Self determination: A comprehensive strategy for making automated tests more effective and efficient

Kesina Baral, Jeff Offutt, and Fiza Mulla
Department of Computer Science, George Mason University
{kbaral4,offutt}@gmu.edu, 818852@lcps.org

***Abstract*—A significant change in software development over the last decade has been the growth of test automation. Most software organizations automate as many tests as possible, which not only saves time and money, but also increases reproducibility and reduces errors during testing. However, as software evolves over time, so must the test suites. For each software change, each test falls into one of four categories: (1) it needs to rerun as is, (2) it does not need to rerun, (3) it needs to change and rerun, (4) it should be deleted. This test management is currently done by hand, leading to shortcuts such as always running all tests (wasteful and expensive), deleting valuable tests that should be fixed, and not deleting unneeded tests. Over time, the test suite becomes larger and more expensive to run while also becoming steadily less effective. This project introduces a novel solution to this problem by giving individual tests the ability to self-manage through self-awareness and self-determination. Each test will encode its purpose (its test requirement), can discover what changed in the software, and then decide whether to run, not run, be changed, or self-delete. We are developing techniques and algorithms to compare syntactically two versions of the same program (previous and new) to identify differences. Tests can then check to see whether their purpose is affected by the change, and decide what to do. We have developed preliminary test framework infrastructure to be used with tests that satisfy edge coverage, based on the control flow graph. We have carried out empirical studies on open-source software to evaluate the accuracy of tests' decisions and the cost of execution. Results are encouraging, indicating strong accuracy and reasonable cost.**

***Index Terms*—Test management, Test automation, Regression testing**

I. Introduction

This paper presents a novel, holistic solution to the problem of test suite management. As software evolves, its associated automated tests must also evolve. For each change, existing tests fall into one of four categories: (1) it needs to rerun to verify the change, (2) it does not need to be rerun, (3) it needs to be changed to adapt to new syntax or behavior, or (4) it is no longer needed and can be deleted. We collectively call these activities *test suite management.* These topics have been previously studied and useful algorithms and techniques have been proposed, [1] [2] [3] [4] [5] yet much of the research is piecemeal and few ideas have been adopted in practice. Crucially, all prior work depended on human testers to do much of the work; part of the novelty of this research is to move the burden of decision-making and acting from the human to individual tests.

The result is that software developers currently manage their test suites by hand, with all the expected difficulties. Sometimes all tests are rerun for every change. This is fine for small programs with small test suites, but does not scale to programs with millions of lines of code and thousands or tens of thousands of tests. Sometimes the decision of which tests to rerun is made intuitively, with the expected loss of fidelity and effectiveness when some tests that need to rerun are skipped and others that should be rerun are not skipped. Often tests that need to be changed to reflect changes in the software are either ignored, if they still compile and execute to completion, or simply deleted if they do not. This leads to tests that are no longer useful or the loss of valuable tests. And finally, when test management is done by hand, few tests are deleted when no longer needed–they simply stay around being rerun for no good reason for years.

The term *test bloat* is used for test suites that continue to grow and contain increasing numbers of useless and unnecessary tests. Test bloat wastes valuable resources. Computer resources are wasted when useless tests are run unnecessarily. Worse, human resources are wasted when tests fail because of the test, not the software, and when humans spend increasing amounts of time poring through voluminous test results that are full of noise, looking for the ever-shrinking signal. When coupled with flaky tests [6] [7] and blind tests [8], these problems lead to systemic and industry-wide waste, in turn making software more expensive and less reliable.

This paper presents a novel self-management approach to maintaining test suites that is inspired by two conference presentations [9] [10]. We propose a practical and comprehensive strategy based on having individual tests self-manage. We say that a test is *self-aware* if it has access to its purpose in an actionable way. For example, if its purpose is to cover a specific edge in a control flow graph (its test requirement), then the test must encode the edge or edges that it covers and be able to access information as to whether that edge is still present after the software changes. Or, if the test's purpose is to verify a specific functional or nonfunctional requirement, it needs to encode the requirement it verifies and be able to access information about whether the requirement is affected by the software's change. This information is typically stored in "soft" form, in natural language documentation such as comments, notes on paper, or even lost completely. For example, software engineers often create tests for a specific purpose but then never document that purpose.

Further, we say that a test has *self-determination* if it has the ability to decide what should happen after the software

changes. That is, depending on its purpose and the change, should the test rerun, not run, be changed, or be deleted? At present time, we expect that a test that needs to be changed to reflect the software's change must be changed by a human. In the future, we plan to investigate the possibility of using automatic program repair techniques to "repair" tests that no longer compile or run.

We discuss background and challenges of test suite management in section II. Section III describes the proposed framework for test self management. Our empirical study is discussed in section IV, followed by results in section V. Section VI presents threats to validity, section VII presents related work, and section VIII concludes the paper.

II. Background and Challenges of Test Suite Management

Every program has a suite of tests that are designed during development to verify that the program's behavior is as expected. As the program evolves, some tests are added, some are modified to reflect the program's changes, some are no longer needed and deleted, and others are unchanged. Running all the tests in a test suite after every program change costs time and effort, and is prohibitively expensive for software at scale. Thus, testers try to only run tests that may behave differently on the modified program.

A. Managing tests by hand

Automated tests encapsulate test inputs and expected results, allowing them to run automatically and report results. However managing tests as software evolves is almost entirely done by hand. Each test was designed for some purpose, but that purpose is seldom recorded in a machine-readable form, putting the responsibility of investigating, evaluating, understanding, and remembering the purpose of tests on the developers. Although researchers have published algorithms for evolving and managing test suites [1] [2], the algorithms are not available in useful tools that practical testers can integrate into their typical workflow. Thus, they are left with the burdens of investigating the cause of a failing test and deciding which tests need to be rerun after the software changes.

For every software change, the developer has to: (1) understand the program's logic and flow, (2) understand why each test was created, (3) evaluate all tests to see if the change affects the test's purpose, and (4) decide which tests to rerun, modify, and discard. Performing these tasks by hand is slow, error-prone, and relies on tester expertise. Despite numerous research advances, the field has not successfully deployed tools for developers. Therefore, this project is creating novel infrastructure that can allow tests to perform the above four tasks and significantly reduce developers' burden.

B. Information needed

A *smart* test can decide if it needs to be run, ignored, modified, or discarded after the software is changed. To do so, it needs to answer five questions:

1) What are the test requirements of the program?
2) What does the test currently cover?
3) Does the test's current coverage match the test requirement?
4) What has changed in the program?
5) Do the program changes affect the test's purpose? If so, how?

This information must be available in a form that can be processed by software.

C. Central management vs. test responsibility

Automated tests are widely used in the software industry, and usually managed by hand. Because modifying and removing out-of-date tests is time-consuming and complicated, test suites are often not really managed, they merely grow. This leads to *test suite bloat*, where the test suite grows unchecked and becomes increasingly harder to manage. Bloated test suites include tests that fail because they no longer match the software, which in turn leads to testers not noticing tests that truly fail.

Researchers have proposed techniques to centrally manage test suites. These include algorithms to prioritize tests for ordered execution, algorithms to remove tests that are no longer needed, and procedures to identify code elements that are not currently covered [1] [11] [12] [13] [14] [15] [16] [17]. However, central management is expensive and cumbersome, and tools that are useful in practice are not available.

D. Test self-management

To move from managing test suites centrally by hand to test self-management, we need tests to be self-aware and to have self-determination. A self-aware test must know its traceability information such as what requirements or code elements it covers (its *purpose*), and be able to discover what has changed in requirements or program. This information is typically kept in inaccessible documentation, not kept at all, or at best, re-computed when needed. Self-determination means that the test can check the available information and decide, after the software changes, whether the test needs to run as is, not run, be changed, or be deleted. We call tests with self-awareness and self-determination *smart*, as they have the ability to manage themselves.

III. A Framework for Test Self-Management

This section describes a test self-management framework[1]. The framework consists of five automated, sequential, steps to answer the five questions listed in II-B. We first give a high-level description in section III-A, followed by a more algorithmic-level description in section III-B.

A. The test framework's five step process

Figure 1 illustrates the five steps, and we discuss each below.

Step 1) *Run the tests to track program statements executed:* This lets us track which statements were covered by each test,

[1]The source code for this project is available at https://github.com/SmartTests/smartTest

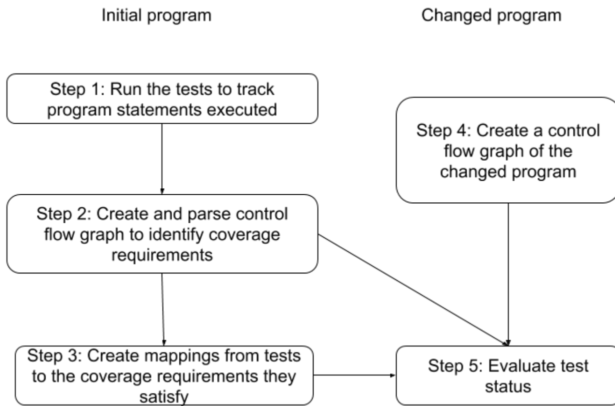

Fig. 1. Test self-management framework

allowing us to identify each tests' purpose and evaluate its status, as discussed in steps 3 and 5.

To track statement coverage, we modify the *.class* file using bytecode injection at runtime to inject tracing code immediately before the class is loaded and run. We use the bytecode manipulation framework ASM [18], which provides common bytecode transformation and analysis algorithms to build code analysis tools.

Step 2) *Create and parse the control flow graph to identify coverage requirements:* We compare the control flow graph (CFG) of a program before and after program evolution to detect precisely how the program was changed. This comparison lets automated tests be aware of program evolution.

We use a cross-platform tool, Progex [19], to create control flow graphs. Progex reads program source code, then generates the control flow graph and exports it into a file format for graphs such as DOT [20], GML, and JSON [21] (we use DOT). Following is an example of a Progex-generated CFG for SourceClass.java in DOT file format:

```
digraph SourceClassCFG {
// graph-vertices
v1 [label="17: SourceClass(int qtyOnHand)"];
v2 [label="18: this.qtyOnHand = qtyOnHand"];

// graph-edges
v1 -> v2;
}
```

Each node in the CFG, v1 and v2 in the example, represents a single line of code. Each node has the line number and the Java statement it represents as its label. Connections between nodes are under `graph-edges`. In this example, the graph has an edge from node v1 to node v2.

Step 3) *Create mappings from tests to the coverage requirements they satisfy*: Coverage criteria provide test requirements, and in turn, each test has a specific purpose—to satisfy one or more test requirements.

Although the concepts behind self-determining tests can be applied to any type of test requirement, whether functional or non-functional, our empirical focused on one test coverage criterion, edge coverage (ECC). Many available tools measure ECC and it is commonly used in industry. The test requirements for ECC are the edges in a graph, that is, a pair of Java statements.

We use an automated script to identify the ECC requirements. The script uses the control flow graph generated by Progex [19] to create a list of edges that must be traversed. Based on the statements a test executed (from step 1), we map the test to the requirements (pairs of Java statements) it satisfied to document the tests' purpose. This mapping is used as the program evolves.

We use another automated script to create the mapping between a test and the requirements it satisfies. The script uses the test requirements identified for a criterion and the test coverage information from step 1 and produces a mapping in JSON format. JSON is a standard, lightweight, data format that uses human readable text to store data in a map structure [21]. Following is a sample JSON mapping.

```
[
  {
   "info": "27-04-2020_17_01_08"
  },
  {
   "requirements":
        {"TR1": "17,18" }
  },
  {
   "testCoverage": {
    "testAdd":
      ["TR1"]
                  }
  }
]
```

The `info` line gives the creation date, and `requirements` contain the test requirements, in this case, to cover edge (17, 18). Thus, this JSON script indicates that the test method *testAdd*() satisfies test requirement TR1.

Step 4) *Create a control flow graph of the changed program:* This step lets us compare the changed CFG to the previous CFG to identify changes in the program. The detected change can then be analyzed to see if it affects the test requirements, and in turn, the purpose of the tests.

Step 5) *Evaluate test status:* For a test to decide what to do after a change, the test needs to know how the program's source code was changed. In our system, this decision is based on the difference between the old and new CFGs. If the change is more than an addition of white space we say the change is *substantial*, and the effects of the change on tests need to be evaluated. Once the test knows its purpose and is aware of the change, it can then compare the two and decide if the change affects its purpose. The test evaluates the change with respect to its purpose to decide. If the test's purpose is affected, it evaluates the extent of the change and acts accordingly.

B. Details of the test framework process

The previous steps were described at a high level, in terms of goals and results. Now we discuss how these steps are carried out. We implemented all automation using Python and bash scripts.

1) Detecting software changes: We compare the control flow graph of the modified program with the prior CFG to detect changes in the code. This includes comparing every statement in the old and new versions in case the change affected the line number or node in the CFG. For efficiency, we identify individual program methods and compare statements within a method. We use the SequenceMatcher [22] class of the diffLibPython [23] library to make this comparison.

SequenceMatcher compares pairs of sequences of any data type. The ratio method of SequenceMatcher returns a measure of two sequences' similarity as a float in the range [0, 1], where 1 represents an exact match and 0 represents no match. This ratio is calculated as: $ratio = 2.0 * \frac{M}{T}$, where M is the number of elements that match and T is the total number of elements in both sequences.

We provide two statements, representing a node in the two control flow graphs, as input to the ratio method. The sequence is simply the characters in the statement, for example, the statement "`x = a*b;`" is a sequence of 8 characters. A higher ratio means more similarity. Low similarity could mean one of two things. First, two different unmodified statements could be being compared, for example, the statement on line 1 could be being compared with line 10. Second, a statement is compared with its modified version in the modified program. The SequenceMatcher documentation [23] says to interpret a value over 0.6 to mean that the sequences are a close match. We follow this recommendation to make the following inferences:

i. If the ratio between two Java statements is 1, they are identical and no change has been made. We categorize these as *perfect match statements*.
ii. If the ratio between two Java statements is 0.6 or above and below 1, we categorize them as *close match statements*.
iii. If the ratio is below 0.6, they have low similarity and need to be evaluated further.

2) Analysis of the program evolution: We consider four types of program evolution: (a) no substantial change from the previous program version, (b) statements in the previous program were modified, (c) statements in the previous program were removed, and (d) statements were added in the new version. The previous step (detecting software changes) creates two categories of statements: perfect match statements and close match statements. These categories need to be further analyzed to understand what kind of program change they represent:

i. *Was there a substantial change in the code?*
   If all the statements from the previous program are present in the changed program and were categorized as perfect match statements, no change was made to the program.
ii. *Were statements in the initial program modified?*
   If statements from the previous program were categorized as close match, the statements were modified. This means the program was modified.
iii. *Were statements removed from the initial program?*
   If statements from original program were not categorized as either perfect match or close match, either the statement from the previous program was significantly changed or it was removed.
iv. *Were statements added to the modified program?*
   If statements from the modified program were neither a close match nor a perfect match, they were added to the modified program.

3) Impact analysis of the program change on test: We categorize program changes into four possible types.

i. *No changes to the previous program:*
   No test requirements were affected and no tests need to be run or modified.
ii. *Some statements were modified in the previous program:*
   We assume edge coverage is being used, so a change to a statement implies a change to all edges that statement appears in. Thus, all tests that cover those edges need to be rerun, and some may need to be modified.
iii. *Statements removed from the previous program:*
   If a statement is removed from the previous program, then any edge it appeared in are no longer present in the CFG, and those test requirements are gone. Tests that covered that edge may no longer be needed. If the change removes only a single test requirement (one edge), then a test that satisfies that test requirement needs to be rerun and possibly modified. If a change removes several test requirements, for example, deleting a complete method, then tests that satisfy those test requirements are no longer needed and can be discarded.
iv. *Statements added to the previous program:*
   If statements were added, one or more new test requirements were created and existing test requirements might be affected. This means all the tests that satisfy the affected test requirements need to be rerun, and new tests may be needed.

We automate these three steps in a Python script. The script takes the control flow graph of the original and modified program, and uses SequenceMatcher to identify the statement level changes. Then our script uses the mapping between tests and test requirements from step 3 and the similarity ratio from SequenceMatcher, to let the tests to decide what action to take. If the test needs to be changed, it informs the (human) developer, including a message such as:

```
code in original program: if (qty < 0)
modified version: if (qty > 0)
```

If multiple tests need to be rerun, we compare requirements satisfied by the affected tests and if there is an exact match of

```
RUNNING TEST testString
RUNNING TEST testQuadratic
RUNNING TEST testSerial
RUNNING TEST testQuintic
==============================
SOURCE CODE STATUS
==============================
Line '70' deleted in new version of PolynomialFunction
==============================
TEST CODE STATUS
==============================
==============================
Affected Requirements
==============================
TR2
TR3
==============================
Affected Tests
==============================
Rerun test:------> testAddition to check requirement  {'TR3', 'TR2'}
Rerun test:------> testConstants to check requirement  {'TR3', 'TR2'}
Rerun test:------> testLinear to check requirement  {'TR3', 'TR2'}
Rerun test:------> testMath341 to check requirement  {'TR3', 'TR2'}
Rerun test:------> testMultiplication to check requirement  {'TR3', 'TR2'}
Rerun test:------> testQuadratic to check requirement  {'TR3', 'TR2'}
Rerun test:------> testQuintic to check requirement  {'TR3', 'TR2'}
Rerun test:------> testSerial to check requirement  {'TR3', 'TR2'}
Rerun test:------> testString to check requirement  {'TR3', 'TR2'}
Rerun test:------> testSubtraction to check requirement  {'TR3', 'TR2'}
Rerun test:------> testfirstDerivativeComparison to check requirement  {'TR3', 'TR2'}
==============================
Minimal Tests Re-ran
==============================
Reran test:------> testString
Reran test:------> testQuadratic
Reran test:------> testSerial
Reran test:------> testQuintic
```

Fig. 2. Test self-management result

satisfied requirements, we run only one of the affected tests. This is a relatively simple approach that could be replaced or augmented by one of the more sophisticated approaches discussed in Section VII.

We use Progex [19] to create a control flow graph of the modified version of the program. We then use the original and modified program CFGs to evaluate the test status, as described in step 5.

C. Scope of the framework

We developed our test framework for Java projects built in Maven [24] and unit tests written in JUnit. We used the ASM [18] framework to perform bytecode manipulation. We used Major [25] mutation framework to generate mutants, and use the mutants as a substitute for real faults. This is a common technique that has been supported by research [26]. We used all the mutation operators provided by the tool. We used Progex [19] to generate CFGs of the source code. The open source projects we used did not have test plans that identified individual test goals. As a proxy, we used the Edge Coverage Criterion (ECC) to specify coverage requirements of the program, and measured the available tests in terms of edges covered. This resulted in test purposes that are valid for our experiment. We used Python scripts to automate the steps in our framework. To flag changes in the source code, we use the SequenceMatcher [22] class of the diffLibPython [23].

Although the tools and environment used in this project limit the scope of this framework, most limitations could be overcome by finding or building more robust tools.

IV. An empirical study

We carried out two studies to analyze our strategy to make automated tests more efficient. We start by asking two research questions:

RQ1. How much time does the analysis use?

RQ2. How accurate are the analysis results from the framework?

A. Methodology

We collected two sets of data from four open source code repositories. All projects had JUnit tests. We first collected requirements for statement coverage and edge coverage, and then measured the coverage the tests achieved. We used mutation analysis to simulate changes to the source code, then compared the mutated (*new*) versions of the program to the original version. This gave each test the information needed to decide what action to take when the software was changed.

1) Study 1: Preliminary Analysis: We performed a preliminary, small scale study to evaluate how comprehensive and accurate our strategy is.

i) Subject selection: The study's goal was to evaluate the reliability of our approach and the correctness of the automated steps. We used four classes from the Apache Commons Math4 project. All the classes have between 100 to 1000 lines of code and came with between 7 and 74 tests. Table I gives statistics from these classes.

ii) Data collection: We used automated scripts to perform steps 1 (run the tests), 2 (create the CFG), and 3 (create mappings from tests to test requirements) of the test management framework process. The scripts collected statement coverage and edge coverage requirements, and created mappings between tests and which test requirements they satisfied. We simulated changes to the software under test by using mutation to modify the original program. We used the PiTest mutation testing tool [27] to generate mutants of each Java class. The mutation operators used are shown in Table II. To allow us to perform manual analysis in a reasonable time frame, we randomly selected 25 killed and 25 surviving mutants for each Java class. We placed these 200 mutations ((25+25)$\times$4) into the code to create 200 versions of the classes. We created the CFG for each "new" version, and compared them with the CFG of the initial program to identify the changes to the program. The change was then analyzed to check if the test requirements were affected, and in turn, the tests. We then analyzed these changes to identify which action the test need to take in response to the change, which was then presented to the tester on the console.

iii) Framework result verification: We then verified the results for each test and each change by hand, allowing us to identify corner cases that needed to be addressed. We found that our strategy correctly identified which lines were changed and how, and which statements were deleted. In turn, the framework was able to identify test requirements that were affected, and which tests those changes affected. We were also able to correctly flag unsatisfied requirements and failing tests.

We also verified the accuracy of decisions to rerun or delete tests. The results of this verification are presented in Section V; all decisions made in this study were correct.

TABLE I
PRELIMINARY STUDY: SUBJECT PROGRAM DETAILS

Class names	SLOC	#tests	#mutants	EC requirements
DerivativeStructure	637	74	50	283
FiniteDifferences-Differentiator	169	16	50	43
SparseGradientFunction	555	70	50	257
DSCompiler	885	7	50	436
Total	**2246**	**167**	**200**	**1019**

TABLE II
PRELIMINARY STUDY: PIT MUTATORS USED

Mutator	Example
Boolean false return	return a $\longmapsto$ return false
Boolean true return	return a $\longmapsto$ return true
Conditional boundary mutators	a $<$ b $\longmapsto$ a $\leq$ b
Empty returns	java.lang.String $\longmapsto$ ""
Increments	i++ $\longmapsto$ i$--$
Invert negatives	return -i $\longmapsto$ return i
Math	+ $\longmapsto$ -
Negate Conditionals	!= $\longmapsto$ ==, $<$ $\longmapsto$ $\geq$
Null return	return a $\longmapsto$ return null
Primitive returns	replaces primitive data type return values with 0
Void method calls	removes calls to void methods

2) Study 2: Apache Projects: We used three Apache projects, Commons Math3, Commons CSV, and Apache Commons CLI, for our second study. The goal of this study was to check the accuracy and comprehensiveness of the framework result, and the time required for the analysis, in a larger project.

i) Subject selection: To scale up the size of our study, we used the Major mutation testing tool [25] to automate the creation of new (mutated) versions of the software. We switched to Major because it has the ability to export mutated program versions. This change also caused us to switch to Apache Commons Math3 because Major was not compatible with some parts of Math4. Details of our subject program are shown in Table III.

ii) Data collection: As in study 1, we used automated scripts to perform steps 1 (run the tests), 2 (create the CFG), and 3 (create mappings from tests to test requirements). We then used Major to generate the new versions of the software, resulting in 4829 changes to 108 classes. The mutation operators used in this study are shown in Table IV. We then created CFGs for each new version and compared them with the CFGs of the original versions. As before, we then analyzed the changes to determine which test requirements were affected, then identified the appropriate action for each test. We recorded the system time for the analysis to answer RQ1. The results of the analysis were shown on the console for the tester to see.

iii) Framework result verification: We used an automated script to verify results on the 4829 mutants. The script compared the result to the expected result based on the initial program version and the new program version. We found several discrepancies between the result reported by the

TABLE III
SECOND STUDY: SUBJECT PROGRAM DETAILS

Package names	# classes	SLOC	# tests	# mutations	# EC requirements
math4 project (9 packages)					
differentiation	5	1708	100	231	971
function	8	1173	138	353	288
integration	8	1116	26	380	264
interpolation	14	1767	85	616	862
polynomials	5	741	41	250	419
solvers	14	1650	52	472	701
complex	6	976	159	229	429
dfp	4	2662	36	200	2019
distribution	27	3560	153	1331	1334
CSV (1 package)	6	1432	205	251	804
CLI (1 package)	11	1962	268	516	873
Total	**108**	**18,747**	**1263**	**4829**	**8964**

TABLE IV
SECOND STUDY: MAJOR MUTATION OPERATORS USED FOR STUDY 2

Mutation operator	Example
AOR (Arithmetic Operator Replacement)	a + b $\longmapsto$ a - b
LOR (Logical Operator Replacement)	a $\wedge$ b $\longmapsto$ a $\mid$ b
COR (Conditional Operator Replacement)	a $\|$ b $\longmapsto$ a && b
ROR (Relational Operator Replacement)	a == b $\longmapsto$a $\geq$ b
SOR (Shift Operator Replacement)	a $\gg$ b $\longmapsto$a $\ll$ b
ORU (Operator Replacement Unary)	-a $\longmapsto$ a
EVR (Expression Value Replacement)	return a $\longmapsto$return 0
LVR (Literal Value Replacement)	0 $\longmapsto$1, true $\longmapsto$ false
STD (STatement Deletion)	return a $\longmapsto < no - op >$

framework and expected results, as reported in Table VI.

V. OBSERVATIONS AND RESULTS

This section presents results from the two studies whose results are shown in Tables V and VI, and our observations from those results. In the preliminary study on four classes from Apache Commons Math4, we checked the test and program status for 50 changes to each Java class. Table V shows that all program and status decisions in study 1 were correct.

Our second study used 4829 changes to 108 classes, as shown in Table VI. We used a maximum of 50 changes per class, drawn randomly from the total number of changes (some classes had fewer than 50 changes). This analysis required a total of 553 minutes, for an average of just over 5 minutes per class. The time required to identify a program change, identify the test status, and determine the appropriate action averaged 6.87 seconds per change. The framework's recommended action was accurate 94.31% of the time.

TABLE V
RESULT: STUDY 1

Packages	Mutants evaluated	Incorrect results
DerivativeStructure	50	None
FiniteDifferencesDifferentiator	50	None
SparseGradientFunction	50	None
DSCompiler	50	None
Total	**200**	**None**

We analyzed each inaccurate result and categorized the reasons into three types.

a) The program change was not identified correctly. Our framework only generates control flow graphs for the methods in the class. Therefore, code changes that do not appear in methods, such as changes to statements that declare or instantiate class attributes outside a method, were not identified. This is a limitation of the tool, rather than a problem with the concepts.

This could be solved by adding such information to the control flow graph or by using a different abstraction that represents that information. In our study, 36 of the 258 (14%) inaccurate results fall in this category, that is, 0.7% of the total number of results.

b) The CFG was not created correctly. Our CFG generation tool, Progex [19], was not able to create control flow graphs for some methods in some classes. This happened when the methods used recently added language features that the CFG tool did not recognize. In our study, 79 of the 258 (31%) inaccurate results fall in this category, that is, 1.6% of the total number of results.

c) The line reported by the framework as changed did not match the actual line changed. The control flow graph considered one statement as one node regardless of the statement length, such as when a program statement spanned multiple lines. For example, the following statement spanned four lines in the source code file:

```
1 Logistic.value(param[1] - x, param[0],
2                         param[2], param[3],
3                         param[4],
4                         param[5]);
```

Fig. 3. Example four-line statement

Our control flow graph generator placed all four lines into a single node, and identified the entire statement as line 1. However, if a change was made on physical source code line 2 (changing param[3] to param[i] for example), our tool would match that to line 2 in the CFG, which does not exist in the generated CFG, since lines 2, 3, 4 from the actual source code are part of line 1 in the CFG. In our study, 143 of the 258 (55%) inaccurate results fall in this category, that is, 2.96% of the total number of results.

In summary, the incorrect results were due to relatively minor issues in our tooling, in particular, the CFG generator, not due to conceptual or practical problems with the problem solution.

Thus our answer to RQ1 is that the time taken by the framework to analyze each change averaged 6.87 seconds. This time can be further decreased using code optimization techniques such as reducing the number of file input output operations and optimizing the database for information storage. Our answer to RQ2 is that the framework was accurate 94.31% of the time. Since the incorrect results were due to incorrect or incomplete representation of the program, we are confident the accuracy can be improved even further.

TABLE VI
RESULT: FREQUENCY OF TEST STATUS MATCH AND TIME TAKEN FOR STUDY 2

Projects	Packages	Changes evaluated	Time (mins)	Result match	Result mismatch	% of test status matches	% of test status mismatches
math4	differentiation	231	31	226	5	97.83%	2.16%
	function	353	39	329	24	93.20%	6.80%
	integration	380	35	348	32	91.58%	8.42%
	interpolation	616	91	601	28	94.45%	4.54%
	polynomials	205	24	157	48	76.59%	23.41%
	solvers	472	33	468	4	99.15%	0.85%
	complex	229	75	220	9	96.07%	3.93%
	dfp	200	8	199	1	99.50%	0.50%
	distribution	1331	35	1289	42	96.84%	3.16%
CSV		251	53	246	5	98.00%	2.00%
CLI		516	129	456	60	88.37%	11.63%
Total		**4829**	**553**	**4539**	**258**	**94.31%**	**5.68%**

VI. THREATS TO VALIDITY

We used an open source control flow graph generator tool from github, Progex [19]. The latest release of the tool was in 2019 and the tool did not appear to have been validated. Therefore, problems with the tool could lead to threats to our study. Indeed, we determined that most incorrect decisions the test framework were directly attributable to shortcomings of Progex.

Another potential threat is that all of our classes were obtained from a small number of open source projects. However, the Apache Commons project is large and diverse, with classes that perform many complicated computations.

The code changes were modeled through a fairly simple program abstraction tool, the control flow graph. Result accuracy could be higher if we included data flow information, or a more sophisticated abstraction such as an interprocedural CFG [28] or an interclass graph [29] to capture more details.

Finally, our model of program changes was fairly simple—single order mutants. Although this allowed us to create and analyze thousands of program changes, a more sophisticated model of program changes, or actual changes made to software as documented in changelogs, could provide different insights to this approach to test management.

VII. RELATED WORK

The literature contains three general techniques to address problems with test suite growth: test suite reduction or minimization, test selection, and test prioritization.

A survey by Yoo and Harman discusses these approaches in detail [30]. Our paper uses test case minimization and selection as intermediate steps, but does not specifically introduce new minimization or selection techniques. We also do not currently address test case prioritization.

Test suite reduction approaches focus on reducing the size of test suites to lower the maintenance cost of large test suite. Chen and Lau proposed minimizing test suites by selecting an essential set of tests that cover requirements that no other tests cover, followed by a greedy algorithm to select more tests [31]. Ammann et al. proposed removing redundant tests until a minimal test set is obtained [32]. The primary focus of our approach is to enable the automated tests to maintain themselves to reduce maintenance costs for developers. We also perform test suite reduction whenever a test requirement becomes invalid due to software evolution. Vaysburg et al. performed dependency analysis of Extended Finite State Machines to minimize test suite [33]. In our work, we analyze control flow graphs to detect the impact of software evolution on tests, although our general approach could work just as well with other models and other techniques. A common concern of test suite reduction techniques is to avoid removing a fault-revealing test. To handle this issue, our framework does not permanently remove tests whose requirements are no longer valid; rather, we comment it out and notify the developer.

Test case selection focuses on selecting a subset of test cases from the test suite after the software under test changes. Various approaches have been explored to perform test case selection using techniques such as data flow analysis [34] [35] [3], symbolic execution [4], dynamic slicing [36], CFG graph-walking [37] [38] [39] [40], textual difference in source code [41] [42], and modification detection [5]. Rothermal and Harrold suggested selecting test cases based on walking control flow graphs [40]. Chen et al. used a modification-based technique to identify test cases that are affected by modification in program entities [5]. In our approach, we compare control flow graphs of two software versions to detect any changes, and select tests that execute the modified statement. We select all test cases that execute the modified statement but only run tests that satisfied a unique set of test requirements.

Test case prioritization is an approach to find an ordering of test cases for execution to get maximum benefit with minimal effort [11] [16] [17] [30] [43]. We do not address test case prioritization in our framework.

VIII. CONCLUSIONS AND THE FUTURE

This paper presents results from a novel, holistic, solution to the problem of managing and evolving automated test suites. As software evolves over time, the suite of automated tests must also evolve. For every change, some tests need to be rerun to verify the change, while other tests are not affected by the change and thus do not need to be run. Further, some tests need to be modified to still run on the modified software, others are

no longer relevant and can be ignored, and some new tests need to be created to verify added or modified functionality. These decisions are generally called test management, and despite years of research, test management is still largely done by hand.

Poor test management leads to unchecked growth in the number of tests (test bloat), tests that are no longer correct with respect to the software under test, flaky tests [6] [7], and blind tests [8]. Over time, testing becomes more expensive and less effective.

The novel approach presented in this paper is to give each test the ability to manage itself. To do that, tests need two things. They need to be *self-aware*, that is, know why they exist. Second, tests need to have *self-determination*, that is, be able to choose whether to run, not run, be changed, or be deleted. We have developed a process to support self-managed tests, and a framework that incorporates algorithms and software to automate the self-management approach.

The paper presents results from an empirical evaluation on open source software, which resulted in two broad findings. First, the *time* needed to create, store, process, and use the information that tests need to manage themselves was quite reasonable. Second, the *accuracy*, in terms of whether tests made correct decisions, was quite high, with the primary limitation stemming from the capabilities of the control flow graph generator that we used.

A. Future work

In the future, we hope to extend these ideas in several ways. As currently configured, when a test discovers that it needs to change to accommodate changes in the software under test, it alerts the test for manual intervention. We believe that automatic repair [44] techniques could be used to automatically update some tests. The scope of change is smaller than for general software, so the potential for success may be higher for automated tests. We also plan to investigate and address scalability to ensure this approach works for larger programs, and applicability, to ensure this approach works for real programs and can be used in practice.

More broadly, our demonstration implementation works in the context of test structural requirements derived from control flow graphs. We hope to apply this approach to other types of test requirements, such as those based on functional or non-functional software requirements. We also plan to evaluate the framework with real software faults, and to further investigate how test requirements can be better captured and maintained as the program evolves. Finally, since our research is attempting to automate work that is currently done mostly by hand, we hope to make a direct measure of improvement by comparing results from our process with results from a strictly human process.

B. A manifesto for research in test automation

For many years, software testing researchers and educators viewed the automatic execution of tests as simple programming problems that did not pose sufficient research challenges or complexity to engage us. We focused on automating the generation of test values, using criteria, algorithms, and approximation procedures, and to a lesser extent, on the problem of automatic generation of oracles and test management issues. But recently, researchers have discovered the deep complexity and troubling problems that arise when automating the execution of tests [45], including the contents of test oracles [46], [47], flaky tests [6] [7] [48], and blind tests [8]. We were wrong!

The authors of this paper firmly believe that the community needs to greatly expand our research into test automation, including and especially the automated execution of tests, and the automated creation of executable tests [49]. This focus needs to be vitalized by funding from industry and government, journal special issues, and topics and tracks within top conferences. Our ultimate vision is ambitious, but fundamentally achievable. We envision testing becoming tightly integrated into compilers and IDEs. After a compiler uses **syntactic validation** (parsing) to correct all syntactic mistakes, leading to an executable version of the software component, we envision the next step to be **semantic validation** (testing). The IDE will automatically generate a collection of tests, automatically run those tests on the software component, and present a report to the developer with the summary that out of N tests, $n1$ crashed, $n2$ appear to cause incorrect behavior, $n3$ have behavior that may or may not be wrong, and $n4$ appear to be result in correct behavior. The developer will then proceed to analyze and resolve each questionable test, in the same way that we currently resolve errors and warnings from parsers. The IDE will also use automatic program repair [44] techniques to suggest specific code corrections. Naturally, it will be impossible to completely replace all human test activities, but smart tests and integration of compiling and testing has enormous potential to automate tedious and mechanical testing tasks, freeing up human testers to focus on more interesting and challenging problems.

References

[1] S. Eldh, J. Brandt, M. Street, H. Hansson, and S. Punnekkat, "Towards fully automated test management for large complex systems," in *International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2010, pp. 412–420.

[2] E. Enoiu and M. Frasheri, "Test agents: The next generation of test cases," in *Workshop on NEXt level of Test Automation (NEXTA)*, 2019, pp. 305–308.

[3] M. J. Harrold and M. L. Soffa, "An incremental approach to unit testing during maintenance," in *International Conference on Software Maintenance*. Los Alamitos, CA, USA: IEEE Computer Society, October 1988, pp. 362–367.

[4] S.-S. Yau and Z. Kishimoto, "Method for revalidating modified programs in the maintenance phase." in *International Computer Software & Applications Conference*. IEEE, 1987, pp. 272–277.

[5] Y.-F. Chen, D. Rosenblum, and K.-P. Vo, "Testtube: A system for selective regression testing," in *International Conference on Software Engineering (ICSE)*. IEEE, 1994, pp. 211–220.

[6] M. Fowler, "Eradicating non-determinism in tests," Online, 2011, https://martinfowler.com/articles/nonDeterminism.html, last accessed October 2019.

[7] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov, "An empirical analysis of flaky tests," in *Foundations of Software Engineering*, Hong Kong, China, November 2014, pp. 643–653.

[8] K. Baral and J. Offutt, “An empirical analysis of blind tests,” in *International Conference on Software Testing, Validation and Verification (ICST)*. IEEE Computer Society, April 2020, p. 254–262.
[9] J. Offutt, “From spec-based testing to test automation and beyond (keynote address),” in *Workshop on Advances in Model Based Testing (A-MOST)*, Vasteros, Sweden, April 2018. [Online]. Available: https://cs.gmu.edu/~offutt/documents/slides/2018AMost-keynote.pptx
[10] ——, “It is great that we automate our tests, but why are they so bad?” SAST Industry Day at the 11th IEEE Conference on Software Testing, Validation, and Verification, Vasteros, Sweden, April 2018. [Online]. Available: https://cs.gmu.edu/~offutt/documents/slides/2018SAST-ICST.pptx
[11] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold, “Prioritizing test cases for regression testing,” *IEEE Transactions on Software Engineering*, vol. 27, no. 10, pp. 929–948, 2001.
[12] S. Elbaum, A. G. Malishevsky, and G. Rothermel, “Prioritizing test cases for regression testing,” *SIGSOFT Software Engineering Notes*, vol. 25, no. 5, pp. 102–112, 2000.
[13] G. Rothermel, S. Elbaum, A. Malishevsky, P. Kallakuri, and B. Davia, “The impact of test suite granularity on the cost-effectiveness of regression testing,” in *International Conference on Software Engineering*, 2002, pp. 130–140.
[14] H. Do, G. Rothermel, and A. Kinneer, “Empirical studies of test case prioritization in a JUnit testing environment,” in *International symposium on software reliability engineering*. IEEE, 2004, pp. 113–124.
[15] H. Do and G. Rothermel, “A controlled experiment assessing test case prioritization techniques via mutation faults,” in *21st IEEE International Conference on Software Maintenance (ICSM)*, 2005, pp. 411–420.
[16] Y. Lou, D. Hao, and L. Zhang, “Mutation-based test-case prioritization in software evolution,” in *26th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2015, pp. 46–57.
[17] L. Zhang, D. Marinov, and S. Khurshid, “Faster mutation testing inspired by test prioritization and reduction,” in *International Symposium on Software Testing and Analysis*. ACM, 2013, pp. 235–245.
[18] F. T. INRIA, Online, https://asm.ow2.io/, last access: May 2020.
[19] S. M. Ghaffarian and H. R. Shahriari, “Neural software vulnerability analysis using rich intermediate graph representations of programs,” *Elsevier’s Information Sciences*, pp. 189–207, April 2021.
[20] Graphviz, “The DOT language,” Online, https://www.graphviz.org/doc/info/lang.html, last access: May 2020.
[21] D. Crockford, “Introducing JSON,” Online, https://www.json.org/json-en.html , last access: May 2020.
[22] P. S. Foundation, “Sequencematcher,” Online, https://docs.python.org/3/library/difflib.html#difflib.SequenceMatcher, last access: May 2020.
[23] ——, “difflib — helpers for computing deltas,” Online, https://docs.python.org/3/library/difflib.html#, last access: May 2020.
[24] T. A. S. Foundation, “Apache maven project,” Online, https://maven.apache.org/what-is-maven.html, last access: May 2020.
[25] R. Just, F. Schweiggert, and G. M. Kapfhammer, “MAJOR: An efficient and extensible tool for mutation analysis in a Java compiler,” in *International Conference on Automated Software Engineering (ASE)*, November 2011, pp. 612–615.
[26] J. H. Andrews, L. C. Briand, and Y. Labiche, “Is mutation an appropriate tool for testing experiments?” in *27th International Conference on Software Engineering*, ser. ICSE ’05. ACM, 2005, pp. 402–411.
[27] H. Coles, “Pit mutation testing tool,” Online, https://github.com/hcoles/pitest, last access: October 2020.
[28] M. J. Harrold and M. L. Soffa, “Selecting and using data for integration testing,” *IEEE Software*, vol. 8, no. 2, pp. 58–65, March 1991.
[29] V. Martena, A. Orso, and M. Pezzé, “Interclass testing of object oriented software,” in *International Conference on Engineering of Complex Computer Systems (ICECCS)*. Greenbelt, MD: IEEE Computer Society, 2002, pp. 135–144.
[30] S. Yoo and M. Harman, “Regression testing minimization, selection and prioritization: A survey,” *Wiley’s Software Testing, Verification and Reliability*, vol. 22, no. 2, pp. 67–120, 2012.
[31] T. Y. Chen and M. F. Lau, “Dividing strategies for the optimization of a test suite,” *Information Processing Letters, Elsevier*, vol. 60, pp. 135–141, 1996.
[32] P. Ammann, M. E. Delamaro, and J. Offutt, “Establishing Theoretical Minimal Sets of Mutants,” in *IEEE International Conference on Software Testing, Verification and Validation (ICST)*, 2014, pp. 21–30.
[33] B. Vaysburg, L. H. Tahat, and B. Korel, “Dependence analysis in reduction of requirement based test suites,” *SIGSOFT Software Engineering Notes*, vol. 27, no. 4, p. 107–111, July 2002.
[34] R. Gupta, M. J. Harrold, and M. L. Soffa, “An approach to regression testing using slicing,” in *International Conference on Software Maintenance*, vol. 92. Citeseer, 1992, pp. 299–308.
[35] M. J. Harrold and M. L. Soffa, “Interprocedual data flow testing,” *ACM SIGSOFT Software Engineering Notes*, vol. 14, no. 8, pp. 158–167, 1989.
[36] H. Agrawal, J. R. Horgan, E. W. Krauser, and S. A. London, “Incremental regression testing,” in *Conference on Software Maintenance*. IEEE, 1993, pp. 348–357.
[37] G. Rothermel and M. J. Harrold, “A safe, efficient algorithm for regression test selection,” in *Conference on Software Maintenance*. IEEE, 1993, pp. 358–367.
[38] ——, “Selecting tests and identifying test coverage requirements for modified software,” in *ACM SIGSOFT international symposium on Software testing and analysis*, 1994, pp. 169–184.
[39] ——, *Efficient, effective regression testing using safe test selection techniques*. Clemson University, 1996.
[40] ——, “A safe, efficient regression test selection technique,” *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 6, no. 2, pp. 173–210, 1997.
[41] F. I. Vokolos and P. G. Frankl, “Pythia: A regression test selection tool based on textual differencing,” in *Reliability, quality and safety of software-intensive systems*. Springer, 1997, pp. 3–21.
[42] F. Vokolos and P. Frankl, “Empirical evaluation of the textual differencing regression testing technique,” in *International Conference on Software Maintenance*. IEEE, 1998, pp. 44–53.
[43] H. Do and G. Rothermel, “On the use of mutation faults in empirical assessments of test case prioritization techniques,” *IEEE Transactions on Software Engineering*, vol. 32, no. 9, pp. 733–752, 2006.
[44] L. Gazzola, D. Micucci, and L. Mariani, “Automatic software repair: A survey,” *IEEE Transactions on Software Engineering*, vol. 45, no. 1, pp. 34–67, January 2019.
[45] Google, “Google test automation conference,” Online, https://developers.google.com/google-test-automation-conference/, last access: September 2019.
[46] L. C. Briand, M. D. Penta, and Y. Labiche, “Assessing and improving state-based class testing: A series of experiments,” *IEEE Transaction on Software Engineering*, vol. 30, no. 11, pp. 770–793, November 2004.
[47] N. Li and J. Offutt, “Test oracle strategies for model-based testing,” *IEEE Transactions on Software Engineering*, vol. 43, no. 4, pp. 372–395, April 2017.
[48] J. Bell, O. Legunsen, M. Hilton, L. Eloussi, T. Yung, and D. Marinov, “DeFlaker: Automatically detecting flaky tests,” in *40th International Conference on Software Engineering*, Gothenburg, Sweden, May 2018.
[49] N. Li and J. Offutt, “A test automation language for behavioral models,” in *11th IEEE Workshop on Advances in Model-based testing (A-MOST)*, Graz, Austria, April 2015.