Bachelor Thesis Description

Testing in the presence of static fields

Patrick Emmisberger

March 11, 2013

1 Abstract

Dynamic test generation (or concolic testing) is a recent technique that allows automatic test generation using an approach where symbolic and concrete execution is combined to find input data to maximize branch coverage for the code under test. An important side-effect is the discovery of execution paths that trigger unexpected exceptions and finally the detection of bugs. However, many dynamic testing tools do not take into account the values of static fields and the fact that the execution of a simple code snippet can trigger the static initializers of the types involved. The most common approach is to keep the values of static fields unmodified over a series of tests and to assume that the initializers have already run. This can lead to non-deterministic test results (because the result depends on the order in which the tests are run) and the initializers could contain arbitrary code that is invisible to the testing tool. The goal of this thesis is to accommodate the fact that as a fundamental part of some languages, static fields are widely used (e.g. in the Windows Presentation Foundation) but have so far been ignored in the context of dynamic test generation. This is achieved by taking static fields into account and finding an approach for making calls to static initializers visible to the testing tool.

2 Problem statement

The concrete project goal is to build an extension for the concolic testing tool Pex developed by Microsoft Research. The extension should allow Pex to take static fields and initializers into account by exercising the code for the cases in which a) types are uninitialized, b) types are initialized, c) types are initialized but before the test runs their static fields have been changed (i.e. let Pex generate values for them). The following core problems should be addressed:

- Determine the set of static fields that is accessed by a test. If possible, this search should be done dynamically while the test code is executed, because a static approach would have to be very conservative and possibly yield far too many fields to handle.
- Pex should generate input values for static fields accessed by the code under test. The path condition must be extended to include information about static fields.
- If a static initializer is called during a test run, this should be visible to the Pex engine and the code of the initializer should be instrumented too. The path condition must be extended to include information about which static initializers have run.
- In order to prevent that state is carried over from one test to another, the values of static fields must be reset. However, the CLR prevents the repeated execution of static initializers. Therefore, measures must be found to circumvent this limitation, trigger the static initializers and reverse the initialization manually.

The following suggestions could be implemented as an extension to the core part:

- Allow the developer to express the state of a type respective to initialization and constrain the values of static fields as a contract precondition by extending *Code Contracts*.
- Find heuristics to determine for which tests including the static fields in the exploration is most likely to improve the dynamic branch coverage.
- Improve the analysis that detects which fields are used by implementing pruning techniques for fields that are a) initialized by a test setup/cleanup, b) only are initialized once and never change their value.
- Taking into account non-deterministic initialization by analysing program points (other than static field reads) that could be affected by a static constructor. For instance, for two classes and two fields A.X and B.Y, if there is an assertion on A.X followed by a static field read from B.Y and the static constructor of B modifies A.X, then, depending on when the static constructor is triggered, the assertion might fail.
- Enhancing Pex with decision procedures for sets.

3 Project definition

The project is subdivided into the following parts:

3.1 Reading phase

Goal: Getting familiar with the source code of Pex and related work to find possible approaches to solving the problem.
Projected completion: Middle of March 2013
Deliverables: Initial bachelor thesis presentation.

3.2 Design phase

Goal: Developing ideas on how to approach the main problems. If appropriate, build a small proof of concept for some of the ideas.
Projected completion: End of March 2013
Deliverables: Main approach for the implementation phase.

3.3 Implementation phase

Goal: Implementation of the extension and the production of a test suite to validate the former. Start in parallel to evaluate the extension in order to guide the implementation process.
Projected completion: Middle of May 2013
Deliverables: Working extension for Pex.

3.4 Evaluation phase

Goal: Evaluating the effectiveness of the developed approach using the test suite, additional tests and real-world applications.
Projected completion: Middle of June 2013
Deliverables: Evaluation results.

3.5 Writing phase

Goal: Documentation of the approach, implementation and evaluation results.
Projected completion: Middle of July 2013
Deliverables: Final report and presentation.