Paradyn Parallel Performance Tools

ParseAPI Programmer's Guide

8.2 Release
Aug 2014

Computer Sciences Department
University of Wisconsin–Madison
Madison, WI 53706

Computer Science Department
University of Maryland
College Park, MD 20742

Email bugs@dyninst.org
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 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;
}
```

3.2 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
```

```
19          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.

```
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.

```
void parseGaps(CodeRegion *cr)
```

Speculatively parse the indicated region of the binary using pattern matching to find likely function entry points. Only enabled on the x86 32-bit platform.

```
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.

```
funclist & funcs()
```

Returns a 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 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.

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

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 std::vector<Block*> 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	edgelist &	List of outgoing call edges from this function.
returnBlocks	blocklist &	List of all blocks ending in return edges.
exitBlocks	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`.

`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, direct branches under certain circumstances).

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(set<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 root LoopTreeNode instance has a null loop member since a function may contain multiple outer loops. The outer loops 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 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. This mode of parsing will be brought to full functionality in an upcoming release.