DynC API Manual

Version 0.0.2

DRAFT

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`:

```
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");
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);
    entryArgs.push_back(funcName);
    entryArgs.push_back(intCounter);
```

Build the exit `printf`:

```
    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));
    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`functionName, intCounter);

// myExitDynC.txt
printf("Leaving %s.\n", dyninst`functionName);
```

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 all DynC API with each function's entry and exit points:

```
BPatch_snippet *entrySnippet =
    dynC_API::createSnippet(entryString.str().c_str(), entry_point, "entrySnippet");
BPatch_snippet *exitSnippet =
    dynC_API::createSnippet(exitString.str().c_str(), exit_point, "exitSnippet");
```

Finally insert the snippets at each function's entry and exit points:

```
appProc->insertSnippet(*entrySnippet, entry_point);
appProc->insertSnippet(*exitSnippet, exit_point);
```

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. If a snippet name is not specified, the default name `Snippet_[#]` is used.

`<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: DynC API input options: code snippets

`<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: DynC API input options: location

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.

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. This includes parameters and return values. Mutatee variables must be accessed through the `inf` 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 the variables and methods of the mutatee.
`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

Note: `inf‘<variable>` searches first for local variables named `<variable>`, then for matching globals if no locals are found.

Example:

```
inf‘printf("n is equal to %d.\n", ++inf‘n);
```

This would print the value of the mutatee variable n, then increment 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 Executed Code

Code enclosed by a pair of `{% <code> %}` is executed only once by a snippet. This creates an ideal section for declaring variables and displaying a message only once.
Example Touch:

```
{%
    inf`printf("Function %s has been touched.\n", dyninst`function_name);
%}
```

DynC will only execute the code in a first-only section for snippets with the same name. Thus, if a snippet is created by calling `createSnippet(...)` with a unique snippet name, the first-only code will be executed upon reaching the first point encountered in the execution of the mutatee where the snippet was inserted.

For example, if the previous example was created with the name `fooTouchSnip` and inserted at the entry to function `foo`, the output would be:

```
Function foo has been touched.
(mutatee exit)
```

If `createSnippet(...)` was called with Example Touch multiple times and each snippet was given the same name, but was 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 Example Touch multiple times and the snippets were 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)
```

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, to create a non-static snippet-local variable:

```
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 the snippet is executed, the variable is 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++);
```

The output from the above if, for example, inserted at the entrance to a function that is called four times 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 one-time section will also behave statically.

```
{%
    int i = 10;
%}
```

2.3.2 Global Variables

To declare a global variable, accessible by 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.
```

2.3.3 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";
```

2.3.4 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, the statement `*s` would evaluate to the character `h`.

2.3.5 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", inf`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. The 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 pretty name of the function where the snippet is executing. Requires that call to `createSnippet(...)` specifies a `BPatch_point`.
`module_name`	char *	Anywhere	Evaluates to the name of the current module. Requires call to `createSnippet(...)` to specifies 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 thread 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 break.
`stopthread()`	void	Anywhere	Causes the thread on which the snippet is executing to stop.

Table 4: Dyninst Domain Values