Paradyn Parallel Performance Tools

DyninstAPI Test Suite

Release 3.0
January 2002

Dyninst Project
Computer Sciences Department
University of Maryland
College Park, MD 20742
dyninst@cs.umd.edu

Table Of Contents

Table Of Contents

INTRODUCTION

The dyninst library provides facilities to patch code into a running program. It implements an interface, the dyninstAPI, for users to call the functionality of the library. In this paper, we discuss the dyninstAPI test suite in detail.

The dyninstAPI test suite is used to verify that the dyninst library has been installed correctly. It is also used by the developers of the dyninst library during regression testing. There are four test programs and up to twenty mutatee programs.

The dyninstAPI introduces two primary abstractions of a running program and its state. These abstractions are *instrumentation points* and *snippets*. To support multiple processes, two additional abstractions, *images* and *processes*, are included in the dyninstAPI. The test programs of the test suite manipulate multiple application processes. They access instrumentation points in the application images and formulate snippets for insertion into the processes.

The test suite encompasses most of the dyninstAPI methods. First, it covers the classes used to manipulate code in execution. Related classes include BPatch and BPatch_thread. Second, it uses a group of classes, BPatch_image, BPatch_module, and BPatch_function, for accessing the original program and its data structures. Third, the test suite covers the classes to construct and insert new instrumentation code. These classes include BPatch_snippet and BPatch_point. Finally, the BPatch_type class of the dyninstAPI provides a type system. The test suite uses this type system to access existing application variables and to allocate variables for use in code snippets.

In the following sections, we present the test programs and their corresponding mutatee programs.

1 TEST1

The test1 program examines the basic features of the dyninstAPI. Test cases 1 through 32 test the interfaces that the mutator program uses to access the mutatee program and its data structures. They also cover the interfaces for constructing new code snippets and inserting them. Test cases 33 through 44 again use these interfaces. However, the main goal of these tests is to explore dyninst support for the C++ language features, such as inheritance and function overloading. The mutator is implemented in test1.C. The corresponding mutatee is implemented in test1.mutatee.c. Utility functions are implemented in test_util.h and test_util.C. The file cpp_test.h defines C++ test case classes.

1.1 Mutator structures and important data variables

As the test1 mutator program begins executing, command line options are parsed in the *main* subroutine. Initialization proceeds with a call to the *mutatorMAIN* function. A single instance of BPatch *bpatch* is created, and the mutatee program is started. Test case functions are called to insert code snippets.

Mutator test functions are called *mutatorTestXX*. Local variables within a test case function are named *exprXX_YY* and *pointXX_YY,* where *XX* denotes test case number, and *YY* is the instance number within a test case.

The constant integer *MAX_TEST* denotes the total number of test cases implemented in test1.C. The boolean variable *runAllTests* defaults to true for activating all test cases. The boolean

variable *runCpp* switches on the C++ test cases when the mutatee executable is built with C++ compilers. The boolean array *runTest* indicates whether to execute a specific test case depending on the command line option `run` specified when a user executes the test1 program. The boolean array *passedTest* records the passed test cases.

1.2 Mutatee structures and important data variables

As the test1 mutatee program begins executing, command line options are parsed in *main* subroutine. Variables are initialized to control test case coverage. The mutatee performs a test by invoking the appropriate test case function *funcXX_1*.

The mutatee defines data variables for the mutator to access, verify and manipulate. They are named *globalVariableXX_YY, constVarXX*, *RETXX_YY* and *MAGICXX_YY.* Test case functions are named *funcXX_YY.* These are stub functions to which the mutator can attach instrumentation snippets. Auxiliary functions are named *callXX_YY.* The mutatee uses these to determine the correctness of respective tests by checking their return values, parameter values, or global variable values.

1.3 How to add a new test case

Adding a new test case requires the following steps:

1. Increment the *MAX_TEST* counts in both test1.C and test1.mutatee.c.
2. On the mutator side, implement a *mutatorTestXX* function for the new test case. Call this function from the function *mutatorMAIN* after calling the existing test case functions.
3. On the mutatee side, implement *funcXX_YY* functions, declare *globalVariableXX_YY* variables, and call this function from the function *main* (refer to Section 1.5.1 for details on adding new C++ test cases).

1.4 Language-independent test case descriptions

We first examine the language-independent test cases in test1.

Test1.1 (zero-argument function call)

Test1.1 verifies inserting calls to zero-argument functions into a mutatee. In the *mutatorTest1* function, the mutator finds the zero-argument function *call1_1* in the mutatee module. The mutator then inserts a snippet that calls the function into the entry point of the *func1_1* function. The mutatee determines the correctness of the above operations by examining the value of *globalVariable1_1*, which was set by the inserted snippet.

Test1.2 (multiple-argument function call)

Test1.2 verifies inserting calls to multiple-argument functions into a mutatee (passing constant arguments). These arguments are integers and character strings. In the *mutatorTest2* function, the mutator finds the multiple-argument function *call2_1* in the mutatee module. It inserts a snippet

that calls this function at the entry point of the *func2_1* function. Then the mutatee determines the correctness of the above operations by examining whether correct input parameter values were passed by the inserted snippet.

Test1.3 (passing variables to functions)

Test1.3 verifies inserting calls to multiple-argument functions into a mutatee (passing variable arguments). These arguments are integers. In the *mutatorTest3* function, the mutator finds the two-argument function *call3_1*. It inserts a snippet that calls the function at the entry point of *func3_1* function. Then the mutatee determines the correctness of the above operations by examining if correct parameter values were passed by the inserted snippet.

Test1.4 (snippet execution sequence)

Test1.4 verifies that the execution order of the inserted code snippets is correct. In the *mutatorTest4* function, the mutator first finds *globleVariable4_1*. Then it constructs snippets which assign *globleVariable4_1* the values of 42 and 43. These snippets are inserted at the entry point of the *func4_1* function. In function *func4_1*, the final value of the *globleVariable4_1* is examined to determine if the snippets were executed in correct order.

Test1.5 (construct if statement without else branches)

Test1.5 constructs IF statements (without ELSE branches). In the *mutatorTest5* function, the mutator constructs a snippet of two IF statements, that assign values to *globalVariabl5_1* and *globalVariable5_2*. It then inserts the snippet at the entry point of the *func5_2* function. In function *func5_2*, the values of the *globalVariable5_1* and the *globalVariable5_2* are examined to determine the correctness of the operations.

Note that test1.16 investigates IF-THEN-ELSE statements.

Test1.6 (arithmetic operators)

Test1.6 examines the arithmetic operators provided by the dyninstAPI. The operators verified in this test are addition, subtraction, division, multiplication, and the comma operator, for both constants and variables. In the *mutatorTest6* function, the mutator finds mutatee variables *globalVariable6's*. It then constructs snippets using the arithmetic operators. The operands of the operators are either constants or variables declared in the mutatee module. The results of the operations are assigned to the *globalVariable6's*. The mutator inserts the snippets at the entry point of the *func6_2* function. In function *func6_2*, the mutatee examines the values of the *globalVariable6's* to determine correctness.

Test1.7 (relational operators)

Test1.7 examines relational operators provided by the dyninstAPI. The operators verified in this test are BPatch_lt, BPatch_eq, BPatch_gt, BPatch_le, BPatch_and, BPatch_or, BPatch_ne, and BPatch_ge. In the *mutatorTest7* function, the mutator finds the variables *globalVariable7's* and

constVars declared in the mutatee module. It then constructs snippets of IF statements, that assign values to the *globalVariable7's* using the relational operators. The operands of the relational operators are either constants or the variable *constVar*s. These snippets are subsequently inserted at the entry point of the *func7_2* function. In function *func7_2*, the values of the *globalVariable7's* are examined to determine the correctness of the inserted snippets.

Test1.8 (preserve registers upon expression insertions)

Test1.8 verifies whether inserting complex AST expressions overwrites mutatee function parameter registers. Mutatee function *func8_1* contains ten integer parameters. In the *mutatorTest8* function, the mutator constructs the nested AST expression *globalVariable8_1 = ((81+82)+(83+84))+((85+86)+(87+88))*. This expression is then inserted at the entry point of *func8_1*. *Func8_1* contains a long list of parameters, and we have inserted a complex AST expression at its entry point. We examine if the *func8_1's* parameters maintain their original values after the expression insertion to determine correctness.

Test1.9 (preserve registers upon function insertions)

Test1.9 verifies whether inserting snippets that call functions will overwrite mutatee function parameter registers. Mutatee function *func9_1* contains ten integer parameters. In the *mutatorTest9* function, we construct a function call snippet that passes five parameters to *call9_1* function. This call snippet is then inserted at the entry point of the *func9_1* function. *Func9_1* contains a long list of parameters, and we have inserted a function call snippet at its entry point. We examine if the *func9_1*'s parameters maintain their values after the call snippet insertion to determine correctness.

Test1.10 (insert snippet order)

Test1.10 verifies whether snippets are inserted into mutatees in the requested order. We insert one snippet and then request two more to be inserted, one before the first snippet and the other after it.

In the *mutatorTest10* function, the mutator finds the zero-argument functions *call10_1*, *call10_2* and *call10_3* defined in the mutatee module. These functions set *globalVariable10's* to predetermined values. The mutator inserts a call to *call10_2* at the entry point of the *func10_1* function, and then inserts calls to *call10_1* and *call10_3* before and after the *call10_2* snippet, respectively. In function *func10_1* of the mutatee module, the values of the *globalVariable10's* are examined to determine whether the inserted snippets were executed in the correct sequence.

Test1.11 (snippets at entry, exit and call points)

Test1.11 verifies inserting snippets at the entry, call-site and exit points of a function. In the *mutatorTest11* function, the mutator finds the functions *call11_1*, *call11_2*, *call11_3* and *call11_4* defined in the mutatee module. It then inserts a call to *call11_1* snippet at the entry point of *func11_1*, it also inserts calls to *call11_2* and *call11_3* before and after the call-site point of the *func11_1* function, respectively. Last, the mutator inserts a call to *call11_4* snippet at the exit

point of the *func11_1* function. If any of the above operations fail, the mutator will exit. Otherwise, the mutatee assumes success and does not conduct validation.

Test1.12 (insert/remove, and malloc/free)

This test case contains two parts, 12a and 12b.

Test1.12a: Verifies inferior memory allocation and deallocation. The function *mutatorTest12a* continuously allocates segments of memory until it exhausts the heap, at which point it reclaims all of the memory. Then a small amount of memory is allocated once again. The *mutatorTest12a* function also inserts a call snippet to *call12_1* at the entry point of the *func12_2* function. Since the inserted snippet increments *globalVariable12_1*, the mutatee determines the correctness of the above operations by inspecting *globalVariable12_1*.

Test1.12b: Verifies the removal of inserted snippets. It also deallocates the heap memory allocated earlier in the test1.12a. First, the *func12_1* function stops the mutatee's execution. The mutator waits for the mutatee process status change, then begins to delete the call snippet to the *call12_1* function inserted earlier in 12a. It also deallocates the heap memory allocated in the test1.12a by freeing *varExpr12_1*, defined in the mutator module. This sub-test case is dependent on test1.12a.

Test1.13 (paramExpr, nullExpr and retExpr)

Test1.13 examines BPatch_paramExpr, BPatch_nullExpr, and BPatch_retExpr expressions. In *mutatorTest13*, the mutator finds the *call13_1* function in the mutatee. It constructs a snippet that calls *call13_1*. A five-element BPatch_paramExpr is passed to *call13_1* as its argument. The mutator then inserts the snippet and a BPatch_nullExpr at the entry point of *func13_1*. To test BPatch_retExpr, the mutator constructs a snippet that calls *call13_2*. A one-element BPatch_retExpr is passed to the call13_2 function as its argument. This snippet is then inserted at the exit point of *func13_2*. To determine whether execution of these inserted snippets is correct, the mutatee inspects the parameter values of *call13_1* and *call13_2* and decides correctness.

Test1.14 (replace/remove function call)

Test1.14 verifies function replacement and removal in mutatees. Initially mutatee function *func14_1* contains calls to *func14_2* and *func14_3*. In the *mutatorTest14* function, the mutator replaces the *func14_2* call with a call to *call14_1*. It also removes the *func14_3* call from the *func14_1* function. To determine the correctness of the replacement and removal, the mutatee examines the values of *globalVariable14_1* and *globalVariable14_2* to determine if the replaced code was executed and the removed code omitted.

Test1.15 (setMutationActive)

Test1.15 verifies the correct operation of *setMutationsActive* method, which enables or disables the execution of all snippets for the mutatee thread. This test case contains two parts, 15a and 15b.

Test1.15a: Initially, the mutatee function *func15_4* contains a call to *func15_3*. In the *mutatorTest15a* function, the mutator replaces the *func15_3* call with a call to *call15_3*. It also inserts a call to *call15_1* at the entry point of *func15_2*.

Test1.15b: In the *mutatorTest15b* function, the mutator disables and then enables all the inserted snippets via the *setMutationsActive* function call. In the *func15_1* function of the mutatee module, the values of *globalVariable15's* are examined to determine correctness. Sub-test 15b is dependent on 15a.

Test1.16 (construct if-then-else statement)

Test1.16 verifies the construction of IF-THEN-ELSE clauses. In the *mutatorTest16* function, the mutator finds *globalVariable16's*, defined in the mutatee module. It then constructs a few IF-THEN-ELSE code snippets in which values are assigned to the *globalVariable16's*. These IF-THEN-ELSE snippets are subsequently inserted at the entry points of *func16_2, func16_3,* and *func16_4*. In *func16_1,* the values of the *globalVariable16's* are examined to determine correctness.

Note that test1.5 investigates IF statements without an ELSE branch.

Test1.17 (return values from function calls)

Test1.17 verifies that instrumentation inserted at a subroutine's exit point does not overwrite the subroutine's return values. In the *mutatorTest17* function, the mutator instruments the mutatee to call *call17_1* with one constant parameter at the exit point of *func17_1*. Similarly the exit point of *func17_2* is instrumented to call *call17_2* with one constant parameter. In *func17_1,* the mutatee compares the return values of *func17_1* and *func17_2* to determine correctness.

Test1.18 (read/write a variable in the mutatee)

Test1.18 verifies the reads and writes of global variables in a mutatee. In the *mutatorTest18* function, the mutator finds *globalVariable18_1* declared in the mutatee module. It reads the variable's original value, and assigns a new value to it. To determine the correctness of the read and write operations, we examine *globalVariable18_1's* original value in the mutator and examine its new value in the mutatee.

Test1.19 (oneTimeCode)

Test1.19 verifies the correct operation of *oneTimeCode,* which causes a snippet expression to be evaluated in the mutatee. On the mutatee side, *func19_1* stops the mutatee process from running. The mutator waits for this mutatee process status change, constructs a piece of oneTimeCode, and resumes the mutatee's execution. This piece of oneTimeCode is then executed. The mutator proceeds to construct a second piece of oneTimeCode. However, the second piece of oneTimeCode never gets a chance to execute. The first piece of oneTimeCode assigns a predetermined value to *globalVariable19_1*. The mutatee examines the variable's final value to determine its correctness.

Test1.20 (instrument arbitrary points)

Test1.20 verifies instrumentation at arbitrary points in a function. In the *mutatorTest20* function, the mutator finds all instruction points in the *func20_2* function. It then inserts calls to *call20_1* at each of these instruction points. Mutatee function *func20_1* subsequently calls the instrumented *func20_2* function. Since the inserted snippets assign *globalVariable20's* with new values, the *func20_1* examines their final values to determine correctness. Test1.20 is currently only implemented on AIX and ALPHA platforms.

Test1.21 (findFunction in module)

Test1.21 verifies the correct operation of the *findfunction* method. In the *mutatorTest21* function, the mutator loads the shared libraries *libtestA* and *libtestB* into the mutatee's image. It then tries to locate function *call21_1* within the shared libraries. If any of the above operations fail, the mutator exits. Otherwise, the mutatee assumes success and does not conduct any validation. Test1.21 is not currently implemented on Windows NT. It is also not implemented on AIX because dynamic linking to shared libraries is not supported on that platform.

Test1.22 (replace functions)

Test1.22 verifies function replacements in mutatee modules. In the *mutatorTest22* function, the mutator loads specific modules into the mutatee's image. After locating the necessary functions, the mutator proceeds with the following four replacement tests:

- It replaces a function defined in the mutatee executable with another function defined in that executable.
- It replaces a function defined in the executable with another function defined in a shared library.
- It replaces a function defined in a shared library with another function defined in a second shared library,
- It replaces a function defined in a shared library with another function defined in the mutatee executable.

Test1.22 requires shared library loading and is only implemented on SPARC_SOLARIS and ALPHA platforms at this point.

Test1.23 (local variables)

Test1.23 verifies finding and manipulating local variables. In the *mutatorTest23* function, the mutator finds *globalVariable23's* defined in the mutatee. It also finds local variables defined inside a mutatee function scope. The mutator then assigns values to each of these variables. In mutatee function *call23_1*, we examine the values of the *globalVariable23's* and the local variables to determine correctness. Test1.23 is not implemented on NT or IRIX platforms.

Test1.24 (array variables)

Test1.24 verifies finding and manipulating array variables. In the *mutatorTest24*, the mutator finds the one-dimensional array *globalVariable24_1* and the two-dimensional array *globalVariable24_8*. Both of the arrays are defined in the mutatee. It also finds the one-dimensional array *localVariable24_1*, defined in a mutatee function. The mutator then constructs snippets and inserts them at the call-site points of the *call24_1* function. The inserted snippets assign values to the arrays' elements. In *func24_1*, the values of these array elements are examined to determine correctness.

Note that the two-dimensional array *globalVariable24_8* is not square so that we may test the array element address computation. Test1.24 is not implemented on NT or IRIX platforms.

Test1.25 (unary operators)

Test1.25 verifies the unary operators provided by the dyninstAPI. The operators are BPatch_addr, BPatch_deref and BPatch_negate. In the *mutatorTest25* function, the mutator finds *globalVariable25's* defined in the mutatee. It then constructs snippets to assign values to the variables using the unary operators. In *func25_1*, the values of the *globalVariable25's* are examined to determine correctness. This test case is not implemented on IRIX platform.

Test1.26 (field operators)

Test1.26 verifies accessing component fields in a structure. The mutatee defines a structure. In the *mutatorTest26* function, the mutator finds mutatee variables *globalVariable26_1* and *localVariable26_1*, which are of the defined structure type. The mutator proceeds to assign the values of their component fields to other mutatee defined variables. In *func26_1*, the values of these variables are examined to determine correctness. This test case is not implemented on NT or IRIX platforms.

Test1.27 (type compatibility)

Test1.27 verifies type-compatibility. The mutatee defines some types. It also declares *globalVariable27's* of the same types. In the *mutatorTest27* function, the mutator examines the type-compatibility of the mutatee defined types. It then looks up the types of the *globalVariable27's* and verifies their type-compatibility. If all type-compatibility checks succeed, the mutator sets an indicator, *globalVariable27_1* so that the *func27_1* function can determine the correctness of the operations. This test case is not implemented on NT or IRIX platforms.

Test1.28 (user defined fields)

Test1.28 verifies the creation of user-defined types. In the *mutatorTest28* function, the mutator creates new structure types by calling the *createStruct* method. It also creates variables of these types. The component fields of these variables are subsequently assigned with predetermined values. The mutator then assigns the values of these component fields to the mutatee variables

globalVariable28's. In *func28_1*, the mutatee inspects the values of the *globalVariable28's* to determine correctness.

Test1.29 (BPatch_srcObj class)

Test1.29 verifies the *BPatch_srcObject* class, which represents a mutatee image. In the *mutatorTest29* function, the mutator iteratively traverses through the mutatee image to its component modules and functions. If the traversal succeeds, the mutator sets an indicator, *globalVariable29_1* so that the mutatee *func29_1* function may determine the correctness of the operations.

Note that the traversal of the mutatee image does not search for particular components (modules or functions of a specific name).

Test1.30 (line information)

Test1.30 verifies the correct operation of *getLineToAddr* methods. In the *mutatorTest30* function, the mutator retrieves the line number of the mutatee function *call30_1*. Once the mutator has the line number, it tries to obtain the function's address via *getLineToAddr* method calls. In particular, the mutator checks *getLineToAddr* methods for the mutatee image, the mutatee module, the function *call30_1*, and the mutatee thread objects. The obtained function addresses are stored in *globalVariable30's*. In *func30_1*, the mutatee examines these values to determine correctness.

Test1.31 (non-recursive base tramp guard)

Test1.31 verifies non-recursive base trampoline guards. In the *mutatorTest31* function, the mutator sets the base trampoline guards to false by calling *setTrampRecursive*. It also inserts a call to mutatee function *func31_3* at the entry point of *func31_2*. This snippet is then executed by the mutatee. The mutator proceeds to insert two additional calls to *func31_4* in *func31_3*. Since *func31_3* has already been instrumented, neither of the new call snippets should be executed. In *func31_1*, the mutatee determines the effectiveness of the trampoline guards by examining if *globalVariable31_4*'s value is reset by the inserted *func31_4* call snippets.

Test1.32 (recursive base tramp guard)

Test1.32 verifies recursive base trampoline guards. In the *mutatorTest32* function, the mutator sets the base trampoline guards to true, and inserts a call to mutatee function *func32_3* at the entry point of *func32_2*. This snippet is then executed by the mutatee. The mutator proceeds to insert two additional calls to *func32_4* in *func32_3*. The mutatee should still execute the two new snippets even though the function *func32_3* has already been instrumented. In *func32_1*, the mutatee determines the effectiveness of the trampoline guards by examining if *globalVariable32_4*'s value is reset by the inserted *func32_4* call snippets.

1.5 C++ language-specific tests

In this section, we explore the dyninst support for C++ language features. Mutatee tests in this section are written in the C++ language. We cover C++ features, including templates and overloaded functions. The test cases are numbered from 33 to 44. The C++ test case hierarchy is shown below in Figure 1.

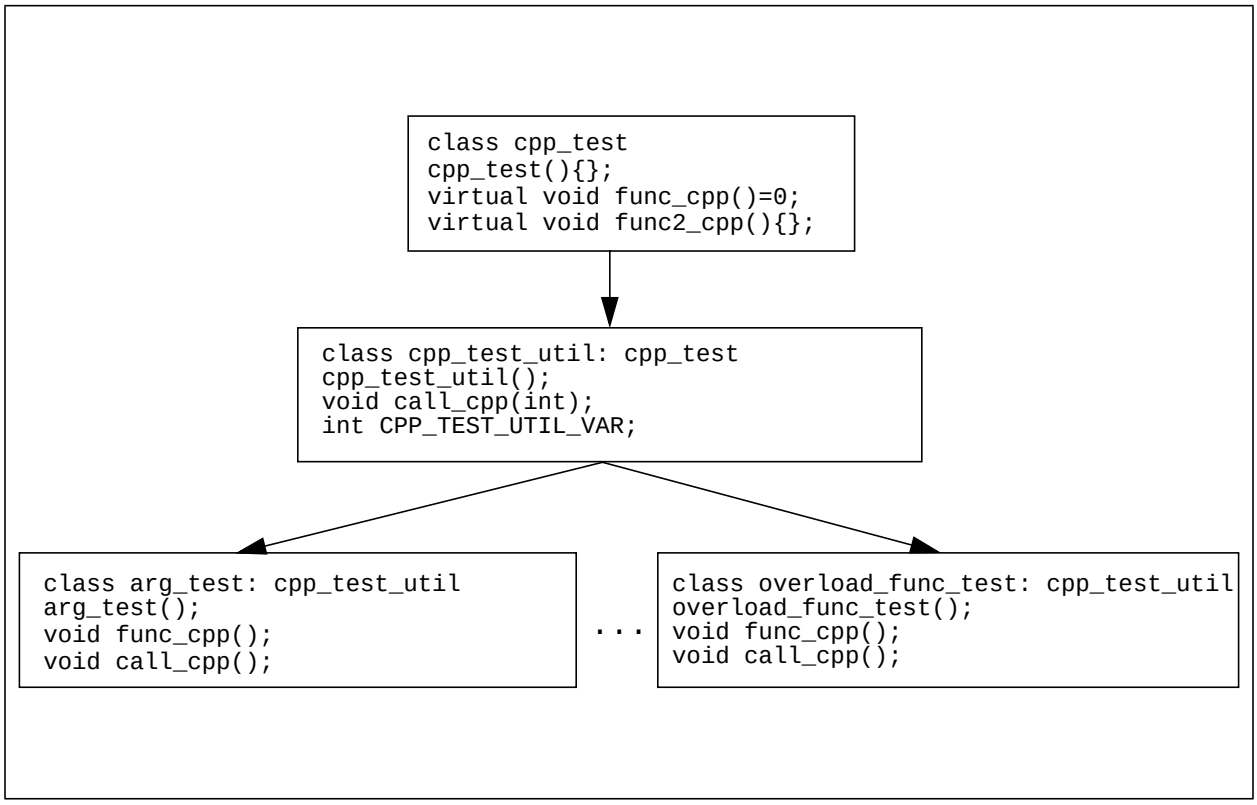

Figure 1: C++ test case class hierarchy

The class *cpp_test* serves as a base class in which virtual and pure virtual functions are defined for later tests. The derived class *cpp_test_util* provides common utilities for the derived test case classes. It also defines common class member variables and functions. The specific C++ features are examined in the derived classes. For example, the *arg_test* class tests class member function argument passing, and the *overload_func_test* class tests function overloading. In each of the test case classes, mutatee function *func_cpp* is used to determine the validity of respective tests. Mutatee function *call_cpp* is a stub function; the mutator may attach instrumentation snippets to it.

The mutatee program is compiled with either C or C++ compilers. When compiling with a C++ compiler, we need to include the C++ test cases. As shown below, we check if the preprocessor symbol *__cplusplus* has been defined to determine whether to include the C++ test cases.

```
#ifdef __cplusplus
/* C++ test cases */
void arg_test::func_cpp() {...}
...
#endif
```

The C++ compilers we use are the GNU g++ compiler and the native C++ compilers provided on the various platforms to build mutatee executables. The resulting mutatee executables are marked with corresponding extensions, such as test1.mutatee**_CC** and test1.mutatee**_g++.**

1.5.1 How to add a new C++ test

Adding a new test case involves the following steps:

1. Increment the *MAX_TEST* counts in both test1.C and test1.mutatee.c.
2. On the mutator side, implement a *mutatorTestXX* function for the new test case and call this function from the *mutatorMAIN* function after all existing test case functions.
3. On the mutatee side, declare a test case class in cpp_test.h (the class should be derived from the class *cpp_test_util*). Implement the member functions *func_cpp* and *call_cpp*. Instantiate an object of the new class and call its member function *func_cpp* from the function *main*.

1.5.2 C++ Language-specific test case descriptions

We now discuss the C++ language-specific test cases.

Test1.33 (class member function argument passing)

Test1.33 verifies class instrumenting member function argument passing. Class member functions may contain constant, reference, and default arguments. The mutatee function *arg_test::func_cpp* calls a class member function that contains constant, reference, and default arguments. In the *mutatorTest33* function, the mutator finds this class member function and verifies the existence and the type of its arguments. The mutator then inserts a call to another class member function, passing predetermined arguments. The mutatee determines the correctness of the above operations by examining the values of the passed arguments.

Test1.34 (overloaded functions)

Test1.34 verifies the dyninst support for instrumenting C++ overloaded functions. The overloaded mutatee functions *call_cpp* are defined in the mutatee class *overload_func_test*. In the *mutatorTest34* function, the mutator verifies the existence of the overloaded class member functions. It then examines the parameter numbers of these functions (note that the types of the parameters have been checked in test1.33). Finally, the mutator inserts a call to a class member function, passing predetermined arguments. The mutatee determines the correctness of the above operations by checking the values of the passed arguments.

Test1.35 (overloaded operators)

Test1.35 verifies the dyninst support for instrumenting C++ overloaded operators. The overloaded operator *operator++* is defined in the mutatee class *overload_op_test*. In the *mutatorTest35* function, the mutator verifies the existence of the overloaded operator. It then inserts a function call snippet into the mutatee, passing predetermined arguments. The mutatee determines the correctness of the above operations by checking the values of the passed arguments.

Test1.36 (static member variables and functions)

Test1.36 verifies instrumenting C++ static member variables and functions. A static member variable and a static function, which manipulates the static variable, are defined in the mutatee class *static_test*. There are multiple calls to the static member function in the *static_test::func_cpp* function. In *mutatorTest36*, the mutator verifies the existence of the static members. To confirm that there is only one instance of the static member variable, the mutator locates the static variable through the different invocation instances of the static function and verifies if the static variable's base address remains unchanged. If all the above operations proceed successfully, the mutator inserts a function call snippet to inform the mutatee of the success of the operations.

Test1.37 (namespace)

Test1.37 verifies that dyninst understands the C++ scoping in the mutatee. Variables of different scopes are declared in the mutatee. The scopes of the variables are 1) local to a function, 2) local to a file, 3) local to a class (not inherited from its parent class), and 4) global. In the *mutatorTest37* function, the mutator verifies the existence of the variables by locating them in their respective scopes. If all the above operations proceed successfully, the mutator inserts a function call snippet to inform the mutatee of the success of the operations.

Test1.38 (exceptions)

Test1.38 verifies that dyninst can instrument C++ exceptions. A sample exception class and its exception handler function are defined in the mutatee module. In the *exception_test::func_cpp* function, the mutatee throws an exception in a try clause and catches it with the handler function in a subsequent catch clause. In the *mutatorTest38* function, the mutator verifies the instrumentability of the try-catch clauses by locating the BPacth_points and finding the called functions and locally defined variables in the clauses. If all the above steps proceed successfully, the mutator instruments the exception handler function of the sample exception class so that the inserted snippet informs the mutatee of the validity of the operations.

Test1.39 (templates)

Test1.39 verifies that dyninst can instrument C++ templates. A template class is defined in the mutatee. In the member function *template_test::func_cpp*, multiple template objects are declared. Each of the objects has a different base element type. In the *mutatorTest30* function, the mutator finds the base elements of the template objects and verifies their respective types. If the base element type verification is successful, the mutator inserts a function call snippet to inform the mutatee of the success of the operations.

Test1.40 (declaration scopes)

Test1.40 verifies that dyninst understands C++ declaration scopes. Variables and objects of different scopes are declared in the mutatee. The scopes include: 1) global, 2) local to a function, and 3) local to a class (inherited from its parent class). In the *mutatorTest40* function, the mutator verifies the existence of the variables by finding them in their respective scopes. If all the above operations

proceed successfully, the mutator inserts a function call snippet to inform the mutatee of the successful operations.

Test1.41 (derived classes)

Test1.41 verifies that dyninst can instrument a C++ derived class. The test case class *derivation_test* inherits member functions and variables from its parent class in the mutatee. In the *mutatorTest41* function, the mutator verifies the derivation hierarchy by locating the inherited class member function in the *derivation_test* class scope.

Test1.42 (standard C++ libraries)

Test1.42 verifies instrumentation of standard C++ libraries. This test attempts to find functions in standard C++ libraries. In the *mutatorTest42* function, the mutator locates the standard C++ library *libstdc++* in the mutatee's image. It then tries to locate the operator *operator<<* in the standard C++ library. This test case is not implemented on NT or AIX platforms

Test1.43 (replace functions in standard C++ libraries)

Test1.43 verifies instrumentation of standard C++ libraries. This test attempts to replace functions in standard C++ libraries. After locating target functions, the mutator proceeds with the following function replacement tests.

- It replaces a function defined in a standard C++ library with another function defined in the same standard library.
- It replaces a function defined in the standard C++ library with another function defined in the mutatee executable.
- It replaces a function defined in the mutatee executable with another function defined in the standard C++ library.

  This test case is only implemented on SPARC_SOLARIS and ALPHA platforms.

Test1.44 (C++ member functions)

Test1.44 verifies that dyninst can instrument C++ member functions, including pure virtual, virtual, constant, and inline functions. Functions of these types are defined in the mutatee module. In the *mutatorTest44* function, the mutator verifies the instrumentability of the functions by locating their BPatch_points. For the inline function, the mutator discerns the inlined instance of the function inside its caller from the template instance in the defined class.

1.5.3 Makefile Changes

We build mutatees using C++ compilers that include GNU g++ and the native C++ compilers on their respective platforms. The mutatees have names such as **`test1_mutatee_g++`** and **`test1_mutatee_CC`**. In the platform-specific Makefiles, we define

```
GNU_CXX = g++
NATIVE_CXX = CC
```

Certain platforms require slight changes to compile the C++ test code, we discuss them below.

The test1.30 verifies C++ templates. Template objects of different base element types are instantiated. When compiling with the native CC compiler on SOLARIS, we add the following to allow template instantiations to be placed into the current object file and give them static linkage

```
ifeq ($(MUTATEE_CC), $(NATIVE_CXX))
CXXFLAGS            += -instances=static
endif
```

2 TEST2

Test2 covers most classes of the dyninstAPI. It is complementary to test1 in that it examines the error reporting features of the library. The mutator is implemented in `test2.C`, and the associated mutatee is implemented in `test2.mutatee.c`.

2.1 Mutator structures and important data variables

As the test2 mutator program begins executing, command line options are parsed in the *main* subroutine. Initialization is then carried out, and a single instance of the BPatch *bpatch* is created. The first four mutator test case functions (*test1*, *test2*, *test3* and *test4*) attempt to create mutatee processes under erroneous conditions. These attempts to create mutatee processes should fail, at which point, a real mutatee process is created via a call to the *mutatorMAIN* function. The mutator then proceeds with subsequent test cases on this mutatee process.

Mutator test functions are named *testXX*. The function *mutatorMAIN* is an auxiliary function which creates a mutatee process according to an executable pathname. *XX* denotes test case number.

The constant integer *MAX_TEST* denotes the total number of test cases implemented in test2.C. The boolean variable *runAllTests* defaults to true for activating all test cases. The boolean array *runTest* indicates whether to execute a specific test case depending on the command line option `run`. The boolean array *passedTest* records the passed test cases. Note that the mutatees are not designed to execute on their own. Instead, they must run under the control of the mutator.

2.2 Mutatee structures and important data variables

As the test2 mutatee program begins executing, command line options are parsed in the *main* subroutine. The mutatee proceeds by invoking the appropriate test case function *funcXX_1*.

Test case functions are named *funcXX_1*. These functions are stub functions to which the mutator can attach instrumentation code.

2.3 How to add a new test case

Adding a new test case requires the following steps:

1. Increment the *MAX_TEST* counts in both test2.C and test2.mutatee.c.
2. On the mutator side, implement a *testXX* function for the new test case and call it from the subroutine *main*.
3. On the mutatee side, implement a *funcXX_1* function and call it from the subroutine *main*.

2.4 Test case descriptions

We now examine the test cases in test2. Note that test2.1 through test2.4 attempt to create mutatee processes and test for the failure of these attempts. However, test2.5 through test2.14 operate on the same mutatee process created in the *main* subroutine.

Test2.1 (run an executable that does not exist)

In test2.1, the mutator attempts to create a mutatee process from a nonexistent executable. The mutator then examines the return value of *createProcess*. Note that this test case is skipped for the command line option `attach`.

Test2.2 (try to execute a file that is not a valid program)

In test2.2, the mutator attempts to create a mutatee process from an invalid file, such as */dev/null* (not an executable). The mutator then examines the return value of *createProcess*. Note that this test case is skipped for the command line option `attach`.

Test2.3 (attach to an invalid PID)

In test2.3, the mutator attempts to attach to an invalid *PID* number (*PID 65539*). It then inspects the *attachProcess* return value.

Test2.4 (attach to a protected PID)

In test2.4, the mutator attempts to attach to a *protected PID* number (*PID 1* - an OS kernel process that the user process can not read or write). It then inspects the *attachProcess* return value.

Test2.5 (look up nonexistent functions)

In test2.5, the mutator attempts to locate a nonexistent function, *NoSuchFunction,* in the mutatee image. It then inspects the *findFunction* return value.

Test2.6 (load a dynamically linked library from the mutatee)

In test2.6, the mutator loads a dynamically linked library from the mutatee's image. In *func6_1*, the mutatee uses *dlopen* to load the shared library *TEST_DYNAMIC_LIB* (defined in test2.h). The mutator then determines if the loading occurred by searching new symbols in the current mutatee image. This test case is not implemented on AIX or NT platforms.

Test2.7 (load a dynamically linked library from the mutator)

In test2.7, the mutator forces the load of a dynamically linked library into the mutatee's image by calling the *loadLibrary* method. It then checks if a new symbol, *TEST_DYNAMIC_LIB2* (defined in test2.h), is found in the current mutatee's image via a call to the *getModules* method. This test case is not implemented on AIX, ALPHA or NT platforms.

Test2.8 (BPatch_breakPointExpr)

There are two parts to the mutator side of this test, test2.8a and test2.8b.

Test2.8a: In test2.8a, the mutator inserts a BPatch_breakPointExpr at the entry point of the mutatee function func8_1. This test needs to be run before the mutatee process continues. For example, it can run just after process creation or attach.

Test2.8b: Following test2.8a, the mutator waits for the instrumented breakpoint to be reached. It then determines if the mutatee process correctly stops.

Test2.9 (dump core but do not terminate the mutatee)

In test2.9, the mutator calls *dumpCore* method to dump a core file from the mutatee without requiring the mutatee process to terminate its execution. The mutator looks for the creation of a core file, *mycore*, in the current directory to determine the correctness of the operations. This test case is implemented only on SPARC_SOLARIS, and IRIX platforms.

Test2.10 (dump image)

In test2.10, the mutator dumps a modified program executable file from the mutatee process, by calling *dumpImage* method. Note that only the modified executable is written. Shared libraries that have been instrumented and the current dyninst library are not written. The mutator searches for the creation of the image file, *myimage*, in the current directory to determine the correctness of the operations. This test case is not implemented on NT platforms.

Test2.11 (getDisplacedInstructions)

Test2.11 verifies the correct operation of the *getDisplacedInstructions* method, which retrieves instructions at an instrumentation point of a specified size. In the *test11* function, the mutator finds instructions at the entry point of the mutatee function *func11_1*. It inspects the contents of the

retrieved instruction buffer to determine the correctness of the operations. This test case is only implemented on AIX platforms.

Test2.12 (BPatch_point query functions)

Test2.12 verifies the correct operation of the BPatch_point query functions, *getAddress* and *usesTrap_NP*. In the *test12* function, the mutator uses these functions to retrieve information about the *func12_1* function's entry point.

Test2.13 (delete threads)

Test2.13 verifies the deletion of a mutatee process from the currently defined processes. The currently defined processes include those created using the dyninst library and those created with the UNIX fork or Windows NT spawn system calls. In the *test13* function, the mutator searches the thread list of all currently defined processes and removes the mutatee thread as requested.

Test2.14 (process management)

Test2.14 verifies the correct operation of the dyninst process management methods, *createThread*, *continueThread*, and *terminateThread*. In the *test14* function, the mutator creates a new mutatee process by calling *createProcess*. It then puts the mutatee into the running state by calling the *continueExecution* method. Finally, the mutator stops the mutatee execution by calling *terminateExecution*. Termination status is then examined.

3 TEST3

The test3 program verifies the correct operation of the dyninst classes used for manipulating code during execution. This group of classes includes BPatch and BPatch_thread. For each test case in test3, the mutator creates **multiple** mutatee processes. It instruments them and examines their behaviors. The mutator is implemented in test3.C, and the associated mutatee is implemented in test3.mutatee.c.

3.1 Mutator structures and important data variables

As the test3 program begins execution, command line options are parsed in the *main* subroutine. Initialization is then carried out, and a single instance of BPatch *bpatch* is created. The subroutine *main* proceeds by calling the test case function *mutatorTestX*s and passing along commands. Inside each of the test case functions, the mutator creates multiple mutatee processes based on the commands given. These mutatee processes run, with or without instrumentation, until completion. The mutator inspects their termination conditions to determine the correctness of operations.

Mutator test case functions are named *mutatorTestX*, where *X* denotes the test case number and *Y* is the instance number within a test case.

The constant integer *MAX_TEST* denotes the total number of test cases implemented in test3.C. The boolean variable *runAllTests* defaults to true to activate all test cases. The boolean

array *runTest* in the *main* subroutine indicates whether to execute a specific test case depending on the command line option `run`. The boolean array *passedTest* records the passed test cases.

3.2 Mutatee structures and important data variables

Based on the commands passed from the subroutine *main*, the mutator creates multiple mutatee processes and executes test code in the respective test case functions. During execution, the mutatee processes change their states by setting variable values or writing to files. The mutator observes the state changes. The mutatee processes terminate with either *exit* or *abort* calls.

The mutatee module defines a data variable, *test2ret*, for the mutator to access, verify, and manipulate. Test case functions are named *testX*, and auxiliary functions are named *funcX_Y*.

3.3 How to add a new test case

Adding a new test case requires the following steps:

1. Increment the *MAX_TEST* count in test3.C.
2. On the mutator side, implement a *mutatorTestX* function for the new test case. Call this function from *main*, passing the commands associated with a corresponding mutatee test case.
3. On the mutatee side, implement a *testX* function and call it from a *switch* statement inside the subroutine *main*.

3.4 Test case descriptions

We now examine the test cases in test3.

Test3.1 (simultaneous multiple-process management)

Test3.1 verifies the management of multiple mutatee processes, including process creation, execution, and state-monitoring. The mutator creates two mutatee processes and allows them to run simultaneously. No instrumentation is added. The mutatee processes run until termination. The mutator monitors and processes the events from each mutatee.

Test3.2 (instrument multiple processes)

Test3.2 verifies the instrumenting of multiple processes. The mutator creates two mutatee processes and inserts different code into each mutatee. The inserted code assigns a value to the global variable *test2ret*. Each mutatee then writes the *test2ret's* value to a file. After both mutatees exit, the mutator reads the files to verify that the correct code executed in each mutatee process. The first mutatee process should write a 1 to the file, and the second should write a 2. If no code is patched into the mutatees, the original value is 0xdeadbeef.

Test3.3 (sequential multiple-process management - exit)

Test3.3 verifies the management of multiple processes, including process creation, execution, and state-monitoring. The mutator creates one mutatee process and waits for it to exit. Then the mutator creates a second process and waits for it to exit. This test case differs from the test3.1 in that the two mutatee processes run one after the other.

Test3.4 (sequential multiple-process management - abort)

Test3.4 verifies the management of multiple processes, including process creation, execution, and state-monitoring. The mutator creates one mutatee process and waits for its termination. Then it creates the second mutatee process and waits for its termination. This test case differs from the test3.3 in that the mutatee processes terminate with an *abort* call rather than an *exit* call.

4 TEST4

The test4 program examines the callback facilities provided by the dyninstAPI. These facilities include callback function registration and callback function invocation. The mutator is implemented in test4.C, and the associated mutatee is implemented in test4a.mutatee.c. Note that test4.3 and test4.4 check the *exec* callback. The mutatee process makes *exec* system calls and overlays its own image with a new mutatee image. The new image is constructed based on the implementation in test4b.mutatee.c.

4.1 Mutator structures and important data variables

As the test4 mutator program begins executing. command line options are parsed in the *main* subroutine. Initialization proceeds with a call to the function *mutatorMAIN*. A single instance of BPatch *bpatch* is created, and callback functions are registered via *registerCallback* calls. Test case functions are then called to conduct their respective tests.

Mutator test functions are named *mutatorTestX*, where *X* is test case number. The constant integer *MAX_TEST* denotes the number of test cases implemented in the test4 program. The boolean variable *runAllTests* defaults to true to activate all test cases. The boolean array *runTest* indicates whether to execute a specific test case depending on the command line option `run` specified when the user executes the test4 program. Array *passedTest* records the passed test cases, array *failedTest* records the failed test cases, and integer *threadCount* records the current number of mutatee threads.

4.2 Mutatee structures and important data variables

As the test4 mutatee program begins executing, command line options are parsed in the *main* subroutine. The mutatee performs a test by calling the appropriate test case function *funcX_1*.

The mutatee defines data variables and auxiliary functions for the mutator to access, verify and manipulate. The variables are named *globalVariableXX_YY.* The auxiliary functions are named *funcX_Y* (*Y* is greater equal 1). The values of *globalVariableXX_YYs* are set in the auxiliary functions.

4.3 Test case descriptions

We now examine the test cases in test4.

Test4.1 (exit callback functions)

Test4.1 verifies the exit callback function registration and invocation. In the *mutatorTest1* function, the mutator creates a mutatee process based on the passed executable pathname and command line options. The mutatee process begins execution and sets *globalVariable1_1*'s value before terminating. This mutatee termination event triggers the execution of the installed exit callback function. The mutator verifies *globalVariable1_1*'s value in the exit callback function. This test case is not implemented on LINUX, IRIX, AIX, APLHA or NT platforms.

Test4.2 (fork callback functions)

Test4.2 verifies the fork callback function registration and invocation. In the *mutatorTest2* function, the mutator creates a mutatee process based on the passed executable pathname and command line options. Once execution has began, the mutatee forks a child process. This spawning event triggers the execution of the installed fork callback function. In the fork callback function, the mutator sets *globalVariable2_1's* value different between the parent and the child processes. This test case is not implemented on LINUX, IRIX, AIX, APLHA or NT platforms.

Test4.3 (exec callback functions)

Test4.3 verifies the exec callback function registration and invocation. In the *mutatorTest3* function, the mutator creates a mutatee process based on the passed executable pathname and command line options. Once the mutatee process begins execution, it overlays its own image with an *execvp* of *test4b.* This exec event triggers the execution of the installed exec callback function. In the exec callback function, the mutator instruments the mutatee process by inserting a call to *func3_2* at the exit point of *func3_1*. Note that both *func3_1* and *func3_2* are defined in *test4b.* This test case is not implemented on LINUX, IRIX, AIX, ALPHA or NT platforms.

Test4.4 (fork and exec callback functions)

Test4.4 verifies that multiple mutatee events trigger the execution of multiple callback functions in the mutator. In the *mutatorTest4* function, the mutator creates a mutatee process based on the passed executable pathname and command line options. Once execution has began, the process forks a child mutatee process. The child process then overlays it own image with an *execvp* of *test4b.* The two mutatee events (fork and exec) trigger the execution of the installed fork and exec callback functions.

The fork callback function operates on the parent mutatee process. It inserts a call to *func4_3* at the exit point of *func4_2*. The exec callback function operates on the child mutatee process. It inserts a call to *func4_4* at the exit point of a different *func4_2* function. Note that the functions, which the exec callback function manipulates, are defined in *test4b.* The inserted call snippets set *globalVariable4_1*'s value different between the parent and the child mutatee processes. This test case is not implemented on LINUX, IRIX, AIX, ALPHA or NT platforms.

APPENDIX A - RUNNING THE TEST CASES

This section describes how to run the dyninstAPI test programs. There are four mutator programs (**`test{1,2,3,4}`**) and currently some twenty or so mutatee programs (**`test{1,2,3,4a,4b}.mutatee_{gcc,cc,g++,CC}`**) in this test suite.

To compile the tests suite, type `make` in the appropriate platform-specific directory under `core/dyninstAPI/tests`. This should produce, depending on the platform and compilers available, sixteen to twenty programs and several shared libraries.

The test programs take the following command line options:

`-attach`
: Run the mutatee process and have the mutator attach to it with the *attachProcess* method rather than using the *createProcess* method. The `-attach` option is only applicable to test1 and test2.

`-mutatee <mutatee name>`
: Run the mutatee named `<mutatee name>` rather than the default mutatee for this test: note, however, that tests require their own particular mutatees, and that mutatees from other tests will fail. The purpose is to run test cases with versions of the mutatee compiled with the system's native C, C++ or the GNU C++ compilers in addition to the GNU C compiler. Mutatees are typically named with the name of the compiler as an extension to the mutatee name, e.g., `testN.mutatee_`**`cc`** for the "cc" compiler, and the mutatee for the GNU C++ compiler is called `testN.mutatee_`**`g++`**. (As a convenience, the name of the compiler, preceded by an underscore, can be used instead of the full mutatee name, e.g., "_gcc" would specify `testN.mutatee_gcc`).

`-n32`
: Run the 32-bit version of the mutatee test. This flag is only valid on IRIX platforms. This command line flag changes the shared libraries that are loaded to `libtest?_n32.so`, it also changes the mutatee to `test?.mutatee_??_n32`. To test 32-bit mutatees compiled with the native compiler, use `-n32` and `-mutatee test?.mutatee_cc_n32`. The order of `-n32` and `-mutatee` is important.

`-run <subtest #> <subtest #> ...`
: Only run the specific sub-tests listed. For example, to run sub-test cases 4, 7 & 9 of test2 you would enter `test2 -run 4 7 9`.

`-skip <subtest #> <subtest #> ...`
: Skip the specific sub-tests listed. For example, to skip sub-test cases 4, 7 & 9 of test2 you would enter `test2 -skip 4 7 9`. All other tests are run.

`-V`
: Print out dyninstAPI library version identification information and the file name of the dyninst runtime library used to run this test. This is useful to check that your environment is correctly setup to run dyninst-based programs.

`-verbose`
: Enable detailed debugging output from the dyninst test programs (mutators and mutatees) themselves. This is useful when trying to track down the reason that one (or more) of the test cases failed.

`-v+`

Enable the printing of dyninst warning-level messages (`BPatchWarning`) to standard output. This is useful for debugging the test cases.

`-v++`

Enable the printing of dyninst information- and warning-level messages via the error reporting callback function (`BPatchWarning` and `BPatchInfo`). These options are useful for debugging the test cases.

Some test cases are not implemented on all the platforms (due to OS restrictions or missing features). If a test is not run on a specific platform, the message `"Skipped test #XX"` will be displayed. If any of the tests produces a line of the form `"**Failed test #XX"` there is something wrong with the version of the API or its installation. Each test should still produce a message of the form `"Passed test #XXX"`, and a message at the end indicating that either all tests were passed, or all requested tests were passed (if the `-run` option is used).

Note: test2 produces a few lines that look like error messages since it is testing the error reporting features of the API (e.g., file not found). Check for the `"All tests passed"` message at the end to confirm correct execution.