Generating Java Unit Tests with AI Planning

Eddie Dingels
edingels@gmail.com

Timothy Fraser
tfraser@cs.umd.edu

Alex Quinn
aq@cs.umd.edu

Department of Computer Science, University of Maryland College Park, College Park, Maryland, USA

ABSTRACT

Although several efforts have applied search-based approaches to the generation of unit tests, none have demonstrated that these approaches scale beyond small programs. Our experiment compares the ability of three general-purpose AI planners to generate unit tests for Java classes with varying sizes and semantics. Although the GraphPlan planner handled most of our test classes, none of our planners could handle them all. We suggest a number of alternate search-based approaches and conclude that further experiments are needed to evaluate them.

Categories and Subject Descriptors

D.3.3 [**Sofware Engineering**]: Testing and Debugging—*Testing tools (e.g., data generators, coverage testing).*

General Terms

Reliability, Experimentation, Verification

1. INTRODUCTION

Unit testing is an effective way to find bugs in programs. Automating the generation of unit tests greatly decreases the difficulty of applying unit testing to software with many complex components. Many approaches to automation have been tried. Some approaches rely on a directed search through the space of potential test plans. These include the work of Howe and others [4] and Memon and others [6] using AI Planning and the work of Purdom [9] and Maurer [5] using formal grammars. Other approaches rely on random or exhaustive search, such as Pacheco and others' feedback method [7] and Visser and others' model-checking method [13].

Our work focuses on the use of AI planning. While previous efforts have shown that general-purpose AI planners are applicable and feasible on small problems, they have not demonstrated that this approach scales to larger programs.

WEASELTech'07, November 5, 2007, Atlanta, Georgia, USA.

We have conducted a controlled experiment to compare the time taken for three general-purpose AI planning algorithms: forward-search, GraphPlan [3], and UCPOP [8], to automatically generate unit tests for Java classes. We have designed our test classes to represent a range of semantic patterns commonly found in real Java programs. During the experiment, we increased the size and semantic complexity of our classes until the planners failed to produce a unit test plan for a class within a time bound of 5 minutes—the time in which a programmer might reasonably be expected to produce a unit test manually.

Our results show that, although all three planners were suitable for generating unit tests for trivially small classes, none were capable of handling all of the larger classes within our time bound. GraphPlan seemed the most suitable planner of the three, but even it was unable to generate unit test plans for seemingly reasonably-sized classes whose semantics required callers to perform explicit locking.

We conclude that similar experiments are needed to compare other search-based solutions. Interesting approaches might include AI planning algorithms that make use of domain-specific knowledge encoded as control rules [1] or some form of HTN planning [11, 10], as well as approaches beyond AI planning such as simulated annealing [12].

2. RELATED WORK

There have been a number of explorations of the use of AI Planning to generate unit tests. Howe and others [4] used UCPOP to generate test plans for a procedural control library for a tape library robot. They tested only a subset of the library's functions in order to reduce their domain to a size that was feasible for UCPOP. We have included UCPOP in our experiment to compare its limitations in the Java domain to that of the newer GraphPlan. Memon and others [6] represented user-visible GUI events as planning operators instead of functions or methods in order to generate GUI unit tests.

Purdom [9] and Maurer [5] explored another directed approach that required testers to describe the semantics of command languages as formal grammars. Automated test generators then generated their test plans by choosing productions from these grammars. While our AI planning approach does not require programmers to define grammars, it does require them to encode method preconditions and postconditions using specially-formatted Javadoc comments above each method. The planners use these pre- and postconditions to determine how each method effects the state of class instances as

A:

```
/**
 * @after:opened=true
 */
void open() {
   opened = true;
}

/**
 * @before:opened=true
 * @after:hasread=true
 */
void read() {
   hasread = true;
}
```

B:

```
public void testRead() {
   file.open();
   file.read();
   assert(file.hasread);
}
```

C:

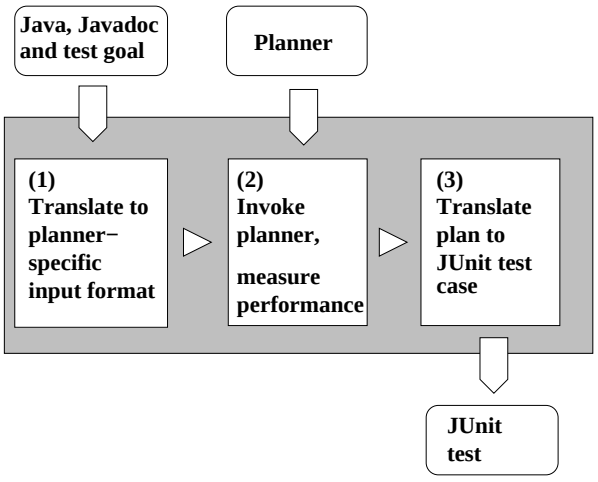

Figure 1: A: fragment of class under test, B: fragment of unit test, C: experimental framework.

they search the state space for a sequence of method invocations that reaches the goal desired by the tester. Just as there are a variety of AI planning strategies for exploring the state space, there are a variety of strategies for steering the choice of grammar productions to improve test plan generation.

Pacheco and others [7] automated Java unit test generation without planning; they built their plans by choosing methods randomly and ran the plans against the program under test to eliminate invalid plans. Their approach generates a set of arbitrary unit tests quickly, but in contrast to the planning approaches, the analyst cannot direct their generator to produce a test that will reach a specific goal.

Visser and others [13] used the Java Path Finder model checker to generate method call sequences for testing Java container classes. They used the model checker to exhaustively enumerate semantically-correct method invocation sequences of a given length. Because this enumeration could lead to intractably large numbers of sequences, they developed abstractions that could detect when the container state reached by one sequence was similar to a state already reached by some previous sequence, and explored redundant sequences no further. This sequence explosion is similar to the explosion in the number of possible states in an AI planning search space and their solution is similar to search space pruning strategies used by automated planners.

3. UNIT TESTS FOR JAVA CLASSES

Semantics complicate the problem of unit test generation for Java classes. For example, to generate a unit test that exercises some method *read*, the semantics of the class under test may require that the unit test program first call some other method *open* to reach a state where the semantics of the class permit an invocation of *read*. Consequently, automated unit test generation in this domain boils down to finding an effective automated approach to discovering semantically valid sequences of method invocations. The first method in the sequence must be one that is legal to invoke on instances of the class in their initial state. The last method in the sequence must be the one that the testers desire to exercise.

For our test cases, we encoded a specification of each method's semantics as pre- and post-conditions using Javadoc formatted comments. Figure 1A contains an example of this markup for the above *open*/*read* example. In order to keep our state spaces small, we restricted our classes to boolean programs, using only boolean variables and operators [2]. Although this boolean restriction is far too limiting for real-world programming, we were still able to represent the method semantics we desired for our tests, including constraints on invocation order and locking discipline. Figure 1B shows a fragment of the resulting unit test for the *read* method.

Figure 1C contains a diagram of our experimental framework. For each of our planners, it takes a Java class with method specifications in Javadoc and a desired unit test goal as input and (Step 1) translates that input into the notation expected by our planner, (Step 2) invokes the planner and measures its performance, and (Step 3) translates the resulting plan into a Java unit test program. The framework generates a test oracle for the unit test directly from the plan's goal conditions. We implemented our experimental framework as a Java plug-in integrated into the popular Eclipse development environment.

4. CONTROLLED EXPERIMENT

We hypothesized that it should be possible to use AI planning to generate unit test plans for Java classes, and that the newer GraphPlan would handle larger and more complex classes than the older UCPOP. To test this hypothesis, we created a collection of test classes that featured five semantic patterns found in real Java programs. For each pattern, we increased the size of the test classes until neither UCPOP nor GraphPlan could produce a plan in the alloted time of 5 minutes. Table 1 presents our experimental measurements from test runs on a typical laptop PC with a 2GHz Intel Pentium IV CPU and 778MB of RAM. We implemented a forward-search planner that chooses actions in a pseudo-random fashion mainly to test initial versions of our framework. The table includes its measurements, as well.

4.1 Semantic patterns

The first "Independent Methods" pattern has the least complex semantics: each version represents a class with a number of methods that must all be invoked, but can be invoked in any order. The second "Sequential Methods" pattern is like the first except the unit test must invoke the methods in a total order. Constraints on invocation order are not uncommon in real programs. For example, file classes often insist that a file must be "opened" before it can be "read."

The next two patterns involve "Locking". One uses classes with 8 "task" methods, the other 16. The unit test must invoke all these task methods. It may do so in any order, except that some of them require it to explicitly get a lock before the invocation and release it afterwards. The table shows results for a series of versions of the 8 and 16 method classes, each version requiring the unit test to acquire a lock for 2, 4, 8, or 16 of the task methods. These classes represent the less common but still significant semantic pattern of classes that require the caller to explicitly get and release locks to enforce mutual exclusion.

The remaining classes are based on the "AddInc" and "AddSub" patterns. They perform arithmetic on binary numbers. Each number is represented by a series of boolean variables. Each class tries to change a 2, 4, or 8-bit binary number's value from all-zeroes to all-ones using a very limited collection of arithmetic operations (+1, -3, and so on). The "AddSub" classes come in a series of versions, each one with an increasing number of available arithmetic operations, as shown by the "fan out" count in the table. These domains are meant to examine cases where an automated planner might fail to produce a unit test plan in time while a human analyst, aided by the insight that the individual booleans and methods represent numbers and arithmetic, might succeed.

4.2 Test bias

In all of the cases shown in Table 1, the goal of the unit test was to take a set of boolean variables from an all-zeroes state to an all-ones state. Table 2 shows the maximum time taken to solve versions of three of the problems with randomly-generated boolean states as goals. (For the other problems, randomized goals are not always reachable.) The maximum times for forward-search and UCPOP show that the all-1's goals in Table 1 do not represent the hardest version of the problem; some of the randomized goals were harder. On the other hand, the randomized 4-bit AddInc results seem to indicate that the all-1's goals are in fact the hardest for GraphPlan. Despite this disadvantage, our results still favor GraphPlan.

5. DISCUSSION

The Independent Methods measurements in Table 1 show that the UCPOP planner can produce effective unit test plans for Java classes with a large number of methods provided that those methods are independent—that they have no ordering or locking constraints. However, the Sequential, Locking, and arithmetic-oriented measurements show that when ordering or locking constraints are added, UCPOP can only handle classes with a very small number of methods. Of the three planners tested, UCPOP is the least suitable for generating Java unit test plans.

GraphPlan generates effective unit test plans for classes whose methods have few ordering or locking constraints, even when these classes have a large number of methods. It can handle many constrained cases that UCPOP cannot, including all of our sequential and arithmetic classes. However GraphPlan could not generate a test plan for most of the 16-method locking classes within the given time limit.

We did not expect our forward-search to be a serious contender. However, our results show that, unlike UCPOP and GraphPlan, forward-search's performance improves rather than degrades when faced with the more highly-constrained classes. Their constraints limit fanout, allowing forward-search to produce effective unit test plans even in the locking cases that defeat GraphPlan. However, in classes with many independent methods, our forward search produces plans that are far longer than those produced by the other planners. Furthermore, in the largest Independent Methods examples it cannot complete its plans within the 5 minute time bound.

Based on GraphPlan's generally excellent performance, we conclude that it is the most suitable of the three planners we examined. However, its inability to handle what seems to be a reasonable locking class leads us to imagine that there may be a better solution (see section 6). Although our results confirm that AI planning can be used to generate unit tests for Java classes, none of the planners we tested were capable of handling all of our semantic patterns.

6. CONCLUSION AND FUTURE WORK

Our experimental results confirmed that it is possible to use AI planning to generate unit tests for Java classes and showed that of the planners tested, GraphPlan fared the best. However, GraphPlan was unable to handle a reasonable case with locking semantics.

In addition to size, the semantics of a Java class are also a factor in determining performance. For example, GraphPlan was unable to solve the 16-method Locking class with 4 critical sections even though that class had fewer methods and variables than the 32-method Independent Method class it had solved earlier. Furthermore, different semantics favor different planners. Forward-search's behavior was the opposite of GraphPlan; it solved the same Locking class but failed on the Independent Methods class. No single planner fared well on all semantic patterns.

Is there a single planner that might handle all of our semantic patterns? One could imagine another experiment. For example, a serious attempt to implement a forward-search planner guided by control rules or some form of HTN planning

meth-ods	Independent Methods FwdSch	GphPlan	UCPOP	const-raints	Sequential Methods FwdSch	GphPlan	UCPOP
2	15ms	0ms	4ms	2	194ms	0ms	timeout
4	21ms	0ms	8ms	4	180ms	0ms	timeout
8	421ms	0ms	13ms	8	314ms	10ms	timeout
16	timeout	0ms	25ms	16	450ms	20ms	timeout
32	timeout	0ms	70ms	32	737ms	30ms	timeout
# crit sects	8-method Locking FwdSch	GphPlan	UCPOP	# crit sects	16-method Locking FwdSch	GphPlan	UCPOP
2	44ms	2ms	112ms	2	131ms	0ms	timeout
4	87ms	50ms	timeout	4	238ms	timeout	timeout
8	88ms	1620ms	timeout	8	415ms	timeout	timeout
16	n/a	n/a	n/a	16	346ms	timeout	timeout
fan out	2-bit AddSub FwdSch	GphPlan	UCPOP	fan out	4-bit AddSub FwdSch	GphPlan	UCPOP
1	4ms	0ms	24ms	1	7ms	0ms	timeout
2	3ms	0ms	22ms	2	12ms	60ms	timeout
4	6ms	0ms	71ms	4	18ms	0ms	timeout
8	4ms	0ms	74ms	8	18ms	0ms	timeout
	2-bit AddInc FwdSch	GphPlan	UCPOP		4-bit AddInc FwdSch	GphPlan	UCPOP
	10ms	0ms	1162ms		662ms	63000ms	timeout

Table 1: Elapsed wall-clock time for planning. 0ms indicates elapsed time less than 1ms. FwdSch entries are the average of 20 actual runs.

	8-bit Independent Methods			2-bit AddSub fanout 2			4-bit AddInc		
	FwdSch	GphPlan	UCPOP	FwdSch	GphPlan	UCPOP	FwdSch	GphPlan	UCPOP
avg:	88ms	0ms	8ms	3ms	0ms	no	124ms	171ms	no
med:	40ms	0ms	7ms	1ms	0ms	result:	77ms	180ms	result:
stdev:	107ms	0ms	2ms	4ms	0ms	timeouts	134ms	153ms	timeouts
min:	3ms	0ms	5ms	0ms	0ms	in 11 of	32ms	0ms	in 19 of
max:	348ms	0ms	11ms	12ms	0ms	20 trials	616ms	610ms	20 trials

Table 2: Statistics on problems with randomized goals after 20 trials.

might enable the tester to apply domain-specific knowledge to direct a search more effectively. For example, an HTN solution that decomposed the Locking problems into a series of get-lock, do-task, release-lock subtasks might produce effective test plans quickly.

7. REFERENCES

[1] F. Bacchus and F. Kabanza. Using temporal logics to express search control knowledge for planning. *Artificial Intelligence*, (116), 2000.

[2] T. Ball and S. K. Rahamani. Bebop: A symbolic model checker for Boolean programs. (LNCS 1885):113–130, August/September 2000.

[3] A. Blum and M. Furst. Fast Planning Through Planning Graph Analysis. *Artificial Intelligence*, (90):281–300, 1997.

[4] A. E. Howe, A. von Mayrhauser, and R. T. Mraz. Test Case Generation as an AI Planning Problem. *Automated Software Engineering*, 4(1):77–106, January 1997.

[5] P. M. Maurer. Generating Test Data with Enhanced Context-Free Grammars. *IEEE Software*, 7(4), 1990.

[6] A. M. Memon, M. E. Pollack, and M. L. Soffa. Hierarchical GUI Test Case Generation Using Automated Planning. *IEEE Transactions on Software Engineering*, 27(2):144–155, 2001.

[7] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-directed Random Test Generation. In *Proceedings of the 29th International Conference on Software Engineering*, May 2007.

[8] J. S. Penberthy and D. Weld. UCPOP: A Sound, Complete, Partial-Order Planner for ADL. In *Proceedings of the Third International Conference on Knowledge Representation and Reasoning (KR-92)*, October 1992.

[9] P. Purdom. A sentence generator for testing parsers. *BIT Numerical Mathematics*, 12(3), September 1972.

[10] E. Sacerdoti. The nonlinear nature of plans. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, 1990.

[11] A. Tate. Generating Project Networks. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, 1977.

[12] N. Tracey, J. Clark, and K. Mander. Automated program flaw finding using simulated annealing. In *Proceedings of the 1998 ACM SIGSOFT international symposium on Software testing and analysis*, 1998.

[13] W. Visser, C. S. Păsăreanu, and R. Pelánek. Test input generation for java containers using state matching. In *ISSTA '06: Proceedings of the 2006 international symposium on Software testing and analysis*, pages 37–48, New York, NY, USA, 2006. ACM Press.