each of these extensions gets the option of running a startup function, that registers observers and incorporation extensions. Subsequent steps of extension processing are interspersed with the C++ compilation described below.

After source-file processing, the next step is source-region processing. For C++ sources, this is an interface parse (a parse of global and namespace declarations, skipping function bodies and variable initializers). These declarations are updated, and dependent nodes are invalidated. For example, if a source region parse causes the size of a class to change, dependent nodes to be invalidated include sub-classes, and function bodies and source regions that call constructors of this class.

After source-region processing, the next step is function-body and variable-initializer processing, comprising parsing, semantic checking, optimization, object-model transformation, and code generation. The final compilation step is to publish the target(s). Source-region, function-body, and variable-initializer processing are not described in this paper, except to illustrate how they can be extended.

Extension processing steps are interspersed throughout the C++ portions of the incorporation. For example, there is an observer callback point each time a declaration has been successfully parsed and added to the CodeStore. As a second example, incorporation extensions constitute extra passes added to function-body and variable-initializer processing. Sections 4, 5 and 6 elaborate on how extension processing is orchestrated.

3 Configuration and Extension Specification

This section describes how Montana extensions are specified. The specification of a given extension has two parts:

1. Specify general properties of the extension as part of the Montana environment.
2. Specify how the extension is used in a given project.

The first specification, provided by the extension *writer*, feeds into a registration mechanism that tells Montana about the extension. The second specification, provided by the extension *user*, describes how the extension is utilized in building a particular project.

As a concrete example we will describe a project that uses an extension of each of the three types. These three extensions will also be the working examples for the rest of the paper. *Resource compilation* (a process familiar to programmers on PC platforms) converts a textual *resource script file*, which describes visual resources for the windowing system (such as menus and dialog boxes), to a binary file that is linked into an executable or library being developed. The sample project uses the *resource extension*, which is a dependency graph extension that handles resource compilation for Montana projects. The sample project also invokes an incorporation extension that performs some simple style checking and an observer extension that performs some incorporation bookkeeping.

3.1 Montana Configurations

A Montana project is defined in an editable text file called a *configuration file*, which plays a similar role to that of a makefile in traditional development environments. Each CodeStore is associated with a single configuration. Configuration information is stored in the CodeStore, and can be queried. This information is used by the system to determine how the configuration has changed from one incorporation (build) to the next.

The configuration file describes the source files and target files that participate in the project, and their relationships, as well as the options to be used when building the target files from the source files. The configuration file also describes what tools to invoke during incorporation.

The sample configuration (Figure 1) defines a project for building the program hello.exe. This target is built from the constituent sources hello.cpp, resource.h and hello.rc. The first two sources are C++ files, and are handled by the Montana kernel. The third source, hello.rc, is a resource script file, which is handled by the resource extension. Association of a given source with an extension is through its *source type*, which defaults to the file suffix. This notion of implicit association based on file type is prevalent in modern operating systems and other extendible systems such as web browsers. The association in Montana is set up as part of extension specification. The first two options are C++ compiler options, whereas the options res_opts and max_nesting_level are used by the extensions (see Sections 5,6).

3.2 Specifying Extensions

Montana extensions are specified in special files called *incremental compiler extension files* (".ice" files), which have the format of Windows ini files. These files are scanned when the CodeStore is created and at the start of every incorporation (so that extension definitions may be added, removed or changed during the lifetime of a CodeStore). The CodeStore contains an ExtensionStore, which contains a list of extensions. An *Extension object* is created for every extension that is registered, based on the following information, which is specified in the .ice file.

```
target "hello.exe"
  {
  option define(TRACE, 1),
         gen(arch, "pentium"),
         res_opts("-dFAST")
     {
     source "hello.cpp",
            "resource.h",
            "hello.rc"
     }
  option max_nesting_level(20)
     { tool "NestingChecker" }
  tool "FunctionBodyCounter"
  }
```

Figure 1: Example of a simple configuration

- The dynamically linked library (dll) that contains the behavior of the extension.
- A distinguishing name ("class name") that is used to associate the Extension object with the class that corresponds to instances of that extension.
- A list of the file suffixes that are handled by the extension. This determines the association of sources in the configuration with dependency graph extensions.
- The options that are recognized by the extension.
- Various entries used primarily by the Montana user interface, such as a descriptive string for the extension, a help URL, and an error-message prefix.
- Any other entries that are understood by the extension.

The last item above enables the extension writer to have part of the extension's behavior encoded as data in the .ice file. One particular application of this is to share a single parameterized implementation between several extensions, where the parameters values appear in the .ice file. Figure 2 shows the .ice specification of the resource extension. Entries for the other extensions have a similar format.

3.3 Activating Extensions

As indicated above, an extension is used by referring to it in the configuration. This reference can take on one of the following two forms.

Observer and Incorporation extensions: activated via a configuration tool directive (e.g. tool "FunctionBodyCounter"), that refers to the extension by its distinctive class name.

Dependency graph extension: via a configuration source directive (e.g. source "hello.rc"), that lists a source whose type is associated with the extension.

```
[Resource]
type=extension
description=Resource Compilation Extension
help=ResourceExtension.html
dll=montres.dll
suffixes=rc RC
prefix=res
options= ResourceCompilerName res_opts

[ResourceCompilerName]
type=option
kind=string
default=irc

[res_opts]
type=option
description=Options for the resource compiler
kind=string
```

Figure 2: Snippet of an incremental compiler extension (".ice") file

In both of these cases, control needs to be passed from the Montana kernel to an entry in the extension dll that knows how to hook into Montana. This is implemented through a "shared" ExtensionLoadPoint object, whose behavior is provided by the Montana kernel, whereas its data is specified in the extension. This is a static object in the extension dll, and is thus created when the dll is loaded. On construction, an ExtensionLoadPoint instance is added to a global list, and is, therefore, available for future lookup by the kernel. The ExtensionLoadPoint's data contains the dll name, the extension's class name and several function pointers, which are responsible for various operations, depending on the type of extension. Further details on activation of the different extension types appear in Sections 4, 5 and 6.

4 Observers

The simplest of the three extensions kinds is the observer extension. Unlike dependency graph and incorporation extensions, observers do not affect incorporation control flow. Observers receive notification callbacks from distinguished, "interesting" points in the incorporation. These interesting points are found throughout the system. Each event is described by an abstract base class with a specialized notify() method. For example, the notify method of a macro-expansion observer will take as parameters the macro and the substitution text; for a dependency-node-removal observer, the parameters are the dependency node and the CodeStore. The abstract base class of the observer dictates when the observer is notified, and the content of notify() gives its behavior. As an example, Figure 3 shows an observer that counts the number of function body updates during the incorporation.

Other approaches to a notification interface, like subclassing the Montana change propagator or before/after meta-object protocols (see e.g. [9]) were rejected as too unwieldy. In particular,

```
class FunctionBodyCounter: public DependencyNodeCompleteObserver
   {
public:
   FunctionBodyCounter: _fbCount(0) {}

   virtual void notify(DependencyNode& node, CodeStore&)
      {
      Implementation* i = node->asImplementation();
      if (i && i->asFunctionBody()) ++_fbCount;
      }

   ~FunctionBodyCounter()
      { cout << "Number of Function Bodies=" << _fbCount << endl; }

private:
   unsigned _fbCount;
   };
```

Figure 3: Example of a simple observer

```
...
if (observerChain != NULL)
  {
  Observer obs = observerChain;
  for (; obs; obs = obs->next())
      obs->notify(observer-specific argument list);
  }
```

Figure 4: Observer dispatch

funneling of events through the change propagator just for the purposes of notification is too cumbersome. Before/after protocols have a similar overhead of at least one dispatch per notification. Additionally, both require the extension writer to explicitly chain notifications.

In contrast, the Montana observer dispatch code is very lightweight, with little or no overhead for events without registered observers. Each event has an associated observer chain which is checked at the event site. Figure 4 shows how observers are dispatched. If there are no observers, the observer chain is empty, and the overhead is a test for a null pointer.

Adding an observer to an incorporation is straightforward as well. The observers have been grouped into class hierarchies, like *PreprocessorObserver* and *ChangeObserver*. Each hierarchy corresponds to a particular component of the system, and that component supplies an addObserver method that is invoked by the extension's startup method. As described in Section 3, the startup code is invoked early in each incorporation. Figure 5 shows the static startup method for the function-body-counting observer, which primes the change propagator.

```
void
FunctionBodyCounter::startup(IncorporationState& state)
    {
    // Obtain the change propagator used by the incorporation.
    //
    ChangePropagator& cp = state.changePropagator();

    // Add a new observer to the change propagator
    //
    cp.addObserver(*new FunctionBodyCounter);
    }
```

Figure 5: Adding an observer to the incorporation

5 Incorporation Extensions

Incorporation extensions are used to modify the way in which function bodies and variable initializers are compiled. Compilation of a function body or variable initializer proceeds in the following sequence:

Type Analysis Semantic checking and function overloading resolution, etc.

Diagnosis Additional semantic error message checking.

Optimization Removal of redundant temporaries, early inlining, etc.

Transformation Object-model transformation, i.e., for virtual function dispatch, member offsets, etc.

An incorporation extension provides a way for an extension writer to add one or more of these incorporation components to an incorporation. The extension writer defines a factory[5] that can create an additional incorporation component for each of the four types. The startup method of the extension is responsible for creating the factory and appending (or prepending) it to the list of component factories. After all extension startup methods have been run, each of component factories are invoked, and each of their defined components is added to the appropriate of the four lists.

Figure 6 shows a factory that creates a new diagnostician, which will be used to create an error message if a function body's nesting level exceeds some maximum level, which is specified as an option in the configuration (see Figure 1). Later, when the incorporation state is queried for diagnosticians, the one specified here will be obtained.

The NestingDiagnostician, shown in Figure 7, provides an entry point, diagnoseStatement(), which is called as part of the incorporation control flow to diagnose the outermost lexical block of a function body. The entry point does three main things: (1) obtain the nesting depth of this statement and the allowable maximum depth; (2) compare the two, and, if needed, generate an error message and add it to the message list; (3) obtain the next diagnostician in the chain and invoke its entry point.

Methods for creating error messages, adding them to a message list, and obtaining the next diagnostician are kernel services provided by Montana.

```
class NestingCheckerFactory: public ImplementationComponentFactory
   {
public:
   NestingDiagnosticianFactory(IncorporationState& state):
      _nestDiag(new NestingDiagnostician(state, state.diagnostician())) {}

   ~NestingDiagnosticianFactory()         { delete _nestDiag; }

   virtual Diagnostician& diagnostician()   { return *_nestDiag; }

private:
   Diagnostician* _nestDiag;
   };
```

Figure 6: Just before each incorporation, this factory is used to create a NestingDiagnostician, which is plugged into the incorporator.

```
class NestingDiagnostician: public Diagnostician
   {
   virtual void diagnoseStatement(Statement& s)
      {
      if (nestingDepth(s) > maxNestingLevel())
         addMessage(error(Severity::Error, s.sourceLocation(), "nested too deeply"));
      if (next())
         next()->diagnoseStatement(s);
      }

   unsigned nestingDepth(Statement& s)
      { /* calculate depth by subclassing Montana's DepthFirstVisitor class */ }

   int maxNestingLevel()
      { /* get the maximum allowable value from the extension option */ }
   };
```

Figure 7: The NestingDiagnostician generates an error message if a function body has more levels of nesting than some specified limit.

6 Dependency Graph Extensions

A dependency graph extension creates new dependency nodes and arcs which augment the CodeStore's dependency graph. These new nodes participate in the incorporation process via change propagation along dependency arcs. For example, the resource extension is triggered by a change in the resource script file or a file included by it, and it triggers processing of Montana target nodes, which also link in the compiled resource file. For an extension to be able to "plug in" to Montana in this fashion, it must be given control to establish itself, and it must be able to create dependency nodes and arcs.

6.1 Activating a Dependency Graph Extension

Referring to Figure 1 and the discussion at the end of Section 3 we proceed to describe the activation scenario for the resource extension.

During configuration processing in the early stages of incorporation, the source name "hello.rc" is encountered. The configuration processor does not recognize the source type as a C++ source, and thus passes this name to the ExtensionStore, which determines that the resource extension is responsible for handling files with suffix rc. The ExtensionStore triggers loading of the resource extension dll, montres.dll, and uses the function pointers in the associated ExtensionLoadPoint to create a special kind of dependency node called at *ExtensionSource* corresponding to hello.rc (more on this in Section 6.2). The list of configuration options in effect is then passed to the extension source hello.rc, which extracts and stores the options that it recognizes. In our sample configuration, the option passed to the resource extension is the string "-dFAST".

6.2 Infrastructure for Dependency Graph Extensions

Montana supplies a variety of services to simplify the extension writer's job. These services are provided for more than just convenience. Incorporation processing is inherently complicated, and the services help by abstracting away complexity that is common to most extensions. These services include two basic implementations of extension sources and other kinds of dependency nodes.

6.2.1 Kernel-Provided Extension Sources

The Montana kernel provides a full implementation of these objects. The four methods below invoke functions, any of which can be supplied as arguments to the ExtensionLoadPoint constructor. This capability has been found adequate for many tools. This kind of extension source is most convenient for extension writers, since much of the bookkeeping is done by the kernel. It is used by the resource extension.

isChanged() This method, which returns a boolean, is called for all CodeStore sources at the start of each incorporation. If the result is true then the source object is put on the incorporation queue for processing. The default implementation tests if the timestamp of the corresponding file differs from the recorded value. The resource extension overrides this method to additionally check if the compiled resource file exists.

update() For any dependency node, this is the method that constitutes "processing" of that node when it reaches the head of the incorporation queue. For the resource extension, a command is constructed and then run as a separate process. In our example, the command "irc -dFAST hello.rc" is run.

topologicalPriority() The incorporation queue is a priority queue (see Section 2), and this method determines when, during incorporation, a given node gets processed. For the resource extension we need a priority that is higher than that of Montana target nodes.

processOptions() This method is called by the configuration processor with the list of configuration options that are in effect for the source. The default implementation extracts the options that it recognizes and stores them in a general-purpose representation. Extensions using their own representation of source options (such as bit vectors) will override this method; the resource extension does not.

6.2.2 Extension-Provided Extension Source

If the extension source has special data members or some unique behavior that cannot be fully covered using the above scheme, then the extension must define its own subclass of ExtensionSource and provide a function pointer to its creator method in the ExtensionLoadPoint. Montana provides a mechanism that lets subclasses of ExtensionSource inherit implementations not exposed in the interface. Since C++ inheritance does not provide this capability, the Montana kernel implements this behavior via *delegation* [10, 15], which requires some careful analysis when written in C++.

6.2.3 Other Services

The most efficient design for an extension may require new nodes (in addition to the extension source) sprinkled throughout the dependency graph. To facilitate this, Montana provides the base class *ExtensionDependencyNode*. Derived classes provided by extension writers need only implement the update() method. Montana also provides "factories" (see e.g. [5] chapter 3) for creating kernel objects, like declarations. These factories are available to extensions, so, for example, an extension that does source code generation can add declaration objects directly to the CodeStore.

7 Discussion and Summary

This paper describes the three kernel extension mechanisms in Montana. To our knowledge, providing such extensive mechanisms for tightly integrating third party tools into a development environment is a novel feature of Montana. We should point out here that Montana's graphical user interface also provides an infrastructure for extensions, which lets tool writers integrate new views into the system. Writing the extension subsystem is challenging, since one has to predict the needs of future tool writers, and aim to provide services which are both generally useful and non-restrictive. Part of the Montana solution is to support several different extension paradigms.

Each of the three extension mechanisms has tasks for which it is particularly well suited. An extension writer may choose to employ more than one extension type to implement a particular tool. For example, a profiling tool may create a dependency node that performs some analysis based on information gathered by several observers, and display the results through a new UI view.

7.1 Requirements for Extension Writers

Each of the three mechanisms also poses different requirements on the extension writer. In all cases the extension writer should be familiar with the parts of the CodeStore that are relevant to the tool's tasks, and must provide the right kind of ExtensionLoadPoint for setting up the extension. Specific additional requirements for the three types are as follows.

observer extension: Familiarity with Montana's observer base classes and basic understanding of the incorporation sequence (in order to coordinate the different notifications) is required. The extension writer must provide the tool's observer classes and a startup function for registering them.

incorporation extension: Familiarity with the different phases of function-body and variable-initializer compilation interaction is required, as well as understanding invariants on the CodeStore function-body representation for the relevant phases. (For example, function bodies look very different after object-model transformation than they do after type analysis.) The extension writer must provide the tool's incorporation components, an Implementation-ComponentFactory for creating them, and a startup function for creating the factories.

dependency graph extension: Familiarity with change propagation and the different states of a dependency node (e.g. invalidated, removed) is required. Also, it is necessary to understand how the new extension nodes interact with the rest of the graph and how they should be updated in different situations (including error situations). The extension writer must provide implementations of the extension source as well as any other new nodes employed by the extension (as explained in Section 6.2).

7.2 Existing Montana Tools

A "generic" dependency graph extension has been written for loosely integrating existing tools (such as the resource compiler), and is provided as part of Montana. As part of its ".ice"-file specification, this extension takes the name of an external command (like make or the name of a parser generator) and names of configuration-specified options. This extension works by running the command on a specified extension source, during incorporation, whenever the source has changed.

An observer extension that instruments the preprocessor to build up views in the Montana GUI for macro expansions has also been written. One of the limitations of many current development environments for C and C++ is the lack of support for macros. The macro-expansion observer overcomes this limitation without compromising compilation time. After the program has been compiled, the Montana GUI reprocesses files on demand in order to build up macro-expansion views in the GUI. This observer illustrates how Montana supplies infrastructure for tools – large parts of Montana are reused to obtain this information after the compilation is complete.

A variety of different kinds of extensions have been implemented using the Montana extension scheme: A correct implementation of "smart pointers" has been implemented using an incorporation extension that adds reference-counting code to pointer uses (see [8]). An implementation of subject-oriented programming [11] has been implemented using these extensions, as well as a meta-object protocol[9] that Montana provides for object-model compilation. This extension for subject-oriented programming is quite sophisticated.

Aside from the object-model transformations, it comprises a fairly sophisticated source code generator, and several dependency graph extensions.

7.3 Future Work

There are a lot of mundane details that still have to be completed. For example, the registration mechanism for incorporation extensions does not give the extension writer enough control over when the extension is to be executed during incorporation. This is especially a problem if several incorporation extensions have to interact. Some sort of naming or prioritization mechanism has to be introduced.

The number of observers that have been provided has been limited. Instead of guessing all of the distinguished points for observation, we added observers as needed. There are probably hundreds of such points that can be defined. As an example, the Montana preprocessor has been completely instrumented, and contains about 50 observers.

Finally, it is clear that many more extensions have to be written in order to solidify the extension support that Montana provides. The system is sufficiently complicated, and extensions are sufficiently varied, that the extension support provided will no doubt be extended in the future.

8 Acknowledgements

A large number of people have contributed to the Montana project. You know who you are. Robert Bowdidge and Peri Tarr also provided much appreciated feedback on early drafts of this paper.

References

[1] John Barton, Y-Min Chee, Michael Karasick, Philippe Charles, Derek Lieber, and Lee Nackman. CodeStore: Infrastructure for C++ Knowledgeable Tools. *Proceedings of OOPSLA Workshop on Object-Oriented Compilation*, 1994.

[2] Craig Chambers, Jeffrey Dean, and David Grove. A Framework for Selective Recompilation in the Presence of Complex Intermodule Dependencies. In *Proceedings of the International Conference on Software Engineering*, Seattle, WA, 1995.

[3] Stuart I. Feldman. Make – a computer program from maintaining computer programs. *Software Practice and Experience*, 9(4):255–265, 1979.

[4] Michael Franz. Dynamic linking of Software Components. *IEEE Computer*, 30(3):74–81, 1997.

[5] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison Wesley, 1995.

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

[7] Nico A. Haberman and David Notkin. Gandalf: Software Development Environments. *Transactions of Software Engineering*, 12(12):1117–1127, 1986.

[8] Jennifer Hamilton. Montana Smart Pointers: They're Smart, and They're Pointers. submitted to COOTS 1997, 1997.

[9] Gregor Kiczales, Jim de Rivieres, and Donald G. Bobrow. *The Art of the Metaobject Protocol.* The MIT Press, 1991.

[10] Henry Lieberman. Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems. In *Proceedings of OOPSLA 86*, pages 214–223, Portland, September 1986. ACM.

[11] Harold Ossher, Matthew Kaplan, Alexander Katz, William Harrison, and Vincent Kruskal. Specifying Subject-Oriented Composition. *TAPOS*, 2(3):179–202, 1996. Special Issue on Subjectivity in Object-Oriented Systems.

[12] Zhong Shao and Andrew W. Appel. Smartest Recompilation. In *Proceedings of POPL 1993*, pages 439–450, Charleston, SC.

[13] Richard M. Stallman. EMACS: The Extensible, Customizable, Self-Documenting Display Editor. AI memo 519, MIT AI Lab, June 1979.

[14] Walter F. Tichy and Mark C. Baker. Smart Recompilation. In *Proceedings of POPL 1985*, pages 236–244, New Orleans, LA.

[15] David Ungar and Randall B. Smith. Self: The Power of Simplicity. In *Proceedings of OOPSLA 87*, pages 227–242, Orlando, October 1987. ACM.

[16] Beata Winnicka, Dennis Gannon, and Francois Bodin. C++ Objects for Representing and Manipulating Program Trees in the Sage++ System. *Proceedings of OOPSLA Workshop on Object-Oriented Compilation*, 1994.