Jager: Symbolic Specification-Based Program Adjustment

Joshua Branchaud and Eric Rizzi
Department of Computer Science and Engineering
University of Nebraska-Lincoln
{jbrancha,erizzi}@cse.unl.edu

Introduction

The field of program synthesis has seen dramatic improvements over the last few years. The increasing power of SAT solvers has allowed researchers to ask more sophisticated questions and to handle larger programs than they ever have before. In short, this is an exciting time in program synthesis. With the increased power available, the field has progressed beyond the merely theoretical stage, to one that is making real contributions with even larger gains expected in the future.

For our project, we decided to implement some of the techniques described in the state of the art research in order to gain a better understanding of their power and limitations, and to investigate ways to increase the speed of these methods. The infrastructure we were eventually able to set up combined five separate steps to achieve the overall synthesis. These steps are symbolic trace gathering, trace separation, fault localization, post bug symbolic execution, and synthesis. By combining these five steps, we were able to achieve a start to finish program repair that ran almost completely automatically. While the programs that we were eventually able to repair were admittedly very small, it was exciting to see even obvious errors disappear without any oversight.

Limitations

With all of the improvements seen in the past few years, however, the synthesis community must contend with the basic limitations that the technique will likely always have to address. These can be split into two main classes. The first set of limitations has to do with the ability to verify a program as correct or not. Foremost among these are the difficulty of writing accurate specifications, the unwillingness of programmers to write these specifications, and the difficulty, in environments lacking specifications, of determining what is the intended

outcome. These fundamental limitations must be at the forefront of the mind of everyone who works in this field. They are also limiting factors to how widely these techniques can be adopted since any critical program is unlikely to use synthesized programs if the results they produce cannot be veified.

In addition to these verification concerns, there are certain computational restrictions with which researchers must contend. The term program synthesis is a bit of a misnomer, due to the fact that we are not, in fact, synthesizing an entire program. To synthesize an entire piece of software from scratch is at this point impossible for all but the smallest toy programs. The reason for this is the difference between verification and creation. Verify a program ignores the huge amount of effort that was required to even get the program close to working. The current techniques which attempt to synthesize an entire program take an iterative approach; creating a program from pieces and checking to see whether this particular combination was meaningful in any way. The sheer number of combinations with which these simplistic pieces can be combined is overwhelming. The best techniques use a genetic algorithm approach to build on the successes of past attempts, but even with this approach, all but the simplest programs are intractable. Therefore, instead of synthesizing an entire program, our technique instead relies on an underlying framework, produced by a programmer, which is assumed to be mostly correct. This assumption is based on the Competent Programmer Hypothesis, and without it, program synthesis would be too expensive an undertaking to attempt. From this framework, however, incremental improvements can be made until the entire program is proven correct.

Approach

Before continuing on to an explanation of our technique, there were several assumptions and restrictions that we made in our project in order to decrease the complexity of the programs with which we were dealing. One of the largest restrictions was the fact that we only tested programs that didn't contain any type of looping structure. These loops would have been problematic for the following reasons. First, since we generated symbolic traces via symbolic execution, we would have encountered the same problem all symbolic execution techniques do. When encountering loops that are either unbounded or require a large number of iterations to traverse, we would have to unroll these loops to a certain finite depth. This would result in an approximation of the overall program's behavior, resulting in an approximate fix. This lack of certainly about the value of a fix would be unacceptable for safety critical programs and is something that must be dealt with before program synthesis can ever be applied in an unsupervised setting. Secondly, the loops would cause a problem due to the recursive properties that would be required of a fix. Not only would the fix be required to fix a series of traces, the fix would be required to not interfere with itself for a variable number of times in order to solve the unrolled loops. Consider for example, a loop with an error that is unrolled 2 times. A fix would

have to fix the first trace. It would also have to be able to fix the second trace by combining with itself in a constructive way. This singular fix takes away the concept, seen in several papers, of incremental improvement. This would make it almost impossible to fix multiple bugs that occur within a single loop, as well as more difficult to solve a single bug.

In addition to only using programs without loops, we decided to limit the types of bugs we could fix. We assumed that we would only have to replace the right hand side of an erroneous assignment. This meant that the variable that is being written to is assumed to be correct; merely the value that is calculated for it to store is incorrect. This draws upon the Competent Programmer Hypothesis, allowing us to use the framework the programmer provided to make progress when complete synthesis would be computationally intractable. We assumed that that the types of holes we would have to fix were either constant values or single variable linear equations. While it would have been possible to calculate more difficult forms of calculations that could occur on the right hand side with the information that we collect, we didn't implement the grammars required by the solvers to answer these questions.

Finally, we assumed that there would only be a single buggy line in each program. This assumption is due to another weakness of synthesis; the necessity to make incremental progress. Synthesis cannot currently change multiple lines at once. Instead it must fix one line, see if it works, and then continue on to the others. While this approach means that it is would be possible to fix multiple independent buggy lines of code, fixing dependent bugs would be much more difficult. The reason that independent bugs can be fixed was explored in the work of Logozzo et al. who classify fixing of a program as the reduction of the number of bad traces. If the buggy lines in the program, therefore, were independent, the first fix would be retained, since it improved the program, allowing full verification upon fixing the second bug. If instead, the bug is the result of multiple lines, no iterative single line change would be able to fix the problem, and therefore the bug would go unfixed. We assumed there would be only a single buggy line in order to avoid the necessity of having to distinguish between independent and dependent errors.

Trace Generation

The first step in synthesizing a program is generating traces that describe the program's behavior. Some of the techniques we read about used concrete traces, generated from test suites. Since we assumed that the program would be accompanied by a comprehensive set of specs, we thought we would achieve a much more complete understanding of the behavior of the program with symbolic traces. Therefore, we created a listener that would output the complete set of instructions encountered along each symbolic path. These paths can be though of as being representative of an entire set of concrete traces. Therefore the amount of information that is captured with these traces is much larger, resulting in a

more precise analysis; one that would be provably correct at the end. Each of these traces is captured in SSA form, which makes manipulating them to put into the SAT solver much easier.

Trace Differentiation

In this step of our program, we add pre and post conditions to the traces that were generated in the previous step and calculate which traces followed a path of execution resulting in satisfying the post condition, and which do not. This is a two-step process. First, since we didn't included the preconditions in our listener, we must remove any traces that wouldn't be possible to follow. We do this by querying the solver PRE AND TRACE. This query is akin to asking the solver, is it possible to start with values that satisfy the precondition and execute every single instruction in the trace without encountering any contradictions. For example, if we generated the trace (x1<0, x2=x1+1) from our original exploration, but the precondition was x>0, then this trace would be completely disregarded from consideration since its behavior is undefined. For the traces that satisfy the pre-conditions, there is still one more test to do. We now must see if it is possible to start out in the pre-conditions, execute every statement in a trace, and end up in a space that is not considered acceptable, ie. end up in a state that is not described by the post condition. To do this, we ask the solver an existential question; whether it is possible to end up in an area that is not the post condition. For all those traces where this is not possible (the solver returns UNSAT), we know that these traces are safe and all possible values that satisfy their constraints will result in satisfying our post conditions. For those traces where the solver returns SAT, this means that exists at least one value that can end up outside the post-conditions. Therefore the entire trace is marked as problematic. The traces which pass the initial filter are then passed to the next stage of the synthesis process. Those where it is possible to end up in a space that is not the post condition being marked *bad*, the other being marked *good*.

Fault Localization

As discussed earlier in the limitations, it is important to isolate a very small portion of the program that can be blamed for the fault. This is where fault localization comes into the picture. In our case, we are interested in identifying a single statement that is potentially faulty. It is that faulty statement that we attempt to adjust in such a way that *fixes* the program. Like SemFix [], we borrow from the approach of Jones et al. [1] by computing a ranked list of *suspicious* statements. Unlike Jones et al. and SemFix though, we do not use traces from the runs of test cases in a test suite. Instead, we utilize the traces produced by symbolic execution.

The approach for localizing faults is to build a ranked list of the most supsicious statements in the program. We present the suspiciousness formula used for this

kind of fault localization in Equation 1.

$$susp(s) = \frac{failing(s)/totalFailing}{failing(s)/totalFailing + passing(s)/totalPassing} \quad (1)$$

Equation 1 relies entirely on the presence of a test suite that can be run to differentiate between passing and failing test cases which refer to concrete traces. It also relies on the test suite being of a certain quality with the suite having as high of a coverage metric as possible. Unlike these approaches, our approach uses *good* and *bad* symbolic traces (discussed in the previous section) to replace the passing and failing concrete traces. Thus, we present a subtle variant of the suspiciousness formula with Equation 2.

$$susp(s) = \frac{bad(s)/totalBad}{bad(s)/totalBad + good(s)/totalGood} \quad (2)$$

By using Equation 2 with all the traces through the program, we can quickly compute a list of statements ranked by suspiciousness. Instead of being tied to a test suite, the suspiciousness is based on all traces through the program and their ability to meet (or not) the specifications of the program. We believe this provides a fuller, more precise picture of what statements may be culpable. However, we do recognize that just as test suites can have low coverage, the specifications for the program may not be strong enough to say much.

Once we have a ranked list of suspicious statements, we can grab the most suspicious statement and pass it to the next phases of our technique so that an adjustment can be made. If there are multiple statements with the same suspiciousness score, we arbitrarily pick one. If a statement doesn't match the kind of statements we attempt to repair, then we grab the next statement that does fall in line. This is necessary as long as we have a limited repair grammar. Additionally, if a suitable repair can not be found for the first statement passed along, then we pass along a subsequent statement.

Pre and Post Condition Generation

Once the bug has been found, we now go through the process of isolating the traces that are associated with the fault. This is a two-part process. First, we slice all traces, both good and bad, that go through the buggy line so that anything at or below the bug no longer exists. The reason for this is that we must account for all possible ways to approach an error within the program, but also do not want the information generated after the bug to effect our synthesis. Following this, we again symbolically execute the program, this time, however, only for the paths emanating from the bug. The reason for this is due to the fact that since the bug caused an incorrect execution, it is possible that it forced a particular trace to go down the wrong path following the bug. It is also possible

that a path that was considered infeasible when executing the buggy line, now may be possible. Therefore, we want to consider all possible flows of execution emanating from the error. With the completion of this step, we have a set of top traces and a set of bottom traces that surround the error. This information is then passed to the final stage of our process, synthesis, in order to repair the program.

Synthesis

This final part of our process is by far the most intensive, requiring not only the creation of a line that will fix the program, but also verification that this fix works for all paths that are involved with the repair. The program pairs each top trace with every possible bottom trace with a series of series of OR statements. The result look something like [(T1 AND1 HOLE AND1 B1) OR (T1 AND1 HOLE AND1 B2) OR ( )...] AND [(T2 AND1 HOLE AND1 B1) OR (T2 AND1 HOLE AND1 B2) OR ( )...] AND ... The reason for this is we know that it is possible to traverse the top traces T1, T2, ..., since we are assuming that there are no errors within this part of the program. We also know they must flow through at least one of the bottom traces (Thus the boolean AND1's in the inner queries). What we do not know, however, is which bottom trace they will eventually progress down. This is the reason that we combine the prospective traces with OR's, since only one needs to hold true in order for the program to work. Finally, we combine all of the information about how the program could progress though each of these individual top traces together with AND2's. This is because we program must be able to properly execute along all possible paths that lead to the hole.

The HOLE part of the program is where the synthesis magic comes in. We are asking the solver two different questions at this stage. First, we are asking whether it is possible for some statement to bridge the gap between the top and the bottom. Second, we ask for a model of this statement that will serve as our prospective repair. If the answer to the first query is false, then we know that our grammar is not powerful enough to fill in the hole. Another way of saying this is that no statement of the form the grammar is capable of expressing can fix the program. If the answer, however, is true, then it is possible that a statement can bridge the gap. We ask the solver for a model of this statement, which can be used to construct a patch.

Unfortunately, our work is not yet done. What this model represents is an existential repair. It basically says that it is possible to start in the precondition, execute some path, and end up in an acceptable post-condition; that some value will satisfy all of the given constraints. Just like our trace separation stage, however, this assertion is not powerful enough. Instead, we want to know whether this patch will work for all values of precondition. In order to show this, we must negate the post condition, and see if it is possible to end up outside of the post condition. If the solver returns UNSAT to this query, meaning that it is

not possible to end up with a value that doesn't satisfy the post condition, then our work is done and we have both created and verified the repair. If instead the solver returns SAT, this means that it is still possible to end up in an incorrect final state. Should this happen, we must return to our original synthesis, add the information that the repair previously generated doesn't work, and try again. Failure to find a repair that works for all possible values of the precondition right away is not uncommon. Therefore, we must iterate between finding a model and seeing if that model works. This iterative process can continue either until we run out of resources, or until we find a repair that works for all traces.

It should be noted that the first query of this stage prevents us from executing this iterative process on a grammar that couldn't possibly satisfy all of the necessary constraints. It basically tells us whether a particular grammar is powerful enough to fill in the hole. We just can't learn which particular form of our grammar will solve it without iterating through models until we finally find one that works.

Future Work

There are a variety of ways in which Jager could be improved at this point. These are all future work items which we will detail in this section. The first of which would be to tie all the phases of Jager together so that they can be performed automatically. Currently, each phase needs to be manually invoked with the inputs relevant to that phase. We would like to add connectors to all of the phases so that when one phase completes, its output can be used to invoke the subsequent phase. This is mainly an engineering task whereas the remaining future work items are related to the approach.

With respect to the trace generation phase, we are interested in exploring ways in which less traces need to be generated. Our approach exhaustively generates all traces for a program. While this may be feasible for the small programs we have worked with, it could quickly becoming intractable for even medium-sized programs. Existing program repair approaches do not suffer from this kind of problem because they generate traces from test suites which remain resonable in size even for large projects. It would be interesting to see if there is some sweet spot sitting between the set of *all* traces through a program and the set of traces generated from a test suite. Ideally, this sweet spot would be less computationally intensive than generating all traces, but still provide better behavior coverage than a traditional test suite.

During the trace differentiation phase and fault localization phase, we identify *good* and *bad* traces and use them to localize faulty program statements. In approaches that generate traces from test suites, we can be reasonably sure that the traces represent interesting and relevant paths through the program based on common use cases. However, with our approach, we generate all traces, some

of which may not be common or interesting. Because of this, we believe it may be useful to consider ways of weighting certain paths through the program

Reduce size of queries via...

Removing redundant top traces

Looking for merge points between traces to reduce number of variables used in whole query

Remove impossible bottom traces based on top traces

Basically if variable unassociated with bug is constraint in top and bottom in infeasible way, then throw it out.

Look into removing variables that aren't involved in bug from consideration

Perhaps through postcondition raising

Integrate steps 1, 2 into single step

Increase complexity of grammars

Investigate dependent bugs by pairing statements

Conclusion

References

[1] James A. Jones, Mary Jean Harrold, and John Stasko. Visualization of test information to assist fault localization. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pages 467–477, New York, NY, USA, 2002. ACM.