Kivonat

Ahogy a beágyazott rendszerek egyre inkább életünk szerves részévé válnak, biztonságos és hibamentes működésük egyre kritikusabb a felhasználók és a gyártók számára egyaránt. A tesztelési módszerekkel ellentétben a formális verifikációs technikák nem csak a hibák jelenlétét, hanem hiányát is képesek bizonyítani, ezáltal kiváló eszközzé téve őket biztonságkritikus rendszerek verifikációjához. Ennek egy módja a már elkészült forráskód formális modellé alakítása és a modell ellenőrzése a hibás állapotok bekövetkezhetősége szempontjából.

Sajnos a forráskódból formális modellt előállító eszközök gyakran állítanak elő kezelhetetlenül nagy és komplex modelleket, így téve ellenőrzésüket rendkívül bonyolulttá és időigényessé. Munkámban bemutatok egy olyan komplex folyamatot, amely képes forráskódból formális modellt előállítani. A folyamat részeként fordítótervezésben gyakran használt optimalizációs algoritmusokat (konstans propagálás, halott kódrészletek törlése, ciklus kihajtogatás, függvények „inline”-olása) alkalmazok a bemeneten, illetve egy program szeletető algoritmus segítségével több egyszerűsített modellt állítok elő egy nagy probléma helyett. Az optimalizációk hatékonyságát és hatását mérésekkel demonstrálom.

Abstract

As embedded systems are becoming more and more common in our lives, the importance of their safe and fault-free operation is becoming even more critical. Unlike testing, formal verification can not only prove the presence of errors, but their absence as well, making it suitable for verifying safety-critical systems. Formal verification may be done by transforming the already implemented source code to a formal model and querying the resulting model's properties on the reachability of an erroneous state.

Sadly, source code to formal model transformations often yield unmanageably large and complex models, resulting in an extremely high computational time for the verifier. In this work I propose a complex workflow which provides a source code to formal model transformation, with program size reduction optimizations. These optimizations include constant propagation, dead branch elimination, loop unrolling, function inlining, extended with a program slicing algorithm which splits a single large problem into multiple smaller ones. Furthermore, I provide benchmarks to demonstrate the usability and efficiency of this workflow and the effect of the optimization algorithms on the formal model.