Test Case Generation for Visual Contracts Using AI Planning

Matthias Schnelte, Baris Güldali
Software Quality Lab, University of Paderborn
Warburger Str. 100, 33098 Paderborn/Germany
{mschnelte,bguldali}@s-lab.upb.de

Abstract: In this paper, we propose a novel approach for computing test case preambles using visual contracts and AI Planning. In unit testing, preambles are required for setting the class under test into a controlled state (*prestate*). The class operation can then be invoked with test inputs. In previous research, we have used model checking for computing preambles. In this paper, we show how preamble computation is conducted by AI Planning and discuss its differences to model checking.

1 Introduction

Model-based testing facilitate abstract models of system under test (SUT) or of its environment for generating test artifacts, e.g. test cases, test oracles or test execution environments. Thereby some interesting research questions arise: (1) how should test cases be selected such that a adequate *coverage* of models is reached; (2) how can we compute *preambles* for test cases such that their requirements are fulfilled; (3) how can we compute expected results for the test cases (*oracle problem*).

In previous work, we have described how visual contracts (VCs) – a visual notation of pre/-post conditions – can be used for model based testing [EGL06]. VCs aim at describing the SUTs behavior by specifying object structures before (*precondition*) and after (*postcondition*) its execution. For testing, we generate a prestate and test inputs fulfilling the precondition of VC and compute the expected test results conform to the postcondition. For computing preambles, we have proposed two techniques: (1) artificial preambles, which simulate a prestate [EEG08]; (2) natural preambles, which construct the prestate in a realistic way [EGL06]. For computing natural preambles, we have applied *model checking* techniques. Other researchers have applied constraint logic programming to solve the same problem [SCP04].

In this paper, we want to discuss the suitability of AI Planning for computing preambles and compare it to graph transformations-based model checking. We want to show that AI Planning outperforms this kind of model checking. In the next section, we give an overview on AI Planning. Sect. 3 introduces the test case generation using AI-Planning in detail. In section 4, we report on early experiences on a case study and compare the result to graph transformation-based model checking.

2 AI Planning

The problem addressed by AI Planning [GNT04] can be described as follows: how can a given initial state be transformed into a desired goal state by using a set of actions? The inputs for this problems are a domain theory and a problem description. The *domain theory* describes the set of actions changing the state. Actions have *preconditions* that must hold before its execution and *effects* that describe the state changes after the execution.

The *problem description* contains a set of conditions that describe the *initial state* and others that must hold for the *goal state*. Given the domain theory and problem description, a planner searches for a sequence of actions whose effects transform the initial state until a state is reached that conforms to the goal state. Having shortly introduced the AI Planning, we explain as next, how test cases can be generated using AI Planning.

3 Test Case Generation using AI Planning

In this paper, we deal with *quasi modal* classes, where their behavior depends on the system state, in which they are tested. While testing such classes, first the system state must be set to a controlled state (*prestate*). Then the operation under test is invoked. The changes in the system state after the test invocation are compared to the specification of the expected state. Particularly, we derive a prestate from the precondition of the visual contract (VC) which specifies the operation under test. After setting the prestate, the operation is invoked with some input parameters conforming to the prestate. Finally, the changes in the system state (*poststate*) is checked against the postcondition of the VC.

As next, we will focus on deriving and setting a prestate. For setting it, we need a *preamble*, which is a sequence of class operations. In [EGL06], we explain how a preamble can be computed by using model checking. In section 3.2, we explain, how preamble computation can be done by using AI Planning. Finally, we compare these two techniques.

3.1 Derivation of Prestates from Visual Contracts

VCs describe the behavior of an operation by specifying requirements on the system state before and after its execution [LSE05]. These requirements are called pre- and postconditions, respectively. The VC in Figure 1.a illustrates the pre- and the postconditions for an operation *commitReservationList* of a hotel booking system. The objects left to the arrow represent the precondition, and the objects right to the arrow represent the postconditions. The objects are typed over a *class diagram*, which is not shown here. The object *hr:HotelReservation* is a *multiobject*, which can address one or many objects of type *HotelReservation*. The objects with a gray underground are called negative application condition (*NAC*), which are not allowed to exist in the prestate. In our example, the customers can add one or many hotels into their reservation lists. After committing the reservation, a receipt is produced and the reservations are deleted from the reservation list.

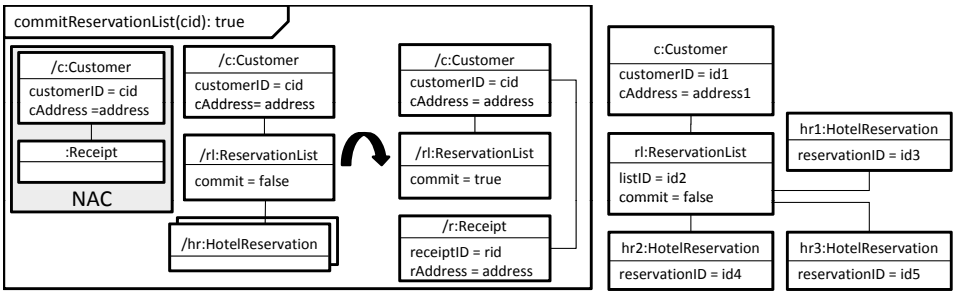

Figure 1: a) VC for operation commitReservationList, b) exemplary prestate

For testing the operation *commitReservationList*, we derive prestates from the precondition. For that, first, the objects and object links from the precondition are instantiated, then the attributes are filled with concrete values. Thereby we use structural coverage criteria [UL07] for the multiobjects and well-known data selection criteria (e.g. boundary values, equivalence classes) for the attributes [EEG08]. Figure 1.b. shows an exemplary prestate.

In [EGL06], we have shown how to compute a preamble for setting the prestate in a natural way by calling other operations. For computing the preamble, we first translated VCs into graph production rules. Then we have conducted a reachability analysis using the model checker GROOVE [KR06]. GROOVE represents the states of the system as graphs and references to graphs, instead of bit vectors as in most explicit state representing approaches. For the analysis, GROOVE first computes a graph transition system and secondly translates it to an ordinary Kripke structure. Then, a counterexample is computed by CTL model checking which represents the preamble for setting the prestate. We have experienced in our case, that GROOVE does not scale very well with the growing number of objects. Next we want to show how this problem can be tackled by using AI planning.

3.2 Translating visual contracts to PDDL

To use AI planning for the computation of preambles, the first step is to translate the VCs into a representation that can be processed by a planning tool. The most popular representation for the description of planning problems is the planning domain definition language (PDDL) [McD98], that serves as the input for most modern planners. In PDDL, a logic formalism, which is similar to first order predicate calculus, is used to describe the world state and actions. In this section, we describe a translation process that transforms VCs into an action representation noted in PDDL.

VCs and the action representation are similar in the sense that both use a pre-/postcondition formalism. Therefore, each VC is translated into one action where the left hand side of the VC is translated into the preconditions in PDDL and the right hand side is translated into the effects of the action.

For representing objects, attributes and links, the translation process introduces several types and predicates that are derived from the classes used in the VCs. For each class, a type with the name of the class is created. To express the fact that two objects *o1* and *o2*, are linked, the predicate `isLinkedTo(?o1 ?o2)` is used. Further predicates are created to represent the attributes of a class. For each attribute of a class, a predicate `attribute_` *className_attributeName* is introduced. If the attribute *a* is of type object, the predicate has two parameters: *a* and *o*. The meaning of this predicate is that *a* is an attribute of object *o*. If the attribute is of type boolean, it is sufficient to just provide the parameter *o*. If the predicate is *true* it means that *o*'s boolean attribute *a* is *true*.

Unfortunately, PDDL does not support the creation and destruction of objects. Therefore, we are using a work-around as proposed in [KB06]. All objects that can potentially be created in the future have to be specified in advance in the problem description. The creation and destruction of an object *o* is than realized by using the predicate `active(?o)`. If this predicate is *true*, the object *o* has been created. To delete the object, the predicate is set to *false*. Listing 1 shows some of the types and predicates for the hotel booking example.

```
(define (domain HotelReservation)
 (:types
   Customer - object ReservationList - object HotelBookingSystem - object
   Hotel - object HotelReservation - object Date - object Receipt - object String)

 (:predicates
   (active ?o - object)
   (isLinkedTo ?o1 - object ?o2 - object)
   (attribute_Customer_customerID ?o - Customer ?a - String)
   (attribute_ReservationList_commit ?o - ReservationList))
```

Listing 1: Type and predicate definition in PDDL.

Using these predicates, the translation process creates one action for each visual contract. As an example we provide an action for the operation `commitReservationList` as shown in Figure 1.a. The action definition can be seen in listing 2. The parameters of the action correspond to the object and attribute names in the visual contract. Lines 4 - 7 encode the positive part of the contract's precondition. Lines 8 - 9 are used to encode the negative precondition. The effects in lines 10 - 11 encode the part of the postcondition, that states that all *HotelReservation* objects that are linked to the *CustomerReservation* object are deleted. Lines 12 - 13 encode the rest of the postcondition.

After the VCs have been translated into PDDL, a planner can be used to compute a sequence of actions (*preamble*) that establish a desired system state.

3.3 Preamble computation using AI-Planning

To compute the preamble, the initial state and the goal state has to be defined. Listing 3 shows a problem definition in PDDL for the prestate shown in Figure 1.b. Lines 3 to 5 define all objects that can possibly be used. The prestate we want to establish is defined in line 9 following the same principles as we used in translating VCs to PDDL, but instead

```
(:action commitReservationList
  :parameters (?c - Customer ?rl - ReservationList ?r - Receipt
    ?cid - String ?lid - String ?address - String)
  :precondition (and  (active ?c) (active ?rl) (not (active ?r))
    (attribute_Customer_customerID  ?c ?cid)
    (attribute_Customer_customerAddress ?c ?address)
    (not (attribute_ReservationList_commit ?rl)) (isLinkedTo ?c ?rl)
    (not (exists (?receipt -  Receipt) ; The NAC
    (isLinkedTo ?c ?receipt)) ))
  :effect (and (forall (?hr - HotelReservation)
    (when (isLinkedTo ?rl ?hr)(and (not (active ?hr))(not (isLinkedTo ?rl ?hr)))))
    (active ?r) (isLinkedTo ?c ?r) (attribute_Receipt_receiptaddress ?r ?address)
    (attribute_ReservationList_commit ?rl)))
```

Listing 2: Action definition for createReservationList.

of using variables for the objects we are using constants. The *init* state is defined in line 8 as a state in which the only existing object is the hotel booking system.

Calling the planner with this problem definition results in a sequence of actions where the action's parameters are replaced with concrete objects. This sequence can be used as the preamble for setting the prestate.

```
(define (problem computePreamble)
  (:domain pddltest)
  (:objects  rl1, rl2, rl3 - ReservationList hr1, hr2, hr3 - HotelReservation
    c1, c2, c3 - Customer re1, re2, re3 - Receipt
    id1, id2, id3, id4, id5  - String hbs - HotelBookingSystem ...  )
  (:init (active hbs))
  (:goal (and (active c1)(isLinkedTo c1 rl1) (active rl1)
    (attribute_ReservationList_listID rl1 id2) ... ))
```

Listing 3: Problem definition for preamble computation.

4 Evaluation and Conclusion

We conducted some performance analysis for our approach using a realistic but small example. We used the planner LAMA [RW08] and searched for preambles with increasing complexity due to the number of objects used and the length of the required preamble. In [EGL06], we used the graph transformation-based model checker GROOVE for computing the preambles. Table 1 compares these two approaches. Column 3 shows the states generated by the planner LAMA followed by the states generated by GROOVE. GROOVE always generates the whole reachable state spaces before searching for the goal state. Therefore, the state space explodes within the number of objects. The used planner on the other hand performs a heuristic search and hence generates only a portion of the whole state space, namely that parts of the state space that are - according to the heuristic - most likely to contain the goal state. It can be seen that the planner scales very well within the problem size, whereas GROOVE does not scale at all.

We conclude that the VCs can be translated very easily and automatically to action repre-

Nr. of objects	Preamble length	States generated (LAMA/GROOVE)
4	4	14 / 10
7	5	27 / 152
10	6	44 / 3248
13	7	65 / > 20000
16	8	90 / n.a.

Table 1: Evaluation

sentation in PDDL due to the mutual use of pre/post paradigm. Furthermore, early results show the good performance of the PDDL-based approach. By stating the preamble computation problem as a planning problem in PDDL, we can reuse the mature technology and tools developed for the area of AI planning. The potential of using AI planning for test case generation has already been identified elsewhere [AZS+02]. In contrast to our work, the planner representation and the test objectives are derived *manually* from an UML model, while our approach is fully automated.

References

[AZS+02] A. Amschler Andrews, C. Zhu, M. Scheetz, E. Dahlman, and A. E. Howe. AI Planner Assisted Test Generation. *Software Quality Journal*, 10(3):225–259, 2002.

[EEG08] Jens Ellerweg, Gregor Engels, and Baris Güldali. Modellbasierter Komponententest mit visuellen Kontrakten. In *INFORMATIK 2008, Band 1, GI*, pages 211–214. GI, 2008.

[EGL06] Gregor Engels, Baris Güldali, and Marc Lohmann. Towards Model-Driven Unit Testing. In *Proc. of the Workshops at MoDELS 2006*, pages 182–192. Springer, 2006.

[GNT04] Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated Planning: Theory and Practice*. Morgan Kaufmann, 2004.

[KB06] A. Korvarsky and M. Buro. A first Look at Build-Order Optimization in Real-Time Strategy Games. In *Proceedings of the GameOn Conference*, 2006.

[KR06] Harmen Kastenberg and Arend Rensink. Model Checking Dynamic States in GROOVE. In *Proc. of SPIN 2006*, LNCS 3925, pages 299–305. Springer-Verlag, 2006.

[LSE05] Marc Lohmann, Stefan Sauer, and Gregor Engels. Executable Visual Contracts. In M. Erwig and A. Schürr, editors, *Proc. of VL/HCC'05*, pages 63–70, 2005.

[McD98] D. V. McDermott. PDDL - The Planning Domain Definition Language. Technical report, Yale Center for Computational Vision and Control, 1998.

[RW08] S. Richter and M. Westphal. The lama planner using landmark counting in heuristic search. In *Proceedings of IPC*, 2008.

[SCP04] B. Legeard S. Colin and F. Peureux. Preamble Computation in Automated Test Case Generation using Constraint Logic Programming. *STVR*, 14:213 – 235, 2004.

[UL07] Mark Utting and Bruno Legeard. *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann, 2007.