Kivonat

Napjaink komplex szoftverrendszerei újabb és újabb kihívások elé állítják a mérnököket. A tervezés és fejlesztés során egyre nagyobb szerepet kap az ellenőrzés, hiszen nem csak a szoftverek mérete növekszik, hanem a bennük maradó hibák száma is. A minőség javulásának és a hibák kiküszöbölésének egyik módja – többek között – a szoftverek mélyreható tesztelése a fejlesztés során. Az ismert fejlesztési modellek, metodikák túlnyomó többsége tartalmaz tesztelést magában foglaló részeket, amelyek a fenti igényeket hivatottak kielégíteni, azonban a tesztek megfelelő kiválasztása meglehetősen összetett feladat. Az automatikus tesztgenerálás megoldást jelenthet a probléma bonyolultságának enyhítésére.

Tesztek automatikus generálására meglehetősen sok technika áll rendelkezésre, ezek kiindulási pontja lehet egy működést leíró modell vagy maga a forráskód. A forráskód alapú módszerek közé sorolható a dinamikus szimbolikus végrehajtás, amelynek központi eleme a kód szimbolikus értékekkel történő végrehajtása, és annak konkrét futásokból nyert kényszerekkel való finomítása. A komplex technika hatékonyan deríti fel a bonyolult szoftverek lehetséges lefutásait, és generál tesztbemeneteket a kód ellenőrzéséhez. Az iparban történő széleskörű használatot azonban számos esetben akadályozza, hogy a feladat komplexitása miatt nem jut eredményre az algoritmus, és a mérnök nem látja át ennek okát. Ugyanakkor további hátráltató tényező lehet, hogy nincs részletes információ arról, hogy a generált tesztbemenetek a szoftver pontosan mely lefutásait fedik le. A probléma tehát a technika elméleti háttere és a mérnöki használat során felmerülő igények összeköttetésének hiányából fakad.

Dolgozatomban bemutatott munkám célja, hogy közelebb hozza egymáshoz a dinamikus szimbolikus végrehajtás technikáját és a mérnöki gyakorlatot azáltal, hogy kiegészítésekkel támogatom a tesztgenerálás hatékonyabb áttekintését. A dolgozatomban javasolok egy vizualizációs módszert, amely egy bejárási fa csomópontjaira képezi le a szimbolikus végrehajtás egyes lépéseit, megjelölve őket a hozzájuk tartozó metainformációkkal (pl. a forráskódban lévő helyük). A módszer során a bejárás áttekinthetőségének növelése érdekében 1) felbontom az egyes lefutásokhoz tartozó útvonalkényszereket, 2) osztályozom az egyes lefutási útvonalakat a generált teszthalmaz szempontjából, és 3) azonosítom, hogy egy adott lefutás hol vezet ki a tesztelt komponensből. Ezt felhasználva mutatok egy automatikus módszert, amely során a komplex, elakadást okozó hívások absztrakt értékekre cserélődnek, ami további lefutásokat biztosíthat, ezáltal újabb tesztbemeneteket eredményezhet.

A mérnöki alkalmazhatóság alátámasztásához a módszert a Microsoft Pex eszközének kiegészítéseként megvalósítottam. Az elkészült keretrendszer használhatóságát és korlátait esettanulmányokon keresztül mutatom be.

Abstract

Nowadays, complex software systems poses new challenges to engineers. Software verification techniques are getting more and more important role in the design and implementation, as the increasing size of software leads to an increasing number of software bugs. Thorough testing during the development phase is one of the key techniques for software quality improvement. Most of the known development methodologies contain testing phases, but choosing the appropriate test inputs is a rather complex task. Automatized test generation increases the efficiency of testing and the productivity.

There are many techniques for automating the generation of test inputs, their basis can be a behavioral model or the source code itself. Dynamic symbolic execution is a code-based technique: it executes the source code symbolically (with symbolic values) and refines the results with constraints derived from concrete executions. This complex algorithm effectively discovers the execution traces of complicated software, and also generates test inputs for checking the code for bugs. However, in many cases, its industrial application is hindered by the complexity and the lack of information about the results for the engineer. Additionally, the lack of information on covered paths can be also a hindering factor. In summary, the problem derives from the gap among the background of this technique and the claims arising from engineering usage.

In this report, my objective is to bring dynamic symbolic execution closer to the engineering practice by supporting the understanding of test generation. I propose a visualization technique, which maps steps of symbolic execution to nodes of a traversal tree, while adding metadata to them (e.g. source code locations, constraints). The technique supports the better understanding of the traversal by 1) decomposing the path conditions of each trace, 2) classification of execution paths in terms of generated tests and 3) identifying situations when a trace exits from the component under test. On top of this technique, I introduce an automated method to alleviate abstraction and decrease the complexity of the test generation by mapping complex function calls to abstract values. This can achieve increased coverage of the source code and provides new test inputs.

I implemented this technique as an extension for Microsoft Pex to prove the applicability in engineering practice. The usability and the limits of this framework are introduced through case studies.