Automatic Testing through Planning

Andreas Leitner[1]⋆ and Roderick Bloem[2]

[1] ETH Zürich
andreas.leitner@inf.ethz.ch
[2] Graz University of Technology
rbloem@ist.tugraz.at

Abstract. We describe a strategy to automatically test software built using pre- and postconditions. The strategy searches for valid routine calls: calls for which the preconditions are satisfied. If such calls fail (because the postcondition or another check is violated), we have found a bug. The testing strategy automatically builds a model of the software under test. The model is an abstract version of the semantics given by the pre- and postconditions. We use planning and learning to find a sequence of instructions that constructs the arguments to a valid routine call. The strategy is fully automatic and can be used to find bugs without intervention of the user. The generated test cases complement the unit tests that a designer may write by hand. We also illustrate the use of our experimental implementation on a non-trivial example.

1 Introduction

Although testing has become an important part of the software development process, facts show that the support for it is still insufficient. According to a NIST Report dated May 2002 [25], the cost of inadequate infrastructure for software testing is approximately $ 60 billion per year in the US alone.

Testing is a notoriously tedious and repetitive activity. Furthermore, once the test cases have been written, they need to be kept up to date as the target system changes. As a consequence, proper tool support is essential. Unit testing frameworks such as JUnit [1] help to reduce the cost of testing. These frameworks automate test execution. However, test cases and their oracles still have to be provided and maintained manually.

A fully automatic testing process requires the automation of both test case and oracle generation. Automating test case generation has recently been the topic of much research [4,20,28,26,14,27]. Automatic oracle generation can be done based on the specification of the system under test, if the latter is available. This idea is exploited in specification-based testing and was first formalized by Richardson et. al. [23]. For testing a software system it is crucial that the specification be

– executable,

⋆ Part of this work was done while the author was at Graz University of Technology.

– mappable – it should be possible to locate the part of the specification that is relevant for the software element under test, and
– integrated – the specification should be embedded within the source code to reduce the risk of desynchronization.

The contracts written using the *Design by Contract* software development methodology [22] have this property. Such contracts can be written in Eiffel [21], in Java (using the *Java Modeling Language* [18] or iContract [17]), in the Object Constraint Language [24,13], or in Spec# [3]. In Design by Contract, a client (the caller of a routine) is required to satisfy a supplier's (the called routine) precondition. If the precondition is not satisfied, the supplier does not guarantee anything. If the precondition is satisfied, the supplier has to fulfill its postcondition. If the supplier does not establish its postcondition, it violates the specification and thus contains a fault. These semantics enable automatic generation of test cases and oracles and thus completely automatic testing [2,12,8].

Throughout this paper we use the term contract-based testing to refer to the testing methodology which uses contracts to derive the testing oracle. A naive approach to contract-based testing, the *random testing strategy*, uses random arguments to call a routine under test and hopes to satisfy its precondition (otherwise it tries again with different arguments). If the precondition is satisfied, it hopes that the routine fails, which uncovers a bug. This process is fully automatic and precisely locates the fault. No knowledge other than the routine's signature and contract is needed.

The clear drawback of the random testing strategy is that it is likely to be unable to test routines with strong preconditions. The main contribution of this paper is a new completely automatic contract-based testing strategy. Our approach uses planning combined with learning to satisfy strong preconditions. We use the information from postconditions to select routines that are likely to change the state of the target object and arguments so that they satisfy the precondition of the routine under test. Using the information that the postconditions provide in order to satisfy preconditions is a novel technique in specification-based testing.

When using instruction sequences to create objects that satisfy preconditions the search space explodes due to the number of possible combinations of routine calls and arguments. Our approach limits the possible state transitions (and thus prunes the search space) using information from both the specification (pre- and postconditions) and implementation (learning by executing).

The proposed strategy complements manually written unit tests. Some mistakes in writing the code are not uncovered when manually writing unit tests. In particular, a developer may write precondition A, while intending precondition B. When called with unexpected input (input satisfying $A \wedge \neg B$), the implementation is unlikely to establish the postcondition. As the developer is likely to use the intended precondition when writing test cases, she will test only with data satisfying B and not with data satisfying $A \wedge \neg B$, which would reveal the fault. An automated tool will not make this mistake. For example, we found a bug in the EiffelBase library in which a function was supposed to be used only as an initializer. This constraint was not stated, and a failure was detected by the testing software when the function was used in a different setting [19].

Our strategy can be applied to any approach using embedded specification such as Eiffel, iContract, JML, OCL, and Spec#. The experimental implementation targets Eiffel for the following reasons:

– It allows us to test existing software as-is,
– Existing Eiffel libraries offer a very large code base on which we can test our approach, and
– The contracts in these libraries have not been written with automatic testing in mind and thus provide an unbiased source to test the effectiveness of our approach.

Related work on automatic test case generation includes the Korat tool [4]. Korat focuses on generation of input data and achieving high data coverage. It generates all input data up to a given bound and tests it against the class invariant. While executing the invariant, it monitors the read accesses to object fields and uses this information to prune the search space. Korat constructs input data by setting the field values directly, not by routine invocations as in our approach. It may still be hard to find combinations of input data satisfying a precondition among the data generated by Korat and, on the other hand, a large amount of data may not always be necessary. Also, Korat cannot automatically handle systems that depend on external state.

Java PathFinder has been extended to generate test input [27] using model checking and symbolic execution. Constraints on inputs that increase code coverage are generated and passed to a constraint solver for instantiation. To avoid state space explosion they use lazy initialization and conservative preconditions (i.e. preconditions that evaluate to false only if an initialized part evaluates to false). In contrast to our approach, their process is not fully automatic: the conservative precondition must be created manually.

Another approach to input value generation uses genetic programing to produce test cases that satisfy a coverage criterion [26]. The eToc tool represents test cases as chromosomes. Each chromosome is a sequence of instructions. Using recombination and mutation, new populations are generated based on a coverage-optimizing fitness criterion. The approach does not use the information provided by contracts and the oracle has to be provided manually to each generated test case.

Adele E. Howe et. al. [14] describe how techniques from the AI planning domain can be used to generate test cases. The system under test is represented as a planning problem. Instead of creating the test case manually, the user specifies what to test as a planning goal. The planner finds a path to reach the goal, which is translated to a test case. This approach is also not automatic: the planning domain and the path translation tool have to be provided manually.

2 Test Environment

The planning strategy introduced in this paper was implemented as a prototype testing strategy for TestStudio[3]. TestStudio implements the idea of contract-based testing. It was developed by Greber [12] and Ciupa [8] at based on the ideas of Arnout, Rousselot, and Meyer [2].

TestStudio is a completely automatic push-button testing environment, which tests source code as it is (without any need for special annotations or modifications). It has a graphical user interface which allows to specify several options on a feature, class, and cluster level.

The idea of contract-based testing is applicable to source code annotated with contracts in any implementation language. Therefore TestStudio can be extended to support any such language. However, we chose Eiffel as the target language, because it has native support for preconditions, postconditions, and invariants. The assertions are written by the developer not with automatic test case generation in mind, but as a design, documentation, and debugging aid. With contract-based testing those assertions serve a fourth purpose: automatic test case generation. Thus source code written in Eiffel is immediately testable without any extra effort.

Standard libraries and third party libraries are equipped with contracts. TestStudio was able to detect several bugs in unmodified production quality libraries using the random strategy [19].

3 Planning

3.1 Motivation

In order to test a routine we must satisfy its precondition. A routine with a strong precondition is unlikely to be satisfied using a random target object and random arguments. The goal of our approach is to find an object and a set of arguments that satisfy a given preconditions. To this end, we use a planning system.

Our experiments [19] have shown that the random strategy leaves routines with strong preconditions untested. For example, all features from the class *LINKED_LIST* (taken from the data structure library EiffelBase) that require *not off* are left untested. To explain the property *off* it is necessary to know that the class *LINKED_LIST* has an internal cursor, which can be moved back and forth. Using this cursor, all items of the list can be accessed. There are two special positions that the cursor can be placed on: *before* (before the first element of the list) and *after* (after the last element of the list). The predicate *off* is defined as $\mathit{before} \vee \mathit{after}$. For example, one can only delete or query the current item, when the internal cursor is *not off*.

We propose a strategy based on *planning with learning* that can find a sequence of feature calls to create objects that satisfy a strong precondition, thus

[3] `http://se.inf.ethz.ch/people/leitner/test_studio/`

making routines with such preconditions testable. For the sake of clarity, in this section we will explain the planning approach assuming that we can determine which variables are modified by a given procedure. In Section 4, we will show that this assumption is not warranted, that this complicates planning, and that we can make up for the missing information by learning about the executions.

3.2 Simple List

We will illustrate our approach using *SIMPLE_LIST*, a simplified version of EiffelBases's class *LINKED_LIST*. The list only stores how many items it has; it does not actually store any items. It also has a notion of an internal cursor. Thus, the property *not off* can still be expressed and automatic testing still fails. Listing 1.1 shows the interface of *SIMPLE_LIST*, written in Eiffel. An Eiffel **feature** can be an attribute or a routine, postconditions are indicated by **ensure**, the class invariant is indicated by **invariant** and comments start with two dashes. Preconditions do not appear in Listing 1.1, which means that they are tautologous.

Listing 1.1. Interface of *SIMPLE_LIST*

```
class SIMPLE_LIST

feature -- Queries

  index: INTEGER
    -- Index of item at current cursor position

  count: INTEGER
    -- Number of items in this list

   is_last : BOOLEAN
    -- Is internal cursor on the last item?

  off: BOOLEAN
    -- Is internal cursor not at a valid position?

feature -- Commands

  force
    -- Add an element.
 ensure
    count = old count + 1

  finish
    -- Move internal cursor to last item.
 ensure
    count > 0 implies is_last

invariant

  count + 1 > 0
  index + 1 > 0
  index < count + 2
   is_last = ((count > 0) and (index = count))
  off = ((index = 0) or (index = count + 1))

end
```

3.3 Planning Systems

Planning systems extract plans (also called solutions) from planning problems.

A planning problem is made up of three parts:

Domain Theory – Description of the world the problem is set in. The domain consists of state and actions. Actions, when applied, modify the state. The semantics of an action is defined via a precondition clause (in what situations can the action be applied) and an effect clause (how does the action change the state).

Initial State(s) – Valid initial state(s). If there are several start states, a plan needs to be applicable to each initial state.

Goal State(s) – Valid goal state(s). One of the goal states must be reached.

The Model Based Planner [6] first encoded a planning problem as a non-deterministic finite automaton (NFA). The NFA itself is represented as a Binary Decision Diagram [5]. Algorithms developed to verify properties written in Computation Tree Logic in model checking [9] are used to extract plans from the planning problem. For our implementation we used the Bifrost planner [15], because of its powerful expression language and support for non-deterministic domains. Both MBP and Bifrost use universal plans instead of the conventional sequential plans. A sequential plan is a sequence of actions to execute, whereas a universal plan is a state-action table. Looking up the current state, the table prescribes the set of applicable actions.

There are different kinds of universal plans that can be extracted from a planning problem. The usually preferred plan, the *strong solution* [7], guarantees that starting in a valid initial state, following the instructions of the solution, one of the goal states will be reached in a finite number of steps. Intuitively, a strong solution is a solution for pessimists. It is even applicable in a world were everything left uncertain by the domain description will go wrong.

There are problems where no strong solution exists, because under the most pessimistic assumptions, the goal is unreachable. In that case, there is no solution promising unconditional success, but *weak solutions* [7] can still be extracted. A weak solution is a solution for optimists. It assumes that the world cooperates to help us reach our goal. This implies that it is only guaranteed to reach a goal state if everything uncertain works in our favor. It might work in other cases, but that is not guaranteed to.

The problems described in this section are such that a strong solution exists, but in Section 4, we will show that in general, neither strong nor weak plans satisfy our requirements, and we need learning.

3.4 Problem Creation

Figure 1 shows the different steps of the simple planning strategy. The system under test is written in Eiffel and thus augmented with pre- and postconditions. This system is converted into a planning problem. The precondition defines when it is possible to call a routine and the postcondition defines the effect of the routine call. From the planning problem a strong solution is extracted. The solution represents the test case, which is then executed on an interpreter. In this section we show how to convert the *SIMPLE_LIST* class into a planning problem.

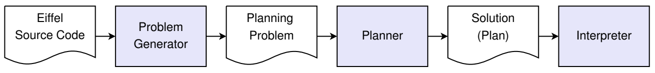

Fig. 1. Planning Strategy without Learning

Planning problems for the Bifrost planner are written in ExtNADL. An extNADL problem consists of the following sections: variables (describing the domains state space) actions (describing the possible transformations), an expression describing the initial states, and an expression for the goal states.

Listing 1.2 shows an extNADL problem representing class *SIMPLE_LIST*. The four attributes *index*, *count*, *is_last*, and *off* are translated to variables. Note that a precision of 2 bits per integer is assumed. With our approach the created model is only an approximation to the original system. These simplifications make the resulting problem smaller and thus make it feasible to extract a solution.

The routines *force* and *finish* are translated to actions. The two routines don't have preconditions, meaning they default to *True*. Note that the postcondition in the planning problem consists of the conjunction of the Eiffel postcondition and the Eiffel invariant, but the precondition of the action does not contain the invariant. This follows from the assumption that it is the duty of a routine to maintain the invariant of its object. It is assumed that the invariant is not destroyed from the outside (only in some rare circumstances does this assumption not hold). Primed variables in the problem denote the value of the variable after the execution of the action, unprimed variables denote the value before the execution. The *modifies clause* sets the variables that can be modified by an action; in this example it is set by hand.

The initial state expression is derived from the default values of the variables and the goal state expression is derived from the precondition of the feature under test.

References and compound objects do not occur in the *SIMPLE_LIST* example, but can be handled by the more general approach described in Section 5.

As mentioned in Section 3.3, the planner converts the problem to an NFA. A strong solution is extracted via a backwards breadth first search starting with the set of goal states, and attempts to construct a shortest path from an initial state to a goal state. Table 1 shows the strong solution that Bifrost extracted from this planning problem. The table tells which actions can be called in which states in order to reach one of the goal states. The values represent bit patterns; a star ("$\star$") represents both 0 and 1. Executing the solution leads to the following path:

1. The initial state is $count = 0 \wedge index = 0 \wedge \neg is\_last \wedge off$. The solution suggests to execute *force*.
2. The state now changed to $count = 1 \wedge index = 0 \wedge \neg is\_last \wedge off$. The solution suggests *finish*.
3. A goal state was reached: $count = 1 \wedge index = 1 \wedge is\_last \wedge \neg off$.

Listing 1.2. Problem for *SIMPLE_LIST*

```
VARIABLES

   nat(2) count
   nat(2) index
   bool is_last
   bool off

SYSTEM

  agt: list
  force
    mod: count
    pre: true
    eff:  (count' = count + 1) /\
      (count' + 1 > 0) /\
      (index' + 1 > 0) /\
      (index' < count' + 2) /\
      (is_last' <=> ((count' > 0) /\ (index' = count'))) /\
      (off' <=> ((index' = 0) \/ (index' = count' + 1)))

  finish
    mod: index, is_last, off
    pre: true
    eff:   ((count' > 0) => (is_last' = 1)) /\
      (count' + 1 > 0) /\
      (index' + 1 > 0) /\
      (index' < count' + 2) /\
      (is_last' <=> ((count' > 0) /\ (index' = count'))) /\
      (off' <=> ((index' = 0) \/ (index' = count' + 1)))


INITIALLY
  count = 0 /\ index = 0 /\ off /\ ~is_last

GOAL
  ~off
```

Table 1. Strong solution for *SIMPLE_LIST*

count	index	is_last	off	
00	$\star$0	0	1	force
01	$\star\star$	$\star$	1	finish
1$\star$	$\star\star$	$\star$	1	finish

The solution is what was expected: first use *force* to add an element, then use *finish* to move the cursor to point at this element.

4 Learning

4.1 Underspecification

The strong solution described in the last section could only be extracted because we set the modifies clauses by hand. It is not easy to extract such information from the source automatically. The reason for this is that the class *SIMPLE_LIST* is underspecified (in the same way that the class *LINKED_LIST* is underspecified). For example, the postcondition of the routine *finish* is *count* > 0 *implies is_last*. Because of the invariant, this is equivalent to *count > 0 implies index* $=$ *count*. The intended meaning of the postcondition is that *finish* moves the list's cursor to the last element of the list by changing *index*. However, the routine

Table 2. Weak solution for second *SIMPLE_LIST* problem

count	index	is_last	off	
$0\star$	$\star\star$	$\star$	1	force
10	$\star\star$	$\star$	1	force
$\star\star$	$\star\star$	$\star$	1	finish

does not explicitly state whether it changes *index*, *count*, or both to satisfy the postcondition. Thus, it would be valid for *finish* to set *count* to *0*. This corresponds to removing all elements — certainly not what a human would expect from a routine called *finish*.

Note that *finish* may change *index* and *off*, although they are not mentioned explicitly in the postcondition. On the other hand, it does not change *count*, although it is mentioned.

In practice, almost all routines are underspecified in the same way that *finish* is.

Since we cannot easily conclude which variables a routine changes, we will assume that every routine can change any variable as long as it satisfies the postcondition. We will first show that this makes it impossible to find a good plan, and then we will show how we can use learning in combination with planning to bring us to our goal.

4.2 Difficulties in Planning

Listing 1.3 describes the same problem as Listing 1.2 except that now every action is able to modify every variable. This planning problem can be inferred automatically from the source code. A strong solution cannot be extracted from this new problem. This is because according to the specification a valid list implementation could have *force* increase *count* and have *finish* reset both *count* and *index* to *0*. With such a list implementation, *not off* is not reachable. In the absence of a strong solution, we extract a weak solution. It is shown in Table 2. Executing this second solution is less straightforward, but works if at every step we take a random action from among the allowed ones:

1. The initial state is $count = 0 \land index = 0 \land \neg is\_last \land off$. The solution suggests to execute *force* or *finish*. Executing *finish* does not change the object state, only *force* is effective in this state. If we execute actions repeatedly, eventually *force* is executed in this state.
2. After executing *force* the state has changed to $count = 1 \land index = 0 \land \neg is\_last \land off$. The solution suggests *finish*.
3. The goal state is reached: $count = 1 \land index = 1 \land is\_last \land off$

As mentioned earlier a weak solution is not guaranteed to work in practice. If, for example, the goal of above problem is strengthened to *not off and count =*

Listing 1.3. Second *SIMPLE_LIST* Problem

```
VARIABLES

   nat(2) count
   nat(2) index
   bool is_last
   bool off

SYSTEM

  agt: list
  force
    mod: count, index, is_last, off
    pre: true
    eff:  (count' = count + 1) /\
      (count' + 1 > 0) /\
      (index' + 1 > 0) /\
      (index' < count' + 2) /\
      (is_last' <=> ((count' > 0) /\ (index' = count'))) /\
      (off' <=> ((index' = 0) \/ (index' = count' + 1)))

  finish
    mod: count, index, is_last, off
    pre: true
    eff:   ((count' > 0) => (is_last' = 1)) /\
      (count' + 1 > 0) /\
      (index' + 1 > 0) /\
      (index' < count' + 2) /\
      (is_last' <=> ((count' > 0) /\ (index' = count'))) /\
      (off' <=> ((index' = 0) \/ (index' = count' + 1)))

INITIALLY
  count = 0 /\ index = 0 /\ off /\ ~is_last

GOAL
  ~off
```

Table 3. Weak solution for third *SIMPLE_LIST* problem

count	index	is_last	off	
01	⋆⋆	⋆	⋆	force
0⋆	⋆⋆	⋆	⋆	finish
10	⋆⋆	⋆	1	finish
11	⋆⋆	⋆	⋆	finish

2, a weak solution fails every time. This new problem can be seen in Listing 1.4. The weak solution to this problem is shown in Table 3. An agent executing this solution will never reach the goal state (assuming the list implementation behaves as expected).

The solution would lead to the following execution.

1. The initial state is $count = 0 \wedge index = 0 \wedge \neg is\_last \wedge off$. The solution suggests to execute *finish*.
2. Executing *finish* does not change the state. We end up in an infinite loop without ever reaching the goal state.

Note that the postcondition of *force* does not preclude its reaching the goal state in one step; *finish* cannot do this, since it can only increase *count* by one. Thus, *force* is on the shortest path from the initial states to the goal states, but *finish* is not.

Listing 1.4. Third *SIMPLE_LIST* Problem

```
VARIABLES

   nat(2) count
   nat(2) index
   bool is_last
   bool off

SYSTEM

  agt: list
  force
    mod: count, index, is_last, off
    pre: true
    eff:  (count' = count + 1) /\
      (count' + 1 > 0) /\
      (index' + 1 > 0) /\
      (index' < count' + 2) /\
      (is_last' <=> ((count' > 0) /\ (index' = count'))) /\
      (off' <=> ((index' = 0) \/ (index' = count' + 1)))

  finish
    mod: count, index, is_last, off
    pre: true
    eff:   ((count' > 0) => (is_last' = 1)) /\
      (count' + 1 > 0) /\
      (index' + 1 > 0) /\
      (index' < count' + 2) /\
      (is_last' <=> ((count' > 0) /\ (index' = count'))) /\
      (off' <=> ((index' = 0) \/ (index' = count' + 1)))


INITIALLY
  count = 0 /\ index = 0 /\ off /\ ~is_last

GOAL
  ~off /\ count = 2
```

4.3 Planning with Learning

The above example clearly demonstrates that in cases where no strong plan can be extracted (and in practice this is true for all Eiffel systems, because of underspecification) weak plans can fail. For instance, in the presence of a routine with no postcondition (in which case the postcondition defaults to *True*) a weak plan will always assume that the routine goes from every start state to a goal state directly. We will now introduce an iterative strategy that interleaves planning with learning. In many cases this strategy can overcome the problem uncertainty presents.

The weak solution fails because of too much uncertainty in post-conditions. It is only guaranteed to work if everything uncertain will work in the favor of the plan executing agent. The learning-based strategy shown in Figure 2 tries to reduce uncertainty with each iteration by learning from previous plan executions:

1. A planning problem is created from the Eiffel source code.
2. A weak solution is extracted from the problem.
3. The solution is executed by the interpreter and the state transitions are recorded: The object state of every routine execution is recorded before (state) and after the execution (state′).
4. If the goal is reached, the process stops and the routine can now be tested. If the goal is not reached after a certain number of routine executions, the recorded state transitions are added to the problem: Let post_i be the post-condition of a routine: the new postcondition is then defined as: $\text{post}_{i+1} := \text{post}_i \wedge (\text{state} \rightarrow \text{state}')$

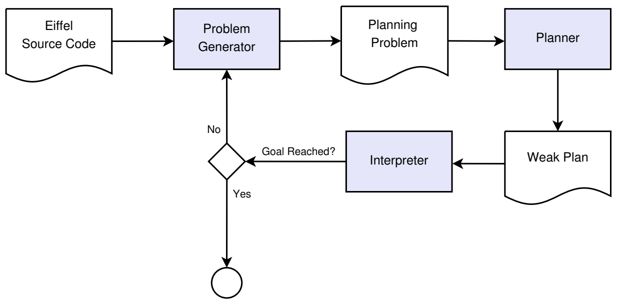

Fig. 2. Iterative Planning Strategy

5. Go to Step 2.

Learning based on state transitions assumes deterministic behavior. If a non-deterministic system is under test, learning may fail, because the same behavior may not always repeat, and learned behavior may not be correct.

Using this learning-based strategy on the *SIMPLE_LIST* problem, the initial plan is again the one from Listing 1.4. As a consequence the extracted weak plan is again the one from Table 3.

1. As before, we start from the state $count = 0 \wedge index = 0 \wedge \neg is\_last \wedge off$ and the solution suggests to execute *finish*. We record the transition $(count = 0 \wedge index = 0 \wedge \neg is\_last \wedge off) \rightarrow (count = 0 \wedge index = 0 \wedge \neg is\_last \wedge off)$. As in the last section, the sugested action leaves us in the initial state again and repeating it would cause an infinite loop.
2. After trying for a few times (to allow for nondeterminism), the new algorithm stops executing actions and creates a new plan using the recorded transition. The new plan can be seen in Listing 1.5. The weak solution for this plan is shown in Table 4. At this point, the new plan is executed.
3. The initial state is $count = 0 \wedge index = 0 \wedge \neg is\_last \wedge off$.
4. The solution suggests to execute *force*. We thus record the transition $(count = 0 \wedge index = 0 \wedge \neg is\_last \wedge off) \rightarrow (count = 1 \wedge index = 0 \wedge \neg is\_last \wedge off)$.
5. In the new state the plan suggests either *force* or *finish*. One action is chosen at random, let's assume it is *force* (the plan succeeds either way). We record the new transition: $(count = 1 \wedge index = 0 \wedge \neg is\_last \wedge off) \rightarrow (count = 2 \wedge index = 0 \wedge \neg is\_last \wedge off)$.
6. In the new state only *finish* is suggested. We again record the transition $((count = 2 \wedge index = 0 \wedge \neg is\_last \wedge off) \rightarrow (count = 2 \wedge index = 2 \wedge is\_last \wedge \neg\ off))$ and end up in the goal state.

It only took one learning cycle for the new learning-based strategy to solve the problem that was initially unsolvable using conventional planning algorithms.

Listing 1.5. Fourth *SIMPLE_LIST* Problem

```
VARIABLES

   nat(2) count
   nat(2) index
   bool is_last
   bool off

SYSTEM

  agt: list
  force
    mod: count, index, is_last, off
    pre: true
    eff:  (count' = count + 1) /\
      (count' + 1 > 0) /\
      (index' + 1 > 0) /\
      (index' < count' + 2) /\
      (is_last' <=> ((count' > 0) /\ (index' = count'))) /\
      (off' <=> ((index' = 0) \/ (index' = count' + 1)))

  finish
    mod: count, index, is_last, off
    pre: true
    eff:   ((count' > 0) => (is_last' = 1)) /\
      (count' + 1 > 0) /\
      (index' + 1 > 0) /\
      (index' < count' + 2) /\
      (is_last' <=> ((count' > 0) /\ (index' = count'))) /\
      (off' <=> ((index' = 0) \/ (index' = count' + 1))) /\
      ((count  = 0 /\ index  = 0 /\ ~is_last  /\ off ) =>
       (count' = 0 /\ index' = 0 /\ ~is_last' /\ off'))

INITIALLY
  count = 0 /\ index = 0 /\ off /\ ~is_last

GOAL
  ~off /\ count = 2
```

5 Modelling Eiffel

In the last section we have shown that a strategy that uses planning interleaved with learning can satisfy preconditions (and thus enable automatic testing) for routines where pure planning would fail. We will now discuss how the conversion algorithm that creates a planning problem from a set of Eiffel classes works. For simplicity, we have so far only dealt with one class and one object per problem.

An object-oriented language such as Eiffel is different from the extNADL problem description language:

ExtNADL has

– Only static memory (finite state),
– Actions with no arguments,
– Only natural numbers and booleans (no composite types), and
– Only limited, side-effect free and non recursive expressions.

Eiffel has

– Dynamic memory management (dynamic object creation),
– Routines with arbitrary many arguments,
– Integers, floating point values, composite objects, references, etc., and
– Multiple inheritance and dynamic binding, and
– side effects and recursive expressions.

Table 4. Weak solution for fourth *SIMPLE_LIST* problem

count	index	is_last	off	
00	00	0	1	force
01	⋆⋆	⋆	⋆	force
00	00	0	0	finish
00	00	1	⋆	finish
00	01	⋆	⋆	finish
00	1⋆	⋆	⋆	finish
01	⋆⋆	⋆	⋆	finish
10	⋆⋆	⋆	1	finish
11	⋆⋆	⋆	⋆	finish

The following algorithm is able to handle multiple classes and objects. The main difference from the planning problems we have presented so far and the ones created by the algorithm below is that the algorithm introduces an object model. Its purpose is to represent the runtime state of an object oriented system. The variables in the planning domain form an array of slots. Each slot is able to store an arbitrary Eiffel object or *Void* (meaning the slot is empty). A slot knows if it contains an object (via a boolean flag) and if so what type the object has (via a natural number representing the type id). The following describes how a planning problem is derived:

Attributes (data members) Each slot has a variable for every attribute of every supported class. The variable is only of significance if the slot is not empty and the attribute is part of the type of the object in the slot. (This model is rather space inefficient; it has been chosen for its simplicity.) Boolean variables are directly supported by extNADL, integers are mapped to natural numbers of a given precision, and references become indices into the table.

Routines (methods) are converted to actions. Instead of one action per routine, 2^n actions per routine are created. There are 2^n slots we thus create one action per slot. Those extra actions help to reduce uncertainty: we assume that an object will only change its own attributes and express this via the modifies clause (this assumption does not hold in general, but we assume that it holds in most cases). Because the modifies clauses only support a list of variables, each slot (and thus object) must have its own specialized list, which contains only those variables that belong to the current slot. The precondition clause makes sure that the target object of the action is alive and of correct type and that the Eiffel precondition is fulfilled.
If $\text{empty}_{\text{slot}}$ indicates whether the slot number slot is empty, $\text{type}_{\text{slot}}$ holds the type id of the object stored in slot number slot, type_id represents the id of the type to which the routine is applicable, and pre_{rout} represents the precondition of the routine, the complete precondition for the action is: $\neg\,\text{empty}_{\text{slot}} \wedge \text{type}_{\text{slot}} = \text{type\_id} \wedge \text{pre}_{\text{rout}}$. The effect clause of the action consists of the routines postcondition and the class invariant.

Creation Routines (constructors) are similar to regular routines. The modifies clause also contains $\text{empty}_{\text{slot}}$ and $\text{type}_{\text{slot}}$ and the effect clause contains the additional contraint $\neg\,\text{empty}_{\text{slot}} \wedge \text{type}_{\text{slot}} = \text{type\_id}$.

(Boolean) Expressions occur in several places: precondition clauses, effect clauses, initial states clause, goal states clause. Because extNADL supports neither arrays nor references, quantification needs to be emulated by a conjunction of implications. For example the Eiffel expression $ref.i = 3$, is translated to extNADL as $(ref = 0) \rightarrow (i_0 = 3) \wedge (ref = 1) \rightarrow (i_1 = 3) \wedge (ref = 2) \rightarrow (i_2 = 3) \wedge (ref = 3) \rightarrow (i_3 = 3)$ (again assuming a precision of two bits for references). Here i_j indicates the variable with the name i in the j-th slot.

In the initial states all slots are empty. The goal states require at least one slot to contain an object satisfying the precondition of the routine under test.

Our approach creates models that are approximations of the system under test, because we lose precision during problem creation and use optimistic algorithms to extract plans. Thus, we may conceivably fail to find a possible plan, and thus a possibility to test a routine. Likewise, the extracted plans may not work on the system under test, i.e., they might not establish the precondition. However, the user will only be presented with valid test cases, because the interpreter executes all infeasible plans to see if they are valid.

6 Conclusions

In this paper we have presented a novel approach to fully automatic testing. The problem that we address is the difficulty of testing routines with strong preconditions automatically. Our strategy combines planning with learning to find sequences of feature calls to create objects that satisfy such strong preconditions. First we derive a planning problem from the system under test. A planner then extracts a weak solution from this problem. The solution is executed and state transition information is recorded. If we reach the planning goal, we satisfy the precondition of the routine under test. Otherwise, we use the information gained by recoding the state transitions to reduce uncertainty in the problem and extract a new solution, which has increased chances to succeed. We have also given an example where conventional planning algorithms fail and have illustrated how our experimental implementation succeeds after just one learning step.

We would like to explore if we could generalize the learning process by using techniques similar to the ones used by Daikon [11]. Another way of removing uncertainty in postconditions is by monitoring write accesses, i.e., we could assume that attributes that are repeatedly observed to not change will never be changed by a certain routine.

Various methods have been developed to increase test coverage and should be evaluated for integration with our strategy. Korat aims towards good input data coverage. It could be combined with planning in different ways: planning can be used to find a first set of arguments satisfying a certain routine. Starting with these objects a Korat-like approach can be used to derive similar input values that still satisfy the precondition. A second way to combine both approaches could be to use Korat first and then planning. After running Korat, the planner-based approach can be used to target those spots not yet covered by Korat.

Mutation testing [16] could be used to verify both the quality of contracts and the generated test cases, as could other coverage measures.

Model checking of finite state models benefits from abstraction [10]. Better methods for abstraction should be integrated into our approach.

Automatic testing is complementary to manual unit testing. Good integration with traditional unit testing frameworks is thus desirable. With contract-based testing the quality of contracts has a great effect on the quality of derived test cases.

7 Acknowledgments

We would like to thank Bertrand Meyer for his insight and motivation for this work. We would also like to thank Ilinca Ciupa for many invaluable technical discussions and providing us Test Studio. Further we would like to thank Armin Biere and Vijay d'Silva for their feedback on our work.

References

1. JUnit. `http://www.junit.org/index.htm`, 2004.
2. K. Arnout, X. Rousselot, and B. Meyer. Test Wizard: Automatic test case generation based on *Design by Contract*$^{\text{TM}}$, draft report. `http://se.inf.ethz.ch/people/arnout/arnout_rousselot_meyer_test_wizard.pdf`, 2003.
3. M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In *CASSIS 2004: Construction and Analysis of Safe, Secure and Interoperable Smart devices*. Springer, 2004.
4. C. Boyapati, S. Khurshid, and D. Marinov. Korat: automated testing based on Java predicates. *SIGSOFT Software Engneering Notes*, 27(4):123–133, 2002.
5. R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computing*, C-35(8):677–691, Aug. 1986.
6. A. Cimatti, F. Giunchiglia, E. Giunchiglia, and P. Traverso. Planning via model checking: A decision procedure for AR. In *Third European Conference on Planning*, pages 130–142, 1997.
7. A. Cimatti, M. Pistore, M. Roveri, and P. Traverso. Weak, strong, and strong cyclic planning via symbolic model checking. *Artificial Intelligence*, 147(1-2):35–84, 2003.
8. I. Ciupa. Test Studio: An environment for automatic test generation based on *Design by Contract*$^{\text{TM}}$. Master's thesis, ETH Zurich, 2004.
9. E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, Cambridge, MA, 1999.
10. M. B. Dwyer, J. Hatcliff, R. Joehanes, S. Laubach, C. S. Pasareanu, Robby, H. Zheng, and W. Visser. Tool-supported program abstraction for finite-state verification. In *International Conference on Software Engineering*, pages 177–187, 2001.
11. M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. In *International Conference on Software Engineering*, pages 213–224, 1999.
12. N. Greber. Test Wizard: Automatic test generation based on *Design by Contract*$^{\text{TM}}$. Master's thesis, ETH Zurich, 2004.
13. A. Hamie. Towards verifying Java realizations of OCL-constrained design models using JML. In *Proc. 6th International Conference on Software Engineering and Applications*, 2002.
14. A. Howe, A. von Mayrhauser, and R. T. Mraz. Test case generation as an AI planning problem. *Automated Software Engineering*, 4(1):77–106, 1997.
15. R. M. Jensen. *Efficient BDD-Based Planning for Non-Deterministic, Fault-Tolerant, and Adversarial Domains*. PhD thesis, Carnegie Mellon University, June 2003.
16. J. Jezequel, D. Deveaux, and Y. L. Traon. Reliable objects: Lightweight testing for OO languages. *IEEE Software*, 18(4):76–83, 2001.
17. R. Kramer. iContract - the Java(tm) Design by Contract(tm) Tool. In *TOOLS '98: Proceedings of the Technology of Object-Oriented Languages and Systems*, page 295, Washington, DC, USA, 1998. IEEE Computer Society.
18. G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06i, Dept. Computer Science, Iowas State University, 2000.
19. A. Leitner. Strategies to automatically test Eiffel programs. Master's thesis, Graz University of Technology, 2004.

20. D. Marinov and S. Khurshid. TestEra: A novel framework for automated testing of Java programs. In *Proc. 16th IEEE International Conference on Automated Software Engineering (ASE)*, pages 22–34, 2001.
21. B. Meyer. *Eiffel: The Language*. Prentice Hall Object-Oriented Series. Prentice Hall, 1992.
22. B. Meyer. *Object-Oriented Software Construction*. Prentice Hall PTR, 2nd edition, 1997.
23. D. Richardson, O. O'Malley, and C. Tittle. Approaches to specification-based testing. In *Third Symposium on Software Testing, Analysis, and Verification (TAV3)*, pages 86–96. ACM Press, 1989.
24. M. Richters and M. Gogolla. On formalizing the UML object constraint language OCL. In T.-W. Ling, S. Ram, and M. L. Lee, editors, *Proc. 17th International Conference on Conceptual Modeling (ER)*, volume 1507, pages 449–464. Springer-Verlag, 1998.
25. G. Tassey. The economic impacts of inadequate infrastructure for software testing. Technical report, National Institute of Standards and Technology, 2002.
26. P. Tonella. Evolutionary testing of classes. In *International symposium on Software testing and analysis (ISSTA'04)*, pages 119–128. ACM Press, 2004.
27. W. Visser, C. S. Pasareanu, and S. Khurshid. Test input generation with Java PathFinder. In *International Symposium on Software Testing and Analysis (ISSTA'04)*, 2004.
28. T. Xie, D. Marinov, W. Schulte, and D. Notkin. Symstra: A framework for generating object-oriented unit tests using symbolic execution. In *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 05)*, pages 365–381, April 2005.