Science of Computer Programming 97 (2015) 383–404

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

Model-based mutation testing via symbolic refinement checking

Bernhard K. Aichernig [a,*], Elisabeth Jöbstl [a], Stefan Tiran [a,b]

[a] Institute for Software Technology, Graz University of Technology, Inffeldgasse 16b/2, 8010 Graz, Austria
[b] Austrian Institute of Technology GmbH, Donau-City-Straße 1, 1220 Vienna, Austria

HIGHLIGHTS

- We deal with model- and mutation-based test case generation.
- The main focus lies on optimisations of the underlying conformance check.
- We explain the construction of test cases based on the conformance check.
- We allow for non-determinism in the test models.
- We demonstrate the effectiveness of our optimisations on industrial case studies.

ARTICLE INFO

Article history:
Received 3 June 2013
Received in revised form 9 May 2014
Accepted 13 May 2014
Available online 21 May 2014

Keywords:
Model-based testing
Mutation testing
Constraint solving
Refinement
Action systems

ABSTRACT

In model-based mutation testing, a test model is mutated for test case generation. The resulting test cases are able to detect whether the faults in the mutated models have been implemented in the system under test. For this purpose, a conformance check between the original and the mutated model is required. The generated counterexamples serve as basis for the test cases. Unfortunately, conformance checking is a hard problem and requires sophisticated verification techniques. Previous attempts using an explicit conformance checker suffered state space explosion. In this paper, we present several optimisations of a symbolic conformance checker using constraint solving techniques. The tool efficiently checks the refinement between non-deterministic test models. Compared to previous implementations, we could reduce our runtimes by 97%. In a new industrial case study, our optimisations can reduce the runtime from over 6 hours to less than 3 minutes.

1. Introduction

Mutation testing is a fault-based software testing technique that receives growing interest [1]. Yet, it is still perceived as being costly and impractical. This remains a barrier to wider uptake within industry. In this paper, we report how these costs can be reduced for a particularly hard problem in mutation testing: the generation of test cases from mutated, non-deterministic models.

Mutation testing is a technique for assessing and improving a test suite [2,3]. A number of faulty versions of a program under test are produced by injecting bugs into its source code. These faulty programs are called mutants. A tester analyses if a given test suite can *kill* all mutants. We say that a test kills a mutant if it is able to distinguish the mutant from the original. The tester improves his test suite until all faulty mutants get killed.

* Corresponding author.
E-mail addresses: aichernig@ist.tugraz.at (B.K. Aichernig), joebstl@ist.tugraz.at (E. Jöbstl), stiran@ist.tugraz.at (S. Tiran).

http://dx.doi.org/10.1016/j.scico.2014.05.004
0167-6423/

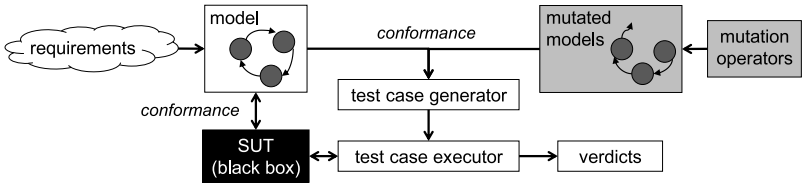

Fig. 1. Model-based mutation testing.

Unfortunately, the method is not that straightforward, because not all mutants are faulty, i.e., not all injected faults cause observable failures. For example, injected faults in dead code have no effect. Such mutants cannot be killed as they are behaviourally equivalent to the original and are therefore called *equivalent mutants*. These mutants need to be singled out by other means than testing. Traditionally, this has been done by manual inspection. However, as we demonstrate in this paper, modern program verification techniques can efficiently deal with equivalent mutants.

We focus on *model-based mutation testing*. It combines ideas from mutation testing and model-based testing. Model-based testing (sometimes also specification-based testing) is a black-box testing technique that avoids the labour of manually writing hundreds of test cases. Instead the expected behaviour of the system under test (SUT) is captured in a model. The test cases are automatically generated from this model [4]. Such models that serve as input for automatic test case generation are called *test models*. The test model is more abstract than the SUT itself and should focus on the aspects of the SUT to be tested. The technique is receiving growing interest in the embedded-systems domain, where models are the rule rather than the exception [5].

In model-based mutation testing, we view the SUT as a black box. Hence, we have no access to the source code and consequently, cannot mutate it. Therefore, we mutate a model of the SUT. This original model is assumed to be correct with respect to some properties derived from the requirements. This can be assured, e.g., via model checking. Then, given the original model and a set of mutated models, we automatically generate test cases that kill the model mutants. The generated test cases are abstract and need to be mapped to the concrete interface level of the SUT. The process is shown in Fig. 1. Note that the aim is not to test models, but to generate test cases that cover certain faults. These faults are modelled by mutating the test models. The mutation is fully automated via *mutation operators*, i.e., syntactic rules for injecting faults.

Equivalent mutants are singled out automatically. Hence, in contrast to program mutation, where we analyse a given set of test cases, here we generate a test suite that will kill all (non-equivalent) mutants. This is non-trivial, since it involves an equivalence check between original and mutated models. Since equivalence is undecidable in general, we restrict ourselves to bounded domains. How such an efficient checker can be implemented with a constraint solver is the topic and main contribution of this paper.

The situation is even more interesting when we consider non-deterministic models. In a non-deterministic model, a given (sequence of) input stimuli may cause several possible output observations. Non-determinism may be required due to non-deterministic behaviour of the SUT or because of abstraction, which characterises good test models. When comparing two non-deterministic models, an original and a mutant, equivalence is insufficient. A (pre-)order relation is needed. Refinement is such an order relation [6]. In this paper, we show how a refinement checker can effectively analyse a large number of mutated models.

Compared to our previous work [7,8], we have reduced the test case generation time by 97%. In a further case study, we demonstrate that our optimisations reduce the test case generation time from more than 6 hours to 2.2 minutes. The specific contributions of this work are the optimisation techniques, their implementation via a constraint solver, and the detailed experimental results. This article is an extension of a previous conference paper on optimisations for refinement checking [9]. It also includes optimisation techniques recently published [10]. The main novel items are the construction of actual test cases and a new larger industrial case study that evaluates all of our optimisations.

The rest of this paper is organised as follows: Section 2 introduces preliminaries, i.e., our modelling language and our notion of refinement. Section 3 explains the principles of refinement checking and Section 4 focuses on our techniques for increasing its efficiency. In Section 5, test case construction is explained, i.e., how the results obtained from the refinement check can be transformed into useful test cases. Section 6 presents our experimental results with a car alarm system and a particle counter, which is an industrial use case. We present related work in Section 7 and in Section 8, we draw our conclusions.

2. Preliminaries

2.1. Action systems

Our chosen modelling formalism are action systems [11], which are well-suited to model reactive and concurrent systems [12]. They have a formal semantics with refinement laws and are compositional [13]. Many extensions exist, but the main idea is that a system state is updated by guarded actions that may be enabled or not. If no action is enabled, the action system terminates. If several actions are enabled, one is chosen non-deterministically. Hence, concurrency is modelled in an interleaving semantics.