Using Control Flow Analysis to Improve the Effectiveness of Incremental Mutation Testing[*]

Luke Bajada
PEST Research Lab,
University of Malta
luke.bajada@um.edu.mt

Mark Micallef
PEST Research Lab,
University of Malta
mark.micallef@um.edu.mt

Christian Colombo
PEST Research Lab,
University of Malta
christian.colombo@um.edu.mt

ABSTRACT

Incremental Mutation Testing attempts to make mutation testing less expensive by applying it incrementally to a system as it evolves. This approach fits current trends of iterative software development with the main idea being that by carrying out mutation analysis in frequent bite-sized chunks focused on areas of the code which have changed, one can build confidence in the adequacy of a test suite incrementally. Yet this depends on how precisely one can characterise the effects of a change to a program. The original technique uses a naïve approach whereby changes are characterised only by syntactic changes. In this paper we propose bolstering incremental mutation testing by using control flow analysis to identify semantic repercussions which a syntactic change will have on a system. Our initial results based on two case studies demonstrate that numerous relevant mutants which would have otherwise not been considered using the naïve approach, are now being generated. However, the cost of identifying these mutants is significant when compared to the naïve approach, although it remains advantageous when compared to traditional mutation testing so long as the increment is sufficiently small.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement; D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Reliability, Performance

Keywords

Test Suite Adequacy Analysis, Incremental Mutation Testing, Dataflow Analysis

1. INTRODUCTION

Mutation testing [9] is a technique which analyses the adequacy of a test suite using fault injection. Whilst it has been shown to be more effective in finding test suite deficiencies than other measures such as code coverage analysis, the expense associated with the approach is still a significant barrier to entry to the industry. To improve the performance, we have proposed a way of splitting the cost of mutation testing over the iterations of the software development process [4]. This was achieved by applying mutation testing to only the changed parts of the code base since a previous commit, thus carrying out mutation testing incrementally over time.

The main challenge in this approach is to precisely characterise the semantic impact of a syntactic change in the code base, given that a change in one part of a program can affect parts which would have been modified indirectly through calls to changed methods, data flows, or through shared resources with modified parts of the system.

In this paper, we recount our experience of attempting to improve the technique by leveraging control flow analysis (see Section 3). In particular, the paper discusses work done to answer the following research questions:

RQ1: Can the effectiveness of incremental testing be improved by using control flow analysis to more precisely characterise the effects of changes between two evolutions of a program?

RQ2: What tradeoffs exist between expense and effectiveness when using this approach?

Two case studies were carried out as part of our research (see Section 4), one on an open source project and the other on an industry case study with a partner in the payment processing industry.

2. BACKGROUND

In this section, we provide a brief overview of incremental mutation testing and relevant static analysis techniques which were used in this research.

2.1 Incremental Mutation Testing

In essence, mutation testing works as follows: given a program P and a test suite T which tests P, the approach involves generating faulty variations of P (called mutants) and checking whether for every mutant, there is at least one test case in T which fails. We write $T(P)$ to denote a successful run of test suite T on program P

[*] Project GOMTA financed by the Malta Council for Science & Technology through the National Research & Innovation Programme 2013

and $\neg T(P)$ to denote that at least one of the tests in the test suite has failed on P.

Mutation testing begins by generating a set of programs $P_1, P_2, \ldots, P_n$ using a set of mutation operators represented by the function $\mathcal{M}$ on the program P, $\mathcal{M}(P) = \{P_1, P_2, \ldots, P_n\}$. These programs are usually syntactically similar to P but never (syntactically) equivalent to it. That is to say $\forall i : 1..n \cdot P_i \not\equiv P$.

Subsequently, T is executed against all $P_i \in \mathcal{M}(P)$. For each mutant P_i, if at least one test results in a failure, we say that T has *killed* the mutant. Otherwise, we say that the mutant remains *unkilled*. Unkilled mutants might indicate a diminished adequacy of T with respect to P. This brings us to our definition of coverage.

Definition 1. (Coverage) A test suite T is said to cover a program P, denoted $T \triangleright P$ if and only if P satisfies T, $T(P)$, while any $P_i \in \mathcal{M}(P)$ fails the test suite, $\neg T(P_i)$:

$$T \triangleright P \overset{\text{def}}{=} T(P) \wedge \forall P_i \in \mathcal{M}(P) \cdot \neg T(P_i)$$

The ratio of killed mutants to total mutants is known as the mutation score and provides a measure of test suite coverage in the context of the generated mutants. Mutation operators are usually designed to change P in a way that corresponds to a fault which could be introduced by a developer. Consequently, in comparison to techniques such as statement coverage analysis, mutation testing provides a significantly more reliable measure of test suite thoroughness [3, 5]. Despite its effectiveness, mutation testing suffers from a significant problem [9]: Whilst the polynomial computational complexity of mutation testing does not seem prohibitive, in a typical commercial system the large amount of potential mutation points would make the computational expense considerably high. Furthermore, once mutants have been generated, each one needs to be tested against the original program's test suite. Considering that test suites on large systems will optimistically take a few minutes to execute, the time required for this task would be considerable.

Incremental mutation testing attempts to alleviate the prohibitive computational expense associated with mutation testing by leveraging the evolutionary nature of modern software development practices such as Agile development. The underpinning idea is that of limiting the scope of mutation testing to code that has changed within the context of two particular versions of the code. This effectively selects areas of the code with elevated *code churn* — a measure of changes made to a software component over a period of time which has been shown to effectively predict defect density during system evolution [12]. By applying mutation testing on each change across successive versions of the code, over the entire evolutionary process, one would have effectively applied mutation testing over the whole system, incrementally. More precisely, incremental mutation testing assumes two programs P and P_{ev} where P_{ev} is an evolution of P such that P_{ev} consists of two parts: a changed part (P^{δ}_{ev}) which has evolved from a corresponding part of P (P^{δ}), and an unchanged part ($P^{\not\delta} = P^{\not\delta}_{ev} = P^{\not\delta}$) with respect to P. We therefore represent P and P_{ev} as $P = P^{\delta} + P^{\not\delta}$ and $P_{ev} = P^{\delta}_{ev} + P^{\not\delta}$. In this context, the composition operator $+$ assumes that there is a way of splitting a program into two parts such that the parts can be tested independently. Similarly, the technique assumes that there is a way of splitting the test suite into (potentially overlapping) parts which test the corresponding program parts. Formally:

Definition 2. (Independent testability) For any program $P = P^{\delta} + P^{\not\delta}$ and test suite $T = T^{\delta} + T^{\not\delta}$, the program passes the test suite if and only if its parts pass the corresponding parts of the test suite respectively:

$$T(P) \iff T^{\delta}(P^{\delta}) \wedge T^{\not\delta}(P^{\not\delta})$$

Using these assumptions, given that a test suite has been shown to adequately cover a system under test, in the following evolution of the code, this information can be used to minimise the number of mutations required to check the test suite. Intuitively, this is achieved by eliminating the unchanged part of the system from mutation testing: if the second version of the code can be split into the changed part and the unchanged part, incremental mutation testing assumes that tests relating to the unchanged part do not need to be analysed for thoroughness because this would have been done in previous evolutions of the code. More formally, this idea is captured in the following theorem:

Theorem 1 (Incremental mutation testing). *If the system code $P = P^{\delta} + P^{\not\delta}$ has been shown to be adequately covered by a test suite T, $T \triangleright (P^{\delta} + P^{\not\delta})$, then to show that the new version is also adequately covered, $T_{ev} \triangleright (P^{\delta}_{ev} + P^{\not\delta})$, it suffices to check that $T^{\delta}_{ev} \triangleright P^{\delta}_{ev}$:*

$$T \triangleright (P^{\delta} + P^{\not\delta}) \wedge T^{\delta}_{ev} \triangleright P^{\delta}_{ev} \implies T_{ev} \triangleright (P^{\delta}_{ev} + P^{\not\delta})$$

Whilst this theorem has been shown to be true [4], the significant assumptions described above which make the theorem work, cannot be overlooked. In particular, being able to split a program into the changed and the unchanged part such that the subparts can be tested independently, is in general a difficult task. In our previous work [4], we have taken the naïve approach of simply separating the new/changed methods from the unchanged methods (and their corresponding tests respectively). Clearly, this approach does not satisfy our assumption since this does not guarantee independent testability — for example changed methods might be called from unchanged methods, or unchanged methods may share global variables with changed ones.

In order to mitigate this issue, we propose to use control flow analysis techniques to analyse the structure of the code under consideration and find better approximations of the impact of a syntactic change. This should enable us to move closer to satisfying the assumption which is crucial for Theorem 1 to hold.

2.2 Static Analysis

Due to the issue of independent testability (see Definition 2), we attempt to utilise information about the *coupling* within a software system — a qualitative measure that shows the degree of interdependence between modules. Whilst Lethbridge and Lagamiere [10] identify nine types of coupling, in this work we choose to target *routine call* coupling since the design principle of *separation of concerns* in the object-oriented paradigm naturally leads to a common occurrence of this type of coupling as objects delegate functionality to each other through method calls. To this extent, we turn our attention to static analysis which enables us to elicit the necessary information about the coupling features of interest.

Control flow analysis is used to determine the control flow relationships in a program [2]. The control flow is the order in which statements or function calls in code are executed. Of particular interest to us is Shivers' work in identifying the possible set of callers to a function or method call [14]. This is because, due to the notion of indirect inputs[1], a change in a particular method is likely to affect the behaviour of methods which call (or transitively call) the changed method. Shivers' work uses a call graph as the internal representation of choice during static analysis. The call graph of a program is a directed graph that represents the relationships between the program's procedures or methods [6]. According to the method being analysed in the call graph, the calling relationships between that method and the methods that call it can span over several depths.

In the next section, we explain how using the call graph, we identify the indirectly affected parts of a code base so that this is considered for mutation testing.

3. PROPOSED APPROACH

Recall from Section 2.1 that in incremental mutation testing, we consider a program P_{ev}, an evolution of P, to be composed of two parts $P_{ev}^{\delta} + P^{\not\delta}$. In [4], we approximate P_{ev}^{δ} by identifying the set of methods (which we will refer to as M^{δ}) that contain syntactic differences between P and P_{ev}. We consider this approach as being naïve because the impact of a syntactic change to the code base is not necessarily limited to the location of that change — unchanged locations of the code which depend on changed locations could potentially be impacted. Therefore our naïve approach risks a situation whereby potentially valuable mutants would not be generated and analysed.

In this work, we improve the approximation of P_{ev}^{δ} by including a set of methods (referred to as *callSet*) which call (transitively up to a particular depth) the methods in M^{δ} as follows:

Definition 3. (callSet) Assuming we have a call graph $\langle V, E\rangle$ where V represents methods and E represents tuples of methods (m', m) signifying that m' calls m,

$$\mathit{callSet}(M^{\delta}, d) =$$

$$M^{\delta} \cup \begin{cases} \emptyset & \text{if } d = 0 \\ \displaystyle\bigcup_{m \in M^{\delta}} \mathit{callSet}\left(\{m' \in V \mid (m', m) \in E\}, d-1\right) & \text{otherwise} \end{cases}$$

Intuitively, consider a call graph of a program as depicted in Figure 1. The grey circle in the middle of the call graph represents a method in which a syntactic change has occurred whilst unshaded circles connected by a directed edge indicate pairs of methods in which one method calls the other (as indicated by the direction of the edge). Whereas in [4], the naïve approach only considered the shaded circle for analysis, in this paper we also take the other circles into account. We are only interested in including methods which call (or transitively call) a method where a syntactic change has occurred because their behaviour could potentially be effected by the change.

Note that we refrain from defining the value of d because as part of our investigation we look at how various depth values affect the performance of our approach.

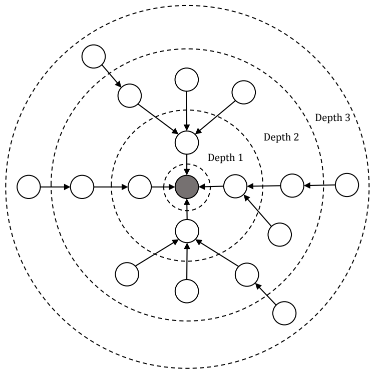

Figure 1: Example of call graph analysis at various depths

Finally, we remark that we do not extend this same approach to refine which tests are executed for each mutant, since in [4] we already selected tests which directly call mutated methods. We consider this to be adequate since we are dealing with unit tests (which should be only concerned with testing the called methods directly).

4. EVALUATION

As with most approximation techniques, there is a trade off between effectiveness and performance. In general, a more precise approximation comes with a higher cost. To this end, our evaluation takes the form of a cost-benefit analysis of the proposed technique via two case studies: one is an open source project having 5KLOC, which has also been used in our previous study [4] — the Apache Commons CLI library, while the second is an 10KLOC industrial system provided by a partner in the payments processing industry who utilises an iterative development cycle. The CLI library was selected in order to maintain consistency with our previous study, as well as having the characteristic of being very mature (thirteen years old with 97% code coverage) and maintained by a community of developers. On the other hand, the payments processing system was selected in order to provide a case study of a system still under development (one year old with 60% code coverage) by a focused team of developers.

Following our previous approach, we selected three different time periods in the life time of each project such that: (*i*) all periods had the same commencement timestamp and (*ii*) the end timestamp of each time period resulted in a day's worth of development, a week's worth of development, and finally a month's worth of development. In the case of the Apache library which was used for a case study in [4], the same time periods were retained in order to maintain consistency.

For each case study, we carried out mutation testing runs[2] as follows: First using the naïve approach from [4], and then using the control flow analysis technique whilst increasing the depth analysis level until no further methods are identified for mutation. Each mutation testing run was executed three times to ensure consistency in the data, with less than 10% variation observed. Due to the fact that the non-disclosure agreement with our industry partner required that source code does not leave their premises, experiments on the two case studies where done on different machines as shown in Ta-

[1] The notion of indirect inputs refers to situations whereby method behaviour is influenced by means other than parameter values, most commonly return values of called methods [11].

[2] We also kept the same set of seventeen basic mutation operators from [4], relying on the mutation coupling effect hypothesis stating implying that complex mutants do not give any significant advantage on simple ones.

	Lab Machine	**Industry Machine**
CPU	Intel Core i7 2.2 GHz	Intel Core i7 2.93 GHz
RAM	6 GB	8 GB
OS	Windows 8.1 (64 bit)	Windows 7 SP1 (64 bit)
Java	1.7 (64 bit)	1.7 (64 bit)

Table 1: Hardware setup details for each case study

Apache Commons CLI					
Experiment	Depth	Mutants	Killed	Score	Time (s)
Day	Trad.	253	110	43%	20
	0	95	45	47%	5
	1	122	58	48%	18
Week	Trad.	340	113	33%	20
	0	183	57	31%	9
	1	210	68	32%	20
Month	Trad.	349	131	38%	16
	0	158	73	46%	6
	1	306	108	35%	19
	2	312	111	36%	20
	3	315	112	36%	21

Industry					
Experiment	Depth	Mutants	Killed	Score	Time (s)
Day	Trad.	954	577	60%	63
	0	57	42	74%	4
	1	182	140	77%	23
	2	191	149	78%	24
Week	Trad.	954	577	60%	63
	0	347	210	61%	25
	1	415	237	57%	39
	2	418	240	57%	39
Month	Trad.	954	577	60%	63
	0	700	374	53%	35
	1	728	399	55%	54
	2	731	402	55%	54

Table 2: Absolute results obtained for both case studies

ble 1. The full results of the experiments, including the mutation score, are shown in Table 2.

The following subsections focus on the benefit and the cost of applying static analysis in the context of our case studies, followed by a discussion in the final subsection.

4.1 Benefit: Mutant Increase Analysis

The main concern of adopting the naïve approach in distinguishing the changed part from the unchanged part of a system is that potentially valuable mutants might never be generated — meaning that test suite deficiencies might not be detected simply due to such missing mutants.

Thus, measuring the benefit of introducing static analysis mainly consists of counting the number of mutants which would otherwise not have been tested.[3] Table 3 show the percentage increase in the number of mutants from one depth to another.

Apache Commons CLI			
Experiment	Depth 0→1	Depth 1→2	Depth 2→3
Day	28%	0%	0%
Week	15%	0%	0%
Month	94%	2%	1%

Industry			
Experiment	Depth 0→1	Depth 1→2	Depth 2→3
Day	219%	5%	0%
Week	20%	1%	0%
Month	4%	<1%	0%

Table 3: Mutant percentage increase according to depth change

Analysing the results horizontally suggests that most changes do not occur deeper than one level below the surface of the call graph, showing significant increase in mutants only when going from depth zero to depth one. Although our case studies show that the influence of a change never reaches beyond a depth of three method calls, this is highly dependent on the topology of the individual system's call graph and the location of the change itself.

Whilst the rows of the tables give us insight into the topology of the call graph — namely its depth and how this is distributed, we hypothesise that the columns of the tables can shed light on the kind of changes in the code churn. More specifically, on how the changes were distributed along the call graph: if the changes are focused along a particular branch of the call graph, then one would not expect to find many affected methods which have not been directly modified. On the contrary, if several unrelated branches have minor changes away from the root, then one would expect to find numerous methods which would have been affected. Naturally, such a distribution depends on numerous factors including the maturity of the project (mature projects would tend to have more minor and unrelated modifications in deeper methods) and on the way the work on the project is managed (an industry team of developers would tend to work more on a feature by feature approach whilst in the case of an open source project, one would expect many unrelated parts of the project to be touched). The observations seem to be corroborated by the disparaging results of the tables where in the case of our industry case study, the changes were more focused on a day by day basis and yet covered most of the system over a month since the system was still being developed. On the other hand, in the case of the open source library the wider breadth of the changes over a month meant that the number of mutants added through static analysis was at its highest point.

4.2 Cost: Execution Time

Compared to the traditional mutation testing approach which attempts to kill all the mutants (irrespective of whether they were affected by the changes), the proposed approach has the advantage of typically excluding a substantial number of mutants from analysis but still has the disadvantage of carrying out analysis on what we consider to be valuable mutants. Effectively, the gains can be characterised as follows:

$$\begin{aligned} \textit{Total Gains} &= \textit{Time(all muts.)} - \textit{Time(selected muts.)} \\ \textit{Actual Gains} &= \textit{Total Gains} - \textit{Time(static analysis)} \end{aligned}$$

More concretely, this means that unless the number of included mutants multiplied by the time required to execute the test suite is high

[3] Note that we assume that it is useful to generate mutants from methods (possibly transitively) calling changed methods. Thus, we ignore whether or not such mutants are actually killed or not. Such information would only have a baring on the quality of the test suite, not the effectiveness of our technique.

Apache Commons CLI				
Experiment	Depth 1		Depth 2	
Measure	%muts.	speedup	%muts.	speedup
Day	48%	1.1x	48%	1.1x
Week	83%	1.0x	83%	1.0x
Month	88%	0.8x	89%	0.8x

Industry				
Experiment	Depth 1		Depth 2	
Measure	%muts.	speedup	%muts.	speedup
Day	19%	2.7x	20%	2.6x
Week	44%	1.6x	44%	1.6x
Month	76%	1.2x	77%	1.2x

Table 4: Performance gain from traditional mutation testing

Apache Commons CLI			
Experiment	Naïve	Proposed	
Measure	(depth 0)	depth 1	depth 2
Day	5	18	18
Week	9	20	20
Month	6	19	20

Industry			
Experiment	Naïve	Proposed	
Measure	(depth 0)	depth 1	depth 2
Day	4	23	24
Week	25	39	39
Month	35	54	54

Table 5: Execution time compared to naïve approach

enough to at least counteract the time spent in static analysis, the approach is not beneficial. In fact, referring to the results shown in Table 4, we note that in the case of the *month* time period for the Apache library, where most of the mutants were selected anyway, incremental mutation testing approach was actually slower than traditional mutation testing.

The proposed technique, whilst still generally faring better than traditional mutation testing, when compared to the naïve approach of mutating only directly changed methods, is naturally slower due to two main reasons: the generation of the call graph and the generation and analysis of the extra mutants. Arguably, the time spent on the latter is justifiable by the increased effectiveness, otherwise one should stick to the naïve approach in the first place. However, the time spent generating the call graph is significant when compared to the time required to analyse the call graph: eight seconds for the Apache library case study and twelve for the industrial one. Table 5 provide the total execution time for the naïve and the proposed approach including the time to generate the call graph. Whilst at face value the difference is staggering, when one considers the increase in mutants being considered and the cost of generating the call graph, the numbers add up: For example the seemingly huge nineteen second gap from four to 23 seconds in the industry day scenario is composed of twelve seconds to generate the call graph, and seven seconds which is approximately 219% of four seconds required in the naïve approach.

Finally, considering the very small differences in readings between the values in *depth 1* and *depth 2* columns, notwithstanding the minimal increase in the number of changed methods identified, we note that compared to the cost of generating the call graph, the cost of analysing it to consider deeper methods is negligible.

We conclude this section by noting a number of limitations our analysis currently has:

Runtime resolution of dependencies There are cases where a control dependency is resolved at runtime: one such case is when a method call is resolved polymorphically, while another example would be J2EE XML configurations. At the moment we do not take these into consideration, possibly missing out on a number of useful mutations.

Depth estimation Due to the limited number of case studies, we are not able to estimate the depth required in general to identify all methods requiring mutation. From our experience, in both cases most methods were identified at depth one, very few at depth two, and even fewer at depth three. Whilst it is desirable to have a way of calculating the depth required upfront, we note that the analysis of the call graph for an extra level is negligible compared to the time needed to carry out mutation operators and rerun the test suite for each mutant.

Effectiveness approach with respect to changes demographic While we have made a number of remarks regarding how the distribution of code modification across the call graph may affect the effectiveness of our approach, we believe that this can be explored more in the future. Ideally, one could have an indication of how useful the technique would be before actually wasting time generating the call graph.

Incrementally building the call graph Our current implementation rebuilds the call graph each time the incremental mutation process is carried out. This is a substantial limitation given how expensive it is to build the call graph. In the future, this can be built incrementally by simply updating the call graph to reflect the changes in the code.

5. RELATED WORK

Several related works dedicated to making mutation testing more feasible already exist, each of which falls into one of three categories: “do smarter", “do faster" and “do fewer" [8]. Perhaps incremental mutation testing could loosely fit under “do fewer”, although our aim is to split the cost of mutation testing over iterations rather than reducing the number of mutants per se. In this particular paper, our focus was on improving an existing technique rather than develop a new one. To the best of our knowledge, there is no other work on improving incremental mutation testing, so we will focus this section on work from the field of *regression test selection*, our main influence of this work.

In the field of regression test selection, given a program P, the modified version of P, P', and a test suite T, the test case selection problem involves finding a subset of T, T', with which to test P' [16]. This problem has strong parallels with our problem of finding a subset of methods in an evolution of the program, which are likely to produce valuable mutants during mutation analysis.

The literature contains a wide variety of approaches to addressing the problem including amongst others *dynamic program slicing* [1], *data flow analysis* [7], symbolic execution [15] and graph walking [13]. Of these, the latter is most closely related to the work presented here. Graph walking techniques involve the generation of graph-based internal representations of a program which are subsequently analysed with respect to a certain question. In general, graph walking techniques involve the generation of one of the graph-based representations for both P and P', and then *walking* through the graphs looking for paths which lead to modified code. Once these paths have been identified, test cases which execute control-dependent predecessors[4] of the modified code are selected. In this work, we adopt the same technique albeit to select methods for mutation generation rather than tests for regression testing.

6. CONCLUSIONS AND FUTURE WORK

In this paper, we presented a technique which utilises control flow analysis in order to more precisely characterise the differences between two versions of a program. We also discussed results from two case studies which indicate that the technique does in fact improve the effectiveness of incremental mutation testing, albeit at a cost in performance. Having said that, in most cases, the loss in performance still resulted in a substantial speedup over traditional mutation testing. We also observed that whilst the nature of change and the topology of the call graph will influence performance, in general, the more frequently incremental mutation testing is executed, the more cost-effective it becomes. That is to say that carrying out mutation analysis at the end of each day is more cost effective than carrying it out once a month. This is because a large number of changes would have occurred in the longer time period, resulting in more mutants needing to be generated and analysed in one run.

We acknowledge that the results presented here suffer from threats to validity in the same way that all case study based evaluations do. Two case studies are by no means representative of the entire population of software systems and further studies are required in order to generalise the results. However, we argue that the results have provided some interesting initial insights into the costs and benefits of the present technique.

6.1 Future Work

There are several avenues of exploration to further improve the effectiveness of incremental mutation testing. Firstly, we would like to carry out wider-reaching case studies on projects in both the industry and the open-source world. This should help us form a better understanding about the contexts in which the technique is likely to be beneficial when compared to traditional mutation testing. Secondly, we would also like to consider other types of coupling when analysing the impact of a syntactic change on a system. More specifically, we would like to investigate how considering data flow within the system would affect incremental mutation testing and what (if any) overlap this would have with control flow analysis.

7. REFERENCES

[1] H. Agrawal, J. R. Horgan, E. W. Krauser, and S. London. Incremental regression testing. In *International Conference on Software Maintenance (ICSM)*, volume 93, pages 348–357, 1993.

[2] F. E. Allen. Control flow analysis. *SIGPLAN Not.*, 5(7):1–19, July 1970.

[3] J. H. Andrews, L. C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *Proceedings of the 27th international conference on Software engineering*, ICSE '05, pages 402–411, New York, NY, USA, 2005. ACM.

[4] M. A. Cachia, M. Micallef, and C. Colombo. Towards incremental mutation testing. *Electronic Notes in Theoretical Computer Science*, 294:2 – 11, 2013. Proceedings of the 2013 Validation Strategies for Software Evolution (VSSE) Workshop.

[5] M. E. Delamaro, J. Maldonado, A. Pasquini, and A. P. Mathur. Interface mutation test adequacy criterion: An empirical evaluation. *Empirical Softw. Engg.*, 6(2):111–142, June 2001.

[6] D. Grove, G. DeFouw, J. Dean, and C. Chambers. Call graph construction in object-oriented languages. *SIGPLAN Not.*, 32(10):108–124, Oct. 1997.

[7] M. J. Harrold and M. L. Soffa. Interprocedual data flow testing. In *ACM SIGSOFT Software Engineering Notes*, volume 14, pages 158–167. ACM, 1989.

[8] Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. *Software Engineering, IEEE Transactions on*, 37(5):649–678, 2011.

[9] H. M. Jia Y. An analysis and survey of the development of mutation testing. *ACM SIGSOFT Software Engineering Notes*, 1993.

[10] T. C. Lethbridge and R. Lagamiere. *Object-oriented software engineering - practical software development using UML and Java.* MacGraw-Hill, 2001.

[11] G. Meszaroz and A. Wesley. Xunit test patterns: Refactoring test code. *Citado na*, page 27, 2007.

[12] N. Nagappan. *A Software Testing and Reliability Early Warning (STREW) Metric Suite*. PhD thesis, 2005.

[13] G. Rothermel and M. J. Harrold. A safe, efficient regression test selection technique. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 6(2):173–210, 1997.

[14] O. Shivers. Control flow analysis in scheme. *SIGPLAN Not.*, 23(7):164–174, June 1988.

[15] S. S. Yau and Z. Kishimoto. A method for revalidating modified programs in the maintenance phase. In *Annual International Computers, Software & Applications Conference (COMPSAC)*, volume 87, pages 272–277, 1987.

[16] S. Yoo and M. Harman. Regression testing minimization, selection and prioritization: a survey. *Software Testing, Verification and Reliability*, 22(2):67–120, 2012.

[4] Control dependent predecessors of a node N in a control-oriented graph representation of a program are nodes which at some point in the execution have an influence as to whether or not N is executed.