An Anytime Approach to Analyzing Software Systems

Dan Rubenstein, Leon Osterweil, and Shlomo Zilberstein
University of Massachusetts at Amherst
{drubenst, ljo, shlomo}@cs.umass.edu

Abstract

Proving that a software system satisfies its requirements is a costly process. This paper discusses the benefits and challenges of structuring the analysis of software as an anytime algorithm. We demonstrate that certain incremental approaches to event sequence analysis that produce partial results are anytime algorithms, and we show how these partial results can be used to optimize the time to complete the full analysis.

1 Introduction

Computers are an integral part of society these days, controlling many activities that affect our daily lives. There is an inherent danger that the software that determines what these computers do has faults, causing the computers to perform in a fashion that fails to meet their requirements. The purpose of testing and analyzing software is to remove, or at least reduce, the possibility of failures.

Performing an accurate analysis to verify a property about a program is often time consuming. *Incremental analyses* have been proposed as a way to reduce analysis time by incrementally performing *conservative* analyses with increasing accuracy [3]. An analysis is said to be conservative if the analysis never concludes that a property holds if the property does not indeed hold. However, it need not always succeed in concluding that the property holds, even when it actually does. When a conservative analysis concludes that the property holds, the result is said to be *conclusive* in that it is correct. Otherwise, the result is said to be *inconclusive*, and a more accurate analysis must be performed to remove all doubt and verify whether or not the property holds.[1] Verifying a property often takes less time using incremental analyses because the property can sometimes be verified after running only a few quick, inaccurate, conservative analyses as opposed to running a single slow, accurate analysis.

In this paper, we show how the accuracy of each analysis used in an incremental analysis can be measured such that the incremental analysis satisfies the properties of an *anytime algorithm* [7]. We also show that the intermediate results obtained from the incremental analysis can then be used to dynamically determine which analyses are to be performed next in order to improve the running time for the overall analysis and increase the chances of obtaining conclusive results of the overall analysis. We provide evidence of this through experimentation on sequential Ada83 code.

Different kinds of analyses use different methods to analyze programs. In this paper we restrict our focus to *sequencing analyses*, which examine certain events within a program, and attempt to verify that only a restricted set of sequences of these events can occur. Section 2 gives a background on how sequencing analyses are performed. Section 3 discusses the benefit obtained in making an analysis an anytime algorithm, and shows how we derive a measure of accuracy for an analysis. We show in section 4 how the anytime features of the analysis can be used to optimize the incremental procedure, and present experimentation and results in section 5. Finally, we summarize and discuss future work in section 6.

[1] Converse analyses are also often called conservative, where the analysis returns a conclusive result only when a property does not hold. For the sake of simplicity, we do not consider such analyses in this paper.

2 Incremental Sequence Analysis

Many faults that occur in programs are due to errors in the sequencing of events within the code. Initially, analysis tools were constructed to detect anomalous sequences of events that were common to many programs, such as dead definition and undefined reference anomalies [6]. Later, methods were developed that gave the user of the analysis system the ability to choose the sequences that should be considered anomalous. With this approach, the user specifies the set of sequences as a *quantified regular expression* (QRE) [1], [2], [4], [5]. Any sequence that is a member of the language described by the QRE is considered to be an *allowable* sequence. All other sequences are considered to be *anomalous*. A QRE is useful in that it is easily converted to a *finite state automaton* (FSA), a representation formalism commonly used as a basis for analyses. There are many methods available that allow one to detect anomalies that are represented as QREs. The results from this paper focus mainly on analyses that use data flow analysis [9].

2.1 Data Flow Analysis of QREs

In data flow analysis, a program is represented as a *control flow graph* (CFG), where each node represents a program execution unit, and directed edges represent flow of control between those units. The user selects a QRE, whose language consists of events that occur in the code. Each node in the control flow graph is then annotated with the events that occur at the code that corresponds to the node. If no event of interest occurs within the execution unit, its corresponding node is annotated with a special null event.

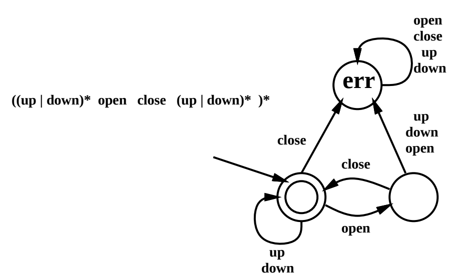

Figure 1: A sample QRE and its corresponding property automaton that is used to verify that a program that controls an elevator never allows the elevator to move up or down with the doors having been opened without subsequently being closed. The state with concentric circles is an accepting state. The state labeled **err** is the error state, which cannot be exited once transitioned to.

The QRE is then converted to an FSA, in which the events that annotate the nodes of the CFG annotate the transitions in the FSA. The FSA has a set of distinguished accepting states that are those states to which the FSA transitions in response to an event sequence described by the QRE. We refer to this FSA as the *property automaton*. An event sequence along a path is said to be allowable if the event sequence drives the property automaton to an accepting state after the sequence's final event. In any CFG with a cycle, there is an infinite number of paths through the graph, so one cannot examine each path individually. Data flow analysis nevertheless allows the examination of all paths through the graph with relatively low time complexity. Data flow analysis can therefore efficiently detect the existence of paths within a CFG that produce anomalous sequences of events.

Such an analysis is inaccurate in that many paths through the CFG do not represent feasible program executions. However, the analysis remains conservative in that it will never miss an executable, anomalous sequence. In other words, if a data flow analysis concludes that there are no anomalous sequences through the CFG, then none can exist within the program. However, if the data flow analysis detects anomalous sequences, these may or may not be executable, so the analysis result must be considered to be inconclusive. The accuracy of the analysis must then be increased by reducing the number of infeasible execution paths that are examined by the data flow analysis.

Accuracy can be increased by adding *constraints*. A constraint tracks an aspect of the program, such as the value of a variable, and ensures that a path is not examined if it violates the constraint. In [2], it is shown that constraints can also be represented as FSAs, which are referred to as *constraint automata*. A constraint automaton has a single non-accepting state, referred to as a *violation state*, that has no exiting transitions, and is entered as soon as a violation of the constraint occurs. The data flow analysis simply refuses to consider for analysis any path that causes a constraint FSA to enter its violation state.

Many such constraints may be used by the analysis simultaneously, in which case the data flow analysis algorithm must keep track of the effects of each path through the graph on all members of this set of FSAs. This is generally done by means of a *product automaton* which is the cross product of the property automaton and the set of all constraint automata. The number of states in the product automaton is equal to the product of the number of states in each of the individual automata, as each state in the product automaton represents a tuple of states from the set of individual automata. We say that a state in the product automaton is an *accepting state* if and only if the property automaton state in the tuple is accepting. Similarly, we say that a state in the product automaton is a *violation state* if and only if some constraint automaton state in the tuple is a violation state.

3 Anytime Algorithms

An anytime analysis incorporates a measure for the accuracy of the analysis, and a mechanism for selecting a sequence of analyses such that the measure varies in a monotonically, non-increasing (or non-decreasing), fashion over time [7]. It is also desirable for the measure to provide *diminishing returns* with time, where the change in accuracy decreases over time. Such a property guarantees that large gains in accuracy will occur in earlier increments of the analysis. One would also like to be able to use the measure of an inaccurate analysis to predict the subsequent analysis that will deliver the best improvement in accuracy.

Data flow analysis for checking sequences of events improves its accuracy by decreasing the set of unexecutable flow graph paths that it considers. A good measure of the analysis's accuracy might entail counting the set of such paths. Although the set of all paths through most CFGs is infinite, it is a countable set.[2] Thus, a density function could be constructed to define a ratio between a measure of the set of paths still considered by the data flow analysis to the measure of the set of all paths through the CFG. We propose to define these measures in terms of the numbers of equivalence classes of paths under an appropriate equivalence relation. We say that two paths A and B are equivalent under our relation for a particular analysis if and only if they take the product automaton used in that analysis to an identical state. Thus, the number of equivalence classes is always finite, being equal to the number of states in the product automaton, and the density function we seek will always be a rational number between 0 and 1.

We define the *inconclusiveness measure* to be X/Y, where X is the number of non-empty equivalence classes containing paths that transition the product automaton to a non-accepting, non-violation state, and Y is the total number of (empty and non-empty) equivalence classes. Hence, Y equals the total number of states in the product automaton, and X equals the number of non-accepting, non-violation states in the product automaton that at least one path through the CFG transitions to.

We have shown in [11] that the inconclusiveness measure is monotonically non-increasing each time a constraint is added and an analysis is performed. We now provide a brief sketch of the proof. Adding a constraint increases the number of states in the property automaton by a factor, s_{new}, which equals the number of states in the automaton associated with the new constraint. If we consider the inconclusiveness measure computed from an analysis that uses the newly formed product automaton that does not enforce violations that occur due to the new constraint, then X and Y each increase by a factor of s_{new}, and the inconclusiveness measure remains constant. Enforcing the violations of the new constraint can only reduce X and has no impact on Y, so that the inconclusiveness measure can only decrease in size. Therefore, the measure is monotonically non-increasing as constraints are added.

[2] The number of paths through any graph with a finite number of nodes is always countable.

4 Optimizing incremental analysis

We can also make use of the inconclusiveness measure to improve performance of the incremental algorithm by using it to estimate the time and accuracy of particular analysis increments. Ideally, we want to locate the set of constraints that should be used in an analysis that returns a measure of 0 (complete accuracy) in the shortest time. Through some initial experimentation on relatively simple programs, we have learned that making such a prediction is extremely difficult. The effects on analysis time and accuracy due to the addition of a constraint differ dramatically depending on the set of constraints that are already part of the analysis.

Vars constrained	Measure	Analysis Time
None	7.14286E-01	1.91
b1	2.14286E-01	1.90
b2	2.14286E-01	1.96
b3	1.78571E-01	2.05
b1, b2	3.57143E-02	1.49
b1, b3	5.10204E-03	11.78
b2, b3	5.10204E-03	10.09
b1, b2, b3	0	4.35

Figure 2: Measure and running time for analysis of Test 1 with different variables constrained. Constraining $b1$ and $b2$ only produces an efficient analysis, whereas constraining $b3$ and either $b1$ or $b2$ only is rather inefficient, especially when compared to the analysis where all three variables are constrained.

An analysis A is better than another analysis B if it runs in less time and produces a result with higher accuracy. In such cases, A is said to be *dominant* over B [7]. However, it is possible for A to run for

Vars constrained	Measure	Analysis Time
None	3.00000E-01	2.22
b1	1.42857E-02	3.36
b2	8.18182E-02	8.30
b3	9.33333E-02	20.79
b1, b2	7.79221E-03	18.04
b1, b3	9.52381E-04	5.70
b2, b3	6.06061E-04	63.53
b1, b2, b3	0	4.88

Figure 3: Measure and running time for analysis with different variables constrained. Again, accuracies and outcomes are difficult to predict based on the individual results of constrained variables. Results from analyses constraining the variables separately does not suggest that constraining $b1$ and $b3$ would be the the most beneficial of the constrained pairs.

a longer time and produce a result with higher accuracy than B. In such a case, neither analysis is dominant over the other, and which analysis is better becomes a matter of subjective opinion. We have used the inconclusiveness measure to define a heuristic that measures our satisfaction with an analysis as a function of running time and accuracy. Having defined this heuristic, we can use the following naive algorithm to select the next constraint to be used in our incremental analyses.

1. Let C be the set of all constraints that can be added to an analysis. Let S be a set that is initially empty. Let $n = 0$
2. Spawn $|C| - n$ separate analyses, where each analysis adds a single constraint from $C - S$ to its set of constraints, S, such that no two analyses add the same constraint.
3. After all analyses have completed, select the analysis which returns the best value with respect to the heuristic.
4. If a conclusive result occurs, return the result.
5. Otherwise, add 1 to n and return to step 2.

This algorithm will return a conclusive result after at most $n+1$ increments, where n is the number of constraints that can be added to the analysis. Of course, on the ith increment, $n - i$ analyses will have to be performed. The negative consequences of this are reduced if we allow the analysis to be performed concurrently, through access to a distributed system with at least n processors. In any case, the maximum number of analyses that must be run is $(n+1)n/2$.

Fastest Time First			
Round	Vars constrained	Time	measure
1	None	5.29	.666667
2	c1	6.95	.200000
3	c1, c2	10.85	.066667
4	c1, c2, c3	16.82	.020000
5	t2, c1, c2, c3	24.48	.004800
6	t1, t2, c1, c2, c3	38.48	.000762
7	t1, t2, x, c1, c2, c3	16.67	0
	Total time	119.54	
Optimal			
Round	Vars constrained	Time	measure
1	None	5.29	.666667
2	c1	6.95	.200000
3	c1, c2	10.85	.066667
4	t2, c1, c2	17.06	.018667
5	t1, t2, c1, c2	45.50	.003806
6	t1, t1, x, c1, c2	15.58	0
	Total time	101.23	

Figure 4: Here, we see that selecting the fastest timed analysis at each iteration does not always yield the optimal (i.e. fastest to reach measure 0) anytime algorithm. Variable $c3$ is modeled unnecessarily. This suggests that other heuristics that not only consider time, but the accuracy as well, might be better suited for predicting constraints to model in attempting to produce the optimal anytime algorithm.

Without the aid of the inconclusiveness measure and a heuristic to prune out those analyses that need not be considered, a concurrent incremental analysis algorithm like the one described above could perform up to $n!$ analyses.

It is often the case that if an analysis using a set of constraints, S_1, performs better than an analysis using a set of constraints, S_2, then the analysis with a new constraint added to S_1 will perform better than the analysis with this new constraint added to S_2. The above algorithm utilizes this fact by only performing those analyses whose sets of constraints can be derived by adding a constraint to the set of constraints used in an analysis that is favored by the heuristic.

4.1 Heuristics

The performance of the anytime algorithm presented above depends on how effectively the heuristic can guide the algorithm toward analyses that 1) provide high levels of accuracy, 2) run for short amounts of time, and 3) allow for large gains in accuracy to occur in earlier increments. We selected a set of heuristics, where each heuristic produces a measure combining one or more of these three properties. A set of analyses can be compared with respect to these properties by comparing the values the heuristics produce as a result of these analyses.

Let M_{cur} be the current accuracy measure, M_{prev} be the accuracy measure from the previous increment, and t be the time it took to run the analysis that produces M_{cur}. The following are the heuristics that we considered:

FTF	Fastest Time First	Minimize t
BA	Best Accuracy	Minimize M_{cur}
MIA	Most Improved Accuracy	Maximize $M_{prev} - M_{cur}$
BUR	Best Unit Ratio	Maximize $(M_{prev} - M_{cur})/t$
BR	Best Ratio	Maximize $(1 - M_{cur})/t$
COM	Combination	Minimize $M_{cur} * t$
CR	Combination ratio	Minimize $M_{cur} * t/(M_{prev} - M_{cur})$

5 Experimentation and Results

We have experimented on 4 sample pieces of Ada83 code so that we could compare the different heuristics, where constraints were added by modeling the different variables in the code. We wish to point out that our experimentation is in its preliminary stages, and we make no claim that these 4 programs represent an accurate distribution of the domain of all programs. The traits that we believe have a strong effect on the analysis time and inconclusiveness measure are listed in figure 6. The conditional statements determine the basic shape of the CFG, and the number and enumeration size of variables determines the effectiveness of the different variable constraints that we could add to an analysis.

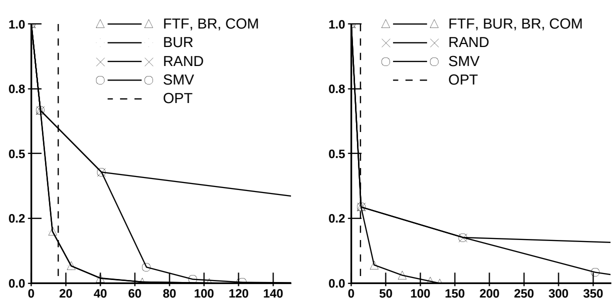

Figure 5: Sample runs using the various heuristics as well as a random strategy (RAND) and a smallest-variable-first strategy (SMV) on two different QREs. The vertical dashed line represents the optimal time in which AFLAVERS can prove the code correct (i.e. a single run with the set of constraints added that causes the analysis to reach a conclusive result the fastest).

We compared the heuristics to random selection of the next constraint to be applied, and also to the outcome obtained by choosing the next variable to constrain based on the number of values the variable type allowed the variable to take on.

Prog	LOC	Variables	while stmts : max nest	if stmts : max nest	nest depth
test1.a	49	2x2, 1x5	1 : 1	3 : 2	3
test2.a	48	2x9, 1x5	1 : 1	3 : 2	3
test3.a	161	4x3, 1x14, 1x5, 1x3	1 : 1	14 : 2	3
test4.a	229	4x3, 4x6, 4x2, 4x7	5 : 2	11 : 2	3

Figure 6: Selected attributes of the 4 test programs that we experimented with. The LOC column contains the lines of code in each program. In the variable column, mxn means that there were m variables with an enumeration size of n. In the *while* and *if* columns, m : n means that there are m statements, and the maximum nesting of such statements is n. The *nest depth* column counts the maximum nest depth of a combination of while loops and if statements.

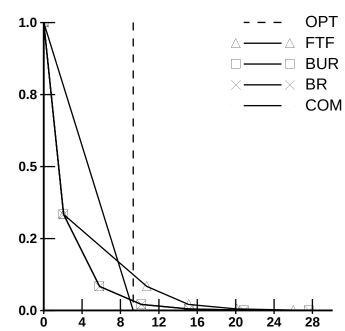

Figure 7: Sample runs using the various heuristics, showing a case where the **COM** heuristic locates a conclusive result faster than the other heuristics. Note that though the heuristic selects the conclusive result before the other heuristics do so, the curve produced satisfies fewer anytime properties. Although the other heuristics would not select the conclusive result, an actual implementation could note that a conclusive result is reached, allowing the anytime algorithm to return 0 for the measure at that point in time and thus terminate.

With the exception of the **Best Accuracy** and **Most Improved Accuracy** heuristics, we found that the heuristics usually produced similar, if not identical, orderings for the selection of constraints. **BA** and **MIA** depend only on accuracy, and often fared poorly due to the fact that they ignore analysis times. Over a complete analysis, the inconclusiveness measure changes by the fixed amount of 1, whereas time varies depending on the program, the QRE, and the order in which constraints are applied. Because of this, a heuristic that depends on time is often dominant over an algorithm that does not.

Heuristics **COM** and **CR** seem slightly more appealing than the other heuristics that involve time in that they always selected as the best analysis a conclusive analysis (see figure 7).

All of our heuristics consistently outperformed the method of selecting constraints randomly, and often obtained a conclusive result in less than half the time taken by the smallest-variable-first strategy (the strategy that models variables in order of the number of values allowed by the variable type). Also, as can be seen from the graphs, the heuristics produce nice anytime curves with the diminishing returns property.

6 Conclusion

We have shown how developing an incremental analysis into an anytime algorithm provides a measure of accuracy, which can then be used to dynamically optimize the incremental analysis. We then showed that the most successful method for selecting variable constraints to add was by performing the analysis in a concurrent setting, and using a heuristic that relied on time and possibly on the measure as well to select the next constraint to include in the analysis, based on the results obtained from the previous analysis.

Not only does posing the analysis as an anytime algorithm have the potential to speed up the algorithm, but it also broadens the scope of the use of analysis by formalizing partial results, which are generally not considered within analysis of software. Our results are preliminary in that our experimental test suite was small and does not provide a good distribution of programs. More testing needs to be performed to further verify the our results.

References

[1] George S. Avrunin, Ugo A. Buy, James C. Corbett, Laura K. Dillon, and Jack C. Wiliden, *Automated Analysis of Concurrent Systems with the Constrained Expression Toolset.* IEEE Transactions on Software Engineering, 17(11), pp. 1204-1222, November, 1991.

[2] Matthew Dwyer and Lori Clarke, *Data Flow Analysis for Verifying Properties of Concurrent Programs.* ACM SIGSOFT'94 Software Engineering Notes, Proceedings of the Second ACM Sigsoft Symposium on Foundations of Software Engineering, vol. 19, no. 5, pp. 62-75, December, 1994.

[3] Matthew B. Dwyer, *Data flow analysis for verifying correctness properties of concurrent programs.* PhD Thesis, University of Massachusetts, Amherst, 1995.

[4] Kurt M. Olender and Leon J. Osterweil, *Cecil: A Sequencing Constraint Language for Automatic Static Analysis Generation.* IEEE Transactions on Software Engineering, Vol. 16, No. 3, March, 1990.

[5] Kurt M. Olender and Leon J. Osterweil, *Interprocedural Static Analysis of Sequencing Constraints.* Transactions on Software Engineering and Methodology, Vol. 1, #1, pp. 21-52, January 1992.

[6] Richard N. Taylor and Leon J. Osterweil, *Anomaly Detection in Concurrent Software by Static Data Flow Analysis.* IEEE Transactions on Software Engineering, g(3), pp. 265-278, May, 1980.

[7] Shlomo Zilberstein, *Using Anytime Algorithms in Intelligent Systems.* AI Magazine, 17(3), pp. 73-83, Fall 1996.

[8] Shlomo Zilberstein, *Optimizing Decision Quality with Contract Algorithms*, 14th IJCAI, pp. 1576-1562, Montreal, Canada, August 1995.

[9] M.S. Hecht, *Flow Analysis of Computer Programs.* North-Holland, New York, 1977.

[10] Leon J. Osterweil, *Perpetually Testing Software.* Proceedings of the 9th International Software Quality Week, 1996.

[11] Dan Rubenstein, *AFLAVERS: An Anytime Approach to Analyzing Software Systems.* Project Report, University of Massachusetts at Amherst, Fall, 1996.