Paradyn Parallel Performance Tools

Dyninst Programmer's Guide

Release 9.2
June 2016

Computer Science Department
University of Wisconsin-Madison
Madison, WI 53706

Computer Science Department
University of Maryland
College Park, MD 20742
Email: dyninst-api@cs.wisc.edu
Web: www.dyninst.org
github.com/dyninst/dyninst

1. Introduction

The normal cycle of developing a program is to edit the source code, compile it, and then execute the resulting binary. However, sometimes this cycle can be too restrictive. We may wish to change the program while it is executing or after it has been linked, thus avoiding the process of re-compiling, re-linking, or even re-executing the program to change the binary. At first, this may seem like a bizarre goal, however, there are several practical reasons why we may wish to have such a system. For example, if we are measuring the performance of a program and discover a performance problem, it might be necessary to insert additional instrumentation into the program to understand the problem. Another application is performance steering; for large simulations, computational scientists often find it advantageous to be able to make modifications to the code and data while the simulation is executing.

This document describes an Application Program Interface (API) to permit the insertion of code into a computer application that is either running or on disk. The API for inserting code into a running application, called dynamic instrumentation, shares much of the same structure as the API for inserting code into an executable file or library, known as static instrumentation. The API also permits changing or removing subroutine calls from the application program. Binary code changes are useful to support a variety of applications including debugging, performance monitoring, and to support composing applications out of existing packages. The goal of this API is to provide a machine independent interface to permit the creation of tools and applications that use runtime and static code patching. The API and a simple test application are described in [1]. This API is based on the idea of dynamic instrumentation described in [3].

The key features of this interface are the abilities to:

- Insert and change instrumentation in a running program.
- Insert instrumentation into a binary on disk and write a new copy of that binary back to disk.
- Perform static and dynamic analysis on binaries and processes.

The goal of this API is to keep the interface small and easy to understand. At the same time, it needs to be sufficiently expressive to be useful for a variety of applications. We accomplished this goal by providing a simple set of abstractions and a way to specify which code to insert into the application[1].

[1] To generate more complex code, extra (initially un-called) subroutines can be linked into the application program, and calls to these subroutines can be inserted at runtime via this interface.

2. Abstractions

The DyninstAPI library provides an interface for instrumenting and working with binaries and processes. The user writes a *mutator*, which uses the DyninstAPI library to operate on the application. The process that contains the *mutator* and DyninstAPI library is known as the *mutator process*. The *mutator process* operates on other processes or on-disk binaries, which are known as *mutatees*.

The API is based on abstractions of a program. For dynamic instrumentation, it can be based on the state while in execution. The two primary abstractions in the API are *points* and *snippets*. A *point* is a location in a program where instrumentation can be inserted. A *snippet* is a representation of some executable code to be inserted into a program at a point. For example, if we wished to record the number of times a procedure was invoked, the *point* would be entry point of the procedure, and the *snippets* would be a statement to increment a counter. *Snippets* can include conditionals and function calls.

Mutatees are represented using an *address space* abstraction. For dynamic instrumentation, the *address space* represents a process and includes any dynamic libraries loaded with the process. For static instrumentation, the *address space* includes a disk executable and includes any dynamic library files on which the executable depends. The *address space* abstraction is extended by *process* and *binary* abstractions for dynamic and static instrumentation. The *process* abstraction represents information about a running process such as threads or stack state. The *binary* abstraction represents information about a binary found on disk.

The code and data represented by an *address space* is broken up into *function* and *variable* abstractions. *Function*s contain *points*, which specify locations to insert instrumentation. *Functions* also contain a *control flow graph* abstraction, which contains information about *basic blocks*, *edges*, *loops*, and *instructions*. If the *mutatee* contains debug information, DyninstAPI will also provide abstractions about variable and function *types*, *local variables, function parameters*, and *source code line information*. The collection of *functions* and *variables* in a mutatee is represented as an *image*.

The API includes a simple type system based on structural equivalence. If mutatee programs have been compiled with debugging symbols and the symbols are in a format that Dyninst understands, type checking is performed on code to be inserted into the mutatee. See Section 4.28 for a complete description of the type system.

Due to language constructs or compiler optimizations, it may be possible for multiple functions to *overlap* (that is, share part of the same function body) or for a single function to have multiple *entry points*. In practice, it is impossible to determine the difference between multiple overlapping functions and a single function with multiple entry points. The DyninstAPI uses a model where each function (BPatch_function object) has a single entry point, and multiple functions may overlap (share code). We guarantee that instrumentation inserted in a particular function is only executed in the context of that function, even if instrumentation is inserted into a location that exists in multiple functions.

3. Examples

To illustrate the ideas of the API, we present several short examples that demonstrate how the API can be used. The full details of the interface are presented in the next section. To prevent confusion, we refer to the application process or binary that is being modified as the mutatee, and the program that uses the API to modify the application as the mutator. The mutator is a separate process from the application process.

The examples in this section are simple code snippets, not complete programs. Appendix A - Complete Examples provides several examples of complete Dyninst programs.

3.1 Instrumenting a function

A mutator program must create a single instance of the class `BPatch`. This object is used to access functions and information that are global to the library. It must not be destroyed until the mutator has completely finished using the library. For this example, we assume that the mutator program has declared a global variable called `bpatch` of class `BPatch`.

All instrumentation is done with a `BPatch_addressSpace` object, which allows us to write codes that work for both dynamic and static instrumentation. During initialization we use either `BPatch_process` to attach to or create a process, or `BPatch_binaryEdit` to open a file on disk. When instrumentation is completed, we will either run the `BPatch_process`, or write the `BPatch_binaryEdit` back onto the disk.

The mutator first needs to identify the application to be modified. If the process is already in execution, this can be done by specifying the executable file name and process id of the application as arguments in order to create an instance of a process object:

```
BPatch_process *appProc = bpatch.processAttach(name, processId);
```

This creates a new instance of the `BPatch_process` class that refers to the existing process. It had no effect on the state of the process (i.e., running or stopped). If the process has not been started, the mutator specifies the pathname and argument list of a program it seeks to execute:

```
BPatch_process *appProc = bpatch.processCreate(pathname, argv);
```

If the mutator is opening a file for static binary rewriting, it executes:

```
BPatch_binaryEdit *appBin = bpatch.openBinary(pathname);
```

The above statements create either a `BPatch_process` object or `BPatch_binaryEdit` object, depending on whether Dyninst is doing dynamic or static instrumentation. The instrumentation and analysis code can be made agnostic towards static or dynamic modes by using a `BPatch_addressSpace` object. Both `BPatch_process` and `BPatch_binaryEdit` inherit from `BPatch_addressSpace`, so we can use cast operations to move between the two:

```
BPatch_process *appProc = static_cast<BPatch_process *>(appAddrSpace)
-or-
BPatch_binaryEdit *appBin = static_cast<BPatch_binaryEdit *>(appAddrSpace)
```

Similarly, all instrumentation commands can be performed on a `BPatch_addressSpace` object, allowing similar codes to be used between dynamic instrumentation and binary rewriting:

```
BPatch_addressSpace *app = appProc;
-or-
BPatch_addressSpace *app = appBin;
```

Once the address space has been created, the mutator defines the snippet of code to be inserted and identifies where the points should be inserted.

If the mutator wants to instrument the entry point of `InterestingProcedure`, it should get a `BPatch_function` from the application's `BPatch_image`, and get the entry `BPatch_point` from that function:

```
std::vector<BPatch_function *> functions;
std::vector<BPatch_point *> *points;

BPatch_image *appImage = app->getImage();
appImage->findFunction("InterestingProcedure", functions);
points = functions[0]->findPoint(BPatch_locEntry);
```

The mutator also needs to construct the instrumentation that it will insert at the `BPatch_point`. It can do this by allocating an integer in the application to store instrumentation results, and then creating a `BPatch_snippet` to increment that integer:

```
BPatch_variableExpr *intCounter =
    app->malloc(*(appImage->findType("int")));

BPatch_arithExpr addOne(BPatch_assign, *intCounter,
        BPatch_arithExpr(BPatch_plus, *intCounter, BPatch_constExpr(1)));
```

The mutator can set the `BPatch_snippet` to be run at the `BPatch_point` by executing an `insertSnippet` call:

```
app->insertSnippet(addOne, *points);
```

Finally, the mutator should either continue the mutate process and wait for it to finish, or write the resulting binary onto the disk, depending on whether it is doing dynamic or static instrumentation:

```
appProc->continueExecution();
while (!appProc->isTerminated()) {
    bpatch.waitForStatusChange();
}
-or-
appBin->writeFile(newPath);
```

A complete example can be found in Appendix A - Complete Examples.

3.2 Binary Analysis

This example will illustrate how to use Dyninst to iterate over a function's control flow graph and inspect instructions. These are steps that would usually be part of a larger data flow or control flow analysis. Specifically, this example will collect every basic block in a function, iterate over them, and count the number of instructions that access memory.

Unlike the previous instrumentation example, this example will analyze a binary file on disk. Bear in mind, these techniques can also be applied when working with processes. This example makes use of InstructionAPI, details of which can be found in the InstructionAPI Reference Manual.

Similar to the above example, the mutator will start by creating a BPatch object and opening a file to operate on:

```
BPatch bpatch;
BPatch_binaryEdit *binedit = bpatch.openFile(pathname);
```

The mutator needs to get a handle to a function to do analysis on. This example will look up a function by name; alternatively, it could have iterated over every function in `BPatch_image` or `BPatch_module`:

```
BPatch_image *appImage = binedit->getImage();

std::vector<BPatch_function *> funcs;
image->findFunction("InterestingProcedure", funcs);
```

A function's control flow graph is represented by the `BPatch_flowGraph` class. The `BPatch_flowGraph` contains, among other things, a set of `BPatch_basicBlock` objects connected by `BPatch_edge` objects. This example will simply collect a list of the basic blocks in `BPatch_flowGraph` and iterate over each one:

```
BPatch_flowGraph *fg = funcs[0]->getCFG();

std::set<BPatch_basicBlock *> blocks;
fg->getAllBasicBlocks(blocks);
```

Each basic block has a list of instructions. Each instruction is represented by a `Dyninst::InstructionAPI::Instruction::Ptr` object.

```
std::set<BPatch_basicBlock *>::iterator block_iter;
for (block_iter = blocks.begin(); block_iter != blocks.end(); ++block_iter)
{
    BPatch_basicBlock *block = *block_iter;
    std::vector<Dyninst::InstructionAPI::Instruction::Ptr> insns;
    block->getInstructions(insns);
}
```

Given an `Instruction` object, which is described in the InstructionAPI Reference Manual, we can query for properties of this instruction. InstructionAPI has numerous methods for inspecting the memory accesses, registers, and other properties of an instruction. This example simply checks whether this instruction accesses memory:

```
std::vector<Dyninst::InstructionAPI::Instruction::Ptr>::iterator
          insn_iter;
for (insn_iter = insns.begin(); insn_iter != insns.end(); ++insn_iter)
{
    Dyninst::InstructionAPI::Instruction::Ptr insn = *insn_iter;
    if (insn->readsMemory() || insn->writesMemory()) {
        insns_access_memory++;
    }
}
```

3.3 Instrumenting Memory Accesses

There are two snippets useful for memory access instrumentation: `BPatch_effectiveAddressExpr` and `BPatch_bytesAccessedExpr`. Both have nullary constructors; the result of the snippet depends on the instrumentation point where the snippet is inserted. `BPatch_effectiveAddressExpr` has type void*, while `BPatch_bytesAccessedExpr` has type int.

These snippets may be used to instrument a given instrumentation point if and only if the point has memory access information attached to it. In this release the only way to create instrumentation points that have memory access information attached is via `BPatch_function.findPoint(const std::set<BPatch_opCode>&)`. For example, to instrument all the loads and stores in a function named `InterestingProcedure` with a call to `printf`, one may write:

```
BPatch_addressSpace *app = ...;
BPatch_image *appImage = proc->getImage();

// We're interested in loads and stores
std::set<BPatch_opCode> axs;
axs.insert(BPatch_opLoad);
axs.insert(BPatch_opStore);

// Scan the function InterestingProcedure and create instrumentation points
std::vector<BPatch_function*> funcs;
appImage->findFunction("InterestingProcedure", funcs);
std::vector<BPatch_point*>* points = funcs[0]->findPoint(axs);

// Create the printf function call snippet
std::vector<BPatch_snippet*> printfArgs;
BPatch_snippet *fmt = new BPatch_constExpr("Access at: %p.\n");
printfArgs.push_back(fmt);
BPatch_snippet *eae = new BPatch_effectiveAddressExpr();
printfArgs.push_back(eae);

// Find the printf function
std::vector<BPatch_function *> printfFuncs;
appImage->findFunction("printf", printfFuncs);

// Construct the function call snippet
BPatch_funcCallExpr printfCall(*(printfFuncs[0]), printfArgs);

// Insert the snippet at the instrumentation points
app->insertSnippet(printfCall, *points);
```

4. Interface

This section describes functions in the API. The API is organized as a collection of C++ classes. The primary classes are `BPatch`, `Bpatch_process`, `BPatch_binaryEdit`, `BPatch_thread`, `BPatch_image`, `BPatch_point`, and `BPatch_snippet`. The API also uses a template class called `std::vector`. This class is based on the Standard Template Library (STL) vector class.

4.1 Class BPatch

The **BPatch** class represents the entire Dyninst library. There can only be one instance of this class at a time. This class is used to perform functions and obtain information that is not specific to a particular thread or image.

std::vector<BPatch_process*> *getProcesses()

Returns the list of processes that are currently defined. This list includes processes that were directly created by calling processCreate/processAttach, and indirectly by the UNIX `fork` or the Windows `CreateProcess` system call. It is up to the user to delete this vector when they are done with it.

BPatch_process *processAttach(const char *path, int pid, BPatch_hybridMode mode=BPatch_normalMode)
BPatch_process *processCreate(const char *path, const char *argv[], const char **envp = NULL, int stdin_fd=0, int stdout_fd=1, int stderr_fd=2, BPatch_hybridMode mode=BPatch_normalMode)

Each of these functions returns a pointer to a new instance of the `BPatch_process` class. The `path` parameter needed by these functions should be the pathname of the executable file containing the process image. The `processAttach` function returns a `BPatch_process` associated with an existing process. On Linux platforms the path parameter can be `NULL` since the executable image can be derived from the process pid. Attaching to a process puts it into the stopped state. The `processCreate` function creates a new process and returns a new `BPatch_process` associated with it. The new process is put into a stopped state before executing any code.

The `stdin_fd`, `stdout_fd`, and `stderr_fd` parameters are used to set the standard input, output, and error of the child process. The default values of these parameters leave the input, output, and error to be the same as the mutator process. To change these values, an open UNIX file descriptor (see `open(1)`) can be passed.

The `mode` parameter is used to select the desired level of code analysis. Activating hybrid code analysis causes Dyninst to augment its static analysis of the code with run-time code discovery techniques. There are three modes: `BPatch_normalMode`, BPatch_exploratoryMode, and BPatch_defensiveMode. Normal mode enables the regular static analysis features of Dyninst. Exploratory mode and defensive mode enable additional dynamic features to correctly analyze programs that contain uncommon code patterns, such as malware. Exploratory mode is primarily oriented towards analyzing dy-

namic control transfers, while defensive mode additionally aims to tackle code obfuscation and self-modifying code. Both of these modes are still experimental and should be used with caution. Defensive mode is only supported on Windows.

Defensive mode has been tested on normal binaries (binaries that run correctly under normal mode), as well as some simple, packed executables (self-decrypting or decompressing). More advanced forms of code obfuscation, such as self-modifying code, have not been tested recently. The traditional Dyninst interface may be used for instrumentation of binaries in defensive mode, but in the case of highly obfuscated code, this interface may prove to be ineffective due to the lack of a complete view of control flow at any given point. Therefore, defensive mode also includes a set of callbacks that enables instrumentation to be performed as new code is discovered. Due to the fact that recent efforts have focused on simpler forms of obfuscation, these callbacks have not been tested in detail. The next release of Dyninst will target more advanced uses of defensive mode.

BPatch_binaryEdit *openBinary(const char *path,
bool openDependencies = false)

This function opens the executable file or library file pointed to by `path` for binary rewriting. If `openDependencies` is true then Dyninst will also open all shared libraries that `path` depends on. Upon success, this function returns a new instance of a `BPatch_binaryEdit` class that represents the opened file and any dependent shared libraries. This function returns `NULL` in the event of an error.

bool pollForStatusChange()

This is useful for a mutator that needs to periodically check on the status of its managed threads and does not want to check each process individually. It returns `true` if there has been a change in the status of one or more threads that has not yet been reported by either `isStopped` or `isTerminated`.

void setDebugParsing (bool state)

Turn on or off the parsing of debugger information. By default, the debugger information (produced by the –g compiler option) is parsed on those platforms that support it. However, for some applications this information can be quite large. To disable parsing this information, call this method with a value of `false` prior to creating a process.

bool parseDebugInfo()

Return true if debugger information parsing is enabled, or `false` otherwise.

void setTrampRecursive (bool state)

Turn on or off trampoline recursion. By default, any snippets invoked while another snippet is active will not be executed. This is the safest behavior, since recursively-calling snippets can cause a program to take up all available system resources and die.

For example, adding instrumentation code to the start of printf, and then calling printf from that snippet will result in infinite recursion.

This protection operates at the granularity of an instrumentation point. When snippets are first inserted at a point, this flag determines whether code will be created with recursion protection. Changing the flag is **not** retroactive, and inserting more snippets will not change the recursion protection of the point. Recursion protection increases the overhead of instrumentation points, so if there is no way for the snippets to call themselves, calling this method with the parameter `true` will result in a performance gain. The default value of this flag is `false`.

bool isTrampRecursive ()

Return whether trampoline recursion is enabled or not. `True` means that it is enabled.

void setTypeChecking(bool state)

Turn on or off type-checking of snippets. By default type-checking is turned on, and an attempt to create a snippet that contains type conflicts will fail. Any snippet expressions created with type-checking off have the type of their left operand. Turning type-checking off, creating a snippet, and then turning type-checking back on is similar to the type cast operation in the C programming language.

bool isTypeChecked()

Return `true` if type-checking of snippets is enabled, or `false` otherwise.

bool waitForStatusChange()

This function waits until there is a status change to some thread that has not yet been reported by either `isStopped` or `isTerminated`, and then returns true. It is more efficient to call this function than to call `pollForStatusChange` in a loop, because `waitForStatusChange` blocks the mutator process while waiting.

void setDelayedParsing (bool)

Turn on or off delayed parsing. When it is activated Dyninst will initially parse only the symbol table information in any new modules loaded by the program, and will postpone more thorough analysis (instrumentation point analysis, variable analysis, and discovery of new functions in stripped binaries). This analysis will automatically occur when the information is necessary.

Users which require small run-time perturbation of a program should not delay parsing; the overhead for analysis may occur at unexpected times if it is triggered by internal Dyninst behavior. Users who desire instrumentation of a small number of functions will benefit from delayed parsing.

bool delayedParsingOn()

Return `true` if delayed parsing is enabled, or `false` otherwise.

void setInstrStackFrames(bool)

Turn on and off stack frames in instrumentation. When on, Dyninst will create stack frames around instrumentation. A stack frame allows Dyninst or other tools to walk a call stack through instrumentation, but introduces overhead to instrumentation. The default is to not create stack frames.

bool getInstrStackFrames()

Return true if instrumentation will create stack frames, or `false` otherwise.

void setMergeTramp (bool)

Turn on or off inlined tramps. Setting this value to true will make each base trampoline have all of its mini-trampolines inlined within it. Using inlined mini-tramps may allow instrumentation to execute faster, but inserting and removing instrumentation may take more time. The default setting for this is `true`.

bool isMergeTramp ()

This returns the current status of inlined trampolines. A value of `true` indicates that trampolines are inlined.

void setSaveFPR (bool)

Turn on or off floating point saves. Setting this value to `false` means that floating point registers will never be saved, which can lead to large performance improvements. The default value is `true`. Setting this flag may cause incorrect program behavior if the instrumentation does clobber floating point registers, so it should only be used when the user is positive this will never happen.

bool isSaveFPROn ()

This returns the current status of the floating point saves. `True` means we are saving floating points based on the analysis for the given platform.

void setBaseTrampDeletion(bool)

If true, we delete the base tramp when the last corresponding minitramp is deleted. If false, we leave the base tramp in. The default value is `false`.

bool baseTrampDeletion()

Return true if base trampolines are set to be deleted, or `false` otherwise.

void setLivenessAnalysis(bool)

If true, we perform register liveness analysis around an `instPoint` before inserting instrumentation, and we only save registers that are live at that point. This can lead to fast-

er run-time speeds, but at the expense of slower instrumentation time. The default value is `true`.

bool livenessAnalysisOn()

Return true if liveness analysis is currently enabled.

void getBPatchVersion(int &major, int &minor, int &subminor)

Return Dyninst's version number. The major version number will be stored in `major`, the minor version number in `minor`, and the subminor version in `subminor`. For example, under Dyninst 5.1.0, this function will return 5 in `major`, 1 in `minor`, and 0 in `subminor`.

int getNotificationFD()

Returns a file descriptor that is suitable for inclusion in a call to `select()`. Dyninst will write data to this file descriptor when it to signal a state change in the process. `BPatch::pollForStatusChange` should then be called so that Dyninst can handle the state change. This is useful for applications where the user does not want to block in `BPatch::waitForStatusChange`. The file descriptor will reset when the user calls `BPatch::pollForStatusChange`.

BPatch_type *createArray(const char *name, BPatch_type *ptr, unsigned int low, unsigned int hi)

Create a new array type. The name of the type is `name`, and the type of each element is `ptr`. The index of the first element of the array is `low`, and the last is `high`. The standard rules of type compatibility, described in Section 4.28, are used with arrays created using this function.

BPatch_type *createEnum(const char *name, std::vector<char *> &elementNames, std::vector<int> &elementIds)
BPatch_type *createEnum(const char *name, std::vector<char *> &elementNames)

Create a new enumerated type. There are two variations of this function. The first one is used to create an enumerated type where the user specifies the identifier (int) for each element. In the second form, the system specifies the identifiers for each element. In both cases, a vector of character arrays is passed to supply the names of the elements of the enumerated type. In the first form of the function, the number of element in the `elementNames` and `elementIds` vectors must be the same, or the type will not be created and this function will return `NULL`. The standard rules of type compatibility, described in Section 4.28, are used with enums created using this function.

BPatch_type *createScalar(const char *name, int size)

Create a new scalar type. The `name` field is used to specify the name of the type, and the `size` parameter is used to specify the size in bytes of each instance of the type. No additional information about this type is supplied. The type is compatible with other scalars with the same name and size.

BPatch_type *createStruct(const char *name, std::vector<char *> &fieldNames, std::vector<BPatch_type *> &fieldTypes)

Create a new structure type. The name of the structure is specified in the `name` parameter. The `fieldNames` and `fieldTypes` vectors specify fields of the type. These two vectors must have the same number of elements or the function will fail (and return `NULL`). The standard rules of type compatibility, described in Section 4.28, are used with structures created using this function. The size of the structure is the sum of the size of the elements in the `fieldTypes` vector.

BPatch_type *createTypedef(const char *name, BPatch_type *ptr)

Create a new type called `name` and having the type `ptr`.

BPatch_type *createPointer(const char *name, BPatch_type *ptr)
BPatch_type *createPointer(const char *name, BPatch_type *ptr, int size)

Create a new type, named `name`, which points to objects of type `ptr`. The first form creates a pointer whose size is equal to `sizeof(void*)` on the target platform where the mutatee is running. In the second form, the size of the pointer is the value passed in the `size` parameter.

BPatch_type *createUnion(const char *name, std::vector<char *> &fieldNames, std::vector<BPatch_type *> &fieldTypes)

Create a new union type. The name of the union is specified in the `name` parameter. The `fieldNames` and `fieldTypes` vectors specify fields of the type. These two vectors must have the same number of elements or the function will fail (and return `NULL`). The size of the union is the size of the largest element in the `fieldTypes` vector.

4.2 Callbacks

The following functions are intended as a way for API users to be informed when an error or significant event occurs. Each function allows a user to register a handler for an event. The return code for all callback registration functions is the address of the handler that was previously registered (which may be `NULL` if no handler was previously registered). For backwards compatibility reasons, some callbacks may pass a `BPatch_thread` object when a `BPatch_process` may be more appropriate. A `BPatch_thread` may be converted into a `BPatch_process` using `BPatch_thread::getProcess()`.

4.2.1 Asynchronous Callbacks

```
typedef void (*BPatchAsyncThreadEventCallback)(
        BPatch_process *proc, BPatch_thread *thread)
bool registerThreadEventCallback(BPatch_asyncEventType type,
        BPatchAsyncThreadEventCallback cb)
bool removeThreadEventCallback(BPatch_asyncEventType type,
        BPatch_AsyncThreadEventCallback cb)
```

The type parameter can be either one of `BPatch_threadCreateEvent` or `BPatch_threadDestroyEvent`. Different callbacks can be registered for different values of `type`.

4.2.2 Code Discovery Callbacks

```
typedef void (*BPatchCodeDiscoveryCallback)( BPatch_Vector<BPatch_function*> &newFuncs,
        BPatch_Vector<BPatch_function*> &modFuncs)
bool registerCodeDiscoveryCallback(
        BPatchCodeDiscoveryCallback cb)
bool removeCodeDiscoveryCallback(BPatchCodeDiscoveryCallback cb)
```

This callback is invoked whenever previously un-analyzed code is discovered through runtime analysis, and delivers a vector of functions whose analysis have been modified and a vector of functions that are newly discovered.

4.2.3 Code Overwrite Callbacks

```
typedef void (*BPatchCodeOverwriteBeginCallback)(
        BPatch_Vector<BPatch_basicBlock*> &overwriteLoopBlocks);
typedef void (*BPatchCodeOverwriteEndCallback)(
        BPatch_Vector<std::pair<Dyninst::Address,int> > &deadBlocks, BPatch_Vector<BPatch_function*>
        &owFuncs, BPatch_Vector<BPatch_function*> &modFuncs, BPatch_Vector<BPatch_function*>
        &newFuncs)
bool registerCodeOverwriteCallbacks(
        BPatchCodeOverwriteBeginCallback cbBegin, BPatchCodeOverwriteEndCallback cbEnd)
```

Register a callback at the beginning and end of overwrite events. Only invoke if Dyninst's hybrid analysis mode is set to BPatch_defensiveMode.

The `BPatchCodeOverwriteBeginCallback` callback allows the user to remove any instrumentation when the program starts writing to a code page, which may be desirable as instrumentation cannot be removed during the overwrite loop's execution, and any breakpoint instrumentation will dramatically slow the loop's execution.

The `BPatchCodeOverwriteEndCallback` callback delivers the effects of the overwrite loop when it is done executing. In many cases no code will have changed.

4.2.4 Dynamic calls

```
typedef void (*BPatchDynamicCallSiteCallback)(
        BPatch_point *at_point, BPatch_function *called_function);
bool registerDynamicCallCallback(BPatchDynamicCallSiteCallback cb);
bool removeDynamicCallCallback(BPatchDynamicCallSiteCallback cb);
```

4.2.5 Dynamic libraries

```
typedef void (*BPatchDynLibraryCallback)(BPatch_thread *thr,
        BPatch_object *obj, bool loaded);
BPatchDynLibraryCallback registerDynLibraryCallback(
        BPatchDynLibraryCallback func)
```

Note that in versions previous to 9.1, BPatchDynLibraryCallback's signature took a BPatch_module instead of a BPatch_object.

4.2.6 Errors

```
enum BPatchErrorLevel { BPatchFatal, BPatchSerious, BPatchWarning, BPatchInfo };
typedef void (*BPatchErrorCallback)(BPatchErrorLevel severity, int number, const char * const *params)
BPatchErrorCallback registerErrorCallback(BPatchErrorCallback func)
```

This function registers the error callback function with the `BPatch` class. The return value is the address of the previous error callback function. Dyninst users can change the error callback during program execution (e.g., one error callback before a GUI is initialized, and a different one after). The `severity` field indicates how important the error is (from fatal to information/status). The `number` is a unique number that identifies this error message. `Params` are the parameters that describe the detail about an error, e.g., the process id where the error occurred. The number and meaning of params depends on the error. However, for a given error number the number of parameters returned will always be the same.

4.2.7 Exec

```
typedef void (*BPatchExecCallback)(BPatch_thread *thr)
BPatchExecCallback registerExecCallback(
        BPatchExecCallback func)  Not implemented on Windows.
```

4.2.8 Exit

```
typedef enum BPatch_exitType { NoExit, ExitedNormally, ExitedViaSignal };
typedef void (*BPatchExitCallback)(BPatch_thread *proc, BPatch_exitType exit_type);
BPatchExitCallback registerExitCallback(
        BPatchExitCallback func)
```

Register a function to be called when a process terminates. For a normal process exit, the callback will actually be called just before the process exits, but while its process state still exists. This allows final actions to be taken on the process before it actually exits. The function `BPatch_thread::isTerminated()` will return true in this context even though the process hasn't yet actually exited. In the case of an exit due to a signal, the process will have already exited.

4.2.9 Fork

```
typedef void (*BPatchForkCallback)(BPatch_thread *parent, BPatch_thread *child);
```

This is the prototype for the pre-`fork` and post-`fork` callbacks. The `parent` parameter is the parent thread, and the `child` parameter is a `BPatch_thread` in the newly created process. When invoked as a pre-`fork` callback, the child is `NULL`.

```
BPatchForkCallback registerPreForkCallback(
        BPatchForkCallback func) not implemented on Windows
BPatchForkCallback registerPostForkCallback(
        BPatchForkCallback func) not implemented on Windows
```

Register callbacks for pre-fork (before the child is created) and post-fork (immediately after the child is created). When a pre-fork callback is executed the `child` parameter will be `NULL`.

4.2.10 One Time Code

```
typedef void (*BPatchOneTimeCodeCallback)(Bpatch_thread *thr,
        void *userData, void *returnValue);
BPatchOneTimeCodeCallback registerOneTimeCodeCallback(
        BPatchOneTimeCodeCallback func)
```

The `thr` field contains the thread that executed the `oneTimeCode` (if thread-specific) or an unspecified thread in the process (if process-wide). The `userData` field contains the value passed to the `oneTimeCode` call. The `returnValue` field contains the return result of the `oneTimeCode` snippet.

4.2.11 Signal Handler

```
typedef void (*BPatchSignalHandlerCallback)(BPatch_point *at_point, long signum, std::vector<Dyninst::Address>
        *handlers)
bool registerSignalHandlerCallback(BPatchSignalHandlerCallback cb, std::set<long> &signal_numbers)
bool registerSignalHandlerCallback(BPatchSignalHandlerCallback cb, BPatch_Set<long> *signal_numbers)
bool removeSignalHandlerCallback(BPatchSignalHandlerCallback cb);
```

This function registers the signal handler callback function with the `BPatch` class. The return value indicates success or failure. The `signal_numbers` set contains those signal numbers for which the callback will be invoked.

The `at_point` parameter indicates the point at which the signal/exception was raised, `signum` is the number of the signal/exception that was raised, and the `handlers` vector contains any registered handler(s) for the signal/exception. In Windows this corresponds to the stack of Structured Exception Handlers, while for Unix systems there will be at most one registered exception handler. This functionality is only fully implemented for the Windows platform.

4.2.12 Stopped Threads

typedef void (*BPatchStopThreadCallback)(BPatch_point *at_point, void *returnValue)

This is the prototype for the callback that is associated with the `stopThreadExpr` snippet class (see Section 4.13). Unlike the other callbacks in this section, `stopThreadExpr` callbacks are registered during the creation of the `stopThreadExpr` snippet type. Whenever a `stopThreadExpr` snippet executes in a given thread, the snippet evaluates the `calculation` snippet that `stopThreadExpr` takes as a parameter, stops the thread's execution and invokes this callback. The `at_point` parameter is the `BPatch_point` at which the `stopThreadExpr` snippet was inserted, and `returnValue` contains the computation made by the calculation snippet.

4.2.13 User-triggered callbacks

typedef void (*BPatchUserEventCallback)(BPatch_process *proc, void *buf, unsigned int bufsize);
bool registerUserEventCallback(BPatchUserEventCallback cb)
bool removeUserEventCallback(BPatchUserEventCallback cb)

Register a callback that is executed when the user sends a message from the mutatee using the `DYNINSTuserMessage` function in the runtime library.

4.3 Class BPatch_addressSpace

The **BPatch_addressSpace** class is a superclass of the `BPatch_process` and BPatch_binaryEdit classes. It contains functionality that is common between the two sub classes.

BPatch_image *getImage()

Return a handle to the executable file associated with this `BPatch_process` object.

bool getSourceLines(unsigned long addr, std::vector< BPatch_statement > & lines)

This function returns the line information associated with the mutatee address, `addr`. The vector `lines` contain pairs of filenames and line numbers that are associated with `addr`. In many cases only one filename and line number is associated with an address, but certain compiler optimizations may lead to multiple filenames and lines at an address. This information is only available if the mutatee was compiled with debug information.

This function returns `true` if it was able to find any line information at `addr`, or false otherwise.

bool getAddressRanges(const char * fileName, unsigned int lineNo, std::vector< std::pair< unsigned long, unsigned long > > & ranges)

Given a filename and line number, `fileName` and `lineNo`, this function this function returns the ranges of mutatee addresses that implement the code range in the output parameter `ranges`. In many cases a source code line will only have one address range implementing it. However, compiler optimizations may transform this into multiple disjoint

address ranges. This information is only available if the mutatee was compiled with debug information.

This function returns `true` if it was able to find any line information, `false` otherwise.

BPatch_variableExpr *malloc(int n,
std::string name = std::string(“”))

BPatch_variableExpr *malloc(const BPatch_type &type,
std::string name = std::string(“”))

These two functions allocate memory. Memory allocation is from a heap. The heap is not necessarily the same heap used by the application. The available space in the heap may be limited depending on the implementation. The first function, `malloc(int n)`, allocates n bytes of memory from the heap. The second function, `malloc(const BPatch_type& t)`, allocates enough memory to hold an object of the specified type. Using the second version is strongly encouraged because it provides additional information to permit better type checking of the passed code. If a name is specified, Dyninst will assign `var_name` to the variable; otherwise, it will assign an internal name. The returned memory is persistent and will not be released until `BPatch_process::free` is called or the application terminates.

BPatch_variableExpr *createVariable(Dyninst::Address addr,
BPatch_type *type,
std::string var_name = std::string(“”),
BPatch_module *in_module = NULL)

This method creates a new variable at the given address `addr` in the module `in_module`. If a name is specified, Dyninst will assign `var_name` to the variable; otherwise, it will assign an internal name. The `type` parameter will become the type for the new variable.

When operating in binary rewriting mode, it is an error for the `in_module` parameter to be `NULL`; it is necessary to specify the module in which the variable will be created. Dyninst will then write the variable back out in the file specified by `in_module`.

bool free(BPatch_variableExpr &ptr)

Free the memory in the passed variable `ptr`. The programmer is responsible for verifying that all code that could reference this memory will not execute again (either by removing all snippets that refer to it, or by analysis of the program). Return true if the free succeeded.

bool getRegisters(std::vector<BPatch_register> &regs)

This function returns a vector of `BPatch_register` objects that represent registers available to snippet code.

BPatchSnippetHandle *insertSnippet(const BPatch_snippet &expr,
BPatch_point &point,
BPatch_callWhen when=[BPatch_callBefore| BPatch_callAfter],
BPatch_snippetOrder order = BPatch_firstSnippet)

BPatchSnippetHandle *insertSnippet(const BPatch_snippet &expr,
const std::vector<BPatch_point *> &points,
BPatch_callWhen when=[BPatch_callBefore| BPatch_callAfter],
BPatch_snippetOrder order = BPatch_firstSnippet)

Insert a snippet of code at the specified `point`. If a list of `points` is supplied, insert the code snippet at each point in the list. The optional `when` argument specifies when the snippet is to be called; a value of `BPatch_callBefore` indicates that the snippet should be inserted just before the specified `point` or `points` in the code, and a value of `BPatch_callAfter` indicates that it should be inserted just after them.

The `order` argument specifies where the snippet is to be inserted relative to any other snippets previously inserted at the same point. The values `BPatch_firstSnippet` and `BPatch_lastSnippet` indicate that the snippet should be inserted before or after all snippets, respectively.

It is illegal to use `BPatch_callAfter` with a `BPatch_entry` point. Use `BPatch_callBefore` when instrumenting entry points, which inserts instrumentation before the first instruction in a subroutine. Likewise, it is illegal to use `BPatch_callBefore` with a `BPatch_exit` point. Use `BPatch_callAfter` with exit points. `BPatch_callAfter` inserts instrumentation at the last instruction in the subroutine. `insertSnippet` will return `NULL` when used with an illegal pair of points.

bool deleteSnippet(BPatchSnippetHandle *handle)

Remove the snippet associated with the passed `handle`. If the `handle` is not defined for the process, then `deleteSnippet` will return false.

void beginInsertionSet()

Normally, a call to `insertSnippet` immediately injects instrumentation into the mutatee. However, users may wish to insert a set of snippets as a single batch operation. This provides two benefits: First, Dyninst may insert instrumentation in a more efficient manner. Second, multiple snippets may be inserted at multiple points as a single operation, with either all snippets being inserted successfully or none. This batch insertion mode is begun with a call to `beginInsertionSet;` after this call, no snippets are actually inserted until a corresponding call to `finalizeInsertionSet`. Dyninst accumulates all calls to `insertSnippet` during batch mode internally, and the returned `BPatchSnippetHandles` are filled in when `finalizeInsertionSet` is called.

Insertion sets are unnecessary when doing static binary instrumentation. Dyninst uses an implicit insertion set around all instrumentation to a static binary.

bool finalizeInsertionSet(bool atomic)

Inserts all snippets accumulated since a call to `beginInsertionSet`. If the `atomic` parameter is `true`, then a failure to insert any snippet results in all snippets being removed; effectively, the insertion is all-or-nothing. If the atomic parameter is `false`, then snippets are inserted individually. This function also fills in the `BPatchSnippetHandle` structures returned by the `insertSnippet` calls comprising this insertion set. It returns `true` on success and `false` if there was an error inserting any snippets.

Insertion sets are unnecessary when doing static binary instrumentation. Dyninst uses an implicit insertion set around all instrumentation to a static binary.

bool removeFunctionCall(BPatch_point &point)

Disable the mutatee function call at the specified location. The `point` specified must be a valid call point in the image of the mutatee. The purpose of this routine is to permit tools to alter the semantics of a program by eliminating procedure calls. The mechanism to achieve the removal is platform dependent, but might include branching over the call or replacing it with NOPs. This function only removes a function call; any parameters to the function will still be evaluated.

bool replaceFunction (BPatch_function &old, BPatch_function &new)
bool revertReplaceFunction (BPatch_function &old)

Replace all calls to user function `old` with calls to `new`. This is done by inserting instrumentation (specifically a `BPatch_funcJumpExpr`) into the beginning of function `old` such that a non-returning jump is made to function `new`. Returns `true` upon success, `false` otherwise.

bool replaceFunctionCall(BPatch_point &point, BPatch_function &newFunc)

Change the function call at the specified `point` to the function indicated by `newFunc`. The purpose of this routine is to permit runtime steering tools to change the behavior of programs by replacing a call to one procedure by a call to another. `Point` must be a function call point. If the change was successful, the return value is `true`, otherwise `false` will be returned.

***WARNING**: Care must be used when replacing functions. In particular if the compiler has performed inter-procedural register allocation between the original caller/callee pair, the replacement may not be safe since the replaced function may clobber registers the compiler thought the callee left untouched. Also the signatures of the both the function being replaced and the new function must be compatible.*

bool wrapFunction(BPatch_function *old, BPatch_function *new, Dyninst::SymtabAPI::Symbol *sym)
bool revertWrapFunction(BPatch_function *old)

Replaces all calls to function `old` with calls to function `new`. Unlike replaceFunction above, the old function can still be reached via the name specified by the provided symbol `sym`. Function wrapping allows existing code to be extended by new code. Consider

the following code that implements a fast memory allocator for a particular size of memory allocation, but falls back to the original memory allocator (referenced by `origMalloc`) for all others.

```
void *origMalloc(unsigned long size);

void *fastMalloc(unsigned long size) {
    if (size == 1024) {
          unsigned long ret = fastPool;
          fastPool += 1024;
          return ret;
    }
    else {
          return origMalloc(size);
    }
}
```

The symbol `sym` is provided by the user and must exist in the program; the easiest way to ensure it is created is to use an undefined function as shown above with the definition of `origMalloc`.

The following code wraps malloc with `fastMalloc`, while allowing functions to still access the original malloc function by calling `origMalloc`. It makes use of the `new` convert interface described in Section 5.

```
using namespace Dyninst;
using namespace SymtabAPI;
BPatch_function *malloc = appImage->findFunction(...);
BPatch_function *fastMalloc = appImage->findFunction(...);
Symtab *symtab = SymtabAPI::convert(fastMalloc->getModule());
std::vector<Symbol *> syms;
symtab->findSymbol(syms, "origMalloc",
                      Symbol::ST_UNKNOWN, // Don't specify type
                      mangledName, // Look for raw symbol name
                      false, // Not regular expression
                      false, // Don't check case
                      true); // Include undefined symbols
app->wrapFunction(malloc, fastMalloc, syms[0]);
```

For a full, executable example, see Appendix A - Complete Examples.

bool replaceCode(BPatch_point *point, BPatch_snippet *snippet)

This function has been removed; users interested in replacing code should instead use the PatchAPI code modification interface described in the PatchAPI manual. For information on accessing PatchAPI abstractions from DyninstAPI abstractions, see Section 5.

BPatch_module * loadLibrary(const char *libname, bool reload=false)

For dynamic rewriting, this function loads a dynamically linked library into the process's address space. For static rewriting, this function adds a library as a library dependency in the rewritten file. In both cases Dyninst creates a new `BPatch_module` to represent this library.

The `libname` parameter identifies the file name of the library to be loaded, in the standard way that dynamically linked libraries are specified on the operating system on which the API is running. This function returns a handle to the loaded library. The `reload` parameter is ignored and only remains for backwards compatibility.

`bool isStaticExecutable()`

This function returns `true` if the original file opened with this `BPatch_addressSpace` is a statically linked executable, or false otherwise.

processType getType()

This function returns a processType that reflects whether this address space is a BPatch_process or a BPatch_binaryEdit.

4.4 Class BPatch_process

The **BPatch_process** class represents a running process, which includes one or more threads of execution and an address space.

bool stopExecution()
bool continueExecution()
bool terminateExecution()

These three functions change the running state of the process. `stopExecution` puts the process into a stopped state. Depending on the operating system, stopping one process may stop all threads associated with a process. `continueExecution` continues execution of the process. `terminateExecution` terminates execution of the process and will invoke the exit callback if one is registered. Each function returns true on success, or false for failure. Stopping or continuing a terminated thread will fail and these functions will return false.

bool isStopped()
int stopSignal()
bool isTerminated()

These three functions query the status of a process. `isStopped` returns `true` if the process is currently stopped. If the process is stopped (as indicated by `isStopped`), then `stopSignal` can be called to find out what signal caused the process to stop. `isTermi-`

`nated` returns true if the process has exited. Any of these functions may be called multiple times, and calling them will not affect the state of the process.

BPatch_variableExpr *getInheritedVariable(BPatch_variableExpr &parentVar)

Retrieve a new handle to an existing variable (such as one created by `BPatch_process::malloc`) that was created in a parent process and now exists in a forked child process. When a process forks all existing `BPatch_variableExprs` are copied to the child process, but the Dyninst handles for these objects are not valid in the child `BPatch_process`. This function is invoked on the child process' `BPatch_process`, `parentVar` is a variable from the parent process, and a handle to a variable in the child process is returned. If `parentVar` was not allocated in the parent process, then `NULL` is returned.

BPatchSnippetHandle *getInheritedSnippet(BPatchSnippetHandle &parentSnippet)

This function is similar to `getInheritedVariable`, but operates on `BPatchSnippetHandles`. Given a child process that was created via fork and a `BPatchSnippetHandle`, `parentSnippet`, from the parent process, this function will return a handle to `parentSnippet` that is valid in the child process. If it is determined that `parentSnippet` is not associated with the parent process, then `NULL` is returned.

void detach(bool cont)

Detach from the process. The process must be stopped to call this function. Instrumentation and other changes to the process will remain active in the detached copy. The `cont` parameter is used to indicate if the process should be continued as a result of detaching.

Linux does not support detaching from a process while leaving it stopped. All processes are continued after detach on Linux.

int getPid()

Return the system id for the mutatee process. On UNIX based systems this is a PID. On Windows this is the HANDLE object for a process.

typedef enum BPatch_exitType { NoExit, ExitedNormally, ExitedViaSignal };

BPatch_exitType terminationStatus()

If the process has exited, `terminationStatus` will indicate whether the process exited normally or because of a signal. If the process has not exited, `NoExit` will be returned. On AIX, the reason why a process exited will not be available if the process was not a child of the Dyninst mutator; in this case, `ExitedNormally` will be returned in both normal and signal exit cases.

int getExitCode()

If the process exited in a normal way, `getExitCode` will return the associated exit code. Prior to Dyninst 8.2, `getExitCode` would return the argument passed to `exit` or the value returned by `main`; in Dyninst 8.2 and later, it returns the actual exit code as provided by the debug interface and seen by the parent process. In particular, on Linux, this means that exit codes are normalized to the range 0-255.

int getExitSignal()

If the process exited because of a received signal, `getExitSignal` will return the associated signal number.

void oneTimeCode(const BPatch_snippet &expr)

Cause the snippet `expr` to be executed by the mutatee immediately. If the process is multithreaded, the snippet is run on a thread chosen by Dyninst. If the user requires the snippet to be run on a particular thread, use the `BPatch_thread` version of this function instead. The process must be stopped to call this function. The behavior is synchronous; `oneTimeCode` will not return until after the snippet has been run in the application.

bool oneTimeCodeAsync(const BPatch_snippet &expr,
void *userData = NULL)

This function sets up a snippet to be evaluated by the process at the next available opportunity. When the snippet finishes running Dyninst will callback any function registered through `BPatch::registerOneTimeCodeCallback`, with `userData` passed as a parameter. This function return `true` on success and `false` if it could not post the `oneTimeCode`.

If the process is multithreaded, the snippet is run on a thread chosen by Dyninst. If the user requires the snippet to be run on a particular thread, use the `BPatch_thread` version of this function instead. The behavior is asynchronous; `oneTimeCodeAsync` returns before the snippet is executed.

If the process is running when `oneTimeCodeAsync` is called, `expr` will be run immediately. If the process is stopped, then `expr` will be run when the process is continued.

void getThreads(std::vector<BPatch_thread *> &thrds)

Get the list of threads in the process.

bool isMultithreaded()
bool isMultithreadCapable()

The former returns true if the process contains multiple threads; the latter returns true if the process can create threads (e.g., it contains a threading library) even if it has not yet.

4.5 Class BPatch_thread

The **BPatch_thread** class represents and controls a thread of execution that is running in a process.

void getCallStack(std::vector<BPatch_frame>& stack)

This function fills the given vector with current information about the call stack of the thread. Each stack frame is represented by a `BPatch_frame` (see section 4.24 for information about this class).

dynthread_t getTid()

This function returns a platform-specific identifier for this thread. This is the identifier that is used by the threading library. For example, on pthread applications this function will return the thread's `pthread_t` value.

Dyninst::LWP getLWP()

This function returns a platform-specific identifier that the operating system uses to identify this thread. For example, on UNIX platforms this returns the LWP id. On Windows this returns a HANDLE object for the thread.

unsigned getBPatchID()

This function returns a Dyninst-specific identifier for this thread. These ID's apply only to running threads, the BPatch ID of an already terminated thread my be repeated in a new thread.

BPatch_function *getInitialFunc()

Return the function that was used by the application to start this thread. For example, on pthread applications this will return the initial function that was passed to `pthread_create`.

unsigned long getStackTopAddr()

Returns the base address for this thread's stack.

bool isDeadOnArrival()

This function returns true if this thread terminated execution before Dyninst was able to attach to it. Since Dyninst performs new thread detection asynchronously, it is possible for a thread to be created and destroyed before Dyninst can attach to it. When this happens, a new `BPatch_thread` is created, but `isDeadOnArrival` always returns true for this thread. It is illegal to perform any thread-level operations on a dead on arrival thread.

BPatch_process *getProcess()

Return the `BPatch_process` that contains this thread.

void *oneTimeCode(const BPatch_snippet &expr, bool *err = NULL)

Cause the snippet `expr` to be evaluated by the process immediately. This is similar to the BPatch_process::oneTimeCode function, except that the snippet is guaranteed to run only on this thread. The process must be stopped to call this function. The behavior is synchronous; `oneTimeCode` will not return until after the snippet has been run in the application.

bool oneTimeCodeAsync(const BPatch_snippet &expr,
void *userData = NULL,
BpatchOneTimeCodeCallback cb = NULL)

This function sets up the snippet `expr` to be evaluated by this thread at the next available opportunity. When the snippet finishes running, Dyninst will callback any function registered through `BPatch::registerOneTimeCodeCallback`, with `userData` passed as a parameter. This function returns `true` if `expr` was posted or false otherwise.

This is similar to the `BPatch_process::oneTimeCodeAsync` function, except that the snippet is guaranteed to run only on this thread. The process must be stopped to call this function. The behavior is asynchronous; `oneTimeCodeAsync` returns before the snippet is executed.

4.6 Class BPatch_binaryEdit

The BPatch_binaryEdit class represents a set of executable files and library files for binary rewriting. BPatch_binaryEdit inherits from the BPatch_addressSpace class, where most functionality for binary rewriting is found.

bool writeFile(const char *outFile)

Rewrite a `BPatch_binaryEdit` to disk. The original file opened with this `BPatch_binaryEdit` is written to the current working directory with the name `outFile`. If any dependent libraries were also opened and have instrumentation or other modifications, then those libraries will be written to disk in the current working directory under their original names.

A rewritten dependency library should only be used with the original file that was opened for rewriting. For example, if the file `a.out` and its dependent library `libfoo.so` were opened for rewriting, and both had instrumentation inserted, then the rewritten `libfoo.so` should not be used without the rewritten `a.out`. To build a rewritten `libfoo.so` that can load into any process, `libfoo.so` must be the original file opened by `BPatch::openBinary`.

This function returns `true` if it successfully wrote a file, or false otherwise.

4.7 Class BPatch_sourceObj

The BPatch_sourceObj class is the C++ superclass for the BPatch_function, BPatch_module, and BPatch_image classes. It provides a set of common methods for all three classes. In addition, it can be used to build a "generic" source navigator using the `getObjParent` and `getSourceObj` methods to get parents and children of a given level (i.e. the parent of a module is an image, and the children will be the functions).

```
enum BPatchErrorLevel { BPatchFatal, BPatchSerious, BPatchWarning, BPatchInfo };

enum BPatch_sourceType {
        BPatch_sourceUnknown,
        BPatch_sourceProgram,
        BPatch_sourceModule,
        BPatch_sourceFunction,
        BPatch_sourceOuterLoop,
        BPatch_sourceLoop,
        BPatch_sourceStatement };

BPatch_sourceType getSrcType()
```

Returns the type of the current source object.

```
void getSourceObj(std::vector<BPatch_sourceObj *> &objs)
```

Returns the child source objects of the current source object. For example, when called on a `BPatch_sourceProgram` object this will return objects of type `BPatch_sourceFunction`. When called on a `BPatch_sourceFunction` object it may return `BPatch_sourceOuterLoop` and `BPatch_sourceStatement` objects.

```
BPatch_sourceObj *getObjParent()
```

Return the parent source object of the current source object. The parent of a `BPatch_image` is `NULL`.

```
typedef enum BPatch_language {
   BPatch_c,
   BPatch_cPlusPlus,
   BPatch_fortran,
   BPatch_fortran77,
   BPatch_fortran90,
   BPatch_f90_demangled_stabstr,
   BPatch_fortran95,
   BPatch_assembly,
   BPatch_mixed,
   BPatch_hpf,
   BPatch_java,
   BPatch_unknownLanguage
} BPatch_language;

BPatch_language getLanguage()
```

Return the source language of the current `BPatch_sourceObject`. For programs that are written in more than one language, `BPatch_mixed` will be returned. If there is insufficient information to determine the language, `BPatch_unknownLanguage` will be returned.

4.8 Class BPatch_function

An object of this class represents a function in the application. A `BPatch_image` object (see description below) can be used to retrieve a `BPatch_function` object representing a given function.

```
std::string getName();
std::string getDemangledName();
std::string getMangledName();
std::string getTypedName();
void getNames(std::vector<std::string> &names);
void getDemangledNames(std::vector<std::string> &names);
void getMangledNames(std::vector<std::string> &names);
void getTypedNames(std::vector<std::string> &names);
```

Return name(s) of the function. The `getName` functions return the primary name; this is typically the first symbol we encounter while parsing the program; `getName` is an alias for `getDemangledName`. The `getNames` functions return all known names for the function, including any names specified by weak symbols.

```
bool getAddressRange(Dyninst::Address &start,
                     Dyninst::Address &end)
```

Returns the bounds of the function; for non-contiguous functions, this is the lowest and highest address of code that the function includes.

```
std::vector<BPatch_localVar *> *getParams()
```

Return a vector of `BPatch_localVar` snippets that refer to the parameters of this function. The position in the vector corresponds to the position in the parameter list (starting

from zero). The returned local variables can be used to check the types of functions, and can be used in snippet expressions.

BPatch_type *getReturnType()

Return the type of the return value for this function.

BPatch_variableExpr *getFunctionRef()

For platforms with complex function pointers (e.g., 64-bit PPC) this constructs and returns the appropriate descriptor.

std::vector<BPatch_localVar *> *getVars()

Returns a vector of `BPatch_localVar` objects that contain the local variables in this function. These `BPatch_localVars` can be used as parts of snippets in instrumentation. This function requires debug information to be present in the mutatee. If Dyninst was unable to find any local variables, this function will return an empty vector. It is up to the user to free the vector returned by this function.

bool isInstrumentable()

Return `true` if the function can be instrumented, and `false` if it cannot. Various conditions can cause a function to be uninstrumentable. For example, there exists a platform-specific minimum function size beyond which a function cannot be instrumented.

bool isSharedLib()

This function returns `true` if the function is defined in a shared library.

BPatch_module *getModule()

Return the module that contains this function. Depending on whether the program was compiled for debugging or the symbol table stripped, this information may not be available. This function returns `NULL` if module information was not found.

char *getModuleName(char *name, int maxLen)

Copies the name of the module that contains this function into the buffer pointed to by `name`. Copies at most `maxLen` characters and returns a pointer to `name`.

```
enum BPatch_procedureLocation {
        BPatch_entry,
        BPatch_exit,
        BPatch_subroutine,
        BPatch_locInstruction,
        BPatch_locBasicBlockEntry,
        BPatch_locLoopEntry,
        BPatch_locLoopExit,
        BPatch_locLoopStartIter,
        BPatch_locLoopStartExit,
        BPatch_allLocations }
const std::vector<BPatch_point *> *findPoint(const BPatch_procedureLocation loc)
```

Return the `BPatch_point` or list of `BPatch_points` associated with the procedure. It is used to select which type of points associated with the procedure will be returned. `BPatch_entry` and `BPatch_exit` request respectively the entry and exit points of the subroutine. `BPatch_subroutine` returns the list of points where the procedure calls other procedures. If the lookup fails to locate any points of the requested type, `NULL` is returned.

```
enum BPatch_opCode { BPatch_opLoad, BPatch_opStore, BPatch_opPrefetch }
std::vector<BPatch_point *> *findPoint(const std::set<BPatch_opCode>& ops)
std::vector<BPatch_point *> *findPoint(const BPatch_Set<BPatch_opCode>& ops)
```

Return the vector of `BPatch_points` corresponding to the set of machine instruction types described by the argument. This version is used primarily for memory access instrumentation. The `BPatch_opCode` is an enumeration of instruction types that may be requested: `BPatch_opLoad`, `BPatch_opStore`, and `BPatch_opPrefetch`. Any combination of these may be requested by passing an appropriate argument set containing the desired types. The instrumentation points created by this function have additional memory access information attached to them. This allows such points to be used for memory access specific snippets (e.g. effective address). The memory access information attached is described under Memory Access classes in section 4.27.1.

```
BPatch_localVar *findLocalVar(const char *name)
```

Search the function's local variable collection for `name`. This returns a pointer to the local variable if a match is found. This function returns `NULL` if it fails to find any variables.

```
std::vector<BPatch_variableExpr *> *findVariable(const char * name)
bool findVariable(const char *name, std::vector<BPatch_variableExpr> &vars)
```

Return a set of variables matching `name` at the scope of this function. If no variables match in the local scope, then the global scope will be searched for matches. This function returns `NULL` if it fails to find any variables.

```
BPatch_localVar *findLocalParam(const char *name)
```

Search the function's parameters for a given name. A `BPatch_localVar *` pointer is returned if a match is found, and `NULL` is returned otherwise.

void *getBaseAddr()

Return the starting address of the function in the mutatee's address space.

BPatch_flowGraph *getCFG()

Return the control flow graph for the function, or `NULL` if this information is not available. The `BPatch_flowGraph` is described in section 4.16.

bool findOverlapping(std::vector<BPatch_function *> &funcs)

Determine which functions overlap with the current function (see Section 2). Return `true` if other functions overlap the current function; the overlapping functions are added to the `funcs` vector. Return `false` if no other functions overlap the current function.

bool addMods(std::set<StackMod *> mods)
implemented on x86 and x86-64

Apply stack modifications in `mods` to the current function; the `StackMod` class is described in section 4.25. Perform error checking, handle stack alignment requirements, and generate any modifications required for cleanup at function exit. `addMods` atomically adds all modifications in `mods`; if any mod is found to be unsafe, none of the modifications in `mods` will be applied.

`addMods` can only be used in binary rewriting mode.

Returns `false` if the stack modifications are unsafe or if Dyninst is unable to perform the analysis required to guarantee safety.

4.9 Class BPatch_point

An object of this class represents a location in an application's code at which the library can insert instrumentation. A `BPatch_image` object (see section 4.10) is used to retrieve a `BPatch_point` representing a desired point in the application.

```
enum BPatch_procedureLocation { BPatch_entry, BPatch_exit,
     BPatch_subroutine, BPatch_address }
```

BPatch_procedureLocation getPointType()

Return the type of the point.

BPatch_function *getCalledFunction()

Return a `BPatch_function` representing the function that is called at the point. If the point is not a function call site or the target of the call cannot be determined, then this function returns `NULL`.

std::string getCalledFunctionName()

Returns the name of the function called at this point. This method is similar to `getCalledFunction()->getName()`, except in cases where DyninstAPI is running in binary rewriting mode and the called function resides in a library or object file that DyninstAPI has not opened. In these cases, Dyninst is able to determine the name of the called function, but is unable to construct a `BPatch_function` object.

BPatch_function *getFunction()

Returns a `BPatch_function` representing the function in which this point is contained.

BPatch_basicBlockLoop *getLoop()

Returns the containing `BPatch_basicBlockLoop` if this point is part of loop instrumentation. Returns `NULL` otherwise.

void *getAddress()

Return the address of the first instruction at this point.

bool usesTrap_NP()

Return `true` if inserting instrumentation at this point requires using a trap. On the x86 architecture, because instructions are of variable size, the instruction at a point may be too small for Dyninst to replace it with the normal code sequence used to call instrumentation. Also, when instrumentation is placed at points other than subroutine entry, exit, or call points, traps may be used to ensure the instrumentation fits. In this case, Dyninst replaces the instruction with a single-byte instruction that generates a trap. A trap handler then calls the appropriate instrumentation code. Since this technique is used only on some platforms, on other platforms this function always returns `false`.

const BPatch_memoryAccess* getMemoryAccess()

Returns the memory access object associated with this point. Memory access points are described in section 4.27.1.

const std::vector<BPatchSnippetHandle *> getCurrentSnippets()
const std::vector<BPatchSnippetHandle *>
getCurrentSnippets(BPatch_callWhen when)

Return the `BPatchSnippetHandles` for the `BPatch_snippets` that are associated with the point. If argument `when` is `BPatch_callBefore`, then `BPatchSnippetHandles` for snippets installed immediately before this point will be returned. Alternatively, if when is `BPatch_callAfter`, then `BPatchSnippetHandles` for snippets installed immediately after this point will be returned.

bool getLiveRegisters(std::vector<BPatch_register> &regs)

Fill `regs` with the registers that are live before this point (e.g., `BPatch_callBefore`). Currently returns only general purpose registers (GPRs).

bool isDynamic()

This call returns `true` if this is a dynamic call site (e.g. a call site where the function call is made via a function pointer).

Dyninst::InstructionAPI::Instruction::Ptr getInstructionAtPoint()

On implemented platforms, this function returns a shared pointer to an InstructionAPI Instruction object representing the first machine instruction at this point's address. On unimplemented platforms, returns a `NULL` shared pointer.

4.10 Class BPatch_image

This class defines a program image (the executable associated with a process). The only way to get a handle to a BPatch_image is via the BPatch_process member function `getImage`.

const BPatch_point *createInstPointAtAddr (caddr_t address)

This function has been removed because it is not safe to use. Instead, use `findPoints`:

bool findPoints(Dyninst::Address addr,
std::vector<BPatch_point *> &points);

Returns a vector of `BPatch_points` that correspond with the provided address, one per function that includes an instruction at that address. There will be one element if there is not overlapping code.

std::vector<BPatch_variableExpr *> *getGlobalVariables()

Return a vector of global variables that are defined in this image.

BPatch_process *getProcess()

Returns the `BPatch_process` associated with this image.

char *getProgramFileName(char *name, unsigned int len)

Fills provided buffer `name` with the program's file name up to `len` characters. The filename may include path information.

bool getSourceObj(std::vector<BPatch_sourceObj *> &sources)

Fill `sources` with the source objects (see section 4.6) that belong to this image. If there are no source objects, the function returns `false`. Otherwise, it returns `true`.

std::vector<BPatch_function *> *getProcedures(
bool incUninstrumentable = false)

Return a vector of the functions in the image. If the `incUninstrumentable` flag is set, the returned table of procedures will include uninstrumentable functions. The default behavior is to omit these functions.

void getObjects(std::vector<BPatch_object *> &objs)

Fill in a vector of objects in the image.

std::vector<BPatch_module *> *getModules()

Return a vector of the modules in the image.

bool getVariables(std::vector<BPatch_variableExpr *> &vars)

Fills `vars` with the global variables defined in this image. If there are no variable, the function returns `false`. Otherwise, it returns `true`.

std::vector<BPatch_function*> *findFunction(
const char *name,
std::vector<BPatch_function*> &funcs,
bool showError = true,
bool regex_case_sensitive = true,
bool incUninstrumentable = false)

Return a vector of `BPatch_functions` corresponding to `name`, or `NULL` if the function does not exist. If `name` contains a POSIX-extended regular expression, and `dont_use_regex` is `false`, a regular expression search will be performed on function names and matching `BPatch_functions` returned. If `showError` is true, then Dyninst will report an error via the `BPatch::registerErrorCallback` if no function is found.

If the `incUninstrumentable` flag is set, the returned table of procedures will include un-instrumentable functions. The default behavior is to omit these functions.

[**NOTE**: If `name` is not found to match any demangled function names in the module, the search is repeated as if `name` is a mangled function name. If this second search succeeds, functions with mangled names matching `name` are returned instead.]

std::vector<BPatch_function*> *findFunction(
std::vector<BPatch_function*> &funcs,
BPatchFunctionNameSieve bpsieve,
void *sieve_data = NULL,
int showError = 0,
bool incUninstrumentable = false)

Return a vector of `BPatch_functions` according to the generalized user-specified filter function `bpsieve`. This permits users to easily build sets of functions according to their own specific criteria. Internally, for each `BPatch_function f` in the image, this method makes a call to `bpsieve(f.getName(), sieve_data)`. The user-specified function `bpsieve` is responsible for taking the `name` argument and determining if it belongs in the output vector, possibly by using extra user-provided information stored in `sieve_data`. If the name argument matches the desired criteria, `bpsieve` should return `true`. If it does not, `bpsieve` should return `false`.

The function `bpsieve` should be defined in accordance with the typedef:

```
bool (*BPatchFunctionNameSieve) (const char *name, void* sieve_data);
```

If the `incUninstrumentable` flag is set, the returned table of procedures will include uninstrumentable functions. The default behavior is to omit these functions.

bool findFunction(Dyninst::Address addr, std::vector<BPatch_function *> &funcs)

Find all functions that have code at the given address, `addr`. Dyninst supports functions that share code, so this method may return more than one `BPatch_function`. These functions are returned via the `funcs` output parameter. This function returns `true` if it finds any functions, `false` otherwise.

BPatch_variableExpr *findVariable(const char *name,
bool showError = true)
BPatch_variableExpr *findVariable(BPatch_point &scope,
const char *name) second form of this method is not implemented on Windows.

Performs a lookup and returns a handle to the named variable. The first form of the function looks up only variables of global scope, and the second form uses the passed `BPatch_point` as the scope of the variable. The returned `BPatch_variableExpr` can be used to create references (uses) of the variable in subsequent snippets. The scoping rules used will be those of the source language. If the image was not compiled with debugging symbols, this function will fail even if the variable is defined in the passed scope.

BPatch_type *findType(const char *name)

Performs a lookup and returns a handle to the named type. The handle can be used as an argument to `BPatch_addressSpace::malloc` to create new variables of the corresponding type.

BPatch_module *findModule(const char *name,
bool substring_match = false)

Returns a module named `name` if present in the image. If the match fails, `NULL` is returned. If `substring_match` is `true`, the first module that has `name` as a substring of its name is returned (e.g. to find `libpthread.so.1`, search for `libpthread` with `substring_match` set to `true`).

bool getSourceLines(unsigned long addr,
std::vector<BPatch_statement> & lines)

Given an address `addr`, this function returns a vector of pairs of filenames and line numbers at that address. This function is an alias for `BPatch_process::getSourceLines` (see section 4.4).

bool getAddressRanges(const char * fileName, unsigned int lineNo, std::vector< std::pair< unsigned long, unsigned long > > & ranges)

Given a file name and line number, `fileName` and `lineNo`, this function returns a list of address ranges that this source line was compiled into. This function is an alias for `BPatch_process::getAddressRanges` (see section 4.4).

bool parseNewFunctions(std::vector<BPatch_module*> &newModules, const std::vector<Dyninst::Address> &funcEntryAddrs)

This function takes as input a list of function entry points indicated by the `funcEntryAddrs` vector, which are used to seed parsing in whatever modules they are found. All affected modules are placed in the `newModules` vector, which includes any existing modules in which new functions are found, as well as modules corresponding to new regions of the binary, for which new `BPatch_modules` are created. The return value is `true` in the event that at least one previously unknown function was identified, or `false` otherwise.

4.11 Class BPatch_object

An object of this class represents the original executable or a library. It serves as a container of BPatch_module objects.

std::string name()
std::string pathName()

Return the name of this file; either just the file name or the fully path-qualified name.

Dyninst::Address fileOffsetToAddr(Dyninst::Offset offset)

Convert the provided offset into the file into a full address in memory.

```
struct Region {
    typedef enum { UNKNOWN, CODE, DATA } type_t;
    Dyninst::Address base;
    unsigned long size;
    type_t type;
};
```
void regions(std::vector<Region> &regions)

Returns information about the address ranges occupied by this object in memory.

void modules(std::vector<BPatch_module *> &modules)

Returns the modules contained in this object.

```
std::vector<BPatch_function*> *findFunction(
        const char *name,
        std::vector<BPatch_function*> &funcs,
        bool showError = true,
        bool regex_case_sensitive = true,
        bool incUninstrumentable = false)
```

Return a vector of `BPatch_functions` corresponding to `name`, or `NULL` if the function does not exist. If `name` contains a POSIX-extended regular expression, and `dont_use_regex` is `false`, a regular expression search will be performed on function names and matching `BPatch_functions` returned. If `showError` is true, then Dyninst will report an error via the `BPatch::registerErrorCallback` if no function is found.

If the `incUninstrumentable` flag is set, the returned table of procedures will include uninstrumentable functions. The default behavior is to omit these functions.

[**NOTE**: If `name` is not found to match any demangled function names in the module, the search is repeated as if `name` is a mangled function name. If this second search succeeds, functions with mangled names matching `name` are returned instead.]

```
bool findPoints(Dyninst::Address addr,
        std::vector<BPatch_point *> &points);
```

Return a vector of `BPatch_points` that correspond with the provided address, one per function that includes an instruction at that address. There will be one element if there is not overlapping code.

```
std::vector<BPatch_function*> *findFunction(
        const char *name,
        std::vector<BPatch_function*> &funcs,
        bool notify_on_failure = true,
        bool regex_case_sensitive = true,
        bool incUninstrumentable = false)
```

Return a vector of `BPatch_functions` matching `name`, or `NULL` if the function does not exist. If `name` contains a POSIX-extended regular expression, a regex search will be performed on function names, and matching `BPatch_functions` returned. [**NOTE**: The `std::vector` argument `funcs` must be declared fully by the user before calling this function. Passing in an uninitialized reference will result in undefined behavior.]

If the `incUninstrumentable` flag is set, the returned table of procedures will include uninstrumentable functions. The default behavior is to omit these functions.

[**NOTE**: If `name` is not found to match any demangled function names in the BPatch_object, the search is repeated as if `name` is a mangled function name. If this second search succeeds, functions with mangled names matching `name` are returned instead.]

4.12 Class BPatch_module

An object of this class represents a program module, which is part of a program's executable image. A BPatch_module represents a source file in either an executable or a shared library. Dyninst automatically creates a module called `DEFAULT_MODULE` in each executable to hold any objects that it cannot match to a source file. BPatch_module objects are obtained by calling the `BPatch_image` member function `getModules`.

```
std::vector<BPatch_function*> *findFunction(
        const char *name,
        std::vector<BPatch_function*> &funcs,
        bool notify_on_failure = true,
        bool regex_case_sensitive = true,
        bool incUninstrumentable = false)
```

Return a vector of `BPatch_functions` matching `name`, or `NULL` if the function does not exist. If `name` contains a POSIX-extended regular expression, a regex search will be performed on function names, and matching `BPatch_functions` returned. [**NOTE**: The `std::vector` argument `funcs` must be declared fully by the user before calling this function. Passing in an uninitialized reference will result in undefined behavior.]

If the `incUninstrumentable` flag is set, the returned table of procedures will include uninstrumentable functions. The default behavior is to omit these functions.

[**NOTE**: If `name` is not found to match any demangled function names in the module, the search is repeated as if `name` is a mangled function name. If this second search succeeds, functions with mangled names matching `name` are returned instead.]

```
BPatch_Vector<BPatch_function *> *findFunctionByAddress(
        void *addr,
        BPatch_Vector<BPatch_function *> &funcs,
        bool notify_on_failure = true,
        bool incUninstrumentable = false)
```

Return a vector of `BPatch_functions` that contains `addr`, or `NULL` if the function does not exist. [**NOTE**: The `std::vector` argument `funcs` must be declared fully by the user before calling this function. Passing in an uninitialized reference will result in undefined behavior.]

If the `incUninstrumentable` flag is set, the returned table of procedures will include uninstrumentable functions. The default behavior is to omit these functions.

```
BPatch_function *findFunctionByEntry(Dyninst::Address addr)
```

Returns the function that begins at the specified address `addr`.

BPatch_function *findFunctionByMangled(
const char *mangled_name,
bool incUninstrumentable = false)

Return a `BPatch_function` for the mangled function `name` defined in the module corresponding to the invoking `BPatch_module`, or `NULL` if it does not define the function.

If the `incUninstrumentable` flag is set, the functions searched will include uninstrumentable functions. The default behavior is to omit these functions.

bool getAddressRanges(char * fileName, unsigned int lineNo, std::vector< std::pair< unsigned long, unsigned long > > & ranges)

Given a filename and line number, `fileName` and `lineNo`, this function this function returns the ranges of mutatee addresses that implement the code range in the output parameter `ranges`. In many cases a source code line will only have one address range implementing it. However, compiler optimizations may turn this into multiple, disjoint address ranges. This information is only available if the mutatee was compiled with debug information.

This function may be more efficient than the `BPatch_process` version of this function. Calling `BPatch_process::getAddressRange` will cause Dyninst to parse line information for all modules in a process. If `BPatch_module::getAddressRange` is called then only the debug information in this module will be parsed.

This function returns `true` if it was able to find any line information, `false` otherwise.

size_t getAddressWidth()

Return the size (in bytes) of a pointer in this module. On 32-bit systems this function will return 4, and on 64-bit systems this function will return 8.

void *getBaseAddr()

Return the base address of the module. This address is defined as the start of the first function in the module.

std::vector<BPatch_function *>
*getProcedures(bool incUninstrumentable = false)

Return a vector containing the functions in the module.

char *getFullName(char *buffer, int length)

Fills `buffer` with the full path name of a module, up to `length` characters when this information is available.

BPatch_hybridMode getHybridMode()

Return the mutator's analysis mode for the mutate; the default mode is the normal mode.

char *getName(char *buffer, int len)

This function copies the filename of the module into `buffer`, up to `len` characters. It returns the value of the `buffer` parameter.

unsigned long getSize()

Return the size of the module. The size is defined as the end of the last function minus the start of the first function.

bool getSourceLines(unsigned long addr, std::vector<BPatch_statement> & lines)

This function returns the line information associated with the mutatee address `addr`. The vector `lines` contain pairs of filenames and line numbers that are associated with `addr`. In many cases only one filename and line number is associated with an address, but certain compiler optimizations may lead to multiple filenames and lines at an address. This information is only available if the mutatee was compiled with debug information.

This function may be more efficient than the `BPatch_process` version of this function. Calling `BPatch_process::getSourceLines` will cause Dyninst to parse line information for all modules in a process. If `BPatch_module::getSourceLines` is called then only the debug information in this module will be parsed.

This function returns `true` if it was able to find any line information at `addr`, or false otherwise.

char *getUniqueString(char *buffer, int length)

Performs a lookup and returns a unique string for this image. Returns a string the can be compared (via `strcmp`) to indicate if two images refer to the same underlying object file (i.e., executable or library). The contents of the string are implementation specific and defined to have no semantic meaning.

bool getVariables(std::vector<BPatch_variableExpr *> &vars)

Fill the vector `vars` with the global variables that are specified in this module. Returns `false` if no results are found and `true` otherwise.

BpatchSnippetHandle* insertInitCallback(Bpatch_snippet& callback)

This function inserts the snippet `callback` at the entry point of this module's `init` function (creating a new `init` function/section if necessary).

BpatchSnippetHandle* insertFiniCallback(Bpatch_snippet& callback)

This function inserts the snippet `callback` at the exit point of this module's `fini` function (creating a new `fini` function/section if necessary).

bool isExploratoryModeOn()

This function returns `true` if the mutator's analysis mode sets to the defensive mode or the exploratory mode.

bool isMutatee()

This function returns `true` if the module is the mutatee.

bool isSharedLib()

This function returns `true` if the module is part of a shared library.

4.13 Class BPatch_snippet

A snippet is an abstract representation of code to insert into a program. Snippets are defined by creating a new instance of the correct subclass of a snippet. For example, to create a snippet to call a function, create a new instance of the class `BPatch_funcCallExpr`. Creating a snippet does not result in code being inserted into an application. Code is generated when a request is made to insert a snippet at a specific point in a program. Sub-snippets may be shared by different snippets (i.e, a handle to a snippet may be passed as an argument to create two different snippets), but whether the generated code is shared (or replicated) between two snippets is implementation dependent.

BPatch_type *getType()

Return the type of the snippet. The `BPatch_type` system is described in section 4.14.

float getCost()

Returns an estimate of the number of seconds it would take to execute the snippet. The problems with accurately estimating the cost of executing code are numerous and out of the scope of this document[2]. It is important to realize that the returned cost value is, at best, an estimate.

The rest of the classes are derived classes of the class `BPatch_snippet`.

BPatch_actualAddressExpr()

This snippet results in an expression that evaluates to the actual address of the instrumentation. To access the original address where instrumentation was inserted, use

`BPatch_originalAddressExpr`. Note that this actual address is highly dependent on a number of internal variables and has no relation to the original address.

BPatch_arithExpr(BPatch_binOp op, const BPatch_snippet &lOperand,
const BPatch_snippet &rOperand)

Perform the required binary operation. The available binary operators are:

Operator	Description
BPatch_assign	assign the value of `rOperand` to `lOperand`
BPatch_plus	add `lOperand` and `rOperand`
BPatch_minus	subtract `rOperand` from `lOperand`
BPatch_divide	divide `rOperand` by `lOperand`
BPatch_times	multiply `rOperand` by `lOperand`
BPatch_ref	Array reference of the form `lOperand[rOperand]`
BPatch_seq	Define a sequence of two expressions (similar to comma in C)

BPatch_arithExpr(BPatch_unOp, const BPatch_snippet &operand)

Define a snippet consisting of a unary operator. The unary operators are:

Operator	Description
BPatch_negate	Returns the negation of an integer
BPatch_addr	Returns a pointer to a BPatch_variableExpr
BPatch_deref	Dereferences a pointer

BPatch_boolExpr(BPatch_relOp op, const BPatch_snippet &lOperand,
const BPatch_snippet &rOperand)

Define a relational snippet. The available operators are:

Operator	Function
BPatch_lt	Return `lOperand < rOperand`
BPatch_eq	Return `lOperand == rOperand`
BPatch_gt	Return `lOperand > rOperand`
BPatch_le	Return `lOperand <= rOperand`
BPatch_ne	Return `lOperand != rOperand`
BPatch_ge	Return `lOperand >= rOperand`
BPatch_and	Return `lOperand && rOperand` (Boolean and)
BPatch_or	Return `lOperand || rOperand` (Boolean or)

The type of the returned snippet is boolean, and the operands are type checked.

BPatch_breakPointExpr()

Define a snippet that stops a process when executed by it. The stop can be detected using the `isStopped` member function of `BPatch_process`, and the program's execution can be resumed by calling the `continueExecution` member function of `BPatch_process`.

BPatch_bytesAccessedExpr()

This expression returns the number of bytes accessed by a memory operation. For most load/store architecture machines it is a constant expression returning the number of bytes for the particular style of load or store. This snippet is only valid at a memory operation instrumentation point.

BPatch_constExpr(signed int value)
BPatch_constExpr(unsigned int value)
BPatch_constExpr(signed long value)
BPatch_constExpr(unsigned long value)
BPatch_constExpr(const char *value)
BPatch_constExpr(const void *value)
BPatch_constExpr(long long value)

Define a constant snippet of the appropriate type. The `char*` form of the constructor creates a constant string; the null-terminated string beginning at the location pointed to by the parameter is copied into the application's address space, and the `BPatch_constExpr` that is created refers to the location to which the string was copied.

BPatch_dynamicTargetExpr()

This snippet calculates the target of a control flow instruction with a dynamically determined target. It can handle dynamic calls, jumps, and return statements.

BPatch_effectiveAddressExpr()

Define an expression that contains the effective address of a memory operation. For a multi-word memory operation (i.e. more than the "natural" operation size of the machine), the effective address is the base address of the operation.

BPatch_funcCallExpr(const BPatch_function& func,
const std::vector<BPatch_snippet*> &args)

Define a call to a function. The passed function must be valid for the current code region. `Args` is a list of arguments to pass to the function; the maximum number of arguments varies by platform and is summarized below. If type checking is enabled, the types of the passed arguments are checked against the function to be called. Availability of type checking depends on the source language of the application and program being compiled for debugging.

Platform	Maximum number of arguments
AMD64/EMT-64	No limit
IA-32	No limit
POWER	8 arguments

BPatch_funcJumpExpr (const BPatch_function &func)

This snippet has been removed; use `BPatch_addressSpace::wrapFunction` instead.

BPatch_ifExpr(const BPatch_boolExpr &conditional,
const BPatch_snippet &tClause,
const BPatch_snippet &fClause)
BPatch_ifExpr(const BPatch_boolExpr &conditional,
const BPatch_snippet &tClause)

This constructor creates an if statement. The first argument, `conditional`, should be a Boolean expression that will be evaluated to decide which clause should be executed. The second argument, `tClause`, is the snippet to execute if the conditional evaluates to `true`. The third argument, `fClause`, is the snippet to execute if the conditional evaluates to `false`. This third argument is optional. Else-if statements, can be constructed by making the `fClause` of an if statement another if statement.

BPatch_insnExpr(BPatch_instruction *insn) implemented on x86-64

This constructor creates a snippet that allows the user to mimic the effect of an existing instruction. In effect, the snippet "wraps" the instruction and provides a handle to particular components of instruction behavior. This is currently implemented for memory operations, and provides two override methods: `overrideLoadAddress` and `overrideStoreAddress`. Both methods take a `BPatch_snippet` as an argument. Unlike other snippets, this snippet should be installed via a call to `BPatch_process::replaceCode` (to replace the original instruction). For example:

```
// Assume that access is of type BPatch_memoryAccess, as
// provided by a call to BPatch_point->getMemoryAccess. A
// BPatch_memoryAccess is a child of BPatch_instruction, and
// is a valid source of a BPatch_insnExpr.

BPatch_insnExpr insn(access);

// This example will modify a store by increasing the target
// address by 16.

BPatch_arithExpr newStoreAddr(BPatch_plus,
                              BPatch_effectiveAddressExpr(),
                              BPatch_constExpr(16));

// now override the original store address

insn.overrideStoreAddress(newStoreAddr)

//  now replace the original instruction with the new one.
// Point is a BPatch_point corresponding to the desired location, and
// process is a BPatch_process.

process.replaceCode(point, insn);
```

BPatch_nullExpr()

Define a null snippet. This snippet contains no executable statements.

BPatch_originalAddressExpr()

This snippet results in an expression that evaluates to the original address of the point where the snippet was inserted. To access the actual address where instrumentation is executed, use `BPatch_actualAddressExpr`.

BPatch_paramExpr(int paramNum)

This constructor creates an expression whose value is a parameter being passed to a function. `ParamNum` specifies the number of the parameter to return, starting at 0. Since the contents of parameters may change during subroutine execution, this snippet type is only valid at points that are entries to subroutines, or when inserted at a call point with the `when` parameter set to `BPatch_callBefore`.

BPatch_registerExpr(BPatch_register reg)
BPatch_registerExpr(Dyninst::MachRegister reg)

This snippet results in an expression whose value is the value in the register at the point of instrumentation.

BPatch_retExpr()

This snippet results in an expression that evaluates to the return value of a subroutine. This snippet type is only valid at `BPatch_exit` points, or at a call point with the `when` parameter set to `BPatch_callAfter`.

`BPatch_scrambleRegistersExpr()`

This snippet sets all General Purpose Registers to the flag value.

BPatch_sequence(const std::vector<BPatch_snippet*> &items)

Define a sequence of snippets. The passed snippets will be executed in the order in which they appear in `items`.

BPatch_shadowExpr(bool entry,
const BPatchStopThreadCallback &cb,
const BPatch_snippet &calculation,
bool useCache = false,
BPatch_stInterpret interp = BPatch_noInterp)

This snippet creates a shadow copy of the snippet `BPatch_stopThreadExpr`.

BPatch_stopThreadExpr(const BPatchStopThreadCallback &cb,
const BPatch_snippet &calculation,
bool useCache = false,
BPatch_stInterpret interp = BPatch_noInterp)

This snippet stops the thread that executes it. It evaluates a calculation snippet and triggers a callback to the user program with the result of the calculation and a pointer to the BPatch_point at which the snippet was inserted.

`BPatch_threadIndexExpr()`

This snippet returns an integer expression that contains the thread index of the thread that is executing this snippet. The thread index is the same value that is returned on the mutator side by `BPatch_thread::getBPatchID`.

BPatch_tidExpr(BPatch_process *proc)

This snippet results in an integer expression that contains the tid of the thread that is executing this snippet. This can be used to record the threadId, or to filter instrumentation so that it only executes for a specific thread.

BPatch_variableExpr(char *in_name,
BPatch_addressSpace *in_addSpace,
AddressSpace *as,
AstNodePtr ast_wrapper_,
BPatch_type *type, void* in_address)
BPatch_variableExpr(BPatch_addressSpace *in_addSpace,
AddressSpace *as,
void *in_address,
int in_register,
BPatch_type *type,
BPatch_storageClass storage = BPatch_storageAddr,
BPatch_point *scp = NULL)
BPatch_variableExpr(BPatch_addressSpace *in_addSpace,
AddressSpace *as,
BPatch_localVar *lv,
BPatch_type *type,
BPatch_point *scp)
BPatch_variableExpr(BPatch_addressSpace *in_addSpace,
AddressSpace *ll_addSpace,
int_variable *iv,
BPatch_type *type)

Define a variable snippet of the appropriate type. The first constructor is used to get function pointers; the second is used to get forked copies of variable expression, used by malloc; the third is used for local variables; and the last is used by `BPatch_addressSpace::findOrCreateVariable()`.

BPatch_whileExpr(const BPatch_snippet &condition,
const BPatch_snippet &body)

This constructor creates a while statement. The first argument, `condition`, should be a Boolean expression that will be evaluated to decide whether `body` should be executed. The second argument, `body`, is the snippet to execute if the condition evaluates to `true`.

4.14 Class BPatch_type

The class BPatch_type is used to describe the types of variables, parameters, return values, and functions. Instances of the class can represent language predefined types (e.g. int, float), mutatee defined types (e.g., structures compiled into the mutatee application), or mutator defined types (created using the `create*` methods of the `BPatch` class).

std::vector<BPatch_field *> *getComponents()

Return a vector of the types of the fields in a `BPatch_struct` or `BPatch_union`. If this method is invoked on a type whose `BPatch_dataClass` is not `BPatch_struct` or `BPatch_union`, `NULL` is returned.

std::vector<BPatch_cblock *> *getCblocks()

Return the common block classes for the type. The methods of the `BPatch_cblock` can be used to access information about the member of a common block. Since the same named (or anonymous) common block can be defined with different members in different functions, a given common block may have multiple definitions. The vector returned by this function contains one instance of `BPatch_cblock` for each unique definition of the common block. If this method is invoked on a type whose `BPatch_dataClass` is not `BPatch_common`, `NULL` will be returned.

BPatch_type *getConstituentType()

Return the type of the base type. For a `BPatch_array` this is the type of each element, for a `BPatch_pointer` this is the type of the object the pointer points to. For `BPatch_typedef` types, this is the original type. For all other types, NULL is returned.

```
enum BPatch_dataClass {
     BPatch_dataScalar,        BPatch_dataEnumerated,
     BPatch_dataTypeClass,     BPatch_dataStructure,
     BPatch_dataUnion,         BPatch_dataArray,
     BPatch_dataPointer,       BPatch_dataReference,
     BPatch_dataFunction,      BPatch_dataTypeAttrib,
     BPatch_dataUnknownType,   BPatch_dataMethod,
     BPatch_dataCommon,        BPatch_dataPrimitive,
     BPatch_dataTypeNumber,    BPatch_dataTypeDefine,
     BPatch_dataNullType  }
```

BPatch_dataClass getDataClass()

Return one of the above data classes for this type.

unsigned long getLow()
unsigned long getHigh()

Return the upper and lower bound of an array. Calling these two methods on non-array types produces an undefined result.

const char *getName()

Return the name of the type.

bool isCompatible(const BPatch_type &otype)

Return true if `otype` is type compatible with this type. The rules for type compatibility are given in Section 4.28. If the two types are not type compatible, the error reporting callback function will be invoked one or more times with additional information about why the types are not compatible.

4.15 Class BPatch_variableExpr

The **BPatch_variableExpr** class is another class derived from `BPatch_snippet`. It represents a variable or area of memory in a process's address space. A `BPatch_variableExpr` can be obtained from a `BPatch_process` using the `malloc` member function, or from a `BPatch_image` using the `findVariable` member function.

Some `BPatch_variableExpr` have an associated `BPatch_type`, which can be accessed by functions inherited from `BPatch_snippet`. `BPatch_variableExpr` objects will have an associated `BPatch_type` if they originate from binaries with sufficient debug information that describes types, or if they were provided with a `BPatch_type` when created by Dyninst.

BPatch_variableExpr provides several member functions not provided by other types of snippets:

void readValue(void *dst)
void readValue(void *dst, int size)

Read the value of the variable in an application's address space that is represented by this `BPatch_variableExpr`. The `dst` parameter is assumed to point to a buffer large enough to hold a value of the variable's type. If the `size` parameter is supplied, then the number of bytes it specifies will be read. For the first version of this method, if the size of the variable is unknown (i.e., no type information), no data is copied and the method returns false.

void writeValue(void *src)
void writeValue(void *src, int size)

Change the value of the variable in an application's address space that is represented by this `BPatch_variableExpr`. The `src` parameter should point to a value of the variable's type. If the `size` parameter is supplied, then the number of bytes it specifies will be written. For the first version of this method, if the size of the variable is unknown (i.e., no type information), no data is copied and the method returns false.

void *getBaseAddr()

Return the base address of the variable. This is designed to let users who wish to access elements of arrays or fields in structures do so. It can also be used to obtain the address of a variable to pass a point to that variable as a parameter to a procedure call. It is similar to the ampersand (&) operator in C.

std::vector<BPatch_variableExpr *> *getComponents()

Return a pointer to a vector containing the components of a struct or union. Each element of the vector is one field of the composite type, and contains a variable expression for accessing it.

4.16 Class BPatch_flowGraph

The **BPatch_flowGraph** class represents the control flow graph of a function. It provides methods for discovering the basic blocks and loops within the function (using which a caller can navigate the graph). A `BPatch_flowGraph` object can be obtained by calling the `getCFG` method of a `BPatch_function` object.

bool containsDynamicCallsites()

Return true if the control flow graph contains any dynamic call sites (e.g., calls through a function pointer).

void getAllBasicBlocks(std::set<BPatch_basicBlock*>&)
void getAllBasicBlocks(BPatch_Set<BPatch_basicBlock*>&)

Fill the given set with pointers to all basic blocks in the control flow graph. `BPatch_basicBlock` is described in section 4.17.

void getEntryBasicBlock(std::vector<BPatch_basicBlock*>&)

Fill the given vector with pointers to all basic blocks that are entry points to the function. `BPatch_basicBlock` is described in section 4.17.

void getExitBasicBlock(std::vector<BPatch_basicBlock*>&)

Fill the given vector with pointers to all basic blocks that are exit points of the function. `BPatch_basicBlock` is described in section 4.17.

void getLoops(std::vector<BPatch_basicBlockLoop*>&)

Fill the given vector with a list of all natural (single entry) loops in the control flow graph.

void getOuterLoops(std::vector<BPatch_basicBlockLoop*>&)

Fill the given vector with a list of all natural (single entry) outer loops in the control flow graph.

BPatch_loopTreeNode *getLoopTree()

Return the root node of the tree of loops in this flow graph.

enum BPatch_procedureLocation { BPatch_locLoopEntry, BPatch_locLoopExit, BPatch_locLoopStartIter, BPatch_locLoopEndIter }

std::vector<BPatch_point*> *findLoopInstPoints(const BPatch_procedureLocation loc, BPatch_basicBlockLoop *loop);

Find instrumentation points for the given loop that correspond to the given location: loop entry, loop exit, the start of a loop iteration and the end of a loop iteration. `BPatch_locLoopEntry` and `BPatch_locLoopExit` instrumentation points respectively execute once before the first iteration of a loop and after the last iteration. `BPatch_locLoopStartIter` and `BPatch_locLoopEndIter` respectively execute at the beginning and end of each loop iteration.

BPatch_basicBlock* findBlockByAddr(Dyninst::Address addr);

Find the basic block within this flow graph that contains `addr`. Returns `NULL` on failure. This method is inefficient but guaranteed to succeed if addr is present in any block in this CFG.

[**NOTE:** Dyninst is not always able to generate a correct flow graph in the presence of indirect jumps. If a function has a case statement or indirect jump instructions, the targets of the jumps are found by searching instruction patterns (peep-hole). The instruction patterns generated are compiler specific and the control flow graph analyses include only the ones we have seen. During the control flow graph generation, if a pattern that is not handled is used for case statement or multi-jump instructions in the function address space, the generated control flow graph may not be complete.]

4.17 Class BPatch_basicBlock

The **BPatch_basicBlock** class represents a basic block in the application being instrumented. Objects of this class representing the blocks within a function can be obtained using the `BPatch_flowGraph` object for the function. `BPatch_basicBlock` includes methods for navigating through the control flow graph of the containing function.

```
void getSources(std::vector<BPatch_basicBlock*>&)
```

Fills the given vector with the list of predecessors for this basic block (i.e, basic blocks that have an outgoing edge in the control flow graph leading to this block).

```
void getTargets(std::vector<BPatch_basicBlock*>&)
```

Fills the given vector with the list of successors for this basic block (i.e, basic blocks that are the destinations of outgoing edges from this block in the control flow graph).

```
void getOutgoingEdges(std::vector<BPatch_edge *> &out)
```

Fill `out` with all of the control flow edges that leave this basic block.

```
void getIncomingEdges(std::vector<BPatch_edge *> &inc)
```

Fills `inc` with all of the control flow edges that point to this basic block.

```
bool getInstructions(std::vector
                        <Dyninst::InstructionAPI::Instruction>&)
bool getInstructions(std::vector <
    std::pair<Dyninst::InstructionAPI::Instruction,
                Address> >&)
```

Fills the given vector with InstructionAPI `Instruction` objects representing the instructions in this basic block, and returns true if successful. See the InstructionAPI Programmer's Guide for details. The second call also returns the address each instruction starts at.

```
bool dominates(BPatch_basicBlock*)
```

This function returns `true` if the argument is pre-dominated in the control flow graph by this block, and `false` if it is not.

```
BPatch_basicBlock* getImmediateDominator()
```

Return the basic block that immediately pre-dominates this block in the control flow graph.

```
void getImmediateDominates(std::vector<BPatch_basicBlock*>&)
```

Fill the given vector with a list of pointers to the basic blocks that are immediately dominated by this basic block in the control flow graph.

```
void getAllDominates(std::set<BPatch_basicBlock*>&)
void getAllDominates(BPatch_Set<BPatch_basicBlock*>&)
```

Fill the given set with pointers to all basic blocks that are dominated by this basic block in the control flow graph.

```
bool getSourceBlocks(std::vector<BPatch_sourceBlock*>&)
```

Fill the given vector with pointers to the source blocks contributing to this basic block's instruction sequence.

int getBlockNumber()

Return the ID number of this basic block. The ID numbers are consecutive from 0 to *n-1,* where *n* is the number of basic blocks in the flow graph to which this basic block belongs.

std::vector<BPatch_point *> findPoint(const std::set<BPatch_opCode> &ops)
std::vector<BPatch_point *> findPoint(const BPatch_Set<BPatch_opCode> &ops)

Find all points in the basic block that match the given operation.

BPatch_point *findEntryPoint()
BPatch_point *findExitPoint()

Find the entry or exit point of the block.

unsigned long getStartAddress()

This function returns the starting address of the basic block. The address returned is an absolute address.

unsigned long getEndAddress()

This function returns the end address of the basic block. The address returned is an absolute address.

unsigned long getLastInsnAddress()

Return the address of the last instruction in a basic block.

bool isEntryBlock()

This function returns `true` if this basic block is an entry block into a function.

bool isExitBlock()

This function returns `true` if this basic block is an exit block of a function.

unsigned size()

Return the size of a basic block. The size is defined as the difference between the end address and the start address of the basic block.

4.18 Class BPatch_edge

The **BPatch_edge** class represents a control flow edge in a `BPatch_flowGraph`.

BPatch_point *getPoint()

Return an instrumentation point for this edge. This point can be passed to `BPatch_process::insertSnippet` to instrument the edge.

```
enum BPatch_edgeType { CondJumpTaken, CondJumpNottaken,
     UncondJump, NonJump }
```

```
BPatch_edgeType getType()
```

Return a type describing this edge. A `CondJumpTaken` edge is found after a conditional branch, along the edge that is taken when the condition is true. A `CondJumpNottaken` edge follows the path when the condition is not taken. `UncondJump` is used along an edge that flows out of an unconditional branch that is always taken. `NonJump` is an edge that flows out of a basic block that does not end in a jump, but falls through into the next basic block.

```
BPatch_basicBlock *getSource()
```

Return the source `BPatch_basicBlock` that this edge flows from.

```
BPatch_basicBlock *getTarget()
```

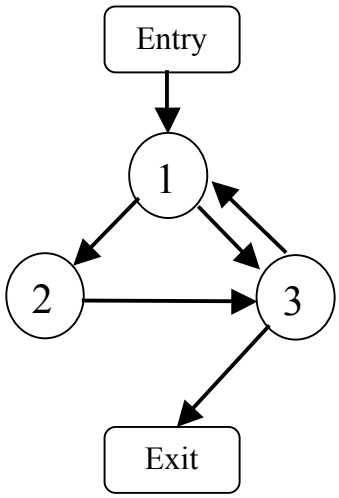
Return the target `BPatch_basicBlock` that this edge flows to.

BPatch_flowGraph *getFlowGraph()

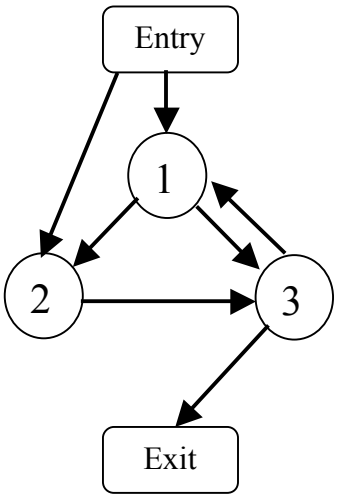
Returns the CFG that contains the edge.

4.19 Class BPatch_basicBlockLoop

An object of this class represents a loop in the code of the application being instrumented. We detect both natural loops (single-entry loops) and irreducible loops (multi-entry loops). For a natural loop, it has only one entry block and this entry block dominates all blocks in the loop; thus the entry block is also called the head or the header of the loop. However, for an irreducible loop, it has multiple entry blocks and none of them dominates all blocks in the loop; thus there is no head or header for an irreducible loop. The following figure illustrates the difference:



(a) An example of natural loop



(b) An example of irreducible loop

Figure (a) above shows a natural loop, where block 1 represents the single entry and block 1 is the head of the loop. Block 1 dominates block 2 and block 3. Figure (b) above shows an irreducible loop, where block 1 and block 2 are the entries of the loop. Neither block 1 nor block 2 domiantes block 3.

bool containsAddress(unsigned long addr)

Return true if `addr` is contained within any of the basic blocks that compose this loop, excluding the block of any of its sub-loops.

bool containsAddressInclusive(unsigned long addr)

Return true if `addr` is contained within any of the basic blocks that compose this loop, or in the blocks of any of its sub-loops.

int getBackEdges(std::vector<BPatch_edge *> &edges)

Returns the number of back edges in this loop and adds those edges to the `edges` vector. An edge is a back edge if it is from a block in the loop to an entry block of the loop.

int getLoopEntries(std::vector<BPatch_basicBlock *> &entries)

Returns the number of entry blocks of this loop and adds those blocks to the `entries` vector. An irreducible loop can have multiple entry blocks.

bool getContainedLoops(std::vector<BPatch_basicBlockLoop*>&)

Fill the given vector with a list of the loops nested within this loop.

BPatch_flowGraph *getFlowGraph()

Return a pointer to the control flow graph that contains this loop.

bool getOuterLoops(std::vector<BPatch_basicBlockLoop*>&)

Fill the given vector with a list of the outer loops nested within this loop.

bool getLoopBasicBlocks(std::vector<BPatch_basicBlock*>&)

Fill the given vector with a list of all basic blocks that are part of this loop.

bool getLoopBasicBlocksExclusive(
std::vector<BPatch_basicBlock*>&)

Fill the given vector with a list of all basic blocks that are part of this loop but not its sub-loops.

bool hasAncestor(BPatch_basicBlockLoop*)

Return true if this loop is nested within the given loop (the given loop is one of its ancestors in the tree of loops).

bool hasBlock(BPatch_basicBlock *b)

Return true if this loop or any of its sub-loops contain `b`, `false` otherwise.

bool hasBlockExclusive(BPatch_basicBlock *b)

Return true if this loop, excluding its sub-loops, contains `b`, `false` otherwise.

4.20 Class BPatch_loopTreeNode

The **BPatch_loopTreeNode** class provides a tree interface to a collection of instances of class `BPatch_basicBlockLoop` contained in a `BPatch_flowGraph`. The structure of the tree follows the nesting relationship of the loops in a function's flow graph. Each `BPatch_loopTreeNode` contains a pointer to a loop (represented by `BPatch_basicBlockLoop`), and a set of sub-loops (represented by other `BPatch_loopTreeNode` objects). The root `BPatch_loopTreeNode` instance has a null loop member since a function may contain multiple outer loops. The outer loops are contained in the root instance's vector of children.

Each instance of `BPatch_loopTreeNode` is given a name that indicates its position in the hierarchy of loops. The name of each root loop takes the form of `loop_x`, where `x` is an integer from 1 to n, where n is the number of outer loops in the function. Each sub-loop has the name of its parent, followed by a `.y`, where `y` is 1 to `m`, where `m` is the number of sub-loops under the outer loop. For example, consider the following C function:

```
void foo() {
     int x, y, z, i;
     for (x=0; x<10; x++) {
          for (y = 0; y<10; y++)
               ...
          for (z = 0; z<10; z++)
               ...
     }
     for (i = 0; i<10; i++) {
          ...
     }
```

```
}
```

The foo function will have a root BPatch_loopTreeNode, containing a NULL loop entry and two BPatch_loopTreeNode children representing the functions outer loops. These children would have names loop_1 and loop_2, respectively representing the x and i loops. loop_2 has no children. loop_1 has two child BPatch_loopTreeNode objects, named loop_1.1 and loop_1.2, respectively representing the y and z loops.

BPatch_basicBlockLoop *loop

A node in the tree that represents a single `BPatch_basicBlockLoop` instance.

std::vector<BPatch_loopTreeNode *> children

The tree nodes for the loops nested under this loop.

const char *name()

Return a name for this loop that indicates its position in the hierarchy of loops.

bool getCallees(std::vector<BPatch_function *> &v, BPatch_addressSpace *p)

This function fills the vector `v` with the list of functions that are called by this loop.

const char *getCalleeName(unsigned int i)

This function return the name of the i^{th} function called in the loop's body.

unsigned int numCallees()

Returns the number of callees contained in this loop's body.

BPatch_basicBlockLoop *findLoop(const char *name)

Finds the loop object for the given canonical loop name.

4.21 Class BPatch_register

A **BPatch_register** represents a single register of the mutatee. The list of `BPatch_registers` can be retrieved with the `BPatch_addressSpace::getRegisters` method.

`std::string name()`

This function returns the canonical name of the register.

4.22 Class BPatch_sourceBlock

An object of this class represents a source code level block. Each source block objects consists of a source file and a set of source lines in that source file. This class is used to fill source line information for each basic block in the control flow graph. For each basic block in the control flow

graph there is one or more source block object(s) that correspond to the source files and their lines contributing to the instruction sequence of the basic block.

const char* getSourceFile()

Returns a pointer to the name of the source file in which this source block occurs.

void getSourceLines(std::vector<unsigned short>&)

Fill the given vector with a list of the lines contained within this source block.

4.23 Class BPatch_cblock

This class is used to access information about a common block.

std::vector<BPatch_field *> *getComponents()

Return a vector containing the individual variables of the common block.

std::vector<BPatch_function *> *getFunctions()

Return a vector of the functions that can see this common block with the set of fields described in `getComponents`. However, other functions that define this common block with a different set of variables (or sizes of any variable) will not be returned.

4.24 Class BPatch_frame

A **BPatch_frame** object represents a stack frame. The `getCallStack` member function of `BPatch_thread` returns a vector of `BPatch_frame` objects representing the frames currently on the stack.

BPatch_frameType getFrameType()

Return the type of the stack frame. Possible types are:

Frame Type	Meaning
BPatch_frameNormal	A normal stack frame.
BPatch_frameSignal	A frame that represents a signal invocation.
BPatch_frameTrampoline	A frame the represents a call into instrumentation code.

void *getFP()

Return the frame pointer for the stack frame.

void *getPC()

Returns the program counter associated with the stack frame.

BPatch_function *findFunction()

Returns the function associated with the stack frame.

BPatch_thread *getThread()

Returns the thread associated with the stack frame.

BPatch_point *getPoint()
BPatch_point *findPoint()

For stack frames corresponding to inserted instrumentation, returns the instrumentation point where that instrumentation was inserted. For other frames, returns `NULL`.

bool isSynthesized()

Returns `true` if this frame was artificially created, `false` otherwise.

4.25 Class StackMod

This class defines modifications to the stack frame layout of a function. Stack modifications are basd on the abstraction of stack locations, not the contents of these locations. All stack offsets are with respect to the original stack frame, even if `BPatch_fuction::addMods` is called multiple times for a single function.
implemented on x86 and x86-64

Insert(int low, int high)

This constructor creates a stack modification that inserts stack space in the range [`low`, `high`), where `low` and `high` are stack offsets.

`BPatch_function::addMods` will find this modification unsafe if any instructions in the function access memory that will be non-contiguous after `[low,high)` is inserted.

Remove(int low, int high)

This constructor creates a stack modification that removes stack space in the range [`low`, `high`), where `low` and `high` are stack offsets.

`BPatch_function::addMods` will find this modification unsafe if any instructions in the function access stack memory in `[low,high)`.

Move(int sLow, int sHigh, int dLow)

This constructor creates a stack modification that moves stack space [`sLow`, `sHigh`) to [`dLow`, `dLow+(sHigh-sLow)`).

`BPatch_function::addMods` will find this modification unsafe if `Insert(dLow, dLow+(sHigh-sLow))` or `Remove(sLow, sHigh)` are unsafe.

Canary()implemented on Linux, GCC only
Canary(BPatch_function* failFunc) implemented on Linux, GCC only

This constructor creates a stack modification that inserts a stack canary at function entry and a corresponding canary check at function exit(s).

This uses the same canary as GCC's `-fstack-protector`. If the canary check at function exit fails, `failFunc` is called. `failFunc` must be non-returning and take no arguments. If no `failFunc` is provided, `__stack_chk_fail` from libc is called; libc must be open in the corresponding BPatch_addressSpace.

This modification will have no effect on functions in which the entry and exit point(s) are the same.

`BPatch_function::addMods` will find this modification unsafe if another Canary has already been added to the function. Note, however, that this modification can be applied to code compiled with `-fstack-protector`.

Randomize()
Randomize(int seed)

This constructor creates a stack modification that rearranges the stack-stored local variables of a function. This modification requires symbol information (e.g., DWARF), and only local variables specified by the symbols will be randomized. If DyninstAPI finds a stack access that is not consistent with a symbol-specified local, that local will not be randomized. Contiguous ranges of local variables are randomized; if there are two or more contiguous ranges of locals within the stack frame, each is randomized separately. More than one local variable is required for randomization.

`BPatch_function::addMods` will return false if Randomize is added to a function without local variable information, without local variables on the stack, or with only a single local variable.

`srand` is used to generate a new ordering of local variables; if `seed` is provided, this value is provided to srand as its seed.

`BPatch_function::addMods` will find this modification unsafe if any other modifications have been applied.

4.26 Container Classes

4.26.1 Class std::vector

The **std::vector** class is a container used to hold other objects used by the API. As of Dyninst 5.0, `std::vector` is an alias for the C++ Standard Template Library (STL) `std::vector`.

4.26.2 Class BPatch_Set

BPatch_Set is another container class, similar to the set class in the STL. THIS CLASS HAS BEEN DEPRECATED AND WILL BE REMOVED IN THE NEXT RELEASE. In addition the methods provided by `std::set`, it provides the following compatibility methods:

BPatch_Set()

A constructor that creates an empty set with the default comparison function.

BPatch_Set(const BPatch_Set<T,Compare>& newBPatch_Set)

Copy constructor.

void remove(const T&)

Remove the given element from the set.

bool contains(const T&)

Return true if the argument is a member of the set, otherwise returns `false`.

T* elements(T*)
void elements(std::vector<T> &)

Fill an array (or vector) with a list of the elements in the set that are sorted in ascending order according to the comparison function. The input argument should point to an array large enough to hold the elements. This function returns its input argument, unless the set is empty, in which case it returns `NULL`.

T minimum()

Return the minimum element in the set, as determined by the comparison function. For an empty set, the result is undefined.

T maximum()

Return the maximum element in the set, as determined by the comparison function. For an empty set, the result is undefined.

BPatch_Set<T,Compare>& operator+= (const T&)

Add the given object to the set.

BPatch_Set<T,Compare>& operator|= (const BPatch_Set<T,Compare>&)

Set union operator. Assign the result of the union to the set on the left hand side.

BPatch_Set<T,Compare>& operator&= (const BPatch_Set<T,Compare>&)

Set intersection operator. Assign the result of the intersection to the set on the left hand side.

BPatch_Set<T,Compare>& operator-= (const BPatch_Set<T,Compare>&)

Set difference operator. Assign the difference of the sets to the set on the left hand side.

BPatch_Set<T,Compare> operator| (const BPatch_Set<T,Compare>&)

Set union operator.

BPatch_Set<T,Compare> operator& (const BPatch_Set<T,Compare>&)

Set intersection operator.

BPatch_Set<T,Compare> operator- (const BPatch_Set<T,Compare>&)

Set difference operator.

4.27 Memory Access Classes

Instrumentation points created through findPoint(const std::set<BPatch_opCode>& ops) get memory access information attached to them. This information is used by the memory access snippets, but is also available to the API user. The classes that encapsulate memory access information are contained in the BPatch_memoryAccess_NP.h header.

4.27.1 Class BPatch_memoryAccess

This class encapsulates a memory access abstraction. It contains information that describes the memory access type: read, write, read/write, or prefetch. It also contains information that allows the effective address and the number of bytes transferred to be determined.

bool isALoad_NP()

Return true if the memory access is a load (memory is read into a register).

bool isAStore_NP()

Return true if the memory access is write. Some machine instructions may both load and store.

bool isAPrefetch_NP()

Return true if memory access is a prefetch (i.e, it has no observable effect on user registers). It this returns true, the instruction is considered neither load nor store. Prefetches are detected only on IA32.

short prefetchType_NP()

If the memory access is a prefetch, this method returns a platform specific prefetch type.

BPatch_addrSpec_NP getStartAddr_NP()

Return an address specification that allows the effective address of a memory reference to be computed. For example, on the x86 platform a memory access instruction operand

may contain a base register, an index register, a scaling value, and a constant base. The `BPatch_addrSpec_NP` describes each of these values.

BPatch_countSpec_NP getByteCount_NP()

Return a specification that describes the number of bytes transferred by the memory access.

4.27.2 Class BPatch_addrSpec_NP

This class encapsulates the information required to determine an effective address at runtime. The general representation for an address is a sum of two registers and a constant; this may change in future releases. Some architectures use only certain bits of a register (e.g. bits 25:31 of XER register on the Power chip family); these are represented as pseudo-registers. The numbering scheme for registers and pseudo-registers is implementation dependent and should not be relied upon; it may change in future releases.

int getImm()

Return the constant offset. This may be positive or negative.

int getReg(unsigned i)

Return the register number for the i^{th} register in the sum, where $0 \leq i \leq 2$. Register numbers are positive; a value of -1 means no register.

`int getScale()`

Returns any scaling factor used in the memory address computation.

4.27.3 Class BPatch_countSpec_NP

This class encapsulates the information required to determine the number of bytes transferred by a memory access.

4.28 Type System

The Dyninst type system is based on the notion of structural equivalence. Structural equivalence was selected to allow the system the greatest flexibility in allowing users to write mutators that work with applications compiled both with and without debugging symbols enabled. Using the `create*` methods of the `BPatch` class, a mutator can construct type definitions for existing mutatee structures. This information allows a mutator to read and write complex types even if the application program has been compiled without debugging information. However, if the application has been compiled with debugging information, Dyninst will verify the type compatibility of the operations performed by the mutator.

The rules for type computability are that two types must be of the same storage class (i.e. arrays are only compatible with other arrays) to be type compatible. For each storage class, the following additional requirements must be met for two types to be compatible:

Bpatch_dataScalar

> Scalars are compatible if their names are the same (as defined by strcmp) and their sizes are the same.

BPatch_dataPointer

> Pointers are compatible if the types they point to are compatible.

BPatch_dataFunc

> Functions are compatible if their return types are compatible, they have same number of parameters, and position by position each element of the parameter list is type compatible.

BPatch_dataArray

> Arrays are compatible if they have the same number of elements (regardless of their lower and upper bounds) and the base element types are type compatible.

BPatch_dataEnumerated

> Enumerated types are compatible if they have the same number of elements and the identifiers of the elements are the same.

BPatch_dataStructure
BPatch_dataUnion

> Structures and unions are compatible if they have the same number of constituent parts (fields) and item by item each field is type compatible with the corresponds field of the other type.

In addition, if either of the types is the type BPatch_unknownType, then the two types are compatible. Variables in mutatee programs that have not been compiled with debugging symbols (or in the symbols are in a format that the Dyninst library does not recognize) will be of type BPatch_unknownType.

5. Using DyninstAPI with the component libraries

In this section, we describe how to access the underlying component library abstractions from corresponding Dyninst abstractions. The component libraries (SymtabAPI, InstructionAPI, ParseAPI, and PatchAPI) often provide greater functionality and cleaner interfaces than Dyninst, and thus users may wish to use a mix of abstractions. In general, users may access component library abstractions via a `convert` function, which is overloaded and namespaced to give consistent behavior. The definitions of all component library abstractions are located in the appropriate documentation.

```
PatchAPI::PatchMgrPtr PatchAPI::convert(BPatch_addressSpace *);

PatchAPI::PatchObject *PatchAPI::convert(BPatch_object *);
ParseAPI::CodeObject *ParseAPI::convert(BPatch_object *);
SymtabAPI::Symtab *SymtabAPI::convert(BPatch_object *);

SymtabAPI::Module *SymtabAPI::convert(BPatch_module *);

PatchAPI::PatchFunction *PatchAPI::convert(BPatch_function *);
ParseAPI::Function *ParseAPI::convert(BPatch_function *);

PatchAPI::PatchBlock *PatchAPI::convert(BPatch_basicBlock *);
ParseAPI::Block *ParseAPI::convert(BPatch_basicBlock *);

PatchAPI::PatchEdge *PatchAPI::convert(BPatch_edge *);
ParseAPI::Edge *ParseAPI::convert(BPatch_edge *);

PatchAPI::Point *PatchAPI::convert(BPatch_point *, BPatch_callWhen);
PatchAPI::SnippetPtr PatchAPI::convert(BPatch_snippet *);

SymtabAPI::Type *SymtabAPI::convert(BPatch_type *);
```

6. Using the API

In this section, we describe the steps needed to compile your mutator and mutatee programs and to run them. First we give you an overview of the major steps and then we explain each one in detail.

6.1 Overview of Major Steps

To use Dyninst, you have to:

(1) Build and install DyninstAPI (Section 6.2): You will need to build and install the DyninstAPI library.
(2) Create a mutator program (Section 6.2.3): You need to create a program that will modify some other program. For an example, see the mutator shown in Appendix A.
(3) Set up the mutatee (Section 6.4): On some platforms, you need to link your application with Dyninst's run time instrumentation library. [**NOTE**: This step is only needed in the current release of the API. Future releases will eliminate this restriction.]
(4) *Run the mutator (Section 6.5):* The mutator will either create a new process or attach to an existing one (depending on the whether createProcess or attachProcess is used).

Sections 6.2 through 6.5 explain these steps in more detail.

6.2 Building and Installing DyninstAPI

This section describes how to build and install Dyninst, which can be downloaded from http://www.dyninst.org. You may either download source code or an installation package; if you choose to use an installation package, you should run the installation package and then skip to section **6.2.3.** We strongly recommend, however, that you build Dyninst from source; this ensures that your mutators and Dyninst itself are built with compatible toolchains.

Dyninst and its components are no longer built with autotools, but with CMake. For a complete guide to using CMake, see the CMake documentation at http://www.cmake.org.

6.2.1 Quick upgrade guide for existing Dyninst users

In your desired build directory, invoke the following:

```
cmake /path/to/dyninst/source -Dfoo=bar …
make
make install
```

in place of:

```
configure --with-foo=bar
make
make install
```

Note that CMake does support out-of-source builds, and does not provide a "distclean" target. Building in a separate build directory is highly recommended. The PLATFORM environment variable will be automatically set to our best guess of your platform; you may manually override this if necessary. For most users, it is best not to set a platform directory. Valid values are in cmake/platforms-unix.cmake and cmake/platforms-win.cmake.

The GNU, Intel, and Microsoft compiler suites are known to build Dyninst successfully. Other compilers will need some modifications within the cmake directory to be properly detected and to have proper flags passed.

The most common configuration options are:

BOOST_ROOT: base directory of your boost installation
CMAKE_CXX_COMPILER: C++ compiler to use.
CMAKE_C_COMPILER: C compiler to use
LIBELF_INCLUDE_DIR: location of elf.h and libelf.h
LIBELF_LIBRARIES: full path of libelf.so
LIBDWARF_INCLUDE_DIR: location of libdwarf.h
LIBDWARF_LIBRARIES: full path of libdwarf.so
IBERTY_LIBRARIES: full path of libiberty.[a|so]; libiberty.a must be built with -fPIC
CMAKE_[C|CXX]_COMPILER_FLAGS: additional C/C++ compiler flags to use
CMAKE_BUILD_TYPE: may be set to Debug, Release, or RelWithDebInfo for unoptimized, optimized, and optimized with debug information builds respectively. This replaces the NO_OPT_FLAG environment variable previously used by Dyninst.
CMAKE_INSTALL_PREFIX: like PREFIX for autotools-based systems. Where to install things.
BUILD_RTLIB_32: enable building a 32-bit runtime library on 64-bit systems. Only enable if you have a fully functional 32-bit build environment.
RT_C_COMPILER: compiler to use for the runtime library. Ordinarily this will be the compiler you use for the rest of Dyninst, but on systems like BlueGene, this needs to be the compiler for the mutatee environment.
For a full list of options, the curses-based ccmake or the GUI-based cmake-gui are the best choices. Note that, unlike with autotools-based systems, the COMPILER variables and the FLAGS variables are wholly separate; setting CMAKE_C_COMPILER="/usr/bin/gcc -m32" will not behave correctly.

6.2.2 New capabilities

CMake allows Dyninst to be built out-of-source; simply invoke CMake in your desired build location. In-source builds are still fully supported as well.
Each component of Dyninst may be built independently: make $component[-install]. Standard make options will work; there is limited support for make -jN. Setting VERBOSE=1 will replace the beautified CMake output with raw commands and their output, which can be useful for troubleshooting.

Libelf, libdwarf, and libiberty will be automatically downloaded and used in the build provided that they are needed, they cannot be found, and your CMake version is at least 2.8.11. With older versions of CMake, you will be required to specify the location of these libraries if they are not present in system default paths.
Dyninst now requires the boost thread libraries in order to build proccontrolAPI. These are available from boost.org, and should be available as prebuilt packages on most Linux distributions.

6.2.3 Building on Windows

You will not need libdwarf or libelf in order to build Dyninst on Windows, but you will need the Debug Information Access (DIA) SDK. This is available through MSDN or with a paid version of Visual Studio; check Microsoft's website for current licensing information and availability. Dyninst has been tested with both the Visual Studio project file generators and the NMake makefile generators; it has not been tested using gcc/Cygwin and is unlikely to work out of the box in that environment.

6.2.4 Configuration notes

The Dyninst runtime library must be built based on the mutatee's environment, not the mutator's environment. For most use cases, these are identical, but there are two common cases where they are not: mixed 32 and 64 bit systems, and binary rewriting for heterogeneous systems (e.g. BlueGene).

The runtime library accepts a user-specified compiler (RT_C_COMPILER). If you are building both a 32-bit and a 64-bit runtime library on a 64-bit system, the assembler used must be able to accept an .S file and the "-m32" flag.

6.3 Creating a Mutator Program

The first step in using Dyninst is to create a mutator program. The mutator program specifies the mutatee (either by naming an executable to start or by supplying a process ID for an existing process). In addition, your mutator will include the calls to the API library to modify the mutatee. For the rest of this section, we assume that the mutator is the sample program given in Appendix A - Complete Examples.

The following fragment of a Makefile shows how to link your mutator program with the Dyninst library on most platforms:

```
# DYNINST_INCLUDE and DYNINST_LIB should be set to locations
# where Dyninst header and library files were installed, respectively

retee.o: retee.c
$(CC) -c $(CFLAGS) -I$(DYNINST_INCLUDE) retee.c –std=c++0x

retee: retee.o
      $(CC) retee.o -L$(DYNINST_LIB) -ldyninstAPI -o retee –std=c++0x
```

On Linux, the options `-lelf` and `-ldwarf` may be required at the link step. You will also need to make sure that the `LD_LIBRARY_PATH` environment variable includes the directory that contains the Dyninst shared library. If libdwarf was specified as a static library, you will need to add the following link options:

```
ld: -export-dynamic --whole-archive -ldwarf --no-whole-archive

g++: -rdynamic -Wl,--whole-archive –ldwarf -Wl,--no-whole-archive
```

Since Dyninst uses the C++11x standard, you will also need to enable this option for your compiler. For GCC versions 4.3 and later, this is done by specifying `-std=c++0x`. For GCC versions 4.7 and later, this is done by specifying `-std=c++11`. Some of these libraries, such as libdwarf and libelf, may not be standard on various platforms. Check the README file in dyninst/dyninstAPI for more information on where to find these libraries.

Under Windows NT, the mutator also needs to be linked with the `dbghelp` library, which is included in the Microsoft Platform SDK. Below is a fragment from a Makefile for Windows NT:

```
# DYNINST_INCLUDE and DYNINST_LIB should be set to locations
# where Dyninst header and library files were installed, respectively

CC = cl

retee.obj: retee.c
      $(CC) -c $(CFLAGS) -I$(DYNINST_INCLUDE)/h

retee.exe: retee.obj
      link -out:retee.exe retee.obj $(DYNINST_LIB)\libdyninstAPI.lib \
       dbghelp.lib
```

6.4 Setting Up the Application Program (mutatee)

On most platforms, any additional code that your mutator might need to call in the mutatee (for example files containing instrumentation functions that were too complex to write directly using the API) can be put into a dynamically loaded shared library, which your mutator program can load into the mutatee at runtime using the loadLibrary member function of BPatch_process.

To locate the runtime library that Dyninst needs to load into your program, an additional environment variable must be set. The variable `DYNINSTAPI_RT_LIB` should be set to the full pathname of the run time instrumentation library, which should be:

```
NOTE: DYNINST_LIB should be set to the location where Dyninst library
files were installed

$(DYNINST_LIB)/libdyninstAPI_RT.so (UNIX)
```

`%DYNINST_LIB/libdyninstAPI_RT.dll` (Windows)

6.5 Running the Mutator

At this point, you should be ready to run your application program with your mutator. For example, to start the sample program shown in Appendix A - Complete Examples:

```
% retee foo <pid>
```

6.6 Optimizing Dyninst Performance

This section describes how to tune Dyninst for optimum performance. During the course of a run, Dyninst will perform several types of analysis on the binary, make safety assumptions about instrumentation that is inserted, and rewrite the binary (perhaps several times). Given some guidance from the user, Dyninst can make assumptions about what work it needs to do and can deliver significant performance improvements.

There are two areas of Dyninst performance users typically care about. First, the time it takes Dyninst to parse and instrument a program. This is typically the time it takes Dyninst to start and analyze a program, and the time it takes to modify the program when putting in instrumentation. Second, many users care about the time instrumentation takes in the modified mutatee. This time is highly dependent on both the amount and type of instrumentation put it, but it is still possible to eliminate some of the Dyninst overhead around the instrumentation.

The following subsections describe techniques for improving the performance of these two areas.

6.6.1 Optimizing Mutator Performance

CPU time in the Dyninst mutator is usually consumed by either parsing or instrumenting binaries. When a new binary is loaded, Dyninst will analyze the code looking for instrumentation points, global variables, and attempting to identify functions in areas of code that may not have symbols. Upon user request, Dyninst will also parse debug information from the binary, which includes local variable, line, and type information.

All of these items are parsed lazily, that is Dyninst won't try to generate this information until it is requested. Information is parsed on a per-library basis, so a request for information about a specific library function will cause Dyninst to parse information about all functions in that library. Much of the Dyninst parsing performance problems can be removed, or mitigated, by structuring the mutator application so that it only requests information from Dyninst if and when it needs it.

Not all operations require Dyninst to trigger parsing. Some common operations that lead to parsing are:

- Requesting a BPatch_point object
- Any operation on a BPatch_function other than getting its name

Debugging information is lazily parsed separately from the rest of the binary parsing. Accessing line, type, or local variable information will cause Dyninst to parse the debug information for all three of these.

Another common source of mutator time is spent re-writing the mutatee to add instrumentation. When instrumentation is inserted into a function, Dyninst may need to rewrite some or all of the function to fit the instrumentation in. If multiple pieces of instrumentation are being inserted into a function, Dyninst may need to rewrite that function multiple times.

If the user knows that they will be inserting multiple pieces of instrumentation into one function, they can batch the instrumentation into one bundle, so that the function will only be re-written once, using the `BPatch_process::beginInsertionSet` and `BPatch_process::endInsertionSet` functions (see section 4.4). Using these functions can result in a significant performance win when inserting instrumentation in many locations.

To use the insertion set functions, add a call to `beginInsertionSet` before inserting instrumentation. Dyninst will start buffering up all instrumentation insertions. After the last piece of instrumentation is inserted, call `finalizeInsertionSet`, and all instrumentation will be atomically inserted into the mutatee, with each function being rewritten at most once.

6.6.2 Optimizing Mutatee Performance

As instrumentation is inserted into a mutatee, it will start to run slower. The slowdown is heavily influenced by three factors: the number of points being instrumented, the instrumentation itself, and the Dyninst overhead around each piece of instrumentation. The Dyninst overhead comes from pieces of protection code (described in more detail below) that do things such as saving/restoring registers around instrumentation, checking for instrumentation recursion, and performing thread safety checks.

The factor by which Dyninst overhead influences mutatee run-time depends on the type of instrumentation being inserted. When inserting instrumentation that runs a memory cache simulator, the Dyninst overhead may be negligible. On the other-hand, when inserting instrumentation that increments a counter, the Dyninst overhead will dominate the time spent in instrumentation. Remember, optimizing the instrumentation being inserted may sometimes be more important than optimizing the Dyninst overhead. Many users have had success writing tools that make use of Dyninst's ability to dynamically remove instrumentation as a performance improvement.

The instrumentation overhead results from safety and correctness checks inserted by Dyninst around instrumentation. Dyninst will automatically attempt to remove as much of this overhead as possible, however it sometimes must make a conservative decision to leave the overhead in. Given additional, user-provided information Dyninst can make better choices about what safety checks to leave in. An unoptimized post-Dyninst 5.0 instrumentation snippet looks like the following:

Save General Purpose Registers	In order to ensure that instrumentation doesn't corrupt the program, Dyninst saves all live general purpose registers.
Save Floating Point Registers	Dyninst may decide to separately save any floating point registers that may be corrupted by instrumentation.
Generate A Stack Frame	Dyninst builds a stack frame for instrumentation to run under. This provides the illusion to instrumentation that it is running as its own function.
Calculate Thread Index	Calculate an index value that identifies the current thread. This is primarily used as input to the Trampoline Guard.
Test and Set Trampoline Guard	Test to see if we are already recursively executing under instrumentation, and skip the user instrumentation if we are.
Execute User Instrumentation	Execute any `BPatch_snippet` code.
Unset Trampoline Guard	Marks the this thread as no longer being in instrumentation
Clean Stack Frame	Clean the stack frame that was generated for instrumentation.
Restore Floating Point Registers	Restore the floating point registers to their original state.
Restore General Purpose Registers	Restore the general purpose registers to their original state.

Dyninst will attempt to eliminate as much of its overhead as is possible. The Dyninst user can assist Dyninst by doing the following:

- **Write BPatch_snippet code that avoids making function calls.** Dyninst will attempt to perform analysis on the user written instrumentation to determine which general purpose and floating point registers can be saved. It is difficult to analyze function calls that may be nested arbitrarily deep. Dyninst will not analyze any deeper than two levels of function calls before assuming that the instrumentation clobbers all registers and it needs to save everything.

  In addition, not making function calls from instrumentation allows Dyninst to eliminate its tramp guard and thread index calculation. Instrumentation that does not make a function call cannot recursively execute more instrumentation.
- **Call `BPatch::setTrampRecursive(true)` if instrumentation cannot execute recursively.** If instrumentation must make a function call, but will not execute recursively, then enable trampoline recursion. This will cause Dyninst to stop generating a trampoline guard and thread index calculation on all future pieces of instrumentation. An example of instrumentation recursion would be instrumenting a call to `write` with instrumentation that calls `printf`—`write` will start calling `printf` printf will re-call `write`.
- **Call `BPatch::setSaveFPR(false)` if instrumentation will not clobber floating point registers**. This will cause Dyninst to stop saving floating point registers, which can be a significant win on some platforms.

- **Use simple `BPatch_snippet` objects when possible**. Dyninst will attempt to recognize, peep-hole optimize, and simplify frequently used code snippets when it finds them. For example, on x86 based platforms Dyninst will recognize snippets that do operations like 'var = constant' or 'var++' and turn these into optimized assembly instructions that take advantage of CISC machine instructions.
- **Call `BPatch::setInstrStackFrames(false)` before inserting instrumentation that does not need to set up stack frames.** Dyninst allows you to force stack frames to be generated for all instrumentation. This is useful for some applications (e.g., debugging your instrumentation code) but allowing Dyninst to omit stack frames wherever possible will improve performance. This flag is false by default; it should be enabled for as little instrumentation as possible in order to maximize the benefit from optimizing away stack frames.
- **Avoid conditional instrumentation wherever possible.** Conditional logic in your instrumentation makes it more difficult to avoid saving the state of the flags.
- **Avoid unnecessary instrumentation.** Dyninst provides you with all kinds of information that you can use to select only the points of actual interest for instrumentation. Use this information to instrument as selectively as possible. The best way to optimize your instrumentation, ultimately, is to know *a priori* that it was unnecessary and not insert it.

Appendix A - Complete Examples

In this section we show two complete examples: the programs from Section 3 and a complete Dyninst program, retee.

1.1 Instrumenting a function

```
#include <stdio.h>

#include "BPatch.h"
#include "BPatch_addressSpace.h"
#include "BPatch_process.h"
#include "BPatch_binaryEdit.h"
#include "BPatch_point.h"
#include "BPatch_function.h"

using namespace std;
using namespace Dyninst;

// Create an instance of class BPatch
BPatch bpatch;

// Different ways to perform instrumentation
typedef enum {
    create,
    attach,
    open
} accessType_t;

// Attach, create, or open a file for rewriting
BPatch_addressSpace* startInstrumenting(accessType_t accessType,
        const char* name,
        int pid,
        const char* argv[]) {
    BPatch_addressSpace* handle = NULL;

    switch(accessType) {
        case create:
            handle = bpatch.processCreate(name, argv);
            if (!handle) { fprintf(stderr, "processCreate failed\n"); }
            break;
        case attach:
            handle = bpatch.processAttach(name, pid);
            if (!handle) { fprintf(stderr, "processAttach failed\n"); }
            break;
        case open:
            // Open the binary file and all dependencies
            handle = bpatch.openBinary(name, true);
            if (!handle) { fprintf(stderr, "openBinary failed\n"); }
            break;
    }

    return handle;
}
```

```
// Find a point at which to insert instrumentation
std::vector<BPatch_point*>* findPoint(BPatch_addressSpace* app,
          const char* name,
          BPatch_procedureLocation loc) {
    std::vector<BPatch_function*> functions;
    std::vector<BPatch_point*>* points;

    // Scan for functions named "name"
    BPatch_image* appImage = app->getImage();
    appImage->findFunction(name, functions);
    if (functions.size() == 0) {
        fprintf(stderr, "No function %s\n", name);
        return points;
    } else if (functions.size() > 1) {
        fprintf(stderr, "More than one %s; using the first one\n", name);
    }

    // Locate the relevant points
    points = functions[0]->findPoint(loc);
    return points;
}

// Create and insert an increment snippet
bool createAndInsertSnippet(BPatch_addressSpace* app,
        std::vector<BPatch_point*>* points) {
    BPatch_image* appImage = app->getImage();

    // Create an increment snippet
    BPatch_variableExpr* intCounter =
        app->malloc(*(appImage->findType("int")), "myCounter");
    BPatch_arithExpr addOne(BPatch_assign,
                            *intCounter,
                            BPatch_arithExpr(BPatch_plus,
                                *intCounter,
                                BPatch_constExpr(1)));

    // Insert the snippet
    if (!app->insertSnippet(addOne, *points)) {
        fprintf(stderr, "insertSnippet failed\n");
        return false;
    }
    return true;
}

// Create and insert a printf snippet
bool createAndInsertSnippet2(BPatch_addressSpace* app,
        std::vector<BPatch_point*>* points) {
    BPatch_image* appImage = app->getImage();

    // Create the printf function call snippet
    std::vector<BPatch_snippet*> printfArgs;
    BPatch_snippet* fmt =
        new BPatch_constExpr("InterestingProcedure called %d times\n");
    printfArgs.push_back(fmt);
```

```
    BPatch_variableExpr* var = appImage->findVariable("myCounter");
    if (!var) {
        fprintf(stderr, "Could not find 'myCounter' variable\n");
        return false;
    } else {
        printfArgs.push_back(var);
    }

    // Find the printf function
    std::vector<BPatch_function*> printfFuncs;
    appImage->findFunction("printf", printfFuncs);
    if (printfFuncs.size() == 0) {
        fprintf(stderr, "Could not find printf\n");
        return false;
    }

    // Construct a function call snippet
    BPatch_funcCallExpr printfCall(*(printfFuncs[0]), printfArgs);

    //  Insert the snippet
    if (!app->insertSnippet(printfCall, *points)) {
        fprintf(stderr, "insertSnippet failed\n");
        return false;
    }
    return true;
}

void finishInstrumenting(BPatch_addressSpace* app, const char* newName)
{
    BPatch_process* appProc = dynamic_cast<BPatch_process*>(app);
    BPatch_binaryEdit* appBin = dynamic_cast<BPatch_binaryEdit*>(app);

    if (appProc) {
        if (!appProc->continueExecution()) {
            fprintf(stderr, "continueExecution failed\n");
        }
        while (!appProc->isTerminated()) {
            bpatch.waitForStatusChange();
        }
    } else if (appBin) {
        if (!appBin->writeFile(newName)) {
            fprintf(stderr, "writeFile failed\n");
        }
    }
}

int main() {
    // Set up information about the program to be instrumented
    const char* progName = "InterestingProgram";
    int progPID = 42;
    const char* progArgv[] = {"InterestingProgram", "-h", NULL};
    accessType_t mode = create;

    // Create/attach/open a binary
    BPatch_addressSpace* app =
        startInstrumenting(mode, progName, progPID, progArgv);
    if (!app) {
```

```
        fprintf(stderr, "startInstrumenting failed\n");
        exit(1);
    }

    // Find the entry point for function InterestingProcedure
    const char* interestingFuncName = "InterestingProcedure";
    std::vector<BPatch_point*>* entryPoint =
        findPoint(app, interestingFuncName, BPatch_entry);
    if (!entryPoint || entryPoint->size() == 0) {
        fprintf(stderr, "No entry points for %s\n", interestingFuncName);
        exit(1);
    }

    // Create and insert instrumentation snippet
    if (!createAndInsertSnippet(app, entryPoint)) {
        fprintf(stderr, "createAndInsertSnippet failed\n");
        exit(1);
    }

    // Find the exit point of main
    std::vector<BPatch_point*>* exitPoint =
        findPoint(app, "main", BPatch_exit);
    if (!exitPoint || exitPoint->size() == 0) {
        fprintf(stderr, "No exit points for main\n");
        exit(1);
    }

    // Create and insert instrumentation snippet 2
    if (!createAndInsertSnippet2(app, exitPoint)) {
        fprintf(stderr, "createAndInsertSnippet2 failed\n");
        exit(1);
    }
    // Finish instrumentation
    const char* progName2 = "InterestingProgram-rewritten";
    finishInstrumenting(app, progName2);
}
```

1.2 Binary Analysis

```
#include <stdio.h>

#include "BPatch.h"
#include "BPatch_addressSpace.h"
#include "BPatch_process.h"
#include "BPatch_binaryEdit.h"
#include "BPatch_function.h"
#include "BPatch_flowGraph.h"

using namespace std;
using namespace Dyninst;

// Create an instance of class BPatch
BPatch bpatch;

// Different ways to perform instrumentation
typedef enum {
    create,
```

```
    attach,
    open
} accessType_t;

BPatch_addressSpace* startInstrumenting(accessType_t accessType,
        const char* name,
        int pid,
        const char* argv[]) {
    BPatch_addressSpace* handle = NULL;

    switch(accessType) {
        case create:
            handle = bpatch.processCreate(name, argv);
            if (!handle) { fprintf(stderr, "processCreate failed\n"); }
            break;
        case attach:
            handle = bpatch.processAttach(name, pid);
            if (!handle) { fprintf(stderr, "processAttach failed\n"); }
            break;
        case open:
            // Open the binary file and all dependencies
            handle = bpatch.openBinary(name, true);
            if (!handle) { fprintf(stderr, "openBinary failed\n"); }
            break;
    }

    return handle;
}

int binaryAnalysis(BPatch_addressSpace* app) {
    BPatch_image* appImage = app->getImage();

    int insns_access_memory = 0;

    std::vector<BPatch_function*> functions;
    appImage->findFunction("InterestingProcedure", functions);

    if (functions.size() == 0) {
        fprintf(stderr, "No function InterestingProcedure\n");
        return insns_access_memory;
    } else if (functions.size() > 1) {
        fprintf(stderr, "More than one InterestingProcedure; using the
first one\n");
    }

    BPatch_flowGraph* fg = functions[0]->getCFG();

    std::set<BPatch_basicBlock*> blocks;
    fg->getAllBasicBlocks(blocks);

    for (auto block_iter = blocks.begin();
            block_iter != blocks.end();
            ++block_iter) {
        BPatch_basicBlock* block = *block_iter;
        std::vector<InstructionAPI::Instruction::Ptr> insns;
        block->getInstructions(insns);
```

```
        for (auto insn_iter = insns.begin();
                insn_iter != insns.end();
                ++insn_iter) {
            InstructionAPI::Instruction::Ptr insn = *insn_iter;
            if (insn->readsMemory() || insn->writesMemory()) {
                insns_access_memory++;
            }
        }
    }

    return insns_access_memory;
}
```

```
int main() {
    // Set up information about the program to be instrumented
    const char* progName = "InterestingProgram";
    int progPID = 42;
    const char* progArgv[] = {"InterestingProgram", "-h", NULL};
    accessType_t mode = create;

    // Create/attach/open a binary
    BPatch_addressSpace* app =
        startInstrumenting(mode, progName, progPID, progArgv);
    if (!app) {
        fprintf(stderr, "startInstrumenting failed\n");
        exit(1);
    }

    int memAccesses = binaryAnalysis(app);

    fprintf(stderr, "Found %d memory accesses\n", memAccesses);
}
```

1.3 Instrumenting Memory Accesses

```
#include <stdio.h>

#include "BPatch.h"
#include "BPatch_addressSpace.h"
#include "BPatch_process.h"
#include "BPatch_binaryEdit.h"
#include "BPatch_point.h"
#include "BPatch_function.h"

using namespace std;
using namespace Dyninst;

// Create an instance of class BPatch
BPatch bpatch;

// Different ways to perform instrumentation
typedef enum {
```

```
    create,
    attach,
    open
} accessType_t;

// Attach, create, or open a file for rewriting
BPatch_addressSpace* startInstrumenting(accessType_t accessType,
        const char* name,
        int pid,
        const char* argv[]) {
    BPatch_addressSpace* handle = NULL;

    switch(accessType) {
        case create:
            handle = bpatch.processCreate(name, argv);
            if (!handle) { fprintf(stderr, "processCreate failed\n"); }
            break;
        case attach:
            handle = bpatch.processAttach(name, pid);
            if (!handle) { fprintf(stderr, "processAttach failed\n"); }
            break;

         case open:
            // Open the binary file; do not open dependencies
            handle = bpatch.openBinary(name, false);
            if (!handle) { fprintf(stderr, "openBinary failed\n"); }
            break;
    }

    return handle;
}

bool instrumentMemoryAccesses(BPatch_addressSpace* app) {
    BPatch_image* appImage = app->getImage();

    // We're interested in loads and stores
    BPatch_Set<BPatch_opCode> axs;
    axs.insert(BPatch_opLoad);
    axs.insert(BPatch_opStore);

    // Scan the function InterestingProcedure
    // and create instrumentation points
    std::vector<BPatch_function*> functions;
    appImage->findFunction("InterestingProcedure", functions);
    std::vector<BPatch_point*>* points =
        functions[0]->findPoint(axs);
    if (!points) {
        fprintf(stderr, "No load/store points found\n");
        return false;
    }

    // Create the printf function call snippet
    std::vector<BPatch_snippet*> printfArgs;
    BPatch_snippet* fmt = new BPatch_constExpr("Access at: 0x%lx\n");
    printfArgs.push_back(fmt);
    BPatch_snippet* eae = new BPatch_effectiveAddressExpr();
    printfArgs.push_back(eae);
```

```
    // Find the printf function
    std::vector<BPatch_function*> printfFuncs;
    appImage->findFunction("printf", printfFuncs);
    if (printfFuncs.size() == 0) {
        fprintf(stderr, "Could not find printf\n");
        return false;
    }

    // Construct a function call snippet
    BPatch_funcCallExpr printfCall(*(printfFuncs[0]), printfArgs);

    // Insert the snippet at the instrumentation points
    if (!app->insertSnippet(printfCall, *points)) {
        fprintf(stderr, "insertSnippet failed\n");
        return false;
    }
    return true;
}

void finishInstrumenting(BPatch_addressSpace* app, const char* newName) {
    BPatch_process* appProc = dynamic_cast<BPatch_process*>(app);
    BPatch_binaryEdit* appBin = dynamic_cast<BPatch_binaryEdit*>(app);

    if (appProc) {
        if (!appProc->continueExecution()) {
            fprintf(stderr, "continueExecution failed\n");
        }
        while (!appProc->isTerminated()) {
            bpatch.waitForStatusChange();
        }
    } else if (appBin) {
        if (!appBin->writeFile(newName)) {
            fprintf(stderr, "writeFile failed\n");
        }
    }
}

int main() {
    // Set up information about the program to be instrumented
    const char* progName = "InterestingProgram";
    int progPID = 42;
    const char* progArgv[] = {"InterestingProgram", "-h", NULL};
    accessType_t mode = create;

    // Create/attach/open a binary
    BPatch_addressSpace* app =
        startInstrumenting(mode, progName, progPID, progArgv);
    if (!app) {
        fprintf(stderr, "startInstrumenting failed\n");
        exit(1);
    }

    // Instrument memory accesses
    if (!instrumentMemoryAccesses(app)) {
        fprintf(stderr, "instrumentMemoryAccesses failed\n");
        exit(1);
```

```
        }

        // Finish instrumentation
        const char* progName2 = "InterestingProgram-rewritten";
        finishInstrumenting(app, progName2);
    }
```

1.4 retee

The final example is a program called “re-tee.” It takes three arguments: the pathname of an executable program, the process id of a running instance of the same program, and a file name. It adds code to the running program that copies to the named file all output that the program writes to its standard output file descriptor. In this way it works like “tee,” which passes output along to its own standard out while also saving it in a file. The motivation for the example program is that you run a program, and it starts to print copious lines of output to your screen, and you wish to save that output in a file without having to re-run the program.

```
#include <stdio.h>
#include <fcntl.h>
#include <vector>
#include "BPatch.h"
#include "BPatch_point.h"
#include "BPatch_process.h"
#include "BPatch_function.h"
#include "BPatch_thread.h"

/*
 * retee.C
 *
 * This program (mutator) provides an example of several facets of
 * Dyninst's behavior, and is a good basis for many Dyninst
 * mutators. We want to intercept all output from a target application
 * (the mutatee), duplicating output to a file as well as the
 * original destination (e.g., stdout).
 *
 * This mutator operates in several phases. In brief:
 * 1) Attach to the running process and get a handle (BPatch_process
 *    object)
 * 2) Get a handle for the parsed image of the mutatee for function
 *    lookup (BPatch_image object)
 * 3) Open a file for output
 *    3a) Look up the "open" function
 *    3b) Build a code snippet to call open with the file name.
 *    3c) Run that code snippet via a oneTimeCode, saving the returned
 *        file descriptor
 * 4) Write the returned file descriptor into a memory variable for
 *    mutatee-side use
 * 5) Build a snippet that copies output to the file
 *    5a) Locate the "write" library call
 *    5b) Access its parameters
 *    5c) Build a snippet calling write(fd, parameters)
 *    5d) Insert the snippet at write
 * 6) Add a hook to exit to ensure that we close the file (using
 *    a callback at exit and another oneTimeCode)
```

```
 */

void usage() {
    fprintf(stderr, "Usage: retee <process pid> <filename>\n");
    fprintf(stderr, "   note: <filename> is relative to the application pro-
cess.\n");
}

// We need to use a callback, and so the things that callback requires
// are made global - this includes the file descriptor snippet (see below)
BPatch_variableExpr *fdVar = NULL;

// Before we add instrumentation, we need to open the file for
// writing. We can do this with a oneTimeCode - a piece of code run at
// a particular time, rather than at a particular location.

int openFileForWrite(BPatch_process *app, BPatch_image *appImage, char
*fileName) {
    // The code to be generated is:
    // fd = open(argv[2], O_WRONLY|O_CREAT, 0666);

    // (1) Find the open function
    std::vector<BPatch_function *>openFuncs;
    appImage->findFunction("open", openFuncs);
    if (openFuncs.size() == 0) {
        fprintf(stderr, "ERROR: Unable to find function for open()\n");
        return -1;
    }

    // (2) Allocate a vector of snippets for the parameters to open
    std::vector<BPatch_snippet *> openArgs;

    // (3) Create a string constant expression from argv[3]
    BPatch_constExpr fileNameExpr(fileName);

    // (4) Create two more constant expressions _WRONLY|O_CREAT and 0666
    BPatch_constExpr fileFlagsExpr(O_WRONLY|O_CREAT);
    BPatch_constExpr fileModeExpr(0666);

    // (5) Push 3 & 4 onto the list from step 2, push first to last parame-
ter.
    openArgs.push_back(&fileNameExpr);
    openArgs.push_back(&fileFlagsExpr);
    openArgs.push_back(&fileModeExpr);

    // (6) create a procedure call using function found at 1 and
    // parameters from step 5.
    BPatch_funcCallExpr openCall(*openFuncs[0], openArgs);

    // (7) The oneTimeCode returns whatever the return result from
    // the BPatch_snippet is. In this case, the return result of
    // open -> the file descriptor.
    void *openFD = app->oneTimeCode( openCall );

    // oneTimeCode returns a void *, and we want an int file handle
    return (int) (long) openFD;
}
```

```
// We have used a oneTimeCode to open the file descriptor. However,
// this returns the file descriptor to the mutator - the mutatee has
// no idea what the descriptor is. We need to allocate a variable in
// the mutatee to hold this value for future use and copy the
// (mutator-side) value into the mutatee variable.

// Note: there are alternatives to this technique. We could have
// allocated the variable before the oneTimeCode and augmented the
// snippet to do the assignment. We could also write the file
// descriptor as a constant into any inserted instrumentation.

BPatch_variableExpr *writeFileDescIntoMutatee(BPatch_process *app,
                                              BPatch_image *appImage,
                                              int fileDescriptor) {
    // (1) Allocate a variable in the mutatee of size (and type) int
    BPatch_variableExpr *fdVar = app->malloc(*appImage->findType("int"));
    if (fdVar == NULL) return NULL;

    // (2) Write the value into the variable
    // Like memcpy, writeValue takes a pointer
    // The third parameter is for functionality called "saveTheWorld",
    // which we don't worry about here (and so is false)
    bool ret = fdVar->writeValue((void *) &fileDescriptor, sizeof(int),
                                 false);
    if (ret == false) return NULL;

    return fdVar;
}

// We now have an open file descriptor in the mutatee. We want to
// instrument write to intercept and copy the output. That happens
// here.

bool interceptAndCloneWrite(BPatch_process *app,
                            BPatch_image *appImage,
                            BPatch_variableExpr *fdVar) {
    // (1) Locate the write call
    std::vector<BPatch_function *> writeFuncs;

    appImage->findFunction("write",
                           writeFuncs);
    if(writeFuncs.size() == 0) {
        fprintf(stderr, "ERROR: Unable to find function for write()\n");
        return false;
    }

    // (2) Build the call to (our) write. Arguments are:
    //   ours: fdVar (file descriptor)
    //   parameter: buffer
    //   parameter: buffer size

    // Declare a vector to hold these.
    std::vector<BPatch_snippet *> writeArgs;
    // Push on the file descriptor
    writeArgs.push_back(fdVar);
    // Well, we need the buffer... but that's a parameter to the
```

```
    // function we're implementing. That's not a problem - we can grab
    // it out with a BPatch_paramExpr.
    BPatch_paramExpr buffer(1); // Second (0, 1, 2) argument
    BPatch_paramExpr bufferSize(2);
    writeArgs.push_back(&buffer);
    writeArgs.push_back(&bufferSize);

    // And build the write call
    BPatch_funcCallExpr writeCall(*writeFuncs[0], writeArgs);

    // (3) Identify the BPatch_point for the entry of write. We're
    // instrumenting the function with itself; normally the findPoint
    // call would operate off a different function than the snippet.

    std::vector<BPatch_point *> *points;
    points = writeFuncs[0]->findPoint(BPatch_entry);
    if ((*points).size() == 0) {
        return false;
    }

    // (4) Insert the snippet at the start of write

    return app->insertSnippet(writeCall, *points);

    // Note: we have just instrumented write() with a call to
    // write(). This would ordinarily be a _bad thing_, as there is
    // nothing to stop infinite recursion - write -> instrumentation
    // -> write -> instrumentation....
    // However, Dyninst uses a feature called a "tramp guard" to
    // prevent this, and it's on by default.
}

// This function is called as an exit callback (that is, called
// immediately before the process exits when we can still affect it)
// and thus must match the exit callback signature:
//
// typedef void (*BPatchExitCallback) (BPatch_thread *, BPatch_exitType)
//
// Note that the callback gives us a thread, and we want a process - but
// each thread has an up pointer.

void closeFile(BPatch_thread *thread, BPatch_exitType) {
    fprintf(stderr, "Exit callback called for process...\n");

    // (1) Get the BPatch_process and BPatch_images
    BPatch_process *app = thread->getProcess();
    BPatch_image *appImage = app->getImage();

    // The code to be generated is:
    // close(fd);

    // (2) Find close
    std::vector<BPatch_function *> closeFuncs;
    appImage->findFunction("close", closeFuncs);
    if (closeFuncs.size() == 0) {
        fprintf(stderr, "ERROR: Unable to find function for close()\n");
        return;
```

```
    }

    // (3) Allocate a vector of snippets for the parameters to open
    std::vector<BPatch_snippet *> closeArgs;

    // (4) Add the fd snippet - fdVar is global since we can't
    // get it via the callback
    closeArgs.push_back(fdVar);

    // (5) create a procedure call using function found at 1 and
    // parameters from step 3.
    BPatch_funcCallExpr closeCall(*closeFuncs[0], closeArgs);

    // (6) Use a oneTimeCode to close the file
    app->oneTimeCode( closeCall );

    // (7) Tell the app to continue to finish it off.
    app->continueExecution();

    return;
}

BPatch bpatch;

// In main we perform the following operations.
// 1) Attach to the process and get BPatch_process and BPatch_image
//    handles
// 2) Open a file descriptor
// 3) Instrument write
// 4) Continue the process and wait for it to terminate

int main(int argc, char *argv[]) {
    int pid;
    if (argc != 3) {
        usage();
        exit(1);
    }
    pid = atoi(argv[1]);

    // Attach to the program - we can attach with just a pid; the
    // program name is no longer necessary
    fprintf(stderr, "Attaching to process %d...\n", pid);
    BPatch_process *app = bpatch.processAttach(NULL, pid);

    if (!app) return -1;

    // Read the program's image and get an associated image object
    BPatch_image *appImage = app->getImage();
    std::vector<BPatch_function*> writeFuncs;

    fprintf(stderr, "Opening file %s for write...\n", argv[2]);
    int fileDescriptor = openFileForWrite(app, appImage, argv[2]);

    if (fileDescriptor == -1) {
        fprintf(stderr, "ERROR: opening file %s for write failed\n",
                argv[2]);
        exit(1);
```

```
    }

    fprintf(stderr, "Writing returned file descriptor %d into"
                    "mutatee...\n", fileDescriptor);

    // This was defined globally as the exit callback needs it.
    fdVar = writeFileDescIntoMutatee(app, appImage, fileDescriptor);
    if (fdVar == NULL) {
        fprintf(stderr, "ERROR: failed to write mutatee-side variable\n");
        exit(1);
    }

    fprintf(stderr, "Instrumenting write...\n");
    bool ret = interceptAndCloneWrite(app, appImage, fdVar);
    if (!ret) {
        fprintf(stderr, "ERROR: failed to instrument mutatee\n");
        exit(1);
    }

    fprintf(stderr, "Adding exit callback...\n");
    bpatch.registerExitCallback(closeFile);

    // Continue the execution...
    fprintf(stderr, "Continuing execution and waiting for termination\n");
    app->continueExecution();

    while (!app->isTerminated())
        bpatch.waitForStatusChange();

    printf("Done.\n");

    return 0;
}
```

Appendix B - Running the Test Cases

This section describes how to run the Dyninst test cases. The primary purpose of the test cases is to verify that the API has been installed correctly (and for use in regression testing by the developers of the Dyninst library). The code may also be of use to others since it provides a fairly complete example of how to call most of the API methods. The test suite consists of mutator programs and their associated mutatee programs.

To compile the test suite, type `make` in the appropriate platform specific directory under dyninst/testsuite. To run, execute `runTests`. Each test will be executed and the result (PASSED/FAILED/CRASHED) printed.

Test mutators are run by the `test_driver` executable (`test_driver.exe` on Windows). The test_driver loads a mutator test from a shared object and runs it on a test mutatee. A single run of the test_driver may execute multiple tests (depending on parameters passed), and each test may execute multiple times with different parameters and on different mutatees.

Dyninst's test space can be very large. Each mutatee can be run under different tests, compiled by different compilers, and run with different parameters. For example, one point in this space would be the test1 mutatee being run under under test1_13, when compiled with the g++ compiler, and in attach mode. When run without any options, the `test_driver` will run all test combinations that are valid on the current platform. Many of the options that are passed to `test_driver` can be used to limit the test space that it runs in.

In order to prevent a crashing test from stopping the `test_driver` from running subsequent tests, `test_driver` can be run under a wrapper application, `runTests`. The `runTests` wrapper invokes the `test_driver` with the any arguments that were passed to `runTests`. It will watch the `test_driver` process, and if `test_driver` exits with a fault it will print an appropriate error message and restart the `test_driver` on the next test.

It is generally recommended that `runTests` be used when running a large sequence of tests, and `test_driver` be used when debugging issues with a single test.

The `test_driver` and `runTests` applications can be invoked with the following list of arguments. Most arguments are used to limit the space of tests that the testsuite will run. For example, to run the above test1_13 example, you could use the following command line:

```
test_driver -run test1_13 -mutatee test1.mutatee_g++ -attach
```

-attach

Only run tests that attach to the mutatees.

-create

Only run tests that create mutatees.

-rewriter

Only run tests that rewrite mutatees.

-staticlink

Run rewriter tests that use statically linked mutatees.

-dynamiclink

Run rewriter tests that use dynamically linked mutatees.

-allmode

Run tests for all modes (create, attach, rewriter on statically linked binaries, rewriter on dynamically linked binaries).

-gcc, -g++, -pgcc, -pgCC, -icc, -icpc

Run tests on mutatees built with the specified compiler.

-noclean

Don't remove rewritten mutatees after running rewriter tests.

-all

Run tests for all possible combinations of unoptimized mutatees.

-none, -low, -high, -max

Only run tests for mutatees of the given optimization level.

-allopt

Run tests for all mutatee optimization levels.

-full

Run tests for all possible combinations of mutatees, including all optimization levels. Requires `make all` to build the optimized mutatees.

-dyninst, -symtab, -instruction, -proccontrol, -stackwalker

Only run tests for the specified component.

-allcomp

Run tests for all components.

-32, -64

Only run tests for 32-bit or 64-bit mutatees. This option is only valid on platforms such as AMD64/Linux and PowerPC/Linux where both 32-bit and 64-bit build environments are available.

-pic

Only run tests for mutatees compiled as position-independent code. Default is non-PIC mutatees. –all and –full include both.

-sp, -mp, -st, -mt

ProcControl-specific: run tests in single process, multiprocess, single thread, multithread modes, respectively.

-j n

This option spawns up to n test_driver instances from a given runTests invocation. This option is highly recommended for large test runs.

-hosts host1 [host2 ... host*n*]

In conjunction with the –j option above, will distribute tests over the hosts *host1*...*host*n. The hosts must share a filesystem with the machine from which runTests is being run, and ssh must be configured to allow password-less authentication to those hosts. Note that –j controls the total number of test_driver instances, not the number per host, so you will need to use a –j N at least equal to the number of hosts you wish to use.

-mutatee <mutatee_name>

Only run tests that use the specified mutatee name. Only certain mutatees can be run with certain tests. The primary test number specifies which mutatees it can be run with. For example, all of the test1_* tests can be run with the test1.mutatee_* mutatees, and all of the test2_* tests can be run with the test2.mutatee_* mutatees.

-run <subtest> <subtest> …

Only runs the specific sub-tests listed. For example, to run sub-test case 4 of test2 you would enter `test_driver -run test2_4`.

-test

Alias for `-run`.

-log

Print more detailed output, including messages generated by the tests. Without this option the testsuite will capture and hide any messages printed by the test, only showing a summary of whether the test passed or failed. By default, output is sent to `stdout`.

-logfile <filename>

Send output from the `-log` option to the given filename rather than to `stdout`.

-verbose

Enables test suite debugging output. This is useful when trying to track down issues in the test suite or tests.

Appendix C - Common pitfalls

This appendix is designed to point out some common pitfalls that users have reported when using the Dyninst system. Many of these are either due to limitations in the current implementations, or reflect design decisions that may not produce the expected behavior from the system.

Attach followed by detach

If a mutator attaches to a mutatee, and immediately exits, the current behavior is that the mutatee is left suspended. To make sure the application continues, call detach with the appropriate flags.

Attaching to a program that has already been modified by Dyninst

If a mutator attaches to a program that has already been modified by a previous mutator, a warning message will be issued. We are working to fix this problem, but the correct semantics are still being specified. Currently, a message is printed to indicate that this has been attempted, and the attach will fail.

Dyninst is event-driven

Dyninst must sometimes handle events that take place in the mutatee, for instance when a new shared library is loaded, or when the mutatee executes a fork or exec. Dyninst handles events when it checks the status of the mutatee, so to allow this the mutator should periodically call one of the functions `BPatch::pollForStatusChange`, `BPatch::waitForStatusChange`, `BPatch_thread::isStopped`, or `BPatch_thread::isTerminated`.

Missing or out-of-date DbgHelp DLL (Windows)

Dyninst requires an up-to-date DbgHelp library on Windows. See the section on Windows-specific architectural issues for details.

Portland Compiler Group – missing debug symbols

The Portland Group compiler (pgcc) on Linux produces debug symbols that are not read correctly by Dyninst. The binaries produced by the compiler do not contain the source file information necessary for Dyninst to assign the debug symbols to the correct module.

I

M

O

P

R

S

T

U

W

References

1. B. Buck and J. K. Hollingsworth, "An API for Runtime Code Patching," *Journal of Supercomputing Applications (to appear)*, 2000.
2. J. K. Hollingsworth and B. P. Miller, "Using Cost to Control Instrumentation Overhead," *Theoretical Computer Science*, **196**(1-2), 1998, pp. 241-258.
3. J. K. Hollingsworth, B. P. Miller, and J. Cargille, "Dynamic Program Instrumentation for Scalable Performance Tools," *1994 Scalable High-Performance Computing Conf.*, Knoxville, Tenn., pp. 841-850.
4. J. K. Hollingsworth, B. P. Miller, M. J. R. Goncalves, O. Naim, Z. Xu, and L. Zheng, "MDL: A Language and Compiler for Dynamic Program Instrumentation," *International Conference on Parallel Architectures and Compilation Techniques (PACT)*. Nov. 1997, San Francisco, pp. 201-212.
5. J. R. Larus and E. Schnarr, "EEL: Machine-Independent Executable Editing," *PLDI*. June 18-21, 1995, La Jolla, CA, ACM, pp. 291-300.

Paradyn Parallel Performance Tools

DynC API Programmer's Guide

9.2 Release
June 2016

Computer Sciences Department
University of Wisconsin–Madison
Madison, WI 53706

Computer Science Department
University of Maryland
College Park, MD 20742

Email dyninst-api@cs.wisc.edu
Web www.dyninst.org

Contents

1 DynC API

1.1 Motivation

Dyninst is a powerful instrumentation tool, but specifying instrumentation code (known as an Abstract Syntax Tree) in the `BPatch_snippet` language can be cumbersome. DynC API answers these concerns, enabling a programmer to easily and quickly build `BPatch_snippets` using a simple C-like language. Other advantages to specifying `BPatch_snippets` using dynC include cleaner, more readable mutator code, automatic variable handling, and runtime-compiled snippets.

As a motivating example, the following implements a function tracer that notifies the user when entering and exiting functions, and keeps track of the number of times each function is called.

1.1.1 Dyninst API

When creating a function tracer using the Dyninst API, the programmer must perform many discrete lookups and create many `BPatch_snippet objects`, which are then combined and inserted into the mutatee.

Look up `printf`:

```
1  std::vector<BPatch_function *> *printf_func;
   appImage->findFunction("printf", printf_func);
   BPatch_function *BPF_printf = printf_func[0];
```

Create each `printf` pattern:

```
   BPatch_constExpr entryPattern("Entering %s, called %d times.\n");
2  BPatch_constExpr exitPattern("Exiting %s.\n");
```

For each function, do the following:

Create snippet vectors:

```
std::vector<BPatch_snippet *> entrySnippetVect;
std::vector<BPatch_snippet *> exitSnippetVect;
```

Create the `intCounter` global variable:

```
appProc->malloc(appImage->findType("int"), std::string("intCounter"));
```

Get the name of the function:

```
char fName[128];
BPatch_constExpr funcName(functions[i]->getName(fName, 128));
```

Build the entry `printf`:

```
std::vector<BPatch_snippet *> entryArgs;
entryArgs.push_back(entryPattern);
3  entryArgs.push_back(funcName);
entryArgs.push_back(intCounter);
```

Build the exit `printf`:

```
1       std::vector<BPatch_snippet *> exitArgs;
        exitArgs.push_back(exitPattern);
        exitArgs.push_back(funcName);
```

Add `printf` to the snippet:

```
        entrySnippetVect.push_back(BPatch_functionCallExpr(*printf_func, entryArgs));
2       exitSnippetVect.push_back(BPatch_functionCallExpr(*printf_func, exitArgs));
```

Increment the counter:

```
        BPatch_arithExpr addOne(BPatch_assign, *intCounter,
               BPatch_arithExpr(BPatch_plus, *intCounter, BPatch_constExpr(1)));
```

Add increment to the entry snippet:

```
        entrySnippetVect.push_back(&addOne);
```

Insert the snippets:

```
        appProc->insertSnippet(*entrySnippetVect, functions[i]->findPoint(BPatch_entry));
        appProc->insertSnippet(*exitSnippetVect, functions[i]->findPoint(BPatch_exit));
```

1.1.2 DynC API

A function tracer is much easier to build in DynC API, especially if reading dynC code from file. Storing dynC code in external files not only cleans up mutator code, but also allows the programmer to modify snippets without recompiling.

In this example, the files `myEntryDynC.txt` and `myExitDynC.txt` contain dynC code:

```
// myEntryDynC.txt
static int intCounter;
printf("Entering %s, called %d times.\n", dyninst‘function_name, intCounter++);

// myExitDynC.txt
printf("Leaving %s.\n", dyninst‘function_name);
```

The code to read, build, and insert the snippets would look something like the following:

First open files:

```
FILE *entryFile = fopen("myEntryDynC.txt", "r");
FILE *exitFile = fopen("myExitDynC.txt", "r");
```

Next call DynC API with each function's entry and exit points:

```
BPatch_snippet *entrySnippet =
     dynC_API::createSnippet(entryFile, entryPoint, "entrySnippet");
BPatch_snippet *exitSnippet =
     dynC_API::createSnippet(exitFile, exitPoint, "exitSnippet");
```

Finally insert the snippets at each function's entry and exit points:

```
appProc->insertSnippet(*entrySnippet, entryPoint);
appProc->insertSnippet(*exitSnippet, exitPoint);
```

1.2 Calling DynC API

All DynC functions reside in the `dynC_API` namespace. The primary DynC API function is:

```
BPatch_Snippet *createSnippet(<dynC code>, <location>, char * name);
```

where `<dynC code>` can be either a constant c-style string or a file descriptor and `<location>` can take the form of a `BPatch_point` or a `BPatch_addressSpace`. There is also an optional parameter to name a snippet. A snippet name makes code and error reporting much easier to read, and allows for the grouping of snippets (see section 2.3.2). If a snippet name is not specified, the default name `Snippet_[<#>]` is used.

The location parameter is the point or address space in which the snippet will be inserted. Inserting a snippet created for one location into another can cause undefined behavior.

<dynC code>	Description
std::string str	A C++ string containing dynC code.
const char *s	A null terminated string containing dynC code
FILE *f	A standard C file descriptor. Facilitates reading dynC code from file.

Table 1: createSnippet(...) input options: dynC code

<location>	Description
BPatch_point &point	Creates a snippet specific to a single point.
BPatch_addressSpace &addSpace	Creates a more flexible snippet specific to an address space. See Section 1.3.

Table 2: createSnippet(...) input options: location

1.3 Creating Snippets Without Point Information

Creating a snippet without point information (i.e., calling createSnippet(...) with a BPatch_addressSpace) results in a far more flexible snippet that may be inserted at any point in the specified address space. There are, however, a few restrictions on the types of operations that may be performed by a flexible snippet. No local variables may be accessed, including parameters and return values. Mutatee variables must be accessed through the global domain.

2 DynC Language Description

The DynC language is a subset of C with a **domain** specification for selecting the location of a resource.

2.1 Domains

Domains are special keywords that allow the programmer to precisely indicate which resource to use. DynC domains follow the form of <domain>‘<identifier>, with a back-tick separating the domain and the identifier. The DynC domains are as follows:

Domain	Description
inf	The inferior process (the program being instrumented). Allows access to functions of the mutatee and it's loaded libraries.
dyninst	Dyninst utility functions. Allows access to context information as well as the break() function. See Appendix A.
local	A mutatee variable local to function in which the snippet is inserted.
global	A global mutatee variable.
param	A parameter of the mutatee function in which the snippet is inserted.
default	The default domain (domain not specified) is the domain of snippet variables.

Table 3: DynC API Domains

Example:

```
inf‘printf("n is equal to %d.\n", ++global‘n);
```

This would increment and print the value of the mutatee global variable n.

2.2 Control Flow

2.2.1 Comments

Block and line comments work as they do in C or C++.

Example:

```
/*
 * This is a comment.
 */
int i; // So is this.
```

2.2.2 Conditionals

Use `if` to conditionally execute code. Example:

```
if(x == 0){
   inf‘printf("x == 0.\n");
}
```

The `else` command can be used to specify code executed if a condition is not true. Example:

```
if(x == 0){
   inf‘printf("x == 0.\n");
}else if(x > 3){
   inf‘printf("x > 3.\n");
}else{
   inf‘printf("x < 3 but x }= 0.\n");
}
```

2.2.3 First-Only Code Block

Code enclosed by a pair of `% <code> %` is executed only once by a snippet. First-only code blocks can be useful for declaring and initilizing variables, or for any task that needs to be executed only once. Any number of first-only code blocks can be used in a dynC code snippet.

A first-only code block is equivalent to the following:

```
static int firstTime = 0;
if(firstTime == 0){
  <code>
  firstTime = 1;
}
```

DynC will only execute the code in a first-only section the first time a snippet is executed. If `createSnippet(...)` is called multiple times and is passed the same name, then the first-only code will be executed only once: the first time that any of those snippets with the same name is executed. In contrast, if a snippet is created by calling `createSnippet(...)` with a unique snippet name (or if a name is unspecified), the first-only code will be executed only once upon reaching the first point encountered in the execution of the mutatee where the returned `BPatch_Snippet` is inserted.

Example Touch:

```
{%
   inf‘printf("Function %s has been touched.\n", dyninst‘function_name);
%}
```

If `createSnippet(...)` is passed the code in Example Touch and the name `"fooTouchSnip"` and the returned `BPatch_snippet` is inserted at the entry to function `foo`, the output would be:

```
Function foo has been touched.
(mutatee exit)
```

If the dynC code in Example Touch is passed to `createSnippet(...)` multiple times and each snippet is given the same name, but is inserted at the entries of the functions `foo`, `bar`, and `run` respectively, the output would be:

```
Function foo has been touched.
(mutatee exit)
```

Creating the snippets with distinct names (e.g. `createSnippet(...)` is called with the dynC code in Example Touch multiple times and the snippets are named `"fooTouchSnip"`, `"barTouchSnip"`, `"runTouchSnip"`) would produce an output like:

```
Function foo has been touched.
Function bar has been touched.
Function run has been touched.
(mutatee exit)
```

A cautionary note: the use of first-only blocks can be expensive, as a conditional must be evaluated each time the snippet is executed. If the option is available, one may opt to insert a dynC snippet initializing all global variables at the entry point of `main`.

2.3 Variables

DynC allows for the creation of *snippet local* variables. These variables are in scope only within the snippet in which they are created.

For example,

```
int i;
i = 5;
```

would create an uninitialized variable named `i` of type integer. The value of `i` is then set to 5. This is equivalent to:

```
int i = 5;
```

2.3.1 Static Variables

Every time a snippet is executed, non-static variables are reinitialized. To create a variable with value that persists across executions of snippets, declare the variable as static.

Example:

```
int i = 10;
inf‘printf("i is %d.\n", i++);
```

If the above is inserted at the entrance to a function that is called four times, the output would be:

```
i is 10.
i is 10.
i is 10.
i is 10.
```

Adding `static` to the variable declaration would make the value of `i` persist across executions:

```
static int i = 10;
inf‘printf("i is %d.\n", i++);
```

Produces:

```
i is 10.
i is 11.
i is 12.
i is 13.
```

A variable declared in a first-only section will also behave statically, as the initialization occurs only once.

```
{%
   int i = 10;
%}
```

2.3.2 An Explanation of the Internal Workings of DynC Variable Creation

DynC uses the DyninstAPI function `malloc(...)` to allocate dynC declared variables when `createSnippet(...)` is called. The variable name is mangled with the name of the snippet passed to createSnippet. Thus, variables declared in dynC snippets are accessible only to those snippets created by calling `createSnippet(...)` with the same name.

If the variables are explicitly initialized, dynC sets the value of the variable with a `BPatch_arithExpr(BPatch_assign...)` snippet. Because of this, each time the snippet is executed, the value is reset to the initialized value. If, however the variables are not explicitly initialized, they are automatically set to a type-specific zero-value. Scalar variables are set to 0, and c-strings are set to empty, null-terminated strings (i.e. `""`).

If a variable is declared with the `static` keyword, then the initialization is performed as if in a first-only block (see section 2.2.3). Thus, a variable is initialized only the first time that snippet is executed, and subsequent executions of the variable initialization are ignored.

2.3.3 Creating Global Variables That Work With DynC

To declare a global variable that is accessible to all snippets inserted into a mutatee, one must use the DyninstAPI `BPatch_addressSpace::malloc(...)` method (see Dyninst Programmer's Guide). This code is located in mutator code (*not* in dynC code).

myMutator.C:

```
...
// Creates a global variable of type in named globalIntN
myAddressSpace->malloc(myImage->getType("int"), "globalIntN");

// file1 and file2 are FILE *, entryPoint and exitPoint are BPatch_point
BPatch_snippet *snippet1 = dynC::createSnippet(file1, &entryPoint, "mySnippet1");
BPatch_snippet *snippet2 = dynC::createSnippet(file2, &exitPoint, "mySnippet2");

assert(snippet1);
assert(snippet2);

myAdressSpace->insertSnippet(snippet1, &entryPoint);
myAdressSpace->insertSnippet(snippet2, &exitPoint);

// run the mutatee
((BPatch_process *)myAdressSpace)->continueExecution();
...
```

file1:

```
{%
   global‘globalIntN = 0; // initialize global variable in first-only section
%}
inf‘printf("Welcome to function %s. Global variable globalIntN = %d.\n",
     dyninst‘function_name, global‘globalIntN++);
```

file2:

```
inf‘printf("Goodbye from function %s. Global variable globalIntN = %d.\n",
     dyninst‘function_name, global‘globalIntN++);
```

When run, the output from the instrumentation would be:

```
Welcome to function foo. Global variable globalIntN = 0.
Goodbye from function foo. Global variable globalIntN = 1.
Welcome to function foo. Global variable globalIntN = 2.
Goodbye from function foo. Global variable globalIntN = 3.
Welcome to function foo. Global variable globalIntN = 4.
Goodbye from function foo. Global variable globalIntN = 5.
```

2.3.4 Data Types

DynC supported data types are restricted by those supported by Dyninst: `int`, `long`, `char *`, and `void *`. Integer and c-string primitives are also recognized:

Example:

```
int i = 12;
char *s = "hello";
```

2.3.5 Pointers

Pointers are dereferenced with the prefix `*<variable>` and the address of variable is specified by `&<variable>`. For example, in reference to the previous example from section 2.3.4, the statement `*s` would evaluate to the character `h`.

2.3.6 Arrays

Arrays in DynC behave much the same way they do in C.

Example:

```
int array[3] = {1, 2, 3};
char *names[] = {"Mark", "Phil", "Deb", "Tracy"};
names[2] = "Gwen" // change Deb to Gwen
inf‘printf("The seventh element of mutArray is %d.\n", global‘mutArray[6]); //Mutatee array
if(inf‘strcmp(*names, "Mark") == 0){} // This will evaluate to true.
```

2.4 DynC Limitations

The DynC, while quite expressive, is limited to those actions supported by the DyninstAPI. As such, it lacks certain abilities that many programmers have come to expect. These differences will be discussed in an exploration of those C abilities that dynC lacks.

2.4.1 Loops

There are no looping structures in DynC.

2.4.2 Enums, Unions, Structures

These features present a unique implementation challenge and are in development. Look to future revisions for full support for enums, unions, and structures.

2.4.3 Preprocessing

DynC does not allow C-style preprocessing macros or importation. Rather than `#define` statements, constant variables are recommended.

2.4.4 Functions

Specifying functions is beyond the scope of the DynC language. DyninstAPI has methods for dynamically loading code into a mutatee, and these loaded functions can be used in DynC snippets.

A The Dyninst Domain

The `dyninst` domain has quite a few useful values and functions:

Identifier	Type	Where Valid	Description
`function_name`	`char *`	Within a function	Evaluates to the name of the current function. Call to `createSnippet(...)` must specify a `BPatch_point`.
`module_name`	`char *`	Anywhere	Evaluates to the name of the current module. Call to `createSnippet(...)` must specify a `BPatch_point`.
`bytes_accessed`	int	At a memory operation	Evaluates to the number of bytes accessed by a memory operation.
`effective_address`	`void *`	At a memory operation	Evaluates the effective address of a memory operation.
`original_address`	`void *`	Anywhere	Evaluates to the original address where the snippet was inserted.
`actual_address`	`void *`	Anywhere	Evaluates to the actual address of the instrumentation.
`return_value`	`void *`	Function exit	Evaluates to the return value of a function.
`thread_index`	int	Anywhere	Returns the index of the thread the snippet is executing on.
`tid`	int	Anywhere	Returns the id of the thread the snippet is executing on.
`dynamic_target`	`void *`	At calls, jumps, returns	Calculates the target of a control flow instruction.
`break()`	void	Anywhere	Causes the mutatee to execute a breakpoint.
`stopthread()`	void	Anywhere	Stops the thread on which the snippet is executing.

Table 4: Dyninst Domain Values

Paradyn Parallel Performance Tools

ParseAPI Programmer's Guide

9.2 Release
June 2016

Computer Sciences Department
University of Wisconsin–Madison
Madison, WI 53706

Computer Science Department
University of Maryland
College Park, MD 20742

Email dyninst-api@cs.wisc.edu
Web www.dyninst.org

Contents

1 Introduction

A binary code parser converts the machine code representation of a program, library, or code snippet to abstractions such as the instructions, basic blocks, functions, and loops that the binary code represents. The ParseAPI is a multi-platform library for creating such abstractions from binary code sources. The current incarnation uses the Dyninst SymtabAPI as the default binary code source; all platforms and architectures handled by the SymtabAPI are supported. The ParseAPI is designed to be easily extensible to other binary code sources. Support for parsing binary code in memory dumps or other formats requires only implementation of a small interface as described in this document.

This API provides the user with a control flow-oriented view of a binary code source. Each code object such as a program binary or library is represented as a top-level collection containing the loops, functions, basic blocks, and edges that represent the control flow graph. A simple query interface is provided for retrieving lower level objects like functions and basic blocks through address or other attribute lookups. These objects can be used to navigate the program structure as described below.

2 Abstractions

The basic representation of code in this API is the control flow graph (CFG). Binary code objects are represented as regions of contiguous bytes that, when parsed, form the nodes and edges of this graph. The following abstractions make up this CFG-oriented representation of binary code:

- BLOCK: Nodes in the CFG represent *basic blocks*: straight line sequences of instructions $I_i \ldots I_j$ where for each $i < k \leq j$, I_k postdominates I_{k-1}. Importantly, on some instruction set architectures basic blocks can *overlap* on the same address range—variable length instruction sets allow for multiple interpretations of the bytes making up the basic block.
- EDGE: Typed edges between the nodes in the CFG represent execution control flow, such as conditional and unconditional branches, fallthrough edges, and calls and returns. The graph therefore represents both *inter-* and *intraprocedural* control flow: traversal of nodes and edges can cross the boundaries of the higher level abstractions like *functions*.
- FUNCTION: The *function* is the primary semantic grouping of code in the binary, mirroring the familiar abstraction of procedural languages like C. Functions represent the set of all basic blocks reachable from a *function entry point* through intraprocedural control flow only (that is, no calls or returns). Function entry points are determined in a variety of ways, such as hints from debugging symbols, recursive traversal along call edges and a machine learning based function entry point identification process.
- LOOP: The *loop* represents code in the binary that may execute repeatedly, corresponding to source language constructs like *for* loop or *while* loop. We use a formal definition of loops from “Nesting of Reducible and Irreducible Loops" by Paul Havlak. We support identifying both natural loops (single-entry loops) and irreducible loops (multi-entry loops).

- CODE OBJECT: A collection of distinct code regions are represented as a single code object, such as an executable or library. Code objects can normally be thought of as a single, discontiguous unique address space. However, the ParseAPI supports code objects in which the different regions have overlapping address spaces, such as UNIX archive files containing unlinked code.
- INSTRUCTION SOURCE: An instruction source describes a backing store containing binary code. A binary file, a library, a memory dump, or a process's executing memory image can all be described as an instruction source, allowing parsing of a variety of binary code objects.
- CODE SOURCE: The code source implements the instruction source interface, exporting methods that can access the underlying bytes of the binary code for parsing. It also exports a number of additional helper methods that do things such as returning the location of structured exception handling routines and function symbols. Code sources are tailored to particular binary types; the ParseAPI provides a SymtabAPI-based code source that understands ELF, COFF and PE file formats.

3 Examples

3.1 Function Disassembly

The following example uses ParseAPI and InstructionAPI to disassemble the basic blocks in a function. As an example, it can be built with G++ as follows: `g++ -std=c++0x -o code_sample code_sample.cc -L<library install path> -I<headers install path> -lparseAPI -linstructionAPI -lsymtabAPI -lsymLite -ldynDwarf -ldynElf -lcommon -L<libelf path> -lelf -L<libdwarf path> -ldwarf`. Note: this example must be compiled with C++11x support; for G++ this is enabled with `-std=c++0x`, and it is on by default for Visual Studio.

```
/*
   Copyright (C) 2015 Alin Mindroc
   (mindroc dot alin at gmail dot com)

   This is a sample program that shows how to use InstructionAPI in order to
   print the assembly code and functions in a provided binary.


   This program is free software; you can redistribute it and/or
   modify it under the terms of the GNU Lesser General Public
   License as published by the Free Software Foundation; either
   version 2.1 of the License, or (at your option) any later version.
   */
#include <iostream>
#include "CodeObject.h"
#include "InstructionDecoder.h"
using namespace std;
using namespace Dyninst;
using namespace ParseAPI;

using namespace InstructionAPI;
```

```
int main(int argc, char **argv){
    if(argc != 2){
        printf("Usage: %s <binary path>\n", argv[0]);
        return -1;
    }
    char *binaryPath = argv[1];

    SymtabCodeSource *sts;
    CodeObject *co;
    Instruction::Ptr instr;
    SymtabAPI::Symtab *symTab;
    std::string binaryPathStr(binaryPath);
    bool isParsable = SymtabAPI::Symtab::openFile(symTab, binaryPathStr);
    if(isParsable == false){
        const char *error = "error: file can not be parsed";
        cout << error;
        return - 1;
    }
    sts = new SymtabCodeSource(binaryPath);
    co = new CodeObject(sts);
    //parse the binary given as a command line arg
    co->parse();

    //get list of all functions in the binary
    const CodeObject::funclist &all = co->funcs();
    if(all.size() == 0){
        const char *error = "error: no functions in file";
        cout << error;
        return - 1;
    }
    auto fit = all.begin();
    Function *f = *fit;
    //create an Instruction decoder which will convert the binary opcodes to strings
    InstructionDecoder decoder(f->isrc()->getPtrToInstruction(f->addr()),
            InstructionDecoder::maxInstructionLength,
            f->region()->getArch());
    for(;fit != all.end(); ++fit){
        Function *f = *fit;
        //get address of entry point for current function

        Address crtAddr = f->addr();
        int instr_count = 0;
        instr = decoder.decode((unsigned char *)f->isrc()->getPtrToInstruction(crtAddr));
        auto fbl = f->blocks().end();
        fbl--;
        Block *b = *fbl;
        Address lastAddr = b->last();
        //if current function has zero instructions, donâĂŹt output it
        if(crtAddr == lastAddr)
            continue;
```

```
            cout << "\n\n\"" << f->name() << "\" :";
            while(crtAddr < lastAddr){
                //decode current instruction
                instr = decoder.decode((unsigned char *)f->isrc()->getPtrToInstruction(crtAddr));
                cout << "\n" << hex << crtAddr;
                cout << ": \"" << instr->format() << "\"";
                //go to the address of the next instruction
                crtAddr += instr->size();
                instr_count++;
            }
        }
        return 0;
    }
```

3.2 Control flow graph traversal

The following complete example uses the ParseAPI to parse a binary and dump its control flow graph in the Graphviz file format. As an example, it can be built with G++ as follows: `g++ -std=c++0x -o example example.cc -L<library install path> -I<headers install path> -lparseAPI -linstructionAPI -lsymtabAPI -lsymLite -ldynDwarf -ldynElf -lcommon -L<libelf path> -lelf -L<libdwarf path> -ldwarf`. Note: this example must be compiled with C++11x support; for G++ this is enabled with `-std=c++0x`, and it is on by default for Visual Studio.

```
// Example ParseAPI program; produces a graph (in DOT format) of the
// control flow graph of the provided binary.
//
// Improvements by E. Robbins (er209 at kent dot ac dot uk)
//

#include <stdio.h>
#include <map>
#include <vector>
#include <unordered_map>
#include <sstream>
#include "CodeObject.h"
#include "CFG.h"

using namespace std;
using namespace Dyninst;
using namespace ParseAPI;

int main(int argc, char * argv[])
{
   map<Address, bool> seen;
   vector<Function *> funcs;
   SymtabCodeSource *sts;
   CodeObject *co;

   // Create a new binary code object from the filename argument
```

```
sts = new SymtabCodeSource( argv[1] );
co = new CodeObject( sts );

// Parse the binary
co->parse();
cout << "digraph G {" << endl;

// Print the control flow graph
const CodeObject::funclist& all = co->funcs();
auto fit = all.begin();
for(int i = 0; fit != all.end(); ++fit, i++) { // i is index for clusters
   Function *f = *fit;

   // Make a cluster for nodes of this function
   cout << "\t subgraph cluster_" << i
        << " { \n\t\t label=\""
        << f->name()
        << "\"; \n\t\t color=blue;" << endl;

   cout << "\t\t\"" << hex << f->addr() << dec
        << "\" [shape=box";
   if (f->retstatus() == NORETURN)
      cout << ",color=red";
   cout << "]" << endl;

   // Label functions by name
   cout << "\t\t\"" << hex << f->addr() << dec
        << "\" [label = \""
        << f->name() << "\\n" << hex << f->addr() << dec
        << "\"];" << endl;

   stringstream edgeoutput;

   auto bit = f->blocks().begin();
   for( ; bit != f->blocks().end(); ++bit) {
      Block *b = *bit;
      // Don't revisit blocks in shared code
      if(seen.find(b->start()) != seen.end())
         continue;

      seen[b->start()] = true;

      cout << "\t\t\"" << hex << b->start() << dec <<
         "\";" << endl;

      auto it = b->targets().begin();
      for( ; it != b->targets().end(); ++it) {

         std::string s = "";
         if((*it)->type() == CALL)
```

```
            s = " [color=blue]";
        else if((*it)->type() == RET)
            s = " [color=green]";

        // Store the edges somewhere to be printed outside of the cluster
        edgeoutput << "\t\""
                   << hex << (*it)->src()->start()
                   << "\" -> \""
                   << (*it)->trg()->start()
                   << "\"" << s << endl;
      }
    }
    // End cluster
    cout << "\t}" << endl;

    // Print edges
    cout << edgeoutput.str() << endl;
  }
  cout << "}" << endl;
}
```

3.3 Loop analysis

The following code example shows how to get loop information using ParseAPI once we have an parsed Function object.

```
void GetLoopInFunc(Function *f) {
    // Get all loops in the function
    vector<Loop*> loops;
    f->getLoops(loops);

    // Iterate over all loops
    for (auto lit = loops.begin(); lit != loops.end(); ++lit) {
        Loop *loop = *lit;

        // Get all the entry blocks of the loop
        vector<Block*> entries;
        loop->getLoopEntries(entries);

        // Get all the blocks in the loop
        vector<Block*> blocks;
        loop->getLoopBasicBlocks(blocks);

        // Get all the back edges in the loop
        vector<Edge*> backEdges;
        loop->getBackEdges(backEdges);
    }
}
```

4 The Parsing API

4.1 Class CodeObject

Defined in: `CodeObject.h`

The CodeObject class describes an individual binary code object, such as an executable or library. It is the top-level container for parsing the object as well as accessing that parse data. The following API routines and data types are provided to support parsing and retrieving parsing products.

```
typedef std::set<Function *, Function::less> funclist
```

Container for access to functions. Refer to Section 4.12 for details. Library users *must not* rely on the underlying container type of std::set, as it is subject to change.

```
CodeObject(CodeSource * cs,
           CFGFactory * fact = NULL,
           ParseCallback * cb = NULL,
           bool defensiveMode = false)
```

Constructs a new CodeObject from the provided CodeSource and optional object factory and callback handlers. Any parsing hints provided by the CodeSource are processed, but the binary is not parsed when this constructor returns.

The `defensiveMode` parameter optionally trades off coverage for safety; this mode is not recommended for most applications as it makes very conservative assumptions about control flow transfer instructions (see Section B).

```
void parse()
```

Recursively parses the binary represented by this CodeObject from all known function entry points (i.e., the hints provided by the CodeSource). This method and the following parsing methods may safely be invoked repeatedly if new information about function locations is provided through the CodeSource. Note that these parsing methods do not automatically perform speculative gap parsing. parseGaps should be used for this purpose.

```
void parse(Address target,
           bool recursive)
```

Parses the binary starting with the instruction at the provided target address. If `recursive` is TRUE, recursive traversal parsing is used as in the default `parse()` method; otherwise only instructions reachable through intraprocedural control flow are visited.

```
void parse(CodeRegion * cr,
           Address target,
           bool recursive)
```

Parses the specified core region of the binary starting with the instruction at the provided target address. If `recursive` is TRUE, recursive traversal parsing is used as in the default `parse()` method; otherwise only instructions reachable through intraprocedural control flow are visited.

```
struct NewEdgeToParse {
    Block *source;
    Address target;
    EdgeTypeEnum type;
}
bool parseNewEdges( vector<NewEdgeToParse> & worklist )
```

Parses a set of newly created edges specified in the worklist supplied that were not included when the function was originally parsed.

ParseAPI is able to speculatively parse gaps (regions of binary that has not been identified as code or data yet) to identify function entry points and perform control flow traversal.

GapParsingType	Technique description
PreambleMatching	If instruction patterns are matched at an adderss, the address is a function entry point
IdiomMatching	Based on a pre-trained model, this technique calculates the probability of an address to be a function entry point and predicts whether which addresses are function entry points

```
void parseGaps(CodeRegion *cr,
               GapParsingType type=IdiomMatching)
```

Speculatively parse the indicated region of the binary using the specified technique to find likely function entry points, enabled on the x86 and x86-64 platforms.

```
Function * findFuncByEntry(CodeRegion * cr,
                           Address entry)
```

Find the function starting at address `entry` in the indicated CodeRegion. Returns NULL if no such function exists.

```
int findFuncs(CodeRegion * cr,
              Address addr,
              std::set<Function*> & funcs)
```

Finds all functions spanning `addr` in the code region, adding each to `funcs`. The number of results of this stabbing query are returned.

```
int findFuncs(CodeRegion * cr,
              Address start,
              Address end,
              std::set<Function*> & funcs)
```

Finds all functions overlapping the range `[start,end)` in the code region, adding each to `funcs`. The number of results of this stabbing query are returned.

```
const funclist & funcs()
```

Returns a const reference to a container of all functions in the binary. Refer to Section 4.12 for container access details.

```
Block * findBlockByEntry(CodeRegion * cr,
                         Address entry)
```

Find the basic block starting at address `entry`. Returns NULL if no such block exists.

```
int findBlocks(CodeRegion * cr,
               Address addr,
               std::set<Block*> & blocks)
```

Finds all blocks spanning `addr` in the code region, adding each to `blocks`. Multiple blocks can be returned only on platforms with variable-length instruction sets (such as IA32) for which overlapping instructions are possible; at most one block will be returned on all other platforms.

```
Block * findNextBlock(CodeRegion * cr,
                      Address addr)
```

Find the next reachable basic block starting at address `entry`. Returns NULL if no such block exists.

```
CodeSource * cs()
```

Return a reference to the underlying CodeSource.

```
CFGFactory * fact()
```

Return a reference to the CFG object factory.

```
bool defensiveMode()
```

Return a boolean specifying whether or not defensive mode is enabled.

```
bool isIATcall(Address insn,
               std::string &calleeName)
```

Returns a boolean specifying if the address at `addr` is located at the call named in `calleeName`.

```
void startCallbackBatch()
```

Starts a batch of callbacks that have been registered.

```
void finishCallbackBatch()
```

Completes all callbacks in the current batch.

```
void registerCallback(ParseCallback *cb);
```

Register a callback `cb`

```
void unregisterCallback(ParseCallback *cb);
```

Unregister an existing callback `cb`

```
void finalize()
```

Force complete parsing of the CodeObject; parsing operations are otherwise completed only as needed to answer queries.

```
void destroy(Edge *)
```

Destroy the edge listed.

```
void destroy(Block *)
```

Destroy the code block listed.

```
void destroy(Function *)
```

Destroy the function listed.

4.2 Class CodeRegion

Defined in: `CodeSource.h`

The CodeRegion interface is an accounting structure used to divide CodeSources into distinct regions. This interface is mostly of interest to CodeSource implementors.

```
void names(Address addr,
           vector<std::string> & names)
```

Fills the provided vector with any names associated with the function at a given address in the region, e.g. symbol names in an ELF or PE binary.

```
virtual bool findCatchBlock(Address addr,
                            Address & catchStart)
```

Finds the exception handler associated with an address, if one exists. This routine is only implemented for binary code sources that support structured exception handling, such as the SymtabAPI-based SymtabCodeSource provided as part of the ParseAPI.

```
Address low()
```

The lower bound of the interval of address space covered by this region.

```
Address high()
```

The upper bound of the interval of address space covered by this region.

```
bool contains(Address addr)
```

Returns TRUE if `addr` $\in$ [`low()`, `high()`), FALSE otherwise.

```
virtual bool wasUserAdded() const
```

Return TRUE if this region was added by the user, FALSE otherwise.

4.3 Class Function

Defined in: `CFG.h`

The Function class represents the portion of the program CFG that is reachable through intraprocedural control flow transfers from the function's entry block. Functions in the ParseAPI have only a single entry point; multiple-entry functions such as those found in Fortran programs are represented as several functions that "share" a subset of the CFG. Functions may be non-contiguous and may share blocks with other functions.

FuncSource	Meaning
RT	recursive traversal (default)
HINT	specified in CodeSource hints
GAP	speculative parsing heuristics
GAPRT	recursive traversal from speculative parse
ONDEMAND	dynamically discovered at runtime
MODIFICATION	Added via user modification

Return status of an function, which indicates whether this function will return to its caller or not; see description below.

FuncReturnStatus	Meaning
UNSET	unparsed function (default)
NORETURN	will not return
UNKNOWN	cannot be determined statically
RETURN	may return

```
typedef boost::transform_iterator<selector, blockmap::iterator> bmap_iterator
typedef boost::transform_iterator<selector, blockmap::const_iterator> bmap_const_iterator
typedef boost::iterator_range<bmap_iterator> blocklist
typedef boost::iterator_range<bmap_const_iterator> const_blocklist
typedef std::set<Edge*> edgelist
```

Containers for block and edge access. Library users *must not* rely on the underlying container type of std::set/std::vector lists, as it is subject to change.

Method name	Return type	Method description
name	string	Name of the function.
addr	Address	Entry address of the function.
entry	Block *	Entry block of the function.
parsed	bool	Whether the function has been parsed.
blocks	blocklist &	List of blocks contained by this function sorted by entry address.
callEdges	const edgelist &	List of outgoing call edges from this function.
returnBlocks	const_blocklist &	List of all blocks ending in return edges.
exitBlocks	const_blocklist &	List of all blocks that end the function, including blocks with no out-edges.
hasNoStackFrame	bool	True if the function does not create a stack frame.
savesFramePointer	bool	True if the function saves a frame pointer (e.g. %ebp).
cleansOwnStack	bool	True if the function tears down stack-passed arguments upon return.
region	CodeRegion *	Code region that contains the function.
isrc	InstructionSource *	The InstructionSource for this function.
obj	CodeObject *	CodeObject that contains this function.
src	FuncSrc	The type of hint that identified this function's entry point.
restatus	FuncReturnStatus *	Returns the best-effort determination of whether this function may return or not. Return status cannot always be statically determined, and at most can guarantee that a function *may* return, not that it *will* return.
getReturnType	Type *	Type representing the return type of the function.

```
Function(Address addr,
         string name,
         CodeObject * obj,
         CodeRegion * region,
         InstructionSource * isource)
```

Creates a function at `addr` in the code region specified. Insructions for this function are given in `isource`.

```
LoopTreeNode* getLoopTree()
```

Return the nesting tree of the loops in the function. See class `LoopTreeNode` for more details

```
Loop* findLoop(const char *name)
```

Return the loop with the given nesting name. See class `LoopTreeNode` for more details about how loop nesting names are assigned.

```
bool getLoops(vector<Loop*> &loops);
```

Fill `loops` with all the loops in the function

```
bool getOuterLoops(vector<Loop*> &loops);
```

Fill `loops` with all the outermost loops in the function

```
bool dominates(Block* A, Block *B);
```

Return true if block `A` dominates block `B`

```
Block* getImmediateDominator(Block *A);
```

Return the immediate dominator of block `AïijŇNULL` if the block `A` does not have an immediate dominator.

```
void getImmediateDominates(Block *A, set<Block*> &imm);
```

Fill `imm` with all the blocks immediate dominated by block `A`

```
void getAllDominates(Block *A, set<Block*> &dom);
```

Fill `dom` with all the blocks dominated by block `A`

```
bool postDominates(Block* A, Block *B);
```

Return true if block `A` post-dominates block `B`

```
Block* getImmediatePostDominator(Block *A);
```

Return the immediate post-dominator of block `AïijŇNULL` if the block `A` does not have an immediate post-dominator.

```
void getImmediatePostDominates(Block *A, set<Block*> &imm);
```

Fill `imm` with all the blocks immediate post-dominated by block `A`

```
void getAllPostDominates(Block *A, set<Block*> &dom);
```

Fill `dom` with all the blocks post-dominated by block `A`

`std::vector<FuncExtent *> const& extents()`

Returns a list of contiguous extents of binary code within the function.

`void setEntryBlock(block * new_entry)`

Set the entry block for this function to `new_entry`.

`void set_retstatus(FuncReturnStatus rs)`

Set the return status for the function to `rs`.

`bool contains(Block *b)`

Return true if this function contains the given block `b`; otherwise false.

`void removeBlock(Block *)`

Remove a basic block from the function.

4.4 Class Block

Defined in: `CFG.h`

A Block represents a basic block as defined in Section 2, and is the lowest level representation of code in the CFG.

`typedef std::vector<Edge *> edgelist`

Container for edge access. Refer to Section 4.12 for details. Library users *must not* rely on the underlying container type of std::vector, as it is subject to change.

Method name	Return type	Method description
start	Address	Address of the first instruction in the block.
end	Address	Address immediately following the last instruction in the block.
last	Address	Address of the last instruction in the block.
lastInsnAddr	Address	Alias of `last`.
size	Address	Size of the block; `end - start`.
parsed	bool	Whether the block has been parsed.
obj	CodeObject *	CodeObject containing this block.
region	CodeRegion *	CodeRegion containing this block.
sources	const edgelist &	List of all in-edges to the block.
targets	const edgelist &	List of all out-edges from the block.
containingFuncs	int	Number of Functions that contain this block.

```
bool consistent(Address addr,
                Address & prev_insn)
```

Check whether address `addr` is *consistent* with this basic block. An address is consistent if it is the boundary between two instructions in the block. As long as `addr` is within the range of the block, `prev_insn` will contain the address of the previous instruction boundary before `addr`, regardless of whether `addr` is consistent or not.

```
void getFuncs(std::vector<Function *> & funcs)
```

Fills in the provided vector with all functions that share this basic block.

```
template <class OutputIterator>
void getFuncs(OutputIterator result)
```

Generic version of the above; adds each Function that contains this block to the provided OutputIterator. For example:

```
std::set<Function *> funcs;
block->getFuncs(std::inserter(funcs, funcs.begin()));
```

```
typedef std::map<Offset, InstructionAPI::Instruction::Ptr> Insns
void getInsns(Insns &insns) const
```

Disassembles the block and stores the result in `Insns`.

```
InstructionAPI::Instruction::Ptr getInsn(Offset o) const
```

Returns the instruction starting at offset `o` within the block. Returns `InstructionAPI::Instruction::Ptr()` if `o` is outside the block, or if an instruction does not begin at `o`.

4.5 Class Edge

Defined in: `CFG.h`

Typed Edges join two blocks in the CFG, indicating the type of control flow transfer instruction that joins the blocks to each other. Edges may not correspond to a control flow transfer instruction at all, as in the case of the FALLTHROUGH edge that indicates where straight-line control flow is split by incoming transfers from another location, such as a branch. While not all blocks end in a control transfer instruction, all control transfer instructions end basic blocks and have outgoing edges; in the case of unresolvable control flow, the edge will target a special "sink" block (see `sinkEdge()`, below).

EdgeTypeEnum	Meaning
CALL	call edge
COND_TAKEN	conditional branch–taken
COND_NOT_TAKEN	conditional branch–not taken
INDIRECT	branch indirect
DIRECT	branch direct
FALLTHROUGH	direct fallthrough (no branch)
CATCH	exception handler
CALL_FT	post-call fallthrough
RET	return

Method name	Return type	Method description
src	Block *	Source of the edge.
trg	Block *	Target of the edge.
type	EdgeTypeEnum	Type of the edge.
sinkEdge	bool	True if the target is the sink block.
interproc	bool	True if the edge should be interpreted as interprocedural (e.g. calls, returns, unconditional or conditional tail calls).

4.6 Class Loop

Defined in: `CFG.h`

The Loop class represents code that may execute repeatedly. We detect both natural loops (loops that have a single entry block) and irreducible loops (loops that have multiple entry blocks). A back edge is defined as an edge that has its source in the loop and has its target being an entry block of the loop. It represents the end of an iteration of the loop. For all the loops detected in a function, we also build a loop nesting tree to represent the nesting relations between the loops. See class `LoopTreeNode` for more details.

`Loop* parent`

Returns the loop which directly encloses this loop. NULL if no such loop.

`bool containsAddress(Address addr)`

Returns true if the given address is within the range of this loop's basic blocks.

`bool containsAddressInclusive(Address addr)`

Returns true if the given address is within the range of this loop's basic blocks or its children.

`int getLoopEntries(vector<Block*>& entries);`

Fills `entries` with the set of entry basic blocks of the loop. Return the number of the entries that this loop has

`int getBackEdges(vector<Edge*> &edges)`

Sets `edges` to the set of back edges in this loop. It returns the number of back edges that are in this loop.

`bool getContainedLoops(vector<Loop*> &loops)`

Returns a vector of loops that are nested under this loop.

`bool getOuterLoops(vector<Loop*> &loops)`

Returns a vector of loops that are directly nested under this loop.

`bool getLoopBasicBlocks(vector<Block*> &blocks)`

Fills `blocks` with all basic blocks in the loop

`bool getLoopBasicBlocksExclusive(vector<Block*> &blocks)`

Fills `blocks` with all basic blocks in this loop, excluding the blocks of its sub loops.

`bool hasBlock(Block *b);`

Returns `true` if this loop contains basic block `b`.

```
bool hasBlockExclusive(Block *b);
```

Returns `true` if this loop contains basic block `b` and `b` is not in its sub loops.

```
bool hasAncestor(Loop *loop)
```

Returns true if this loop is a descendant of the given loop.

```
Function * getFunction();
```

Returns the function that this loop is in.

4.7 Class LoopTreeNode

Defined in: `CFG.h` The LoopTreeNode class provides a tree interface to a collection of instances of class Loop contained in a function. The structure of the tree follows the nesting relationship of the loops in a function. Each LoopTreeNode contains a pointer to a loop (represented by Loop), and a set of sub-loops (represented by other LoopTreeNode objects). The `loop` field at the root node is always `NULL` since a function may contain multiple outer loops. The `loop` field is never `NULL` at any other node since it always corresponds to a real loop. Therefore, the outer most loops in the function are contained in the vector of `children` of the root.

Each instance of LoopTreeNode is given a name that indicates its position in the hierarchy of loops. The name of each outermost loop takes the form of `loop_x`, where `x` is an integer from 1 to n, where n is the number of outer loops in the function. Each sub-loop has the name of its parent, followed by a `.y`, where `y` is 1 to m, where m is the number of sub-loops under the outer loop. For example, consider the following C function:

```
void foo() {
  int x, y, z, i;
  for (x=0; x<10; x++) {
    for (y = 0; y<10; y++)
      ...
    for (z = 0; z<10; z++)
      ...
  }
  for (i = 0; i<10; i++) {
     ...
  }
}
```

The `foo` function will have a root LoopTreeNode, containing a NULL loop entry and two LoopTreeNode children representing the functions outermost loops. These children would have names `loop_1` and `loop_2`, respectively representing the `x` and `i` loops. `loop_2` has no children. `loop_1` has two child LoopTreeNode objects, named `loop_1.1` and `loop_1.2`, respectively representing the `y` and `z` loops.

```
Loop *loop;
```

The Loop instance it points to.

```
std::vector<LoopTreeNode *> children;
```

The LoopTreeNode instances nested within this loop.

```
const char * name();
```

Returns the hierarchical name of this loop.

```
const char * getCalleeName(unsigned int i)
```

Returns the function name of the ith callee.

```
unsigned int numCallees()
```

Returns the number of callees contained in this loop's body.

```
bool getCallees(vector<Function *> &v);
```

Fills `v` with a vector of the functions called inside this loop.

```
Loop * findLoop(const char *name);
```

Looks up a loop by the hierarchical name

4.8 Class CodeSource

Defined in: `CodeSource.h`

The CodeSource interface is used by the ParseAPI to retrieve binary code from an executable, library, or other binary code object; it also can provide hints of function entry points (such as those derived from debugging symbols) to seed the parser. The ParseAPI provides a default implementation based on the SymtabAPI that supports many common binary formats. For details on implementing a custom CodeSource, see Appendix A.

```
virtual bool nonReturning(Address func_entry)
virtual bool nonReturning(std::string func_name)
```

Looks up whether a function returns (by name or location). This information may be statically known for some code sources, and can lead to better parsing accuracy.

```
virtual bool nonReturningSyscall(int /*number*/)
```

Looks up whether a system call returns (by system call number). This information may be statically known for some code sources, and can lead to better parsing accuracy.

```
virtual Address baseAddress()
virtual Address loadAddress()
```

If the binary file type supplies non-zero base or load addresses (e.g. Windows PE), implementations should override these functions.

```
std::map< Address, std::string > & linkage()
```

Returns a reference to the external linkage map, which may or may not be filled in for a particular CodeSource implementation.

```
struct Hint {
    Address _addr;
    CodeRegion *_region;
    std::string _name;
    Hint(Addr, CodeRegion *, std::string);
}
std::vector< Hint > const& hints()
```

Returns a vector of the currently defined function entry hints.

```
std::vector<CodeRegion *> const& regions()
```

Returns a read-only vector of code regions within the binary represented by this code source.

```
int findRegions(Address addr,
                set<CodeRegion *> & ret)
```

Finds all CodeRegion objects that overlap the provided address. Some code sources (e.g. archive files) may have several regions with overlapping address ranges; others (e.g. ELF binaries) do not.

```
bool regionsOverlap()
```

Indicates whether the CodeSource contains overlapping regions.

4.9 Class ParseCallback

Defined in: `ParseCallback.h`

The ParseCallback class allows ParseAPI users to be notified of various events during parsing. For most users this notification is unnecessary, and an instantiation of the default ParseCallback can be passed to the CodeObject during initialization. Users who wish to be notified must implement a class that inherits from ParseCallback, and implement one or more of the methods described below to receive notification of those events.

```
struct default_details {
    default_details(unsigned char * b,size_t s, bool ib);
    unsigned char * ibuf;
    size_t isize;
    bool isbranch;
}
```

Details used in the `unresolved_cf` and `abruptEnd_cf` callbacks.

```
virtual void instruction_cb(Function *,
                            Block *,
                            Address,
                            insn_details *)
```

Invoked for each instruction decoded during parsing. Implementing this callback may incur significant overhead.

```
struct insn_details {
    InsnAdapter::InstructionAdapter * insn;
}
```

```
void interproc_cf(Function *,
                  Address,
                  interproc_details *)
```

Invoked for each interprocedural control flow instruction.

```
struct interproc_details {
    typedef enum {
        ret,
        call,
        branch_interproc, // tail calls, branches to plts
        syscall
    } type_t;
    unsigned char * ibuf;
```

```
    size_t isize;
    type_t type;
    union {
        struct {
            Address target;
            bool absolute_address;
            bool dynamic_call;
        } call;
    } data;
}
```

Details used in the `interproc_cf` callback.

```
void overlapping_blocks(Block *,
                        Block *)
```

Noification of inconsistent parse data (overlapping blocks).

4.10 Class FuncExtent

Defined in: `CFG.h`

Function Extents are used internally for accounting and lookup purposes. They may be useful for users who wish to precisely identify the ranges of the address space spanned by functions (functions are often discontiguous, particularly on architectures with variable length instruction sets).

Method name	Return type	Method description
func	Function *	Function linked to this extent.
start	Address	Start of the extent.
end	Address	End of the extent (exclusive).

4.11 Edge Predicates

Defined in: `CFG.h`

Edge predicates control iteration over edges. For example, the provided `Intraproc` edge predicate can be used with filter iterators and standard algorithms, ensuring that only intraprocedural edges are visited during iteration. Two other examples of edge predicates are provided: `SingleContext` only visits edges that stay in a single function context, and `NoSinkPredicate` does not visit edges to the *sink* block. The following code traverses all of the basic blocks within a function:

```
    #include <boost/filter_iterator.hpp>
2   using boost::make_filter_iterator;
    struct target_block
    {
      Block* operator()(Edge* e) { return e->trg(); }
```

```
    };


    vector<Block*> work;
    Intraproc epred; // ignore calls, returns

    work.push_back(func->entry()); // assuming 'func' is a Function*

    // do_stuff is a functor taking a Block* as its argument
    while(!work.empty()) {
        Block * b = work.back();
        work.pop_back();

        Block::edgelist & targets = block->targets();
        // Do stuff for each out edge
        std::for_each(make_filter_iterator(targets.begin(), epred),
                      make_filter_iterator(targets.end(), epred),
                      do_stuff());
        std::transform(make_filter_iterator(targets.begin(), epred),
                       make_filter_iterator(targets.end(), epred),
                       std::back_inserter(work),
                       std::mem_fun(Edge::trg));
        Block::edgelist::const_iterator found_interproc =
                std::find_if(targets.begin(), targets.end(), Interproc());
        if(interproc != targets.end()) {
                // do something with the interprocedural edge you found
        }
    }
```

Anything that can be treated as a function from `Edge*` to a `bool` can be used in this manner. This replaces the beta interface where all `EdgePredicate`s needed to descend from a common parent class. Code that previously constructed iterators from an edge predicate should be replaced with equivalent code using filter iterators as follows:

```
// OLD
for(Block::edgelist::iterator i = targets.begin(epred);
    i != targets.end(epred);
    i++)
{
  // ...
}
// NEW
for_each(make_filter_iterator(epred, targets.begin(), targets.end()),
         make_filter_iterator(epred, targets.end(), targets,end()),
         loop_body_as_function);
// NEW (C++11)
for(auto i = make_filter_iterator(epred, targets.begin(), targets.end());
    i != make_filter_iterator(epred, targets.end(), targets.end());
    i++)
{
  // ...
```

```
}
```

4.12 Containers

Several of the ParseAPI data structures export containers of CFG objects; the CodeObject provides a list of functions in the binary, for example, while functions provide lists of blocks and so on. To avoid tying the internal storage for these structures to any particular container type, ParseAPI objects export a ContainerWrapper that provides an iterator interface to the internal containers. These wrappers and predicate interfaces are designed to add minimal overhead while protecting ParseAPI users from exposure to internal container storage details. Users *must not* rely on properties of the underlying container type (e.g. storage order) unless that property is explicity stated in this manual.

ContainerWrapper containers export the following interface (`iterator` types vary depending on the template parameters of the ContainerWrapper, but are always instantiations of the PredicateIterator described below):

```
iterator begin()
iterator begin(predicate *)
```

Return an iterator pointing to the beginning of the container, with or without a filtering predicate implementation (see Section 4.11 for details on filter predicates).

```
iterator const& end()
```

Return the iterator pointing to the end of the container (past the last element).

```
size_t size()
```

Returns the number of elements in the container. Execution cost may vary depending on the underlying container type.

```
bool empty()
```

Indicates whether the container is empty or not.

The elements in ParseAPI containers can be accessed by iteration using an instantiation of the PredicateIterator. These iterators can optionally act as filters, evaluating a boolean predicate for each element and only returning those elements for which the predicate returns TRUE. *Iterators with non-*NULL *predicates may return fewer elements during iteration than their* `size()` *method indicates.* Currently PredicateIterators only support forward iteration. The operators `++` (prefix and postfix), `==`, `!=`, and `*` (dereference) are supported.

A Extending ParseAPI

The ParseAPI is design to be a low level toolkit for binary analysis tools. Users can extend the ParseAPI in two ways: by extending the control flow structures (Functions, Blocks, and Edges) to incorporate additional data to support various analysis applications, and by adding additional binary code sources that are unsupported by the default SymtabAPI-based code source. For example, a code source that represents a program image in memory could be implemented by fulfilling the CodeSource and InstructionSource interfaces described in Section 4.8 and below. Implementations that extend the CFG structures need only provide a custom allocation factory in order for these objects to be allocated during parsing.

A.1 Instruction and Code Sources

A CodeSource, as described above, exports its own and the InstructionSource interface for access to binary code and other details. In addition to implementing the virtual methods in the CodeSource base class (Section 4.8), the methods in the pure-virtual InstructionSource class must be implemented:

`virtual bool isValidAddress(const Address)`

Returns TRUE if the address is a valid code location.

`virtual void* getPtrToInstruction(const Address)`

Returns pointer to raw memory in the binary at the provided address.

`virtual void* getPtrToData(const Address)`

Returns pointer to raw memory in the binary at the provided address. The address need not correspond to an executable code region.

`virtual unsigned int getAddressWidth()`

Returns the address width (e.g. four or eight bytes) for the represented binary.

`virtual bool isCode(const Address)`

Indicates whether the location is in a code region.

`virtual bool isData(const Address)`

Indicates whether the location is in a data region.

`virtual Address offset()`

The start of the region covered by this instruction source.

`virtual Address length()`

The size of the region.

`virtual Architecture getArch()`

The architecture of the instruction source. See the Dyninst manual for details on architecture differences.

`virtual bool isAligned(const Address)`

For fixed-width instruction architectures, must return TRUE if the address is a valid instruction boundary and FALSE otherwise; otherwise returns TRUE. This method has a default implementation that should be sufficient.

CodeSource implementors need to fill in several data structures in the base CodeSource class:

`std::map<Address, std::string> _linkage`

Entries in the linkage map represent external linkage, e.g. the PLT in ELF binaries. Filling in this map is optional.

`Address _table_of_contents`

Many binary format have "table of contents" structures for position independant references. If such a structure exists, its address should be filled in.

```
std::vector<CodeRegion *> _regions
Dyninst::IBSTree<CodeRegion> _region_tree
```

One or more contiguous regions of code or data in the binary object must be registered with the base class. Keeping these structures in sync is the responsibility of the implementing class.

`std::vector<Hint> _hints`

CodeSource implementors can supply a set of Hint objects describing where functions are known to start in the binary. These hints are used to seed the parsing algorithm. Refer to the CodeSource header file for implementation details.

A.2 CFG Object Factories

Users who which to incorporate the ParseAPI into large projects may need to store additional information about CFG objects like Functions, Blocks, and Edges. The simplest way to associate the ParseAPI-level CFG representation with higher-level implementation is to extend the CFG classes provided as part of the ParseAPI. Because the parser itself does not know how to construct such extended types, implementors must provide an implementation of the CFGFactory that is specialized for their CFG classes. The CFGFactory exports the following simple interface:

```
virtual Function * mkfunc(Address addr,
                          FuncSource src,
                          std::string name,
                          CodeObject * obj,
                          CodeRegion * region,
                          Dyninst::InstructionSource * isrc)
```

Returns an object derived from Function as though the provided parameters had been passed to the Function constructor. The ParseAPI parser will never invoke `mkfunc()` twice with identical `addr`, and `region` parameters—that is, Functions are guaranteed to be unique by address within a region.

```
virtual Block * mkblock(Function * func,
                        CodeRegion * region,
                        Address addr)
```

Returns an object derived from Block as though the provided parameters had been passed to the Block constructor. The parser will never invoke `mkblock()` with identical `addr` and `region` parameters.

```
virtual Edge * mkedge(Block * src,
                      Block * trg,
                      EdgeTypeEnum type)
```

Returns an object derived from Edge as though the provided parameters had been passed to the Edge constructor. The parser *may* invoke `mkedge()` multiple times with identical parameters.

```
virtual Block * mksink(CodeObject *obj,
                       CodeRegion *r)
```

Returns a "sink" block derived from Block to which all unresolvable control flow instructions will be linked. Implementors may return a unique sink block per CodeObject or a single global sink.

Implementors of extended CFG classes are required to override the default implementations of the *mk** functions to allocate and return the appropriate derived types statically cast to the base type. Implementors must also add all allocated objects to the following internal lists:

```
fact_list<Edge> edges_
fact_list<Block> blocks_
fact_list<Function> funcs_
```

O(1) allocation lists for CFG types. See the CFG.h header file for list insertion and removal operations.

Implementors *may* but are *not required to* override the deallocation following deallocation routines. The primary reason to override these routines is if additional action or cleanup is necessary upon CFG object release; the default routines simply remove the objects from the allocation list and invoke their destructors.

```
virtual void free_func(Function * f)
virtual void free_block(Block * b)
virtual void free_edge(Edge * e)
virtual void free_all()
```

CFG objects should be freed using these functions, rather than delete, to avoid leaking memory.

B Defensive Mode Parsing

Binary code that defends itself against analysis may violate the assumptions made by the the ParseAPI's standard parsing algorithm. Enabling defensive mode parsing activates more conservative assumptions that substantially reduce the percentage of code that is analyzed by the ParseAPI. For this reason, defensive mode parsing is best-suited for use of ParseAPI in conjunction with dynamic analysis techniques that can compensate for its limited coverage of the binary code.

Paradyn Parallel Performance Tools

PatchAPI Programmer's Guide

9.2 Release
June 2016

Computer Sciences Department
University of Wisconsin–Madison
Madison, WI 53706

Computer Science Department
University of Maryland
College Park, MD 20742

Email dyninst-api@cs.wisc.edu
Web www.dyninst.org

Contents

1 Introduction

This manual describes PatchAPI, a programming interface and library for binary code patching. A programmer uses PatchAPI to instrument (insert code into) and modify a binary executable or library by manipulating the binary's control flow graph (CFG). We allow the user to instrument a binary by annotating a CFG with *snippets*, or sequences of inserted code, and to modify the binary by directly manipulating the CFG. The PatchAPI interface, and thus tools written with PatchAPI, is designed to be flexible and extensible. First, users may *inherit* from PatchAPI abstractions in order to store their own data. Second, users may create *plugins* to extend PatchAPI to handle new types of instrumentation, different binary types, or different patching techniques.

PatchAPI represents the binary as an annotatable and modifiable CFG. The CFG consists of abstractions for binary objects, functions, basic blocks, edges connecting basic blocks, and loops representing code that may execute repeatedly, which are similar to the CFG abstractions used by the ParseAPI component.

Users instrument the binary by annotating this CFG using three additional high-level abstractions: Point, Snippet, and Instance. A Point supports instrumentation by representing a particular aspect of program behavior (e.g., entering a function or traversing an edge) and containing instances of Snippets. Point lookup is performed with a single PatchAPI manager (PatchMgr) object by Scope (e.g., a CFG object) and Type (e.g., function entry). In addition, a user may provide an optional Filter that selects a subset of matching Points. A Snippet represents a sequence of code to be inserted at certain points. To maximize flexibility, PatchAPI does not prescribe a particular snippet form; instead, users may provide their own (e.g., a binary buffer, a Dyninst abstract syntax tree (AST), or code written in the DynC language). Users instrument the binary by adding Snippets to the desired Points. An Instance represents the insertion of a particular Snippet at a particular Point.

The core PatchAPI representation of an annotatable and modifiable CFG operates in several domains, including on a running process (dynamic instrumentation) or a file on disk (binary rewriting). Furthermore, PatchAPI may be used both in the same address space as the process (1st-party instrumentation) or in a different address space via the debug interface (3rd-party instrumentation). Similarly, developers may define their own types of Snippets to encapsulate their own code generation techniques. These capabilities are provided by a plugin interface; by implementing a plugin a developer may extend PatchAPI's capabilities.

This manual is structured as follows. Section 2 presents the core abstractions in the public and plugin interface of PatchAPI. Section 3 shows several examples in C++ to illustrate the usage of PatchAPI. Detailed API reference can be found in Section 4 and Section 6. Finally, Appendix A provides a quick tutorial of PatchAPI to those who are already familiar with DyninstAPI.

2 Abstractions

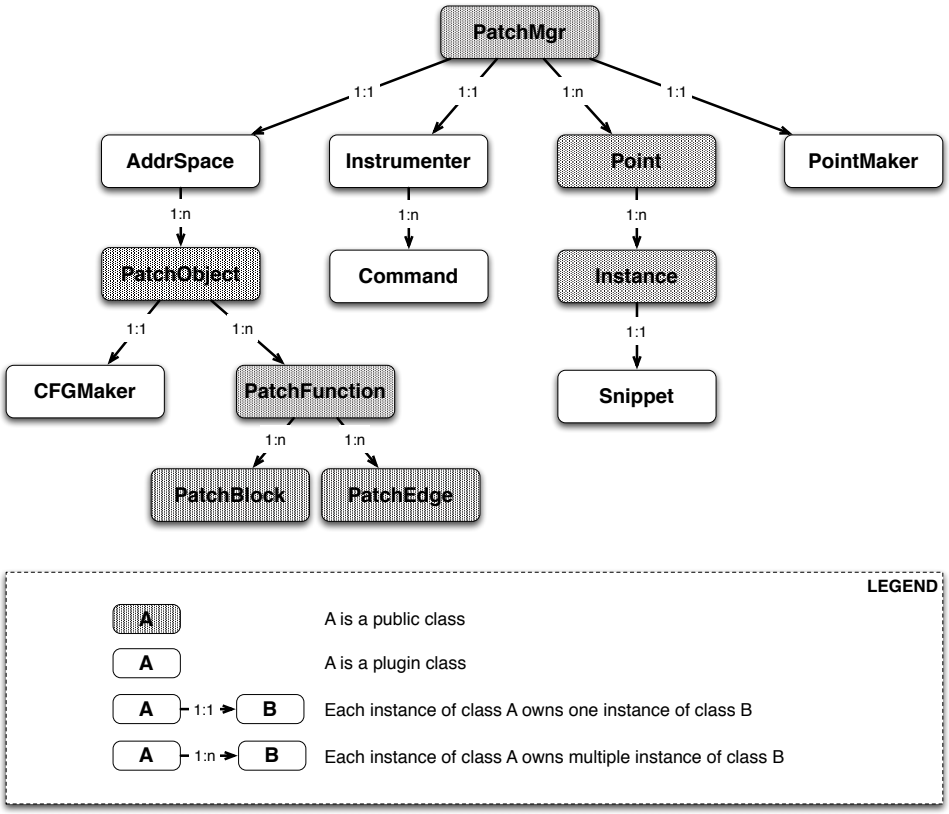


Figure 1: Object Ownership

PatchAPI contains two interfaces: the public interface and the plugin interface. The public interface is used to find instrumentation points, insert or delete code snippets, and register plugins provided by programmers. The plugin interface is used to customize different aspects in the binary code patching. PatchAPI provides a set of default plugins for first party code patching, which is easy to extend to meet different requirements in practice.

Figure 1 shows the ownership hierarchy for PatchAPI's classes. Ownership is a "contains" relationship. If one class owns another, then instances of the owner class maintain exactly one or possibly more than one instances of the other, which depends on whether the relationship is a "1:1" or a "1:n" relationship. In Figure 1, for example, each PatchMgr instance contains exactly one instance of a AddrSpace object, while a PatchMgr instance may contains more than one instances of a Point object.

The remainder of this section briefly describes the classes that make up PatchAPI's two interfaces. For more details, see the class descriptions in Section 4 and Section 6.

2.1 Public Interface

PatchMgr, Point, and Snippet are used to perform the main process of binary code patching: 1) find some **Point**; 2) insert or delete **Snippet** at some **Point**.

- *PatchMgr* - The PatchMgr class is the top-level class for finding instrumentation **Points**, inserting or deleting **Snippets**, and registering user-provided plugins.
- *Point* - The Point class represents a location on the CFG that acts as a container of inserted snippet **Instances**. Points of different types are distinct even the underlying code relocation and generation engine happens to put instrumentation from them at the same place.
- *Instance* - The Instance class is a representation of a particular snippet inserted at a particular point.
- *PatchObject* - The PatchObject class is a wrapper of ParseAPI's CodeObject class, which represents an individual binary code object, such as an executable or a library.
- *PatchFunction* - The PatchFunction class is a wrapper of ParseAPI's Function class, which represents a function.
- *PatchBlock* - The PatchBlock class is a wrapper of ParseAPI's Block class, which represents a basic block.
- *PatchEdge* - The PatchEdge class is a wrapper of ParseAPI's Edge class, which join two basic blocks in the CFG, indicating the type of control flow transfer instruction that joins the basic blocks to each other.
- *PatchLoop* - The PatchLoop class is a wrapper of ParseAPI's Loop class, which repreents a piece of code that may execute repeatedly.
- *PatchLoopTreeNode* - The PatchLoopTreeNode class is a wrapper of ParseAPI's LoopTreeNode class, which provides a tree interface to a collection of instances of class PatchLoop contained in a function. The structure of the tree follows the nesting relationship of the loops in a function.

2.2 Plugin Interface

The address space abstraction determines whether the code patching is 1st party, 3rd party or binary rewriting.

- *AddrSpace* - The AddrSpace class represents the address space of a **Mutatee** (a program that is instrumented), where it contains a collection of **PatchObjects** that represent shared libraries or a binary executable. In addition, programmers implement some memory management interfaces in the AddrSpace class to determines the type of the code patching - 1st party, 3rd party, or binary rewriting.

Programmers can decide the representation of a **Snippet**, for example, the representation can be in high level language (e.g., C or C++), or can simply be in binary code (e.g., 0s and 1s).

- *Snippet* - The Snippet class allows programmers to easily plug in their own snippet representation and the corresponding mini-compiler to translate the representation into the binary code.

PatchAPI provides a thin layer on top of ParseAPI's Control Flow Graph (CFG) layer, which associates some useful information for the ease of binary code patching, for example, a shared library's load address. This layer of CFG structures include PatchObject, PatchFunction, PatchBlock and PatchEdge classes. Programmers can extend these four CFG classes, and use the derived class of CFGMaker to build a CFG with the augmented CFG structures.

- *CFGMaker* - The CFGMaker class is a factory class that constructs the above CFG structures. This class is used in CFG parsing.

Similar to customizing the PatchAPI layer, programmers can also customize the Point class by extending it.

- *PointMaker* - The PointMaker class is a factory class that constructs a subclass of the Point class.

The implementation of an instrumentation engine may be very sophisticated (e.g., relocating a function), or very simple (e.g., simply overwrite an instruction). Therefore, PatchAPI provides a flexible framework for programmers to customize the instrumentation engine.

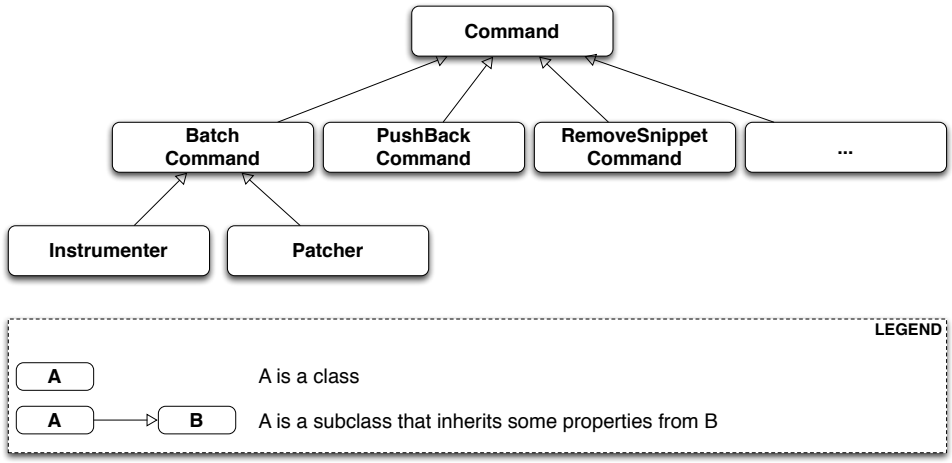


Figure 2: Inheritance Hierarchy

This framework is based on Command Pattern [1]. The instrumentation engine has transactional semantics, where all instrumentation requests should succeed or all should fail. In our framework, the **Command** abstraction represents an instrumentation request or a logical step in the code patching process. We accumulate a list of **Commands**, and execute them one by one. If one **Command** fails, we undo all preceding finished **Commands**. Figure 2 illustrates the inheritance hierarchy for related classes. There is a default implementation of instrumentation engine in PatchAPI for 1st party code patching.

- *Command* - The Command class represents an instrumentation request (e.g., snippet insertion or removal), or a logical step in the code patching (e.g., install instrumentation). This class provides a run() method and an undo() method, where run() will be called for normal execution, and undo() will be called for undoing this Command.
- *BatchCommand* - The BatchCommand class is a subclass of Command, and it is in fact a container of a list of Commands to be executed atomically.
- *Instrumenter* - The Instrumenter class inherits BatchCommand to encapsulate the core code patching logic, which includes binary code generation. Instrumenter would contain several logical steps that are individual Commands.
- *Patcher* - The Patcher class is also a subclass of BatchCommand. It accepts instrumentation requests from users, where these instrumentation requests are Commands (e.g., snippet insertion). Furthermore, Patcher implicitly adds Instrumenter to the end of the Command list to generate binary code and install the instrumentation.

[1] http://en.wikipedia.org/wiki/Command_pattern

3 Examples

To illustrate the ideas of PatchAPI, we present some simple code examples that demonstrate how the API can be used.

3.1 Using the public interface

The basic flow of doing code patching is to first find some points in a program, and then to insert, delete or update a piece of code at these points.

3.1.1 CFG Traversal

Listing 1: Example of CFG traversal

```
1  ParseAPI::CodeObject* co = ...
2  PatchObject* obj = PatchObject::create(co, code_base);
3
4  // Find all functions in the object
5  std::vector<PatchFunction*> all;
6  obj->funcs(back_inserter(all));
7
8  for (std::vector<PatchFunction*>::iterator fi = all.begin();
9       fi != all.end(); fi++) {
10   // Print out each function's name
11   PatchFunction* func = *fi;
12   std::cout << func->name() << std::endl;
13
14   const PatchFunction::Blockset& blks = func->blocks();
15   for (PatchFunction::BlockSet::iterator bi = blks.begin();
16        bi != blks.end(); bi++) {
17     // Print out each block's size
18     PatchBlock* blk = *bi;
19     std::cout << "\tBlock size:" << blk->size() << std::endl;
20   }
21 }
```

In the above code, we illustrate how to traverse CFG structures in PatchAPI. First, we construct an instance of PatchObject using an instance of ParseAPI's CodeObject. Then, we traverse all functions in that object, and print out each function's name. For each function, we also print out the size of each basic block.

3.1.2 Point Finding

Listing 2: Example of point finding

```
1  PatchFunction *func = ...;
2  PatchBlock *block = ...;
3  PatchEdge *edge = ...;
4
5  PatchMgr *mgr = ...;
6
7  std::vector<Point*> pts;
8  mgr->findPoints(Scope(func),
9                  Point::FuncEntry |
10                 Point::PreCall |
11                 Point::FuncExit,
12                 back_inserter(pts));
13 mgr->findPoints(Scope(block),
14                 Point::BlockEntry,
15                 back_inserter(pts));
16 mgr->findPoints(Scope(edge),
17                 Point::EdgeDuring,
18                 back_inserter(pts));
```

The above code shows how to use the PatchMgr::findPoints method to find some instrumentation points. There are three invocations of findPoints. For the first invocation (Line 8), it finds points only within a specific function *func*, and output the found points to a vector *pts*. The result should include all points at this function's entry, before all function calls inside this function, and at the function's exit. Similarly, for the second invocation (Line 13), it finds points only within a specific basic *block*, and the result should include the point at the block entry. Finally, for the third invocation (Line 16), it finds the point at a specific CFG *edge* that connects two basic blocks.

3.1.3 Code Patching

Listing 3: Example of code patching

```
1 MySnippet::ptr snippet = MySnippet::create(new MySnippet);
2
3 Patcher patcher(mgr);
4 for (vector<Point*>::iterator iter = pts.begin();
5      iter != pts.end(); ++iter) {
6   Point* pt = *iter;
7   patcher.add(PushBackCommand::create(pt, snippet));
8 }
9 patcher.commit();
```

The above code is to insert the same code *snippet* to all points *pts* found in Section 3.1.2. We'll explain the snippet (Line 1) in the example in Section 3.2.2. Each point maintains a

list of snippet instances, and the PushBackCommand is to push a snippet instance to the end of that list. An instance of Patcher is to represent a transaction of code patching. In this example, all snippet insertions (or all PushBackCommands) are performed atomically when the Patcher::commit method is invoked. That is, all snippet insertions would succeed or all would fail.

3.2 Using the plugin interface

3.2.1 Address Space

Listing 4: Example of implementing address space plugin

```
1  class MyAddrSpace : public AddrSpace {
2    public:
3      ...
4      virtual Address malloc(PatchObject* obj, size_t size, Address near) {
5        Address buffer = ...
6        // do memory allocation here
7        return buffer;
8      }
9      virtual bool write(PatchObject* obj, Address to_addr, Address from_addr,
10                        size_t size) {
11       // copy data from the address from_addr to the address to_addr
12       return true;
13     }
14     ...
15 };
```

The above code is to implement the address space plugin, in which, a set of memory management methods should be specified, including malloc, free, realloc, write and so forth. The instrumentation engine will utilize these memory management methods during the code patching process. For example, the instrumentation engine needs to *malloc* a buffer in Mutatee's address space, and then *write* the code snippet into this buffer.

3.2.2 Snippet Representation

Listing 5: Example of implementing snippet plugin

```
1  class MySnippet : public Snippet {
2    public:
3      virtual bool generate(Point *pt, Buffer &buf) {
4        // Generate and store binary code in the Buffer buf
5        return true;
6      }
7  };
8  MySnippet::ptr snippet = MySnippet::create(new MySnippet);
```

The above code illustrates how to customize a user-defined snippet *MySnippet* by implementing the "mini-compiler" in the *generate* method, which will be used later in the instrumentation engine to generate binary code.

3.2.3 Code Parsing

Listing 6: Example of customizing CFG parsing

```
class MyFunction : public PatchFunction {
  ...
};
class MyCFGMaker : public CFGMaker {
  public:
    ...
    virtual PatchFunction* makeFunction(ParseAPI::Function *f, PatchObject* o) {
      return new MyFunction(f, o);
    }
    ...
};
```

Programmers can augment PatchAPI's CFG structures by annotating their own data. In this case, a factory class should be built by inheriting from the CFGMaker class, to create the augmented CFG structures. The factory class will be used for CFG parsing.

3.2.4 Point Making

Listing 7: Example of point making

```
class MyPoint : public Point {
  public:
    MyPoint(Point::Type t, PatchMgrPtr m, PatchFunction *f);
    ...
};

class MyPointMaker: public PointMaker {
  protected:
    virtual Point *mkFuncPoint(Point::Type t, PatchMgrPtr m, PatchFunction *f) {
      return new MyPoint(t, m, f);
    }
};
```

In the above example, the MyPoint class inherits from the Point class, and the MyPointMaker class inherits from the PointMaker class. The mkFuncPoint method in MyPointMaker simply returns a new instance of MyPoint. The mkFuncPoint method will be invoked

by PatchMgr::findPoint(s).

3.2.5 Instrumentation Engine

Listing 8: Example of customizing instrumentation engine

```
1 class MyInstrumenter : public Instrumenter {
2   public:
3     virtual bool run() {
4       // Specify how to install instrumentation
5     }
6 };
```

Programmers can customize the instrumentation engine by extending the Instrumenter class, and implement the installation of instrumentation inside the method *run()*.

3.2.6 Plugin Registration

Listing 9: Example of registering plugins

```
1  MyCFGMakerPtr cm = ...
2  PatchObject* obj = PatchObject::create(..., cm);
3
4  MyAddrSpacePtr as = ...
5  as->loadObject(obj);
6
7  MyInstrumenter inst = ...
8  PatchMgrPtr mgr = PatchMgr::create(as, ..., inst);
9
10 MySnippet::ptr snippet = MySnippet::create(new MySnippet);
```

The above code shows how to register the above four types of plugins. An instance of the factory class for creating CFG structures is registered to an PatchObject (Line 1 and 2), which is in turn loaded into an instance of AddrSpace (Line 4 and 5). The AddrSpace (or its subclass implemented by programmers) instance is passed to PatchMgr::create (Line 7 and 8), together with an instance of Instrumenter (or its subclass). Finally, a snippet of custom snippet representation MySnippet is created (Line 10). Therefore, all plugins are glued together in PatchAPI.

4 Public API Reference

This section describes public interfaces in PatchAPI. The API is organized as a collection of C++ classes. The classes in PatchAPI fall under the C++ namespace Dyninst::PatchAPI. To access them, programmers should refer to them using the "Dyninst::PatchAPI::" prefix, e.g., Dyninst::PatchAPI::Point. Alternatively, programmers can add the C++ *using* keyword above any references to PatchAPI objects, e.g.,*using namespace Dyninst::PatchAPI* or *using Dyninst::PatchAPI::Point*.

Classes in PatchAPI use either the C++ raw pointer or the boost shared pointer (*boost::shared_ptr<T>*) for memory management. A class uses a raw pointer whenever it is returning a handle to the user that is controlled and destroyed by the PatchAPI runtime library. Classes that use a raw pointer include the CFG objects, a Point, and various plugins, e.g., AddrSpace, CFGMaker, PointMaker, and Instrumenter. A class uses a shared_pointer whenever it is handing something to the user that the PatchAPI runtime library is not controlling and destroying. Classes that use a boost shared pointer include a Snippet, PatchMgr, and Instance, where we typedef a class's shared pointer by appending the Ptr to the class name, e.g., PatchMgrPtr for PatchMgr.

4.1 CFG Interface

4.1.1 PatchObject

Declared in: PatchObject.h

The PatchObject class is a wrapper of ParseAPI's CodeObject class (has-a), which represents an individual binary code object, such as an executable or a library.

```
static PatchObject* create(ParseAPI::CodeObject* co, Address base,
                           CFGMaker* cm = NULL, PatchCallback *cb = NULL);
```

> Creates an instance of PatchObject, which has *co* as its on-disk representation (ParseAPI::CodeObject), and *base* as the base address where this object is loaded in the memory. For binary rewriting, base should be 0. The *cm* and *cb* parameters are for registering plugins. If *cm* or *cb* is NULL, then we use the default implementation of CFGMaker or PatchCallback.

```
static PatchObject* clone(PatchObject* par_obj, Address base,
                          CFGMaker* cm = NULL, PatchCallback *cb = NULL);
```

Returns a new object that is copied from the specified object *par_obj* at the loaded address *base* in the memory. For binary rewriting, base should be 0. The *cm* and *cb* parameters are for registering plugins. If *cm* or *cb* is NULL, then we use the default implementation of CFGMaker or PatchCallback.

```
Address codeBase();
```

Returns the base address where this object is loaded in memory.

```
PatchFunction *getFunc(ParseAPI::Function *func, bool create = true);
```

Returns an instance of PatchFunction in this object, based on the *func* parameter. PatchAPI creates a PatchFunction on-demand, so if there is not any PatchFunction created for the ParseAPI function *func*, and the *create* parameter is false, then no any instance of PatchFunction will be created.

It returns NULL in two cases. First, the function *func* is not in this PatchObject. Second, the PatchFunction is not yet created and the *create* is false. Otherwise, it returns a PatchFunction.

```
template <class Iter>
void funcs(Iter iter);
```

Outputs all instances of PatchFunction in this PatchObject to the STL inserter *iter*.

```
PatchBlock *getBlock(ParseAPI::Block* blk, bool create = true);
```

Returns an instance of PatchBlock in this object, based on the *blk* parameter. PatchAPI creates a PatchBlock on-demand, so if there is not any PatchBlock created for the ParseAPI block *blk*, and the *create* parameter is false, then no any instance of PatchBlock will be created.

It returns NULL in two cases. First, the ParseAPI block *blk* is not in this PatchObject. Second, the PatchBlock is not yet created and the *create* is false. Otherwise, it returns a PatchBlock.

```
template <class Iter>
void blocks(Iter iter);
```

Outputs all instances of PatchBlock in this object to the STL inserter *iter*.

```
PatchEdge *getEdge(ParseAPI::Edge* edge, PatchBlock* src, PatchBlock* trg,
                   bool create = true);
```

Returns an instance of PatchEdge in this object, according to the parameters ParseAPI::Edge *edge*, source PatchBlock *src*, and target PatchBlock *trg*. PatchAPI creates a PatchEdge on-demand, so if there is not any PatchEdge created for the ParseAPI *edge*, and the *create* parameter is false, then no any instance of PatchEdge will be created.

It returns NULL in two cases. First, the ParseAPI *edge* is not in this PatchObject. Second, the PatchEdge is not yet created and the *create* is false. Otherwise, it returns a PatchEdge.

```
template <class Iter>
void edges(Iter iter);
```

Outputs all instances of PatchEdge in this object to the STL inserter *iter*.

```
PatchCallback *cb() const;
```

Returns the PatchCallback object associated with this PatchObject.

4.1.2 PatchFunction

Declared in: PatchCFG.h

The PatchFunction class is a wrapper of ParseAPI's Function class (has-a), which represents a function.

```
const string &name();
```

Returns the function's mangled name.

```
Address addr() const;
```

Returns the address of the first instruction in this function.

```
ParseAPI::Function *function();
```

Returns the ParseAPI::Function associated with this PatchFunction.

```
PatchObject* obj();
```

Returns the PatchObject associated with this PatchFunction.

```
typedef std::set<PatchBlock *> PatchFunction::Blockset;
const Blockset &blocks();
```

Returns a set of all PatchBlocks in this PatchFunction.

```
PatchBlock *entry();
```

Returns the entry block of this PatchFunction.

```
const Blockset &exitBlocks();
```

Returns a set of exit blocks of this PatchFunction.

```
const Blockset &callBlocks();
```

Returns a set of all call blocks of this PatchFunction.

```
PatchCallback *cb() const;
```

Returns the PatchCallback object associated with this PatchFunction.

`PatchLoopTreeNode* getLoopTree()`

Return the nesting tree of the loops in the function. See class `PatchLoopTreeNode` for more details

`PatchLoop* findLoop(const char *name)`

Return the loop with the given nesting name. See class `PatchLoopTreeNode` for more details about how loop nesting names are assigned.

`bool getLoops(vector<PatchLoop*> &loops);`

Fill `loops` with all the loops in the function

`bool getOuterLoops(vector<PatchLoop*> &loops);`

Fill `loops` with all the outermost loops in the function

`bool dominates(PatchBlock* A, PatchBlock *B);`

Return true if block `A` dominates block `B`

`PatchBlock* getImmediateDominator(PatchBlock *A);`

Return the immediate dominator of block `AïijŇNULL` if the block `A` does not have an immediate dominator.

`void getImmediateDominates(PatchBlock *A, set<PatchBlock*> &imm);`

Fill `imm` with all the blocks immediate dominated by block `A`

`void getAllDominates(PatchBlock *A, set<PatchBlock*> &dom);`

Fill `dom` with all the blocks dominated by block `A`

```
bool postDominates(PatchBlock* A, PatchBlock *B);
```

Return true if block `A` post-dominates block `B`

```
PatchBlock* getImmediatePostDominator(PatchBlock *A);
```

Return the immediate post-dominator of block `AïijŇNULL` if the block `A` does not have an immediate post-dominator.

```
void getImmediatePostDominates(PatchBlock *A, set<PatchBlock*> &imm);
```

Fill `imm` with all the blocks immediate post-dominated by block `A`

```
void getAllPostDominates(PatchBlock *A, set<PatchBlock*> &dom);
```

Fill `dom` with all the blocks post-dominated by block `A`

4.1.3 PatchBlock

Declared in: PatchCFG.h

The PatchBlock class is a wrapper of ParseAPI's Block class (has-a), which represents a basic block.

```
Address start() const;
```

Returns the lower bound of this block (the address of the first instruction).

```
Address end() const;
```

Returns the upper bound (open) of this block (the address immediately following the last byte in the last instruction).

```
Address last() const;
```

Returns the address of the last instruction in this block.

```
Address size() const;
```

Returns end() - start().

```
bool isShared();
```

Indicates whether this block is contained by multiple functions.

```
int containingFuncs() const;
```

Returns the number of functions that contain this block.

```
typedef std::map<Address, InstructionAPI::Instruction::Ptr> Insns;
void getInsns(Insns &insns) const;
```

This function outputs Instructions that are in this block to *insns*.

```
InstructionAPI::Instruction::Ptr getInsn(Address a) const;
```

Returns an Instruction that has the address a as its starting address. If no any instruction can be found in this block with the starting address a, it returns InstructionAPI::Instruction::Ptr().

```
std::string disassemble() const;
```

Returns a string containing the disassembled code for this block. This is mainly for debugging purpose.

```
bool containsCall();
```

Indicates whether this PatchBlock contains a function call instruction.

```
bool containsDynamicCall();
```

Indicates whether this PatchBlock contains any indirect function call, e.g., via function pointer.

```
PatchFunction* getCallee();
```

Returns the callee function, if this PatchBlock contains a function call; otherwise, NULL is returned.

```
PatchFunction *function() const;
```

Returns a PatchFunction that contains this PatchBlock. If there are multiple PatchFunctions containing this PatchBlock, then a random one of them is returned.

```
ParseAPI::Block *block() const;
```

Returns the ParseAPI::Block associated with this PatchBlock.

```
PatchObject* obj() const;
```

Returns the PatchObject that contains this block.

```
typedef std::vector<PatchEdge*> PatchBlock::edgelist;

const edgelist &sources();
```

Returns a list of the source PatchEdges. This PatchBlock is the target block of the returned edges.

```
const edgelist &targets();
```

Returns a list of the target PatchEdges. This PatchBlock is the source block of the returned edges.

```
template <class OutputIterator>
void getFuncs(OutputIterator result);
```

Outputs all functions containing this PatchBlock to the STL inserter *result*.

```
PatchCallback *cb() const;
```

Returns the PatchCallback object associated with this PatchBlock.

4.1.4 PatchEdge

Declared in: PatchCFG.h

The PatchEdge class is a wrapper of ParseAPI's Edge class (has-a), which joins two PatchBlocks in the CFG, indicating the type of control flow transfer instruction that joins the basic blocks to each other.

```
ParseAPI::Edge *edge() const;
```

Returns a ParseAPI::Edge associated with this PatchEdge.

```
PatchBlock *src();
```

Returns the source PatchBlock.

```
PatchBlock *trg();
```

Returns the target PatchBlock.

```
ParseAPI::EdgeTypeEnum type() const;
```

Returns the edge type (ParseAPI::EdgeTypeEnum, please see ParseAPI Manual).

```
bool sinkEdge() const;
```

Indicates whether this edge targets the special sink block, where a sink block is a block to which all unresolvable control flow instructions will be linked.

```
bool interproc() const;
```

Indicates whether the edge should be interpreted as interprocedural (e.g., calls, returns, direct branches under certain circumstances).

```
PatchCallback *cb() const;
```

Returns a Patchcallback object associated with this PatchEdge.

4.1.5 PatchLoop

Declared in: PatchCFG.h

The PatchLoop class is a wrapper of ParseAPI's Loop class (has-a). It represents code structure that may execute repeatedly.

```
PatchLoop* parent
```

Returns the loop which directly encloses this loop. NULL if no such loop.

```
bool containsAddress(Address addr)
```

Returns true if the given address is within the range of this loop's basic blocks.

```
bool containsAddressInclusive(Address addr)
```

Returns true if the given address is within the range of this loop's basic blocks or its children.

```
int getLoopEntries(vector<PatchBlock*>& entries);
```

Fills `entries` with the set of entry basic blocks of the loop. Return the number of the entries that this loop has

```
int getBackEdges(vector<PatchEdge*> &edges)
```

Sets `edges` to the set of back edges in this loop. It returns the number of back edges that are in this loop.

```
bool getContainedLoops(vector<PatchLoop*> &loops)
```

Returns a vector of loops that are nested under this loop.

```
bool getOuterLoops(vector<PatchLoop*> &loops)
```

Returns a vector of loops that are directly nested under this loop.

```
bool getLoopBasicBlocks(vector<PatchBlock*> &blocks)
```

Fills `blocks` with all basic blocks in the loop

```
bool getLoopBasicBlocksExclusive(vector<PatchBlock*> &blocks)
```

Fills `blocks` with all basic blocks in this loop, excluding the blocks of its sub loops.

```
bool hasBlock(PatchBlock *b);
```

Returns `true` if this loop contains basic block `b`.

```
bool hasBlockExclusive(PatchBlock *b);
```

Returns `true` if this loop contains basic block `b` and `b` is not in its sub loops.

```
bool hasAncestor(PatchLoop *loop)
```

Returns true if this loop is a descendant of the given loop.

```
PatchFunction * getFunction();
```

Returns the function that this loop is in.

4.1.6 PatchLoopTreeNode

Declared in: PatchCFG.h

The PatchLoopTreeNode class provides a tree interface to a collection of instances of class PatchLoop contained in a function. The structure of the tree follows the nesting relationship of the loops in a function. Each PatchLoopTreeNode contains a pointer to a loop (represented by PatchLoop), and a set of sub-loops (represented by other PatchLoopTreeNode objects). The `loop` field at the root node is always `NULL` since a function may contain multiple outer loops. The `loop` field is never `NULL` at any other node since it always corresponds to a real loop. Therefore, the outer most loops in the function are contained in the vector of `children` of the root.

Each instance of PatchLoopTreeNode is given a name that indicates its position in the hierarchy of loops. The name of each outermost loop takes the form of `loop_x`, where `x` is an integer from 1 to n, where n is the number of outer loops in the function. Each sub-loop has the name of its parent, followed by a `.y`, where `y` is 1 to m, where m is the number of sub-loops under the outer loop. For example, consider the following C function:

```
void foo() {
  int x, y, z, i;
  for (x=0; x<10; x++) {
    for (y = 0; y<10; y++)
      ...
    for (z = 0; z<10; z++)
      ...
  }
  for (i = 0; i<10; i++) {
     ...
  }
}
```

The `foo` function will have a root PatchLoopTreeNode, containing a NULL loop entry and two PatchLoopTreeNode children representing the functions outermost loops. These children would have names `loop_1` and `loop_2`, respectively representing the `x` and `i` loops. `loop_2` has no children. `loop_1` has two child PatchLoopTreeNode objects, named `loop_1.1` and `loop_1.2`, respectively representing the `y` and `z` loops.

```
PatchLoop *loop;
```

The PatchLoop instance it points to.

```
std::vector<PatchLoopTreeNode *> children;
```

The PatchLoopTreeNode instances nested within this loop.

```
const char * name();
```

Returns the hierarchical name of this loop.

```
const char * getCalleeName(unsigned int i)
```

Returns the function name of the ith callee.

```
unsigned int numCallees()
```

Returns the number of callees contained in this loop's body.

```
bool getCallees(vector<PatchFunction *> &v);
```

Fills v with a vector of the functions called inside this loop.

```
PatchLoop * findLoop(const char *name);
```

Looks up a loop by the hierarchical name

4.2 Point/Snippet Interface

4.2.1 PatchMgr

Declared in: PatchMgr.h

The PatchMgr class is the top-level class for finding instrumentation **Points**, inserting or deleting **Snippets**, and registering user-provided plugins.

```
static PatchMgrPtr create(AddrSpace* as, Instrumenter* inst = NULL,
                          PointMaker* pm = NULL);
```

This factory method creates a new PatchMgr object that performs binary code patching. It takes input three plugins, including AddrSpace *as*, Instrumenter *inst*, and PointMaker *pm*. PatchAPI uses default plugins for PointMaker and Instrumenter, if *pm* and *inst* are not specified (NULL by default).

This method returns PatchMgrPtr() if it was unable to create a new PatchMgr object.

```
Point *findPoint(Location loc, Point::Type type, bool create = true);
```

This method returns a unique Point according to a Location *loc* and a Type *type*. The Location structure is to specify a physical location of a Point (e.g., at function entry, at block entry, etc.), details of Location will be covered in Section 4.2.2. PatchAPI creates Points on demand, so if a Point is not yet created, the *create* parameter is to indicate whether to create this Point. If the Point we want to find is already created, this method simply returns a pointer to this Point from a buffer, no matter whether *create* is true or false. If the Point we want to find is not yet created, and *create* is true, then this method constructs this Point and put it in a buffer, and finally returns a Pointer to this Point. If the Point creation fails, this method also returns false. If the Point we want to find is not yet created, and *create* is false, this method returns NULL. The basic logic of finding a point can be found in the Listing 10.

Listing 10: Pseudocode of finding a point

```
if (point is in the buffer) {
  return point;
} else {
  if (create == true) {
    create point
    if (point creation fails) return NULL;
    put the point in the buffer
  } else {
    return NULL;
  }
}
```

```
template <class OutputIterator>
bool findPoint(Location loc, Point::Type type, OutputIterator outputIter,
               bool create = true);
```

This method finds a Point at a physical Location *loc* with a *type*. It adds the found Point to *outputIter* that is a STL inserter. The point is created on demand. If the Point is already created, then this method outputs a pointer to this Point from a buffer. Otherwise, the *create* parameter indicates whether to create this Point.

This method returns true if a point is found, or the *create* parameter is false; otherwise, it returns false.

```
template <class OutputIterator>
bool findPoints(Location loc, Point::Type types, OutputIterator outputIter,
                bool create = true);
```

This method finds Points at a physical Location *loc* with composite *types* that are combined using the overloaded operator "|". This function outputs Points to the STL inserter *outputIter*. The point is created on demand. If the Point is already created, then this method outputs a pointer to this Point from a buffer. Otherwise, the *create* parameter indicates whether to create this Point.

This method returns true if a point is found, or the *create* parameter is false; otherwise, it returns false.

```
template <class FilterFunc,
          class FilterArgument,
          class OutputIterator>
bool findPoints(Location loc, Point::Type types, FilterFunc filter_func,
                FilterArgument filter_arg, OutputIterator outputIter,
                bool create = true);
```

This method finds Points at a physical Location *loc* with composite *types* that are combined using the overloaded operator "|". Then, this method applies a filter functor *filter_func* with an argument *filter_arg* on each found Point. The method outputs Points to the inserter *outputIter*. The point is created on demand. If the Point is already created, then this method returns a pointer to this Point from a buffer. Otherwise, the *create* parameter indicates whether to create this Point.

If no any Point is created, then this method returns false; otherwise, true is returned. The code below shows the prototype of an example functor.

Listing 11: Code template for the filter function in findPoint

```
template <class T>
class FilterFunc {
  public:
    bool operator()(Point::Type type, Location loc, T arg) {
      // The logic to check whether this point is what we need
      return true;
    }
};
```

In the functor FilterFunc above, programmers check each candidate Point by looking at the Point::Type, Location, and the user-specified parameter *arg*. If the return value is true, then the Point being checked will be put in the STL inserter *outputIter*; otherwise, this Point will be discarded.

```
struct Scope {
  Scope(PatchBlock *b);
  Scope(PatchFunction *f, PatchBlock *b);
  Scope(PatchFunction *f);
};
```

The Scope structure specifies the scope to find points, where a scope could be a function, or a basic block. This is quite useful if programmers don't know the exact Location, then they can use Scope as a wildcard. A basic block can be contained in multiple functions. The second constructor only specifies the block *b* in a particular function *f*.

```
template <class FilterFunc,
          class FilterArgument,
          class OutputIterator>
bool findPoints(Scope scope, Point::Type types, FilterFunc filter_func,
                FilterArgument filter_arg, OutputIterator output_iter,
                bool create = true);
```

This method finds points in a *scope* with certain *types* that are combined together by using the overloaded operator "|". Then, this method applies the filter functor *filter_func* on each found Point. It outputs Points where *filter_func* returns true to the STL inserter *output_iter*. Points are created on demand. If some points are already created, then this method outputs pointers to them from a buffer. Otherwise, the *create* parameter indicates whether to create Points.

If no any Point is created, then this function returns false; otherwise, true is returned.

```
template <class OutputIterator>
bool findPoints(Scope scope, Point::Type types, OutputIterator output_iter, bool create = true);
```

This method finds points in a *scope* with certain *types* that are combined together by using the overloaded operator "|". It outputs the found points to the STL inserter *output_iter*. If some points are already created, then this method outputs pointers to them from a buffer. Otherwise, the *create* parameter indicates whether to create Points.

If no any Point is created, then this method returns false; otherwise, true is returned.

```
bool removeSnippet(InstancePtr);
```

This method removes a snippet Instance.

It returns false if the point associated with this Instance cannot be found; otherwise, true is returned.

```
template <class FilterFunc,
          class FilterArgument>
bool removeSnippets(Scope scope, Point::Type types, FilterFunc filter_func,
                    FilterArgument filter_arg);
```

This method deletes ALL snippet instances at certain points in certain *scope* with certain *types*, and those points pass the test of *filter_func*.

If no any point can be found, this method returns false; otherwise, true is returned.

```
bool removeSnippets(Scope scope, Point::Type types);
```

This method deletes ALL snippet instances at certain points in certain *scope* with certain *types*.

If no any point can be found, this method returns false; otherwise, true is returned.

```
void destroy(Point *point);
```

This method is to destroy the specified *Point*.

```
AddrSpace* as() const;
PointMaker* pointMaker() const;
Instrumenter* instrumenter() const;
```

The above three functions return the corresponding plugin: AddrSpace, PointMaker, Instrumenter.

4.2.2 Point

Declared in: Point.h

The Point class is in essence a container of a list of snippet instances. Therefore, the Point class has methods similar to those in STL.

```
struct Location {
   static Location Function(PatchFunction *f);
   static Location Block(PatchBlock *b);
   static Location BlockInstance(PatchFunction *f, PatchBlock *b, bool trusted = false);
   static Location Edge(PatchEdge *e);
```

```
   static Location EdgeInstance(PatchFunction *f, PatchEdge *e);
   static Location Instruction(PatchBlock *b, Address a);
   static Location InstructionInstance(PatchFunction *f, PatchBlock *b, Address a);
   static Location InstructionInstance(PatchFunction *f, PatchBlock *b, Address a,
                                       InstructionAPI::Instruction::Ptr i,
                                       bool trusted = false);
   static Location EntrySite(PatchFunction *f, PatchBlock *b, bool trusted = false);
   static Location CallSite(PatchFunction *f, PatchBlock *b);
   static Location ExitSite(PatchFunction *f, PatchBlock *b);
};
```

The Location structure uniquely identifies the physical location of a point. A Location object plus a Point::Type value uniquely identifies a point, because multiple Points with different types can exist at the same physical location. The Location structure provides a set of static functions to create an object of Location, where each function takes the corresponding CFG structures to identify a physical location. In addition, some functions above (e.g., InstructionInstance) takes input the *trusted* parameter that is to indicate PatchAPI whether the CFG structures passed in is trusted. If the *trusted* parameter is false, then PatchAPI would have additional checking to verify the CFG structures passed by users, which causes nontrivial overhead.

```
enum Point::Type {
  PreInsn,
  PostInsn,
  BlockEntry,
  BlockExit,
  BlockDuring,
  FuncEntry,
  FuncExit,
  FuncDuring,
  EdgeDuring,
  PreCall,
  PostCall,
  OtherPoint,
  None,
  InsnTypes = PreInsn | PostInsn,
  BlockTypes = BlockEntry | BlockExit | BlockDuring,
  FuncTypes = FuncEntry | FuncExit | FuncDuring,
  EdgeTypes = EdgeDuring,
  CallTypes = PreCall | PostCall
};
```

The enum Point::Type specifies the logical point type. Multiple enum values can be OR-ed to form a composite type. For example, the composite type of "PreCall | BlockEntry | FuncExit" is to specify a set of points with the type PreCall, or BlockEntry, or FuncExit.

```
typedef std::list<InstancePtr>::iterator instance_iter;
instance_iter begin();
instance_iter end();
```

The method begin() returns an iterator pointing to the beginning of the container storing snippet Instances, while the method end() returns an iterator pointing to the end of the container (past the last element).

```
InstancePtr pushBack(SnippetPtr);
InstancePtr pushFront(SnippetPtr);
```

Multiple instances can be inserted at the same Point. We maintain the instances in an ordered list. The pushBack method is to push the specified Snippet to the end of the list, while the pushFront method is to push to the front of the list.

Both methods return the Instance that uniquely identifies the inserted snippet.

```
bool remove(InstancePtr instance);
```

This method removes the given snippet *instance* from this Point.

```
void clear();
```

This method removes all snippet instances inserted to this Point.

```
size_t size();
```

Returns the number of snippet instances inserted at this Point.

```
Address addr() const;
```

Returns the address associated with this point, if it has one; otherwise, it returns 0.

```
Type type() const;
```

Returns the Point type of this point.

```
bool empty() const;
```

Indicates whether the container of instances at this Point is empty or not.

```
PatchFunction* getCallee();
```

Returns the function that is invoked at this Point, which should have Point::Type of Point::PreCall or Point::PostCall. It there is not a function invoked at this point, it returns NULL.

```
const PatchObject* obj() const;
```

Returns the PatchObject where the Point resides.

```
const InstructionAPI::Instruction::Ptr insn() const;
```

Returns the Instruction where the Point resides.

```
PatchFunction* func() const;
```

Returns the function where the Point resides.

```
PatchBlock* block() const;
```

Returns the PatchBlock where the Point resides.

```
PatchEdge* edge() const;
```

Returns the Edge where the Point resides.

```
PatchCallback *cb() const;
```

Returns the PatchCallback object that is associated with this Point.

```
static bool TestType(Point::Type types, Point::Type type);
```

This static method tests whether a set of *types* contains a specific *type*.

```
static void AddType(Point::Type& types, Point::Type type);
```

This static method adds a specific *type* to a set of *types*.

```
static void RemoveType(Point::Type& types, Point::Type trg);
```

This static method removes a specific *type* from a set of *types*.

4.2.3 Instance

Declared in: Point.h

The Instance class is a representation of a particular snippet inserted at a particular point. If a Snippet is inserted to N points or to the same point for N times (N > 1), then there will be N Instances.

```
bool destroy();
```

This method destroys the snippet Instance itself.

```
Point* point() const;
```

Returns the Point where the Instance is inserted.

```
SnippetPtr snippet() const;
```

Returns the Snippet. Please note that, the same Snippet may have multiple instances inserted at different Points or the same Point.

4.3 Callback Interface

4.3.1 PatchCallback

Declared in: PatchCallback.h

The PatchAPI CFG layer may change at runtime due to program events (e.g., a program loading additional code or overwriting its own code with new code). The `PatchCallback` interface allows users to specify callbacks they wish to occur whenever the PatchAPI CFG changes.

```
virtual void destroy_cb(PatchBlock *);
virtual void destroy_cb(PatchEdge *);
virtual void destroy_cb(PatchFunction *);
virtual void destroy_cb(PatchObject *);
```

Programmers implement the above virtual methods to handle the event of destroying a PatchBlock, a PatchEdge, a PatchFunction, or a PatchObject respectively. All the above methods will be called before corresponding object destructors are called.

```
virtual void create_cb(PatchBlock *);
virtual void create_cb(PatchEdge *);
virtual void create_cb(PatchFunction *);
virtual void create_cb(PatchObject *);
```

Programmers implement the above virtual methods to handle the event of creating a PatchBlock, a PatchEdge, a PatchFunction, or a PatchObject respectively. All the above methods will be called after the objects are created.

```
virtual void split_block_cb(PatchBlock *first, PatchBlock *second);
```

Programmers implement the above virtual method to handle the event of splitting a PatchBlock as a result of a new edge being discovered. The above method will be called after the block is split.

```
virtual void remove_edge_cb(PatchBlock *, PatchEdge *, edge_type_t);
virtual void add_edge_cb(PatchBlock *, PatchEdge *, edge_type_t);
```

Programmers implement the above virtual methods to handle the events of removing or adding an PatchEdge respectively. The method remove_edge_cb will be called before the event triggers, while the method add_edge_cb will be called after the event triggers.

```
virtual void remove_block_cb(PatchFunction *, PatchBlock *);
virtual void add_block_cb(PatchFunction *, PatchBlock *);
```

Programmers implement the above virtual methods to handle the events of removing or adding a PatchBlock respectively. The method remove_block_cb will be called before the event triggers, while the method add_block_cb will be called after the event triggers.

```
virtual void create_cb(Point *pt);
virtual void destroy_cb(Point *pt);
```

Programmers implement the create_cb method above, which will be called after the Point *pt* is created. And, programmers implement the destroy_cb method, which will be called before the point *pt* is deleted.

```
virtual void change_cb(Point *pt, PatchBlock *first, PatchBlock *second);
```

Programmers implement this method, which is to be invoked after a block is split. The provided Point belonged to the first block and is being moved to the second.

5 Modification API Reference

This section describes the modification interface of PatchAPI. While PatchAPI's main goal is to allow users to insert new code into a program, a secondary goal is to allow safe modification of the original program code as well.

To modify the binary, a user interacts with the `PatchModifier` class to manipulate a PatchAPI CFG. CFG modifications are then instantiated as new code by the PatchAPI. For example, if PatchAPI is being used as part of Dyninst, executing a `finalizeInsertionSet` will generate modified code.

The three key benefits of the PatchAPI modification interface are abstraction, safety, and interactivity. We use the CFG as a mechanism for transforming binaries in a platform-independent way that requires no instruction-level knowledge by the user. These transformations are limited to ensure that the CFG can always be used to instantiate code, and thus the user can avoid unintended side-effects of modification. Finally, modifications to the CFG are represented in that CFG, allowing users to iteratively combine multiple CFG transformations to achieve their goals.

Since modification can modify the CFG, it may invalidate any analyses the user has performed over the CFG. We suggest that users take advantage of the callback interface described in Section 4.3.1 to update any such analysis information.

The PatchAPI modification capabilities are currently in beta; if you experience any problems or bugs, please contact `bugs@dyninst.org`.

Many of these methods return a boolean type; true indicates a successful operation, and false indicates a failure. For methods that return a pointer, a `NULL` return value indicates a failure.

```
bool redirect(PatchEdge *edge, PatchBlock *target);
```

Redirects the edge specified by `edge` to a new target specified by `target`. In the current implementation, the edge may not be indirect.

```
PatchBlock *split(PatchBlock *orig, Address addr,
                  bool trust = false,
                  Address newlast = (Address) -1);
```

Splits the block specified by `orig`, creating a new block starting at `addr`. If `trust` is true, we do not verify that `addr` is a valid instruction address; this may be useful to reduce overhead. If `newlast` is not -1, we use it as the last instruction address of the first block. All Points are updated to belong to the appropriate block. The second block is returned.

```
bool remove(std::vector<PatchBlock *> &blocks, bool force = true)
```

Removes the blocks specified by `blocks` from the CFG. If `force` is true, blocks are removed even if they have incoming edges; this may leave the CFG in an unsafe state but may be useful for reducing overhead.

```
bool remove(PatchFunction *func)
```

Removes `func` and all of its non-shared blocks from the CFG; any shared blocks remain.

```
class InsertedCode {
  typedef boost::shared_ptr<...> Ptr;
  PatchBlock *entry();
  const std::vector<PatchEdge *> &exits();
  const std::set<PatchBlock *> &blocks();
}

InsertedCode::Ptr insert(PatchObject *obj, SnippetPtr snip, Point *point);
InsertedCode::Ptr insert(PatchObject *obj, void *start, unsigned size);
```

Methods for inserting new code into a CFG. The `InsertedCode` structure represents a CFG subgraph generated by inserting new code; the graph has a single entry point and multiple exits, represented by edges to the sink node. The first `insert` call takes a PatchAPI Snippet structure and a Point that is used to generate that Snippet; the point is only passed through to the snippet code generator and thus may be `NULL` if the snippet does not use Point information. The second `insert` call takes a raw code buffer.

6 Plugin API Reference

This section describes the various plugin interfaces for extending PatchAPI. We expect that most users should not have to ever explicitly use an interface from this section; instead, they will use plugins previously implemented by PatchAPI developers.

As with the public interface, all objects and methods in this section are in the "Dyninst::PatchAPI" namespace.

6.1 AddrSpace

Declared in: AddrSpace.h

The AddrSpace class represents the address space of a **Mutatee**, where it contains a collection of **PatchObjects** that represent shared libraries or a binary executable. In addition, programmers implement some memory management interfaces in the AddrSpace class to determine the type of the code patching - 1st party, 3rd party, or binary rewriting.

```
virtual bool write(PatchObject* obj, Address to, Address from, size_t size);
```

This method copies *size*-byte data stored at the address *from* on the **Mutator** side to the address *to* on the **Mutatee** side. The parameter *to* is the relative offset for the PatchObject *obj*, if the instrumentation is for binary rewriting; otherwise *to* is an absolute address.

If the write operation succeeds, this method returns true; otherwise, false.

```
virtual Address malloc(PatchObject* obj, size_t size, Address near);
```

This method allocates a buffer of *size* bytes on the **Mutatee** side. The address *near* is a relative address in the object *obj*, if the instrumentation is for binary rewriting; otherwise, *near* is an absolute address, where this method tries to allocate a buffer near the address *near*.

If this method succeeds, it returns a non-zero address; otherwise, it returns 0.

```
virtual Address realloc(PatchObject* obj, Address orig, size_t size);
```

This method reallocates a buffer of *size* bytes on the **Mutatee** side. The original buffer is at the address *orig*. This method tries to reallocate the buffer near the address *orig*, where *orig* is a relative address in the PatchObject *obj* if the instrumentation is for binary rewriting; otherwise, *orig* is an absolute address.

If this method succeeds, it returns a non-zero address; otherwise, it returns 0.

```
virtual bool free(PatchObject* obj, Address orig);
```

This method deallocates a buffer on the **Mutatee** side at the address *orig*. If the instrumentation is for binary rewriting, then the parameter *orig* is a relative address in the object *obj*; otherwise, *orig* is an absolute address.

If this method succeeds, it returns true; otherwise, it returns false.

```
virtual bool loadObject(PatchObject* obj);
```

This method loads a PatchObject into the address space. If this method succeeds, it returns true; otherwise, it returns false.

```
typedef std::map<const ParseAPI::CodeObject*, PatchObject*> AddrSpace::ObjMap;

ObjMap& objMap();
```

Returns a set of mappings from ParseAPI::CodeObjects to PatchObjects, where PatchObjects in all mappings represent all binary objects (either executable or libraries loaded) in this address space.

```
PatchObject* executable();
```

Returns the PatchObject of the executable of the **Mutatee**.

```
PatchMgrPtr mgr();
```

Returns the PatchMgr's pointer, where the PatchMgr contains this address space.

6.2 Snippet

Declared in: Snippet.h

The Snippet class allows programmers to customize their own snippet representation and the corresponding mini-compiler to translate the representation into the binary code.

```
static Ptr create(Snippet* a);
```

Creates an object of the Snippet.

```
virtual bool generate(Point *pt, Buffer &buf);
```

Users should implement this virtual function for generating binary code for the snippet.

Returns false if code generation failed catastrophically. Point *pt* is an in-param that identifies where the snippet is being generated. Buffer *buf* is an out-param that holds the generated code.

6.3 Command

Declared in: Command.h

The Command class represents an instrumentation request (e.g., snippet insertion or removal), or an internal logical step in the code patching (e.g., install instrumentation).

```
virtual bool run() = 0;
```

Executes the normal operation of this Command.

It returns true on success; otherwise, it returns false.

```
virtual bool undo() = 0;
```

Undoes the operation of this Command.

```
virtual bool commit();
```

Implements the transactional semantics: all succeed, or all fail. Basically, it performs such logic:

```
if (run()) {
  return true;
} else {
  undo();
  return false;
}
```

6.4 BatchCommand

Declared in: Command.h

The BatchCommand class inherits from the Command class. It is actually a container of a list of Commands that will be executed in a transaction: all Commands will succeed, or all will fail.

```
typedef std::list<CommandPtr> CommandList;

CommandList to_do_;
CommandList done_;
```

This class has two protected members *to_ do_* and *done_*, where *to_ do_* is a list of Commands to execute, and *done_* is a list of Commands that are executed.

```
virtual bool run();
virtual bool undo();
```

The method run() of BatchCommand invokes the run() method of each Command in *to_ do_* in order, and puts the finished Commands in *done_*. The method undo() of BatchCommand invokes the undo() method of each Command in *done _* in order.

```
void add(CommandPtr command);
```

This method adds a Command into *to_ do_*.

```
void remove(CommandList::iterator iter);
```

This method removes a Command from *to_ do_*.

6.5 Instrumenter

Declared in: Command.h

The Instrumenter class inherits BatchCommand to encapsulate the core code patching logic, which includes binary code generation. Instrumenter would contain several logical steps that are individual Commands.

```
CommandList user_commands_;
```

This class has a protected data member *user_commands_* that contains all Commands issued by users, e.g., snippet insertion. This is to facilitate the implementation of the instrumentation engine.

```
static InstrumenterPtr create(AddrSpacePtr as);
```

Returns an instance of Instrumenter, and it takes input the address space *as* that is going to be instrumented.

```
virtual bool replaceFunction(PatchFunction* oldfunc, PatchFunction* newfunc);
```

Replaces a function *oldfunc* with a new function *newfunc*.

It returns true on success; otherwise, it returns false.

```
virtual bool revertReplacedFunction(PatchFunction* oldfunc);
```

Undoes the function replacement for *oldfunc*.

It returns true on success; otherwise, it returns false.

```
typedef std::map<PatchFunction*, PatchFunction*> FuncModMap;
```

The type FuncModMap contains mappings from an PatchFunction to another PatchFunction.

```
virtual FuncModMap& funcRepMap();
```

Returns the FuncModMap that contains a set of mappings from an old function to a new function, where the old function is replaced by the new function.

```
virtual bool wrapFunction(PatchFunction* oldfunc, PatchFunction* newfunc, string name);
```

Replaces all calls to *oldfunc* with calls to wrapper *newfunc* (similar to function replacement). However, we create a copy of original using the *name* that can be used to call the original. The wrapper code would look like follows:

```
void *malloc_wrapper(int size) {
  // do stuff
  void *ret = malloc_clone(size);
  // do more stuff
  return ret;
}
```

This interface requires the user to give us a name (as represented by clone) for the original function. This matches current techniques and allows users to use indirect calls (function pointers).

```
virtual bool revertWrappedFunction(PatchFunction* oldfunc);
```

Undoes the function wrapping for *oldfunc*.

It returns true on success; otherwise, it returns false.

```
virtual FuncModMap& funcWrapMap();
```

The type FuncModMap contains mappings from the original PatchFunction to the wrapper PatchFunction.

```
bool modifyCall(PatchBlock *callBlock, PatchFunction *newCallee,
                PatchFunction *context = NULL);
```

Replaces the function that is invoked in the basic block *callBlock* with the function *newCallee*. There may be multiple functions containing the same *callBlock*, so the *context* parameter specifies in which function the *callBlock* should be modified. If *context* is NULL, then the *callBlock* would be modified in all PatchFunctions that contain it. If the *newCallee* is NULL, then the *callBlock* is removed.

It returns true on success; otherwise, it returns false.

```
bool revertModifiedCall(PatchBlock *callBlock, PatchFunction *context = NULL);
```

Undoes the function call modification for *oldfunc*. There may be multiple functions containing the same *callBlock*, so the *context* parameter specifies in which function the *callBlock* should be modified. If *context* is NULL, then the *callBlock* would be modified in all PatchFunctions that contain it.

It returns true on success; otherwise, it returns false.

```
bool removeCall(PatchBlock *callBlock, PatchFunction *context = NULL);
```

Removes the *callBlock*, where a function is invoked. There may be multiple functions containing the same *callBlock*, so the *context* parameter specifies in which function the *callBlock* should be modified. If *context* is NULL, then the *callBlock* would be modified in all PatchFunctions that contain it.

It returns true on success; otherwise, it returns false.

```
typedef map<PatchBlock*,        // B  : A call block
            map<PatchFunction*, // F_c: Function context
                PatchFunction*> // F  : The function to be replaced
           > CallModMap;
```

The type CallModMap maps from B -> F_c -> F, where B identifies a call block, and F_c identifies an (optional) function context for the replacement. If F_c is not specified, we use NULL. F specifies the replacement callee; if we want to remove the call entirely, we use NULL.

```
CallModMap& callModMap();
```

Returns the CallModMap for function call replacement / removal.

```
AddrSpacePtr as() const;
```

Returns the address space associated with this Instrumenter.

6.6 Patcher

Declared in: Command.h

The class Patcher inherits from the class BatchCommand. It accepts instrumentation requests from users, where these instrumentation requests are Commands (e.g., snippet insertion). Furthermore, Patcher implicitly adds an instance of Instrumenter to the end of the Command list to generate binary code and install the instrumentation.

```
Patcher(PatchMgrPtr mgr)
```

The constructor of Patcher takes input the relevant PatchMgr *mgr*.

```
virtual bool run();
```

Performs the same logic as BatchCommand::run(), except that this function implicitly adds an internal Command – Instrumenter, which is executed after all other Commands in the *to_ do_* .

6.7 CFGMaker

Declared in: CFGMaker.h

The CFGMaker class is a factory class that constructs the above CFG structures (PatchFunction, PatchBlock, and PatchEdge). The methods in this class are used by PatchObject. Programmers can extend PatchFunction, PatchBlock and PatchEdge by annotating their own data, and then use this class to instantiate these CFG structures.

```
virtual PatchFunction* makeFunction(ParseAPI::Function* func, PatchObject* obj);
virtual PatchFunction* copyFunction(PatchFunction* func, PatchObject* obj);

virtual PatchBlock* makeBlock(ParseAPI::Block* blk, PatchObject* obj);
virtual PatchBlock* copyBlock(PatchBlock* blk, PatchObject* obj);

virtual PatchEdge* makeEdge(ParseAPI::Edge* edge, PatchBlock* src,
                            PatchBlock* trg, PatchObject* obj);
virtual PatchEdge* copyEdge(PatchEdge* edge, PatchObject* obj);
```

Programmers implement the above virtual methods to instantiate a CFG structure (either a PatchFunction, a PatchBlock, or a PatchEdge) or to copy (e.g., when forking a new process).

6.8 PointMaker

Declared in: Point.h

The PointMaker class is a factory class that constructs instances of the Point class. The methods of the PointMaker class are invoked by PatchMgr's findPoint methods. Programmers can extend the Point class, and then implement a set of virtual methods in this class to instantiate the subclasses of Point.

```
PointMaker(PatchMgrPtr mgr);
```

The constructor takes input the relevant PatchMgr *mgr*.

```
virtual Point *mkFuncPoint(Point::Type t, PatchMgrPtr m, PatchFunction *f);
virtual Point *mkFuncSitePoint(Point::Type t, PatchMgrPtr m, PatchFunction *f, PatchBlock *b);
virtual Point *mkBlockPoint(Point::Type t, PatchMgrPtr m, PatchBlock *b, PatchFunction *context);
virtual Point *mkInsnPoint(Point::Type t, PatchMgrPtr m, PatchBlock *, Address a,
                           InstructionAPI::Instruction::Ptr i, PatchFunction *context);
virtual Point *mkEdgePoint(Point::Type t, PatchMgrPtr m, PatchEdge *e, PatchFunction *context);
```

Programmers implement the above virtual methods to instantiate the subclasses of Point.

6.9 Default Plugin

6.10 PushFrontCommand and PushBackCommand

Declared in: Command.h

The class PushFrontCommand and the class PushBackCommand inherit from the Command class. They are to insert a snippet to a point. A point maintains a list of snippet instances. PushFrontCommand would add the new snippet instance to the front of the list, while PushBackCommand would add to the end of the list.

```
static Ptr create(Point* pt, SnippetPtr snip);
```

This static method creates an object of PushFrontCommand or PushBackCommand.

```
InstancePtr instance();
```

Returns a snippet instance that is inserted at the point.

6.11 RemoveSnippetCommand

Declared in: Command.h

The class RemoveSnippetCommand inherits from the Command class. It is to delete a snippet Instance.

```
static Ptr create(InstancePtr instance);
```

This static function creates an instance of RemoveSnippetCommand.

6.12 RemoveCallCommand

Declared in: Command.h

The class RemoveCallCommand inherits from the class Command. It is to remove a function call.

```
static Ptr create(PatchMgrPtr mgr, PatchBlock* call_block, PatchFunction* context = NULL);
```

This static method takes input the relevant PatchMgr *mgr*, the *call_ block* that contains the function call to be removed, and the PatchFunction *context*. There may be multiple PatchFunctions containing the same *call_ block*. If the *context* is NULL, then the *call_ block* would be deleted from all PatchFunctions that contains it; otherwise, the *call_ block* would be deleted only from the PatchFuncton *context*.

6.13 ReplaceCallCommand

Declared in: Command.h

The class ReplaceCallCommand inherits from the class Command. It is to replace a function call with another function.

```
static Ptr create(PatchMgrPtr mgr, PatchBlock* call_block,
                  PatchFunction* new_callee, PatchFunction* context);
```

This Command replaces the *call_ block* with the new PatchFunction *new_ callee*. There may be multiple functions containing the same *call_ block*, so the *context* parameter specifies in which function the *call_ block* should be replaced. If *context* is NULL, then the *call_ block* would be replaced in all PatchFunctions that contains it.

6.14 ReplaceFuncCommand

Declared in: Command.h

The class ReplaceFuncCommand inherits from the class Command. It is to replace an old function with the new one.

```
static Ptr create(PatchMgrPtr mgr, PatchFunction* old_func, PatchFunction* new_func);
```

This Command replaces the old PatchFunction *old_func* with the new PatchFunction *new_func*.

A PatchAPI for Dyninst Programmers

The PatchAPI is a Dyninst component and as such is accessible through the main Dyninst interface (BPatch objects). However, the PatchAPI instrumentation and CFG models differ from the Dyninst models in several critical ways that should be accounted for by users. This section summarizes those differences and describes how to access PatchAPI abstractions from the DyninstAPI interface.

A.1 Differences Between DyninstAPI and PatchAPI

The DyninstAPI and PatchAPI differ primarily in their CFG representations and instrumentation point abstractions. In general, PatchAPI is more powerful and can better represent complex binaries (e.g., highly optimized code or malware). In order to maintain backwards compatibility, the DyninstAPI interface has not been extended to match the PatchAPI. As a result, there are some caveats.

The PatchAPI uses the same CFG model as the ParseAPI. The primary representation is an interprocedural graph of basic blocks and edges. Functions are defined on top of this graph as collections of blocks. **A block may be contained by more than one function;** we call this the *shared block* model. Functions are defined to have a single entry block, and functions may overlap if they contain the same blocks. Call and return edges exist in the graph, and therefore traversing the graph may enter different functions. PatchAPI users may specify instrumenting a particular block within a particular function (a *block instance*) by specifying both the block and the function.

The DyninstAPI uses a historic CFG model. The primary representation is the function. Functions contain a intraprocedural graph of blocks and edges. As a result, a basic block belongs to only one function, but two blocks from different functions may be *clones* of each other. No interprocedural edges are represented in the graph, and thus traversing the CFG from a particular function is guaranteed to remain inside that function.

As a result, multiple DyninstAPI blocks may map to the same PatchAPI block. If instrumenting a particular block instance is desired, the user should provide both the DyninstAPI basic block and function.

In addition, DyninstAPI uses a *module* abstraction, where a `BPatch_module` represents a collection of functions from a particular source file (for the executable) or from an entire library (for all libraries). PatchAPI, like ParseAPI, instead uses an *object* representation, where a `PatchObject` object represents a collection of functions from a file on disk (executable or libraries).

The instrumentation point (*instPoint*) models also differ between DyninstAPI and PatchAPI. We classify an instPoint either as a *behavior* point (e.g., function entry) or *location* point (e.g., a particular instruction). PatchAPI fully supports both of these models, with the added extension that a location point explicitly specifies whether instrumentation will execute before or after the corresponding location. Dyninst does not support the behavior model, instead mapping behavior instPoints to a corresponding instruction. For example, if a user requests a function entry instPoint they instead receive an instPoint for the first instruction in the function. These may not always be the same (see Bernat_AWAT). In addition, location instPoints represent an instruction, and the user must later specify whether they wish to instrument before or after that instruction.

As a result, there are complications for using both DyninstAPI and PatchAPI. We cannot emphasize enough, though, that users *can combine DyninstAPI and PatchAPI* with some care. Doing so offers several benefits:

- The ability to extend legacy code that is written for DyninstAPI.
- The ability to use the DyninstAPI extensions and plugins for PatchAPI, including snippet-based or dynC-based code generation and our instrumentation optimizer.

We suggest the following best practices to be followed when coding for PatchAPI via Dyninst:

- For legacy code, do not attempt to map between DyninstAPI instPoints and PatchAPI instPoints. Instead, use DyninstAPI CFG objects to acquire PatchAPI CFG objects, and use a `PatchMgr` (acquired through a `BPatch_addressSpace`) to look up PatchAPI instPoints.
- For new code, acquire a `PatchMgr` directly from a `BPatch_addressSpace` and use its methods to look up both CFG objects and instPoints.

A.2 PatchAPI accessor methods in Dyninst

To access a PatchAPI class from a Dyninst class, use the `PatchAPI::convert` function, as in the following example:

```
BPatch_basicBlock *bp_block = ...;
PatchAPI::PatchBlock *block = PatchAPI::convert(bp_block);
```

We support the following mappings, where all PatchAPI objects are within the `Dyninst::PatchAPI` namespace:

From	To	Comments
`BPatch_function`	`PatchFunction`	
`BPatch_basicBlock`	`PatchBlock`	See above.
`BPatch_edge`	`PatchEdge`	See above.
`BPatch_module`	`PatchObject`	See above.
`BPatch_image`	`PatchMgr`	
`BPatch_addressSpace`	`PatchMgr`	
`BPatch_snippet`	`Snippet`	

We do not support a direct mapping between `BPatch_point`s and `Point`s, as the failure of Dyninst to properly represent behavior instPoints leads to confusing results. Instead, use the PatchAPI point lookup methods.

Paradyn Parallel Performance Tools

SymtabAPI Programmer's Guide

9.2 Release
June 2016

Computer Sciences Department
University of Wisconsin–Madison
Madison, WI 53706

Computer Science Department
University of Maryland
College Park, MD 20742

Email dyninst-api@cs.wisc.edu
Web www.dyninst.org

Contents

1 Introduction

SymtabAPI is a multi-platform library for parsing symbol tables, object file headers and debug information. SymtabAPI currently supports the ELF (IA-32, AMD-64, and POWER) and PE (Windows) object file formats. In addition, it also supports the DWARF and stabs debugging formats.

The main goal of this API is to provide an abstract view of binaries and libraries across multiple platforms. An abstract interface provides two benefits: it simplifies the development of a tool since the complexity of a particular file format is hidden, and it allows tools to be easily ported between platforms. Each binary object file is represented in a canonical platform independent manner by the API. The canonical format consists of four components: a header block that contains general information about the object (e.g., its name and location), a set of symbol lists that index symbols within the object for fast lookup, debug information (type, line number and local variable information) present in the object file and a set of additional data that represents information that may be present in the object (e.g., relocation or exception information). Adding a new format requires no changes to the interface and hence will not affect any of the tools that use the SymtabAPI.

Our other design goal with SymtabAPI is to allow users and tool developers to easily extend or add symbol or debug information to the library through a platform-independent interface. Often times it is impossible to satify all the requirements of a tool that uses SymtabAPI, as those requirements can vary from tool to tool. So by providing extensible structures, SymtabAPI allows tools to modify any structure to fit their own requirements. Also, tools frequently use more sophisticated analyses to augment the information available from the binary directly; it should be possible to make this extra information available to the SymtabAPI library. An example of this is a tool operating on a stripped binary. Although the symbols for the majority of functions in the binary may be missing, many can be determined via more sophisticated analysis. In our model, the tool would then inform the SymtabAPI library of the presence of these functions; this information would be incorporated and available for subsequent analysis. Other examples of such extensions might involve creating and adding new types or adding new local variables to certain functions.

2 Abstractions

SymtabAPI provides a simple set of abstractions over complicated data structures which makes it straight-forward to use. The SymtabAPI consists of five classes of interfaces: the symbol table interface, the type interface, the line map interface, the local variable interface, and the address translation interface.

Figure 1 shows the ownership hierarchy for the SymtabAPI classes. Ownership here is a "contains" relationship; if one class owns another, then instances of the owner class maintain an exclusive instance of the other. For example, each Symtab class instance contains multiple instances of class Symbol and each Symbol class instance belongs to one Symtab class instance. Each of four interfaces and the classes belonging to these interfaces are described in the rest of this section. The API functions in each of the classes are described in detail in Section 6.

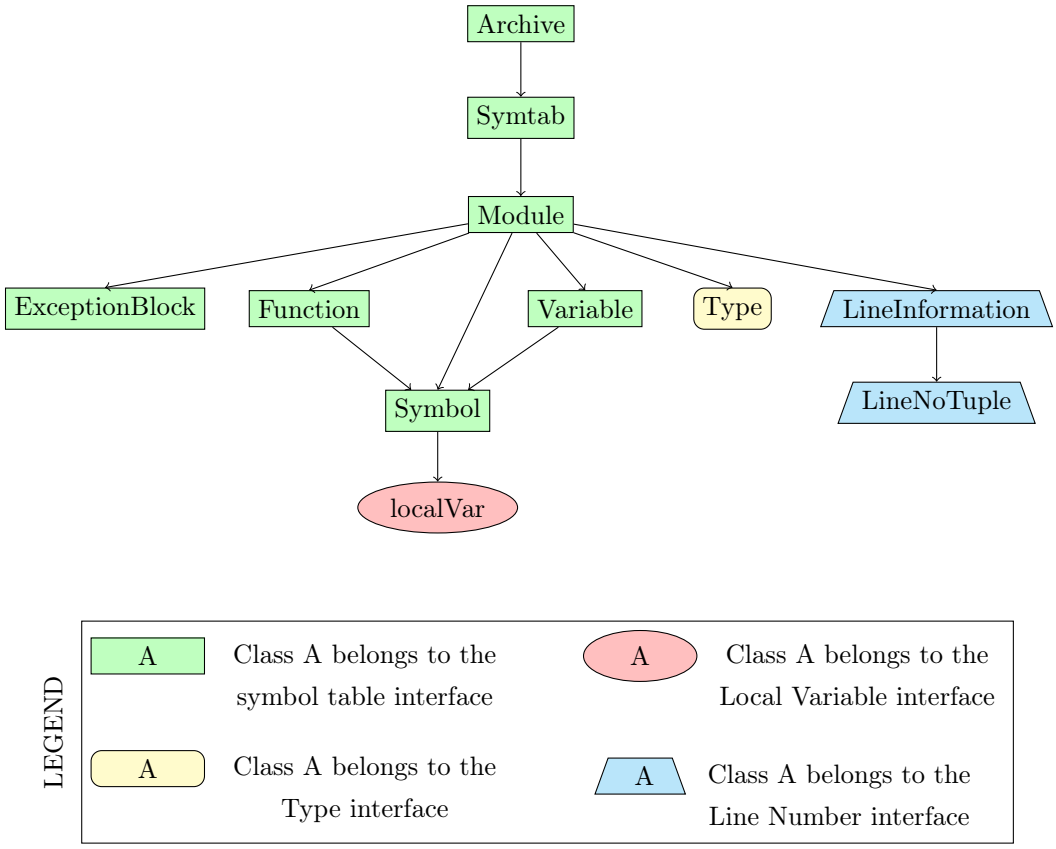


Figure 1: SymtabAPI Object Ownership Diagram

2.1 Symbol Table Interface

The symbol table interface is responsible for parsing the object file and handling the look-up and addition of new symbols. It is also responsible for the emit functionality that SymtabAPI supports. The Symtab and the Module classes inherit from the LookupInterface class, an abstract class, ensuring the same lookup function signatures for both Module and Symtab classes.

Symtab A Symtab class object represents either an object file on-disk or in-memory that the SymtabAPI library operates on.

Symbol A Symbol class object represents an entry in the symbol table.

Module A Module class object represents a particular source file in cases where multiple files were compiled into a single binary object; if this information is not present, we use a single default module.

Archive An Archive class object represents a collection of binary objects stored in a single file (e.g., a static archive).

ExceptionBlock An ExceptionBlock class object represents an exception block which contains the information necessary for run-time exception handling.

In addition, we define two symbol aggregates, Function and Variable. These classes collect multiple symbols with the same address and type but different names; for example, weak and strong symbols for a single function.

2.2 Type Interface

The Type interface is responsible for parsing type information from the object file and handling the look-up and addition of new type information. Figure 2 shows the class inheritance diagram for the type interface. Class Type is the base class for all of the classes that are part of the interface. This class provides the basic common functionality for all the types, such as querying the name and size of a type. The rest of the classes represent specific types and provide more functionality based on the type.

Some of the types inherit from a second level of type classes, each representing a separate category of types.

fieldListType - This category of types represent the container types that contain a list of fields. Examples of this category include structure and the union types.

derivedType - This category of types represent types derived from a base type. Examples of this category include typedef, pointer and reference types.

rangedType - This category represents range types. Examples of this category include the array and the sub-range types.

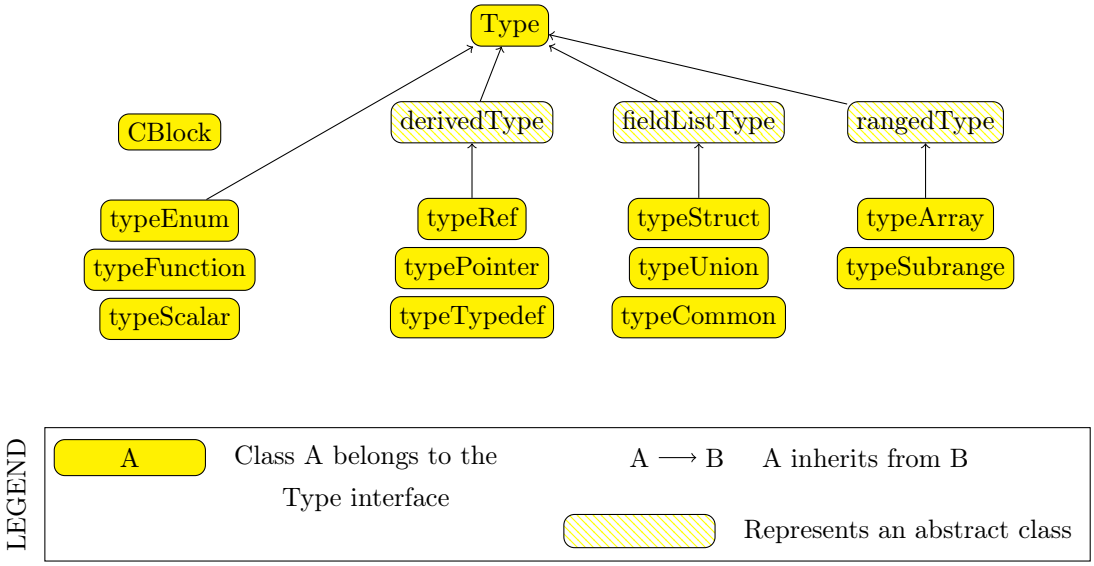


Figure 2: SymtabAPI Type Interface - Class Inheritance Diagram

The enum, function, common block and scalar types do not fall under any of the above category of types. Each of the specific types is derived from Type.

2.3 Line Number Interface

The Line Number interface is responsible for parsing line number information from the object file debug information and handling the look-up and addition of new line information. The main classes for this interface are LineInformation and LineNoTuple.

LineInformation - A LineInformation class object represents a mapping of line numbers to address range within a module (source file).

Statement/LineNoTuple - A Statement class object represents a location in source code with a source file, line number in that source file and start column in that line. For backwards compatibility, Statements may also be referred to as LineNoTuples.

2.4 Local Variable Interface

The Local Variable Interface is responsible for parsing local variable and parameter information of functions from the object file debug information and handling the look-up and addition of new add new local variables. All the local variables within a function are tied to the Symbol class object representing that function.

localVar - A localVar class object represents a local variable or a parameter belonging to a function.

2.5 Dynamic Address Translation

The AddressLookup class is a component for mapping between absolute addresses found in a running process and SymtabAPI objects. This is useful because libraries can load at different addresses in different processes. Each AddressLookup instance is associated with, and provides mapping for, one process.

3 Simple Examples

To illustrate the ideas in the API, this section presents several short examples that demonstrate how the API can be used. SymtabAPI has the ability to parse files that are on-disk or present in memory. The user program starts by requesting SymtabAPI to parse an object file. SymtabAPI returns a handle if the parsing succeeds, whcih can be used for further interactions with the SymtabAPI library. The following example shows how to parse a shared object file on disk.

```
using namespace Dyninst;
using namespace SymtabAPI;

//Name the object file to be parsed:
std::string file = "libfoo.so";

//Declare a pointer to an object of type Symtab; this represents the file.
Symtab *obj = NULL;

// Parse the object file
bool err = Symtab::openFile(obj, file);
```

Once the object file is parsed successfully and the handle is obtained, symbol look up and update operations can be performed in the following way:

```
using namespace Dyninst;
using namespace SymtabAPI;
std::vector <Symbol *> syms;
std::vector <Function *> funcs;

// search for a function with demangled (pretty) name "bar".
if (obj->findFunctionsByName(funcs, "bar")) {
       // Add a new (mangled) primary name to the first function
       funcs[0]->addMangledName("newname", true);
}

// search for symbol of any type with demangled (pretty) name "bar".
if (obj->findSymbol(syms, "bar", Symbol::ST_UNKNOWN)) {

    // change the type of the found symbol to type variable(ST_OBJECT)
    syms[0]->setType(Symbol::ST_OBJECT);

    // These changes are automatically added to symtabAPI; no further
    // actions are required by the user.
}
```

New symbols, functions, and variables can be created and added to the library at any point using the handle returned by successful parsing of the object file. When possible, add a function or variable rather than a symbol directly.

```
using namespace Dyninst;
using namespace SymtabAPI;

//Module for the symbol
Module *mod;

// obj represents a handle to a parsed object file.
// Lookup module handle for "DEFAULT_MODULE"
obj->findModuleByName(mod, "DEFAULT_MODULE");

// Create a new function symbol
Variable *newVar = mod->createVariable("newIntVar",  // Name of new variable
                                       0x12345,      // Offset from data section
                                       sizeof(int)); // Size of symbol
```

SymtabAPI gives the ability to query type information present in the object file. Also, new user defined types can be added to SymtabAPI. The following example shows both how to query type information after an object file is successfully parsed and also add a new structure type.

```
// create a new struct Type
// typedef struct{
//int field1,
//int field2[10]
// } struct1;

using namespace Dyninst;
using namespace SymtabAPI;

// Find a handle to the integer type; obj represents a handle to a parsed object file
Type *lookupType;
obj->findType(lookupType, "int");

// Convert the generic type object to the specific scalar type object
typeScalar *intType = lookupType->getScalarType();

// container to hold names and types of the new structure type
vector<pair<string, Type *> >fields;

//create a new array type(int type2[10])
typeArray *intArray = typeArray::create("intArray",intType,0,9, obj);

//types of the structure fields
fields.push_back(pair<string, Type *>("field1", intType));
fields.push_back(pair<string, Type *>("field2", intArray));

//create the structure type
typeStruct *struct1 = typeStruct::create("struct1", fields, obj);
```

Users can also query line number information present in an object file. The following example shows how to use SymtabAPI to get the address range for a line number within a source file.

```
using namespace Dyninst;
using namespace SymtabAPI;

// obj represents a handle to a parsed object file using symtabAPI
// Container to hold the address range
vector< pair< Offset, Offset > > ranges;

// Get the address range for the line 30 in source file foo.c
obj->getAddressRanges(ranges, "foo.c", 30);
```

Local variable information can be obtained using symtabAPI. You can query for a local variable within the entire object file or just within a function. The following example shows how to find local variable foo within function bar.

```
using namespace Dyninst;
using namespace SymtabAPI;

// Obj represents a handle to a parsed object file using symtabAPI
// Get the Symbol object representing function bar
vector<Symbol *> syms;
obj->findSymbol(syms, "bar", Symbol::ST_FUNCTION);

// Find the local var foo within function bar
vector<localVar *> *vars = syms[0]->findLocalVarible("foo");
```

The rest of this document describes the class hierarchy and the API in detail.

4 Definitions and Basic Types

The following definitions and basic types are referenced throughout the rest of this document.

4.1 Definitions

Offset Offsets represent an address relative to the start address(base) of the object file. For executables, the Offset represents an absolute address. The following definitions deal with the symbol table interface.

Object File An object file is the representation of code that a compiler or assembler generates by processing a source code file. It represents .o's, a.out's and shared libraries.

Region A region represents a contiguous area of the file that contains executable code or readable data; for example, an ELF section.

Symbol A symbol represents an entry in the symbol table, and may identify a function, variable or other structure within the file.

Function A function represents a code object within the file represented by one or more symbols.

Variable A variable represents a data object within the file represented by one or more symbols.

Module A module represents a particular source file in cases where multiple files were compiled into a single binary object; if this information is not present, or if the binary object is a shared library, we use a single default module.

Archive An archive represents a collection of binary objects stored in a single file (e.g., a static archive).

Relocations These provide the necessary information for inter-object references between two object files.

Exception Blocks These contain the information necessary for run-time exception handling The following definitions deal with members of the Symbol class.

Mangled Name A mangled name for a symbol provides a way of encoding additional information about a function, structure, class or another data type in a symbol name. It is a technique used to solve various problems caused by the need to resolve unique names for programming entities in many modern programming languages. For example, a function foo with signature void foo() has a mangled name $_Z8foov$ when compiled with gcc.

Pretty Name A pretty name for a symbol represents a user-level symbolic name for a symbol. For example, foo can be a pretty name for a function declared as void foo().

Typed Name A typed name for a symbol represents the user-level symbolic name complete with the signature. For example, void foo() can be a typed name for the function foo.

Symbol Linkage The symbol linkage for a symbol gives information on the visibility (binding) of this symbol, whether it is visible only in the object file where it is defined (local), if it is visible to all the object files that are being linked (global), or if its a weak alias to a global symbol.

Symbol Type Symbol type for a symbol represents the category of symbols to which it belongs. It can be a function symbol or a variable symbol or a module symbol. The following definitions deal with the type and the local variable interface.

Type A type represents the data type of a variable or a parameter. This can represent language pre-defined types (e.g. int, float), pre-defined types in the object (e.g., structures or unions), or user-defined types.

Local Variable A local variable represents a variable that has been declared within the scope of a sub-routine or a parameter to a sub-routine.

4.2 Basic Types

`typedef unsigned long Offset`

An integer value that contains an offset from base address of the object file.

`typedef int typeId_t`

A unique handle for identifying a type. Each of types is assigned a globally unique ID. This way it is easier to identify any data type of a variable or a parameter.

`typedef ... PID`

A handle for identifying a process that is used by the dynamic components of SymtabAPI. On UNIX platforms PID is a int, on Windows it is a HANDLE that refers to a process.

`typedef unsigned long Address`

An integer value that represents an address in a process. This is used by the dynamic components of SymtabAPI.

5 Namespace SymtabAPI

The classes described in the following sections are under the C++ namespace Dyninst::SymtabAPI. To access them a user should refer to them using the Dyninst:: and SymtabAPI:: prefixes, e.g. Dyninst::SymtabAPI::Type. Alternatively, a user can add the C++ using keyword above any references to SymtabAPI objects, e.g, using namespace Dyninst and using namespace SymtabAPI.

6 API Reference - Symbol Table Interface

This section describes the symbol table interface for the SymtabAPI library. Currently this interface has the following capabilities:

- Parsing the symbols in a binary, either on disk or in memory
- Querying for symbols
- Updating existing symbol information
- Adding new symbols
- Exporting symbols in standard formats
- Accessing relocation and exception information
- Accessing and modifying header information

The symbol table information is represented by the Symtab, Symbol, Archive, and Region classes. Module, Function, and Variable provide abstractions that support common use patterns. Finally, LocalVar represents function-local variables and parameters.

6.1 Class Symtab

Defined in: `Symtab.h`

The `Symtab` class represents an object file either on-disk or in-memory. This class is responsible for the parsing of the `Object` file information and holding the data that can be accessed through look up functions.

Method name	Return type	Method description
`file`	std::string	Full path to the opened file or provided name for the memory image.
`name`	std::string	File name without path.
`memberName`	std::string	For archive (.a) files, returns the object file (.o) this Symtab represents.
`getNumberOfRegions`	unsigned	Number of regions.
`getNumberOfSymbols`	unsigned	Total number of symbols in both the static and dynamic tables.
`mem_image`	char *	Pointer to memory image for the Symtab; not valid for disk files.
`imageOffset`	Offset	Offset of the first code segment from the start of the binary.
`dataOffset`	Offset	Offset of the first data segment from the start of the binary.
`imageLength`	Offset	Size of the primary code-containing region, typically .text.
`dataLength`	Offset	Size of the primary data-containing region, typically .data.
`isStaticBinary`	Offset	True if the binary was compiled statically.
`isExec`	bool	True if the file is an executable, false if it is a shared library or object file.
`isStripped`	bool	True if the file was stripped of symbol table information.
`getAddressWidth`	unsigned	Size (in bytes) of a pointer value in the Symtab; 4 for 32-bit binaries and 8 for 64-bit binaries.
`getArchitecture`	Architecture	Representation of the system architecture for the binary.
`getLoadOffset`	Offset	The suggested load offset of the file; typically 0 for shared libraries.
`getEntryOffset`	Offset	The entry point (where execution begins) of the binary.
`getBaseOffset`	Offset	(Windows only) the OS-specified base offset of the file.

```
ObjectType getObjectType() const
```

This method queries information on the type of the object file.

6.1.1 File opening/parsing

```
static bool openFile(Symtab *&obj,
                     string filename)
```

Creates a new `Symtab` object for an object file on disk. This object serves as a handle to the parsed object file. `filename` represents the name of the `Object` file to be parsed. The `Symtab` object is returned in `obj` if the parsing succeeds. Returns `true` if the file is parsed without an error, else returns `false`. `getLastSymtabError()` and `printError()` should be called to get more error details.

```
static bool openFile(Symtab *&obj,
                     char *mem_image,
                     size_t size,
                     std::string name)
```

This factory method creates a new `Symtab` object for an object file in memory. This object serves as a handle to the parsed object file. `mem_image` represents the pointer to the `Object` file in memory to be parsed. `size` indicates the size of the image. `name` specifies the name we will give to the parsed object. The `Symtab` object is returned in `obj` if the parsing succeeds. Returns `true` if the file is parsed without an error, else returns `false`. `getLastSymtabError()` and `printError()` should be called to get more error details.

```
static Symtab *findOpenSymtab(string name)
```

Find a previously opened `Symtab` that matches the provided name.

6.1.2 Module lookup

```
Module *getDefaultModule()
```

Returns the default module, a collection of all functions, variables, and symbols that do not have an explicit module specified.

```
bool findModuleByName(Module *&ret,
                      const string name)
```

This method searches for a module with name `name`. If the module exists returns `true` with `ret` set to the module handle; otherwise returns `false` with `ret` set to `NULL`.

```
bool findModuleByOffset(Module *&ret,
                        Offset offset)
```

This method searches for a module that starts at offset `offset`. If the module exists returns `true` with `ret` set to the module handle; otherwise returns `false` with `ret` set to `NULL`.

```
bool getAllModules(vector<module *> &ret)
```

This method returns all modules in the object file. Returns `true` on success and `false` if there are no modules. The error value is set to `No_Such_Module`.

6.1.3 Function, Variable, and Symbol lookup

```
bool findFuncByEntryOffset(Function *&ret,
                           const Offset offset)
```

This method returns the `Function` object that begins at `offset`. Returns `true` on success and `false` if there is no matching function. The error value is set to `No_Such_Function`.

```
bool findFunctionsByName(std::vector<Function *> &ret,
                         const std::string name,
                         NameType nameType = anyName,
                         bool isRegex = false,
                         bool checkCase = true)
```

This method finds and returns a vector of `Function`s whose names match the given pattern. The `nameType` parameter determines which names are searched: mangled, pretty, typed, or any. If the `isRegex` flag is set a regular expression match is performed with the symbol names. `checkCase` is applicable only if `isRegex` has been set. This indicates if the case be considered while performing regular expression matching. `ret` contains the list of matching Functions, if any. Returns `true` if it finds functions that match the given name, otherwise returns `false`. The error value is set to `No_Such_Function`.

```
bool getContainingFunction(Offset offset,
                           Function *&ret)
```

This method returns the function, if any, that contains the provided `offset`. Returns `true` on success and `false` on failure. The error value is set to `No_Such_Function`. Note that this method does not parse, and therefore relies on the symbol table for information. As a result it may return incorrect information if the symbol table is wrong or if functions are either non-contiguous or overlapping. For more precision, use the ParseAPI library.

```
bool getAllFunctions(vector<Function *> &ret)
```

This method returns all functions in the object file. Returns `true` on success and `false` if there are no modules. The error value is set to `No_Such_Function`.

```
bool findVariableByOffset(Variable *&ret,
                          const Offset offset)
```

This method returns the `Variable` object at offset. Returns `true` on success and `false` if there is no matching variable. The error value is set to `No_Such_Variable`.

```
bool findVariablesByName(std::vector<Variable *> &ret,
                         const std::string name,
                         NameType nameType = anyName,
                         bool isRegex = false,
                         bool checkCase = true)
```

This method finds and returns a vector of `Variable`s whose names match the given pattern. The `nameType` parameter determines which names are searched: mangled, pretty, typed, or any (note: a `Variable` may not have a typed name). If the `isRegex` flag is set a regular expression match is performed with the symbol names. `checkCase` is applicable only if `isRegex` has been set. This indicates if the case be considered while performing regular expression matching. `ret` contains the list of matching `Variable`s, if any. Returns `true` if it finds variables that match the given name, otherwise returns `false`. The error value is set to `No_Such_Variable`.

```
bool getAllVariables(vector<Variable *> &ret)
```

This method returns all variables in the object file. Returns `true` on success and `false` if there are no modules. The error value is set to `No_Such_Variable`.

```
bool findSymbol(vector <Symbol *> &ret,
                const string name,
                Symbol::SymbolType sType,
                NameType nameType = anyName,
                bool isRegex = false,
                bool checkCase = false)
```

This method finds and returns a vector of symbols with type `sType` whose names match the given name. The `nameType` parameter determines which names are searched: mangled, pretty, typed, or any. If the `isRegex` flag is set a regular expression match is performed with the symbol names. `checkCase` is applicable only if `isRegex` has been set. This indicates if the case be considered while performing regular expression matching. `ret` contains the list of matched symbols if any. Returns `true` if it finds symbols with the given attributes. or else returns `false`. The error value is set `to` `No_Such_Function` / `No_Such_Variable`/ `No_Such_Module`/ `No_Such_Symbol` based on the type.

```
const vector<Symbol *> *findSymbolByOffset(Offset offset)
```

Return a pointer to a vector of `Symbol`s with the specified offset. The pointer belongs to `Symtab` and should not be modified or freed.

```
bool getAllSymbols(vector<Symbol *> &ret)
```

This method returns all symbols. Returns `true` on success and `false` if there are no symbols. The error value is set to `No_Such_Symbol`.

```
bool getAllSymbolsByType(vector<Symbol *> &ret,
                         Symbol::SymbolType sType)
```

This method returns all symbols whose type matches the given type `sType`. Returns `true` on success and `false` if there are no symbols with the given type. The error value is set to `No_Such_Symbol`.

```
bool getAllUndefinedSymbols(std::vector<Symbol *> &ret)
```

This method returns all symbols that reference symbols in other files (e.g., external functions or variables). Returns `true` if there is at least one such symbol or else returns `false` with the error set to `No_Such_Symbol`.

6.1.4 Region lookup

```
bool getCodeRegions(std::vector<Region *>&ret)
```

This method finds all the code regions in the object file. Returns `true` with `ret` containing the code regions if there is at least one code region in the object file or else returns `false`.

```
bool getDataRegions(std::vector<Region *>&ret)
```

This method finds all the data regions in the object file. Returns `true` with `ret` containing the data regions if there is at least one data region in the object file or else returns `false`.

```
bool getMappedRegions(std::vector<Region *>&ret)
```

This method finds all the loadable regions in the object file. Returns `true` with `ret` containing the loadable regions if there is at least one loadable region in the object file or else returns `false`.

```
bool getAllRegions(std::vector<Region *>&ret)
```

This method retrieves all the regions in the object file. Returns `true` with `ret` containing the regions.

```
bool getAllNewRegions(std::vector<Region *>&ret)
```

This method finds all the new regions added to the object file. Returns `true` with `ret` containing the regions if there is at least one new region that is added to the object file or else returns `false`.

```
bool findRegion(Region *&reg,
                string sname)
```

Find a region (ELF section) wih name `sname` in the binary. Returns `true` if found, with `reg` set to the region pointer. Otherwise returns `false` with `reg` set to `NULL`.

```
bool findRegion(Region *&reg,
                const Offset addr,
                const unsigned long size)
```

Find a region (ELF section) with a memory offset of `addr` and memory size of `size`. Returns `true` if found, with `reg` set to the region pointer. Otherwise returns `false` with `reg` set to `NULL`.

```
bool findRegionByEntry(Region *&reg,
                       const Offset soff)
```

Find a region (ELF section) with a memory offset of `addr`. Returns `true` if found, with `reg` set to the region pointer. Otherwise returns `false` with `reg` set to `NULL`.

```
Region *findEnclosingRegion(const Offset offset)
```

Find the region (ELF section) whose virtual address range contains `offset`. Returns the region if found; otherwise returns `NULL`.

6.1.5 Insertion and modification

```
bool emit(string file)
```

Creates a new file using the specified name that contains all changes made by the user.

```
bool addLibraryPrereq(string lib)
```

Add a library dependence to the file such that when the file is loaded, the library will be loaded as well. Cannot be used for static binaries.

```
Function *createFunction(std::string name,
                         Offset offset,
                         size_t size,
                         Module *mod = NULL)
```

This method creates a `Function` and updates all necessary data structures (including creating Symbols, if necessary). The function has the provided mangled name, offset, and size, and is added to the Module `mod`. Symbols representing the function are added to the static and dynamic symbol tables. Returns the pointer to the new `Function` on success or `NULL` on failure.

```
Variable *createVariable(std::string name,
                         Offset offset,
                         size_t size,
                         Module *mod = NULL)
```

This method creates a `Variable` and updates all necessary data structures (including creating Symbols, if necessary). The variable has the provided mangled name, offset, and size, and is added to the Module `mod`. Symbols representing the variable are added to the static and dynamic symbol tables. Returns the pointer to the new `Variable` on success or `NULL` on failure.

```
bool addSymbol(Symbol *newsym)
```

This method adds a new symbol `newsym` to all of the internal data structures. The primary name of the `newsym` must be a mangled name. Returns `true` on success and `false` on failure. A new copy of `newsym` is not made. `newsym` must not be deallocated after adding it to symtabAPI. We suggest using `createFunction` or `createVariable` when possible.

```
bool addSymbol(Symbol *newsym,
               Symbol *referringSymbol)
```

This method adds a new dynamic symbol `newsym` which refers to `referringSymbol` to all of the internal data structures. `newsym` must represent a dynamic symbol. The primary name of the newsym must be a mangled name. All the required version names are allocated automatically. Also if the `referringSymbol` belongs to a shared library which is not currently a dependency, the shared library is added to the list of dependencies implicitly. Returns `true` on success and `false` on failure. A new copy of `newsym` is not made. `newsym` must not be deallocated after adding it to symtabAPI.

```
bool deleteFunction(Function *func)
```

This method deletes the `Function` `func` from all of symtab's data structures. It will not be available for further queries. Return `true` on success and `false` if `func` is not owned by the `Symtab`.

```
bool deleteVariable(Variable *var)
```

This method deletes the variable `var` from all of symtab's data structures. It will not be available for further queries. Return `true` on success and `false` if `var` is not owned by the `Symtab`.

```
bool deleteSymbol(Symbol *sym)
```

This method deletes the symbol `sym` from all of symtab's data structures. It will not be available for further queries. Return `true` on success and `false` if func is not owned by the `Symtab`.

```
bool addRegion(Offset vaddr,
               void *data,
               unsigned int dataSize,
               std::string name,
               Region::RegionType rType_,
               bool loadable = false,
               unsigned long memAlign = sizeof(unsigned),
               bool tls = false)
```

Creates a new region using the specified parameters and adds it to the file.

```
Offset getFreeOffset(unsigned size)
```

Find a contiguous region of unused space within the file (which may be at the end of the file) of the specified size and return an offset to the start of the region. Useful for allocating new regions.

```
bool addRegion(Region *newreg);
```

Adds the provided region to the file.

6.1.6 Catch and Exception block lookup

```
bool getAllExceptions(vector<ExceptionBlock *> &exceptions)
```

This method retrieves all the exception blocks in the `Object` file. Returns `false` if there are no exception blocks else returns `true` with exceptions containing a vector of `ExceptionBlock`s.

```
bool findException(ExceptionBlock &excp,
                   Offset addr)
```

This method returns the exception block in the binary at the offset `addr`. Returns `false` if there is no exception block at the given offset else returns `true` with `excp` containing the exception block.

```
bool findCatchBlock(ExceptionBlock &excp,
                    Offset addr,
                    unsigned size = 0)
```

This method returns `true` if the address range `[addr, addr+size]` contains a catch block, with `excp` pointing to the appropriate block, else returns `false`.

6.1.7 Symtab information

```
typedef enum {
    obj_Unknown,
    obj_SharedLib,
    obj_Executable,
    obj_RelocatableFile,
} ObjectType;

bool isCode(const Offset where) const
```

This method checks if the given offset `where` belongs to the text section. Returns `true` if that is the case or else returns `false`.

```
bool isData(const Offset where) const
```

This method checks if the given offset `where` belongs to the data section. Returns `true` if that is the case or else returns `false`.

```
bool isValidOffset(const Offset where) const
```

This method checks if the given offset `where` is valid. For an offset to be valid it should be aligned and it should be a valid code offset or a valid data offset. Returns `true` if it succeeds or else returns `false`.

6.1.8 Line number information

```
bool getAddressRanges(vector<pair<Offset, Offset> > & ranges,
                      string lineSource,
                      unsigned int LineNo)
```

This method returns the address ranges in `ranges` corresponding to the line with line number `lineNo` in the source file `lineSource`. Searches all modules for the given source. Return `true` if at least one address range corresponding to the line number was found and returns `false` if none found.

```
bool getSourceLines(vector<LineNoTuple> &lines,
                    Offset addressInRange)
```

This method returns the source file names and line numbers corresponding to the given address `addressInRange`. Searches all modules for the given source. Return `true` if at least one tuple corresponding to the offset was found and returns `false` if none found.

```
bool addLine(string lineSource,
             unsigned int lineNo,
             unsigned int lineOffset,
             Offset lowInclusiveAddr,
             Offset highExclusiveAddr)
```

This method adds a new line to the line map. `lineSource` represents the source file name. `lineNo` represents the line number. Returns `true` on success and `false` on error.

```
bool addAddressRange(Offset lowInclusiveAddr,
                     Offset highExclusiveAddr,
                     string lineSource,
                     unsigned int lineNo,
                     unsigned int lineOffset = 0);
```

This method adds an address range `[lowInclusiveAddr, highExclusiveAddr)` for the line with line number `lineNo` in source file `lineSource` at offset `lineOffset`. Returns `true` on success and `false` on error.

6.1.9 Type information

```
void parseTypesNow()
```

Forces SymtabAPI to perform type parsing instead of delaying it to when needed.

```
bool findType(Type *&type,
              string name)
```

Performs a look up among all the built-in types, standard types and user-defined types and returns a handle to the found type with name `name`. Returns `true` if a type is found with type containing the handle to the type, else return `false`.

```
bool addType(Type * type)
```

Adds a new type `type` to symtabAPI. Return `true` on success.

```
static std::vector<Type *> * getAllstdTypes()
```

Returns all the standard types that normally occur in a program.

```
static std::vector<Type *> * getAllbuiltInTypes()
```

Returns all the bulit-in types defined in the binary.

```
bool findLocalVariable(vector<localVar *> &vars,
                       string name)
```

The method returns a list of local variables named name within the object file. Returns `true` with `vars` containing a list of `localVar` objects corresponding to the local variables if found or else returns `false`.

```
bool findVariableType(Type *&type,
                      std::string name)
```

This method looks up a global variable with name `name` and returns its type attribute. Returns `true` if a variable is found or returns `false` with type set to `NULL`.

```
typedef enum ... SymtabError
```

`SymtabError` can take one of the following values.

SymtabError enum	Meaning
Obj_Parsing	An error occurred during object parsing(internal error).
Syms_To_Functions	An error occurred in converting symbols to functions(internal error).
Build_Function_Lists	An error occurred while building function lists(internal error).
No_Such_Function	No matching function exists with the given inputs.
No_Such_Variable	No matching variable exists with the given inputs.
No_Such_Module	No matching module exists with the given inputs.
No_Such_Symbol	No matching symbol exists with the given inputs.
No_Such_Region	No matching region exists with the given inputs.
No_Such_Member	No matching member exists in the archive with the given inputs.
Not_A_File	Binary to be parsed may be an archive and not a file.
Not_An_Archive	Binary to be parsed is not an archive.
Duplicate_Symbol	Duplicate symbol found in symbol table.
Export_Error	Error occurred during export of modified symbol table.
Emit_Error	Error occurred during generation of modified binary.
Invalid_Flags	Flags passed are invalid.
Bad_Frame_Data	Stack walking DWARF information has bad frame data.
No_Frame_Entry	No stack walking frame data found in debug information for this location.
Frame_Read_Error	Failed to read stack frame data.
Multiple_Region_Matches	Multiple regions match the provided data.
No_Error	Previous operation did not result in failure.

```
static SymtabError getLastSymtabError()
```

This method returns an error value for the previously performed operation that resulted in a failure. SymtabAPI sets a global error value in case of error during any operation. This call returns the last error that occurred while performing any operation.

```
static string printError(SymtabError serr)
```

This method returns a detailed description of the enum value serr in human readable format.

6.2 Class Module

This class represents the concept of a single source file. Currently,Modules are only identified for the executable file; each shared library is made up of a single Module, ignoring any source file information that may be present. We also create a single module, called `DEFAULT_MODULE`, for each Symtab that contains any symbols for which module information was unavailable. This may be compiler template code, or files produced without module information.

supportedLanguages	Meaning
lang_Unknown	Unknown source language
lang_Assembly	Raw assembly code
lang_C	C source code
lang_CPlusPlus	C++ source code
lang_GnuCPlusPlus	C++ with GNU extensions
lang_Fortran	Fortran source code
lang_Fortran_with_pretty_debug	Fortran with debug annotations
lang_CMFortran	Fortran with CM extensions

Method name	Return type	Method description
isShared	bool	True if the module is for a shared library, false for an executable.
fullName	std::string &	Name, including path, of the source file represented by the module.
fileName	std::string &	Name, not including path, of the source file represented by the module.
language	supportedLanguages	The source language used by the Module.
addr	Offset	Offset of the start of the module, as reported by the symbol table, assuming contiguous modules.
exec	Symtab *	Symtab object that contains the module.
hasLineInformation	bool	True if the module has line information.

6.2.1 Function, Variable, Symbol lookup

```
bool findFunctionByEntryOffset(Function *&ret,
                               const Offset offset)
```

This method returns the `Function` object that begins at `offset`. Returns `true` on success and `false` if there is no matching function. The error value is set to `No_Such_Function`.

```
typedef enum {
    mangledName,
    prettyName,
    typedName,
    anyName
} NameType;
```

```
bool findFunctionsByName(vector<Function> &ret,
                         const string name,
                         Symtab::NameType nameType = anyName,
                         bool isRegex = false,
                         bool checkCase = true)
```

This method finds and returns a vector of `Functions` whose names match the given pattern. The `nameType` parameter determines which names are searched: mangled, pretty, typed, or any. If the `isRegex` flag is set a regular expression match is performed with the symbol names. `checkCase` is applicable only if `isRegex` has been set. This indicates if the case be considered while performing regular expression matching. `ret` contains the list of matching `Functions`, if any. Returns `true` if it finds functions that match the given name, otherwise returns `false`. The error value is set to `No_Such_Function`.

```
bool getAllFunctions(vector<Function *> &ret)
```

This method returns all functions in the object file. Returns `true` on success and `false` if there are no modules. The error value is set to `No_Such_Function`.

```
bool findVariableByOffset(Variable *&ret,
                          const Offset offset)
```

This method returns the `Variable` object at `offset`. Returns `true` on success and `false` if there is no matching variable. The error value is set to `No_Such_Variable`.

```
bool findVariablesByName(vector<Function> &ret,
                         const string &name,
                         Symtab::NameType nameType,
                         bool isRegex = false,
                         bool checkCase = true)
```

This method finds and returns a vector of `Variables` whose names match the given pattern. The `nameType` parameter determines which names are searched: mangled, pretty, typed, or any (note: a `Variable` may not have a typed name). If the `isRegex` flag is set a regular expression match is performed with the symbol names. `checkCase` is applicable only if `isRegex` has been set. This indicates if the case be considered while performing regular expression matching. `ret` contains the list of matching `Variables`, if any. Returns `true` if it finds variables that match the given name, otherwise returns `false`. The error value is set to `No_Such_Variable`.

```
bool getAllVariables(vector<Variable *> &ret)
```

This method returns all variables in the object file. Returns `true` on success and `false` if there are no modules. The error value is set to `No_Such_Variable`.

```
bool getAllSymbols(vector<Symbol *> &ret)
```

This method returns all symbols. Returns `true` on success and `false` if there are no symbols. The error value is set to `No_Such_Symbol`.

```
bool getAllSymbolsByType(vector<Symbol *> &ret,
                         Symbol::SymbolType sType)
```

This method returns all symbols whose type matches the given type `sType`. Returns `true` on success and `false` if there are no symbols with the given type. The error value is set to `No_Such_Symbol`.

6.2.2 Line number information

```
bool getAddressRanges(vector<pair<unsigned long, unsigned long> > & ranges,
                      string lineSource, unsigned int lineNo)
```

This method returns the address ranges in `ranges` corresponding to the line with line number `lineNo` in the source file `lineSource`. Searches only this module for the given source. Return `true` if at least one address range corresponding to the line number was found and returns false if none found.

```
bool getSourceLines(vector<Statement *> &lines,
                    Offset addressInRange)
```

This method returns the source file names and line numbers corresponding to the given address `addressInRange`. Searches only this module for the given source. Return `true` if at least one tuple corresponding to the offset was found and returns `false` if none found. The `Statement` class used to be named `LineNoTuple`; backwards compatibility is provided via typedef.

```
LineInformation *getLineInformation() const
```

This method returns the line map (section 7.1) corresponding to the module. Returns `NULL` if there is no line information existing for the module.

```
bool getStatements(std::vector<Statement *> &statements)
```

Returns all line information (section 7.2) available for the module.

6.2.3 Type information

```
bool findType(Type * &type,
              string name)
```

This method performs a look up and returns a handle to the named `type`. This method searches all the built-in types, standard types and user-defined types within the module. Returns `true` if a type is found with type containing the handle to the type, else return `false`.

```
bool findLocalVariable(vector<localVar *> &vars,
                       string name)
```

The method returns a list of local variables within the module with name `name`. Returns `true` with vars containing a list of `localVar` objects corresponding to the local variables if found or else returns `false`.

```
bool findVariableType(Type *&type,
                      std::string name)
```

This method looks up a global variable with name `name` and returns its type attribute. Returns `true` if a variable is found or returns `false` with `type` set to `NULL`.

6.3 Class FunctionBase

The `FunctionBase` class provides a common interface that can represent either a regular function or an inlined function.

Method name	Return type	Method description
getModule	const Module *	Module this function belongs to.
getSize	unsigned	Size encoded in the symbol table; may not be actual function size.
getRegion	Region *	Region containing this function.
getReturnType	Type *	Type representing the return type of the function.
getName	std::string	Returns primary name of the function (first mangled name or DWARF name)

```
bool setModule (Module *module)
```

This function changes the module to which the function belongs to `module`. Returns `true` if it succeeds.

```
bool setSize (unsigned size)
```

This function changes the size of the function to `size`. Returns `true` if it succeeds.

```
bool setOffset (Offset offset)
```

The method changes the offset of the function to `offset`. Returns `true` if it succeeds.

```
bool addMangledName(string name,
                    bool isPrimary)
```

This method adds a mangled name `name` to the function. If `isPrimary` is `true` then it becomes the default name for the function. This method returns `true` on success and `false` on failure.

```
bool addPrettyName(string name,
                   bool isPrimary)
```

This method adds a pretty name `name` to the function. If `isPrimary` is `true` then it becomes the default name for the function. This method returns `true` on success and `false` on failure.

```
bool addTypedName(string name,
                  bool isPrimary)
```

This method adds a typed name `name` to the function. If `isPrimary` is `true` then it becomes the default name for the function. This method returns `true` on success and `false` on failure.

```
bool getLocalVariables(vector<localVar *> &vars)
```

This method returns the local variables in the function. `vars` contains the list of variables found. If there is no debugging information present then it returns `false` with the error code set to `NO_DEBUG_INFO` accordingly. Otherwise it returns `true`.

```
std::vector<VariableLocation> &getFramePtr()
```

This method returns a list of frame pointer offsets (abstract top of the stack) for the function. See the `VariableLocation` class description for more information.

```
bool getParams(vector<localVar *> &params)
```

This method returns the parameters to the function. `params` contains the list of parameters. If there is no debugging information present then it returns `false` with the error code set to `NO_DEBUG_INFO` accordingly. Returns `true` on success.

```
bool findLocalVariable(vector<localVar *> &vars,
                       string name)
```

This method returns a list of local variables within a function that have name `name`. `vars` contains the list of variables found. Returns `true` on success and `false` on failure.

```
bool setReturnType(Type *type)
```

Sets the return type of a function to `type`.

```
FunctionBase* getInlinedParent()
```

Gets the function that this function is inlined into, if any. Returns `NULL` if there is no parent.

```
const InlineCollection& getInlines()
```

Gets the set of functions inlined into this one (possibly empty).

6.4 Class Function

The `Function` class represents a collection of symbols that have the same address and a type of `ST_FUNCTION`. When appropriate, use this representation instead of the underlying `Symbol` objects.

Method name	Return type	Method description
getModule	const Module *	Module this function belongs to.
getOffset	Offset	Offset in the file associated with the function.
getSize	unsigned	Size encoded in the symbol table; may not be actual function size.
mangled_names_begin	Aggregate::name_iter	Beginning of a range of unique names of symbols pointing to this function.
mangled_names_end	Aggregate::name_iter	End of a range of unique names of symbols pointing to this function.
pretty_names_begin	Aggregate::name_iter	As above, but prettified with the demangler.
pretty_names_end	Aggregate::name_iter	As above, but prettified with the demangler.
typed_names_begin	Aggregate::name_iter	As above, but including full type strings.
typed_names_end	Aggregate::name_iter	As above, but including full type strings.
getRegion	Region *	Region containing this function.
getReturnType	Type *	Type representing the return type of the function.

```
bool getSymbols(vector<Symbol *> &syms) const
```

This method returns the vector of `Symbol`s that refer to the function.

```
bool setModule (Module *module)
```

This function changes the module to which the function belongs to `module`. Returns `true` if it succeeds.

```
bool setSize (unsigned size)
```

This function changes the size of the function to `size`. Returns `true` if it succeeds.

```
bool setOffset (Offset offset)
```

The method changes the offset of the function to `offset`. Returns `true` if it succeeds.

```
bool addMangledName(string name,
                    bool isPrimary)
```

This method adds a mangled name `name` to the function. If `isPrimary` is `true` then it becomes the default name for the function. This method returns `true` on success and `false` on failure.

```
bool addPrettyName(string name,
                   bool isPrimary)
```

This method adds a pretty name `name` to the function. If `isPrimary` is `true` then it becomes the default name for the function. This method returns `true` on success and `false` on failure.

```
bool addTypedName(string name,
                  bool isPrimary)
```

This method adds a typed name `name` to the function. If `isPrimary` is `true` then it becomes the default name for the function. This method returns `true` on success and `false` on failure.

```
bool getLocalVariables(vector<localVar *> &vars)
```

This method returns the local variables in the function. `vars` contains the list of variables found. If there is no debugging information present then it returns `false` with the error code set to `NO_DEBUG_INFO` accordingly. Otherwise it returns `true`.

```
std::vector<VariableLocation> &getFramePtr()
```

This method returns a list of frame pointer offsets (abstract top of the stack) for the function. See the `VariableLocation` class description for more information.

```
bool getParams(vector<localVar *> &params)
```

This method returns the parameters to the function. `params` contains the list of parameters. If there is no debugging information present then it returns `false` with the error code set to `NO_DEBUG_INFO` accordingly. Returns `true` on success.

```
bool findLocalVariable(vector<localVar *> &vars,
                       string name)
```

This method returns a list of local variables within a function that have name `name`. `vars` contains the list of variables found. Returns `true` on success and `false` on failure.

```
bool setReturnType(Type *type)
```

Sets the return type of a function to `type`.

6.5 Class InlinedFunction

The `InlinedFunction` class represents an inlined function, as found in DWARF information. Its interface is almost entirely inherited from `FunctionBase`.

```
std::pair<std::string, Dyninst::Offset> getCallsite()
```

Returns the file and line corresponding to the call site of an inlined function.

6.6 Class Variable

The `Variable` class represents a collection of symbols that have the same address and represent data.

Method name	Return type	Method description
getOffset	Offset	Offset associated with this variable.
getSize	unsigned	Size of this variable in the symbol table.
mangled_names_begin	Aggregate::name_iter	Beginning of a range of unique names of symbols pointing to this variable.
mangled_names_end	Aggregate::name_iter	End of a range of unique names of symbols pointing to this variable.
getType	Type *	Type of this variable, if known.
getModule	const Module *	Module that contains this variable.
getRegion	Region *	Region that contains this variable.

```
bool getSymbols(vector<Symbol *> &syms) const
```

This method returns the vector of `Symbol`s that refer to the variable.

```
bool setModule (Module *module)
```

This method changes the module to which the variable belongs. Returns `true` if it succeeds.

```
bool setSize (unsigned size)
```

This method changes the size of the variable to `size`. Returns `true` if it succeeds.

```
bool setOffset (Offset offset)
```

The method changes the offset of the variable. Returns `true` if it succeeds.

```
bool addMangledName(string name,
                    bool isPrimary)
```

This method adds a mangled name `name` to the variable. If `isPrimary` is `true` then it becomes the default name for the variable. This method returns `true` on success and `false` on failure.

```
bool addPrettyName(string name,
                   bool isPrimary)
```

This method adds a pretty name `name` to the variable. If `isPrimary` is `true` then it becomes the default name for the variable. This method returns `true` on success and `false` on failure.

```
bool addTypedName(string name,
                  bool isPrimary)
```

This method adds a typed name `name` to the variable. If `isPrimary` is `true` then it becomes the default name for the variable. This method returns `true` on success and `false` on failure.

```
bool setType(Type *type)
```

Sets the type of the variable to `type`.

6.7 Class Symbol

The `Symbol` class represents a symbol in the object file. This class holds the symbol information such as the mangled, pretty and typed names, the module in which it is present, type, linkage, offset and size.

SymbolType	Meaning
ST_UNKNOWN	Unknown type
ST_FUNCTION	Function or other executable code sequence
ST_OBJECT	Variable or other data object
ST_MODULE	Source file declaration
ST_SETION	Region declaration
ST_TLS	Thread-local storage declaration
ST_DELETED	Deleted symbol
ST_NOTYPE	Miscellaneous symbol

SymbolLinkage	Meaning
SL_UNKNOWN	Unknown linkage
SL_GLOBAL	Process-global symbol
SL_LOCAL	Process-local (e.g., static) symbol
SL_WEAK	Alternate name for a function or variable

The following two types are platform-specific:

```
typedef enum {
    SV_UNKNOWN,
    SV_DEFAULT,
    SV_INTERNAL,
    SV_HIDDEN,
    SV_PROTECTED
} SymbolVisibility;
```

```
typedef enum {
    TAG_UNKNOWN,
    TAG_USER,
    TAG_LIBRARY,
    TAG_INTERNAL
} SymbolTag;
```

Method name	Return type	Method description
getMangledName	string	Raw name of the symbol in the symbol table, including name mangling.
getPrettyName	string	Demangled name of the symbol with parameters (for functions) removed.
getTypedName	string	Demangled name of the symbol including full function parameters.
getModule	Module *	The module, if any, that contains the symbol.
getType	SymbolType	The symbol type (as defined above) of the symbol.
getLinkage	SymbolLinkage	The linkage (as defined above) of the symbol.
getVisibility	SymbolVisibility	The visibility (as defined above) of the symbol.
tag	SymbolTag	The tag (as defined above) of the symbol.
getOffset	Offset	The offset of the object the symbols refers to.
getSize	unsigned	The size of the object the symbol refers to.
getRegion	Region *	The region containing the symbol.
getIndex	int	The index of the symbol within the symbol table.
getStrIndex	int	The index of the symbol name in the string table.
isInDynSymtab	bool	If true, the symbol is dynamic and can be used as the target of an intermodule reference. Implies isInSymtab is false.
isInSymtab	bool	If true, the symbol is static. Implies isInDynSymtab is false.
isAbsolute	bool	If true, the offset encoded in the symbol is an absolute value rather than an offset.
isFunction	bool	If true, the symbol refers to a function.
getFunction	Function *	The Function that contains this symbol if such a Function exists.
isVariable	bool	If true, the symbol refers to a variable.
getVariable	Variable *	The Variable that contains this symbol if such a Variable exists.
getSymtab	Symtab *	The Symtab that contains this symbol.
getPtrOffset	Offset	For binaries with an OPD section, the offset in the OPD that contains the function pointer data structure for this symbol.
getLocalTOC	Offset	For platforms with a TOC register, the expected TOC for the object referred to by this symbol.
isCommonStorage	bool	True if the symbol represents a common section (Fortran).

```
SYMTAB_EXPORT Symbol(const std::string& name,
                     SymbolType type,
                     SymbolLinkage linkage,
                     SymbolVisibility visibility,
                     Offset offset,
                     Module *module = NULL,
                     Region *region = NULL,
                     unsigned size = 0,
                     bool dyamic = false,
                     bool absolute = false,
                     int index = -1,
                     int strindex = -1,
```

```
                    bool commonStorage = false)
```

Symbol creation interface:

name The mangled name of the symbol.

type The type of the symbol as specified above.

linkage The linkage of the symbol as specified above.

visibility The visibility of the symbol as specified above.

offset The offset within the file that the symbol refers to.

module The source code module the symbol should belong to; default is no module.

region The region the symbol belongs to; if left unset this will be determined if a new binary is generated.

size The size of the object the symbol refers to; defaults to 0.

dynamic If true, the symbol belongs to the dynamic symbol table (ELF) and may be used as the target of inter-module references.

absolute If true, the offset specified is treated as an absolute value rather than an offset.

index The index in the symbol table. If left unset, it will be determined when generating a new binary.

strindex The index in the string table that contains the symbol name. If left unset, it will be determined when generating a new binary.

commonStorage If true, the symbol references common storage (Fortran).

```
bool getVersionFileName(std::string &fileName)
```

This method retrieves the file name in which this symbol is present. Returns `false` if this symbol does not have any version information present otherwise returns `true`.

```
bool getVersions(std::vector<std::string> *&vers)
```

This method retrieves all the version names for this symbol. Returns `false` if the symbol does not have any version information present.

```
bool getVersionNum(unsigned &verNum)
```

This method retrieves the version number of the symbol. Returns `false` if the symbol does not have any version information present.

6.7.1 Symbol modification

Most elements of a `Symbol` can be modified using the functions below. Each returns `true` on success and `false` otherwise.

```
bool setSize (unsigned size)
bool setOffset (Offset newOffset)
bool setMangledName (string name)
bool setType (SymbolType sType)
bool setModule (Module *module)
bool setRegion (Region *region)
bool setDynamic (bool dyn)
bool setAbsolute (bool absolute)
bool setCommonStorage (bool common)
bool setFunction (Function *func)
bool setVariable (Variable *var)
bool setIndex (int index)
bool setStrIndex (int index)
bool setPtrOffset (Offset ptr)
bool setLocalTOC (Offset toc)
bool setVersionNum (unsigned num)
bool setVersionFileName (std::string &fileName)
bool setVersions (std::vector<std::string> &vers)
```

6.8 Class Archive

This is used only on ELF platforms. This class represents an archive. This class has information of all the members in the archives.

```
static bool openArchive(Archive *&img,
                        string name)
```

This factory method creates a new `Archive` object for an archive file on disk. This object serves as a handle to the parsed archive file. `name` represents the name of the archive to be parsed. The `Archive` object is returned in `img` if the parsing succeeds. This method returns `false` if the given file is not an archive. The error is set to `Not_An_Archive`. This returns `true` if the archive is parsed without an error. `printSymtabError()` should be called to get more error details.

```
static bool openArchive(Archive *&img,
                        char *mem_image,
                        size_t size)
```

This factory method creates a new `Archive` object for an archive file in memory. This object serves as a handle to the parsed archive file. `mem_image` represents the pointer to the archive to be parsed. `size` represents the size of the memory image. The `Archive` object is returned in `img` if the parsing succeeds. This method returns `false` if the given file is not an archive. The error is set to

`Not_An_Archive`. This returns `true` if the archive is parsed without an error. `printSymtabError()` should be called to get more error details. This method is not supported currently on all ELF platforms.

```
bool getMember(Symtab *&img,
               string member_name)
```

This method returns the member object handle if the member exists in the archive. `img` corresponds to the object handle for the member. This method returns `false` if the member with name `member_name` does not exist else returns `true`.

```
bool getMemberByOffset(Symtab *&img,
                       Offset memberOffset)
```

This method returns the member object handle if the member exists at the start offset `memberOffset` in the archive. `img` corresponds to the object handle for the member. This method returns `false` if the member with name `member_name` does not exist else returns `true`.

```
bool getAllMembers(vector <Symtab *> &members)
```

This method returns all the member object handles in the archive. Returns `true` on success with `members` containing the `Symtab` Objects for all the members in the archive.

```
bool isMemberInArchive(string member_name)
```

This method returns `true` if the member with name `member_name` exists in the archive or else returns `false`.

```
bool findMemberWithDefinition(Symtab *&obj,
                              string name)
```

This method retrieves the member in an archive which contains the definition to a symbol with mangled name `name`. Returns `true` with `obj` containing the `Symtab` handle to that member or else returns `false`.

```
static SymtabError getLastError()
```

This method returns an error value for the previously performed operation that resulted in a failure. SymtabAPI sets a global error value in case of error during any operation. This call returns the last error that occurred while performing any operation.

```
static string printError(SymtabError serr)
```

This method returns a detailed description of the enum value `serr` in human readable format.

6.9 Class Region

This class represents a contiguous range of code or data as encoded in the object file. For ELF, regions represent ELF sections.

perm_t	Meaning
RP_R	Read-only data
RP_RW	Read/write data
RP_RX	Read-only code
RP_RWX	Read/write code

RegionType	Meaning
RT_TEXT	Executable code
RT_DATA	Read/write data
RT_TEXTDATA	Mix of code and data
RT_SYMTAB	Static symbol table
RT_STRTAB	String table used by the symbol table
RT_BSS	0-initialized memory
RT_SYMVERSIONS	Versioning information for symbols
RT_SYMVERDEF	Versioning information for symbols
RT_SYMVERNEEDED	Versioning information for symbols
RT_REL	Relocation section
RT_RELA	Relocation section
RT_PLTREL	Relocation section for PLT (inter-library references) entries
RT_PLTRELA	Relocation section for PLT (inter-library references) entries
RT_DYNAMIC	Decription of library dependencies
RT_HASH	Fast symbol lookup section
RT_GNU_HASH	GNU-specific fast symbol lookup section
RT_OTHER	Miscellaneous information

Method name	Return type	Method description
getRegionNumber	unsigned	Index of the region in the file, starting at 0.
getRegionName	std::string	Name of the region (e.g. .text, .data).
getPtrToRawData	void *	Read-only pointer to the region's raw data buffer.
getDiskOffset	Offset	Offset within the file where the region begins.
getDiskSize	unsigned long	Size of the region's data in the file.
getMemOffset	Offset	Location where the region will be loaded into memory, modified by the file's base load address.
getMemSize	unsigned long	Size of the region in memory, including zero padding.
isBSS	bool	Type query for uninitialized data regions (zero disk size, non-zero memory size).
isText	bool	Type query for executable code regions.
isData	bool	Type query for initialized data regions.
getRegionPermissions	perm_t	Permissions for the region; perm_ t is defined above.
getRegionType	RegionType	Type of the region as defined above.
isLoadable	bool	True if the region will be loaded into memory (e.g., code or data), false otherwise (e.g., debug information).
isDirty	bool	True if the region's raw data buffer has been modified by the user.

```
static Region *createRegion(Offset diskOff,
                            perm_t perms,
                            RegionType regType,
                            unsigned long diskSize = 0,
                            Offset memOff = 0,
                            unsigned long memSize = 0,
                            std::string name = "",
                            char *rawDataPtr = NULL,
                            bool isLoadable = false,
                            bool isTLS = false,
                            unsigned long memAlign = sizeof(unsigned))
```

This factory method creates a new region with the provided arguments. The `memOff` and `memSize` parameters identify where the region should be loaded in memory (modified by the base address of the file); if `memSize` is larger than `diskSize` the remainder will be zero-padded (e.g., bss regions).

```
bool isOffsetInRegion(const Offset &offset) const
```

Return `true` if the offset falls within the region data.

```
void setRegionNumber(unsigned index) const
```

Sets the region index; the value must not overlap with any other regions and is not checked.

```
bool setPtrToRawData(void *newPtr,
                     unsigned long rawsize)
```

Set the raw data pointer of the region to `newPtr`. `rawsize` represents the size of the raw data buffer. Returns `true` if success or `false` when unable to set/change the raw data of the region. Implicitly changes the disk and memory sizes of the region.

```
bool setRegionPermissions(perm_t newPerms)
```

This sets the regions permissions to `newPerms`. Returns `true` on success.

```
bool setLoadable(bool isLoadable)
```

This method sets whether the region is loaded into memory at load time. Returns `true` on success.

```
bool addRelocationEntry(Offset relocationAddr,
                        Symbol *dynref,
                        unsigned long relType,
                        Region::RegionType rtype = Region::RT_REL)
```

Creates and adds a relocation entry for this region. The symbol `dynref` represents the symbol used by he relocation, `relType` is the (platform-specific) relocation type, and `rtype` represents whether the relocation is REL or RELA (ELF-specific).

```
vector<relocationEntry> &getRelocations()
```

Get the vector of relocation entries that will modify this region. The vector should not be modified.

```
bool addRelocationEntry(const relocationEntry& rel)
```

Add the provided relocation entry to this region.

```
bool patchData(Offset off,
               void *buf,
               unsigned size);
```

Patch the raw data for this region. `buf` represents the buffer to be patched at offset `off` and size `size`.

6.9.1 REMOVED

The following methods were removed since they were inconsistent and dangerous to use.

`Offset getRegionAddr() const`

Please use `getDiskOffset` or `getMemOffset` instead, as appropriate.

`unsigned long getRegionSize() const`

Please use `getDiskSize` or `getMemSize` instead, as appropriate.

6.10 Relocation Information

This class represents object relocation information.

`Offset target_addr() const`

Specifies the offset that will be overwritten when relocations are processed.

`Offset rel_addr() const`

Specifies the offset of the relocation itself.

`Offset addend() const`

Specifies the value added to the relocation; whether this value is used or not is specific to the relocation type.

`const std::string name() const`

Specifies the user-readable name of the relocation.

`Symbol *getDynSym() const`

Specifies the symbol whose final address will be used in the relocation calculation. How this address is used is specific to the relocation type.

`unsigned long getRelType() const`

Specifies the platform-specific relocation type.

6.11 Class ExceptionBlock

This class represents an exception block present in the object file. This class gives all the information pertaining to that exception block.

Method name	Return type	Method description
hasTry	bool	True if the exception block has a try block.
tryStart	Offset	Start of the try block if it exists, else 0.
tryEnd	Offset	End of the try block if it exists, else 0.
trySize	Offset	Size of the try block if it exists, else 0.
catchStart	Offset	Start of the catch block.

```
bool contains(Offset addr) const
```

This method returns `true` if the offset `addr` is contained with in the try block. If there is no try block associated with this exception block or the offset does not fall within the try block, it returns `false`.

6.12 Class localVar

This represents a local variable or parameter of a function.

Method name	Return type	Method description
getName	string &	Name of the local variable or parameter.
getType	Type *	Type associated with the variable.
getFileName	string &	File where the variable was declared, if known.
getLineNum	int	Line number where the variable was declared, if known.

```
vector<VariableLocation> &getLocationLists()
```

A local variable can be in scope at different positions and based on that it is accessible in different ways. Location lists provide a way to encode that information. The method retrieves the location list, specified in terms of `VariableLocation` structures (section 6.13) where the variable is in scope.

6.13 Class VariableLocation

The `VariableLocation` class is an encoding of the location of a variable in memory or registers.

```
typedef enum {
   storageUnset,
   storageAddr,
   storageReg,
```

```
   storageRegOffset
} storageClass;

typedef enum {
   storageRefUnset,
   storageRef,
   storageNoRef
} storageRefClass;

struct VariableLocation  {
    storageClass stClass;
    storageRefClass refClass;
    MachRegister mr_reg;
    long frameOffset;
    Address lowPC;
    Address hiPC;
}
```

A `VariableLocation` is valid within the address range represented by `lowPC` and `hiPC`. If these are 0 and (Address) -1, respectively, the `VariableLocation` is always valid.

The location represented by the `VariableLocation` can be determined by the user as follows:

- stClass == storageAddr

  refClass == storageRef The frameOffset member contains the address of a pointer to the variable.

  refClass == storageNoRef The frameOffset member contains the address of the variable.

- stClass == storageReg

  refClass == storageRef The register named by mr_reg contains the address of the variable.

  refClass == storageNoRef The register named by mr_reg member contains the variable.

- stClass == storageRegOffset

  refClass == storageRef The address computed by adding frameOffset to the contents of mr_reg contains a pointer to the variable.

  refClass == storageNoRef The address computed by adding frameOffset to the contents of mr_reg contains the variable.

7 API Reference - Line Number Interface

This section describes the line number interface for the SymtabAPI library. Currently this interface has the following capabilities:

- Look up address ranges for a given line number.
- Look up source lines for a given address.
- Add new line information. This information will be available for lookup, but will not be included with an emitted object file.

In order to look up or add line information, the user/application must have already parsed the object file and should have a Symtab handle to the object file. For more information on line information lookups through the Symtab class refer to Section 6. The rest of this section describes the classes that are part of the line number interface.

7.1 Class LineInformation

This class represents an entire line map for a module. This contains mappings from a line number within a source to the address ranges.

```
bool getAddressRanges(const char * lineSource,
                      unsigned int LineNo,
                      std::vector<AddressRange> & ranges)
```

This methos returns the address ranges in `ranges` corresponding to the line with line number `lineNo` in the source file `lineSource`. Searches within this line map. Return `true` if at least one address range corresponding to the line number was found and returns `false` if none found.

```
bool getSourceLines(Offset addressInRange,
                    std::vector<Statement *> & lines)
bool getSourceLines(Offset addressInRange,
                    std::vector<LineNoTuple> & lines)
```

These methods returns the source file names and line numbers corresponding to the given address `addressInRange`. Searches within this line map. Return `true` if at least one tuple corresponding to the offset was found and returns `false` if none found. Note that the order of arguments is reversed from the corresponding interfaces in `Module` and `Symtab`.

```
bool addLine(const char * lineSource,
             unsigned int lineNo,
             unsigned int lineOffset,
             Offset lowInclusiveAddr,
             Offset highExclusiveAddr)
```

This method adds a new line to the line Map. `lineSource` represents the source file name. `lineNo` represents the line number.

```
bool addAddressRange(Offset lowInclusiveAddr,
                     Offset highExclusiveAddr,
                     const char* lineSource,
                     unsigned int lineNo,
                     unsigned int lineOffset = 0);
```

This method adds an address range `[lowInclusiveAddr, highExclusiveAddr)` for the line with line number `lineNo` in source file `lineSource`.

`LineInformation::const_iterator begin() const`

This method returns an iterator pointing to the beginning of the line information for the module. This is useful for iterating over the entire line information present in a module. An example described in Section **??** gives more information on how to use `begin()` for iterating over the line information.

`LineInformation::const_iterator end() const`

This method returns an iterator pointing to the end of the line information for the module. This is useful for iterating over the entire line information present in a module. An example described in Section **??** gives more information on how to use `end()` for iterating over the line information.

7.2 Class Statement

A `Statement` is the base representation of line information.

Method name	Return type	Method description
startAddr	Offset	Starting address of this line in the file.
endAddr	Offset	Ending address of this line in the file.
getFile	std::string	File that contains the line.
getLine	unsigned int	Line number.
getColumn	unsigned int	Starting column number.

For backwards compatibility, this class may also be referred to as a `LineNoTuple`, and provides the following legacy member variables. They should not be used and will be removed in a future version of SymtabAPI.

Member	Return type	Method description
first	const char *	Equivalent to getFile.
second	unsigned int	Equivalent to getLine.
column	unsigned int	Equivalent to getColumn.

7.3 Iterating over Line Information

The `LineInformation` class also provides the ability for iterating over its data (line numbers and their corresponding address ranges). The following example shows how to iterate over the line information for a given module using SymtabAPI.

```
//Example showing how to iterate over the line information for a given module.
using namespace Dyninst;
using namespace SymtabAPI;
```

```
//Obj represents a handle to a parsed object file using symtabAPI
//Module handle for the module
Module *mod;

//Find the module \lq foo\rq within the object.
obj->findModuleByName(mod, "foo");

// Get the Line Information for module foo.
LineInformation *info = mod->getLineInformation();

//Iterate over the line information
LineInformation::const_iterator iter;
for( iter = info->begin(); iter != info->end(); iter++)
{
// First component represents the address range for the line
const std::pair<Offset, Offset> addrRange = iter->first;

//Second component gives information about the line itself.
LineNoTuple lt = iter->second;
}
```

8 API Reference - Type Interface

This section describes the type interface for the SymtabAPI library. Currently this interface has the following capabilities:

- Look up types within an object file.
- Extend the types to create new types and add them to the Symtab file representation. These types will be available for lookup but will not be added if a new object file is produced.

The rest of the section describes the classes that are part of the type interface.

8.1 Class Type

The class `Type` represents the types of variables, parameters, return values, and functions. Instances of this class can represent language predefined types (e.g. `int`, `float`), already defined types in the Object File or binary (e.g., structures compiled into the binary), or newly created types (created using the create factory methods of the corresponding type classes described later in this section) that are added to SymtabAPI by the user.

As described in Section 2.2, this class serves as a base class for all the other classes in this interface. An object of this class is returned from type look up operations performed through the Symtab class described in Section 6. The user can then obtain the specific type object from the generic Type class object. The following example shows how to get the specific object from a given `Type` object returned as part of a look up operation.

```
//Example shows how to retrieve a structure type object from a given ‘‘Type’’ object
using namespace Dyninst;
using namespace SymtabAPI;

//Obj represents a handle to a parsed object file using symtabAPI
//Find a structure type in the object file
Type *structType = obj->findType(‘‘structType1’’);

// Get the specific typeStruct object
typeStruct *stType = structType->isStructType();
```

`string &getName()`

This method returns the name associated with this type. Each of the types is represented by a symbolic name. This method retrieves the name for the type. For example, in the example above "structType1" represents the name for the `structType` object.

`bool setName(string zname)`

This method sets the name of this type to name. Returns `true` on success and `false` on failure.

```
typedef enum{
    dataEnum,
    dataPointer,
    dataFunction,
    dataSubrange,
    dataArray,
    dataStructure,
    dataUnion,
    dataCommon,
    dataScalar,
    dataTypeDefine,
    dataReference,
    dataUnknownType,
    dataNullType,
    dataTypeClass
} dataClass;
```

`dataClass getType()`

This method returns the data class associated with the type. This value should be used to convert this generic type object to a specific type object which offers more functionality by using the corresponding query function described later in this section. For example, if this method returns `dataStructure` then the `isStructureType()` should be called to dynamically cast the `Type` object to the `typeStruct` object.

`typeId_t getID()`

This method returns the ID associated with this type. Each type is assigned a unique ID within the object file. For example an integer scalar built-in type is assigned an ID -1.

`unsigned getSize()`

This method returns the total size in bytes occupied by the type.

`typeEnum *getEnumType()`

If this `Type` hobject represents an enum type, then return the object casting the `Type` object to `typeEnum` otherwise return `NULL`.

`typePointer *getPointerType()`

If this `Type` object represents an pointer type, then return the object casting the `Type` object to `typePointer` otherwise return `NULL`.

`typeFunction *getFunctionType()`

If this `Type` object represents an `Function` type, then return the object casting the `Type` object to `typeFunction` otherwise return `NULL`.

`typeRange *getSubrangeType()`

If this `Type` object represents a `Subrange` type, then return the object casting the `Type` object to `typeSubrange` otherwise return `NULL`.

`typeArray *getArrayType()`

If this `Type` object represents an `Array` type, then return the object casting the `Type` object to `typeArray` otherwise return `NULL`.

`typeStruct *getStructType()`

If this `Type` object represents a `Structure` type, then return the object casting the `Type` object to `typeStruct` otherwise return `NULL`.

`typeUnion *getUnionType()`

If this `Type` object represents a `Union` type, then return the object casting the `Type` object to `typeUnion` otherwise return `NULL`.

`typeScalar *getScalarType()`

If this `Type` object represents a `Scalar` type, then return the object casting the `Type` object to `typeScalar` otherwise return `NULL`.

`typeCommon *getCommonType()`

If this `Type` object represents a `Common` type, then return the object casting the `Type` object to `typeCommon` otherwise return `NULL`.

`typeTypedef *getTypeDefType()`

If this `Type` object represents a `TypeDef` type, then return the object casting the `Type` object to `typeTypedef` otherwise return `NULL`.

`typeRef *getRefType()`

If this `Type` object represents a `Reference` type, then return the object casting the `Type` object to `typeRef` otherwise return `NULL`.

8.2 Class typeEnum

This class represents an enumeration type containing a list of constants with values. This class is derived from `Type`, so all those member functions are applicable. `typeEnum` inherits from the `Type` class.

```
static typeEnum *create(string &name,
                        vector<pair<string, int> *> &consts,
                        Symtab *obj = NULL)
static typeEnum *create(string &name,
                        vector<string> &constNames,
                        Symtab *obj)
```

These factory methods create a new enumerated type. There are two variations to this function. `consts` supplies the names and IdÂŠs of the constants of the enum. The first variant is used when user-defined identifiers are required; the second variant is used when system-defined identifiers will be used. The newly created type is added to the `Symtab` object `obj`. If `obj` is `NULL` the type is not added to any object file, but it will be available for further queries.

```
bool addConstant(const string &constname,
                 int value)
```

This method adds a constant to an enum type with name `constName` and value `value`. Returns `true` on success and `false` on failure.

```
std::vector<std::pair<std::string, int> > &getConstants();
```

This method returns the vector containing the enum constants represented by a (name, value) pair of the constant.

```
bool setName(const char* name)
```

This method sets the new name of the enum type to `name`. Returns `true` if it succeeds, else returns `false`.

```
bool isCompatible(Type *type)
```

This method returns `true` if the enum type is compatible with the given type `type` or else returns `false`.

8.3 Class typeFunction

This class represents a function type, containing a list of parameters and a return type. This class is derived from `Type`, so all the member functions of class `Type` are applicable. `typeFunction` inherits from the `Type` class.

```
static typeFunction *create(string &name,
                            Type *retType,
                            vector<Type *> &paramTypes,
                            Symtab *obj = \code{NULL})
```

This factory method creates a new function type with name `name`. `retType` represents the return type of the function and `paramTypes` is a vector of the types of the parameters in order. The the newly created type is added to the `Symtab` object `obj`. If `obj` is `NULL` the type is not added to any object file, but it will be available for further queries.

```
bool isCompatible(Type *type)
```

This method returns `true` if the function type is compatible with the given type `type` or else returns `false`.

`bool addParam(Type *type)`

This method adds a new function parameter with type `type` to the function type. Returns `true` if it succeeds, else returns `false`.

`Type *getReturnType() const`

This method returns the return type for this function type. Returns `NULL` if there is no return type associated with this function type.

`bool setRetType(Type *rtype)`

This method sets the return type of the function type to `rtype`. Returns `true` if it succeeds, else returns `false`.

`bool setName(string &name)`

This method sets the new name of the function type to `name`. Returns `true` if it succeeds, else returns `false`.

`vector< Type *> &getParams() const`

This method returns the vector containing the individual parameters represented by their types in order. Returns `NULL` if there are no parameters to the function type.

8.4 Class typeScalar

This class represents a scalar type. This class is derived from `Type`, so all the member functions of class `Type` are applicable. `typeScalar` inherits from the Type class.

`static typeScalar *create(string &name, int size, Symtab *obj = NULL)`

This factory method creates a new scalar type. The `name` field is used to specify the name of the type, and the `size` parameter is used to specify the size in bytes of each instance of the type. The newly created type is added to the `Symtab` object `obj`. If `obj` is `NULL` the type is not added to any object file, but it will be available for further queries.

`bool isSigned()`

This method returns `true` if the scalar type is signed or else returns `false`.

```
bool isCompatible(Type *type)
```

This method returns `true` if the scalar type is compatible with the given type `type` or else returns `false`.

8.5 Class Field

This class represents a field in a container. For e.g. a field in a structure/union type.

```
typedef enum {
    visPrivate,
    visProtected,
    visPublic,
    visUnknown
} visibility_t;
```

A handle for identifying the visibility of a certain `Field` in a container type. This can represent private, public, protected or unknown(default) visibility.

```
Field(string &name,
      Type *type,
      visibility_t vis = visUnknown)
```

This constructor creates a new field with name `name`, type `type` and visibility `vis`. This newly created `Field` can be added to a container type.

```
const string &getName()
```

This method returns the name associated with the field in the container.

```
Type *getType()
```

This method returns the type associated with the field in the container.

```
int getOffset()
```

This method returns the offset associated with the field in the container.

```
visibility_t getVisibility()
```

This method returns the visibility associated with a field in a container. This returns `visPublic` for the variables within a common block.

8.6 Class fieldListType

This class represents a container type. It is one of the three categories of types as described in Section 2.2. The structure and the union types fall under this category. This class is derived from `Type`, so all the member functions of class `Type` are applicable. `fieldListType` inherits from the `Type` class.

```
vector<Field *> *getComponents()
```

This method returns the list of all fields present in the container. This gives information about the name, type and visibility of each of the fields. Returns `NULL` of there are no fields.

```
void addField(std::string fieldname,
              Type *type,
              int offsetVal = -1,
              visibility_t vis = visUnknown)
```

This method adds a new field at the end to the container type with field name `fieldname`, type `type` and type visibility `vis`.

```
void addField(unsigned num,
              std::string fieldname,
              Type *type,
              int offsetVal = -1,
              visibility_t vis = visUnknown)
```

This method adds a field after the field with number `num` with field name `fieldname`, type `type` and type visibility `vis`.

```
void addField(Field *fld)
```

This method adds a new field `fld` to the container type.

```
void addField(unsigned num,
              Field *fld)
```

This method adds a field `fld` after field `num` to the container type.

8.6.1 Class typeStruct : public fieldListType

This class represents a structure type. The structure type is a special case of the container type. The fields of the structure represent the fields in this case. As a subclass of class `fieldListType`, all methods in `fieldListType` are applicable.

```
static typeStruct *create(string &name,
                          vector<pair<string, Type *>*> &flds,
                          Symtab *obj = NULL)
```

This factory method creates a new struct type. The name of the structure is specified in the `name` parameter. The `flds` vector specifies the names and types of the fields of the structure type. The newly created type is added to the `Symtab` object `obj`. If `obj` is `NULL` the type is not added to any object file, but it will be available for further queries.

```
static typeStruct *create(string &name,
                          vector<Field *> &fields,
                          Symtab *obj = NULL)
```

This factory method creates a new struct type. The name of the structure is specified in the `name` parameter. The `fields` vector specifies the fields of the type. The newly created type is added to the `Symtab` object `obj`. If `obj` is `NULL` the type is not added to any object file, but it will be available for further queries.

```
bool isCompatible(Type *type)
```

This method returns `true` if the struct type is compatible with the given type `type` or else returns `false`.

8.6.2 Class typeUnion

This class represents a union type, a special case of the container type. The fields of the union type represent the fields in this case. As a subclass of class `fieldListType`, all methods in `fieldListType` are applicable. `typeUnion` inherits from the `fieldListType` class.

```
static typeUnion *create(string &name,
                         vector<pair<string, Type *>*> &flds,
                         Symtab *obj = NULL)
```

This factory method creates a new union type. The name of the union is specified in the `name` parameter. The `flds` vector specifies the names and types of the fields of the union type. The newly created type is added to the `Symtab` object `obj`. If `obj` is `NULL` the type is not added to any object file, but it will be available for further queries.

```
static typeUnion *create(string &name,
                         vector<Field *> &fields,
                         Symtab *obj = NULL)
```

This factory method creates a new union type. The name of the structure is specified in the `name` parameter. The `fields` vector specifies the fields of the type. The newly created type is added to the `Symtab` object `obj`. If `obj` is `NULL` the type is not added to any object file, but it will be available for further queries.

```
bool isCompatible(Type *type)
```

This method returns `true` if the union type is compatible with the given type `type` or else returns `false`.

8.6.3 Class typeCommon

This class represents a common block type in fortran, a special case of the container type. The variables of the common block represent the fields in this case. As a subclass of class `fieldListType`, all methods in `fieldListType` are applicable. `typeCommon` inherits from the `Type` class.

```
vector<CBlocks *> *getCBlocks()
```

This method returns the common block objects for the type. The methods of the `CBlock` can be used to access information about the members of a common block. The vector returned by this function contains one instance of `CBlock` for each unique definition of the common block.

8.6.4 Class CBlock

This class represents a common block in Fortran. Multiple functions can share a common block.

```
bool getComponents(vector<Field *> *vars)
```

This method returns the vector containing the individual variables of the common block. Returns `true` if there is at least one variable, else returns `false`.

```
bool getFunctions(vector<Symbol *> *funcs)
```

This method returns the functions that can see this common block with the set of variables described in `getComponents` method above. Returns `true` if there is at least one function, else returns `false`.

8.7 Class derivedType

This class represents a derived type which is a reference to another type. It is one of the three categories of types as described in Section 2.2. The pointer, reference and the typedef types fall under this category. This class is derived from `Type`, so all the member functions of class `Type` are applicable.

```
Type *getConstituentType() const
```

This method returns the type of the base type to which this type refers to.

8.7.1 Class typePointer

This class represents a pointer type, a special case of the derived type. The base type in this case is the type this particular type points to. As a subclass of class `derivedType`, all methods in `derivedType` are also applicable.

```
static typePointer *create(string &name,
                           Type *ptr,
                           Symtab *obj = NULL)
static typePointer *create(string &name,
                           Type *ptr,
                           int size,
                           Symtab *obj = NULL)
```

These factory methods create a new type, named `name`, which points to objects of type `ptr`. The first form creates a pointer whose size is equal to sizeof(void*) on the target platform where the application is running. In the second form, the size of the pointer is the value passed in the `size` parameter. The newly created type is added to the `Symtab` object `obj`. If obj is `NULL` the type is not added to any object file, but it will be available for further queries.

```
bool isCompatible(Type *type)
```

This method returns `true` if the Pointer type is compatible with the given type `type` or else returns `false`.

```
bool setPtr(Type *ptr)
```

This method sets the pointer type to point to the type in `ptr`. Returns `true` if it succeeds, else returns `false`.

8.7.2 Class typeTypedef

This class represents a `typedef` type, a special case of the derived type. The base type in this case is the `Type`. This particular type is typedefed to. As a subclass of class `derivedType`, all methods in `derivedType` are also applicable.

```
static typeTypedef *create(string &name,
                           Type *ptr,
                           Symtab *obj = NULL)
```

This factory method creates a new type called `name` and having the type `ptr`. The newly created type is added to the `Symtab` object `obj`. If `obj` is `NULL` the type is not added to any object file, but it will be available for further queries.

```
bool isCompatible(Type *type)
```

This method returns `true` if the typedef type is compatible with the given type `type` or else returns `false`.

8.7.3 Class typeRef

This class represents a reference type, a special case of the derived type. The base type in this case is the `Type` this particular type refers to. As a subclass of class `derivedType`, all methods in `derivedType` are also applicable here.

```
static typeRef *create(string &name,
                       Type *ptr,
                       Symtab * obj = NULL)
```

This factory method creates a new type, named `name`, which is a reference to objects of type `ptr`. The newly created type is added to the `Symtab` object `obj`. If `obj` is `NULL` the type is not added to any object file, but it will be available for further queries.

```
bool isCompatible(Type *type)
```

This method returns `true` if the ref type is compatible with the given type `type` or else returns `false`.

8.8 Class rangedType

This class represents a range type with a lower and an upper bound. It is one of the three categories of types as described in section 2.2. The sub-range and the array types fall under this category. This class is derived from `Type`, so all the member functions of class `Type` are applicable.

```
unsigned long getLow() const
```

This method returns the lower bound of the range. This can be the lower bound of the range type or the lowest index for an array type.

```
unsigned long getHigh() const
```

This method returns the higher bound of the range. This can be the higher bound of the range type or the highest index for an array type.

8.8.1 Class typeSubrange

This class represents a sub-range type. As a subclass of class `rangedType`, all methods in `rangedType` are applicable here. This type is usually used to represent a sub-range of another type. For example, a `typeSubrange` can represent a sub-range of the array type or a new integer type can be declared as a sub range of the integer using this type.

```
static typeSubrange *create(string &name,
                            int size,
                            int low,
                            int hi,
                            symtab *obj = NULL)
```

This factory method creates a new sub-range type. The name of the type is `name`, and the size is `size`. The lower bound of the type is represented by `low`, and the upper bound is represented by `high`. The newly created type is added to the `Symtab` object `obj`. If `obj` is `NULL` the type is not added to any object file, but it will be available for further queries.

```
bool isCompatible(Type *type)
```

This method returns `true` if this sub range type is compatible with the given type `type` or else returns `false`.

8.8.2 Class typeArray

This class represents an `Array` type. As a subclass of class `rangedType`, all methods in `rangedType` are applicable.

```
static typeArray *create(string &name,
                         Type *type,
                         int low,
                         int hi,
                         Symtab *obj = NULL)
```

This factory method creates a new array type. The name of the type is `name`, and the type of each element is `type`. The index of the first element of the array is `low`, and the last is `high`. The newly created type is added to the `Symtab` object `obj`. If `obj` is `NULL` the type is not added to any object file, but it will be available for further queries.

```
bool isCompatible(Type *type)
```

This method returns `true` if the array type is compatible with the given type `type` or else returns `false`.

```
Type *getBaseType() const
```

This method returns the base type of this array type.

9 API Reference - Dynamic Components

Unlike the static components discussed in Section 6, which operate on files, SymtabAPI's dynamic components operate on a process. The dynamic components currently consist of the Dynamic Address Translation system, which translates between absolute addresses in a running process and static SymtabAPI objects.

9.1 Class AddressLookup

The `AddressLookup` class provides a mapping interface for determining the address in a process where a SymtabAPI object is loaded. A single dynamic library may load at different addresses in different processes. The 'address' fields in a dynamic library's symbol tables will contain offsets rather than absolute addresses. These offsets can be added to the library's load address, which is computed at runtime, to determine the absolute address where a symbol is loaded.

The `AddressLookup` class examines a process and finds its dynamic libraries and executables and each one's load address. This information can be used to map between SymtabAPI objects and absolute addresses.

Each `AddressLookup` instance is associated with one process. An `AddressLookup` object can be created to work with the currently running process or a different process on the same system.

On the Linux platform the `AddressLookup` class needs to read from the process' address space to determine its shared objects and load addresses. By default, `AddressLookup` will attach to another process using a debugger interface to read the necessary information, or simply use `memcpy` if reading from the current process. The default behavior can be changed by implementing a new ProcessReader class and passing an instance of it to the create`AddressLookup` factor constructors. The ProcessReader class is discussed in more detail in Section 9.2.

When an `AddressLookup` object is created for a running process it takes a snapshot of the process' currently loaded libraries and their load addresses. This snapshot is used to answer queries into the `AddressLookup` object, and is not automatically updated when the process loads or unloads libraries. The refresh function can be used to updated an `AddressLookup` object's view of its process.

```
static AddressLookup *createAddressLookup(ProcessReader *reader = NULL)
```

This factory constructor creates a new `AddressLookup` object associated with the process that called this function. The returned `AddressLookup` object should be cleaned with the delete operator when it is no longer needed. If the reader parameter is non-NULL on Linux then the new `AddressLookup` object will use reader to read from the target process. This function returns the new `AddressLookup` object on success and `NULL` on error.

```
static AddressLookup *createAddressLookup(PID pid,
                                          ProcessReader *reader = NULL)
```

This factory constructor creates a new `AddressLookup` object associated with the process referred to by `pid`. The returned `AddressLookup` object should be cleaned with the delete operator when it is no longer needed. If the `reader` parameter is non-NULL on Linux then the new `AddressLookup` object will use it to read from the target process. This function returns the new `AddressLookup` object on success and `NULL` on error.

```
typedef struct {
    std::string name;
    Address codeAddr;
    Address dataAddr;
} LoadedLibrary;

static AddressLookup *createAddressLookup(const std::vector<LoadedLibrary> &ll)
```

This factory constructor creates a new `AddressLookup` associated with a previously collected list of libraries from a process. The list of libraries can initially be collected with the `getLoadAddresses` function. The list can then be used with this function to re-create the `AddressLookup` object, even if the original process no longer exists. This can be useful for off-line address lookups, where only the load addresses are collected while the process exists and then all address translation is done after the process has terminated. This function returns the new `AddressLookup` object on success and `NULL` on error.

```
bool getLoadAddresses(std::vector<LoadedLibrary> &ll)
```

This function returns a vector of LoadedLibrary objects that can be used by the `createAddressLookup(const std::vector<LoadedLibrary> &ll)` function to create a new `AddressLookup` object. This function is usually used as part of an off-line address lookup mechanism. This function returns `true` on success and `false` on error.

```
bool refresh()
```

When a `AddressLookup` object is initially created it takes a snapshot of the libraries currently loaded in a process, which is then used to answer queries into this API. As the process runs more libraries may be loaded and unloaded, and this snapshot may become out of date. An `AddressLookup`'s view of a process can be updated by calling this function, which causes it to examine the process for loaded and unloaded objects and update its data structures accordingly. This function returns `true` on success and `false` on error.

```
bool getAddress(Symtab *tab,
                Symbol *sym,
                Address &addr)
```

Given a `Symtab` object, `tab`, and a symbol, `sym`, this function returns the address, `addr`, where the symbol can be found in the process associated with this `AddressLookup`. This function returns `true` if it was able to successfully lookup the address of `sym` and `false` otherwise.

```
bool getAddress(Symtab *tab,
                Offset off,
                Address &addr)
```

Given a `Symtab` object, `tab`, and an offset into that object, `off`, this function returns the address, `addr`, of that location in the process associated with this `AddressLookup`. This function returns `true` if it was able to successfully lookup the address of sym and `false` otherwise.

```
bool getSymbol(Address addr,
               Symbol * &sym,
               Symtab* &tab,
               bool close = false)
```

Given an address, `addr`, this function returns the `Symtab` object, `tab`, and `Symbol`, `sym`, that reside at that address. If the close parameter is `true` then `getSymbol` will return the nearest symbol that comes before `addr`; this can be useful when looking up the function that resides at an address. This function returns `true` if it was able to find a symbol and `false` otherwise.

```
bool getOffset(Address addr,
               Symtab* &tab,
               Offset &off)
```

Given an address, `addr`, this function returns the `Symtab` object, `tab`, and an offset into `tab`, `off`, that reside at that address. This function returns `true` on success and `false` otherwise.

```
bool getOffset(Address addr,
               LoadedLibrary &lib,
               Offset &off)
```

As above, but returns a `LoadedLibrary` data structure instead of a Symtab.

```
bool getAllSymtabs(std::vector<Symtab *> &tabs)
```

This function returns all `Symtab` objects that are contained in the process represented by this `AddressLookup` object. This will include the process's executable and all shared objects loaded by this process. This function returns `true` on success and `false` otherwise.

```
bool getLoadAddress(Symtab *sym,
                    Address &load_address)
```

Given a `Symtab` object, `sym`, that resides in the process associated with this `AddressLookup`, this function returns `sym`'s load address. On the AIX system, where an object can have one load address for its code and one for its data, this function will return the code's load address. Use `getDataLoadAddress` to get the data load address. This function returns `true` on success and `false` otherwise.

```
bool getDataLoadAddress(Symtab *sym,
                        Address &load_addr)
```

Given a Symtab object, `sym`, this function returns the load address of its data section. This function will return the data load address on AIX systems only, all other supported operating systems return zero. This function returns `true` on success and `false` otherwise.

9.2 Class ProcessReader

The implementation of the `AddressLookup` on Linux requires it to be able to read from the target process's address space. By default, reading from another process on the same system this is done through the operating system debugger interface. A user can provide their own process reading mechanism by implementing a child of the `ProcessReader` class and passing it to the `AddressLookup` constructors. The API described in this section is an interface that a user can implement. With the exception of the `ProcessReader` constructor, these functions should not be called by user code.

The `ProcessReader` is defined, but not used, on non-Linux systems.

```
ProcessReader()
```

This constructor for a `ProcessReader` should be called by any child class constructor.

```
virtual bool ReadMem(Address traced,
                     void *inSelf,
                     unsigned size) = 0
```

This function should read `size` bytes from the address at `traced` into the buffer pointed to by `inSelf`. This function must return `true` on success and `false` on error.

```
virtual bool GetReg(MachRegister reg,
                    MachRegisterVal &val) = 0
```

This function reads from the register specified by `reg` and places the result in `val`. It must return `true` on success and `false` on failure.

Paradyn Parallel Performance Tools

InstructionAPI Programmer's Guide

9.2 Release

June 2016

Computer Sciences Department
University of Wisconsin–Madison
Madison, WI 53706

Computer Science Department
University of Maryland
College Park, MD 20742

Email dyninst-api@cs.wisc.edu
Web www.dyninst.org

Contents

1 Introduction

When analyzing and modifying binary code, it is necessary to translate between raw binary instructions and an abstract form that describes the semantics of the instructions. As a part of the Dyninst project, we have developed the Instruction API, an API and library for decoding and representing machine instructions in a platform-independent manner. The Instruction API includes methods for decoding machine language, convenient abstractions for its analysis, and methods to produce disassembly from those abstractions. The current implementation supports the x86, x86-64, PowerPC-32, and PowerPC-64 instruction sets. The Instruction API has the following basic capabilities:

- Decoding: interpreting a sequence of bytes as a machine instruction in a given machine language.
- Abstract representation: representing the behavior of that instruction as an abstract syntax tree.
- Disassembly: translating an abstract representation of a machine instruction into a string representation of the corresponding assembly language instruction.

Our goal in designing the Instruction API is to provide a representation of machine instructions that can be manipulated by higher-level algorithms with minimal knowledge of platform-specific details. In addition, users who need platform-specific information should be able to access it. To do so, we provide an interface that disassembles a machine instruction, extracts an operation and its operands, converts the operands to abstract syntax trees, and presents this to the user. A user of the Instruction API can work at a level of abstraction slightly higher than assembly language, rather than working directly with machine language. Additionally, by converting the operands to abstract syntax trees, we make it possible to analyze the operands in a uniform manner, regardless of the complexity involved in the operand's actual computation.

2 InstructionAPI Modules and Abstractions

The Instruction API contains three major components: the top-level instruction representation, the abstract syntax trees representing the operands of an instruction, and the decoder that creates the entire representation. We will present an overview of the features and uses of each of these three components, followed by an example of how the Instruction API can be applied to binary analysis.

2.1 Instruction Interface

The Instruction API represents a machine language instruction as an Instruction object, which contains an Operation and a collection of Operands. The Operation contains the following items:

- The mnemonic for the machine language instruction represented by its associated Instruction

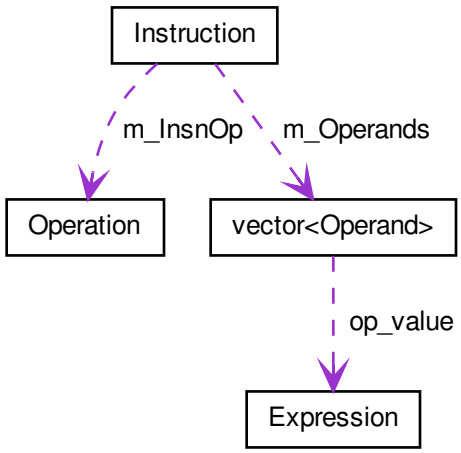


Figure 1: An Instruction and the objects it owns

- The number of operands accepted by the Operation
- Which Operands are read and/or written by the associated machine operation
- What other registers (if any) are affected by the underlying machine operation

Each Operand contains flags to indicate whether it is read, written, or both by the machine instruction represented by its parent Instruction, and contains a Expression abstract syntax tree representing the operations required to compute the value of the operand. Figure 1 depicts these ownership relationships within an Instruction.

Instruction objects provide two types of interfaces: direct read access to their components, and common summary operations on those components. The first interface allows access to the Operation and Operand data members, and each Operand object in turn allows traversal of its abstract syntax tree. More details about how to work with this abstract syntax tree can be found in Section 2.3. This interface would be used, for example, in a data flow analysis where a user wants to evaluate the results of an effective address computation given a known register state.

The second interface allows a user to get the sets of registers read and written by the instruction, information about how the instruction accesses memory, and information about how the instruction affects control flow, without having to manipulate the Operands directly. For instance, a user could implement a register liveness analysis algorithm using just this second interface (namely the `getReadSet` and `getWriteSet` functions).

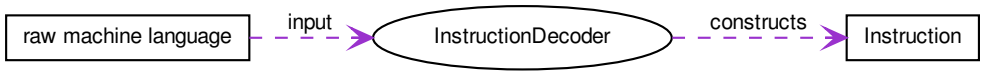


Figure 2: The InstructionDecoder's inputs and outputs

2.2 Instruction Decoding

An InstructionDecoder interprets a sequence of bytes according to a given machine language and transforms them into an instruction representation. It determines the opcode of the machine instruction, translates that opcode to an Operation object, uses that Operation to determine how to decode the instruction's Operands, and produces a decoded Instruction.

Instruction decoders are built from the following elements:

- A function to find and extract an opcode given a pointer into a buffer that points to the beginning of a machine instruction
- A table that, for a particular architecture, maps opcodes to Operations and functions that decode Operands

From these elements, it is possible to generalize the construction of Instructions from Operations and Operands to an entirely platform-independent algorithm. Likewise, much of the construction of the ASTs representing each operand can be performed in a platform-independent manner.

2.3 InstructionAST Hierarchy

The AST representation of an operand encapsulates the operations performed on registers and immediates to produce an operand for the machine language instruction.

The inheritance hierarchy of the AST classes is shown in Figure 3.

The grammar for these AST representations is simple: all leaves must be RegisterAST or Immediate nodes. These nodes may be combined using a BinaryFunction node, which may be constructed as either an addition or a multiplication. Also, a single node may descend from a Dereference node, which treats its child as a memory address. Figure 4 shows the allowable parent/child relationships within a given tree, and Figure 5 shows how an example IA32 instruction is represented using these objects.

These ASTs may be searched for leaf elements or subtrees (via `getUses` and `isUsed`) and traversed breadth-first or depth-first (via `getChildren`).

Any node in these ASTs may be evaluated. Evaluation attempts to determine the value represented by a node. If successful, it will return that value and cache it in the node. The tree structure, combined with the evaluation mechanism, allows the substitution of known register and memory values into an operand,

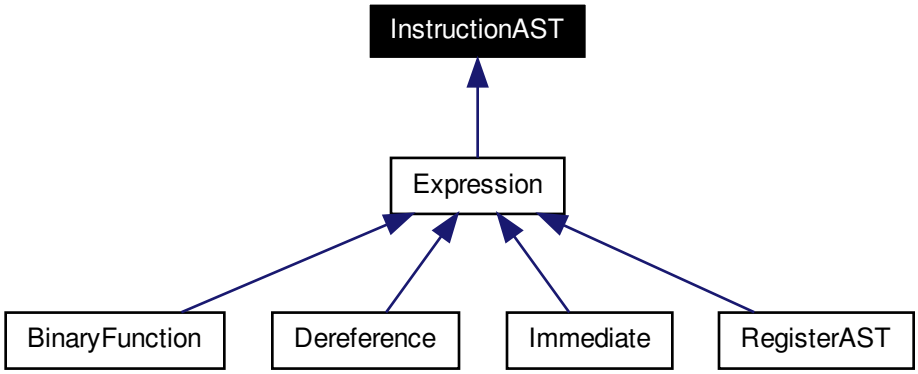


Figure 3: The InstructionAST inheritance hierarchy

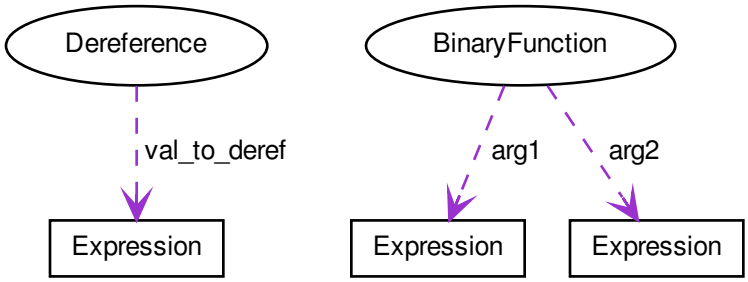


Figure 4: InstructionAST intermediate node types and the objects they own

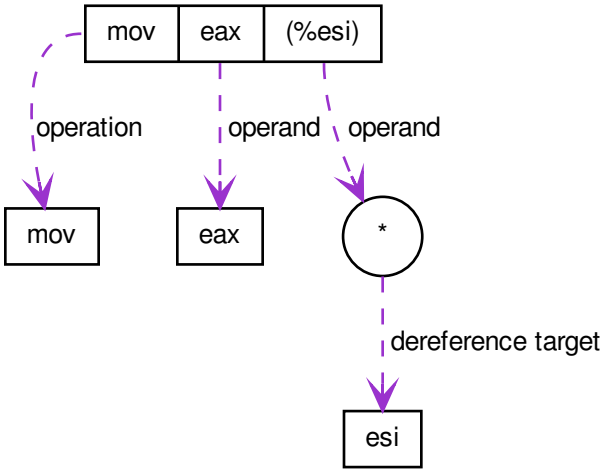


Figure 5: The decomposition of `mov %eax, (%esi)`

regardless of whether those values are known at the time an instruction is decoded. More details on this mechanism may be found in Section 3.5.

3 InstructionAPI Class Reference

3.1 Instruction Class

The Instruction class is a generic instruction representation that contains operands, read/write semantic information about those operands, and information about what other registers and memory locations are affected by the operation the instruction performs.

The purpose of an Instruction object is to join an Operation with a sequence of Operands, and provide an interface for some common summary analyses (namely, the read/write sets, memory access information, and control flow information).

The Operation contains knowledge about its mnemonic and sufficient semantic details to answer the following questions:

- What Operands are read/written?
- What registers are implicitly read/written?
- What memory locations are implicitly read/written?
- What are the possible control flow successors of this instruction?

Each Operand is an AST built from RegisterAST and Immediate leaves. For each Operand, you may determine:

- Registers read
- Registers written
- Whether memory is read or written
- Which memory addresses are read or written, given the state of all relevant registers

Instructions should be constructed from an `unsigned char*` pointing to machine language, using the InstructionDecoder class. See InstructionDecoder for more details.

```
Instruction (Operation::Ptr what,
             size_t size,
             const unsigned char * raw,
             Dyninst::Architecture arch)
```

> `what` is the opcode of the instruction, `size` contains the number of bytes occupied by the corresponding machine instruction, and `raw` contains a pointer to the buffer from which this `Instruction` object was decoded. The architecture is specified by `arch`, and may be an element from the following set: `{Arch_x86, Arch_x86_64, Arch_ppc32, Arch_ppc64}` (as defined in `dyn_regs.h`).
>
> Construct an Instruction from an Operation and a collection of Expressions. This method is not intended to be used except by the InstructionDecoder class, which serves as a factory class for producing Instruction objects.

While an Instruction object may be built "by hand" if desired, using the decoding interface ensures that the operation and operands are a sensible combination, and that the size reported is based on the actual size of a legal encoding of the machine instruction represented.

`const Operation & getOperation() const`

Returns the `Operation` used by the `Instruction`. See Section 3.2 for details of the `Operation` interface.

`void getOperands(std::vector<Operand> & operands) const`

The vector operands has the instruction's operands appended to it in the same order that they were decoded.

`Operand getOperand(int index) const`

The `getOperand` method returns the operand at position `index`, or an empty operand if `index` does not correspond to a valid operand in this instruction.

`unsigned char rawByte(unsigned int index) const`

Returns the indexth byte in the instruction.

`size_t size() const`

Returns the size of the corresponding machine instruction, in bytes.

`const void * ptr() const`

Returns a pointer to the raw byte representation of the corresponding machine instruction.

`void getWriteSet(std::set<RegisterAST::Ptr> & regsWritten) const`

Insert the set of registers written by the instruction into `regsWritten`. The list of registers returned in `regsWritten` includes registers that are explicitly written as destination operands (like the destination of a move). It also includes registers that are implicitly written (like the stack pointer in a push or pop instruction). It does not include any registers used only in computing the effective address of a write. `pop` $*$`eax`, for example, writes to `esp`, reads `esp`, and reads `eax`, but despite the fact that $*$`eax` is the destination operand, `eax` is not itself written.

For both the write set and the read set (below), it is possible to determine whether a register is accessed implicitly or explicitly by examining the Operands. An explicitly accessed register appears as an operand that is written or read; also, any registers used in any address calculations are explicitly read. Any element of the write set or read set that is not explicitly written or read is implicitly written or read.

```
void getReadSet(std::set<RegisterAST::Ptr> & regsRead) const
```

Insert the set of registers read by the instruction into `regsRead`.

If an operand is used to compute an effective address, the registers involved are read but not written, regardless of the effect on the operand.

```
bool isRead(Expression::Ptr candidate) const
```

`candidate` is the subexpression to search for among the values read by this `Instruction` object.

Returns `true` if `candidate` is read by this `Instruction`.

```
bool isWritten(Expression::Ptr candidate) const
```

`candidate` is the subexpression to search for among the values written by this `Instruction` object.

Returns `true` if `candidate` is written by this `Instruction`.

```
bool readsMemory() const
```

Returns `true` if the instruction reads at least one memory address as data.

If any operand containing a `Dereference` object is read, the instruction reads the memory at that address. Also, on platforms where a stack pop is guaranteed to read memory, `readsMemory` will return `true` for a pop instruction.

```
bool writesMemory() const
```

Returns `true` if the instruction writes at least one memory address as data.

If any operand containing a `Dereference` object is write, the instruction writes the memory at that address. Also, on platforms where a stack push is guaranteed to write memory, `writesMemory` will return `true` for a pop instruction.

```
void getMemoryReadOperands(std::set<Expression::Ptr> & memAccessors) const
```

Addresses read by this instruction are inserted into `memAccessors`.

The addresses read are in the form of `Expression`s, which may be evaluated once all of the registers that they use have had their values set. Note that this method returns ASTs representing address computations, and not address accesses. For instance, an instruction accessing memory through a register dereference would return an `Expression` tree containing just the register that determines the address being accessed, not a tree representing a dereference of that register.

```
void getMemoryWriteOperands(std::set<Expression::Ptr> & memAccessors) const
```

Addresses written by this instruction are inserted into `memAccessors`.

The addresses written are in the same form as those returned by `getMemoryReadOperands` above.

```
Expression::Ptr getControlFlowTarget() const
```

When called on an explicitly control-flow altering instruction, returns the non-fallthrough control flow destination. When called on any other instruction, returns `NULL`.

For direct absolute branch instructions, `getControlFlowTarget` will return an immediate value. For direct relative branch instructions, `getControlFlowTarget` will return the expression `PC` + offset. In the case of indirect branches and calls, it returns a dereference of a register (or possibly a dereference of a more complicated expression). In this case, data flow analysis will often allow the determination of the possible targets of the instruction. We do not do analysis beyond the single-instruction level in the Instruction API; if other code performs this type of analysis, it may update the information in the Dereference object using the setValue method in the Expression interface. More details about this may be found in Section 3.5 and Section 3.11.

Returns an `Expression` evaluating to the non-fallthrough control targets, if any, of this instruction.

```
bool allowsFallThrough() const
```

Returns `false` if control flow will unconditionally go to the result of `getControlFlowTarget` after executing this instruction.

```
std::string format(Address addr = 0)
```

Returns the instruction as a string of assembly language. If `addr` is specified, the value of the program counter as used by the instruction (e.g., a branch) is set to `addr`.

```
bool isValid() const
```

Returns `true` if this `Instruction` object is valid. Invalid instructions indicate than an `InstructionDecoder` has reached the end of its assigned range, and that decoding should terminate.

```
bool isLegalInsn() const
```

Returns `true` if this Instruction object represents a legal instruction, as specified by the architecture used to decode this instruction.

```
Architecture getArch() const
```

Returns the architecture containing the instruction. As above, this will be an element from the set `{Arch_x86, Arch_x86_64, Arch_ppc32, Arch_ppc64}`.

```
InsnCategory getCategory() const
```

Alpha: Returns the category into which an instruction falls. This feature is presently incomplete, and we welcome feedback on ways to extend it usefully.

Currently, the valid categories are `c_CallInsn`, `c_ReturnInsn`, `c_BranchInsn`, `c_CompareInsn`, and `c_NoCategory`, as defined in `InstructionCategories.h`.

```
struct CFT {
  Expression::Ptr target;
  bool isCall;
  bool isIndirect;
  bool isConditional;
  bool isFallthrough;
}

typedef ... cftConstIter;
cftConstIter cft_begin() const;
cftConstIter cft_end() const;
```

On certain platforms (e.g., PowerPC with conditional call/return instructions) the `getControlFlowTarget` function is insufficient to represent the successors of an instruction. The `cft_begin` and `cft_end` functions return iterators into a list of all control flow target expressions as represented by a list of `CFT` structures. In most cases, `getControlFlowTarget` suffices.

3.2 Operation Class

An Operation object represents a family of opcodes (operation encodings) that perform the same task (e.g. the `MOV` family). It includes information about the number of operands, their read/write semantics, the implicit register reads and writes, and the control flow behavior of a particular assembly language operation. It additionally provides access to the assembly mnemonic, which allows any semantic details that are not encoded in the Instruction representation to be added by higher layers of analysis.

As an example, the `CMP` operation on IA32/AMD64 processors has the following properties:

- Operand 1 is read, but not written
- Operand 2 is read, but not written
- The following flags are written:
  - Overflow
  - Sign
  - Zero
  - Parity
  - Carry
  - Auxiliary
- No other registers are read, and no implicit memory operations are performed

Operations are constructed by the `InstructionDecoder` as part of the process of constructing an Instruction.

`const Operation::registerSet & implicitReads () const`

Returns the set of registers implicitly read (i.e. those not included in the operands, but read anyway).

`const Operation::registerSet & implicitWrites () const`

Returns the set of registers implicitly written (i.e. those not included in the operands, but written anyway).

`std::string format() const`

Returns the mnemonic for the operation. Like `instruction::format`, this is exposed for debugging and will be replaced with stream operators in the public interface.

`entryID getID() const`

Returns the entry ID corresponding to this operation. Entry IDs are enumerated values that correspond to assembly mnemonics.

`prefixEntryID getPrefixID() const`

Returns the prefix entry ID corresponding to this operation, if any. Prefix IDs are enumerated values that correspond to assembly prefix mnemonics.

`bool isRead(Expression::Ptr candidate) const`

Returns `true` if the expression represented by `candidate` is read implicitly.

`bool isWritten(Expression::Ptr candidate) const`

Returns `true` if the expression represented by `candidate` is written implicitly.

`const Operation::VCSet & getImplicitMemReads() const`

Returns the set of memory locations implicitly read.

`const Operation::VCSet & getImplicitMemWrites() const`

Returns the set of memory locations implicitly write.

3.3 Operand Class

An Operand object contains an AST built from RegisterAST and Immediate leaves, and information about whether the Operand is read, written, or both. This allows us to determine which of the registers that appear in the Operand are read and which are written, as well as whether any memory accesses are reads, writes, or both. An Operand, given full knowledge of the values of the leaves of the AST, and knowledge of the logic associated with the tree's internal nodes, can determine the result of any computations that are encoded in it. It will rarely be the case that an Instruction is built with its Operands' state fully specified. This mechanism is instead intended to allow a user to fill in knowledge about the state of the processor at the time the Instruction is executed.

`Operand(Expression::Ptr val, bool read, bool written)`

Create an operand from an `Expression` and flags describing whether the ValueComputation is read, written, or both.

`val` is a reference-counted pointer to the `Expression` that will be contained in the `Operand` being constructed. `read` is true if this operand is read. `written` is true if this operand is written.

`void getReadSet(std::set<RegisterAST::Ptr> & regsRead) const`

Get the registers read by this operand. The registers read are inserted into `regsRead`.

`void getWriteSet(std::set<RegisterAST::Ptr> & regsWritten) const`

Get the registers written by this operand. The registers written are inserted into `regsWritten`.

`bool isRead() const`

Returns `true` if this operand is read.

`bool isWritten() const`

Returns `true` if this operand is written.

`bool isRead(Expression::Ptr candidate) const`

Returns `true` if `candidate` is read by this operand.

`bool isWritten(Expression::Ptr candidate) const`

Returns `true` if `candidate` is written to by this operand.

`bool readsMemory() const`

Returns `true` if this operand reads memory.

`bool writesMemory() const`

Returns `true` if this operand writes memory.

`void addEffectiveReadAddresses(std::set<Expression::Ptr> & memAccessors) const`

If `Operand` is a memory read operand, insert the `ExpressionPtr` representing the address being read into `memAccessors`.

`void addEffectiveWriteAddresses(std::set<Expression::Ptr> & memAccessors) const`

If `Operand` is a memory write operand, insert the `ExpressionPtr` representing the address being written into `memAccessors`.

`std::string format(Architecture arch, Address addr = 0) const`

Return a printable string representation of the operand. The `arch` parameter must be supplied, as operands do not record their architectures. The optional `addr` parameter specifies the value of the program counter.

`Expression::Ptr getValue() const`

The `getValue` method returns an `ExpressionPtr` to the AST contained by the operand.

3.4 InstructionAST Class

The InstructionAST class is the base class for all nodes in the ASTs used by the Operand class. It defines the necessary interfaces for traversing and searching an abstract syntax tree representing an operand. For the purposes of searching an InstructionAST, we provide two related interfaces. The first, `getUses`, will return the registers that appear in a given tree. The second, `isUsed`, will take as input another tree and return true if that tree is a (not necessarily proper) subtree of this one. `isUsed` requires us to define an equality relation on these abstract syntax trees, and the equality operator is provided by the InstructionAST, with the details implemented by the classes derived from InstructionAST. Two AST nodes are equal if the following conditions hold:

- They are of the same type
- If leaf nodes, they represent the same immediate value or the same register
- If non-leaf nodes, they represent the same operation and their corresponding children are equal

```
typedef boost::shared_ptr<InstructionAST> Ptr
```

A type definition for a reference-counted pointer to an `InstructionAST`.

```
bool operator==(const InstructionAST &rhs) const
```

Compare two AST nodes for equality.

Non-leaf nodes are equal if they are of the same type and their children are equal. `RegisterAST`s are equal if they represent the same register. `Immediate`s are equal if they represent the same value. Note that it is not safe to compare two `InstructionAST::Ptr` variables, as those are pointers. Instead, test the underlying `InstructionAST` objects.

```
virtual void getChildren(vector<InstructionAPI::Ptr> & children) const
```

Children of this node are appended to the vector `children`.

```
virtual void getUses(set<InstructionAPI::Ptr> & uses)
```

The use set of an `InstructionAST` is defined as follows:

- A `RegisterAST` uses itself
- A `BinaryFunction` uses the use sets of its children
- A `Immediate` uses nothing
- A `Dereference uses the use set of its child`

The use set oft his node is appended to the vector `uses`.

```
virtual bool isUsed(InstructionAPI::Ptr findMe) const
```

Unlike `getUses`, `isUsed` looks for `findMe` as a subtree of the current tree. `getUses` is designed to return a minimal set of registers used in this tree, whereas `isUsed` is designed to allow searches for arbitrary subexpressions. `findMe` is the AST node to find in the use set of this node.

Returns `true` if `findMe` is used by this AST node.

```
virtual std::string format(formatStyle how == defaultStyle) const
```

The `format` interface returns the contents of an `InstructionAPI` object as a string. By default, `format` produces assembly language.

3.5 Expression Class

An `Expression` is an AST representation of how the value of an operand is computed.

The `Expression` class extends the `InstructionAST` class by adding the concept of evaluation to the nodes of an `InstructionAST`. Evaluation attempts to determine the `Result` of the computation that the AST being evaluated represents. It will fill in results of as many of the nodes in the tree as possible, and if full evaluation is possible, it will return the result of the computation performed by the tree.

Permissible leaf nodes of an `Expression` tree are RegisterAST and Immediate objects. Permissible internal nodes are `BinaryFunction` and Dereference objects. An `Expression` may represent an immediate value, the contents of a register, or the contents of memory at a given address, interpreted as a particular type.

The `Result`s in an `Expression` tree contain a type and a value. Their values may be an undefined value or an instance of their associated type. When two `Result`s are combined using a `BinaryFunction`, the `BinaryFunction` specifies the output type. Sign extension, type promotion, truncation, and all other necessary conversions are handled automatically based on the input types and the output type. If both of the `Result`s that are combined have defined values, the combination will also have a defined value; otherwise, the combination's value will be undefined. For more information, see Section 3.7, Section 3.10, and Section 3.11.

A user may specify the result of evaluating a given `Expression`. This mechanism is designed to allow the user to provide a Dereference or RegisterAST with information about the state of memory or registers. It may additionally be used to change the value of an Immediate or to specify the result of a `BinaryFunction`. This mechanism may be used to support other advanced analyses.

In order to make it more convenient to specify the results of particular subexpressions, the `bind` method is provided. `bind` allows the user to specify that a given subexpression has a particular value everywhere that it appears in an expression. For example, if the state of certain registers is known at the time an instruction is executed, a user can `bind` those registers to their known values throughout an `Expression`.

The evaluation mechanism, as mentioned above, will evaluate as many sub-expressions of an expression as possible. Any operand that is more complicated than a single immediate value, however, will depend on register or memory values. The `Result`s of evaluating each subexpression are cached automatically using the `setValue` mechanism. The `Expression` then attempts to determine its `Result` based on the `Result`s of its children. If this `Result` can be determined (most likely because register contents have been filled in via

setValue or bind), it will be returned from eval; if it can not be determined, a Result with an undefined value will be returned. See Figure 6 for an illustration of this concept; the operand represented is [EBX + 4 * EAX]. The contents of EBX and EAX have been determined through some outside mechanism, and have been defined with setValue. The eval mechanism proceeds to determine the address being read by the Dereference, since this information can be determined given the contents of the registers. This address is available from the Dereference through its child in the tree, even though calling eval on the Dereference returns a Result with an undefined value.

```
typedef boost::shared_ptr<Expression> Ptr
```

A type definition for a reference-counted pointer to an Expression.

```
const Result & eval() const
```

If the Expression can be evaluated, returns a Result containing its value. Otherwise returns an undefined Result.

```
const setValue(const Result & knownValue)
```

Sets the result of eval for this Expression to knownValue.

```
void clearValue()
```

clearValue sets the contents of this Expression to undefined. The next time eval is called, it will recalculate the value of the Expression.

```
int size() const
```

size returns the size of this Expression's Result, in bytes.

```
bool bind(Expression * expr, const Result & value)
```

bind searches for all instances of the Expression expr within this Expression, and sets the result of eval for those subexpressions to value. bind returns true if at least one instance of expr was found in this Expression.

bind does not operate on subexpressions that happen to evaluate to the same value. For example, if a dereference of 0xDEADBEEF is bound to 0, and a register is bound to 0xDEADBEEF, a deference of that register is not bound to 0.

```
virtual void apply(Visitor *)
```

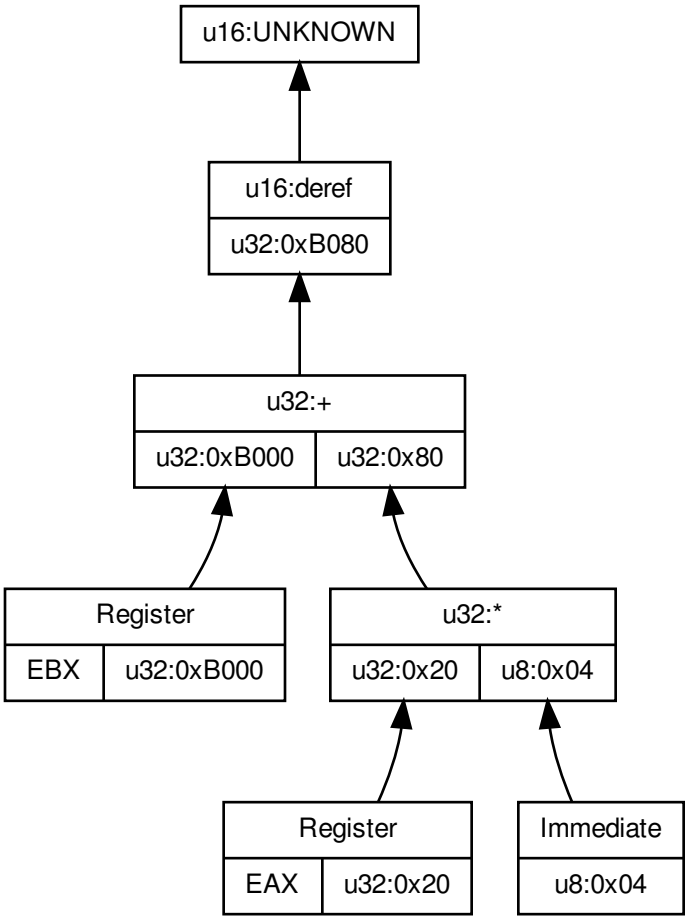


Figure 6: Applying `eval` to a Dereference tree with two registers having user-provided values.

`apply` applies a `Visitor` to this `Expression`. Visitors perform postfix-order traversal of the ASTs represented by an `Expression`, with user-defined actions performed at each node of the tree. We present a thorough discussion with examples in Section 3.6.

```
virtual void getChildren(std::vector<Expression::Ptr> & children) const
```

`getChildren` may be called on an `Expression` taking a vector of `ExpressionPtr`s, rather than `InstructionAST`Ptrs. All children which are `Expression`s will be appended to `children`.

3.6 Visitor Paradigm

An alternative to the bind/eval mechanism is to use a *visitor* [1] over an expression tree. The visitor concept applies a user-specified visitor class to all nodes in an expression tree (in a post-order traversal). The visitor paradigm can be used as a more efficient replacement for bind/eval, to identify whether an expression has a desired pattern, or to locate children of an expression tree.

A visitor is a user-defined class that inherits from the `Visitor` class defined in `Visitor.h`. That class is repeated here for reference:

```
class Visitor {
  public:
    Visitor() {}
    virtual ~Visitor() {}
    virtual void visit(BinaryFunction* b) = 0;
    virtual void visit(Immediate* i) = 0;
    virtual void visit(RegisterAST* r) = 0;
    virtual void visit(Dereference* d) = 0;
};
```

A user provides implementations of the four `visit` methods. When applied to an `Expression` (via the `Expression::apply` method) the InstructionAPI will perform a post-order traversal of the tree, calling the appropriate `visit` method at each node.

As a simple example, the following code prints out the name of each register used in an `Expression`:

```
#include "Instruction.h"
#include "Operand.h"
#include "Expression.h"
#include "Register.h"
#include "Visitor.h"
#include <iostream>

using namespace std;
using namespace Dyninst;
using namespace InstructionAPI;
```

[1] From *Elements of Reusable Object-Oriented Software* by Gamma, Helm, Johnson, and Vlissides

```
class PrintVisitor : public Visitor {
  public:
    PrintVisitor() {};
    ~PrintVisitor() {};
    virtual void visit(BinaryFunction* b) {};
    virtual void visit(Immediate* i) {};
    virtual void visit(RegisterAST* r) {
      cout << "\tVisiting register " << r->getID().name() << endl;
    }
    virtual void visit(Dereference* d) {};
};

void printRegisters(Instruction::Ptr insn) {
   PrintVisitor pv;
   std::vector<Operand> operands;
   insn->getOperands(operands);
   // c++11x allows auto to determine the type of a variable;
   // if not using c++11x, use 'std::vector<Operand>::iterator' instead.
   // For gcc, use the -std=c++0x argument.
   for (auto iter = operands.begin(); iter != operands.end(); ++iter) {
      cout << "Registers used for operand" << endl;
      (*iter).getValue()->apply(&pv);
   }
}
```

Visitors may also set and use internal state. For example, the following visitor (presented without surrounding use code) matches x86 and x86-64 instructions that add 0 to a register (effectively a noop).

```
class nopVisitor : public Visitor
{
  public:
   nopVisitor() : foundReg(false), foundImm(false), foundBin(false), isNop(true) {}
   virtual ~nopVisitor() {}

   bool foundReg;
   bool foundImm;
   bool foundBin;
   bool isNop;

   virtual void visit(BinaryFunction*)
   {
      if (foundBin) isNop = false;
      if (!foundImm) isNop = false;
      if (!foundReg) isNop = false;
      foundBin = true;
   }
   virtual void visit(Immediate *imm)
   {
      if (imm != 0) isNop = false;
      foundImm = true;
   }
```

```
    virtual void visit(RegisterAST *)
    {
        foundReg = true;
    }
    virtual void visit(Dereference *)
    {
        isNop = false;
    }
};
```

3.7 Result Class

A `Result` object represents a value computed by an `Expression` AST.

The `Result` class is a tagged-union representation of the results that Expressions can produce. It includes 8, 16, 32, 48, and 64 bit integers (signed and unsigned), bit values, and single and double precision floating point values. For each of these types, the value of a Result may be undefined, or it may be a value within the range of the type.

The `type` field is an enum that may contain any of the following values:

- `u8:` an unsigned 8-bit integer
- `s8:` a signed 8-bit integer
- `u16:` an unsigned 16-bit integer
- `s16:` a signed 16-bit integer
- `u32:` an unsigned 32-bit integer
- `s32:` a signed 32-bit integer
- `u48:` an unsigned 48-bit integer (IA32 pointers)
- `s48:` a signed 48-bit integer (IA32 pointers)
- `u64:` an unsigned 64-bit integer
- `s64:` a signed 64-bit integer
- `sp_float:` a single-precision float
- `dp_float:` a double-precision float
- `bit_flag:` a single bit (individual flags)
- `m512:` a 512-bit memory value
- `dbl128:` a 128-bit integer, which often contains packed floating point values - `m14:` a 14 byte memory value

`Result (Result_Type t)`

A `Result` may be constructed from a type without providing a value. This constructor creates a `Result` of type `t` with undefined contents.

`Result (Result_Type t, T v)`

A `Result` may be constructed from a type and any value convertible to the type that the tag represents. This constructor creates a `Result` of type `t` and contents `v` for any `v` that is implicitly convertible to type `t`. Attempting to construct a `Result` with a value that is incompatible with its type will result in a compile-time error.

`bool operator== (const Result & o) const`

Two `Result`s are equal if any of the following hold:

- Both `Result`s are of the same type and undefined
- Both `Result`s are of the same type, defined, and have the same value

Otherwise, they are unequal (due to having different types, an undefiend `Result` compared to a defined `Result`, or different values).

`std::string format () const`

`Result`s are formatted as strings containing their contents, represented as hexadecimal. The type of the `Result` is not included in the output.

```
template <typename to_type>
to_type convert() const
```

Converts the `Result` to the desired datatype. For example, to convert a `Result res` to a signed char, use `res.convert<char>()`; to convert it to an unsigned long, use `res.convert<unsigned long>()`.

`int size () const`

Returns the size of the contained type, in bytes.

3.8 RegisterAST Class

A `RegisterAST` object represents a register contained in an operand. As a `RegisterAST` is an `Expression`, it may contain the physical register's contents if they are known.

`typedef dyn\_detail::boost::shared\_ptr<RegisterAST> Ptr`

A type definition for a reference-counted pointer to a `RegisterAST`.

`RegisterAST (MachRegister r)`

Construct a register using the provided register object `r`. The `MachRegister` datatype is Dyninst's register representation and should not be constructed manually.

`void getChildren (vector< InstructionAST::Ptr > & children) const`

By definition, a `RegisterAST` object has no children. Since a `RegisterAST` has no children, the `children` parameter is unchanged by this method.

`void getUses (set< InstructionAST::Ptr > & uses)`

By definition, the use set of a `RegisterAST` object is itself. This `RegisterAST` will be inserted into `uses`.

`bool isUsed (InstructionAST::Ptr findMe) const`

`isUsed` returns `true` if `findMe` is a `RegisterAST` that represents the same register as this `RegisterAST`, and `false` otherwise.

`std::string format (formatStyle how = defaultStyle) const`

The format method on a `RegisterAST` object returns the name associated with its ID.

`RegisterAST makePC (Dyninst::Architecture arch) [static]`

Utility function to get a `Register` object that represents the program counter. `makePC` is provided to support platform-independent control flow analysis.

`bool operator< (const RegisterAST & rhs) const`

We define a partial ordering on registers by their register number so that they may be placed into sets or other sorted containers.

`MachRegister getID () const`

The `getID` function returns underlying register represented by this AST.

`RegisterAST::Ptr promote (const InstructionAST::Ptr reg) [static]`

Utility function to hide aliasing complexity on platforms (IA-32) that allow addressing part or all of a register

3.9 Immediate Class

The Immediate class represents an immediate value in an operand.

Since an Immediate represents a constant value, the `setValue` and `clearValue` interface are disabled on Immediate objects. If an immediate value is being modified, a new Immediate object should be created to represent the new value.

```
virtual bool isUsed(InstructionAST::Ptr findMe) const
```

```
void getChildren(vector<InstructionAST::Ptr> &) const
```

By definition, an `Immediate` has no children.

```
void getUses(set<InstructionAST::Ptr> &)
```

By definition, an `Immediate` uses no registers.

```
bool isUsed(InstructionAPI::Ptr findMe) const
```

`isUsed`, when called on an Immediate, will return true if `findMe` represents an Immediate with the same value. While this convention may seem arbitrary, it allows `isUsed` to follow a natural rule: an `InstructionAST` is used by another `InstructionAST` if and only if the first `InstructionAST` is a subtree of the second one.

3.10 BinaryFunction Class

A `BinaryFunction` object represents a function that can combine two `Expression`s and produce another `ValueComputation`.

For the purposes of representing a single operand of an instruction, the `BinaryFunction`s of interest are addition and multiplication of integer values; this allows an `Expression` to represent all addressing modes on the architectures currently supported by the Instruction API.

```
BinaryFunction(Expression::Ptr arg1,
               Expression::Ptr arg2,
               Result_Type result_type,
               funcT:Ptr func)
```

The constructor for a `BinaryFunction` may take a reference-counted pointer or a plain C++ pointer to each of the child `Expression`s that represent its arguments. Since the reference-counted implementation requires explicit construction, we provide overloads for all four combinations of

plain and reference-counted pointers. Note that regardless of which constructor is used, the pointers `arg1` and `arg2` become owned by the `BinaryFunction` being constructed, and should not be deleted. They will be cleaned up when the `BinaryFunction` object is destroyed.

The `func` parameter is a binary functor on two `Result`s. It should be derived from `funcT`. `addResult` and `multResult`, which respectively add and multiply two `Result`s, are provided as part of the InstructionAPI, as they are necessary for representing address calculations. Other `funcT`s may be implemented by the user if desired. `funcT`s have names associated with them for output and debugging purposes. The addition and multiplication functors provided with the Instruction API are named "+" and "*", respectively.

```
const Result & eval () const
```

The `BinaryFunction` version of `eval` allows the `eval` mechanism to handle complex addressing modes. Like all of the `ValueComputation` implementations, a `BinaryFunction`'s `eval` will return the result of evaluating the expression it represents if possible, or an empty `Result` otherwise. A `BinaryFunction` may have arguments that can be evaluated, or arguments that cannot. Additionally, it may have a real function pointer, or it may have a null function pointer. If the arguments can be evaluated and the function pointer is real, a result other than an empty `Result` is guaranteed to be returned. This result is cached after its initial calculation; the caching mechanism also allows outside information to override the results of the `BinaryFunction`'s internal computation. If the cached result exists, it is guaranteed to be returned even if the arguments or the function are not evaluable.

```
void getChildren (vector< InstructionAST::Ptr > & children) const
```

The children of a `BinaryFunction` are its two arguments. Appends the children of this BinaryFunction to `children`.

```
void getUses (set< InstructionAST::Ptr > & uses)
```

The use set of a `BinaryFunction` is the union of the use sets of its children. Appends the use set of this `BinaryFunction` to `uses`.

```
bool isUsed (InstructionAST::Ptr findMe) const
```

`isUsed` returns `true` if `findMe` is an argument of this `BinaryFunction`, or if it is in the use set of either argument.

3.11 Dereference Class

A `Dereference` object is an `Expression` that dereferences another `ValueComputation`.

A `Dereference` contains an `Expression` representing an effective address computation. Its use set is the same as the use set of the `Expression` being dereferenced.

It is not possible, given the information in a single instruction, to evaluate the result of a dereference. `eval` may still be called on an `Expression` that includes dereferences, but the expected use case is as follows:

- Determine the address being used in a dereference via the `eval` mechanism
- Perform analysis to determine the contents of that address
- If necessary, fill in the `Dereference` node with the contents of that addresss, using `setValue`

The type associated with a `Dereference` node will be the type of the value *read from memory*, not the type used for the address computation. Two `Dereference`s that access the same address but interpret the contents of that memory as different types will produce different values. The children of a `Dereference` at a given address are identical, regardless of the type of dereference being performed at that address. For example, the `Expression` shown in Figure 6 could have its root `Dereference`, which interprets the memory being dereferenced as a unsigned 16-bit integer, replaced with a `Dereference` that interprets the memory being dereferenced as any other type. The remainder of the `Expression` tree would, however, remain unchanged.

```
Dereference (Expression::Ptr addr, Result_Type result_type)
```

A `Dereference` is constructed from an `Expression` pointer (raw or shared) representing the address to be dereferenced and a type indicating how the memory at the address in question is to be interpreted.

```
virtual void getChildren (vector< InstructionAST::Ptr > & children) const
```

A `Dereference` has one child, which represents the address being dereferenced. Appends the child of this `Dereference` to `children`.

```
virtual void getUses (set< InstructionAST::Ptr > & uses)
```

The use set of a `Dereference` is the same as the use set of its children. The use set of this `Dereference` is inserted into `uses`.

```
virtual bool isUsed (InstructionAST::Ptr findMe) const
```

An `InstructionAST` is used by a `Dereference` if it is equivalent to the `Dereference` or it is used by the lone child of the `Dereference`

3.12 InstructionDecoder Class

The `InstructionDecoder` class decodes instructions, given a buffer of bytes and a length, and constructs an Instruction.

```
InstructionDecoder(const unsigned char *buffer, size_t size,
                   Architecture arch)
InstructionDecoder(const void *buffer, size_t size,
                   Architecture arch)
```

Construct an `InstructionDecoder` over the provided `buffer` and `size`. We consider the buffer to contain instructions from the provided `arch`, which must be from the set `{Arch_x86, Arch_x86_64, Arch_ppc32, Arch_ppc64}`.

```
Instruction::Ptr decode();
```

Decode the next address in the buffer provided at construction time, returning either an instruction pointer or `NULL` if the buffer contains no undecoded instructions.

Paradyn Parallel Performance Tools

StackwalkerAPI Programmer's Guide

9.2 Release

June 2016

Computer Sciences Department
University of Wisconsin–Madison
Madison, WI 53706

Computer Science Department
University of Maryland
College Park, MD 20742

Email dyninst-api@cs.wisc.edu
Web www.dyninst.org

Contents

1 Introduction

This document describes StackwalkerAPI, an API and library for walking a call stack. The call stack (also known as the run-time stack) is a stack found in a process that contains the currently active stack frames. Each stack frame is a record of an executing function (or function-like object such as a signal handler or system call). StackwalkerAPI provides an API that allows users to collect a call stack (known as walking the call stack) and access information about its stack frames. The current implementation supports Linux/x86, Linux/x86-64, Linux/Power, Linux/Power-64, Windows/x86, BlueGene/L, and BlueGene/P.

StackwalkerAPI is designed to be both easy-to-use and easy-to-extend. Users can easily use StackwalkerAPI to walk a call stack without needing to understand how call stacks are laid out on their platform. Users can easily extend StackwalkerAPI to work with new platforms and types of stack frames by implementing a set of callbacks that can be plugged into StackwalkerAPI.

StackwalkerAPI's ease-of-use comes from it providing a platform independent interface that allows users to access detailed information about the call stack. For example, the following C++ code-snippet is all that is needed to walk and print the call stack of the currently running thread.

```
std::vector<Frame> stackwalk;
string s;

Walker *walker = Walker::newWalker();
walker->walkStack(stackwalk);
for (unsigned i=0; i<stackwalk.size(); i++) {
                stackwalk[i].getName(s);
                cout << "Found function " << s << endl;
}
```

StackwalkerAPI can walk a call stack in the same address space as where the StackwalkerAPI library lives (known as a first party stackwalk), or it can walk a call stack in another process (known as a third party stackwalk). To change the above example to perform a third party stackwalk, we would only need to pass a process identifier to newWalker, e.g:

```
Walker *walker = Walker::newWalker(pid);
```

Our other design goal with StackwalkerAPI is to make it easy-to-extend. The mechanics of how to walk through a stack frame can vary between different platforms, and even between different types of stack frames on the same platform. In addition, different platforms may have different mechanisms for reading the data in a call stack or looking up symbolic names that go with a stack frame. StackwalkerAPI provides a callback interface for plugging in mechanisms for handling new systems and types of stack frames. The callback interface can be used to port StackwalkerAPI to new platforms, extend StackwalkerAPI support on existing systems, or more easily integrate StackwalkerAPI into existing tools. There are callbacks for the following StackwalkerAPI operations:

Walk through a stack frame StackwalkerAPI will find different types of stack frames on different platforms and even within the same platform. For example, on Linux/x86 the stack frame generated by a typical function looks different from the stack frame generated by a signal

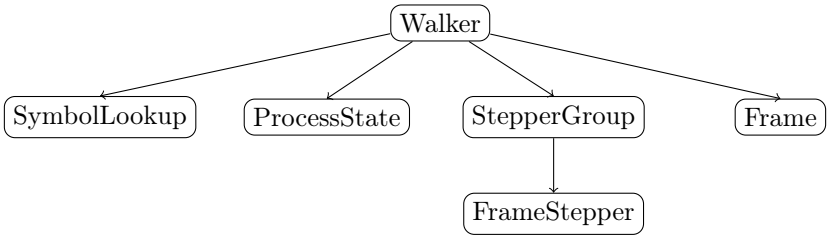


Figure 1: Object Ownership

handler. The callback interface can be used to register a handler with StackwalkerAPI that knows how to walk through a new type of stack frame. For example, the DyninstAPI tool registers an object with StackwalkerAPI that describes how to walk through the stack frames generated by its instrumentation.

Access process data To walk a call stack, StackwalkerAPI needs to be able to read a process' memory and registers. When doing a first party stackwalk, this is done by directly reading them from the current address space. When doing a third party stackwalk, this is done by reading them using a debugger interface. The callback interface can be used to register new objects for accessing process data. This can be used, for example, to port StackwalkerAPI to a new operating system or make it work with a new debugger interface.

Look up symbolic names When StackwalkerAPI finds a stack frame, it gets an address that points into the piece of code that created that stack frame. This address is not necessarily meaningful to a user, so StackwalkerAPI attempts to associate the address with a symbolic name. The callback interface can be used to register an object with StackwalkerAPI that performs an address to name mapping, allowing StackwalkerAPI to associate names with stack frames.

2 Abstractions

StackwalkerAPI contains two interfaces: the Stackwalking Interface and the Callback Interface. The stackwalking interface is used to walk the call stack, query information about stack frames, and collect basic information about threads. The Callback Interface is used to provide custom mechanisms for walking a call stack. Users who operate in one of StackwalkerAPI's standard configurations do not need to use the Callback Interface.

Figure 1 shows the ownership hierarchy for StackwalkerAPI's classes. Ownership is a "contains" relationship; if one class owns another, then instances of the owner class maintain an exclusive instance of the other. For example, in Figure 1 the each Walker instance contains exactly one instance of a ProcessState object. No other instance of Walker uses that instance of ProcessState.

This remainder of this section briefly describes the six classes that make up StackwalkerAPI's two interfaces. For more details, see the class descriptions in Section 3.

2.1 Stackwalking Interface

Walker The Walker class is the top-level class used for collecting stackwalks. It provides a simple interface for requesting a stackwalk. Each Walker object is associated with one process, but may walk the call stacks of multiple threads within that process.

Frame A call stack is returned as a vector of Frame objects, where each Frame object represents a stack frame. It can provide information about the stack frame and basic information about the function, signal handler or other mechanism that created it. Users can request information such as the symbolic name associated with the Frame object, and values of its saved registers.

2.2 Callback Interface

StackwalkerAPI includes default implementations of the Callback Interface on each of its supported platforms. These default implementations allow StackwalkerAPI to work "out of the box" in a standard configuration on each platform. Users can port StackwalkerAPI to new platforms or customize its call stack walking behavior by implementing their own versions of the classes in the Callback Interface.

FrameStepper A FrameStepper object describes how to walk through a single type of stack frame. Users can provide an implementation of this interface that allows StackwalkerAPI to walk through new types of stack frames. For example, the DyninstAPI uses this interface to extend StackwalkerAPI to allow it to walk through stack frames created by instrumentation code.

StepperGroup A StepperGroup is a collection of FrameStepper objects and criteria that describes when to use each type of FrameStepper. These criteria are based on simple address ranges in the code space of the target process. In the above example with DyninstAPI, it would be the job of the StepperGroup to identify a stack frame as belonging to instrumentation code and use the instrumentation FrameStepper to walk through it.

ProcessState A ProcessState interface describes how to access data in the target process. To walk a call stack, StackwalkerAPI needs to access both registers and memory in the target process; ProcessState provides an interface that StackwalkerAPI can use to access that information. StackwalkerAPI includes two default implementation of ProcessState for each platform: one to collect a first party stackwalk in the current process, and one that uses a debugger interface to collect a third party stackwalk in another process.

SymbolLookup The SymbolLookup interface is used to associate a symbolic name with a stack frame. A stackwalk returns a collection of addresses in the code space of a binary. This class uses the binary's symbol table to map those addresses into symbolic names. A default implementation of this class, which uses the DynSymtab package, is provided with StackwalkerAPI. A user could, for example, use this interface to allow StackwalkerAPI to use libelf to look up symbol names instead.

3 API Reference

This section describes the StackwalkerAPI interface. It is divided into three sub-sections: a description of the definitions and basic types used by this API, a description of the interface for collecting stackwalks, and a description of the callback interface.

3.1 Definitions and Basic Types

The following definitions and basic types are referenced throughout the rest of this manual.

3.1.1 Definitions

Stack Frame A stack frame is a record of a function (or function-like object) invocation. When a function is executed, it may create a frame on the call stack. StackwalkerAPI finds stack frames and returns a description of them when it walks a call stack. The following three definitions deal with stack frames.

Bottom of the Stack The bottom of the stack is the earliest stack frame in a call stack, usually a thread's initial function. The stack grows from bottom to the top.

Top of the Stack The top of the stack is the most recent stack frame in a call stack. The stack frame at the top of the stack is for the currently executing function.

Frame Object A Frame object is StackwalkerAPI's representation of a stack frame. A Frame object is a snapshot of a stack frame at a specific point in time. Even if a stack frame changes as a process executes, a Frame object will remain the same. Each Frame object is represented by an instance of the Frame class.

The following three definitions deal with fields in a Frame object.

SP (Stack Pointer) A Frame object's SP member points to the top of its stack frame (a stack frame grows from bottom to top, similar to a call stack). The Frame object for the top of the stack has a SP that is equal to the value in the stack pointer register at the time the Frame object was created. The Frame object for any other stack frame has a SP that is equal to the top address in the stack frame.

FP (Frame Pointer) A Frame object's FP member points to the beginning (or bottom) of its stack frame. The Frame object for the top of the stack has a FP that is equal to the value in the frame pointer register at the time the Frame object was created. The Frame object for any other stack frame has a FP that is equal to the beginning of the stack frame.

RA (Return Address) A Frame object's RA member points to the location in the code space where control will resume when the function that created the stack frame resumes. The Frame object for the top of the stack has a RA that is equal to the value in the program counter register at the time the Frame object was created. The Frame object for any other stack frame has a RA that is found when walking a call stack.

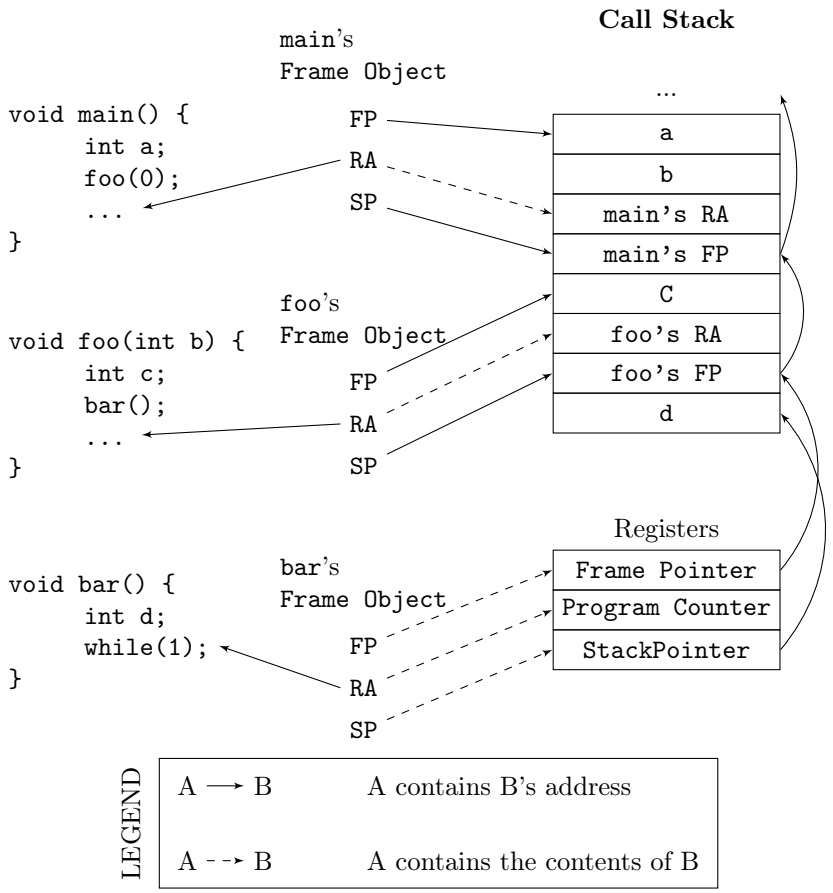


Figure 2: Stack Frame and Frame Object Layout (x86 Architecture)

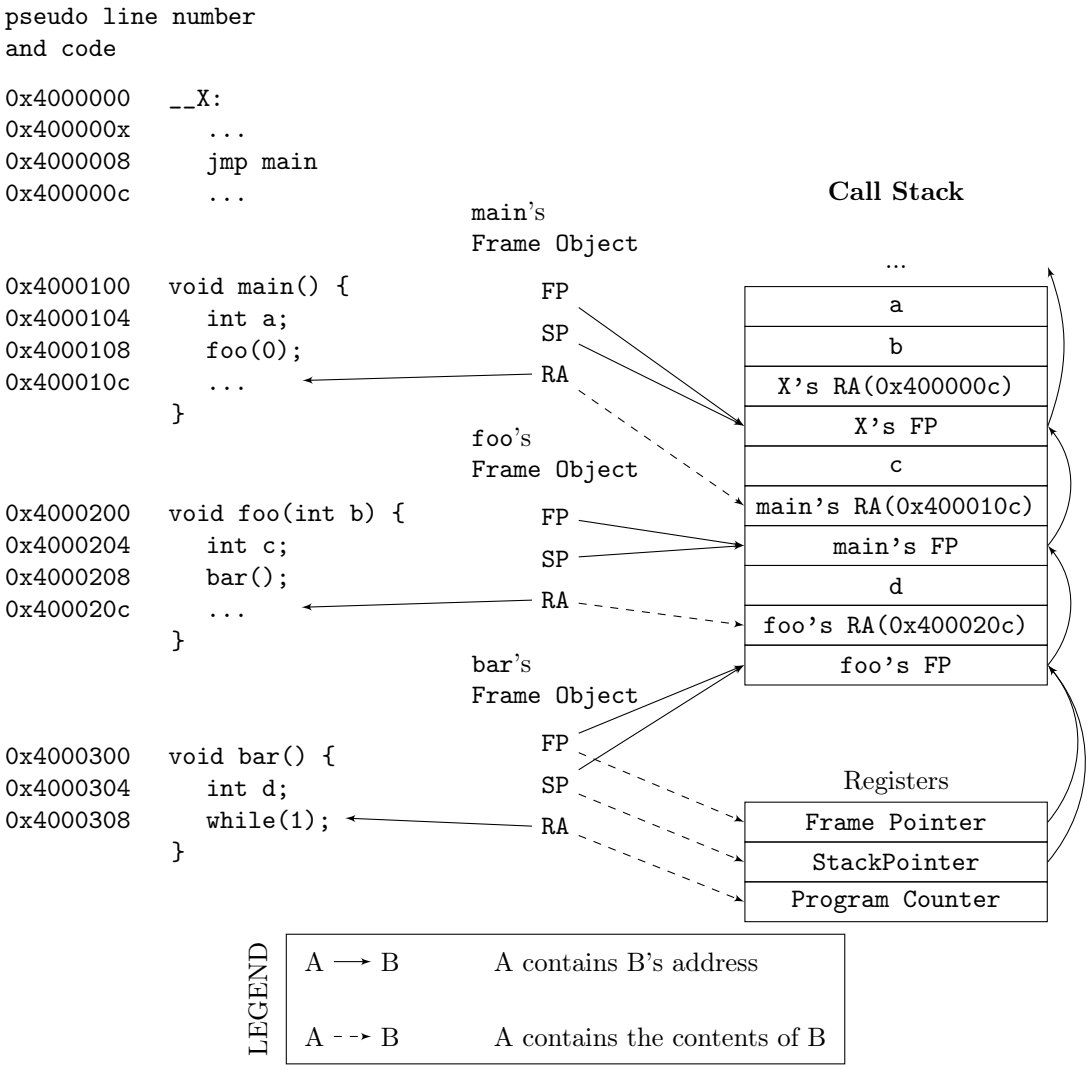


Figure 3: Stack Frame and Frame Object Layout (ARMv8 Architecture)

Figure 2 shows the relationship between application code, stack frames, and Frame objects. In the figure, the source code on the left has run through the main and foo functions, and into the bar function. It has created the call stack in the center, which is shown as a sequence of words growing down. The current values of the processor registers, while executing in bar, are shown below the call stack. When StackwalkerAPI walks the call stack, it creates the Frame objects shown on the right. Each Frame object corresponds to one of the stack frames found in the call stack or application registers.

The call stack in Figure 2 is similar to one that would be found on the x86 architecture. Details about how the call stack is laid out may be different on other architectures, but the meanings of the FP, SP, and RA fields in the Frame objects will remain the same. The layout of the ARM64 stack may be found in Figure 3 as an example of the scope of architectural variations.

The following four definitions deal with processes involved in StackwalkerAPI.

Target Process The process from which StackwalkerAPI is collecting stackwalks.

Host Process The process in which StackwalkerAPI code is currently running.

First Party Stackwalk StackwalkerAPI collects first party stackwalk when it walks a call stack in the same address space it is running in, i.e. the target process is the same as the host process.

Third Party Stackwalk StackwalkerAPI collects third party stackwalk when it walks the call stack in a different address space from the one it is running in, i.e. the target process is different from the host process. A third party stackwalk is usually done through a debugger interface.

3.1.2 Basic Types

```
typedef unsigned long Address
```

An integer value capable of holding an address in the target process. Address variables should not, and in many cases cannot, be used directly as a pointer. It may refer to an address in a different process, and it may not directly match the target process' pointer representation. Address is guaranteed to be at least large enough to hold an address in a target process, but may be larger.

```
typedef ... Dyninst::PID
```

A handle for identifying a process. On UNIX systems this will be an integer representing a PID. On Windows this will be a HANDLE object.

```
typedef ... Dyninst::THR_ID
```

A handle for identifying a thread. On Linux and BlueGene platforms this is an integer referring to a TID (Thread Identifier). On Windows it is a HANDLE object.

`class Dyninst::MachRegister`

A value that names a machine register.

`typedef unsigned long Dyninst::MachRegisterVal`

A value that holds the contents of a register. A Dyninst::MachRegister names a specific register, while a Dyninst::MachRegisterVal represents the value that may be in that register.

3.2 Namespace StackwalkerAPI

The classes in Section 3.3 and Section 3.6 fall under the C++ namespace Dyninst::Stackwalker. To access them, a user should refer to them using the Dyninst::Stackwalker:: prefix, e.g. Dyninst::Stackwalker::Walker. Alternatively, a user can add the C++ using keyword above any references to StackwalkerAPI objects, e.g, using namespace Dyninst and using namespace Stackwalker.

3.3 Stackwalking Interface

This section describes StackwalkerAPI's interface for walking a call stack. This interface is sufficient for walking call stacks on all the systems and variations covered by our default callbacks.

To collect a stackwalk, first create new Walker object associated with the target process via

```
Walker::newWalker()
```

or

```
Walker::newWalker(Dyninst::PID pid)
```

Once a Walker object has been created, a call stack can be walked with the

`Walker::walkStack`

method. The new stack walk is returned as a vector of Frame objects.

3.3.1 Class Walker

Defined in: `walker.h`

The `Walker` class allows users to walk call stacks and query basic information about threads in a target process. The user should create a `Walker` object for each process from which they are walking call stacks. Each `Walker` object is associated with one process, but may walk call stacks on multiple threads within that process. The `Walker` class allows users to query for the threads available for walking, and it allows you to specify a particular thread whose call stack should be walked. Stackwalks are returned as a vector of Frame objects.

Each Walker object contains three objects:

- ProcessState
- StepperGroup
- SymbolLookup

These objects are part of the Callback Interface and can be used to customize StackwalkerAPI. The `ProcessState` object tells `Walker` how to access data in the target process, and it determines whether this `Walker` collects first party or third party stackwalks. `Walker` will pick an appropriate default `ProcessState` object based on which factory method the users calls. The `StepperGroup` object is used to customize how the `Walker` steps through stack frames. The `SymbolLookup` object is used to customize how StackwalkerAPI looks up symbolic names of the function or object that created a stack frame.

```
static Walker *newWalker()
static Walker *newWalker(Dyninst::PID pid)
static Walker *newWalker(Dyninst::PID pid, std::string executable)
static Walker *newWalker(Dyninst::ProcControlAPI::Process::ptr proc);
static Walker *newWalker(std::string executable,
                         const std::vector<std::string> &argv)
static Walker *newWalker(ProcessState *proc,
                         StepperGroup *steppergroup = NULL ,
                         SymbolLookup *lookup = NULL)
```

These factory methods return new Walker objects:

- The first takes no arguments and returns a first-party stackwalker.
- The second takes a PID representing a running process and returns a third-party stackwalker on that process.
- The third takes the name of the executing binary in addition to the PID and also returns a third-party stackwalker on that process.
- The fourth takes a ProcControlAPI process object and returns a third-party stackwalker.
- The fifth takes the name of an executable and its arguments, creates the process, and returns a third-party stackwalker.
- The sixth takes a ProcessState pointer representing a running process as well as user-defined StepperGroup and SymbolLookup pointers. It can return both first-party and third-party Walkers, depending on the ProcessState parameter.

  Unless overriden with the sixth variant, the new Walker object uses the default StepperGroup and SymbolLookup callbacks for the current platform. First-party walkers use the ProcSelf callback for its ProcessState object. Third-party walkers use ProcDebug instead. See Section 3.5.1 for more information about defaults in the Callback Interface.

This method returns NULL if it was unable to create a new Walker object. The new Walker object was created with the new operator, and should be deallocated with the delete operator when it is no longer needed.

```
static bool newWalker(const std::vector<Dyninst::PID> &pids,
                      std::vector<Walker *> &walkers_out)
static bool newWalker(const std::vector<Dyninst::PID> &pids,
                      std::vector<Walker *> &walkers_out,
                      std::string executable)
```

This method attaches to a group of processes and returns a vector of Walker objects that perform third-party stackwalks. As above, the first variant takes a list of PIDs and attaches to those processes; the second variant also specifies the executable binary.

```
bool walkStack(std::vector<Frame> &stackwalk,
               Dyninst::THR_ID thread = NULL_THR_ID)
```

This method walks a call stack in the process associated with this `Walker`. The call stack is returned as a vector of `Frame` objects in stackwalk. The top of the stack is returned in index 0 of stackwalk, and the bottom of the stack is returned in index `stackwalk.size()-1`.

A stackwalk can be taken on a specific thread by passing a value in the thread parameter. If `thread` has the value `NULL_THR_ID`, then a default thread will be chosen. When doing a third party stackwalk, the default thread will be the process' initial thread. When doing a first party stackwalk, the default thread will be the thread that called `walkStack`. The default StepperGroup provided to a Walker will support collecting call stacks from almost all types of functions, including signal handlers and optimized, frameless functions.

This method returns `true` on success and `false` on failure.

```
bool walkStackFromFrame(std::vector<Frame> &stackwalk, const Frame &frame)
```

This method walks a call stack starting from the given stack frame, `frame`. The call stack will be output in the `stackwalk` vector, with frame stored in index 0 of `stackwalk` and the bottom of the stack stored in index `stackwalk.size()-1`.

This method returns `true` on success and `false` on failure.

```
bool walkSingleFrame(const Frame &in, Frame &out)
```

This methods walks through single frame, `in`. Parameter `out` will be set to `in`'s caller frame.

This method returns `true` on success and `false` on failure.

```
bool getInitialFrame(Frame &frame, Dyninst::THR_ID thread = NULL_THR_ID)
```

This method returns the `Frame` object on the top of the stack in parameter frame. Under `walkStack`, `frame` would be the one returned in index 0 of the `stackwalk` vector. A stack frame can be found on a specific thread by passing a value in the thread parameter. If `thread` has the value `NULL_THR_ID`, then a default thread will be chosen. When doing a third party stackwalk, the default thread will be the process' initial thread. When doing a first party stackwalk, the default thread will be the thread that called `getInitialFrame`.

This method returns `true` on success and `false` on failure.

```
bool getAvailableThreads(std::vector<Dyninst::THR_ID> &threads)
```

This method returns a vector of threads in the target process upon which StackwalkerAPI can walk call stacks. The threads are returned in output parameter `threads`. Note that this method may return a subset of the actual threads in the process. For example, when walking call stacks on the current process, it is only legal to walk the call stack on the currently running thread. In this case, `getAvailableThreads` returns a vector containing only the current thread.

This method returns `true` on success and `false` on failure.

```
ProcessState *getProcessState() const
```

This method returns the `ProcessState` object associated with this `Walker`.

```
StepperGroup *getStepperGroup() const
```

This method returns the `StepperGroup` object associated with this `Walker`.

```
SymbolLookup *getSymbolLookup() const
```

This method returns the `SymbolLookup` object associated with this `Walker`.

```
bool addStepper(FrameStepper *stepper)
```

This method adds a provided FrameStepper to those used by the Walker.

```
static SymbolReaderFactory *getSymbolReader()
```

This method returns a factory for creating process-specific symbol readers. Unlike the above methods it is global across all Walkers and is thus defined static.

```
static void setSymbolReader(SymbolReaderFactory *);
```

Set the symbol reader factory used when creating `Walker` objects.

```
static void version(int &major, int &minor, int &maintenance)
```

This method returns version information (e.g., 8, 0, 0 for the 8.0 release).

3.3.2 Class Frame

Defined in: `frame.h`

The `Walker` class returns a call stack as a vector of `Frame` objects. As described in Section 3.1.1, each Frame object represents a stack frame, and contains a return address (RA), stack pointer (SP) and frame pointer (FP). For each of these values, optionally, it stores the location where the values were found. Each Frame object may also be augmented with symbol information giving a function name (or a symbolic name, in the case of non-functions) for the object that created the stack frame.

The Frame class provides a set of functions (getRALocation, getSPLocation and getFPLocation) that return the location in the target process' memory or registers where the RA, SP, or FP were found. These functions may be used to modify the stack. For example, the DyninstAPI uses these functions to change return addresses on the stack when it relocates code. The RA, SP, and FP may be found in a register or in a memory address on a call stack.

```
static Frame *newFrame(Dyninst::MachRegisterVal ra,
                       Dyninst::MachRegisterVal sp,
                       Dyninst::MachRegisterVal fp,
                       Walker *walker)
```

This method creates a new `Frame` object and sets the mandatory data members: RA, SP and FP. The new `Frame` object is associated with `walker`.

The optional location fields can be set by the methods below.

The new `Frame` object is created with the `new` operator, and the user should be deallocate it with the `delete` operator when it is no longer needed.

```
bool operator==(const Frame &)
```

`Frame` objects have a defined equality operator.

```
Dyninst::MachRegisterVal getRA() const
```

This method returns this `Frame` object's return address.

```
void setRA(Dyninst::MachRegisterVal val)
```

This method sets this `Frame` object's return address to `val`.

```
Dyninst::MachRegisterVal getSP() const
```

This method returns this `Frame` object's stack pointer.

```
void setSP(Dyninst::MachRegisterVal val)
```

This method sets this `Frame` object's stack pointer to `val`.

```
Dyninst::MachRegisterVal getFP() const
```

This method returns this `Frame` object's frame pointer.

```
void setFP(Dyninst::MachRegisterVal val)
```

This method sets this `Frame` object's frame pointer to `val`.

```
bool isTopFrame() const;
bool isBottomFrame() const;
```

These methods return whether a `Frame` object is the top (e.g., most recently executing) or bottom of the stack walk.

```
typedef enum {
    loc_address,
    loc_register,
    loc_unknown
} storage_t;

typedef struct {
    union {
        Dyninst::Address addr;
        Dyninst::MachRegister reg;
    } val;
    storage_t location;
} location_t;
```

The `location_t` structure is used by the `getRALocation`, `getSPLocation`, and `getFPLocation` methods to describe where in the process a `Frame` object's RA, SP, or FP were found. When walking a call stack these values may be found in registers or memory. If they were found in

memory, the `location` field of `location_t` will contain `loc_address` and the `addr` field will contain the address where it was found. If they were found in a register the `location` field of `location_t` will contain `loc_register` and the `reg` field will refer to the register where it was found. If this `Frame` object was not created by a stackwalk (using the `newframe` factory method, for example), and has not had a set location method called, then location will contain `loc_unknown`.

```
location_t getRALocation() const
```

This method returns a `location_t` describing where the RA was found.

```
void setRALocation(location_t newval)
```

This method sets the location of where the RA was found to newval.

```
location_t getSPLocation() const
```

This method returns a `location_t` describing where the SP was found.

```
void setSPLocation(location_t newval)
```

This method sets the location of where the SP was found to `newval`.

```
location_t getFPLocation() const
```

This method returns a `location_t` describing where the FP was found.

```
void setFPLocation(location_t newval)
```

This method sets the location of where the FP was found to `newval`.

```
bool getName(std::string &str) const
```

This method returns a stack frame's symbolic name. Most stack frames are created by functions, or function-like objects such as signal handlers or system calls. This method returns the name of the object that created this stack frame. For stack frames created by functions, this symbolic name will be the function name. A symbolic name may not always be available for all `Frame` objects, such as in cases of stripped binaries or special stack frames types.

The function name is obtained by using this `Frame` object's RA to call the `SymbolLookup` callback. By default StackwalkerAPI will attempt to use the `SymtabAPI` package to look up symbol names in binaries. If `SymtabAPI` is not found, and no alternative `SymbolLookup` object is present, then this method will return an error.

This method returns `true` on success and `false` on error.

`bool getObject(void* &obj) const`

In addition to returning a symbolic name (see `getName`) the `SymbolLookup` interface allows for an opaque object, a `void*`, to be associated with a `Frame` object. The contents of this `void*` is determined by the `SymbolLookup` implementation. Under the default implementation that uses SymtabAPI, the `void*` points to a Symbol object or NULL if no symbol is found.

This method returns `true` on success and `false` on error.

```
Walker *getWalker() const;
```

This method returns the `Walker` object that constructed this stack frame.

```
THR_ID getThread() const;
```

This method returns the execution thread that the current `Frame` represents.

`FrameStepper* getStepper() const`

This method returns the `FrameStepper` object that was used to construct this `Frame` object in the `stepper` output parameter.

This method returns `true` on success and `false` on error.

`bool getLibOffset(std::string &lib, Dyninst::Offset &offset, void* &symtab) const`

This method returns the DSO (a library or executable) and an offset into that DSO that points to the location within that DSO where this frame was created. `lib` is the path to the library that was loaded, and `offset` is the offset into that library. The return value of the `symtab` parameter is dependent on the SymbolLookup implementation-by default it will contain a pointer to a Dyninst::Symtab object for this DSO. See the SymtabAPI Programmer's Guide for more information on using Dyninst::Symtab objects.

`bool nonCall() const`

This method returns whether a `Frame` object represents a function call; if `false`, the `Frame` may represent instrumentation, a signal handler, or something else.

3.4 Mapping Addresses to Libraries

Defined in: `procstate.h`

StackwalkerAPI provides an interface to access the addresses where libraries are mapped in the target process.

```
typedef std::pair<std::string, Address> LibAddrPair;
```

A pair consisting of a library filename and its base address in the target process.

```
class LibraryState
```

Class providing interfaces for library tracking. Only the public query interfaces below are user-facing; the other public methods are callbacks that allow StackwalkerAPI to update its internal state.

```
virtual bool getLibraryAtAddr(Address addr, LibAddrPair &lib) = 0;
```

Given an address `addr` in the target process, returns `true` and sets `lib` to the name and base address of the library containing addr. Given an address outside the target process, returns `false`.

```
virtual bool getLibraries(std::vector<LibAddrPair> &libs, bool allow\_refresh = true) = 0;
```

Fills `libs` with the libraries loaded in the target process. If `allow_refresh` is true, this method will attempt to ensure that this list is freshly updated via inspection of the process; if it is false, it will return a cached list.

```
virtual bool getLibc(LibAddrPair &lc);
```

Convenience function to find the name and base address of the standard C runtime, if present.

```
virtual bool getLibthread(LibAddrPair &lt);
```

Convenience function to find the name and base address of the standard thread library, if present (e.g. pthreads).

```
virtual bool getAOut(LibAddrPair &ao) = 0;
```

Convenience function to find the name and base address of the executable.

3.5 Accessing Local Variables

Defined in: `local_var.h`

StackwalkerAPI can be used to access local variables found in the frames of a call stack. The StackwalkerAPI interface for accessing the values of local variables is closely tied to the SymtabAPI interface for collecting information about local variables–SymtabAPI handles for functions, local variables, and types are part of this interface.

Given an initial handle to a SymtabAPI Function object, SymtabAPI can look up local variables contained in that function and the types of those local variables. See the SymtabAPI Programmer's Guide for more information.

```
static Dyninst::SymtabAPI::Function *getFunctionForFrame(Frame f)
```

This method returns a SymtabAPI function handle for the function that created the call stack frame, f.

```
static int glvv_Success = 0;
static int glvv_EParam = -1;
static int glvv_EOutOfScope = -2;
static int glvv_EBufferSize = -3;
static int glvv_EUnknown = -4;

static int getLocalVariableValue(Dyninst::SymtabAPI::localVar *var,
                                 std::vector<Frame> &swalk,
                                 unsigned frame,
                                 void *out_buffer,
                                 unsigned out_buffer_size)
```

Given a local variable and a stack frame from a call stack, this function returns the value of the variable in that frame. The local variable is specified by the SymtabAPI variable object, `var`. `swalk` is a call stack that was collected via StackwalkerAPI, and `frame` specifies an index into that call stack that contains the local variable. The value of the variable is stored in `out_buffer` and the size of `out_buffer` should be specified in `out_buffer_size`.

A local variable only has a limited scope with-in a target process' execution. StackwalkerAPI cannot guarantee that it can collect the correct return value of a local variable from a call stack if the target process is continued after the call stack is collected.

Finding and collecting the values of local variables is dependent on debugging information being present in a target process' binary. Not all binaries contain debugging information, and in some cases, such as for binaries built with high compiler optimization levels, that debugging information may be incorrect.

`getLocalVariableValue` will return on of the following values:

glvv_Success getLocalVariableValue was able to correctly read the value of the given variable.

glvv_EParam An error occurred, an incorrect parameter was specified (frame was larger than `swalk.size()`, or var was not a variable in the function specified by frame).

glvv_EOutOfScope An error occurred, the specified variable exists in the function but isn't live at the current execution point.

glvv_EBufferSize An error occurred, the variable's value does not fit inside `out_buffer`.

glvv_EUnknown An unknown error occurred. It is most likely that the local variable was optimized away or debugging information about the variable was incorrect.

3.6 Callback Interface

This subsection describes the Callback Interface for StackwalkerAPI. The Callback Interface is primarily used to port StackwalkerAPI to new platforms, extend support for new types of stack frames, or integrate StackwalkerAPI into existing tools.

The classes in this subsection are interfaces, they cannot be instantiated. To create a new implementation of one of these interfaces, create a new class that inherits from the callback class and implement the necessary methods. To use a new ProcessState, StepperGroup, or SymbolLookup class with StackwalkerAPI, create a new instance of the class and register it with a new Walker object using the

```
Walker::newWalker(ProcessState *, StepperGroup *, SymbolLookup *)
```

factory method (see Section 3.3.1). To use a new FrameStepper class with StackwalkerAPI, create a new instance of the class and register it with a StepperGroup using the

```
StepperGroup::addStepper(FrameStepper *)
```

method (see Section 3.6.3).

Some of the classes in the Callback Interface have methods with default implementations. A new class that inherits from a Callback Interface can optionally implement these methods, but it is not required. If a method requires implementation, it is written as a C++ pure virtual method (`virtual funcName() = 0`). A method with a default implementation is written as a C++ virtual method (`virtual funcName()`).

3.6.1 Default Implementations

The classes described in the Callback Interface are C++ abstract classes, or interfaces. They cannot be instantiated. For each of these classes StackwalkerAPI provides one or more default implementations on each platform. These default implementations are classes that inherit from the abstract classes described in the Callback Interface. If a user creates a Walker object without providing their own `FrameStepper`, `ProcessState`, and `SymbolLookup` objects, then StackwalkerAPI will use the default implementations listed in Table 1. These implementations are described in Section 4.2.

3.6.2 Class FrameStepper

Defined in: `framestepper.h`

	StepperGroup	ProcessState	SymbolLookup	FrameStepper
Linux/x86 Linux/x86-64	1. AddrRange	1. ProcSelf 2. ProcDebug	1. SwkSymtab	1. FrameFuncStepper 2. SigHandlerStepper 3. DebugStepper 4. AnalysisStepper 5. StepperWanderer 6. BottomOfStackStepper
Linux/PPC Linux/PPC-64	1. AddrRange	1. ProcSelf 2. ProcDebug	1. SwkSymtab	1. FrameFuncStepper 2. SigHandlerStepper 3. AnalysisStepper
Windows/x86	1. AddrRange	1. ProcSelf 2. ProcDebug	1. SwkSymtab	1. FrameFuncStepper 2. AnalysisStepper 3. StepperWanderer 4. BottomOfStackStepper
BlueGene	1. AddrRange	1. ProcDebug	1. SwkSymtab	1. FrameFuncStepper

Table 1: Callback Interface Defaults

The `FrameStepper` class is an interface that tells StackwalkerAPI how to walk through a specific type of stack frame. There may be many different ways of walking through a stack frame on a platform, e.g, on Linux/x86 there are different mechanisms for walking through system calls, signal handlers, regular functions, and frameless functions. A single `FrameStepper` describes how to walk through one of these types of stack frames.

A user can create their own `FrameStepper` classes that tell StackwalkerAPI how to walk through new types of stack frames. A new `FrameStepper` object must be added to a `StepperGroup` before it can be used.

In addition to walking through individual stack frames, a `FrameStepper` tells its `StepperGroup` when it can be used. The `FrameStepper` registers address ranges that cover objects in the target process' code space (such as functions). These address ranges should contain the objects that will create stack frames through which the `FrameStepper` can walk. If multiple `FrameStepper` objects have overlapping address ranges, then a priority value is used to determine which `FrameStepper` should be attempted first.

`FrameStepper` is an interface class; it cannot be instantiated. Users who want to develop new `FrameStepper` objects should inherit from this class and implement the the desired virtual functions. The `getCallerFrame`, `getPriority`, and `getName` functions must be implemented; all others may be overridden if desired.

```
typedef enum {
    gcf_success,
    gcf_stackbottom,
    gcf_not_me,
    gcf_error
} gcframe_ret_t
```

```
virtual gcframe_ret_t getCallerFrame(const Frame &in, Frame &out) = 0
```

This method walks through a single stack frame and generates a Frame object that represents the caller's stack frame. Parameter in will be a Frame object that this FrameStepper is capable of

walking through. Parameter out is an output parameter that this method should set to the Frame object that called in.

There may be multiple ways of walking through a different types of stack frames. Each `FrameStepper` class should be able to walk through a type of stack frame. For example, on x86 one `FrameStepper` could be used to walk through stack frames generated by ABI-compliant functions; out's FP and RA are found by reading from in's FP, and out's SP is set to the word below in's FP. A different `FrameStepper` might be used to walk through stack frames created by functions that have optimized away their FP. In this case, in may have a FP that does not point out's FP and RA. The `FrameStepper` will need to use other mechanisms to discover out's FP or RA; perhaps the `FrameStepper` searches through the stack for the RA or performs analysis on the function that created the stack frame.

If `getCallerFrame` successfully walks through in, it is required to set the following parameters in out. See Section 3.3.2 for more details on the values that can be set in a Frame object:

Return Address (RA) The RA should be set with the `Frame::setRA` method.

Stack Pointer (SP) The SP should be set with the `Frame::setSP` method.

Frame Pointer (FP) The FP should be set with the `Frame::setFP` method

Optionally, getCallerFrame can also set any of following parameters in out:

Return Address Location (RALocation) The RALocation should be set with the `Frame::setRALocation()` method.

Stack Pointer Location (SPLocation) The SPLocation should be set with the `Frame::setRALocation()` method.

Frame Pointer Location (FPLocation) The FPLocation should be set with the `Frame::setFPLocation()` method.

If a location field in out is not set, then the appropriate `Frame::getRALocation`, `Frame::getSPLocation` or `Frame::getFPLocation` method will return `loc_unknown`.

`getCallerFrame` should return `gcf_success` if it successfully walks through in and creates an `out Frame` object. It should return `gcf_stackbottom` if in is the bottom of the stack and there are no stack frames below it. It should return `gcf_not_me` if in is not the correct type of stack frame for this `FrameStepper` to walk through. StackwalkerAPI will then attempt to locate another `FrameStepper` to handle `in` or abort the stackwalk. It should return `gcf_error` if there was an error and the stack walk should be aborted.

```
virtual void registerStepperGroup(StepperGroup *steppergroup)
```

This method is used to notify a `FrameStepper` when StackwalkerAPI adds it to a `StepperGroup`. The `StepperGroup` to which this `FrameStepper` is being added is passed in parameter steppergroup. This method can be used to initialize the `FrameStepper` (in addition to any `FrameStepper` constructor).

```
virtual unsigned getPriority() const = 0
```

This method is used by the `StepperGroup` to decide which `FrameStepper` to use if multiple `FrameStepper` objects are registered over the same address range (see addAddressRanges in Section 3.6.3 for more information about address ranges). This method returns an integer representing a priority level, the lower the number the higher the priority.

The default `FrameStepper` objects provided by StackwalkerAPI all return priorities between `0x1000` and `0x2000`. If two `FrameStepper` objects have an overlapping address range, and they have the same priority, then the order in which they are used is undefined.

```
FrameStepper(Walker *w);
```

Constructor definition for all `FrameStepper` instances.

```
virtual ProcessState *getProcessState();
```

Return the `ProcessState` used by the `FrameStepper`. Can be overridden if the user desires.

```
virtual Walker *getWalker();
```

Return the `Walker` associated with the `FrameStepper`. Can be overridden if the user desires.

```
typedef std::pair<std::string, Address> LibAddrPair;
typedef enum { library_load, library_unload } lib_change_t;
virtual void newLibraryNotification(LibAddrPair *libAddr,
                                    lib_change_t change);
```

This function is called when a new library is loaded by the process; it should be implemented if the `FrameStepper` requires such information.

```
virtual const char *getName() const = 0;
```

Returns a name for the `FrameStepper`; must be implemented by the user.

3.6.3 Class StepperGroup

Defined in: `steppergroup.h`

The `StepperGroup` class contains a collection of `FrameStepper` objects. The `StepperGroup`'s primary job is to decide which `FrameStepper` should be used to walk through a stack frame given a return address. The default `StepperGroup` keeps a set of address ranges for each `FrameStepper`. If multiple `FrameStepper` objects overlap an address, then the default `StepperGroup` will use a priority system to decide.

`StepperGroup` provides both an interface and a default implementation of that interface. Users who want to customize the `StepperGroup` should inherit from this class and re-implement any of the below virtual functions.

```
StepperGroup(Walker *walker)
```

This factory constructor creates a new `StepperGroup` object associated with `walker`.

```
virtual bool addStepper(FrameStepper *stepper)
```

This method adds a new `FrameStepper` to this `StepperGroup`. The newly added stepper will be tracked by this `StepperGroup`, and it will be considered for use when walking through stack frames.

This method returns `true` if it successfully added the `FrameStepper`, and `false` on error.

```
virtual bool addStepper(FrameStepper *stepper, Address start, Address
end) = 0;
```

Add the specified `FrameStepper` to the list of known steppers, and register it to handle frames in the range [`start`, `end`).

```
virtual void registerStepper(FrameStepper *stepper);
```

Add the specified `FrameStepper` to the list of known steppers and use it over the entire address space.

```
virtual bool findStepperForAddr(Address addr, FrameStepper* &out,
                                const FrameStepper *last_tried = NULL) = 0
```

Given an address that points into a function (or function-like object), addr, this method decides which `FrameStepper` should be used to walk through the stack frame created by the function at that address. A pointer to the `FrameStepper` will be returned in parameter `out`.

It may be possible that the `FrameStepper` this method decides on is unable to walk through the stack frame (it returns `gcf_not_me` from `FrameStepper::getCallerFrame`). In this case StackwalkerAPI will call findStepperForAddr again with the last_tried parameter set to the failed `FrameStepper`. findStepperForAddr should then find another `FrameStepper` to use. Parameter `last_tried` will be set to NULL the first time getStepperToUse is called for a stack frame.

The default version of this method uses address ranges to decide which `FrameStepper` to use. The address ranges are contained within the process' code space, and map a piece of the code space to a `FrameStepper` that can walk through stack frames created in that code range. If multiple `FrameStepper` objects share the same range, then the one with the highest priority will be tried first.

This method returns `true` on success and `false` on failure.

```
typedef std::pair<std::string, Address> LibAddrPair;
typedef enum { library_load, library_unload } lib_change_t;
virtual void newLibraryNotification(LibAddrPair *libaddr, lib_change_t
change);
```

Called by the StackwalkerAPI when a new library is loaded.

```
Walker *getWalker() const
```

This method returns the Walker object that associated with this StepperGroup.

```
void getSteppers(std::set<FrameStepper *> &);
```

Fill in the provided set with all `FrameSteppers` registered in the `StepperGroup`.

3.6.4 Class ProcessState

Defined in: `procstate.h`

The ProcessState class is a virtual class that defines an interface through which StackwalkerAPI can access the target process. It allows access to registers and memory, and provides basic information about the threads in the target process. StackwalkerAPI provides two default types of `ProcessState` objects: `ProcSelf` does a first party stackwalk, and `ProcDebug` does a third party stackwalk.

A new `ProcessState` class can be created by inheriting from this class and implementing the necessary methods.

```
static ProcessState *getProcessStateByPid(Dyninst::PID pid)
```

Given a `PID`, return the corresponding `ProcessState` object.

```
virtual unsigned getAddressWidth() = 0;
```

Return the number of bytes in a pointer for the target process. This value is 4 for 32-bit platforms (x86, PowerPC-32) and 8 for 64-bit platforms (x86-64, PowerPC-64).

```
typedef enum { Arch_x86, Arch_x86_64, Arch_ppc32, Arch_ppc64 }
Architecture;
virtual Dyninst::Architecture getArchitecture() = 0;
```

Return the appropriate architecture for the target process.

```
virtual bool getRegValue(Dyninst::MachRegister reg,
                         Dyninst::THR_ID thread,
                         Dyninst::MachRegisterVal &val) = 0
```

This method takes a register name as input, `reg`, and returns the value in that register in `val` in the thread thread.

This method returns `true` on success and `false` on error.

```
virtual bool readMem(void *dest, Address source, size_t size) = 0
```

This method reads memory from the target process. Parameter `dest` should point to an allocated buffer of memory at least `size` bytes in the host process. Parameter `source` should contain an address in the target process to be read from. If this method succeeds, `size` bytes of memory is copied from `source`, stored in `dest`, and `true` is returned. This method returns `false` otherwise.

```
virtual bool getThreadIds(std::vector<Dyninst::THR_ID> &threads) = 0
```

This method returns a list of threads whose call stacks can be walked in the target process. Thread are returned in the `threads` vector. In some cases, such as with the default `ProcDebug`, this method returns all of the threads in the target process. In other cases, such as with `ProcSelf`, this method returns only the calling thread.

The first thread in the `threads` vector (index 0) will be used as the default thread if the user requests a stackwalk without specifying an thread (see `Walker::WalkStack`).

This method returns `true` on success and `false` on error.

```
virtual bool getDefaultThread(Dyninst::THR_ID &default_tid) = 0
```

This method returns the thread representing the initial process in the `default_tid` output parameter.

This method returns `true` on success and `false` on error.

```
virtual Dyninst::PID getProcessId()
```

This method returns a process ID for the target process. The default `ProcessState` implementations (`ProcDebug` and `ProcSelf`) will return a PID on UNIX systems and a HANDLE object on Windows.

```
Walker *getWalker() const;
```

Return the `Walker` associated with the current process state.

```
std::string getExecutablePath();
```

Returns the name of the executable associated with the current process state.

3.6.4.1 Class LibraryState Defined in: procstate.h

LibraryState is a helper class for ProcessState that provides information about the current DSOs (libraries and executables) that are loaded into a process' address space. FrameSteppers frequently use the LibraryState to get the DSO through which they are attempting to stack walk.

Each Library is represented using a LibAddrPair object, which is defined as follows:

```
typedef std::pair<std::string, Dyninst::Address> LibAddrPair
```

LibAddrPair.first refers to the file path of the library that was loaded, and LibAddrPair.second is the load address of that library in the process' address space. The load address of a library can be added to a symbol offset from the file in order to get the absolute address of a symbol.

```
virtual bool getLibraryAtAddr(Address addr, LibAddrPair &lib) = 0
```

This method returns a DSO, using the lib output parameter, that is loaded over address addr in the current process.

This method returns false if no library is loaded over addr or an error occurs, and true if it successfully found a library.

```
virtual bool getLibraries(std::vector<LibAddrPair> &libs) = 0
```

This method returns all DSOs that are loaded into the process' address space in the output vector parameter, libs.

This method returns true on success and false on error.

```
virtual void notifyOfUpdate() = 0
```

This method is called by the ProcessState when it detects a change in the process' list of loaded libraries. Implementations of LibraryStates should use this method to refresh their lists of loaded libraries.

```
virtual Address getLibTrapAddress() = 0
```

Some platforms that implement the System/V standard (Linux, BlueGene/P) use a trap event to determine when a process loads a library. A trap instruction is inserted into a certain address, and that trap will execute whenever the list of loaded libraries change.

On System/V platforms this method should return the address where a trap should be inserted to watch for libraries loading and unloading. The ProcessState object will insert a trap at this address and then call notifyOfUpdate when that trap triggers.

On non-System/V platforms this method should return 0.

3.6.5 Class SymbolLookup

Defined in: `symlookup.h`

The `SymbolLookup` virtual class is an interface for associating a symbolic name with a stack frame. Each `Frame` object contains an address (the RA) pointing into the function (or function-like object) that created its stack frame. However, users do not always want to deal with addresses when symbolic names are more convenient. This class is an interface for mapping a `Frame` object's RA into a name.

In addition to getting a name, this class can also associate an opaque object (via a `void*`) with a Frame object. It is up to the `SymbolLookup` implementation what to return in this opaque object.

The default implementation of `SymbolLookup` provided by StackwalkerAPI uses the `SymLite` tool to lookup symbol names. It returns a Symbol object in the anonymous `void*`.

```
SymbolLookup(std::string exec_path = "");
```

Constructor for a `SymbolLookup` object.

```
virtual bool lookupAtAddr(Address addr,
                          string &out_name,
                          void* &out_value) = 0
```

This method takes an address, `addr`, as input and returns the function name, `out_name`, and an opaque value, `out_value`, at that address. Output parameter `out_name` should be the name of the function that contains `addr`. Output parameter `out_value` can be any opaque value determined by the `SymbolLookup` implementation. The values returned are used by the `Frame::getName` and `Frame::getObject` functions.

This method returns `true` on success and `false` on error.

```
virtual Walker *getWalker()
```

This method returns the `Walker` object associated with this `SymbolLookup`.

```
virtual ProcessState *getProcessSate()
```

This method returns the `ProcessState` object associated with this `SymbolLookup`.

4 Callback Interface Default Implementations

StackwalkerAPI provides one or more default implementations of each of the callback classes described in Section 3.5. These implementations are used by a default configuration of StackwalkerAPI.

4.1 Debugger Interface

This section describes how to use StackwalkerAPI for collecting 3rd party stack walks. In 3rd party mode StackwalkerAPI uses the OS's debugger interface to connect to another process and walk its call stacks. As part of being a debugger StackwalkerAPI receives and needs to handle debug events. When a debugger event occurs, StackwalkerAPI must get control of the host process in order to receive the debugger event and continue the target process.

To illustrate the complexities with running in 3rd party mode, consider the follow code snippet that uses StackwalkerAPI to collect a stack walk every five seconds.

```
Walker *walker = Walker::newWalker(pid);
std::vector<Frame> swalk;
for (;;) {
                walker->walkStack(swalk);
                sleep(5);
}
```

StackwalkerAPI is running in 3rd party mode, since it attached to the target process, `pid`. As the target process runs it may be generating debug events such a thread creation and destruction, library loads and unloads, signals, forking/execing, etc. When one of these debugger events is generated the OS will pause the target process and send a notice to the host process. The target process will remain paused until the host process handles the debug event and resumes the target process.

In the above example the host process is spending almost all of its time in the sleep call. If a debugger event happens during the sleep, then StackwalkerAPI will not be able to get control of the host process and handle the event for up to five seconds. This will cause long pauses in the target process and lead to a potentially very large slowdown.

To work around this problem StackwalkerAPI provides a notification file descriptor. This file descriptor represents a connection between the StackwalkerAPI library and user code. StackwalkerAPI will write a single byte to this file descriptor when a debug event occurs, thus notifying the user code that it needs to let StackwalkerAPI receive and handle debug events. The user code can use system calls such as select to watch for events on the notification file descriptor.

The following example illustrates how to properly use StackwalkerAPI to collect a stack walk from another process at a five second interval. Details on the `ProcDebug` class, `getNotificationFD` method, and `handleDebugEvent` method can be found in Section 4.1.1. See the UNIX man pages for more information on the `select` system call. Note that this example does not include all of the proper error handling and includes that should be present when using `select`.

```
Walker *walker = Walker::newWalker(pid);
ProcDebug *debugger = (ProcDebug *) walker->getProcessState();
std::vector<Frame> swalk;
for (;;) {
    walker->walkStack(swalk);
    struct timeval timeout;
    timeout.tv_sec = 5;
    timeout.tv_usec = 0;
    int max = 1;
    fd_set readfds, writefds, exceptfds;
    FD_ZERO(&readfds); FD_ZERO(&writefds); FD_ZERO(&exceptfds);
    FD_SET(ProcDebug::getNotificationFD(), &readfds);
```

```
    for (;;) {
        int result = select(max, &readfds, &writefds, &exceptfds, &timeout);
        if (FD_ISSET(ProcDebug::getNotificationFD(), readfds)) {
            //Debug event
            ProcDebug::handleDebugEvent();
        }
        if (result == 0) {
            //Timeout
            break;
        }
    }
}
```

4.1.1 Class ProcDebug

Defined in: `procstate.h`

Access to StackwalkerAPI's debugger is through the `ProcDebug` class, which inherits from the `ProcessState` interface. The easiest way to get at a `ProcDebug` object is to cast the return value of `Walker::getProcessState` into a `ProcDebug`. C++'s `dynamic_cast` operation can be used to test if a `Walker` uses the `ProcDebug` interface:

```
ProcDebug *debugger;
debugger = dynamic_cast<ProcDebug*>(walker->getProcessState());
if (debugger != NULL) {
    //3rd party
    ...
} else {
    //1st party
    ...
}
```

In addition to the handling of debug events, described in Section 4.1, the `ProcDebug` class provides a process control interface; users can pause and resume process or threads, detach from a process, and test for events such as process death. As an implementation of the `ProcessState` class, `ProcDebug` also provides all of the functionality described in Section 3.6.4.

```
virtual bool pause(Dyninst::THR_ID tid = NULL_THR_ID)
```

> This method pauses a process or thread. The paused object will not resume execution until `ProcDebug::resume` is called. If the `tid` parameter is not `NULL_THR_ID` then StackwalkerAPI will pause the thread specified by `tid`. If `tid` is `NULL_THR_ID` then StackwalkerAPI will pause every thread in the process.
>
> When StackwalkerAPI collects a call stack from a running thread it first pauses the thread, collects the stack walk, and then resumes the thread. When collecting a call stack from a paused thread StackwalkerAPI will collect the stack walk and leave the thread paused. This method is thus useful for pausing threads before stack walks if the user needs to keep the returned stack walk synchronized with the current state of the thread.

This method returns `true` if successful and `false` on error.

`virtual bool resume(Dyninst::THR_ID tid = NULL_THR_ID)`

This method resumes execution on a paused process or thread. This method only resumes threads that were paused by the `ProcDebug::pause` call, using it on other threads is an error. If the `tid` parameter is not `NULL_THR_ID` then StackwalkerAPI will resume the thread specified by `tid`. If `tid` is `NULL_THR_ID` then StackwalkerAPI will resume all paused threads in the process.

This method returns `true` if successful and `false` on error.

`virtual bool detach(bool leave_stopped = false)`

This method detaches StackwalkerAPI from the target process. StackwalkerAPI will no longer receive debug events on this target process and will no longer be able to collect call stacks from it. This method invalidates the associated `Walker` and `ProcState` objects, they should be cleaned using C++'s `delete` operator after making this call. It is an error to attempt to do operations on these objects after a detach, and undefined behavior may result.

If the `leave_stopped` parameter is `true` StackwalkerAPI will detach from the process but leave it in a paused state so that it does resume progress. This is useful for attaching another debugger back to the process for further analysis. The `leave_stopped` parameter is not supported on the Linux platform and its value will have no affect on the detach call.

This method returns `true` if successful and `false` on error.

`virtual bool isTerminated()`

This method returns `true` if the associated target process has terminated and `false` otherwise. A target process may terminate itself by calling exit, returning from main, or receiving an unhandled signal. Attempting to collect stack walks or perform other operations on a terminated process is illegal an will lead to undefined behavior.

A process termination will also be signaled through the notification FD. Users should check processes for the isTerminated state after returning from handleDebugEvent.

`static int getNotificationFD()`

This method returns StackwalkerAPI's notification FD. The notification FD is a file descriptor that StackwalkerAPI will write a byte to whenever a debug event occurs that need. If the user code sees a byte on this file descriptor it should call `handleDebugEvent` to let StackwalkerAPI handle the debug event. Example code using `getNotificationFD` can be found in Section 4.1.

StackwalkerAPI will only create one notification FD, even if it is attached to multiple 3rd party target processes.

```
static bool handleDebugEvent(bool block = false)
```

When this method is called StackwalkerAPI will receive and handle all pending debug events from each 3rd party target process to which it is attached. After handling debug events each target process will be continued (unless it was explicitly stopped by the ProcDebug::pause method) and any bytes on the notification FD will be cleared. It is generally expected that users will call this method when a event is sent to the notification FD, although it can be legally called at any time.

If the `block` parameter is `true`, then `handleDebugEvents` will block until it has handled at least one debug event. If the block parameter is `false`, then handleDebugEvents will handle any currently pending debug events or immediately return if none are available.

StackwalkerAPI may receive process exit events for target processes while handling debug events. The user should check for any exited processes by calling `ProcDebug::isTerminated` after handling debug events.

This method returns `true` if successful and `false` on error.

4.2 FrameSteppers

Defined in: `framestepper.h`

StackwalkerAPI ships with numerous default implementations of the `FrameStepper` class. Each of these `FrameStepper` implementations allow StackwalkerAPI to walk a type of call frames. Section 3.6.1 describes which `FrameStepper` implementations are available on which platforms. This sections gives a brief description of what each `FrameStepper` implementation does. Each of the following classes implements the `FrameStepper` interface described in Section 3.6.2, so we do not repeat the API description for the classes here.

Several of the `FrameStepper`s use helper classes (see `FrameFuncStepper` as an example). Users can further customize the behavior of a `FrameStepper` by providing their own implementation of these helper classes.

4.2.1 Class FrameFuncStepper

This class implements stack walking through a call frame that is setup with the architectures standard stack frame. For example, on x86 this `FrameStepper` will be used to walk through stack frames that are setup with a `push %ebp/mov %esp,%ebp` prologue.

4.2.1.1 Class FrameFuncHelper

`FrameFuncStepper` uses a helper class, `FrameFuncHelper`, to get information on what kind of stack frame it's walking through. The `FrameFuncHelper` will generally use techniques such as binary analysis to determine what type of stack frame the `FrameFuncStepper` is walking through. Users can have StackwalkerAPI use their own binary analysis mechanisms by providing an implementation of this `FrameFuncHelper`.

There are two important types used by `FrameFuncHelper` and one important function:

```
typedef enum {
    unknown_t=0,
```

```
    no_frame,
    standard_frame,
    savefp_only_frame,
} frame_type;
```

The `frame_type` describes what kind of stack frame a function uses. If it does not set up a stack frame then `frame_type` should be `no_frame`. If it sets up a standard frame then `frame_type` should be `standard_frame`. The `savefp_only_frame` value currently only has meaning on the x86 family of systems, and means that a function saves the old frame pointer, but does not setup a new frame pointer (it has a `push %ebp` instruction, but no `mov %esp,%ebp`). If the `FrameFuncHelper` cannot determine the `frame_type`, then it should be assigned the value `unknown_t`.

```
typedef enum {
    unknown_s=0,
    unset_frame,
    halfset_frame,
    set_frame
} frame_state;
```

The `frame_state` type determines the current state of function with a stack frame at some point of execution. For example, a function may set up a standard stack frame and have a `frame_type` of `standard_frame`, but execution may be at the first instruction in the function and the frame is not yet setup, in which case the `frame_state` will be `unset_frame`.

If the function sets up a standard stack frame and the execution point is someplace where the frame is completely setup, then the `frame_state` should be `set_frame`. If the function sets up a standard frame and the execution point is at a point where the frame does not yet exist or has been torn down, then `frame_state` should be `unset_frame`. The `halfset_frame` value of `frame_state` is currently only meaningful on the x86 family of architecture, and should if the function has saved the old frame pointer, but not yet set up a new frame pointer.

```
typedef std::pair<frame_type, frame_state> alloc_frame_t;
virtual alloc_frame_t allocatesFrame(Address addr) = 0;
```

The `allocatesFrame` function of `FrameFuncHelper` returns a `alloc_frame_t` that describes the frame_type of the function at `addr` and the `frame_state` of the function when execution reached `addr`.

If `addr` is invalid or an error occurs, allocatedFrame should return `alloc_frame_t(unknown_t, unknown_s)`.

4.2.2 Class SigHandlerStepper

The `SigHandlerStepper` is used to walk through UNIX signal handlers as found on the call stack. On some systems a signal handler generates a special kind of stack frame that cannot be walked through using normal stack walking techniques.

4.2.3 Class DebugStepper

This class uses debug information found in a binary to walk through a stack frame. It depends on SymtabAPI to read debug information from a binary, then uses that debug information to walk through a call frame.

Most binaries must be built with debug information (`-g` with `gcc`) in order to include debug information that this `FrameStepper` uses. Some languages, such as C++, automatically include stackwalking debug information for use by exceptions. The `DebugStepper` class will also make use of this kind of exception information if it is available.

4.2.4 Class AnalysisStepper

This class uses dataflow analysis to determine possible stack sizes at all locations in a function as well as the location of the frame pointer. It is able to handle optimized code with omitted frame pointers and overlapping code sequences.

4.2.5 Class StepperWanderer

This class uses a heuristic approach to find possible return addresses in the stack frame. If a return address is found that matches a valid caller of the current function, we conclude it is the actual return address and construct a matching stack frame. Since this approach is heuristic it can make mistakes leading to incorrect stack information. It has primarily been replaced by the `AnalysisStepper` described above.

4.2.6 Class BottomOfStackStepper

The `BottomOfStackStepper` doesn't actually walk through any type of call frame. Instead it attempts to detect whether the bottom of the call stack has been reached. If so, `BottomOfStackStepper` will report `gcf_stackbottom` from its `getCallerFrame` method. Otherwise it will report `gcf_not_me`. `BottomOfStackStepper` runs with a higher priority than any other `FrameStepper` class.

Paradyn Parallel Performance Tools

DataflowAPI Programmer's Guide

9.2 Release
June 2016

Computer Sciences Department
University of Wisconsin–Madison
Madison, WI 53706

Computer Science Department
University of Maryland
College Park, MD 20742

Email dyninst-api@cs.wisc.edu
Web www.dyninst.org

Contents

1 Introduction

DataFlowAPI aggregates a collection of dataflow analysis algorithms that are useful in Dyninst development into a single library. These algorithms can also be foundations for users to build customized analyses. Currently, these algorithms include:

- Slicing takes a program location as input and can either slice backward to determine which instructions affect the results of the given program location, or slice forward to determine which instructions are affected by the results of the given program location. One key feature of our slicing implementation is that users can control where and when to stop slicing through a set of call back functions.
- Stack Analysis determines whether or not a register or memory location points to the stack. If it does point to the stack, Stack Analysis may be able to determine the exact stack location that is pointed to.
- Symbolic Expansion and Evaluation convert instructions to several symbolic expressions. Each symbolic expression represents the overall effects of these instructions on a register or a memory location.
- Register Liveness determines whether a register is live or not at a program location. A register is live at a program location if it will be used later in the program before its content is overwritten.

2 Abstractions

DataflowAPI starts from the control flow graphs generated by ParseAPI and the instructions generated by InstructionAPI. From these, it provides dataflow facts in a variety of forms. The key abstractions used by DataflowAPI are:

- Abstract Location represents a register or memory location in the program. DataflowAPI provides three types of abstract locations: register, stack, and heap. A register abstract location represents a register, and the same register at two different program locations is treated as the same abstract location. A stack abstract location consists of the stack frame to which it belongs and the offset within the stack frame. A heap abstract location consists of the virtual address of the heap variable.
- Abstract Region represents a set of abstract locations of the same type. If an abstract region contains only a single abstract location, the abstract location is precisely represented. If an abstract region contains more than one abstract location, the region contains the type of the locations. In the cases where it represents memory (either heap or stack), an abstract region also contains the memory address calculation that gives rise to this region.
- Abstract Syntax Tree (AST) represents a symbolic expression of an instruction's semantics. Specifically, an AST specifies how the value of an abstract location is modified by the instruction.

- ASSIGNMENT represents a single data dependency of abstract regions in an instruction. For example, `xchg eax, ebx` creates two assignments: one from pre-instruction `eax` to post-instruction `ebx`, and one from pre-instruction `ebx` to post-instruction `eax`.
- STACK HEIGHT represents the difference between a value in an abstract location and the stack pointer at a function's call site.

3 Examples

We show several examples of how to use DataflowAPI. In these examples, we assume that the mutatee has been parsed and we have function and block objects to analyze. Users may refer to the ParseAPI manual for how to obtain these function and block objects.

3.1 Slicing

The following example uses DataflowAPI to perform a backward slice on an indirect jump instruction to determine the instructions that affect the calculation of the jump target. The goal of this example is to show (1) how to convert an instruction to assignments; (2) how to perform slicing on a given assignment; (3) how to extend the default `Slicer::Predicates` and write call back functions to control the behavior of slicing.

```
#include "Instruction.h"
#include "InstructionDecoder.h"
#include "slicing.h"

using namespace Dyninst;
using namespace ParseAPI;
using namespace InstructionAPI;
using namespace DataflowAPI;

// We extend the default predicates to control when to stop slicing
class ConstantPred : public Slicer::Predicates {
  public:
    // We do not want to track through memory writes
    virtual bool endAtPoint(Assignment::Ptr ap) {
        return ap->insn()->writesMemory();
    }

    // We can treat PC as a constant as its value is the address of the instruction
    virtual bool addPredecessor(AbsRegion reg) {
        if (reg.absloc().type() == Absloc::Register) {
            MachRegister r = reg.absloc().reg();
            return !r.isPC();
        }
        return true;
    }
```

```
};

// Assume that block b in function f ends with an indirect jump.
void AnalyzeJumpTarget(Function *f, Block *b) {
    // Decode the last instruction in this block, which should be a jump
    const unsigned char * buf =
         (const unsigned char*) b->obj()->cs()->getPtrToInstruction(b->last());
    InstructionDecoder dec(buf,
                           InstructionDecoder::maxInstructionLength,
                           b->obj()->cs()->getArch());
    Instruction::Ptr insn = dec.decode();

    // Convert the instruction to assignments
    AssignmentConverter ac(true);
    vector<Assignment::Ptr> assignments;
    ac.convert(insn, b->last(), f, b, assignments);

    // An instruction can corresponds to multiple assignment.
    // Here we look for the assignment that changes the PC.
    Assignment::Ptr pcAssign;
    for (auto ait = assignments.begin(); ait != assignments.end(); ++ait) {
        const AbsRegion &out = (*ait)->out();
        if (out.absloc().type() == Absloc::Register && out.absloc().reg().isPC()) {
            pcAssign = *ait;
            break;
        }
    }

    // Create a Slicer that will start from the given assignment
    Slicer s(pcAssign, b, f);

    // We use the customized predicates to control slicing
    ConstantPred mp;
    GraphPtr slice = s.backwardSlice(mp);
}
```

3.2 Symbolic Evaluation

The following example shows how to expand a slice to ASTs and analyze an AST. Suppose we have a slice representing the instructions that affect the jump target of an indirect jump instruction. We can get the expression of the jump targets and visit the expression to see if it is a constant.

```
#include "SymEval.h"
#include "DynAST.h"

// We extend the default ASTVisitor to check whether the AST is a constant
class ConstVisitor: public ASTVisitor {
  public:
    bool resolved;
```

```
    Address target;
    ConstVisitor() : resolved(true), target(0){}

    // We reach a constant node and record its value
    virtual AST::Ptr visit(DataflowAPI::ConstantAST * ast) {
        target = ast->val().val;
        return AST::Ptr();
    };

    // If the AST contains a variable
    // or an operation, then the control flow target cannot
    // be resolved through constant propagation
    virtual AST::Ptr visit(DataflowAPI::VariableAST *) {
        resolved = false;
        return AST::Ptr();
    };
    virtual AST::Ptr visit(DataflowAPI::RoseAST * ast) {
        resolved = false;

        // Recursively visit all children
        unsigned totalChildren = ast->numChildren();
        for (unsigned i = 0 ; i < totalChildren; ++i) {
            ast->child(i)->accept(this);
        }
        return AST::Ptr();
    };
};

Address ExpandSlice(GraphPtr slice, Assignment::Ptr pcAssign) {
    Result_t symRet;
    SymEval::expand(slice, symRet);

    // We get AST representing the jump target
    AST::Ptr pcExp = symRet[pcAssign];

    // We analyze the AST to see if it can actually be resolved by constant propagation
    ConstVisitor cv;
    pcExp->accept(&cv);
    if (cv.resolved) return cv.target;
    return 0;
}
```

3.3 Liveness Analysis

The following example shows how to query for live registers.

```
#include "Location.h"
#include "liveness.h"
#include "bitArray.h"
```

```
using namespace std;
using namespace Dyninst;
using namespace Dyninst::ParseAPI;

void LivenessAnalysis(Function *f, Block *b) {
    // Construct a liveness analyzer based on the address width of the mutatee.
    // 32-bit code and 64-bit code have different ABI.
    LivenessAnalyzer la(f->obj()->cs()->getAddressWidth());

    // Construct a liveness query location
    Location loc(f, b);

    // Query live registers at the block entry
    bitArray liveEntry;
    if (!la.query(loc, LivenessAnalyzer::Before, liveEntry)) {
        printf("Cannot look up live registers at block entry\n");
    }

    printf("There are %d registers live at the block entry\n", liveEntry.count());

    // Query live register at the block exit
    bitArray liveExit;
    if (!la.query(loc, LivenessAnalyzer::After, liveExit)) {
       printf("Cannot look up live registers at block exit\n");
    }

    printf("rbx is live or not at the block exit: %d\n", liveExit.test(la.getIndex(x86_64::rbx)))
}
```

3.4 Stack Analysis

The following example shows how to use stack analysis to print out all defined stack heights at the first instruction in a block.

```
#include "stackanalysis.h"

void StackHeight(Function *f, Block *b) {
    // Get the address of the first instruction of the block
    Address addr = block->start();

    // Get the stack heights at that address
    StackAnalysis sa(func);
    std::vector<std::pair<Absloc, StackAnalysis::Height>> heights;
    sa.findDefinedHeights(block, addr, heights);

    // Print out the stack heights
    for (auto iter = heights.begin(); iter != heights.end(); iter++) {
        const Absloc &loc = iter->first;
```

```
            const StackAnalysis::Height &height = iter->second;
            printf("%s := %s\n", loc.format().c_str(), height.format().c_str());
        }
19  }
```

4 API Reference

4.1 Class Assignment

Defined in: `Absloc.h`

An assignment represents data dependencies between an output abstract region that is modified by this instruction and several input abstract regions that are used by this instruction. An instruction may modify several abstract regions, so an instruction can correspond to multiple assignments.

```
typedef boost::shared_ptr<Assignment> Ptr;
```

Shared pointer for Assignment class.

```
const std::vector<AbsRegion> &inputs() const;
std::vector<AbsRegion> &inputs();
```

Return the input abstract regions.

```
const AbsRegion &out() const;
AbsRegion &out();
```

Return the output abstract region.

```
InstructionAPI::Instruction::Ptr insn() const;
```

Return the instruction that contains this assignment.

```
Address addr() const;
```

Return the address of this assignment.

```
ParseAPI::Function *func() const;
```

Return the function that contains this assignment.

```
ParseAPI::Block *block() const;
```

Return the block that contains this assignment.

```
const std::string format() const;
```

Return the string representation of this assignment.

4.2 Class AssignmentConverter

Defined in: `AbslocInterface.h`

This class should be used to convert instructions to assignments.

```
AssignmentConverter(bool cache, bool stack = true);
```

Construct an AssignmentConverter. When `cache` is `true`, this object will cache the conversion results for converted instructions. When `stack` is `true`, stack analysis is used to distinguish stack variables at different offset. When `stack` is `false`, the stack is treated as a single memory region.

```
void convert(InstructionAPI::Instruction::Ptr insn,
             const Address &addr,
             ParseAPI::Function *func,
             ParseAPI::Block *blk,
             std::vector<Assignment::Ptr> &assign);
```

Convert instruction `insn` to assignments and return these assignments in `assign`. The user also needs to provide the context of `insn`, including its address `addr`, function `func`, and block `blk`.

4.3 Class Absloc

Defined in: `Absloc.h`

Class Absloc represents an abstract location. Abstract locations can have the following types

Type	Meaning
Register	The abstract location represents a register
Stack	The abstract location represents a stack variable
Heap	The abstract location represents a heap variable
Unknown	The default type of abstract location

```
static Absloc makePC(Dyninst::Architecture arch);
static Absloc makeSP(Dyninst::Architecture arch);
static Absloc makeFP(Dyninst::Architecture arch);
```

Shortcut interfaces for creating abstract locations representing PC, SP, and FP

```
bool isPC() const;
bool isSP() const;
bool isFP() const;
```

Check whether this abstract location represents a PC, SP, or FP.

```
Absloc();
```

Create an Unknown type abstract location.

```
Absloc(MachRegister reg);
```

Create a Register type abstract location, representing register `reg`.

```
Absloc(Address addr):
```

Create a Heap type abstract location, representing a heap variable at address `addr`.

```
Absloc(int o,
       int r,
       ParseAPI::Function *f);
```

Create a Stack type abstract location, representing a stack variable in the frame of function `f`, within abstract region `r`, and at offset `o` within the frame.

```
std::string format() const;
```

Return the string representation of this abstract location.

```
const Type& type() const;
```

Return the type of this abstract location.

```
bool isValid() const;
```

Check whether this abstract location is valid or not. Return `true` when the type is not Unknown.

```
const MachRegister &reg() const;
```

Return the register represented by this abstract location. This method should only be called when this abstract location truly represents a register.

```
int off() const;
```

Return the offset of the stack variable represented by this abstract location. This method should only be called when this abstract location truly represents a stack variable.

```
int region() const;
```

Return the region of the stack variable represented by this abstract location. This method should only be called when this abstract location truly represents a stack variable.

```
ParseAPI::Function *func() const;
```

Return the function of the stack variable represented by this abstract location. This method should only be called when this abstract location truly represents a stack variable.

```
Address addr() const;
```

Return the address of the heap variable represented by this abstract location. This method should only be called when this abstract location truly represents a heap variable.

```
bool operator<(const Absloc &rhs) const;
bool operator==(const Absloc &rhs) const;
bool operator!=(const Absloc &rhs) const;
```

Comparison operators

4.4 Class AbsRegion

Defined in: `Absloc.h`

Class AbsRegion represents a set of abstract locations of the same type.

```
AbsRegion();
```

Create a default abstract region.

```
AbsRegion(Absloc::Type t);
```

Create an abstract region representing all abstract locations with type `t`.

```
AbsRegion(Absloc a);
```

Create an abstract region representing a single abstract location `a`.

```
bool contains(const Absloc::Type t) const;
bool contains(const Absloc &abs) const;
bool contains(const AbsRegion &rhs) const;
```

Return `true` if this abstract region contains abstract locations of type `t`, contains abstract location `abs`, or contains abstract region `rhs`.

```
bool containsOfType(Absloc::Type t) const;
```

Return `true` if this abstract region contains abstract locations in type `t`.

```
bool operator==(const AbsRegion &rhs) const;
bool operator!=(const AbsRegion &rhs) const;
bool operator<(const AbsRegion &rhs) const;
```

Comparison operators

```
const std::string format() const;
```

Return the string representation of the abstract region.

```
Absloc absloc() const;
```

Return the abstract location in this abstract region.

```
Absloc::Type type() const;
```

Return the type of this abstract region.

```
AST::Ptr generator() const;
```

If this abstract region represents memory locations, this method returns address calculation of the memory access.

```
bool isImprecise() const;
```

Return `true` if this abstract region represents more than one abstract locations.

4.5 Class AbsRegionConverter

Defined in: `AbslocInterface.h`

Class AbsRegionConverter converts instructions to abstract regions.

```
AbsRegionConverter(bool cache, bool stack = true);
```

Create an AbsRegionConverter. When `cache` is `true`, this object will cache the conversion results for converted instructions. When `stack` is `true`, stack analysis is used to distinguish stack variables at different offsets. When `stack` is `false`, the stack is treated as a single memory region.

```
void convertAll(InstructionAPI::Expression::Ptr expr,
                Address addr,
                ParseAPI::Function *func,
                ParseAPI::Block *block,
                std::vector<AbsRegion> &regions);
```

Create all abstract regions used in `expr` and return them in `regions`. All registers appear in `expr` will have a separate abstract region. If the expression represents a memory access, we will also create a heap or stack abstract region depending on where it accesses. `addr`, `func`, and `blocks` specify the contexts of the expression. If PC appears in this expression, we assume the expression is at address `addr` and replace PC with a constant value `addr`.

```
void convertAll(InstructionAPI::Instruction::Ptr insn,
                Address addr,
                ParseAPI::Function *func,
                ParseAPI::Block *block,
                std::vector<AbsRegion> &used,
                std::vector<AbsRegion> &defined);
```

Create abstract regions appearing in instruction `insn`. Input abstract regions of this instructions are returned in `used` and output abstract regions are returned in `defined`. If the expression represents a memory access, we will also create a heap or stack abstract region depending on where it accesses. `addr`, `func`, and `blocks` specify the contexts of the expression. If PC appears in this expression, we assume the expression is at address `addr` and replace PC with a constant value `addr`.

```
AbsRegion convert(InstructionAPI::RegisterAST::Ptr reg);
```

Create an abstract region representing the register `reg`.

```
AbsRegion convert(InstructionAPI::Expression::Ptr expr,
                  Address addr,
                  ParseAPI::Function *func,
                  ParseAPI::Block *block);
```

Create and return the single abstract region represented by `expr`.

4.6 Class Graph

Defined in: `Graph.h`

We provide a generic graph interface, which allows users to add, delete, and iterate nodes and edges in a graph. Our slicing algorithms are implemented upon this graph interface, so users can inherit the defined classes for customization.

```
typedef boost::shared_ptr<Graph> Ptr;
```

Shared pointer for Graph

```
virtual void entryNodes(NodeIterator &begin, NodeIterator &end);
```

The entry nodes (nodes without any incoming edges) of the graph.

```
virtual void exitNodes(NodeIterator &begin, NodeIterator &end);
```

The exit nodes (nodes without any outgoing edges) of the graph.

```
virtual void allNodes(NodeIterator &begin, NodeIterator &end);
```

Iterate all nodes in the graph.

```
bool printDOT(const std::string& fileName);
```

Output the graph in dot format.

```
static Graph::Ptr createGraph();
```

Return an empty graph.

```
void insertPair(NodePtr source, NodePtr target, EdgePtr edge = EdgePtr());
```

Insert a pair of nodes into the graph and create a new edge `edge` from `source` to `target`.

```
virtual void insertEntryNode(NodePtr entry);
virtual void insertExitNode(NodePtr exit);
```

Insert a node as an entry/exit node

```
virtual void markAsEntryNode(NodePtr entry);
virtual void markAsExitNode(NodePtr exit);
```

Mark a node that has been added to this graph as an entry/exit node.

```
void deleteNode(NodePtr node);
void addNode(NodePtr node);
```

Delete / Add a node.

```
bool isEntryNode(NodePtr node);
bool isExitNode(NodePtr node);
```

Check whether a node is an entry / exit node

```
void clearEntryNodes();
void clearExitNodes();
```

Clear the marking of entry / exit nodes. Note that the nodes are not deleted from the graph.

```
unsigned size() const;
```

Return the number of nodes in the graph.

4.7 Class Node

Defined in: `Node.h`

```
typedef boost::shared_ptr<Node> Ptr;
```

Shared pointer for Node

```
void ins(EdgeIterator &begin, EdgeIterator &end);
void outs(EdgeIterator &begin, EdgeIterator &end);
```

Iterate over incoming/outgoing edges of this node.

```
void ins(NodeIterator &begin, NodeIterator &end);
void outs(NodeIterator &begin, NodeIterator &end);
```

Iterate over adjacent nodes connected with incoming/outgoing edges of this node.

```
bool hasInEdges();
bool hasOutEdges();
```

Return `true` if this node has incoming/outgoing edges.

```
void deleteInEdge(EdgeIterator e);
void deleteOutEdge(EdgeIterator e);
```

Delete an incoming/outgoing edge.

```
virtual Address addr() const;
```

Return the address of this node.

```
virtual std::string format() const = 0;
```

Return the string representation.

```
class NodeIterator;
```

Iterator for nodes. Common iterator operations including `++`, `-`, and dereferencing are supported.

4.8 Class Edge

Defined in: `Edge.h`

```
typedef boost::shared_ptr<Edge> Edge::Ptr;
```

Shared pointer for `Edge`.

```
static Edge::Ptr Edge::createEdge(const Node::Ptr source, const Node::Ptr target);
```

Create a new directed edge from `source` to `target`.

```
Node::Ptr Edge::source() const;
Node::Ptr Edge::target() const;
```

Return the source / target node.

```
void Edge::setSource(Node::Ptr source);
void Edge::setTarget(Node::Ptr target);
```

Set the source / target node.

```
class EdgeIterator;
```

Iterator for edges. Common iterator operations including `++`, `-`, and dereferencing are supported.

4.9 Class Slicer

Defined in: `slicing.h`

Class Slicer is the main interface for performing forward and backward slicing. The slicing algorithm starts with a user provided Assignment and generates a graph as the slicing results. The nodes in the generated Graph are individual assignments that affect the starting assignment (backward slicing) or are affected by the starting assignment (forward slicing). The edges in the graph are directed and represent either data flow dependencies or control flow dependencies.

We provide call back functions and allow users to control when to stop slicing. In particular, class `Slicer::Predicates` contains a collection of call back functions that can control the specific behaviors of the slicer. Users can inherit from the Predicates class to provide customized stopping criteria for the slicer.

```
Slicer(AssignmentPtr a,
       ParseAPI::Block *block,
       ParseAPI::Function *func,
       bool cache = true,
       bool stackAnalysis = true);
```

Construct a slicer, which can then be used to perform forward or backward slicing starting at the assignment `a`. `block` and `func` represent the context of assignment `a`. `cache` specifies whether the slicer will cache the results of conversions from instructions to assignments. `stackAnalysis` specifies whether the slicer will invoke stack analysis to distinguish stack variables.

```
GraphPtr forwardSlice(Predicates &predicates);
GraphPtr backwardSlice(Predicates &predicates);
```

Perform forward or backward slicing and use `predicates` to control the stopping criteria and return the slicing results as a graph

A slice is represented as a Graph. The nodes and edges are defined as below:

```
class SliceNode : public Node
```

The default node data type in a slice graph.

```
typedef boost::shared_ptr<SliceNode> Ptr;
static SliceNode::Ptr SliceNode::create(AssignmentPtr ptr,
                                        ParseAPI::Block *block,
                                        ParseAPI::Function *func);
```

Create a slice node, which represents assignment `ptr` in basic block `block` and function `func`.

Class SliceNode has the following methods to retrieve information associated the node:

Method name	Return type	Method description
block	ParseAPI::Block*	Basic block of this SliceNode.
func	ParseAPI::Function*	Function of this SliceNode.
addr	Address	Address of this SliceNode.
assign	Assignment::Ptr	Assignment of this SliceNode.
format	std::string	String representation of this SliceNode.

```
class SliceEdge : public Edge
```

The default edge data type in a slice graph.

```
typedef boost::shared_ptr<SliceEdge> Ptr;
static SliceEdge::Ptr create(SliceNode::Ptr source,
                             SliceNode::Ptr target,
                             AbsRegion const&data);
```

Create a slice edge from `source` to `target` and the edge presents a dependency about abstract region `data`.

```
const AbsRegion &data() const;
```

Get the data annotated on this edge.

4.10 Class Slicer::Predicates

Defined in: `slicing.h`

Class Predicates abstracts the stopping criteria of slicing. Users can inherit this class to control slicing in various situations, including whether or not to perform inter-procedural slicing, whether or not to search for control flow dependencies, and whether or not to stop slicing after discovering certain assignments. We provide a set of call back functions that allow users to dynamically control the behavior of the Slicer.

```
Predicates();
```

Construct a default predicate, which will only search for intraprocedural data flow dependencies.

```
bool searchForControlFlowDep();
```

Return `true` if this predicate will search for control flow dependencies. Otherwise, return `false`.

```
void setSearchForControlFlowDep(bool cfd);
```

Change whether or not to search for control flow dependencies according to `cfd`.

```
virtual bool widenAtPoint(AssignmentPtr) { return false; }
```

The default behavior is to return `false`.

```
virtual bool endAtPoint(AssignmentPtr);
```

In backward slicing, after we find a match for an assignment, we pass it to this function. This function should return `true` if the user does not want to continue searching for this assignment. Otherwise, it should return `false`. The default behavior of this function is to always return `false`.

```
typedef std::pair<ParseAPI::Function *, int> StackDepth_t;
typedef std::stack<StackDepth_t> CallStack_t;
virtual bool followCall(ParseAPI::Function * callee,
                        CallStack_t & cs,
                        AbsRegion argument);
```

This predicate function is called when the slicer reaches a direct call site. If it returns `true`, the slicer will follow into the callee function `callee`. This function also takes input `cs`, which represents the call stack of the followed callee functions from the starting point of the slicing to this call site, and `argument`, which represents the variable to slice with in the callee function. This function defaults to always returning `false`. Note that as Dyninst currently does not try to resolve indirect calls, the slicer will NOT call this function at an indirect call site.

```
virtual std::vector<ParseAPI::Function *>
          followCallBackward(ParseAPI::Block * caller,
                             CallStack_t & cs,
                             AbsRegion argument);
```

This predicate function is called when the slicer reaches the entry of a function in the case of backward slicing or reaches a return instruction in the case of forward slicing. It returns a vector of caller functions that the user wants the slicer to continue to follow. This function takes input `caller`, which represents the call block of the caller, `cs`, which represents the caller functions that have been followed to this place, and `argument`, which represents the variable to slice with in the caller function. This function defaults to always returning an empty vector.

```
virtual bool addPredecessor(AbsRegion reg);
```

In backward slicing, after we match an assignment at a location, the matched AbsRegion `reg` is passed to this predicate function. This function should return `true` if the user wants to continue to search for dependencies for this AbsRegion. Otherwise, this function should return `true`. The default behavior of this function is to always return `true`.

```
virtual bool addNodeCallback(AssignmentPtr assign,
                             std::set<ParseAPI::Edge*> &visited);
```

In backward slicing, this function is called when the slicer adds a new node to the slice. The newly added assignment `assign` and the set of control flow edges `visited` that have been visited so far are passed to this function. This function should return `true` if the user wants to continue slicing. If this function returns `false`, the Slicer will not continue to search along the path. The default behavior of this function is to always return `true`.

4.11 Class StackAnalysis

The StackAnalysis interface is used to determine the possible stack heights of abstract locations at any instruction in a function. Due to there often being many paths through the CFG to reach a given instruction, abstract locations may have different stack heights depending on the path taken to reach that instruction. In other cases, StackAnalysis is unable to adequately determine what is contained in an abstract location. In both situations, StackAnalysis is conservative in its reported stack heights. The table below explains what the reported stack heights mean.

Reported stack height	Meaning
TOP	On all paths to this instruction, the specified abstract location contains a value that does not point to the stack.
x (some number)	On at least one path to this instruction, the specified abstract location has a stack height of x. On all other paths, the abstract location either has a stack height of x or doesn't point to the stack.
BOTTOM	There are three possible meanings: 1. On at least one path to this instruction, StackAnalysis was unable to determine whether or not the specified abstract location points to the stack. 2. On at least one path to this instruction, StackAnalysis determined that the specified abstract location points to the stack but could not determine the exact stack height. 3. On at least two paths to this instruction, the specified abstract location pointed to different parts of the stack.

```
StackAnalysis(ParseAPI::Function *f)
```

Constructs a StackAnalysis object for function `f`.

```
StackAnalysis::Height find(ParseAPI::Block *b, Address addr, Absloc loc)
```

Returns the stack height of abstract location `loc` before execution of the instruction with address `addr` contained in basic block `b`. The address `addr` must be contained in block `b`, and block `b` must be contained in the function used to create this StackAnalysis object.

```
StackAnalysis::Height findSP(ParseAPI::Block *b, Address addr)
StackAnalysis::Height findFP(ParseAPI::Block *b, Address addr)
```

Returns the stack height of the stack pointer and frame pointer, respectively, before execution of the instruction with address `addr` contained in basic block `b`. The address `addr` must be contained in block `b`, and block `b` must be contained in the function used to create this StackAnalysis object.

```
void findDefinedHeights(ParseAPI::Block *b,
                        Address addr,
                        std::vector<std::pair<Absloc, StackAnalysis::Height>> &heights)
```

Writes to the vector `heights` all defined <abstract location, stack height> pairs before execution of the instruction with address `addr` contained in basic block `b`. Note that abstract locations with stack heights of TOP (i.e. they do not point to the stack) are not written to `heights`. The address `addr` must be contained in block `b`, and block `b` must be contained in the function used to create this StackAnalysis object.

4.12 Class StackAnalysis::Height

Defined in: `stackanalysis.h`

The Height class is used to represent the abstract notion of stack heights. Every Height object represents a stack height of either TOP, BOTTOM, or x, where x is some integral number. The Height class also defines methods for comparing, combining, and modifying stack heights in various ways.

```
typedef signed long Height_t
```

The underlying data type used to convert between Height objects and integral values.

Method name	Return type	Method description
height	Height_t	This stack height as an integral value.
format	std::string	This stack height as a string.
isTop	bool	True if this stack height is TOP.
isBottom	bool	True if this stack height is BOTTOM.

```
Height(const Height_t h)
```

Creates a Height object with stack height `h`.

```
Height()
```

Creates a Height object with stack height TOP.

```
bool operator<(const Height &rhs) const
bool operator>(const Height &rhs) const
bool operator<=(const Height &rhs) const
bool operator>=(const Height &rhs) const
bool operator==(const Height &rhs) const
bool operator!=(const Height &rhs) const
```

Comparison operators for Height objects. Compares based on the integral stack height treating TOP as MAX_HEIGHT and BOTTOM as MIN_HEIGHT.

```
Height &operator+=(const Height &rhs)
Height &operator+=(const signed long &rhs)
const Height operator+(const Height &rhs) const
const Height operator+(const signed long &rhs) const
const Height operator-(const Height &rhs) const
```

Returns the result of basic arithmetic on Height objects according to the following rules, where x and y are integral stack heights and S represents any stack height:

- $TOP + TOP = TOP$
- $TOP + x = BOTTOM$
- $x + y = (x + y)$
- $BOTTOM + S = BOTTOM$

Note that the subtraction rules can be obtained by replacing all + signs with - signs.

The `operator+` and `operator-` methods leave this Height object unmodified while the `operator+=` methods update this Height object with the result of the computation. For the methods where `rhs` is a `const signed long`, it is not possible to set `rhs` to TOP or BOTTOM.

4.13 Class AST

Defined in: `DynAST.h`

We provide a generic AST framework to represent tree structures. One example use case is to represent instruction semantics with symbolic expressions. The AST framework includes the base class definitions for tree nodes and visitors. Users can inherit tree node classes to create their own AST structure and AST visitors to write their own analyses for the AST.

All AST node classes should be derived from the AST class. Currently we have the following types of AST nodes.

AST::ID	Meaning
V_AST	Base class type
V_BottomAST	Bottom AST node
V_ConstantAST	Constant AST node
V_VariableAST	Variable AST node
V_RoseAST	ROSEOperation AST node
V_StackAST	Stack AST node

```
typedef boost::shared_ptr<AST> Ptr;
```

Shared pointer for class AST.

```
typedef std::vector<AST::Ptr> Children;
```

The container type for the children of this AST.

```
bool operator==(const AST &rhs) const;
bool equals(AST::Ptr rhs);
```

Check whether two AST nodes are equal. Return `true` when two nodes are in the same type and are equal according to the `==` operator of that type.

```
virtual unsigned numChildren() const;
```

Return the number of children of this node.

```
virtual AST::Ptr child(unsigned i) const;
```

Return the `i`th child.

```
virtual const std::string format() const = 0;
```

Return the string representation of the node.

```
static AST::Ptr substitute(AST::Ptr in, AST::Ptr a, AST::Ptr b);
```

Substitute every occurrence of `a` with `b` in AST `in`. Return a new AST after the substitution.

```
virtual AST::ID AST::getID() const;
```

Return the class type ID of this node.

```
virtual Ptr accept(ASTVisitor *v);
```

Apply visitor `v` to this node. Note that this method will not automatically apply the visitor to its children.

```
virtual void AST::setChild(int i, AST::Ptr c);
```

Set the `i`th child of this node to `c`.

4.14 Class SymEval

Defined in: `SymEval.h`

Class SymEval provides interfaces for expanding an instruction to its symbolic expression and expanding a slice graph to symbolic expressions for all abstract locations defined in this slice.

```
typedef std::map<Assignment::Ptr, AST::Ptr, AssignmentPtrValueComp> Result_t;
```

This data type represents the results of symbolic expansion of a slice. Each assignment in the slice has a corresponding AST.

```
static std::pair<AST::Ptr, bool> expand(const Assignment::Ptr &assignment,
                                        bool applyVisitors = true);
```

This interface expands a single assignment given by `assignment` and returns a `std::pair`, in which the first element is the AST after expansion and the second element is a bool indicating whether the expansion succeeded or not. `applyVisitors` specifies whether or not to perform stack analysis to precisely track stack variables.

```
static bool expand(Result_t &res,
                   std::set<InstructionPtr> &failedInsns,
   bool applyVisitors = true);
```

This interface expands a set of assignment prepared in `res`. The corresponding ASTs are written back into `res` and all instructions that failed during expansion are inserted into `failedInsns`. `applyVisitors` specifies whether or not to perform stack analysis to precisely track stack variables. This function returns `true` when all assignments in `res` are successfully expanded.

Retval_t	Meaning
FAILED	failed
WIDEN_NODE	widen
FAILED_TRANSLATION	failed translation
SKIPPED_INPUT	skipped input
SUCCESS	success

```
static Retval_t expand(Dyninst::Graph::Ptr slice, DataflowAPI::Result_t &res);
```

This interface expands a slice and returns an AST for each assignment in the slice. This function will perform substitution of ASTs.

We use an AST to represent the symbolic expressions of an assignment. A symbolic expression AST contains internal node type `RoseAST`, which abstracts the operations performed with its child nodes, and two leave node types: `VariableAST` and `ConstantAST`.

`RoseAST`, `VariableAST`, and `ConstantAST` all extend class `AST`. Besides the methods provided by class `AST`, `RoseAST`, `VariableAST`, and `ConstantAST` each have a different data structure associated with them.

```
Variable& VariableAST::val() const;
Constant& ConstantAST::val() const;
ROSEOperation & RoseAST::val() const;
```

We now describe data structure `Variable`, `Constant`, and `ROSEOperation`.

```
struct Variable;
```

A `Variable` represents an abstract region at a particular address.

```
Variable::Variable();
Variable::Variable(AbsRegion r);
Variable::Variable(AbsRegion r, Address a);
```

The constructors of class Variable.

```
bool Variable::operator==(const Variable &rhs) const;
bool Variable::operator<(const Variable &rhs) const;
```

Two Variable objects are equal when their AbsRegion are equal and their addresses are equal.

```
const std::string Variable::format() const;
```

Return the string representation of the Variable.

```
AbsRegion Variable::reg;
Address Variable::addr;
```

The abstraction region and the address of this Variable.

```
struct Constant;
```

A `Constant` object represents a constant value in code.

```
Constant::Constant();
Constant::Constant(uint64_t v);
Constant::Constant(uint64_t v, size_t s);
```

Construct Constant objects.

```
bool Constant::operator==(const Constant &rhs) const;
bool Constant::operator<(const Constant &rhs) const;
```

Comparison operators for Constant objects. Comparison is based on the value and size.

```
const std::string Constant::format() const;
```

Return the string representation of the Constant object.

```
uint64_t Constant::val;
size_t Constant::size;
```

The numerical value and bit size of this value.

```
struct ROSEOperation;
```

`ROSEOperation` defines the following operations and we represent the semantics of all instructions with these operations.

ROSEOperation::Op	Meaning
nullOp	No operation
extractOp	Extract bit ranges from a value
invertOp	Flip every bit
negateOp	Negate the value
signExtendOp	Sign-extend the value
equalToZeroOp	Check whether the value is zero or not
generateMaskOp	Generate mask
LSBSetOp	LSB set op
MSBSetOp	MSB set op
concatOp	Concatenate two values to form a new value
andOp	Bit-wise and operation
orOp	Bit-wise or operation
xorOp	Bit-wise xor operation
addOp	Add operation
rotateLOp	Rotate to left operation
rotateROp	Rotate to right operation
shiftLOp	Shift to left operation
shiftROp	Shift to right operation
shiftRArithOp	Arithmetic shift to right operation
derefOp	Dereference memory operation
writeRepOp	Write rep operation
writeOp	Write operation
ifOp	If operation
sMultOp	Signed multiplication operation
uMultOp	Unsigned multiplication operation
sDivOp	Signed division operation
sModOp	Signed modular operation
uDivOp	Unsigned division operation
uModOp	Unsigned modular operation
extendOp	Zero extend operation
extendMSBOp	Extend the most significant bit operation

```
ROSEOperation::ROSEOperation(Op o) : op(o);
ROSEOperation::ROSEOperation(Op o, size_t s);
```

Constructors for ROSEOperation

```
bool ROSEOperation::operator==(const ROSEOperation &rhs) const;
```

Equal operator

```
const std::string ROSEOperation::format() const;
```

Return the string representation.

```
ROSEOperation::Op ROSEOperation::op;
size_t ROSEOperation::size;
```

4.15 Class ASTVisitor

The ASTVisitor class defines callback functions to apply during visiting an AST for each AST node type. Users can inherit from this class to write customized analyses for ASTs.

```
typedef boost::shared_ptr<AST> ASTVisitor::ASTPtr;
virtual ASTVisitor::ASTPtr ASTVisitor::visit(AST *);
virtual ASTVisitor::ASTPtr ASTVisitor::visit(DataflowAPI::BottomAST *);
virtual ASTVisitor::ASTPtr ASTVisitor::visit(DataflowAPI::ConstantAST *);
virtual ASTVisitor::ASTPtr ASTVisitor::visit(DataflowAPI::VariableAST *);
virtual ASTVisitor::ASTPtr ASTVisitor::visit(DataflowAPI::RoseAST *);
virtual ASTVisitor::ASTPtr ASTVisitor::visit(StackAST *);
```

Callback functions for visiting each type of AST node. The default behavior is to return the input parameter.

Paradyn Parallel Performance Tools

ProcControlAPI Programmer's Guide

Release 9.2
June 2016

Computer Science Department
University of Wisconsin-Madison
Madison, WI 53706

Computer Science Department
University of Maryland
College Park, MD 20742
Email: dyninst-api@cs.wisc.edu

WEB: WWW.DYNINST.ORG
github.com/dyninst/dyninst

1. Introduction

This document describes ProcControlAPI, an API and library for controlling processes. ProcControlAPI runs as part of a *controller process* and manages one or more *target processes*. It allows the controller process to perform operations on target processes, such as writing to memory, stopping and running threads, or receiving notification when certain events occur. ProcControlAPI presents these operations through a platform-independent API and high-level abstractions. Users can describe what they want ProcControlAPI to do, and ProcControlAPI handles the details.

An example use for ProcControlAPI would be as the underlying mechanism for a debugger. A user writing a debugger could provide their own user interface and debugging strategies, while using ProcControlAPI to perform operations such as creating processes, running threads, and handling breakpoints.

ProcControlAPI exposes a C++ interface. This document assumes some familiarity with several concepts from C++, such as const types, iterators, and inheritance.

The interface for ProcControlAPI can be generally divided into two parts: an interface for managing a process (e.g., reading and writing to target process memory, stopping and running threads), and an interface for monitoring a target process for certain events (e.g., watching the target process for fork or thread creation events). The manager interface uses a set of C++ objects to represent a target process and its threads, libraries, registers and other interesting aspects. Operations performed on these C++ objects in the controller process are translated into corresponding operations on the target process. The event interface uses a callback system to notify the ProcControlAPI user of interesting events in the target process.

1.1. Simple Example

As an example, consider the code in Figure 1 that creates a target process and prints a message whenever that target process creates a new thread. Details on the API function used in this example can be found in latter sections of this manual, but we will provide a high level description of the operations here. Note that proper error handling and checking have been left out for brevity.

1. We start by parsing the arguments passed to the controller process, turning them into arguments that will be passed to the new target process.

```
#include "PCProcess.h"
#include "Event.h"
#include <iostream>
#include <string>

using namespace Dyninst;
using namespace ProcControlAPI;
using namespace std;

Process::cb_ret_t on_thread_create(Event::const_ptr ev) {
   //Callback when the target process creates a thread.
   EventNewThread::const_ptr new_thrd_ev = ev->getEventNewThread();
   Thread::const_ptr new_thrd = new_thrd_ev->getNewThread();

   cout << "Got a new thread with LWP " << new_thrd->getLWP() << endl;
   return Process::cbDefault;
}

int main(int argc, char *argv[]) {
   vector<string> args;

   //Create a new target process
   string exec = argv[1];
   for (unsigned i=1; i<argc; i++)
      args.push_back(std::string(argv[i]));
   Process::ptr proc = Process::createProcess(exec, args);

   //Tell ProcControlAPI about our callback function
   Process::registerEventCallback(EventType::ThreadCreate, on_thread_create);

   //Run the process and wait for it to terminate.
   proc->continueProc();
   while (!proc->isTerminated())
      Process::handleEvents(true);

   return 0;
}
```

Figure 1

2. We ask ProcControlAPI to create a new Process using the given arguments. ProcControlAPI will spawn a new target process and leave it in a stopped state to prevent it from executing.
3. After creating the new target process we register a callback function. We ask ProcControlAPI to call our function, `on_thread_create`, when an event of type `EventType::ThreadCreate` occurs in the target process.
4. The `on_thread_create` function takes a pointer to an object of type `Event` and returns a `Process::cb_ret_t`. The Event describes the target process event that triggered this callback. In this case, it provides information about the new thread in the target process. It is worth noting that `Event::const_ptr` is a not a regular pointer, but a reference counted shared pointer. This means that we do not have to be concerned with cleaning the Event—it will be automatically cleaned when the last reference disappears. The `Process::cb_ret_t` describes what action should be taken on the process in response to this event, which is described in more detail in section 6.

5. The `Event` class has several child classes, one of which is `EventNewThread`. We start by casting the Event into an `EventNewThread` and then extract information about the new thread from the `EventNewThread`.
6. In step 6, we've finished handling the new thread event and need to tell ProcControlAPI what to do in response to this event. For example, we could choose to stop the process from further execution by returning a value of Process::cbProcStop. Instead, we choose let ProcControlAPI take its default action for an EventNewThread by returning Process::cbDefault, which is to continue the process and its new thread (which were both stopped before delivery of the callback).
7. The registering of our callback in step 3 did not actually trigger any calls to the callback function—the target process was created in a stopped state and has not yet been able to create any threads. We tell ProcControlAPI to continue the target process in this step, which allows it to execute and possibly start generating new events.
8. In this step we wait for the target process to finish executing and terminate. Calling `Process::handleEvents` blocks the controller process until an event occurs, allowing us to wait for events without needing to spin the controller process on the CPU.

2. Important Concepts

This section focuses on some of the more important concepts in ProcControlAPI and gives a high level overview before the detailed API is presented in Section 3.

2.1. Processes and Threads

There are two central classes to ProcControlAPI, `Process` and `Thread`. Each class respectively represents a single target process or thread running on the system. By performing operations on the `Process` and `Thread` objects, a ProcControlAPI user is able to control the target process and its threads.

Each `Process` is guaranteed to have at least one `Thread` associated with it. A multi-threaded process may have a `Process` object with more than one `Thread`. Each process has an address space associated with it, which can be written or read through the `Process` object. Each thread has a set of registers associated with it, which can be access through the `Thread` object.

At any one time a `Thread` will be in either a *stopped state* or a *running state*. A thread in a stopped state has had its execution paused by ProcControlAPI—the OS will not schedule the thread to run. A thread in a running state is allowed to execute as normal. A thread in a running state may block for other reasons, e.g. blocking on IO calls, but this does not affect ProcControlAPI's view of the thread state. A thread is only in the stopped state if ProcControlAPI has explicitly stopped it.

A `Process` object is not considered to have a stopped or running state—only its `Thread` objects are stopped or running. A stop operation on a `Process` triggers a stop operation on each of its `Threads`, and similarly a continue operation on a `Process` triggers continue operations on each `Thread`.

2.2. Callbacks

In addition to controlling a target process through the Process and Thread objects, a ProcControlAPI user can also receive notification of events that happen in that process. Examples of these events would be a new thread being created, a breakpoint being executed, or a process exiting.

The ProcControlAPI user receives notice of events through a callback system. The user can register callback function that will be called by ProcControlAPI whenever a particular type of event occurs. Details about the event are passed to the callback function via an `Event` object.

2.2.1. Events

Each event can be broken up into an `EventType` object and an `Event` object. The `EventType` describes a type of event that can happen, and `Event` describes a specific instance of an event happening. Each `Event` will have one and only one `EventType`.

Each `EventType` has two primary fields: its time and its code. The code field of describes what type of event occurred, e.g. `EventType::Exit` represents a target process exiting. The time field of an `EventType` represents whether the `EventType` is happening

before or after will have code and will have a value of `EventType::Pre`, `EventType::Post`, or `EventType::None`.

For example, an `EventType` with time and code of `EventType::Pre` and `EventType::Exit` will occur just before a target process exits, and a code of `EventType::Exec` with a time of `EventType::Post` will occur after an exec system call occurs. In this document we will abbreviate `EventTypes` such as these as pre-exit and post-exec. Some `EventTypes` do not have a time associated with them, for example `EventType::Breakpoint` does not have an associated time and thus has a time value of `EventType::none`.

An `Event` represents an instance of an `EventType` occurring. In addition to an `EventType`, each `Event` also has pointer to the `Process` and `Thread` that it occurred on. Certain events may also have event specific information associated with them, which is represented in a sub-class of `Event`. Each EventType is associated with a specific sub-class of Event.

For example, `EventType::Library` is used to signify a shared library being loaded into the target process. When an `EventType::Library` occurs ProcControlAPI will deliver an object of type `EventLibrary`, which is a subclass of `Event`, to any registered callback functions. In addition to the information inherited from `Event`, the `EventLibrary` will contain extra information about the library that was loaded into the target process.

Table 1 shows the `Event` subclass that is used for each `EventType`. Not all `EventTypes` are available on every platform—a checkmark under the specific OS column means that the `EventType` is available on that OS.

EventType	**Event Subclass**	**Linux**	**FreeBSD**	**Windows**	**BG/Q**
Stop	EventStop	✓			
Breakpoint	EventBreakpoint	✓	✓	✓	✓
Signal	EventSignal	✓	✓	✓	✓
UserThreadCreate	EventNewUserThread	✓	✓		✓
LWPCreate	EventNewLWP	✓		✓	
Pre-UserThreadDestroy	EventUserThreadDestroy	✓	✓		✓
Post-UserThreadDestroy	EventUserThreadDestroy	✓	✓		
Pre-LWPDestroy	EventLWPDestroy	✓		✓	
Post-LWPDestroy	EventLWPDestroy	✓			
Pre-Fork	EventFork				
Post-Fork	EventFork	✓			
Pre-Exec	EventExec				
Post-Exec	EventExec	✓	✓		
RPC	EventRPC	✓	✓	✓	✓
SingleStep	EventSingleStep	✓	✓	✓	✓
Breakpoint	EventBreakpoint	✓	✓	✓	✓
Library	EventLibrary	✓	✓	✓	✓
Pre-Exit	EventExit	✓			
Post-Exit	EventExit	✓	✓	✓	✓
Crash	EventCrash	✓	✓	✓	✓
ForceTerminate	EventForceTerminate	✓	✓	✓	

Table 1 – EventTypes and Events

Details about specific events can be found in Section 3.14.

2.2.2. Callback Functions

Events are delivered via a callback function. A ProcControlAPI user can register callback functions for an `EventType` using the `Process::registerEventCallback` function. All callback functions must be declared using the signature:

```
Process::cb_ret_t callback_func_name(Event::ptr ev)
```

In order to prevent a class of race conditions, ProcControlAPI does not allow a callback function to perform any operation that would require another callback to be recursively delivered. At most one callback function can be running at a time.

To enforce this, the event that is passed to a callback function contains only const pointers to the triggering `Process` and `Thread` objects. Any member function that could trigger callbacks is not marked const, thus triggering a compilation error if they are called on an object passed to a callback. If the ProcControlAPI user uses const_cast or global variables to get around the const restriction it will result in a runtime error. API functions that cannot be used from a callback are mentioned in the API entries.

Operations such as `Process::stopProc`, `Process::continueProc`, `Thread::stopThread`, and `Thread::continueThread` are not safe to call from a callback function, but it would still be useful to perform these operations. ProcControlAPI allows the user to use the return value from a callback function to specify whether process or thread that triggered the event should be stopped or continued. More details on this can be found in the `Process::cb_ret_t` section of the API reference.

2.2.3. Callback Delivery

When ProcControlAPI needs to deliver a callback it must first gain control of a user visible thread in the controller process. This thread will be used to invoke the callback function. ProcControlAPI does not use its internal threads for delivering callbacks, as this would expose the ProcControlAPI user to race conditions.

Unfortunately, the user thread is not always accessible to ProcControlAPI when it needs to invoke a callback function. For example, the user visible thread may be performing network IO or waiting for input from a GUI when an event occurs.

ProcControlAPI uses a notification system built around the `EventNotify` class to alert the ProcControlAPI user that a callback is ready to be delivered. Once the user is notified then they can call the `Process::handleEvents` function, under which ProcControlAPI will invoke any pending callback functions.

The `EventNotify` class has two mechanisms for notifying the ProcControlAPI user that a callback is pending: writing to a file descriptor and a light-weight callback function. The `EventNotify::getFD` function returns a file descriptor that will have a byte written to it when a callback is ready. This file descriptor can be added to a `select` or `poll` to block a thread that handles ProcControlAPI events. Alternatively, the ProcControlAPI user can register a light-weight callback that is invoked when a callback is ready. This light-weight callback provides no information about the Event and may occur on another thread or from a signal handler—the ProcControlAPI user is encouraged to keep this callback minimal.

It is important for a user to respond promptly to a callback notification. A target process may remain blocked while a notification is pending. If a target process is generating many events that need callbacks, a long delay in notification could have a significant performance impact.

Once the ProcControlAPI user knows that a callback is ready to be delivered they can call `Process::handleEvents`, which will invoke all callback functions. Alternatively, if the ProcControlAPI user does not need to handle events outside of ProcControlAPI, they can continue to block in `Process::handleEvents` without going through the notification system.

2.3. iRPCs

An iRPC (Inferior Remote Procedure Call) is a mechanism for executing code in a target process. Despite the name, an iRPC does not necessarily have to involve a procedure call—any piece of code can be executed.

A ProcControlAPI user can invoke an iRPC by providing ProcControlAPI with a buffer of machine code and specifying a Process or Thread on which to run the machine code. ProcControlAPI will insert the machine code into the address space, save the register set, run the machine code, and then remove the machine code after execution completes. When the iRPC completes (but before the registers and memory are cleaned) ProcControlAPI will deliver an `EventIRPC` to any registered callback function. The ProcControlAPI user may use this callback to collect any results from the registers or memory used by the iRPC.

Note that ProcControlAPI will preserve the registers of the thread running the iRPC, and it will preserve the memory used by the machine code. Other memory or system state changed by the iRPC may remain visible to the target process after the iRPC completes.

The machine code for each iRPC must contain at least one trap instruction (e.g., a `0xCC` instruction on x86 family or a `0x7D821008` instruction on the PPC family). ProcControlAPI will stop executing the iRPC upon invocation of the trap. Note that the trap instruction must fall within the original machine code for the iRPC. If the iRPC calls or jumps to another piece of code that executes a trap instruction then ProcControlAPI will not treat it as the end of the iRPC.

Before an iRPC can be run it must be posted to a process or thread using the `Process::postIRPC` or `Thread::postIRPC` API functions. The `Process::postIRPC` function will select a thread to post the iRPC to. Multiple iRPCs can be posted to the same thread, but only one iRPC will run at a time—subsequent iRPCs will be queued and run after the preceding iRPC completes. If multiple iRPCs are posted to different threads in a multi-threaded process, then they may run in parallel.

An iRPC can be posted to a stopped or running thread. If posted to a stopped thread, then the iRPC will run when the thread is continued. If posted to a running thread, then the iRPC will run immediately or, if posted from a callback function, when the callback function completes.

An iRPC may be blocking or non-blocking. If a blocking iRPC is posted to any Process, then calls to `Process::handleEvents` will block until the iRPC is completed.

2.4. Memory Management

ProcControlAPI manages memory using a shared pointer system provided by Boost (http://www.boost.org). Many of the ProcControlAPI interface objects contain a `ptr typedef`

as part of their class (e.g, `Process::ptr`). This type refers to a shared pointer that points to the object. The `const_ptr` type (e.g., `Process::const_ptr`) refers to a shared pointer that points to a constant object.

The shared pointer system will use reference counting to decide when to clean objects. The ProcControlAPI user should not explicitly clean any ProcControlAPI objects, instead they should drop their references to the objects and let them be automatically cleaned. ProcControlAPI will maintain its own references for any object that is still "live" (i.e., a process or thread that is still running) so that these objects will not be pre-maturely cleaned.

A "`NULL`" value is specified by a shared pointer using the default constructor on the `ptr` type. E.g., `Process::ptr()` represents a `NULL` pointer to a `Process`.

See the Boost web-site for more details on shared pointers.

3. API Reference

This section gives an API reference for all classes, functions and types in ProcControlAPI. Everything defined in this section is under the namespaces `Dyninst` and `ProcControlAPI`. These types can be accessed by prepending a `Dyninst::ProcControlAPI::` in-front of them (e.g., `Dyninst::ProcControlAPI::Process`) or by adding a using namespace directive before the references (e.g., `using namespace Dyninst; using namespace ProcControlAPI;`)

3.1. Process

The `Process` class is the primary handle for operating on a single target process. `Process` objects may be created by calls to the static functions `Process::createProcess` or `Process::attachProcess`, or in response to certain types of events (e.g, fork on UNIX systems).

The static functions of the `Process` class serve as a central location for performing general ProcControlAPI operations, such as `handleEvents` and `registerEventCallback` when dealing with callbacks.

Process Declared In:
`PCProcess.h`

Process Types:
`Process::ptr`
`Process::const_ptr`

The `Process::ptr` and `Process::const_ptr` respectively represent a pointer and a const pointer to a `Process` object. Both pointer types are reference counted and will cause the underlying `Process` object to be cleaned when there are no more references. ProcControlAPI will maintain internal references to any `Process` it actively controls, relinquishing those references when the process either exits or is detached.

```
enum Process::cb_action_t {
   cbDefault,
   cbThreadContinue,
   cbThreadStop,
   cbProcContinue,
   cbProcStop
}
struct Process::cb_ret_t {
   cb_ret_t(cb_action_t p) : parent(p), child(cbDefault) {}
   cb_ret_t(cb_action_t p, cb_action_t c) : parent(p), child(c)
            {}
   cb_action_t parent;
   cb_action_t child;
}
```

The `cb_ret_t` enum is used as the return type for callback functions registered through `Process::registerEventCallback()`. A callback function can specify whether

the thread or process associated with its event should be stopped or continued by respectively returning `cbThreadContinue`, `cbThreadStop`, `cbProcContinue`, or `cbProcStop`. The `cbDefault` return value returns a `Process` and `Thread` to the original state before the event occurred.

Some events, such as process spawn or thread create involve two processes or threads. In this case the ProcControlAPI user can specify a `cb_action_t` value for both the parent and child using the two parameter constructor for `cb_ret_t`.

```
typedef Process::cb_ret_t(*cb_func_t)(Event::const_ptr)
```

The `cb_func_t` type is a function pointer type for functions that can handle event callbacks. The callback function gets an `Event::const_ptr` as input, which points to the `Event` that triggered the callback. The `cb_func_t` function should return a `cb_ret_t` describing what to do with the process after handling the event.

```
typedef enum {
   OSNone,
   Linux,
   FreeBSD,
   Windows
   VxWorks
   BlueGeneL
   BlueGeneP
   BlueGeneQ
} Dyninst::OSType
```

A value from this enum is returned from `Process::getOS` and signifies the current OS on which the target process is running.

This type is used by `Process`, but it is declared in the `Dyninst` namespace in `dyntypes.h`.

```
typedef std::pair<Dyninst::Address, Dyninst::Address> MemoryRegi
on;
```

The `MemoryRegion` type represents a region of allocated memory, and the first part of the pair is the start address, the second, the end.

Process Static Member Functions:

```
static Process::ptr createProcess(
   std::string executable,
   const std::vector<std::string> &argv,
   const std::vector<std::string> &envp = emptyEnv,
   const std::map<int,int> &fds = emptyFDs)
```

This function creates a new process by launching an executable file named by `executable` with the arguments specified by `argv`, the environment specified in `envp`, and it returns a pointer to the new `Process` object upon success. The new process will be created with its initial thread in the `stopped` state.

It is an error to call this function from a callback.

If the `fds` map is not empty, then the new process will be created with the file descriptors from the `fds' first` elements `dup2` mapped to the file descriptors in `fds' second` elements.

If `envp` is empty, the environment will be inherited from the calling process.

ProcControlAPI may deliver callbacks when this function is called.

This function returns `Process::ptr()` on error, and a subsequent call to `getLastError` returns details on the error.

```
static Process::ptr attachProcess(
     Dyninst::PID pid,
     std::string executable = "")
```

This function creates a new `Process` object by attaching to the PID specified by `pid`. The new `Process` object will be returned from this function upon success. The executable argument is optional, and can be used to assist ProcControlAPI in finding the process' executable on operating systems where this cannot be easily determined (currently on AIX). The new process will be returned with all of its threads in the stopped state.

It is an error to call this function from a callback.

ProcControlAPI may deliver callbacks when this function is called.

This function return `Process::ptr()` on error, and a subsequent call to `getLastError` returns details on the error.

```
static bool handleEvents(bool block)
```

This function causes ProcControlAPI to handle any pending debug events and deliver callbacks. When an event requires a callback ProcControlAPI needs control of the main thread in order to deliver the callback. This function gives control of the main thread to ProcControlAPI for callback delivery. A user can know when to call `handleEvents` by using the `EventNotify` interface; See Sections 2.2.3 and 0 for more details on EventNotify.

If the `block` parameter is true, then `handleEvents` will block until at least one debug event has been handled. If `block` is false then `handleEvents` returns immediately if no events are ready to be handled.

This function returns true if it handled at least one event and false otherwise.

It is an error to call this function from a callback.

```
static bool registerEventCallback(
     EventType evt,
     cb_func_t cbfunc)
```

This function registers a new callback function with ProcControlAPI. Upon receiving an event with type `evt`, ProcControlAPI will deliver a callback with that event to the `cbfunc` function. Multiple functions can be registered to receive callbacks for a single `EventType`, and a single function can be registered with multiple `EventTypes`.

If multiple callback functions are registered with a single `EventType`, then it is undefined what order those callback functions will be invoked in. In this case the `cb_ret_t` result of

the last callback function called will be used to determine what stop or continue operations should be performed on the process. If a single callback function is registered for the same `EventType` multiple times, then ProcControlAPI will only invoke one call to the callback function for each instance of the `EventType`.

This function returns true on success and false on error. Upon an error a subsequent call to `getLastError` returns details on the error.

```
static bool removeEventCallback(
    EventType evt,
    cb_func_t cbfunc)
```

This function un-registers a callback that was registered with `registerEventCallback`. After a successful call to this function the callback function `cbfunc` will stop being called for events with `EventType evt`. Other callback functions registered for `evt` will not be affected. Other instances of `cbfunc` registered for different `EventTypes` will not be affected.

This function returns true if a callback was successfully removed and false otherwise. Upon an error a subsequent call to `getLastError` returns details on the error.

`static bool removeEventCallback(EventType evt)`

This function unregisters all callback functions associated with the `EventType evt`. After a successful call to this function ProcControlAPI will stop delivering callbacks for evt until a new callback function is registered.

This function returns true if a callback was successfully removed and false otherwise. Upon an error a subsequent call to `getLastError` returns details on the error.

`static bool removeEventCallback(cb_func_t func)`

This function unregisters all instances of callback function `func` from any callback with any `EventType`.

This function returns true if a callback was successfully removed and false otherwise. Upon an error a subsequent call to `getLastError` returns details on the error.

Process Member Functions:

`Dyninst::PID getPid() const`

This function returns an OS handle referencing the process. On UNIX systems this is the pid of the process.

`Dyninst::Architecture getArchitecture() const`

This function returns an enum that describes the architecture of the target process. See Appendix A for the definition of `Dyninst::Architecture`.

`Dyninst::OSType getOS () const`

This function returns an enum that describes the OS of the target process. See the beginning of this section for the definition of `Dyninst::OSType`.

`bool supportsLWPEvents () const`

This function returns true if the target process can throw LWP create and destroy events and false otherwise.

`bool supportsUserThreadEvents () const`
This function returns true if the target process can throw user thread create and destroy events and false otherwise.

`bool supportsFork () const`
This function returns true if the fork system call is supported in the target process and false otherwise.

`bool supportsExec () const`
This function returns true if the exec system call is supported in the target process and false otherwise.

`bool isTerminated() const`
This function returns true if the target process has terminated (either via a crash or normal exit) or if the ProcControlAPI has detached from the target process. It returns false otherwise.

`bool isExited() const`
This function returns true of the target process exited via a normal exit (e.g, calling the `exit` function or returning from `main`). It returns false otherwise.

`int getExitCode() const`
If a target process exited normally then this function returns its exit code. The return result of this function is undefined if the `Process' isExited` function returns `false`.

`bool isCrashed() const`
This function returns true if the target process exited because of a crash. It returns false otherwise.

`int getCrashSignal() const`
If a target process exited because of a crash, then this function returns the signal that caused the target process to crash. The return result of this function is undefined if the `Process' isCrashed` function returns `false`.

`bool hasStoppedThread() const`
This function returns true if the target process has at least one thread in the stopped state. It returns false otherwise or if an error occurs. In the event of an error a call to `getLastError` returns details on the error.

`bool hasRunningThread() const`
This function returns true if the target process has at least one thread in the running state. It returns false otherwise or if an error occurs. In the event of an error a call to `getLastError` returns details on the error.

`bool allThreadsStopped() const`
This function returns true if all threads in the target process are in the stopped state. It returns false otherwise or if an error occurs. In the event of an error a call to `getLastError` returns details on the error.

`bool allThreadsRunning() const`
This function returns true if all threads in the target process are in the running state. It returns false otherwise or if an error occurs. In the event of an error a call to `getLastError` returns details on the error.

`bool allThreadsRunningWhenAttached() const`
This function returns true if all threads were running when the controller process attached to this process. It returns false if any threads were stopped. If the target process was created instead of attached, this function returns true.

`bool continueProc()`
This function will move all threads in the target process into the running state. This function returns `true` if at least one thread was continued as part of the call, and `false` otherwise.

It is an error to call this function from a callback.

ProcControlAPI may deliver callbacks when this function is called.

This function return `false` on error, and a subsequent call to `getLastError` returns details on the error.

`bool stopProc()`
This function will move all threads in the target process into the stopped state. This function returns `true` if at least one thread was stopped as part of the call, and `false` otherwise.

It is an error to call this function from a callback.

ProcControlAPI may deliver callbacks when this function is called.

This function return `false` on error, and a subsequent call to `getLastError` returns details on the error.

`bool detach(bool leaveStopped = false)`
This function will detach ProcControlAPI from the target process. ProcControlAPI will no longer be able to control or receive events from the target process. All breakpoints will be removed from the target. This function returns `true` on success and `false` on error. Upon an error a subsequent call to `getLastError` returns details on the error.

If the `leaveStopped` parameter is set to `true`, and the process is in a stopped state, then the target process will be left in a stopped state after the detach.

It is an error to call this function from a callback.

`bool temporaryDetach()`
This function temporarily detaches from the target process, but leaves the Process data structure intact. This functionality is commonly called detach-on-the-fly. The target process will not report new events nor be controllable or able to be queried by the user. Breakpoints are removed from the process. The `reAttach` function will reconnect the process after this call.

This function returns true on success and false upon error.

It is an error to call this function from a callback.

`bool reAttach()`

This function reconnects to the target process after a `temporaryDetach` call. Any breakpoints will be re-inserted back into the function, and if threads have been created or destroyed during the time detached new events will be thrown for them.

This function returns true on success and false upon error.

It is an error to call this function from a callback.

`bool terminate()`

This function forcefully terminated the target process. Upon a successful call to this function the target process will end execution. The `Process` object will record the target process as having crashed. This function returns `true` on success and `false` on error. Upon an error a subsequent call to `getLastError` returns details on the error.

It is an error to call this function from a callback.

```
const ThreadPool &threads() const
ThreadPool &threads()
```

These functions respectively return a const reference or a reference to the Process' `ThreadPool`. The `ThreadPool` object can be used to iterate over and query the `Process' Thread` objects—see the Section 3.6 for more details on `ThreadPool`.

```
const LibraryPool &libraries() const
LibraryPool &libraries()
```

These functions respectively return a const reference or a reference to the Process' `LibraryPool`. The `LibraryPool` object can be used to iterate over and query the `Process' Library` objects—see the Section 3.7 for more details on `LibraryPool`.

`bool addLibrary(std::string libname)`

This function causes the specified library to be loaded into the process. It will trigger an event (and thus a user callback) for each library loaded (including dependencies).

```
void *getData () const
void setData (void *p) const
```

These functions respectively get and set an opaque data object that can be associated with this process. The data is not interpreted by ProcControlAPI, but remains associated with the process.

`unsigned getMemoryPageSize() const`

This function returns memory page size for the current OS on which the target process is running.

```
Dyninst::Address mallocMemory(size_t long size)
Dyninst::Address mallocMemory(
     size_t size,
     Dyninst::Address addr)
```

These functions allocate a region of memory in the target process' address space of size `size`. Upon a successful call these functions will map an area of memory in the target process that is readable, writeable and executable. The `mallocMemory(size_t)` function will allocate memory at any available address. The `mallocMemory(size_t,`

Dyninst::Address) function will only allocate memory at the specified address, addr.

It is an error to call this function from a callback.

ProcControlAPI may deliver callbacks when this function is called.

Upon success these functions return the start address of memory that was allocated and 0 otherwise. Upon an error a subsequent call to getLastError returns details on the error.

bool freeMemory(Dyninst::Address addr)

This function will free a region of memory that was allocated by the mallocMemory function. Upon a successful call to this function, the area of memory starting at addr will be unmapped and no longer accessible to the target process. It is an error to call this function with an address that was not returned by mallocMemory.

It is an error to call this function from a callback.

ProcControlAPI may deliver callbacks when this function is called.

Upon success this function returns true, otherwise it returns false. Upon an error a subsequent call to getLastError returns details on the error.

```
bool writeMemory(
    Dyninst::Address addr,
    void *buffer,
    size_t size) const
```

This function writes to the target process's memory. The addr parameter specifies an address in the target process to which ProcControlAPI should write. The buffer and size parameters specify a region of controller process memory that will be copied into the target process.

It is an error to call this function on a Process that does not have at least one Thread in a stopped state.

This function returns true on success and false on error. Upon an error a subsequent call to getLastError returns details on the error.

```
bool readMemory(
    void *buffer,
    Dyninst::Address addr,
    size_t size) const
```

This function reads from the target process' memory. The addr and size parameters specify an address in the target process from which ProcControlAPI should read. The buffer parameter specifies an address in the controller process where ProcControlAPI should write the copied bytes.

It is an error to call this function on a Process that does not have at least one Thread in a stopped state.

This function returns true on success and false on error. Upon an error a subsequent call to getLastError returns details on the error.

```
bool getMemoryAccessRights(
     Dyninst::Address addr,
     mem_perm& rights)
bool setMemoryAccessRights(
     Dyninst::Address addr,
     size_t size,
     mem_perm rights,
     mem_perm& oldRights)
```

These functions respectively get and set memory permission at the specified address, `addr`. The `setMemoryAccessRights` function also affects a region of memory in the target process's address space of size `size`.

```
bool findAllocatedRegionAround(
     Dyninst::Address addr,
     MemoryRegion& memRegion)
```

This function finds a region of allocated memory `memRegion` contains address `addr`, and returns `true` on success, otherwise `false`.

```
bool addBreakpoint(
     Dyninst::Address addr,
     Breakpoint::ptr bp) const
```

This function inserts the `Breakpoint` specified by `bp` into the target process at address `addr`. See the Section 3.4 for more details on `Breakpoint`.

It is an error to call this function on a `Process` that does not have at least one `Thread` in a `stopped` state.

This function returns `true` on success and `false` on error. Upon an error a subsequent call to `getLastError` returns details on the error.

```
bool rmBreakpoint(
     Dyninst::Address addr,
     Breakpoint::ptr bp) const
```

This function removes the `Breakpoint` specified by `bp` at address `addr` from the target process. See the section 3.4 on `Breakpoint` for more details.

This function returns `true` on success and `false` on error. Upon an error a subsequent call to `getLastError` returns details on the error.

```
bool postIRPC(IRPC::ptr irpc) const
```

This function posts the given `irpc` to the `Process`. ProcControlAPI selects a `Thread` from the `Process` to run the iRPC and put `irpc` into that `Thread`'s queue of posted IRPCs. See Sections 2.3 and 3.5 for more information on iRPCs.

Each instance of an `IRPC` object can be posted at most once. It is an error to attempt to post a single `IRPC` object multiple times.

This function returns `true` on success and `false` on error. Upon an error a subsequent call to `getLastError` returns details on the error.

`bool runIRPCSync(IRPC::ptr irpc)`

This function posts an irpc, similar to `Process::postIRPC`; continues the thread the `irpc` was posted to; and returns when the `irpc` has completed running. The thread will be returned to its original running state when this function returns.

This function returns true if the `irpc` was successfully run, and false otherwise. Note that stopping the thread that is running the `irpc` while this function waits for `irpc` completion causes this function to return an error.

It is an error to call this function from a callback.

`bool runIRPCAsync(IRPC::ptr irpc)`

This function posts an `irpc`, similar to `Process::postIRPC`, and then continues the thread the `irpc` was posted to.

This function returns true if the `irpc` was successfully posted and run, and false otherwise.

It is an error to call this function from a callback.

`bool getPostedIRPCs(std::vector<IRPC::ptr> &rpcs) const`

This function returns all `IRPCs` posted to this `Process` in the `rpcs` vector. This list does not include any `IRPCs` currently running—see `Thread::getRunningIRPC`() for this functionality.

This function returns `true` on success and `false` on error. Upon an error a subsequent call to `getLastError` returns details on the error.

```
LibraryTracking *getLibraryTracking()
ThreadTracking *getThreadTracking()
LWPTracking *getLWPTracking()
FollowFork *getFollowFork()
SignalMask *getSignalMask()
```

These functions return pointers to configuration objects for platform-specific features associated with this Process object.

These functions return `NULL` if the specified feature is unsupported on the current platform.

3.1.1. mem_perm

The `mem_perm` nested class, which defined within `Process` class, represents general memory page permission for the given memory page in the process.

mem_perm Declared In:

`PCProcess.h`

mem_perm Types:

```
Process::mem_perm::read
Process::mem_perm::write
Process::mem_perm::execute
```

The Process::mem_perm::read, Process::mem_perm::write, and Process::mem_perm::execute, just as their names imply, respectively represent read, write, and execution permission of given memory page.

mem_perm Member Functions:

```
mem_perm() : read(false), write(false), execute(false) {}
mem_perm(const mem_perm& p) : read(p.read), write(p.write),
execute(p.execute) {}
mem_perm(bool r, bool w, bool x) : read(r), write(w), execute(x)
{}
```

These constructors provide a convenient way to create the specific memory permission for the given page.

```
bool getR() const
bool getW() const
bool getX() const
```

These functions return true if the given memory page has read, write, and execution permission, respectively, and false otherwise.

```
bool isNone() const
bool isR()    const
bool isX()    const
bool isRW()   const
bool isRX()   const
bool isRWX()  const
```

These functions return true if the permission of given memory page is NO_ACCESS, READ_ONLY, EXECUTE, READ_WRITE, READ_EXECUTE, and READ_WRITE_EXECUTE, respectively, and false otherwise.

```
Process::mem_perm& setR()
Process::mem_perm& setW()
Process::mem_perm& setX()
```

These functions enable read, write, and execution permission for the given page, respectively, and return this `mem_perm`.

```
Process::mem_perm& clrR()
Process::mem_perm& clrW()
Process::mem_perm& clrX()
```

These functions disable read, write, and execution permission for the given page, respectively, and return this `mem_perm`.

`bool operator==(const mem_perm& p) const`
This function returns true if memory permission `p` is the same as this `mem_perm` and false otherwise.

`bool operator!=(const mem_perm& p) const`
This function returns true if memory permission `p` is different from this `mem_perm` and false otherwise.

`bool operator<(const mem_perm& p) const`
This function returns true if this `mem_perm` is less than `p` according to the notation that read permission encodes to 4, write, 2, and execute, 1, and false otherwise.

`std::string getPermName()`
Return the memory permission name for this `mem_perm`.

3.2. Thread

The `Thread` class represents a single thread of execution in the target process. Any `Process` has at least one `Thread`, and multi-threaded target processes may have more. Each `Thread` has an associated integral value known as its LWP, which serves as a handle for communicating with the OS about the thread (e.g., a PID value on Linux). On some systems, depending on availability, a `Thread` may have information from the user space threading library.

Thread Declared In:
`PCProcess.h`

Thread Types:
`Thread::ptr`
`Thread::const_ptr`
The `Thread::ptr` and `Thread::const_ptr` respectively represent a pointer and a const pointer to a `Thread` object. Both pointer types are reference counted and cause the underlying `Thread` object to be cleaned when there are no more references. ProcControlAPI maintains internal references to any `Thread` it actively controls, relinquishing those references when the thread exits or is detached.

Thread Member Functions:
`Dyninst::LWP getLWP() const`
This function returns an OS handle for this thread. On Linux this returns a `pid_t` for this thread. On FreeBSD, this returns a lwpid_t.

`Process::ptr getProcess()`
`Process::const_ptr getProcess() const`
These functions return a pointer to the `Process` object that contains this thread.

`bool isStopped() const`
This function returns true if this thread is in a stopped state and false otherwise.

`bool isRunning() const`
This function returns true if this thread is in a running state and false otherwise.

`bool isLive() const`
This function returns true if this thread is alive, and it returns false if this thread has been destroyed.

`bool isDetached() const`
This function returns true if this thread has been detached via `Process::temporaryDetach` and false otherwise.

`bool isInitialThread() const`
This function returns true if this thread is the initial thread for the process and false otherwise.

`bool stopThread()`
This function moves the thread to into a stopped state. Upon a successful call to this function the `Thread` object will be paused and will not resume execution until the `Thread` is continued. It is an error to call this function from a callback. Instead of calling this function, a callback can stop a thread by returning `Process::cbThreadStop` or `Process::cbProcStop`.

ProcControlAPI may deliver callbacks when this function is called.

Upon success this function returns `true`, otherwise it returns `false`. Upon an error a subsequent call to `getLastError` returns details on the error.

`bool continueThread()`
This function moves the thread into a running state. It is an error to call this function from a callback. Instead of calling this function, a callback can stop a thread by returning `Process::cbThreadContinue` or `Process::cbProcContinue.`

ProcControlAPI may deliver callbacks when this function is called.

Upon success this function returns `true`, otherwise it returns `false`. Upon an error a subsequent call to `getLastError` returns details on the error.

```
bool getRegister(
     Dyninst::MachRegister reg,
     Dyninst::MachRegisterVal &val) const
```
This function gets the value of a single register from this thread. The register is specified by the `reg` parameter, and the value of the register is returned by the `val` parameter. See Appendix A for an explanation of the `MachRegister` class.

It is an error to call this function on a thread that is not in the stopped state.

Upon success this function returns `true`, otherwise it returns `false`. Upon an error a subsequent call to `getLastError` returns details on the error.

`bool getAllRegisters(RegisterPool pool) const`
This function reads the values of every register in the thread and returns them as part of the `RegisterPool` object `pool`. Depending on the OS, this call may be more efficient that calling `Thread::getRegister` multiple times. See Section 3.8 for a discussion of the RegisterPool class.

It is an error to call this function on a thread that is not in the stopped state.

Upon success this function returns `true`, otherwise it returns `false`. Upon an error a subsequent call to `getLastError` returns details on the error.

```
bool setRegister(
     Dyninst::MachRegister reg,
     Dyninst::MachRegisterVal val) const
```

This function writes the value of a single register in this thread. The register is specified by the `reg` parameter, and the value that should be written is specified by the `val` parameter. See Appendix A for an explanation of the `MachRegister` class.

It is an error to call this function on a thread that is not in the stopped state.

Upon success this function returns `true`, otherwise it returns `false`. Upon an error a subsequent call to `getLastError` returns details on the error.

```
bool setAllRegisters(RegisterPool &pool) const
```

This function sets the values of every register in this thread to the values specified in the `RegisterPool` object `pool`. Depending on the OS, this call may be more efficient that calling `Thread::setRegister` multiple times. See Section 3.8 for a discussion of the `RegisterPool` class.

It is an error to call this function on a thread that is not in the stopped state.

Upon success this function returns `true`, otherwise it returns `false`. Upon an error a subsequent call to `getLastError` returns details on the error.

```
bool haveUserThreadInfo() const;
```

This function returns true if information about this `Thread's` underlying user-level thread is available.

```
Dyninst::THR_ID getTID() const;
```

This function returns the unique identifier for the user-level thread. This value is only valid if `haveUserThreadInfo` returns true.

```
Dyninst::Address getStartFunction() const;
```

This function returns the address of the initial function for the user-level thread. This value is only valid if `haveUserThreadInfo` returns true.

```
Dyninst::Address getStackBase() const;
```

This function returns the address of the bottom of the user-level thread's stack. This value is only valid if `haveUserThreadInfo` returns true.

```
unsigned long getStackSize() const;
```

This function returns the size in bytes of the user-level thread's allocated stack. This value is only valid if `haveUserThreadInfo` returns true.

```
Dyninst::Address getTLS() const;
```

This function returns the address of the user-level thread's thread local storage area. This value is only valid if `haveUserThreadInfo` returns true.

```
bool readThreadLocalMemory(
     void *buffer,
     Library::const_ptr lib,
     Dyninst::Offset tls_symbol_offset,
     size_t size) const
```

This function reads from a symbol in thread local storage (TLS) memory. TLS is memory that is local to a thread and has a lifetime matching the thread. The `tls_symbol_offset` is the TLS symbol's offset in `lib`, and can be found by reading a TLS symbol's value. The `lib` parameter can point to a library or the executable. The `buffer` parameter specifies an address in the controller process where ProcControlAPI should write the copied bytes.

It is an error to call this function on a `Thread` that is not in a `stopped` state. It is also an error to call this function on a `Thread` that has not have user-level thread information, which can be tested with `haveUserThreadInfo`.

This function returns `true` on success and `false` on error. Upon an error a subsequent call to `getLastError` returns details on the error.

```
bool writeThreadLocalMemory(
     Library::const_ptr lib,
     Dyninst::Offset tls_symbol_offset,
     const void *buffer,
     size_t size) const
```

This function writes to a symbol in thread local storage (TLS) memory. TLS is memory that is local to a thread and has a lifetime matching the thread. The `tls_symbol_offset` is the TLS symbol's offset in `lib`, and can be found by reading a TLS symbol's value. The `lib` parameter can point to a library or the executable. The `buffer` parameter specifies an address in the controller process where ProcControlAPI should read the bytes to be copied.

It is an error to call this function on a `Thread` that is not in a `stopped` state. It is also an error to call this function on a `Thread` that has not have user-level thread information, which can be tested with `haveUserThreadInfo`.

This function returns `true` on success and `false` on error. Upon an error a subsequent call to `getLastError` returns details on the error.

```
bool getThreadLocalAddress(
     Library::const_ptr lib,
     Dyninst::Offset tls_symbol_offset,
     Dyninst::Address &result_addr) const
```

This function looks up the address of a symbol in thread local storage (TLS) memory. The `tls_symbol_offset` is the TLS symbol's offset in `lib`, and can be found by reading a TLS symbol's value. The `lib` parameter can point to a library or the executable. The `result_addr` parameter will be set to the target address for the TLS symbol in this Thread.

It is an error to call this function on a `Thread` that is not in a `stopped` state. It is also an error to call this function on a `Thread` that has not have user-level thread information, which can be tested with `haveUserThreadInfo`.

This function returns `true` on success and `false` on error. Upon an error a subsequent call to `getLastError` returns details on the error.

`bool postIRPC(IRPC::ptr irpc) const`
This function posts the given `irpc` to the `Thread`. The `IRPC` is put irpc into the `Thread's` queue of posted `IRPCs` and will be run when ready. See Section 2.3 for more information on posting `IRPCs`.

Each instance of an `IRPC` object can be posted at most once. It is an error to attempt to post a single `IRPC` object multiple times.

This function returns `true` on success and `false` on error. Upon an error a subsequent call to `getLastError` returns details on the error.

`bool getPostedIRPCs(std::vector<IRPC::ptr> &rpcs) const`
This function returns all `IRPCs` posted to this `Thread` in the vector `rpcs`. This does not include any running `IRPC`.

This function returns `true` on success and `false` on error. Upon an error a subsequent call to `getLastError` returns details on the error.

`IRPC::const_ptr getRunningIRPC() const`
This function returns a const pointer to any `IRPC` that is actively running on this `Thread`. If there is no `IRPC` actively running, then this function returns `IRPC::const_ptr()`.

`void setSingleStepMode(bool mode) const`
This function sets whether a `Thread` is in single-step mode. If called with a `mode` of `true`, then the `Thread` is put in single-step mode. If called with a `mode` of `false`, then the `Thread` is taken out of single-step mode.

A `Thread` in single-step mode will pause execution at each instruction and trigger an `EventSingleStep` event. After each `EventSingleStep` is handled (and presuming the `Thread` is still running and in single-step mode) the `Thread` will execute one more instruction and trigger another `EventSingleStep`.

`bool getSingleStepMode() const`
This function returns `true` if the `Thread` is in single-step mode and `false` otherwise.

`void *getData() const`
`void setData(void *p) const`
These functions respectively get and set an opaque data object that can be associated with this `Thread`. The data is not interpreted by ProcControlAPI, but remains associated with the `Thread`.

3.3. Library

A `Library` represents a single shared library (frequently referred to as a DLL or DSO, depending on the OS) that has been loaded into the target process. In addition, a `Library` will be used to represent the process' executable. `Process`' with statically linked executables will only contain the single `Library` that represents the executable.

Each Library contains a *load address* and a file name. The load address is the address at which the OS loaded the library, and the file name is the path to the library's file. Note that on some operating systems (Linux, Solaris, BlueGene, FreeBSD) the load address does not necessarily represent the beginning of the library in memory; instead it is a value that can be added to a library's symbol offsets to compute the dynamic address of a symbol.

Libraries may be loaded and unloaded by the process during execution. A library load or unload can trigger a callback with an EventLibrary parameter. The current list of libraries loaded into a process can be accessed via a `Process`' `LibraryPool` object (see Section 3.7).

Library Types

`Library::ptr`
`Library::const_ptr`

The `Library::ptr` and `Library::const_ptr` types are respective typedefs for a pointer and a const pointer to a library.

These pointers are not shared pointers—ProcControl will automatically clean a Library object when it is unloaded. It is not recommended that the user maintains copies of pointers to Library objects after an `EventLibrary` delivers notice of a library unload.

Library Member Functions

`std::string getName() const`

Returns the file name for this `Library`.

`std::string getAbsoluteName() const`

Returns a file name for this `Library` that does not contain symlinks or a relative path.

`Dyninst::Address getLoadAddress() const`

Returns the load address for this `Library`.

`Dyninst::Address getDataLoadAddress() const`

The AIX operating system can have two load addresses for a library: one for the code region and one for the data region. On AIX `Library::getLoadAddress` returns the load address of the code region and `Library::getDataLoadAddress` returns the load address of the data region. On non-AIX systems this function returns 0.

`Dyninst::Address getDynamicAddress`() const

On ELF based systems (FreeBSD, Linux, BlueGene) this function will return the address of the Library's dynamic section. On other systems this function returns 0.

`bool isSharedLib() const`

This function returns `true` of this `Library` object is refers to a shared library and `false` if it refers to an executable.

`void *getData() const`

This function returns an opaque data object that user code can associate with this library. Use `setData` to set this opaque value.

`void setData(void *p) const`

This function sets associates an opaque data object with the `Library`. ProcControlAPI does not try to interpret this value, but will return it with the `getData` function.

3.4. Breakpoint

A breakpoint is a point in the code region of a target process that, when executed, stops the process execution and notifies ProcControlAPI. Upon being continued the process will resume execution at the point. A `Breakpoint` object is a handle that can represents one or more breakpoints in one or more processes. Upon receiving notification that a breakpoint has executed, ProcControlAPI will deliver a callback with an `EventBreakpoint`, (see Section 3.15.7).

Some `Breakpoint` objects can be created as *control-transfer breakpoints*. When a process is continued after executing a control-transfer the process will resume at an alternate location, rather than at the breakpoint's installation point.

A single `Breakpoint` can be inserted into multiple locations within a target process. This can be useful when a user has wants to perform a single action at multiple locations in a target process. For example, if a user wants to insert a breakpoint at the entry to function `foo`, and `foo` has multiple instantiations in a process, then a single `Breakpoint` can be inserted at each instance of `foo`.

A single `Breakpoint` object can be inserted into multiple target processes at the same time. When a process does an operation that copies an address space, such as fork on UNIX, the child process will receive all `Breakpoint` objects that were installed in the parent process.

Multiple `Breakpoint` objects can be inserted into the same location with-in the same process. When this location is executed in the target process a single callback will be delivered, and the `EventBreakpoint` object will contain a reference to each Breakpoint inserted at the location. At most one control-transfer breakpoint can be inserted at any one point in a process.

Due to the many-to-many nature of `Breakpoints` and `Processes`, a single installation of a `Breakpoint` can be identified by a `Breakpoint`, `Process`, `Address` triple. The functions for inserting and removing breakpoints (`Process::addBreakpoint` and `Process::rmBreakpoint`) need all three pieces of information.

A `Breakpoint` can be a hardware breakpoint or a software breakpoint. A hardware breakpoint is typically implemented by setting special debug register in the process and can trigger on code execution, data reads or data write. A software breakpoint is typically implemented by writing a special instruction into a code sequence and can only be triggered by code execution. There are typically a limited number of hardware breakpoints available at the same time.

Breakpoint Types

`Breakpoint::ptr`
`Breakpoint::const_ptr`

The `Breakpoint::ptr` and `Breakpoint::const_ptr` types are respectively a pointer and a const pointer to a `Breakpoint` object. These pointers are shared pointers, and the underlying `Breakpoint` object will be automatically clean when there are no more references to it. ProcControlAPI will automatically maintain at least one reference to any Breakpoint that is installed in a target process.

Breakpoint Constant Values

static const int BP_X = 1
static const int BP_W = 2

static const int BP_R = 4

These constant values are used to set execute, write and read bits on hardware breakpoints.

Breakpoint Static Functions

`Breakpoint::ptr newBreakpoint()`

This function creates a new software `Breakpoint` object and returns it. The `Breakpoint` is not inserted into a `Process` until it is passed to `Process::addBreakpoint()`.

`Breakpoint::ptr newTransferBreakpoint(Dyninst::Address ctrl_to)`

This function creates a new control transfer software breakpoint. Upon resumption after executing this Breakpoint, control will resume at the address specified by the `ctrl_to` parameter.

```
Breakpoint::ptr newHardwareBreakpoint(unsigned int mode,
    unsigned int size)
```

This function creates a new hardware breakpoint. The `mode` parameter is a bitfield that contains an OR combination the values `BP_X`, `BP_W` and `BP_R`. These control whether the breakpoint will trigger when its target address is executed, written or read.

The `size` parameter specifies a range of memory that this breakpoint monitors. If memory is accessed between the target address and target address + `size`, then the breakpoint will trigger.

Breakpoint Member Functions

`bool isCtrlTransfer() const`

This function returns `true` if the `Breakpoint` is a control transfer breakpoint, and `false` if it is a regular `Breakpoint`.

`Dyninst::Address getToAddress() const`

If this `Breakpoint` is a control transfer breakpoint, then this function returns the address to which it transfers control. If this `Breakpoint` is not a control transfer breakpoint, then this function returns 0.

`void setData(void *data) const`

This function sets the value of an opaque handle that is associated with each `Breakpoint`. The opaque handle can be any value, and it can be retrieved with the `getData` function.

`void *getData() const`

This function returns the value of the opaque handled that is associated with this `Breakpoint`.

`void setSuppressCallbacks(bool val)`

This function can be used to suppress callbacks stemming from a specific breakpoint when called with `val` set to `true` value. All other effects from this breakpoint will still occur, but it will not generate a callback. By default callbacks occur from every breakpoint.

`bool suppressCallbacks() const`

This function returns `true` if callbacks have been suppressed for this breakpoint from `Breakpoint::setSuppressCallbacks` and false otherwise.

3.5. IRPC

`IRPC` is a class representing an Inferior Remote Procedure Call that can be run in a target process. See Section 2.3 for a high level discussion of iRPCs. Also see `Process::postIRPC` and `Thread::postIRPC` for information about posting an `IRPC`.

IRPC Declared In:
`PCProcess.h`

IRPC Types:
`IRPC::ptr`
`IRPC::const_ptr`

The `IRPC::ptr` and `IRPC::const_ptr` respectively represent a pointer and a const pointer to an `IRPC` object. Both pointer types are reference counted and will cause the underlying `IRPC` object to be cleaned when there are no more references. ProcControlAPI will maintain internal references to any `IRPC` currently posted or executing.

IRPC Static Member Functions:

```
IRPC::ptr createIRPC(
     void *binary_blob,
     unsigned int size,
     bool non_blocking = false)

IRPC::ptr createIRPC(
     void *binary_blob,
     unsigned int size,
     Dyninst::Address addr,
     bool non_blocking = false)
```

The `createIRPC` static function creates and returns a new IRPC object. The `binary_blob` and `size` parameters specify a buffer of machine code bytes that this IRPC should execute when invoked. ProcControlAPI will maintain its own copy of the `binary_blob` buffer, the ProcControlAPI user can free the buffer once this function completes.

If the `non_blocking` parameter is true then calls to `Process::handleEvents` will block until this IRPC is completed.

If the `addr` parameter is given, then ProcControlAPI will write and run the binary code at `addr`. Otherwise ProcControlAPI will select a location at which to run the IRPC.

IRPC Member Functions:

```
Dyninst::Address getAddress() const
```

The `getAddress` function returns the address at which the `IRPC` will be run. If the `IRPC` was not given an address at construction and has not yet started running, then this function may return 0.

`void *getBinaryCodeBlob() const`
: The `getBinaryCodeBlob` returns a pointer to memory that contains the binary code for this IRPC.

`unsigned int getBinaryCodeSize() const`
: The `getBinaryCodeSize` function returns the size of the binary code blob buffer.

`unsigned long getID() const`
: The `getID` function returns an integer identifier that uniquely identifies this IRPC.

`void setStartOffset(unsigned long off)`
: By default an `IRPC` will start executing its code blob at the beginning of the blob. This function can be used to tell ProcControlAPI to start execution of the code blob at some byte offset, `off`, into the blob.

  This function should be called before the IRPC is posted.

`unsigned long getStartOffset() const`
: If a start offset has been set for this `IRPC`, then `getStartOffset` returns it. Otherwise this function returns 0.

3.6. ThreadPool

A `ThreadPool` object is a collection for holding the `Threads` that make up a `Process`. Each `Process` object has one `ThreadPool` object, and each `ThreadPool` object has one or more `Thread` objects. A `ThreadPool` is typically used to iterate over or search the set of `Threads`.

Note that it is not safe to make assumptions about having consistent contents of a `ThreadPool` for a running target process. As the target process runs `Thread` objects may be inserted or removed from the `ThreadPool`. It is generally safer to stop a `Process` before operating on its `ThreadPool`. When used on a running process the `ThreadPool` iterator methods guarantee that they will not return invalid `Thread` objects (e.g, nothing that would lead to a segfault), but they do not guarantee that the `Thread` objects will refer to live threads or that they return all `Threads`.

ThreadPool Declared In:

`PCProcess.h`

ThreadPool Types:

```
class iterator {
public:
     iterator();
     ~iterator();
     Thread::ptr operator*() const;
     bool operator==(const iterator &i);
     bool operator!=(const iterator &i);
     ThreadPool::iterator operator++();
     ThreadPool::iterator operator++(int);
};

class const_iterator {
public:
     const_iterator();
     ~const_iterator();
     Thread::const_ptr operator*() const;
     bool operator==(const const_interator &i);
     bool operator!=(const const_iterator &i);
     ThreadPool::const_iterator operator++();
     ThreadPool::const_iterator operator++(int);
};
```

The `iterator` and `const_iterator` types of ThreadPool are respectively C++ iterators and const iterators over the set of threads represented by the ThreadPool. The behavior of `operator*`, `operator==`, `operator!=`, `operator++`, and `operator++(int)` match the standard behavior of C++ iterators.

ThreadPool Member Functions:

`ThreadPool::iterator begin()`
`ThreadPool::const_iterator begin() const`

These functions respectively return an `iterator` and a `const_iterator` that point to the beginning of the set of `Thread` objects.

`ThreadPool::iterator end()`
`ThreadPool::const_iterator end() const`

These functions respectively return an `iterator` and a `const_iterator` that point to the iterator element after the end of the set of `Thread` objects.

`ThreadPool::iterator find(Dyninst::LWP lwp)`
`ThreadPool::const_iterator find(Dyninst::LWP lwp) const`

The functions respectively return an `iterator` and a `const_iterator` that points to the `Thread` with a LWP of `lwp`. If `lwp` is not found in the thread list, then this function returns `end()`.

`size_t size() const`

This function returns the number of `Threads` in the `Process`.

`Process::ptr getProcess()`
`Process::const_ptr getProcess() const`

These functions respectively return a pointer or a const pointer to the `Process` that owns this `ThreadPool`.

`Thread::ptr getInitialThread()`
`Thread::const_ptr getInitialThread() const`

These functions respectively return a pointer or a const pointer to the initial `Thread` in a `Process`. The initial thread is the thread that started execution of the process (i.e., the thread that called `main`).

3.7. LibraryPool

A LibraryPool is a container representing the executable and set shared libraries (e.g., .dll and .so libraries) loaded into the target process' address space. A statically linked target process will only have a single executable, while a dynamically linked target process will have an executable and zero or more shared libraries.

The LibraryPool class contains iterators and search functions for operating on the set of libraries.

LibraryPool Declared In:

`PCProcess.h`

LibraryPool Types:

```
class iterator {
public:
     iterator();
     ~iterator();
     Library::ptr operator*() const;
     bool operator==(const iterator &i);
     bool operator!=(const iterator &i);
     LibraryPool::iterator operator++();
     LibraryPool::iterator operator++(int);
};
class const_iterator {
     const_iterator();
     ~const_iterator();
     Library::const_ptr operator*() const;
     bool operator==(const const_interator &i);
     bool operator!=(const const_iterator &i);
     LibraryPool::const_iterator operator++();
     LibraryPool::const_iterator operator++(int);
};
```

The `iterator` and `const_iterator` types of `LibraryPool` are respectively C++ iterators and const iterators over the set of libraries represented by the `LibraryPool`. The behavior of `operator*`, `operator==`, `operator!=`, `operator++`, and `operator++(int)` match the standard behavior of C++ iterators.

LibraryPool Member Functions:

`LibraryPool::iterator begin()`
`LibraryPool::const_iterator begin() const`

These functions respectively return an `iterator` and a `const_iterator` that point to the beginning of the set of `Library` objects.

`LibraryPool::iterator end()`
`LibraryPool::const_iterator end() const`

These functions respectively return an `iterator` and a `const_iterator` that point to the iterator element after the end of the set of `Library` objects.

`size_t size() const`

This function returns the number of elements in the library set.

`Library::ptr getExecutable()`
`Library::const_ptr getExecutable() const`

These functions respectively return a pointer or a const pointer to the `Library` object that represents the target process' executable.

`Library::ptr getLibraryByName(std::string name)`
`Library::const_ptr getLibraryByName(std::string name) const`

These functions respectively return a pointer or a const pointer to the `Library` object that with a file name equal to `name`. If no library is found then these functions respectively return `Library::ptr()` or `Library::const_ptr()`.

3.8. RegisterPool

The `RegisterPool` object represents a set of registers. It can be used to get or set all registers in a Thread at once. See the `Thread::getAllRegisters` and `Thread::setAllRegisters` functions. See Appendix A for more information about `MachRegister` and `MachRegisterVal`.

RegisterPool Declared In:
`PCProcess.h`

RegisterPool Types:

```
class iterator {
public:
     iterator();
     ~iterator();
     std::pair<MachRegister, MachRegisterVal> operator*() const;
     bool operator==(const iterator &i);
     bool operator!=(const iterator &i);
     RegisterPool::iterator operator++();
     RegisterPool::iterator operator++(int);
};
```

This is a C++ `iterator` over the set of registers contained in the registerPool. The behavior of `operator*`, `operator==`, `operator!=`, `operator++`, and `operator++(int)` match the standard behavior of C++ iterators.

RegisterPool Member Functions:

`LibraryPool::iterator begin()`
This function returns an iterator that points to the beginning of the set of registers.

`LibraryPool::iterator end()`
This function returns an iterator that points element after the end of the set of registers .

`LibraryPool::iterator find(Dyninst::MachRegister r)`
This function returns an iterator that points to the element in the register pool that equals register `r`. If `r` is not found, then this function returns `end()`.

`size_t size() const`
This function returns the number of elements in the register set.

`Dyninst::MachRegisterVal& operator[](Dyninst::MachRegister r)`

This function returns a reference to the value associated with the register `r` in this register pool. It can be used to efficiently get and set the values of registers in this pool, or to fill the pool with new `MachRegister` objects.

3.9. AddressSet

The `AddressSet` class is a set container of `Process` and `Dyninst::Address` tuples, with each set element a `std::pair<Address, Process::ptr>`. `AddressSet` is used by the `ProcessSet` and `ThreadSet` classes for performing group operations on large numbers of processes. An `AddressSet` might, for example, represent the location of a symbol across numerous processes, or the location of a buffer in each process where data can be written or read.

The iteration interfaces of AddressSet resemble a C++ STL `std::multimap<Address, Process::ptr>`. When iterating all `Addresses` will appear in sequential order from smallest to largest.

AddressSet Declared In:
`ProcessSet.h`

AddressSet Types:
`AddressSet::ptr`
`AddressSet::const_ptr`

The `AddressSet::ptr` and `AddressSet::const_ptr` respectively represent a pointer and a const pointer to an AddressSet object. Both pointer types are reference counted and will cause the underlying `AddressSet` object to be cleaned when there are no more references.

```
typedef std::pair<Dyninst::Address, Process::ptr> value_type
class iterator {
public:
     iterator();
     ~iterator();
     value_type operator*() const;
     bool operator==(const iterator &i);
     bool operator!=(const iterator &i);
     AddressSet::iterator operator++();
     AddressSet::iterator operator++(int);
};
class const_iterator {
public:
     const_iterator();
     ~const_iterator();
     value_type operator*() const;
     bool operator==(const const_interator &i);
     bool operator!=(const const_iterator &i);
     AddressSet::const_iterator operator++();
     AddressSet::const_iterator operator++(int);
};
```

These are C++ iterators over the Address and Process pairs contained in the AddressSet. The behavior of `operator*`, `operator==`, `operator!=`, `operator++`, and `operator++(int)` match the standard behavior of C++ iterators.

AddressSet Static Member Functions:

`static AddressSet::ptr newAddressSet()`

This function returns a new `AddressSet` that is empty.

`static AddressSet::ptr newAddressSet(ProcessSet::const_ptr ps, Dyninst::Address addr)`

This function returns a new `AddressSet` initialized with the elements from `ps` paired with the Address `addr`.

`static AddressSet::ptr newAddressSet(ProcessSet::const_ptr ps, std::string library_name, Dyninst::Offset off)`

This function returns a new `AddressSet` initialized with the elements from `ps`. The `Address` element for each process is calculated by looking up the load address of `library_name` in each `Process` and adding it to `off`.

AddressSet Member Functions

`iterator begin()`
`const_iterator begin() const`

These functions return an `iterator` that points to the first element in the `AddressSet`, or `end()` if the `AddressSet` is empty.

`iterator end()`
`const_iterator end() const`
These functions return an `iterator` that points to the element that comes after the final element in the `AddressSet`.

`iterator find(Dyninst::Address addr)`
`const_iterator find(Dyninst::Address addr) const`
These functions return an `iterator` that points to the first element in the `AddressSet` with an address of `addr`. They return `end()` if no element matches `addr`.

```
iterator find(Dyninst::Address addr, Process::const_ptr proc)
const_iterator find(Dyninst::Address addr,
    Process::const_ptr proc) const
```
These functions return an `iterator` that points to any element that has a process and address of `proc` and `addr`. It returns `end()` if no element matches.

`size_t count(Dyninst::Address addr) const`
This function returns the number of elements with address `addr`.

`size_t size() const`
This function returns the number of elements in the `AddressSet`.

`bool empty() const`
This function returns true if the `AddressSet` has zero elements and false otherwise.

```
std::pair<iterator, bool> insert(Dyninst::Address addr,
    Process::const_ptr proc)
```
This function inserts a new element into the `AddressSet` with `addr` and `proc` as its values. If another element with those values already exists, then no new element will be inserted. It returns an `iterator` that points to the new or existing element and a boolean value that is true if a new element was inserted and false otherwise.

`size_t insert(Dyninst::Address addr, ProcessSet::const_ptr ps)`
For every element in `ps`, this function inserts it and `addr` into the AddressSet. It returns the number of new elements created.

`void erase(iterator pos)`
This function removes the element pointed to by `pos` from the `AddressSet`.

`size_t erase(Process::const_ptr proc)`
This function removes every element with a process of `proc` from the `AddressSet`. It returns the number of elements removed.

`size_t erase(Dyninst::Address addr, Process::const_ptr proc)`
This function removes any element that has and address and process of `addr` and `proc` from the `AddressSet`. It returns the number of elements removed.

`void clear()`
This function erases all elements from the `AddressSet` leaving an `AddressSet` of size zero.

`iterator lower_bound(Dyninst::Address addr)`
This function returns an `iterator` pointing to the first element in the `AddressSet` that has an address greater than or equal to `addr`.

`iterator upper_bound(Dyninst::Address addr)`
This function returns an `iterator` pointing to the first element in the `AddressSet` that has an address greater than `addr`.

`std::pair<iterator, iterator> equal_range(Address addr) const`
This function returns a pair of `iterators`. The first `iterator` has the same value as the return of `lower_bound(addr)` and the second iterator has the same value as the return of `upper_bound(addr)`.

`AddressSet::ptr set_union(AddressSet::const_ptr aset)`
This function returns a new `AddressSet` whose elements are the set union of this `AddressSet` and `aset`.

`AddressSet::ptr set_intersection(AddressSet::const_ptr aset)`
This function returns a new `AddressSet` whose elements are the set intersection of this `AddressSet` and `aset`.

`AddressSet::ptr set_difference(AddressSet::const_ptr aset)`
This function returns a new `AddressSet` whose elements are the set difference of this `AddressSet` minus `aset`.

3.10. ProcessSet

The `ProcessSet` class is a set container for multiple `Process` objects. It shares many of the same operations as the `Process` class, but when an operation is performed on a ProcessSet it is done on every Process in the `ProcessSet`. On some systems, such as Blue Gene/Q, a `ProcessSet` can achieve better performance when repeating an operation across many target processes.

ProcessSet Declared In:
`ProcessSet.h`

ProcessSet Types
`ProcessSet::ptr`
`ProcessSet::const_ptr`
The `ptr` and `const_ptr` types are smart pointers to a `ProcessSet` object. When the last smart pointer to the `ProcessSet` is cleaned, then the underlying `ProcessSet` is cleaned.

`ProcessSet::weak_ptr`
`ProcessSet::const_weak_ptr`
The `weak_ptr` and `const_weak_ptr` are weak smart pointers to a `ProcessSet` object. Unlike regular smart pointers, weak pointers are not counted as references when determining whether to clean the `ProcessSet` object.

```
struct CreateInfo {
   std::string executable;
   std::vector<std::string> argv;
   std::vector<std::string> envp;
   std::map<int, int> fds;
   ProcControlAPI::err_t error_ret;
   Process::ptr proc;
}
struct AttachInfo {
   Dyninst::PID pid;
   std::string executable;
   ProcControlAPI::err_t error_ret;
   Process::ptr proc;
}
```

The `CreateInfo` and `AttachInfo` types are used by the `ProcessSet::createProcessSet` and `ProcessSet::attachProcessSet` functions when creating groups of processes.

```
class iterator {
public:
   iterator()
   ~iterator()
   Process::ptr operator*() const
   bool operator==(const iterator &i) const
   bool operator!=(const iterator &i) const
   ProcessSet::iterator operator++();
   ProcessSet::iterator operator++(int);
}
class const_iterator {
public:
   const_iterator()
   ~const_iterator()
   Process::const_ptr operator*() const
   bool operator==(const const_iterator &i) const
   bool operator!=(const const_iterator &i) const
   ProcessSet::const_iterator operator++();
   ProcessSet::const_iterator operator++(int);
}
```

These are C++ iterators over the `Process` pointers contained in the `ProcessSet`. The behavior of `operator*`, `operator==`, `operator!=`, `operator++`, and `operator++(int)` match the standard behavior of C++ iterators.

```
struct write_t {
   void *buffer
   Dyninst::Address addr
   size_t size
   err_t err
   bool operator<(const write_t &w)
}
struct read_t {
   Dyninst::Address addr
   void *buffer
   size_t size
   err_t err
   bool operator<(const read_t &r)
}
```

The `write_t` and `read_t` types are used by `ProcessSet::readMemory` and `ProcessSet::writeMemory`.

ProcessSet Static Member Functions

`static ProcessSet::ptr newProcessSet()`

This function creates a new `ProcessSet` that is empty.

`static ProcessSet::ptr newProcessSet(Process::const_ptr proc)`

This function creates a new `ProcessSet` containing `proc`.

`static ProcessSet::ptr newProcessSet(ProcessSet::const_ptr pset)`

This function creates a new `ProcessSet` that is a copy of `pset`.

```
static ProcessSet::ptr newProcessSet(
     const std::set<Process::const_ptr> &procs)
```

This function creates a new `ProcessSet` containing every element from `procs`.

```
static ProcessSet newProcessSet(AddressSet::const_iterator ab,
     AddressSet::const_iterator ae)
```

This function creates a new `ProcessSet` containing the processes that are found within [`ab`, `ae`) of an `AddressSet`.

```
static ProcessSet::ptr createProcessSet(
     std::vector<CreateInfo> &cinfo)
```

This function creates a new `ProcessSet` by launching new processes. Each element in `cinfo` specifies an executable, arguments, environment and file descriptor mappings (with similar semantics to `Process::createProcess`), which are used to launch a new process.

Every successfully created `Process` will be added to a new `ProcessSet` that is returned by this function.

In addition, the `cinfo` vector will be updated so that each entry's `proc` field points to the `Process` created by that entry, and the `error_ret` entry will contain an error code for any process launch that failed.

`static ProcessSet::ptr attachProcessSet(`
`std::vector<AttachInfo> &ainfo)`

This function creates a new `ProcessSet` by attaching to existing processes. Each element in `ainfo` specifies a PID and executable (with similar semantics to `Process::attachProcess`), which are used to attach to the processes.

Every successfully attached `Process` will be added to a new `ProcessSet` that is returned by this function.

In addition, the `ainfo` vector will be updated so that each entry's `proc` field points to the `Process` attached by that entry, and the `error_ret` entry will contain an error code any process attach that failed.

ProcessSet Member Functions

`ProcessSet::ptr set_union(ProcessSet::ptr pset) const`

This function returns a new `ProcessSet` whose elements are a set union of this `ProcessSet` and `pset`.

`ProcessSet::ptr set_intersection(ProcessSet::ptr pset) const`

This function returns a new `ProcessSet` whose elements are a set intersection of this `ProcessSet` and `pset`.

`ProcessSet::ptr set_difference(ProcessSet::ptr pset) const`

This function returns a new `ProcessSet` whose elements are a set difference of this `ProcessSet` minus `pset`.

`iterator begin()`
`const_iterator begin() const`

These functions return iterators to the first element in the `ProcessSet`.

`iterator end()`
`const_iterator end() const`

These functions return iterators that come after the last element in the `ProcessSet`.

`iterator find(Process::const_ptr proc)`
`const_iterator find(Process::const_ptr proc) const`

These functions search a `ProcessSet` for the `Process` pointed to by `proc` and returns an iterator that points to that element. It returns `ProcessSet::end()` if no element is found.

`iterator find(Dyninst::PID pid)`
`const_iterator find(Dyninst::PID pid) const`

These functions search a `ProcessSet` for the Process `pointed` to by `proc` and returns an iterator that points to that element. It returns `ProcessSet::end()` if no element is found.

`bool empty() const`

This function returns true if the `ProcessSet` has zero elements, false otherwise.

`size_t size() const`

This function returns the number of elements in the `ProcessSet`.

`std::pair<iterator, bool> insert(Process::const_ptr proc)`
This function inserts `proc` into the `ProcessSet`. If `proc` already exists in the `ProcessSet`, then no change will occur. This function returns an iterator pointing to either the new or existing element and a boolean that is true if an insert happened and false otherwise.

`void erase(iterator pos)`
This function removes the element pointed to by `pos` from the `ProcessSet`.

`size_t erase(Process::const_ptr proc)`
This function searches the `ProcessSet` for `proc`, then erases that element from the `ProcessSet`. It returns 1 if it erased an element and 0 otherwise.

`void clear()`
This function erases all elements in the `ProcessSet`.

`ProcessSet::ptr getErrorSubset() const`
This function returns a new `ProcessSet` containing every `Process` from this `ProcessSet` that has a non-zero error code. Error codes are reset upon every `ProcessSet` API call, so this function shows which `Processes` had an error on the last `ProcessSet` operation.

```
void getErrorSubsets(std::map<ProcControlAPI::err_t,
    ProcessSet::ptr> &err_sets) const
```
This function returns a set of new `ProcessSets` containing every `Process` from this `ProcessSet` that has non-zero error codes, and grouped by error code. For each error code generated by the last `ProcessSet` API operation an element will be added to `err_sets`, and every `Process` that has the same error code will be added to the new `ProcessSet` associated with that error code.

```
bool anyTerminated() const;
bool allTerminated() const;
```
These functions respectively return true if any or all processes in this `ProcessSet` are terminated, and false otherwise.

```
bool anyExited() const;
bool allExited() const;
```
These functions respectively return true if any or all processes in this `ProcessSet` have exited normally, and false otherwise.

```
bool anyCrashed() const
bool allCrashed() const
```
These functions respectively return true if any or all processes in this `ProcessSet` have crashed normally, and false otherwise.

```
bool anyDetached();
bool allDetached();
```
These functions respectively return true if any or all processes in this `ProcessSet` have been detached, and false otherwise.

```
bool anyThreadStopped();
bool allThreadStopped();
```
These functions respectively return true if any or all threads in this `ProcessSet` are stopped, and false otherwise.

```
bool anyThreadRunning();
bool allThreadRunning();
```
These functions respectively return true if any or all threads in this `ProcessSet` are running, and false otherwise.

`ProcessSet::ptr getTerminatedSubset() const`
This function returns a new `ProcessSet`, which is a subset of this `ProcessSet`, and contains every `Process` that is terminated.

`ProcessSet::ptr getExitedSubset() const`
This function returns a new `ProcessSet`, which is a subset of this `ProcessSet`, and contains every `Process` that has exited normally.

`ProcessSet::ptr getCrashedSubset() const`
This function returns a new `ProcessSet`, which is a subset of this `ProcessSet`, and contains every `Process` that has crashed.

`ProcessSet::ptr getDetachedSubset() const`
This function returns a new `ProcessSet`, which is a subset of this `ProcessSet`, and contains every `Process` that is detached.

```
ProcessSet::ptr getAllThreadRunningSubset() const
ProcessSet::ptr getAnyThreadRunningSubset() const
```
This function returns a new `ProcessSet`, which is a subset of this `ProcessSet`, and contains every `Process` that respectively has any or all threads running.

```
ProcessSet::ptr getAllThreadStoppedSubset() const
ProcessSet::ptr getAnyThreadStoppedSubset() const
```
This function returns a new `ProcessSet`, which is a subset of this `ProcessSet`, and contains every `Process` that respectively has any or all threads stopped.

`bool continueProcs()`
This function continues every thread in every process of this `ProcessSet`, similar to `Process::continueProc`. It returns true if every process was successfully continued and false otherwise.

`bool stopProcs()`
This function stops every thread in every process of this `ProcessSet`, similar to `Process::stopProc`. It returns true if every process was successfully stopped and false otherwise.

`bool detach(bool leaveStopped = true)`
This function detaches from every process in this `ProcessSet`, similar to `Process::detach`. It returns true if every process detach was successful and false otherwise.

If the `leaveStopped` parameter is set to `true`, and the processes in this `ProcessSet` are stopped, then the processes will be left in a stopped state after the detach.

`bool terminate()`
: This function terminates every process in this `ProcessSet`, similar to `Process::terminate`. It returns true if every process was successfully terminated and false otherwise.

`bool temporaryDetach()`
: This function does a temporary detach from every process in this `ProcessSet`, similar to `Process::temporaryDetach`. It returns true if every process was successfully detached and false otherwise.

`bool reAttach()`
: This function reattaches to every process in this `ProcessSet`, similar to `Process::reAttach`. It returns true if every process was successfully reAttached and false otherwise.

`AddressSet::ptr mallocMemory(size_t sz) const`
: This function allocates a block of memory of size `sz` in each process in this `ProcessSet`. The addresses of the allocations are returned in a new `AddressSet` object.

`bool mallocMemory(size_t size, AddressSet::ptr location)`
: This function allocates a block of memory of size `sz` in each process in this `ProcessSet`. The memory will be allocated in each process based on the `Process/Address` pairs in `location`.

  This function's behavior is undefined if `location` contains processes not included in this `ProcessSet`.

  This function returns true if every allocation happened without error and false otherwise.

`bool freeMemory(AddressSet::ptr addrs) const`
: This function frees memory allocated by Process::mallocMemory or `ProcessSet::mallocMemory`. The `AddressSet addrs` should contain a list of `Process/Address` pairs that point to the memory that should be freed.

  This function's behavior is undefined if `addrs` contains processes not included in this `ProcessSet`.

  This function returns true if every free happened without error and false otherwise.

```
bool readMemory(AddressSet::ptr addrs,
     std::multimap<Process::ptr, void *> &result,
     size_t size) const
```
: This function reads memory from processes in this `ProcessSet`. `addrs` should contain the addresses to read memory from. `size` should be the amount of memory read from each process. The results of the memory reads will be returned by filling in the `result multimap`. Each process that is read from will have an entry in `result` along with a `malloc` allocated buffer containing the results of the read.

It is the ProcControlAPI user's responsibility to `free` the memory buffers returned by this function.

This function's behavior is undefined if `addrs` contains processes not included in this `ProcessSet`.

This function returns true if every read happened without error, and false otherwise.

```
bool readMemory(AddressSet::ptr addrs,
     std::map<void *, ProcessSet::ptr> &result,
     size_t size)
```

This function reads memory from processes in this `ProcessSet`. `addrs` should contain the addresses to read memory from. `size` should be the amount of memory to read from each process. The results of the memory reads will be aggregated together into the `result map`. If any two processes read equivalent byte-for-byte data, then those processes are grouped together in a new `ProcessSet` associated with a common `malloc` allocated buffer containing their memory contents.

It is the ProcControlAPI user's responsibility to `free` the memory buffers returned by this function.

This function's behavior is undefined if `addrs` contains processes not included in this `ProcessSet`.

This function returns true if every read happened without error, and false otherwise.

```
bool readMemory(std::multimap<Process::const_ptr, read_t> &addr)
```

This function reads memory from processes in this `ProcessSet`. The processes to read from are specified in the indexes of `addr`. The remote address, read size and local buffer are specified in the `read_t` elements of `addr`.

This function's behavior is undefined if `addr` contains processes not included in this `ProcessSet`.

This function returns true if every read happened without error, and false otherwise. If any read results in an error, then the `error_ret` field of the associated `addr` element will be set.

```
bool writMemory(AddressSet::ptr addrs,
     const void *buffer,
     size_t sz) const
```

This function will write the contents of `buffer` of size `sz` into the memory of each process at `addrs`.

This function's behavior is undefined if `addrs` contains processes not included in this `ProcessSet`.

This function returns true if every write happened without error, and false otherwise.

`bool writeMemory(`
`std::multimap<Process::const_ptr, write_t> &addrs) const`

This function writes to the memory of each process in `addrs`. The processes to write to are specified as the indexes of `addrs`. The local memory buffer, buffer size, and target location are specified in the `write_t` element of `addrs`.

This function's behavior is undefined if `addrs` contains processes not included in this `ProcessSet`.

This function returns true if every write happened without error, and false otherwise. If any write results in an error, then the `error_ret` field of the associated `addr` element will be set.

`bool addBreakpoint(AddressSet::ptr as, Breakpoint::ptr bp) const`

This function inserts the `Breakpoint bp` into every process and address specified by as. It is similar to `Process::addBreakpoint`.

This function's behavior is undefined if `addrs` contains processes not included in this `ProcessSet`.

This function returns true if every breakpoint add happened without error, and false otherwise.

`bool rmBreakpoint(AddressSet::ptr as, Breakpoint::ptr bp) const`

The function removes the `Breakpoint bp` from each process at the locations specified in `as`. It is similar to `Process::rmBreakpoint`.

This function's behavior is undefined if `as` contains processes not included in this `ProcessSet`.

This function returns true if every breakpoint remove happened without error, and false otherwise.

`bool postIRPC(const std::multimap<Process::const_ptr, IRPC::ptr>`
`&rpcs) const`

This function posts the `IRPC` objects specified in `rpcs` to their associated processes in the `multimap`. It is similar to `Process::postIRPC`.

This function's behavior is undefined if `rpcs` contains processes not included in this `ProcessSet`.

This function returns true if every post happened without error, and false otherwise.

`bool postIRPC(IRPC::ptr irpc,`
`std::multimap<Process::ptr, IRPC::ptr> *result = NULL)`

This function makes a copy of `irpc` for each `Process` in this `ProcessSet` and posts it to that `Process`. If `result` is non-NULL, then the `multimap` will be filled with each newly created `IRPC` and the `Process` to which it was posted. It is similar to `Process::postIRPC`.

This function returns true if every post happened without error, and false otherwise.

```
bool postIRPC(IRPC::ptr irpc
    AddressSet::ptr addrs,
    std::multimap<Process::ptr, IRPC::ptr> *result = NULL)
```

This function makes a copy of `irpc` and posts it to each Process in `addrs` at the given `Address`. If `result` is non-NULL, then the `multimap` will be filled with each newly created `IRPC` and the `Process` to which it was posted. It is similar to `Process::postIRPC`.

This function's behavior is undefined if `rpcs` contains processes not included in this `ProcessSet`.

This function returns true if every post happened without error, and false otherwise.

3.11. ThreadSet

The `ThreadSet` class is a set container for `Thread` pointers. It has similar operations as `Thread`, and operations done on a `ThreadSet` affect every `Thread` in that `ThreadSet`. One some system, such as Blue Gene Q, using a `ThreadSet` is more efficient when doing the same operation across a large number of `Threads`.

ThreadSet Declared In:

ProcessSet.h

ThreadSet Types:

`ThreadSet::ptr`
`ThreadSet::const_ptr`

The `ptr` and `const_ptr` types are smart pointers to a `ThreadSet` object. When the last smart pointer to the `ThreadSet` is cleaned, then the underlying `ThreadSet` is cleaned. The `const_ptr` type is a const smart pointer.

`ThreadSet::weak_ptr`
`ThreadSet::const_weak_ptr`

The `weak_ptr` and `const_weak_ptr` are weak smart pointers to a `ThreadSet` object. Unlike regular smart pointers, weak pointers are not counted as references when determining whether to clean the `ThreadSet` object. The `const_weak_ptr` type is a const weak smart pointer.

```
class iterator {
public:
   iterator()
   ~iterator()
   Thread::ptr operator*() const
   bool operator==(const iterator &i) const
   bool operator!=(const iterator &i) const
   ThreadSet::iterator operator++();
   ThreadSet::iterator operator++(int);
}

class const_iterator {
public:
   const_iterator()
   ~const_iterator()
   Thread::const_ptr operator*() const
   bool operator==(const const_iterator &i) const
   bool operator!=(const const_iterator &i) const
   ThreadSet::const_iterator operator++();
   ThreadSet::const_iterator operator++(int);
}
```

These are C++ `iterators` over the Thread pointers contained in the ThreadSet. The behavior of `operator*`, `operator==`, `operator!=`, `operator++`, and `operator++(int)` match the standard behavior of C++ iterators.

ThreadSet Static Member Functions

`static ThreadSet::ptr newThreadSet()`

This function creates a new `ThreadSet` that is empty.

`static ThreadSet::ptr newThreadSet(Thread::ptr thr)`

This function creates a new `ThreadSet` that contains `thr`.

`static ThreadSet::ptr newThreadSet(const ThreadPool &threadp)`

This function creates a new `ThreadSet` that contains all of the `Threads` currently in `threadp`.

```
static ThreadSet::ptr newThreadSet (
     const std::set<Thread::const_ptr> &thrds)
```

This function creates a new `ThreadSet` that contains all of the threads in `thrds`.

`static ThreadSet::ptr newThreadSet(ProcessSet::ptr pset)`

This function creates a new `ThreadSet` that contains every live thread currently in every process in `pset`.

ThreadSet Member Functions

`ThreadSet::ptr set_union(ThreadSet::ptr tset) const`

This function returns a new `ThreadSet` whose elements are a set union of this `ThreadSet` and `tset`.

`ThreadSet::ptr set_intersection(ThreadSet::ptr tset) const`
This function returns a new `ThreadSet` whose elements are a set intersection of this `ThreadSet` and `tset`.

`ThreadSet::ptr set_difference(ThreadSet::ptr tset) const`
This function returns a new `ThreadSet` whose elements are a set difference of this `ThreadSet` minus `tset`.

`iterator begin()`
`const_iterator begin() const`
These functions return iterators to the first element in the `ThreadSet`.

`iterator end()`
`const_iterator end() const`
These functions return iterators that come after the last element in the `ThreadSet`.

`iterator find(Thread::const_ptr thr)`
`const_iterator find(Thread::const_ptr thr) const`
These functions search a `ThreadSet` for `thr` and returns an iterator pointing to that element. It returns `ThreadSet::end()` if no element is found

`bool empty() const`
This function returns true if the `ThreadSet` has zero elements and false otherwise.

`size_t size() const`
This function returns the number of elements in the `ThreadSet`.

`std::pair<iterator, bool> insert(Thread::const_ptr thr)`
This function inserts `thr` into the `ThreadSet`. If `thr` already exists in the `ThreadSet`, then no change will occur. This function returns an iterator pointing to either the new or existing element and a boolean that is true if an insert happened and false otherwise.

`void erase(iterator pos)`
This function removes the element pointed to by `pos` from the `ThreadSet`.

`size_t erase(Thread::const_ptr thr)`
This function searches the `ThreadSet` for `thr`, then erases that element from the `ThreadSet`. It returns 1 if it erased an element and 0 otherwise.

`void clear()`
This function erases all elements in the `ThreadSet`.

`ThreadSet::ptr getErrorSubset() const`
This function returns a new `ThreadSet` containing every `Thread` from this `ThreadSet` that has a non-zero error code. Error codes are reset upon every `ThreadSet` API call, so this function shows which `Threads` had an error on the last `ThreadSet` operation.

```
void getErrorSubsets(
     std::map<ProcControlAPI::err_t, ThreadSet::ptr> &err) const
```
This function returns a set of new `ThreadSets` containing every `Thread` from this `ThreadSet` that has non-zero error codes, and grouped by error code. For each error code generated by the last `ThreadSet` API operation an element will be added to `err`, and

every Thread that has that error code will be added to the new ThreadSet associated with that error code.

```
bool allStopped() const
bool anyStopped() const
```
These functions respectively return true if any or all threads in this ThreadSet are stopped and false otherwise.

```
bool allRunning() const
bool anyRunning() const
```
These functions respectively return true if any or all threads in this ThreadSet are running and false otherwise.

```
bool allTerminated() const
bool anyTerminated() const
```
These functions respectively return true if any or all threads in this ThreadSet are terminated and false otherwise.

```
bool allSingleStepMode() const
bool anySingleStepMode() const
```
These functions respectively return true if any or all threads in this ThreadSet are running in single step mode, and false otherwise.

```
bool allHaveUserThreadInfo() const
bool anyHaveUserThreadInfo() const
```
These functions respectively return true if any or all threads in this ThreadSet have user thread information available and false otherwise.

```
ThreadSet::ptr getStoppedSubset() const
```
This function returns a new ThreadSet, which is a subset of this ThreadSet, and contains every Thread that is stopped.

```
ThreadSet::ptr getRunningSubset() const
```
This function returns a new ThreadSet, which is a subset of this ThreadSet, and contains every Thread that is running.

```
ThreadSet::ptr getTerminatedSubset() const
```
This function returns a new ThreadSet, which is a subset of this ThreadSet, and contains every Thread that is terminated.

```
ThreadSet::ptr getSingleStepSubset() const
```
This function returns a new ThreadSet, which is a subset of this ThreadSet, and contains every Thread that is in single step mode.

```
ThreadSet::ptr getHaveUserThreadInfoSubset() const
```
This function returns a new ThreadSet, which is a subset of this ThreadSet, and contains every Thread that has user thread information available.

```
bool getStartFunctions(AddressSet::ptr result) const
```
This function fills in the AddressSet pointed to by result with the address of every start function of each Thread in this ThreadSet. This information is only available on threads that have user thread information available.

This function return true if it succeeded for every Thread, and false otherwise.

`bool getStackBases(AddressSet::ptr result) const`
This function fills in the `AddressSet` pointed to by `result` with the address of every stack base of each `Thread` in this `ThreadSet`. This information is only available on threads that have user thread information available.

This function return true if it succeeded for every Thread, and false otherwise.

`bool getTLSs(AddressSet::ptr result) const`
This function fills in the `AddressSet` pointed to by `result` with the address of every thread-local storage region of each `Thread` in this `ThreadSet`. This information is only available on threads that have user thread information available.

This function return true if it succeeded for every Thread, and false otherwise.

`bool stopThreads() const`
This function stops every `Thread` in this `ThreadSet`. It is similar to `Thread::stopThread`.

This function return true if it succeeded for every `Thread`, and false otherwise.

`bool continueThreads() const`
This function stops every `Thread` in this `ThreadSet`. It is similar to `Thread::continueThread`.

This function return true if it succeeded for every `Thread`, and false otherwise.

`bool setSingleStepMode(bool v) const`
This function puts every `Thread` in this `ThreadSet` into single step mode if `v` is true. It clears single step mode if `v` is false. It is similar to `Thread::setSingleStepMode`.

This function return true if it succeeded for every `Thread`, and false otherwise.

```
bool getRegister(Dyninst::MachRegister reg,
    std::map<Thread::ptr, Dyninst::MachRegisterVal> &res) const
```
This function gets the value of register `reg` in every `Thread` in this `ThreadSet`. The collected values are returned in the `res` map, with each `Thread` mapped to the value of `reg` in that thread. It is similar to `Thread::getRegister`.

This function return true if it succeeded for every `Thread`, and false otherwise.

```
bool getRegister(Dyninst::MachRegister reg,
    std::map<Dyninst::MachRegisterVal, ThreadSet::ptr> &res)
    const
```
This function gets the value of register `reg` in every Thread in this `ThreadSet` and then aggregates all identical values together. The `res` map will contain an entry for each unique register value, and map that value to a new `ThreadSet` that contains every `Thread` that produced that register value. It is similar to `Thread::getRegister`.

This function return true if it succeeded for every `Thread`, and false otherwise.

```
bool setRegister(Dyninst::MachRegister reg,
     const std::map<ThreadSet::const_ptr,
     Dyninst::MachRegisterVal> &vals) const
```

This function sets the value of register `reg` in each `Thread` in this `ThreadSet`. The value set in each thread is looked up in the `vals` map. It is similar to `Thread::setRegister`.

This function's behavior is undefined if it is passed a `Thread` that is not in this `ThreadSet`.

This function return true if it succeeded for every `Thread`, and false otherwise.

```
bool setRegister(Dyninst::MachRegister reg,
     Dyninst::MachRegisterVal val) const
```

This function sets the register `reg` to `val` in each `Thread` in this `ThreadSet`. It is similar to `Thread::setRegister`.

This function return true if it succeeded for every `Thread`, and false otherwise.

```
bool getAllRegisters(
     std::map<Thread::ptr, RegisterPool> &results) const
```

This function gets the values of every register in each `Thread` in this `ThreadSet`. The register values are returned as `RegisterPools` in the `results` map, with each `Thread` mapped to its `RegisterPool`. It is similar to `Thread::getAllRegisters`.

This function return true if it succeeded for every `Thread`, and false otherwise.

```
bool setAllRegisters(
     const std::map<Thread::const_ptr, RegisterPool> &val) const
```

This function sets the values of every register in each `Thread` in this `ThreadSet`. The register values are extracted from the `val` map, with each `Thread` specifying its register values via the map. This function is similar to `Thread::setAllRegisters`.

This function's behavior is undefined if it is passed a `Thread` that is not in this `ThreadSet`.

This function return true if it succeeded for every `Thread`, and false otherwise.

```
bool postIRPC(const std::multimap<Thread::const_ptr,
     IRPC::ptr> &rpcs) const
```

This function posts an `IRPC` to every `Thread` in this `ThreadSet`. The `IRPC` to post to each `Thread` is specified in the `rpcs multimap`. This function is similar to Thread::postIRPC.

This function return true if it succeeded for every `Thread`, and false otherwise.

```
bool postIRPC(IRPC::ptr irpc,
     std::multimap<Thread::ptr, IRPC::ptr> *result = NULL)
```

This function posts a copy of `irpc` to every `Thread` in this `ThreadSet`. If `result` is non-NULL, then the new `IRPC` objects are returned in the `result` multimap, with the `Thread` mapped to the `IRPC` that was posted there. This function is similar to `Thread::postIRPC`.

This function return true if it succeeded for every `Thread`, and false otherwise.

3.12. EventNotify

The EventNotify class is used to notify the user when ProcControlAPI is ready to deliver a callback function. EventNotify is a singleton class, which means only one instance of it is ever instantiated. See Section 2.2.3 for a high level description of notification.

EventNotify Declared In:
`PCProcess.h`

EventNotify Types:
`typedef void (*notify_cb_t)()`
This function signature is used for light-weight notification callback.

EventNotify Related Global Functions:
`EventNotify *evNotify()`
This function returns the singleton instance of the EventNotify class.

EventNotify Member Functions:
`int getFD()`
This function returns a file descriptor. ProcControlAPI will write a byte that will be available for reading on this file descriptor when a callback function is ready to be invoked. Upon seeing that a byte has been written to this file descriptor (likely via `select` or `poll`) the user should call the `Process::handleEvents` function. The user should never actually read the byte from this file descriptor; ProcControlAPI will handle clearing the byte after the callback function is invoked.

This function returns `-1` on error. Upon an error a subsequent call to `getLastError` returns details on the error.

`void registerCB(notify_cb_t cb)`
This function registers a light-weight callback function that will be invoked when a ProcControlAPI wishes to notify the user when a callback function is ready to be invoked. This light-weight callback may be called by a ProcControlAPI internal thread or from a signal handler; the user is encouraged to keep its implementation appropriately safe for these circumstances.

`void removeCB(notify_cb_t cb)`
This function removes a light-weight callback that was previously registered with `EventNotify::registerCB`. ProcControlAPI will no longer invoke the cb function after this function completes.

3.13. EventType

The `EventType` class represents a type of event. Each instance of an `Event` happening has one associated `EventType`, and callback functions can be registered against `EventTypes`. All `EventTypes` have an associate code—an integral value that identifies the `EventType`. Some `EventTypes` also have a time associated with them (`Pre`, `Post`, or

None)—describing when an Event may occur relative to the Code. For example, an EventType with a code of Exit and a time of Pre (written as pre-exit) would be associated with an Event that occurs just before a process exits and its address space is cleaned. An EventType with code Exit and a time of Post would be associated with an Event that occurs after the process exits and the address space is cleaned.

When using EventTypes to register for callback functions a special time value of Any can be used. This signifies that the callback function should trigger for both Pre and Post time events. ProcControlAPI will never deliver an Event that has an EventType with time code Any.

More details on Events and EventTypes can be found in Section 2.2.1.

EventType Types:

```
typedef enum {
     Pre = 0,
     Post,
     None,
     Any
} Time;

typedef int Code;
```

The Time and Code types are respectively used to describe the time and code values of an EventType.

EventType Constants:

```
static const int Error = -1
static const int Unset = 0
static const int Exit = 1
static const int Crash = 2
static const int Fork = 3
static const int Exec = 4
static const int ThreadCreate = 5
static const int ThreadDestroy = 6
static const int Stop = 7
static const int Signal = 8
static const int LibraryLoad = 9
static const int LibraryUnload = 10
static const int Bootstrap = 11
static const int Breakpoint = 12
static const int RPC = 13
static const int SingleStep = 14
static const int Library = 15
static const int MaxProcCtrlEvent = 1000
```

These constants describe possible values for an EventType's code. The Error and Unset codes are for handling error cases and should not be used for callback functions or be associated with Events.

The EventType codes were implemented as an integer (rather than an enum) to allow users to create custom EventTypes. Any custom EventType should begin at the MaxProcCtrlEvent value, all smaller values are reserved by ProcControlAPI.

EventType Related Types:

```
struct eventtype_cmp {
     bool operator()(const EventType &a, const EventType &b);
}
```

This type defines a less-than comparison function for EventTypes. While a comparison of EventTypes does not have a semantic meaning, this can be useful for inserting EventTypes into maps or other STL data structures.

EventType Member Functions:

`EventType(Code e)`

Constructs an EventType with the given code and a time of Any.

`EventType(Time t, Code e)`

Constructs an EventType with the given time and code values.

`EventType()`

Constructs an EventType with an Unset code and None time value.

`Code code() const`

Returns the code value of the EventType.

`Time time() const`

Returns the time value of the EventType.

`std::string name() const`

Returns a human readable name for this EventType.

3.14.Event

The Event class represents an instance of an event happening. Each Event has an EventType that describes the event and pointers to the Process and Thread that the event occurred on.

The Event class is an abstract class that is never instantiated. Instead, ProcControlAPI will instantiate children of the Event class, each of which add information specific to the EventType. For example, an Event representing a thread creation will have an EventType of ThreadCreate and can be cast into an EventNewThread for specific information about the new thread. The specific events that are instantiated from Event are described in the Section 3.15.

An event that occurs on a running thread may cause the process, thread, or neither to stop running until the event has been handled. The specifics of what is stopped can change between different event types and operating systems. Each Event describes whether it stopped the associated process or thread with a SyncType field. The values of this field can be async (the event stopped neither the process nor thread), sync_thread (the event stopped its thread), or

sync_process (the event stopped all threads in the process). A callback function can choose how to resume or stop a process or thread using its return value (see Section 2.2.2).

More details on Event can be found in Section 2.2.1.

Event Declared In:

Event.h

Event Types:

```
typedef enum {
     unset,
     async,
     sync_thread,
     sync_process
} SyncType
```

The SyncType type is used to describe how a process or thread is stopped by an Event. See the above explanation for more details.

Event Member Functions:

Thread::const_ptr getThread() const

This function returns a const pointer to the Thread object that represents the thread this event occurred on.

Process::const_ptr getProcess() const

This function returns a const pointer to the Process object that represents the process this event occurred on.

EventType getEventType() const

This function returns the EventType associated with this Event.

SyncType getSyncType() const

This function returns the SyncType associated with this Event.

std::string name() const

This function returns a human readable name for this Event.

```
EventTerminate::ptr getEventTerminate()
EventTerminate::const_ptr getEventTerminate() const

EventExit::ptr getEventExit()
EventExit::const_ptr getEventExit() const

EventCrash::ptr getEventCrash()
EventCrash::const_ptr getEventCrash() const

EventForceTerminate::ptr getEventForceTerminate()
EventForceTerminate::const_ptr getEventForceTerminate() const

EventExec::ptr getEventExec()
EventExec::const_ptr getEventExec() const

EventStop::ptr getEventStop()
EventStop::const_ptr getEventStop() const

EventBreakpoint::ptr getEventBreakpoint()
EventBreakpoint::const_ptr getEventBreakpoint() const

EventNewThread::ptr getEventNewThread()
EventNewThread::const_ptr getEventNewThread() const

EventNewUserThread::ptr getEventNewUserThread()
EventNewUserThread::const_tr getEventNewUserThread() const

EventNewLWP::ptr getEventNewLWP()
EventNewLWP::const_ptr getEventNewLWP() const

EventThreadDestroy::ptr getEventThreadDestroy()
EventThreadDestroy::const_ptr getEventThreadDestroy() const

EventUserThreadDestroy::ptr getEventUserThreadDestroy()
EventUserThreadDestroy::const_ptr getEventUserThreadDestroy()
const

EventLWPDestroy::ptr getEventLWPDestroy()
EventLWPDestroy::const_ptr getEventLWPDestroy() const

EventFork::ptr getEventFork()
EventFork::const_ptr getEventFor() const

EventSignal::ptr getEventSignal()
EventSignal::const_ptr getEventSignal() const
```

```
EventRPC::ptr getEventRPC()
EventRPC::const_ptr getEventRPC() const

EventSingleStep::ptr getEventSingleStep()
EventSingleStep::const_ptr getEventSingleStep() const

EventLibrary::ptr getEventLibrary()
EventLibrary::const_ptr getEventLibrary() const
```

These functions serve as a form of `dynamic_cast`. They cast the `Event` into a child type and return the result of that cast. If the `Event` object is not of the appropriate type for the given function, then they return a shared pointer `NULL` equivalent (`ptr()` or `const_ptr()`).

For example, if an `Event` was an instance of an `EventRPC`, then the `getEventRPC()` function would cast it to `EventRPC` and return the resulting value.

3.15. Event Child Classes

The `Event` class is an abstract parent class, while the classes listed in this section are the child classes that are actually instantiated. Given an `Event` object passed to a callback function, a ProcControlAPI user can inspect the `Event`'s `EventType` and cast it to the appropriate child class listed below.

Note that each child class inherits the member functions described in the Event class in Section 3.14.

Common Types:

```
<EventChildClassHere>::ptr
<EventChildClassHere>::const_ptr
```

These types are common to all `Event` children classes. Rather than repeat them for each class, they are listed once here for brevity.

The `ptr` and `const_ptr` respectively represent a pointer and a const pointer to an `Event` child class. Both pointer types are reference counted and will cause the underlying object will be cleaned when there are no more references.

3.15.1. EventTerminate

The `EventTerminate` class is a parent class for `EventExit` and `EventCrash`. It is never instantiated by ProcControlAPI and simply serves as a place-holder type for a user to deal with process termination without dealing with the specifics of whether a process exited properly or crashed.

Associated EventType Codes:
`Exit,Crash and ForceTerminate`

3.15.2. EventExit

An `EventExit` triggers when a process performs a normal exit (e.g., calling the `exit` function or returning from `main`). The process that exited is referenced with `Event`'s `getProcess` function.

An `EventExit` may be associated with an `EventType` of pre-exit or post-exit. Pre-exit means the process has not yet cleaned up its address space, and thus memory can still be read or written. Post-exit means the process has cleaned up its address space, memory is no longer accessible.

Associated EventType Code:
`Exit`

EventExit Member Functions:
`int getExitCode() const`
This function returns the process' exit code.

3.15.3. EventCrash

An `EventCrash` triggers when a process performs an abnormal exit (e.g., crashing on a memory violation). The process that crashed is referenced with `Event`'s `getProcess` function.

An `EventCrash` may be associated with an `EventType` of pre-crash or post-crash. Pre-crash means the process has not yet cleaned up its address space, and thus memory can still be read or written. Post-crash means the process has cleaned up its address space, memory is no longer accessible.

Associated EventType Code:
`Crash`

EventCrash Member Functions:
`int getTermSignal() const`
This function returns the signal that caused the process to crash.

3.15.4. EventForceTerminate

An `EventForceTerminate` triggers when a process is forcefully terminated via the `Process::terminate` function. When the callback is delivered for this event, the address space of the corresponding process will no longer be available.

Associated EventType Code:
`ForceTerminate`

EventForceTerminate Member Functions:
`int getTermSignal() const`
This function returns the signal that was used to terminate the process.

3.15.5. EventExec

An `EventExec` triggers when a process performs a UNIX-style `exec` operation. An `EventType` of post-Exec means the process has completed the `exec` and setup its new address space. An EventType of pre-Exec means the process has not yet torn down its old address space.

Associated EventType Code:
`Exec`

EventExec Member Functions:
`std::string getExecPath() const`
This function returns the file path to the process' new executable.

3.15.6. EventStop

An `EventStop` is triggered when a process is stopped by a non-ProcControlAPI source. On UNIX based systems, this is triggered by receipt of a SIGSTOP signal.

Unlike most other events, an `EventStop` will explicitly move the associated thread or process (see the `Event`'s `SyncType` to tell which) to a stopped state. Returning `cbDefault` from a callback function that has received `EventStop` will leave the target process in a stopped state rather than restore it to the pre-event state.

Associated EventType Code:
`Stop`

3.15.7. EventBreakpoint

An `EventBreakpoint` triggers when the target process encounters a breakpoint inserted by the ProcControlAPI (see Section 3.4).

Similar to EventStop, `EventBreakpoint` will explicitly move the thread or process to a stopped state. Returning `cbDefault` from a callback function that has received `EventBreakpoint` will leave the target process in a stopped state rather than restore it to the pre-event state.

Associated EventType Code:
`Breakpoint`

EventBreakpoint Member Functions:
`Dyninst::Address getAddress() const`
This function returns the address at which the breakpoint was hit.

`void getBreakpoints(std::vector<Breakpoint::const_ptr> &b) const`
This function returns a vector of pointers to the `Breakpoints` that were hit. Since it is possible to insert multiple `Breakpoints` at the same location, it is possible for this function to return more than one `Breakpoint`.

3.15.8. EventNewThread

An `EventNewThread` triggers when a process spawns a new thread. The `Event` class' `getThread` function returns the original `Thread` that performed the spawn operation, while `EventNewThread`'s `getNewThread` returns the newly created `Thread`.

This event is never instantiated by ProcControlAPI and simply serves as a place-holder type for a user to deal with thread creation without having to deal with the specifics of LWP and user thread creation.

A callback function that receives an `EventNewThread` can use the two field form of `Process::cb_ret_t` (see Sections 2.2.2 and 3.1) to control the parent and child thread.

Associated EventType Codes:
`ThreadCreate, UserThreadCreate, LWPCreate`

EventNewThread Member Functions:
`Thread::const_ptr getNewThread() const`

This function returns a const pointer to the `Thread` object that represents the newly spawned thread.

3.15.9. EventNewUserThread

An `EventNewUserThread` triggers when a process spawns a new user-level thread. The `Event` class' `getThread` function returns the original `Thread` that performed the spawn operation. This thread may have already been created if the platform supports the `EventNewLWP` event. If not, `the getNewThread function` returns the newly created `Thread`.

A callback function that receives an `EventNewThread` can use the two field form of `Process::cb_ret_t` (see Sections 2.2.2 and 3.1) to control the parent and child thread.

Associated EventType Code:
`UserThreadCreate`

EventNewThread Member Functions:
`Thread::const_ptr getNewThread() const`

This function returns a const pointer to the `Thread` object that represents the newly spawned thread or the corresponding thread, if the thread has already been created.

3.15.10. EventNewLWP

An `EventNewLWP` triggers when a process spawns a new LWP. The `Event` class' `getThread` function returns the original `Thread` that performed the spawn operation, while `EventNewThread`'s `getNewThread` returns the newly created `Thread`.

A callback function that receives an `EventNewThread` can use the two field form of `Process::cb_ret_t` (see Sections 2.2.2 and 3.1) to control the parent and child thread.

Associated EventType Code:
`LWPCreate`

EventNewThread Member Functions:
`Thread::const_ptr getNewThread() const`
This function returns a const pointer to the `Thread` object that represents the newly spawned thread.

3.15.11. EventThreadDestroy

An `EventThreadDestroy` triggers when a thread exits. Event's `getThread` member function returns the thread that exited.

This event is never instantiated by ProcControlAPI and simply serves as a place-holder type for a user to deal with thread destruction without having to deal with the specifics of LWP and user thread destruction.

Associated EventType Codes:
`ThreadDestroy, UserThreadDestroy, LWPDestroy`

3.15.12. EventUserThreadDestroy

An `EventUserThreadDestroy` triggers when a thread exits. Event's `getThread` member function returns the thread that exited.

If the platform also supports `EventLWPDestroy` events, this event will precede an `EventLWPDestroy` event.

Associated EventType Code:
`UserThreadDestroy`

3.15.13. EventLWPDestroy

An `LWPThreadDestroy` triggers when a thread exits. Event's `getThread` member function returns the thread that exited.

Associated EventType Code:
`LWPDestroy`

3.15.14. EventFork

An `EventFork` triggers when a process performs a UNIX-style fork operation. The process that performed the initial fork is returned via Event's `getProcess` member function, while the newly created process can be found via `EventFork`'s `getChildProcess` member function.

Associated EventType Code:
`Fork`

EventFork Member Functions:
`Process::const_ptr getChildProcess() const`
This function returns a pointer to the `Process` object that represents the newly created child process.

3.15.15. EventSignal

An `EventSignal` triggers when a process receives a UNIX style signal.

Associated EventType Code:
`Signal`

EventSignal Member Functions:
`int getSignal() const`
This function returns the signal number that triggered the `EventSignal`.

3.15.16. EventRPC

An EventRPC triggers when a process or thread completes a ProcControlAPI iRPC (see Sections 2.3 and 3.5). When a callback function receives an EventRPC, the memory and registers that were used by the iRPC can still be found in the address space and thread that the iRPC ran on. Once the callback function completes, the registers and memory are restored to their original state.

Associated EventType Code:
`RPC`

EventRPC Member Functions:
`IRPC::const_ptr getIRPC() const`
This function returns a const pointer to the IRPC object that completed.

3.15.17. EventSingleStep

An `EventSingleStep` triggers when a thread, which was put in single-step mode by `Thread::setSingleStep`, completes a single step operation. The `Thread` will remain in single-step mode after completion of this event (presuming it has not be explicitly disabled by `Thread::setSingleStep`).

Associated EventType Code:
`SingleStep`

3.15.18. EventLibrary

An `EventLibrary` triggers when the process either loads or unloads a shared library. ProcControlAPI will not trigger an `EventLibrary` for library unloads associated with a

Process being terminated, and it will not trigger `EventLibrary` for library loads that happened before a ProcControlAPI attach operation.

It is possible for multiple libraries to be loaded or unloaded at the same time. In this case, an `EventLibrary` will contain multiple libraries in its load and unload sets.

Associated EventType Code:
`Library`

EventLibrary Member Functions:

`const std::set<Library::ptr> &libsAdded() const`
This function returns the set of Library objects that were loaded into the target process' address space. The set will be empty if no libraries were loaded.

`const std::set<Library::ptr> &libsRemoved() const`
This function returns the set of libraries that were unloaded from the target process' address space. The set will be empty if no libraries were unloaded.

3.15.19. EventPreSyscall, EventPostSyscall

An `EventPreSyscall` is triggered when a thread enters a system call, provided that the thread is in system call tracing mode. An `EventPostSyscall` is triggered when a system call returns. These are both children of `EventSyscall`, which provides all the relevant methods.

Associated EventType Code:
`Syscall`

EventPreSyscall and EventPostSyscall Member Functions:

`Dyninst::Address getAddress() const`
This function returns the address where the system call occurred.

`MachSyscall getSyscall() const`
This function returns information about the system call. See Appendix B for information about the `MachSyscall` class.

3.16. Platform-Specific Features

The classes described in this section are all used to configure platform-specific features for `Process` objects. The three tracking classes (`LibraryTracking`, `ThreadTracking`, `LWPTracking`) all contain member functions to set either interrupt-driven or polling-driven handling for different events associated with `Process` objects. When interrupt-driven handling is enabled, the associated process may be modified to accommodate timely handling (e.g., inserting breakpoints). When polling-driven handling is enabled, the associated process is not modified and events are handled on demand by calling the appropriate "refresh" member function. All of these classes are defined in `PlatFeatures.h`.

3.16.1. LibraryTracking

The `LibraryTracking` class is used to configure the handling of library events for its associated `Process`.

LibraryTracking Declared In:
`PlatFeatures.h`

LibraryTracking Static Member Functions:

`static void setDefaultTrackLibraries(bool b)`
Sets the default handling mechanism for library events across all `Process` objects to interrupt-driven (`b = true`) or polling-driven (`b = false`).

`static bool getDefaultTrackLibraries()`
Returns the current default handling mechanism for library events across all `Process` objects. A return value of `true` indicates interrupt-driven while `false` indicates polling-driven.

LibraryTracking Member Functions:

`bool setTrackLibraries(bool b) const`
Sets the library event handling mechanism for the associated `Process` object to interrupt-driven (`b = true`) or polling-driven (`b = false`).

Returns `true` on success and `false` on failure.

`bool getTrackLibraries() const`
Returns the current library event handling mechanism for the associated `Process` object. A return value of `true` indicates interrupt-driven while `false` indicates polling-driven.

`bool refreshLibraries()`
Manually polls for queued library events to handle.

Returns `true` on success and `false` on failure.

3.16.2. ThreadTracking

The `ThreadTracking` class is used to configure the handling of thread events for its associated `Process`.

ThreadTracking Declared In:
`PlatFeatures.h`

ThreadTracking Static Member Functions:

`static void setDefaultTrackThreads(bool b)`
Sets the default handling mechanism for thread events across all `Process` objects to interrupt-driven (`b = true`) or polling-driven (`b = false`).

`static bool getDefaultTrackThreads()`
Returns the current default handling mechanism for thread events across all `Process` objects. A return value of `true` indicates interrupt-driven while `false` indicates polling-driven.

ThreadTracking Member Functions:

`bool setTrackThreads(bool b) const`

Sets the thread event handling mechanism for the associated `Process` object to interrupt-driven (`b = true`) or polling-driven (`b = false`).

Returns `true` on success and `false` on failure.

`bool getTrackThreads() const`

Returns the current thread event handling mechanism for the associated `Process` object. A return value of `true` indicates interrupt-driven while `false` indicates polling-driven.

`bool refreshThreads()`

Manually polls for queued thread events to handle.

Returns `true` on success and `false` on failure.

3.16.3. LWPTracking

The `LWPTracking` class is used to configure the handling of LWP events for its associated `Process`.

LWPTracking Declared In:

`PlatFeatures.h`

LWPTracking Static Member Functions:

`static void setDefaultTrackLWPs(bool b)`

Sets the default handling mechanism for LWP events across all `Process` objects to interrupt-driven (`b = true`) or polling-driven (`b = false`).

`static bool getDefaultTrackLWPs()`

Returns the current default handling mechanism for LWP events across all `Process` objects. A return value of `true` indicates interrupt-driven while `false` indicates polling-driven.

LWPTracking Member Functions:

`bool setTrackLWPs(bool b) const`

Sets the LWP event handling mechanism for the associated `Process` object to interrupt-driven (`b = true`) or polling-driven (`b = false`).

Returns `true` on success and `false` on failure.

`bool getTrackLWPs() const`

Returns the current LWP event handling mechanism for the associated `Process` object. A return value of `true` indicates interrupt-driven while `false` indicates polling-driven.

`bool refreshLWPs()`

Manually polls for queued LWP events to handle.

Returns `true` on success and `false` on failure.

3.16.4. FollowFork

The FollowFork class is used to configure ProcControlAPI's behavior when the associated Process forks.

FollowFork Declared In:
PlatFeatures.h

FollowFork Types:
```
typedef enum {
   None,
   ImmediateDetach,
   DisableBreakpointsDetach,
   Follow
} follow_t
```
The follow_t type represents the configurable behaviors under forking.

None is specified when fork tracking is not available for the current platform.

ImmediateDetach means that forked children are never attached to.

DisableBreakpointsDetach means that inherited breakpoints are removed from forked children, and then the children are detached.

Follow is the default behavior, and it means that forked children are attached to and remain under full control of ProcControlAPI.

FollowFork Static Member Functions:
```
static void setDefaultFollowFork(follow_t f)
```
This function sets the default forking behavior across all Process objects to f.

```
static follow_t getDefaultFollowFork()
```
This function returns the current default forking behavior across all Process objects.

FollowFork Member Functions:
```
bool setFollowFork(follow_t f) const
```
This function sets the forking behavior for the associated Process object to f.

This function returns true on success and false on failure.

```
follow_t getFollowFork() const
```
This function returns the current forking behavior for the associated Process object.

3.16.5. SignalMask

The SignalMask class is used to configure the signal mask for its associated Process.

SignalMask Declared In:
`PlatFeatures.h`

SignalMask Types:
`dyn_sigset_t`
On POSIX systems, this type is equivalent to `sigset_t`.

SignalMask Static Member Functions:
`static void setDefaultSigMask(dyn_sigset_t s)`
This function sets the default signal mask across all `Process` objects to `s`.

`static dyn_sigset_t getDefaultSigMask()`
This function returns the current default signal mask across all `Process` objects.

SignalMask Member Functions:
`bool setSigMask(dyn_sigset_t s)`
This function sets the signal mask for the associated `Process` object to `s`.

This function returns `true` on success and `false` on failure.

`dyn_sigset_t getSigMask() const`
This function returns the current signal mask for the associated `Process` object.

Appendix A. Registers

This appendix describes the `MachRegister` interface, which is used for accessing registers in ProcControlAPI. The entire definition of `MachRegister` contains more register names than are listed here; this appendix only lists the registers that can be accessed through ProcControlAPI.

An instance of `MachRegister` is defined for each register ProcControlAPI can name. These instances live inside a namespace that represents the register's architecture. For example, we can name a register from an AMD64 machine with `Dyninst::x86_64::rax` or a register from a Power machine with `Dyninst::ppc32::r1`.

All functions, types, and objects listed below are part of the C++ namespace `Dyninst`.

Declared In:
`dyn_regs.h`

Related Types:

`typedef unsigned long MachRegisterVal`

The `MachRegisterVal` type is used to represent the contents of a register. If a register's contents are smaller than `MachRegisterVal`, then it will be up cast into a `MachRegisterVal`.

```
typedef enum {
     Arch_none,
     Arch_x86,
     Arch_x86_64,
     Arch_ppc32,
     Arch_ppc64
} Architecture
```

The `Architecture` enum represents a system's architecture.

Related Global Functions

`unsigned int getArchAddressWidth(Architecture arch)`

Returns the size of a pointer, in bytes, on the given architecture, `arch`.

MachRegister Static Member Functions

```
MachRegister getPC(Dyninst::Architecture arch)
MachRegister getFramePointer(Dyninst::Architecture arch)
MachRegister getStackPointer(Dyninst::Architecture arch)
```

These functions respectively return the register that represents the program counter, frame pointer, or stack pointer for the given architecture. If an architecture does not support a frame pointer (ppc32 and ppc64) then getFramePointer returns an invalid register.

MachRegister Member Functions

`MachRegister getBaseRegister() const`
This function returns the largest register that may alias with the given register. For example, `getBaseRegister` on `x86_64::ah` returns `x86_64::rax`.

`Architecture getArchitecture() const`
This function returns the `Architecture` for this register.

`bool isValid() const`
This function returns true if this register is valid. Some API functions may return invalid registers upon error.

```
MachRegisterVal getSubRegVal(
    MachRegister subreg,
    MachRegisterVal orig)
```
Given a value for this register, `orig`, and a smaller aliased register, `subreg`, then this function returns the value of the aliased register. For example, if this function were called on `x86::eax` with `subreg` as `x86::al` and an `orig` value of `0x11223344`, then it would return `0x44`.

`const char *name() const`
This function returns a human readable name that identifies this register.

`unsigned int size() const`
This function returns the size of the register in bytes.

`signed int val() const`
This function returns a unique integer that represents this register. This can be useful for writing switch statements that take a MachRegister. The unique integers for a MachRegister can be found by preceding the register object name with an 'i'. For example, a switch statement for `MachRegister`, `reg`, could be written as:

```
switch (reg.val()) {
     case x86_64::irax:
     case x86_64::irbx:
     case x86_64::ircx:
     …
}
```

`bool isPC() const`
`bool isFramePointer() const`
`bool isStackPointer() const`
These functions respectively return true if the register is the program counter, frame pointer, or stack pointer for its architecture. They return false otherwise.

MachRegister Objects

The following list describes instances of MachRegister that can be passed to ProcControlAPI. These can be named by prepending the namespace to the listed names, e.g., `x86::eax`.

```
namespace x86
```

eax	edi	fs
ebx	oeax	gs
ecx	eip	ss
edx	flags	fsbase
ebp	cs	gsbase
esp	ds	
esi	es	

```
namespace x86_64
```

rax	r9	flags
rbx	r10	cs
rcx	r11	ds
rdx	r12	es
rbp	r13	fs
rsp	r14	gs
rsi	r15	ss
rdi	orax	fsbase
r8	rip	gsbase

```
namespace ppc32
```

r0	r13	r26
r1	r14	r27
r2	r15	r28
r3	r16	r29
r4	r17	r30
r5	r18	r31
r6	r19	fpscw
r7	r20	lr
r8	r21	cr
r9	r22	xer
r10	r23	ctr
r11	r24	pc
r12	r25	msr

```
namespace ppc64
```

r0	r9	r18
r1	r10	r19
r2	r11	r20
r3	r12	r21
r4	r13	r22
r5	r14	r23
r6	r15	r24
r7	r16	r25
r8	r17	r26

r27	r31	xer
r28	fpscw	ctr
r29	lr	pcmsr
r30	cr	

`namespace aarch64`

x0	x23	q15
x1	x24	q16
x2	x25	q17
x3	x26	q18
x4	x27	q19
x5	x28	q20
x6	x29	q21
x7	x30	q22
x8	q0	q23
x9	q1	q24
x10	q2	q25
x11	q3	q26
x12	q4	q27
x13	q5	q28
x14	q6	q29
x15	q7	q30
x16	q8	q31
x17	q9	sp
x18	q10	pc
x19	q11	pstate
x20	q12	fpcr
x21	q13	fpsr
x22	q14	

Appendix B. System Calls

The MachSyscall class, found in MachSyscall.h, represents system calls in a platform-independent manner. Currently, syscall events are only supported on Linux.

MachSyscall Methods

`SyscallIDPlatform num() const`

Returns the platform-specific syscall number

`SyscallName name() const`

Returns the name of the system call (e.g. “getpid”)

`bool operator==(const MachSyscall&) const`

Equality operator. Two system calls are equal if they are for the same platform and have the same syscall number.

`static MachSyscall makeFromPlatform(Platform, SyscallIDIndependent)`

Constructs a MachSyscall from a Platform and a platform-independent ID (e.g. dyn_getpid). The platform-independent syscall IDs may be found in dyn_syscalls.h.

Appendix C. Known Issues

Prior to Linux 2.6.38, some kernels allowed the debug interface to return multiple pending signals without receiving an explicit debugger continue. Proccontrol's architecture relies on receiving a single debugger event for each continue that it issues except at exit time. This can cause an unrecoverable assertion. We have only observed this behavior when a process is receiving signals both from itself and from another process, and we have only observed it when the self-signaling behavior is a breakpoint. This behavior does not occur with 2.6.38 and subsequent kernels, and it has not been observed on any kernels with utrace support (which covers all RedHat kernels that would otherwise be affected by this kernel bug).

Library load/unload, user-level thread creation/destruction, inferior RPCs, and user-inserted breakpoints can all cause self-signaling, and a process's children exiting or stopping will cause the parent to be signaled as well. While we cannot provide a general prescription for avoiding this bug (other than upgrading to an unaffected kernel), the above should suggest strategies for reducing the likelihood you will be affected by it.