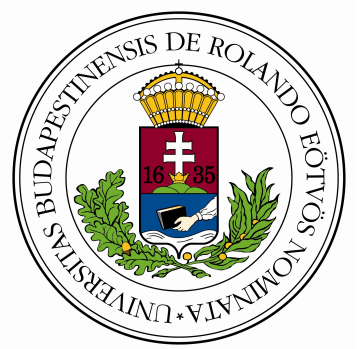
A modern szoftverfejlesztés eszközei

Pataki Norbert
patakino@elte.hu

Habilitációs tézisfüzet
2019



Tartalomjegyzék

1. fejezet

Bevezetés

Napjainkban egyre több szoftverfejlesztési feladatot kell ellátnia egyre több szoftverfejlesztő csapatnak, ami jelentősen átalakította napjaink szoftverfejlesztési feladatait, eszközeit. Újabb és újabb verziókat kell eljuttatni a végfelhasználókhoz. Ezek a feladatok egyre nagyobb terhet raknak a szoftverfejlesztő mérnökökre is. Az általuk elkövetett hibákat is minél hamarabb szeretnénk észrevenni, kijavítani.

A forráskódok tesztlefedettsége egyre nagyobb, de emellett olyan statikus elemző eszközöket is alkalmaznak, amelyek a kód végrehajtása nélkül detektálják a hibákat a szoftver forráskódjában [35]. Egyre több helyen használnak különféle statikus kódelemző eszközöket: akár túl bonyolult kód detektálására, kód konvenciók automatikus betartására, szekvenciális kódok párhuzamosításra, stb.

A kutatásainkban mi az LLVM fordítóprogram infrastruktúrára épülő Clanget használtuk a statikus elemzési feladatokhoz, mely C/C++/Objective-C nyelvek frontendje [1]. Az infrastruktúra azt jelenti, hogy nem csak fordítóként, hanem könyvtárként is használható, így a fordítás közben előálló struktúrák, adatok, szintaxisfák API-ként elérhetőek saját eszközök megvalósításához, valamint számos kapcsolódó eszközt nyújt a hatékony statikus elemzési (refaktorálási, hibakeresési, stb.) feladatok ellátásához. Nyílt forráskóddal rendelkezik, de számos nagy cég (Google, Ericsson, Apple) áll mögötte. A Clang egyik komponense a Static Analyzer (SA), mely egy olyan statikus elemző, amelyik szimbolikus végrehajtással keresi a forráskódban a hibákat.

A fejlesztők Continuous Integration (CI) szerverek webes felületein tudják nyomonkövetni a forráskód állapotát, a build folyamatok eredményeit, a teszt futtatások változásait, stb. Cloudokban indított virtuális gépeken történő automatikus deployment segítségével, akár gombnyomásra el- vagy újraindítható akár az alkalmazás, akár annak egy-egy komponense. Szükség

esetén a skálázódás horizontálisan elvégezhető, anélkül, hogy a végfelhasználók emiatt kényelmetlenségekkel szembesüljenek.

Jelen tézisfüzetben összefoglalom a doktori disszertációm megvédése óta megjelent publikációimat, kutatási témáimat. A dolgozatban három tézist fogalmazok meg, valamint további kutatásokhoz köthető eredményeimet mutatom be. Az első tézis a C++ programozási nyelvhez köthető. A C++ szabványkönyvtárának generikus mivolta miatt új jellegű hibalehetőségek adódnak a használatában, amit a fordítóprogramok nem realizálnak, valamint a könyvtár implementációi futási idejű mechanizmusok segítségével nem biztosítja ezek jelzését. Az ilyen típusú konstrukciók validációs módszereit mutatja be a tézis. A második tézis a statikus elemzésen alapuló szimbolikus végrehajtáshoz kapcsolódik. A C/C++/Objective-C nyelvekhez köthető Clang Static Analyzer nevű szimbolikus végrehajtási eszköz hibadetektáló képességeinek javítását mutatja be. A harmadik tézis egy modern szoftverfejlesztési módszertanhoz, a DevOps-hoz kapcsolódik. A DevOps megoldások, pipeline-ok és eszközök javítását ismerteti a tézis. További kutatási témáimat is bemutatom röviden.

2. fejezet

A C++ Standard Template Library hibás használatának validációs módszerei

A C++ Standard Template Library (STL) a generikus programozási paradigmán alapuló könyvtárak mintapéldája [38]. A könyvtár két alapvető komponense, a konténerek (pl. `vector`) és az algoritmusok (pl. `find_if`) külön-külön bővíthetőek, az iterátorok teremtik meg a kapcsolatot ezek között. A könyvtár további fontos komponense a funktor, amellyel felhasználói kódrészleteket lehet hatékonyan végrehajtani a könyvtáron belül. Az STL részletes felépítését a [29] bemutatja.

Az STL egy hatékony könyvtár, amellyel a hagyományos C/C++-jellegű problémák elkerülhetőek, használata jelentősen növeli a kódminőséget, az átláthatóságot pedig szabványban lefektetett azonosítók és az aszimptotikus futási idő garanciák biztosítják [25].

Ugyanakkor az STL új típusú megközelítése olyan hibalehetőséget vezetett be, amelyekre nincsenek a fordítóprogramok felkészülve, de gyakran a programozók is értetlenül állnak egy-egy hiba előtt. Meyers részletesen mutat be hibákat az STL használata kapcsán, de semmilyen eszközt nem ad a hibák jelzésére, kiszűrésére, csak informálisan írja le a helytelen használatot, illetve annak következményét [25].

Korábbi munkáim között több szoftveres megoldást biztosítottam az STL használatából adódó hibák jelzésére, jellemzően template metaprogramozási megközelítéssel [29]. A metaprogramok előnye, hogy a fordítóprogram dolgozza fel ezeket a kódokat, így megkerülhetetlen a használatuk, nem okoznak futási idő növekedést sem, viszont sok problémát nem lehet metaprogramokkal validáltatni.

Ebben a fejezetben áttekintem azokat az eredményeket, amelyek az STL

használatának validációjára szolgál és a doktori disszertációm után készültek el. Ezek egy része statikus elemzésen alapul, de készültek olyan megoldások is, amelyek speciális, nem-intruzív futási idejű megoldásként validálják a könyvtár használatát [19].

2.1. Statikus elemzési módszer

Doktori disszertációmban számos STL használatával kapcsolatos hibalehetőséget bemutattam, de nem mindegyikre tudtam eszközt adni a template metaprogramozás eszközeivel. Észrevehető, hogy sajnos a C++ fordítóprogramok nem tudnak átfogó ellenőrzéseket elvégezni az STL használatával kapcsolatban. Ezért egy olyan statikus elemző eszköz elkészítését céloztuk meg, ami a Clang fordító infrastruktúrát használja és egy-egy funkcionális megközelítésű matcher segítségével egy-egy jellegzetes hibát detektál. Nyilvánvaló előnye ennek a megközelítésének, hogy az AST-ben sokkal nagyobb kifejezőerővel lehet a validációkat implementálni, mint a template-ek példányosításán alapuló metaprogramokkal. Több olyan hibás konstrukciót megtaláló checkert sikerült megtervezni és megvalósítani, ami metaprogramokkal nem volt megvalósítható: például a konténerek polimorfikus használatát vagy 0-hoz történő méret összehasonlítást az `empty` tagfüggvény meghívása helyett [25].

A szoftverben kialakított architektúra hasznosnak bizonyult. Az alábbi 14 checkert implementáltunk az eszközben [13]:

- Bool Vector checker az `std::vector<bool>` használatára figyelmeztet, ami egy specializációja a vectornak, de nem tudja teljesíteni a C++ szabvány konténerekkel kapcsolatos elvárásait [25].

- Az `std::auto_ptr`-eket tároló konténerek (COAP-ok) rendkívül veszélyes adatszerkezetek. Az `std::auto_ptr` volt sokáig az egyetlen szabványos smart pointer C++-ban, azért volt felelős, hogy biztosítsa leallokált heap memória automatikus, élettartamhoz kötődő felszabadítását. Ugyanakkor a konténerek belső másolásainak köszönhetően a programozó konténerében a pointerek nullpointerre változhattak az `std::auto_ptr` esetében. A C++11 bevezette az `std::unique_ptr` smart pointert ennek lecserélésre [4]. A checker figyelmeztet a COAP-ok használatára.

- Rendezett asszociatív konténereken meghívott `std::count`, valamint `std::find` algoritmus felesleges teljesítmény vesztéssel jár, mivel az általános algoritmusok nem ismerik a konténerek belső reprezentációját, nem tudják a rendezettséget felhasználni. A programozóknak ilyen

esetben a `find` és `count` tagfüggvényeket érdemes használnia. A checker ellenőrzi, hogy ne használják ezeket az algoritmusokat ezeken a konténereken.

- Egy reverse iterátort a `base` metódus segítségével lehet iterátorrá konvertálni, de fontos odafigyelni arra, hogy átkonvertált iterátor nem ugyanarra a tárterületre hivatkozik, mint konverzió előtt. A checker figyelmeztet a `base` problémás használati eseteire.

- Az alkalmazkodóképes funktorok alapproblémája, hogy az átalakított (negált vagy lekötött változóval ellátott) funktor `operator()` visszatérési értékének és paramétereinek típusait nem az eredeti `operator()` definiálja, hanem ettől függetlenül megadható typedef-ek definiálják, amiket jellemzően speciális bázistípuson keresztül állít be a felhasználó, hogy ne kelljen ismernie ezeket a typedef-eket. Ez kódduplikátumot okoz, ami karbantartási vagy egyéb okokból inkonzisztenssé válhat [29]. A checker észreveszi ezt az inkonzisztenciát.

- Az allokátoroknak állapotmentesnek kell lennie a C++98/C++03-as szabvány szerint, valamint az újabb szabványban is felmerül állapottal rendelkező allokátorok használatából adódó nem-definiált viselkedés. A checker figyelmeztet az állapottal rendelkező allokátorok használatára.

- A predikátumként felhasznált funktorok problémát okozhatnak, ha van állapotuk, mivel a szabvány szerint tetszőleges számú másolat készülhet belőlük. A checker figyelmeztet, ha azt tapasztalja, hogy a predikátum állapottal rendelkezik.

- A funktor objektumokat érték szerint veszik át az algoritmusok. A polimorfizmus csak pointereken és referenciákon keresztül érhető el, így a funktor osztályokban nem kellene virtuális tagfüggvényeknek lenniük, mert hibás tervezésre utal. A checker figyelmeztet a polimorfikus funktor osztályokra.

- A konténerek üresség vizsgálatát nem célszerű a konténer `size` függvényével végezni, mert szükségtelenül lassabb lehet, mint az `empty` tagfüggvény alkalmazása. A checker figyelmeztet, ha mégis a `size`-ot használják erre a feladatra.

- Egy `std::vector` esetében a méret és kapacitás két eltérő fogalmat jelöl. A kapacitás megadja, hogy ténylegesen hány elem számára van memória allokálva, ami sokszor több lehet, mint amennyire valójában

szükség lenne, mert az törléskor nem csökken automatikusan. Korábban a konténer nem adott publikus műveletet a felesleges memóriakapacitás csökkentésére, ezért egy speciális trükkel próbálták ezt elérni [25]. A C++11 bevezette a `shrink_to_fit` tagfüggvényt, amellyel hatékonyabban, ésszerűbben lehet ugyanezt megtenni. A checker figyelmeztet, ha mégis az elavult, kényelmetlen megoldást választja valaki.

- Valójában soha nem törölnek elemet azok az algoritmusok, amelyek törléssel foglalkoznak (például: `std::remove`). Úgy működnek, hogy a konténer elejére helyezik a megmaradó elemeket és visszaadnak egy iterátort, ami a konténer logikai végét jelenti, az ezutáni elemeket `erase` tagfüggvénnyel még ténylegesen törölni kell. A checker figyelmeztet, ha ilyenkor nem törlik ténylegesen az elemeket.

- A másoló algoritmusok (például `std::copy`, `std::transform`) nem tudják garantálni, hogy van elegendő allokált tárterület, ahova másolhatnak. Ha iterátor adaptorokat használunk, akkor biztonságosan megtörténik a másolás. A checker figyelmeztet, ha nem garantált, hogy a másolatok számára lesz tárterület allokálva.

- Az STL konténerei nem polimorfikus bázisosztálynak lettek tervezve. A konténerek polimorfikus módon történő használata hiba, amit érdemes elkerülni. A checker figyelmeztet, ha egy konténert polimorfikus bázistípusként használnak.

- Egy számot értékül lehet adni egy `std::string` objektumnak az implicit típuskonverziók miatt. Ez leginkább egy programozói hiba vagy explicit érdemes lenne jelezni a kódban a tudatosságot. Ezt ellenőrzi a checker.

2.2. Include-függőségek validációja

Az STL implementációk, mint template könyvtárak, header file-okból állnak. A szükséges header file-t egy `#include` direktíva segítségével tudjuk elérhetővé tenni a saját kódunkban. Az STL implementációk rendelkeznek egy belső include-függőséggel, így előfordulhat, hogy olyan STL funkciókat is elérünk, amire mi nem írtuk ki az `#include` direktívát. Ennek kihasználása semmilyen fordítási hibát nem okoz, hiszen a funkció fordításkor elérhető. Ha a kódunkat átvisszük egy másik rendszerre, ahol másik STL implementáció található (vagy akár csak másik verzió), akkor előfordulhat, hogy a kódunk már nem fordul le. A problémát már a doktori disszertációmban felvetettem,

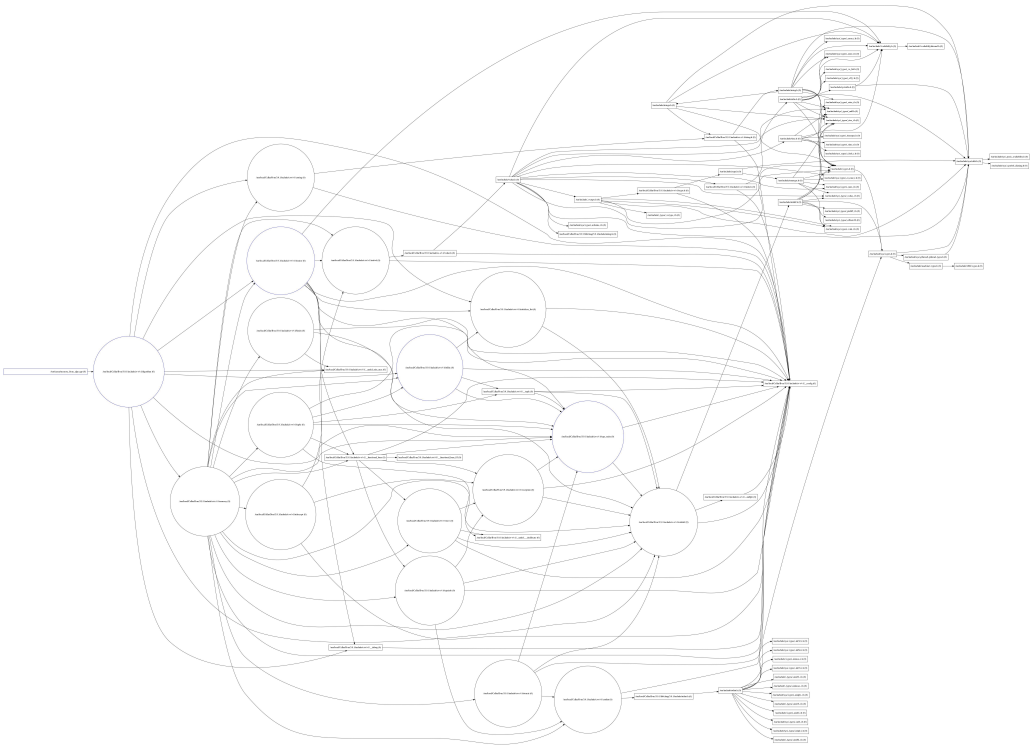
de megoldást nem adtam [29]. A helyzetet bonyolítja, hogy vannak olyan könyvtáron belüli függőségek, amelyek szükségesek, így ki is használhatóak. Erre példa, hogy a `std::map` konténer `insert` tagfüggvénye visszaad egy `std::pair`-t érték szerint, így muszáj, hogy meglegyen a `pair` definíciója, ezért a `map` fejállomány minden környezetben include-olja `utility` header file-t. Hasonló helyzet alakul ki olyan rendszereknél, ahol C és C++ fordítóprogram is rendelkezésre áll (pl. gcc, g++), ott a hagyományos C-s header file-ok (pl. `string.h`, `stdlib.h`) is elérhetőek C++ fordítás közben, nem szükséges a C++-os megfelelő fejállomány használata (`cstring`, `cstdlib`). Ezek szintén hordozhatósági problémát okoznak, ha olyan környezetre kell portolni a programunkat, ahol a C nyelv már nem támogatott.

Egy olyan Clang-alapú statikus elemzőt készítettünk, amellyel az include-dependenciák validálhatóak. Preprocesszálási fázisban egy include-függőségi gráfot épít fel a szoftver, melynek csúcsai file-ok és az élei pedig az include-olás tényét fejezik ki. Egy ilyen gráf látható a 2.1 ábrán. Ezek után a forráskód elemzésével feltérképezzük a kódban történő hivatkozásokat, végül különböző gráf elemzési mechanizmusok segítségével felkeressük a hiányzó, a szükségtelen és a megváltoztatandó include direktívákat [3].

2.3. Funktorok matematikai tulajdonságai

Az STL rendezett asszociatív konténereinek felhasználói rendezés adható át funktor osztály segítségével [38]. Ezeknek a felhasználói rendezéseknek szigorú részbenrendezési tulajdonsággal kell rendelkezniük, de ez sem fordításkor, sem futáskor nincs validálva [25]. Ha ezt a szabályt megsértjük, akkor az adatszerkezeteink inkonzisztenssé válnak és különös módon viselkednek [29]. Korábban egy futási idejű mechanizmus segítségével, limitált kiértékelési lehetőséggel működő megoldást mutattunk be [28].

Kidolgoztunk egy statikus elemzésen alapuló megoldást is: megkeressük azokat a rendezési funktor osztályokat, amelyeket STL konténerek használnak és feldolgozzuk a `operator()` belsejében lévő összehasonlítás matematikai tulajdonságait. Ehhez szükség van különböző egyszerűsítésekre, mert például a negálás befolyásolja a matematikai rendezés tulajdonságát [28]. Az elkészült eszköz működését, használatát publikáltuk [5].



2.1. ábra. Egy példa az include-függőségi gráfra

2.4. Futási idejű megoldások

Egy speciális, *gdb* debuggeren keresztüli megoldást dolgoztunk ki, amellyel futási időben lehet validálni az STL használatát [19]. Az alapkoncepció, hogy egy konkrét STL implementáció esetében jól meghatározható pontokra breakpointokat helyezve a debugger megállítja a szoftver végrehajtását. Ekkor további kiértékeléseket tudunk elvégezni például a konténerekkel, intervallumokkal kapcsolatban, majd tovább lehet engedni a szoftver végrehajtását. A belső kiértékelési logikákat gdb scriptekként valósítottuk meg [19]. Terveztünk és megvalósítottunk egy automatizáló rendszert is, amelynek következő funkciókkal rendelkezik:

- Előkészítés: a debugger elindítása megfelelő logolással, valamint a breakpointok beállításával
- Adatfeldolgozás: feldolgozza a debugger outputját, a lényegi adatokat kiszűri, elküldi az elemző komponensnek
- Adminisztráció: ez a komponens tartja karban az objektumok állapotát, hiszen két breakpoint között megváltozhatnak. Egyebek közötti elvégzi az iterátorok és konténerek adatainak frissítését, hova mutatnak az iterátorok, mekkora a konténer kapacitása, mérete, stb.
- Elemzés: ez a komponens iteratívan értékeli ki a karbantartott objektumok állapotát. Vezérli a debuggert, az adatfeldolgozó komponenst és logoló komponensnek bemenetet biztosít. Ez a komponens biztosítja, hogy a debugger folytassa a szoftver végrehajtását.
- Logok elkészítése: az STL-hez kapcsolható hibákat és figyelmeztetéseket különböző logfile-okba rendezi.

A debugger rendszerek elterjedtsége miatt ez a megközelítése elég általánosan használható. A hátránya, hogy a szoftver debug módú fordítását követeli meg, ami jelentős overheadet okoz.

Egy hasonló megközelítés alapja az *aspektus-orientált* programozás [22]. Ebben a paradigmában lehetőség van olyan szeparált modulokat definiálni, amelyek olyan összefüggő kódrészletek, ami egy AOP lehetőségeit nélkülöző nyelven írt programban a forráskód különböző részein szétszórtan helyezkedik el [26].

Az AspectC++ egy nem-szabványos aspektus-orientált bővítése a C++ nyelvnek [37]. Sokáig nem támogatta a sablonok használatát, de amikor megjelent ez a lehetőség, akkor ezt használtuk arra, hogy a validációinkat beszőhessük a szabványos STL implementációkba egy-egy végrehajtás erejéig.

2.1. táblázat. A megvalósított futási idejű validációk összehasonlítása

Validáció	gdb	AOP
Valid intervallumok	+	-
Iterátor invalidáció	+	+
Iterátorok konverziója	+	-
A `vector<bool>` használata	+	+
COAP használata	+	-
A copy-jellegű algoritmusok helyes használata	+	+
A remove-jellegű algoritmusok helyes használata	+	-
Speciális előfeltételek (pl. rendezett input intervallumok)	+	-
A `find`, `copy` algoritmus asszociatív konténereken	+	+
Állapottal rendelkező predikátumok	-	+
A `unique` algoritmus előfeltétele	+	+
Szigorú részbenrendezés	-	+

Amikor az AspectC++ elérte azt, hogy támogassa a sablonokba történő szövést, akkor megvizsgáltuk, hogy mit adhat az aspektus-orientált megközelítés az STL használatának validálásakor [2]. Az aspektusok és az aspektusszövő átveszi a feladatot a debuggerről és breakpointokon történő megállástól, így ez a megoldás több szempontból is gyorsabb: nem igényel debugger módú fordítást, valamint nem kell futási időben kezelni a breakpointokon történő megállást, mint a debugger esetében [2]. Az aspektusszövő már fordítási időben beszövi a futási idejű ellenőrzéseket. A leimplementált ellenőrzéseket a 2.1 táblázat mutatja be a két megközelítés kapcsán. Ez alapján a debugger-szintű megoldás tekinthető a kényelmesebbnek, de ezt nem az elméleti limitációk biztosítják.

Az STL hibás használatának validációs módszerei tézishez tartozó publikációim: [2, 3, 5, 13].

3. fejezet

Szimbolikus végrehajtás a Clang Static Analyzer eszköz segítségével

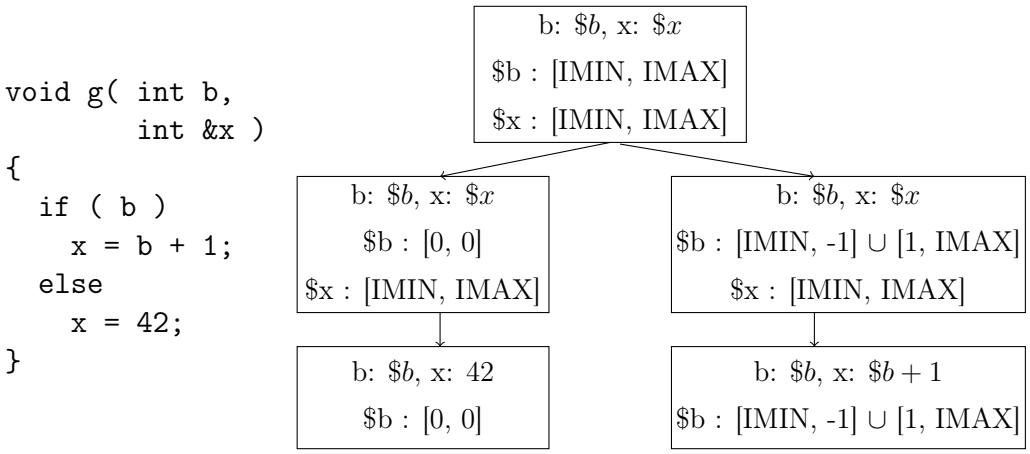
A szimbolikus végrehajtás egy olyan statikus elemzési módszer, melyben változók értékét egy szimbolikus értékkel helyettesítjük. A program végrehajtásának a szimulálása közben minden változóra, ami életben van egy adott pontján a programnak, rendelkezünk az adott változóra érvényes megkötésekkel is (például `x != 0`).

A Clang fordító része a Static Analyzer, ami a fordítóprogram által generált szintaxisfát és Control Flow Graph-ot felhasználva képes szimbolikus végrehajtást végezni a kódbázison [16]. Ehhez egy úgynevezett *exploded graph*-ot épít fel [31]. Erre egy példa látható a 3.1 ábrán. Mivel a szimbolikus végrehajtás tipikusan jelentősen hosszabb ideig tart, mint a fordítási procedúra, azért a Static Analyzer futása nem része a fordítási folyamatnak, a fordítót speciális paraméterezéssel kell meghívni, ha a fejlesztő futtatni akarja a Static Analyzert [12].

Az elemzés hatóköre jelentős hatással van az elemzés pontosságára, ezért az alábbi kutatásokkal járultunk hozzá a Clang Static Analyzer hatékonyságának javítására.

3.1. Gyorsan elemezhető modellek

Sajnos a C++ nyelv fordítási modellje nem segíti elő a kényelmes kódelemzést. Egy fordítási egységnek nevezünk egy olyan egységet, amit a fordítóprogram egyben dolgoz fel. A C++ esetében egy fordítási egység az a fordítandó C++ fájl és az általa importált (include-olt) header állományok



3.1. ábra. Egy példa a szimbolikus végrehajtás közben használt exploded graph adatszerkezetre

tranzitív lezártját jelenti. A probléma forrása, hogy általában egy header állomány több fordítási egységbe is bekerül, ezért az ilyen állományok tartalma több alkalommal is fordításra kerül. Ez jelentős többletmunkát jelent a statikus ellenőrző eszközök részére is. Bizonyos esetekben lehetősége adódhat az eszköznek egy header állomány elemzése utáni belső reprezentációját szerializálni egy másik állományba és ezt a szerializált állományt használni a továbbiakban a header állomány helyett. Ezzel sok többletmunka megspórolható, viszont a megvalósítás nem egyszerű feladat. Egyrészt különböző fordítási egységekben különböző paraméterekkel kerülhet egy header fordításra, ami befolyásolhatja a headerben található kód szemantikáját. Továbbá a headerek tartalmazhatnak template definíciókat, amik más és más fordítási egységek esetén más és más paraméterekkel kerülnek kifejtésre [12].

Felkészítettük a Clang Static Analyzert arra, hogy külön fordítási egységben lévő függvényekről C/C++ nyelven írt modellek (summary-k) segítségével tudjon érvelni. A modell kódja a függvény elő- és utófeltételeit kezeli az implementációs részletek elhagyásával [7]. Tehát amikor külső fordítási egység funkciók meghívása miatt az elemzés megakadna, akkor a modellek kódját használja a Clang Static Analyzer szimbolikus végrehajtó modul. Ennek előnye, hogy a modellek leírásához nem kell egy új nyelvet megtanulni és alkalmazni, valamint algoritmikusan megadható egy mechanizmus, amivel a modell megvalósítható. További kedvezősége a megoldásunknak, hogy az include-függőségektől meg lehet szabadulni, így gyorsítva az eljárást. A bemutatott eljárásunk hasznos jellemzője az is, hogy a Static Analyzer checkereit nem érinti az a módosítás, hogy a modellek segítségével részletesebb elemzéshez jutunk [14].

3.2. Fordítási egységeken átívelő elemzés

Az eredetileg egymenetes Static Analyzert felkészítettük egy speciális kétmenetes elemzésre. Ebben a megoldásban az első menetben a fordítási adatbázis file és a forráskód alapján készül egy indexfile, amiben benne lesz az, hogy melyik függvény definíciója melyik fordítási egységben található. Megtörténik a forráskód parse-olása, az AST-k felépítése és bináris formátumban történő szerializálása. A második menetben az összes fordítási egységet elemzi az SA. Amikor egy olyan függvényhíváshoz jut, ahol a meghívott függvény kódja nincs az aktuális fordítási egységben, akkor az indexfile alapján megkeresi, hogy hol vannak a meghívott függvényhez tartozó szerializált AST részletek. Ekkor az aktuális fordítási egységhez tartozó, memóriában lévő AST-t össze kell fésülni a lemezről felolvasott függvényhez tartozó AST részlettel, amelyek saját szimbólumtáblával rendelkeznek, az egyes felhasznált típusok külön-külön mögöttes reprezentációt tartalmaznak és a kódpozíciókat is külön manager kezeli. Jelentős járulékos részfeladat volt az ASTMatcher könyvtár felkészítése erre az összefésülési feladatra.

Könnyen válik költségessé az AST-k betöltése és összefésülése, ezért gyorsítási mechanizmust dolgoztunk ki: amikor másodjára igényel egy külső AST darabot az elemzés, akkor már azt az AST részletet a memóriában hagyjuk.

A CTU megoldás teljesítményét nyílt forráskódú projekteken validáltuk [21]. A megoldás a lefedettséget, teljesítményt és a találatok minőségét tekintve megfelelő, a találatok száma jelentősen megnőtt [20].

3.3. Élettartam hibák felderítésének javítása típusok kategorizálásával

A C++ összetett programozási nyelv. Támogatja az értékek, C-ből származó raw (nyers) pointerek, smart pointerek, iterátorok, bal- és jobbérték referenciák használatát is. Az eltérő élettartam- és láthatósági szabályok miatt nem nehéz csellengési problémákat okozni a kódban – egy pointer vagy referencia egy olyan tárterületre hivatkozik, ahol már nincs tényleges érték. A fordítóprogramok nem jelzik ezt hibaként, jellemzően futási idejű hibát okoznak. Statikus elemzők általában kezelni tudják azt az esetet, amikor nyers pointerek vagy referenciák csellengenek el. Amikor ez egy felhasználói típuson vagy egy API-n keresztül történik meg, akkor már nem tudják jelezni [17].

Kidolgoztunk egy statikus elemzésen alapuló eszközt, ami a felhasználói típusokat képes kategorizálni. Az alábbi kategóriákat támogatja az eszköz: tulajdonos, pointer, aggregátum és érték. A 3.1 táblázat bemutat egy-egy

3.1. táblázat. Példák az egyes kategóriákra

Kategória	Példa típus
Tulajdonos	`std::string`
Pointer	`std::string::iterator`
Aggregátum	`struct complex { double re, im; };`
Érték	`std::ostream`

példát kategóriánként. A kategorizálás alapján hatékonyabb, átfogóbb élettartam hibákat tud realizálni a statikus elemző rendszer [17].

Az alábbi tulajdonságokkal bíró típusokat tekintjük tulajdonosnak:

- Minden olyan típust, ami teljesíti a szabványos konténer elvárásokat és rendelkezik egy felhasználói destruktorral.
- Minden olyan típus, ami rendelkezik egy túlterhelt unáris `operator*` művelettel és felhasználói destruktorral.
- Minden olyan típust, ami publikusan származik egy tulajdonos kategóriájú típusból.

Az alábbi tulajdonságokkal bíró típusokat tekintjük pointereknek:

- Minden olyan típust, ami kielégíti a szabványos *iterátor* elvárásokat.
- Minden olyan típust, ami kielégíti a *range* conceptet.
- Minden triviálisan másolható, dereferálható típust.
- Minden olyan típust, ami publikusan származik egy pointer kategóriájú típusból.

Ökölszabályként elmondható, hogy ha egy típus rendelkezik felhasználói destruktorral, akkor azt nem kategorizáljuk pointernek, hiszen nagy valószínűséggel ebben a műveletben felszabadítja a tulajdonában lévő erőforrást [38].

Az alábbi szabályokkal általánosítottuk a Clang statikus elemzőjében az élettartam ellenőrzéseket:

- Egy függvény egy olyan pointernek kategorizált adattal tér vissza, ami a függvényben lokálisként vagy temporálisként létrehozott tulajdonosból lett létrehozva, akár konstruktorhívással, akár implicit konverzióval.
- Egy függvény olyan lokális vagy temporális tulajdonos objektumból pointernek kategorizált adatot ad vissza, ami a `begin`, `end`, `std::*::data`, `std::*::c_str` műveletekkel jött létre.
- Tulajdonosból létrehozott pointernek kategorizált adattag létrehozását (akár konstruktorhívással, akár implicit konverzióval, akár egyéb, ismert metódus segítségével).
- Egy `new` kifejezéssel létrehozott, heapre mutató pointer, ami egy temporális, tulajdonosnak kategorizált objektumra mutatna
- Bármilyen pointernek kategorizált a stacken vagy globális tárterületen, ami egy temporális, tulajdonosnak kategorizált objektumra mutatna.

Az implementációs egyszerűsítések ellenére a megoldásunk jól teljesített. A konverzatív megközelítés miatt rendkívül alacsony fals pozitív arányt sikerült kimérni. A pontosítások után gyakorlatilag nem maradt fals pozitív és a félrekategorizálásokat is sikerült megszüntetni. Emellett új, korábban nem detektált hibákat is sikerült megtalálni éles kódbázisban [17].

A tézishez kapcsolódó publikációim: [14, 17, 20, 21].

4. fejezet

A DevOps megközelítés

A modern szoftverfejlesztés segítségével egyre komplexebb alkalmazások készülnek. Az egyre komplexebb alkalmazásokhoz egyre több fejlesztő munkájára van szükség, amit valahogyan koordinálni kell [30]. A komplex alkalmazások beüzemelése, upgrade-je, valamint folyamatos üzemeltetése is egyre nagyobb erőforrás-igénnyel rendelkezik. A párhuzamos munkavégzés sok problémát vetett fel, amire az idők folyamán egyre bővülő eszközkészlettel igyekeznek egyszerűsíteni a fejlesztők: verziókövető rendszerekben tárolják a forráskódot, Continuous Integration (CI) szerverek folyamatosan értékelik, hogy a forráskód milyen állapotban van: lefordul-e, sikeresen lefutnak-e a tesztek, stb. [33]. A Continuous Delivery (CD) rendszerek ezen túlmenően az automatizált release készítési és deployment lehetőségével segítik a fejlesztést és a beüzemelést. A DevOps módszertan ennek a legátfogóbb kezelése, ahol a verziózott kódtól változásától a fordítás, statikus elemzés, átfogó tesztelés lefut és a teljes szoftver mögötti (gyakran virtualizált) infrastruktúra automatizáltan újratelepítésre kerül, a teljes szoftverrendszer elindul annak szükséges függőségeivel (pl. adatbázisok) együtt [10].

A DevOps pipeline-ok működési elve a következő: a fejlesztők dolgoznak a forráskódon, amikor elkészülnek a verziókövetőbe átvezetik a módosításaikat. Ezt a CI rendszer automatikusan észleli és elindítja a fordítási folyamatokat, melynek gyakran velejárója a legfontosabb unittesztek futtatása is. Ha ez sikeresen végigment, akkor a további, általában lassabb, azaz nagyobb futási idővel rendelkező komponens, GUI és performancia tesztek indulnak el. A tesztek mellett nagyon gyakran statikus elemzők is vizsgálják a kódot, egyéb eszközök a szoftver törhetőségét és a felhasznált könyvtárak licenceit elemzik. Ha a szoftver megbízhatónak minősül, akkor az éles rendszerben (productionben) automatizáltan elindul. A szükséges virtuális gépek, konténerek elindulnak a szoftver legújabb verziójával és a szükséges szolgáltatásokkal. A futó szoftvert pedig átfogó monitoring rendszerek elemzik, a logokat automa-

tikusan kiértékélik, hogy a szoftver minél stabilabban futhasson, hibák esetén a fejlesztőket értesíthessék. A DevOps hangsúlyozza annak fontosságát, hogy a pipeline bármelyik pontjáról kaphassanak visszajelzéseket a fejlesztők [36].

4.1. A/B tesztelés

Az A/B tesztelés lényege, hogy két hasonló megoldás (kérdőív, reklám, szoftver, stb.) kapcsán az ügyfelek elégedettségét mérjük le és hasonlítjuk össze. Ilyenkor a tesztelés hosszabb időt is igénybe vehet, hogy megfelelő visszajelzésekkel lehessen dolgozni. Mi ezt a fajta összehasonlítást Docker konténerizált környezetben indított alkalmazások esetében alkalmaztuk. A Docker előnye, hogy az alkalmazás különállóan futtatható komponenseit külön konténerben tudja futtatni az összes szükséges függőségével együtt, így megvalósul a komponens teljesen izolált futtatása a hostot üzemeltető operációs rendszerének megosztott kernele segítségével [6].

Első lépésként egy olyan futtató környezetet definiáltunk, melyben párhuzamosan tud futni a szoftver A és B verziója webes környezetben. Egy-egy végfelhasználóhoz mindig ugyanaz a verzió jut el a loadbalanceren keresztül. Python script segítségével lehet indítani a tesztelést, majd a kiértékelés végén a vesztes rendszer helyére is győztes verzió kerül. A megoldásunk biztosít egy speciális logging megoldást, ami alapján a kiértékelést el lehet végezni. A Python script a Docker hoston fut, parancssori felülettel (command line interface, CLI) rendelkezik [34].

Két metrikát definiáltunk annak eldöntésére, hogy melyik a ”jobb”: a kattintások száma a weboldalon, valamint az ott töltött idő. Mivel általánosan nem interpretálhatóak ezek metrikák, ezért az üzemeltetők tetszőlegesen használhatják ezen adatokat, annak eldöntésére, hogy számukra melyik a fontosabb.

Készítettünk egy olyan API-t, amellyel az A/B tesztelések felügyelhetőek: indíthatóak, monitorozhatóak, megszakíthatóak, valamint tetszőleges CI/CD környezetből is kényelmesen elérhető [32].

4.2. Multivendor rendszerek

Vannak olyan komplex rendszerek, ahol maguk a különálló komponensek is rendkívül bonyolult rendszerek. Ilyenek például napjaink telekommunikációs rendszerei [11]. Ezeknél a rendszereknél az operátorok (például Vodafone, Orange) az alrendszerekből tetszés szerint válogathatnak a különböző cégek (Ericsson, Huawei, Nokia) termékeiből. Ezen rendszereknél több problémát

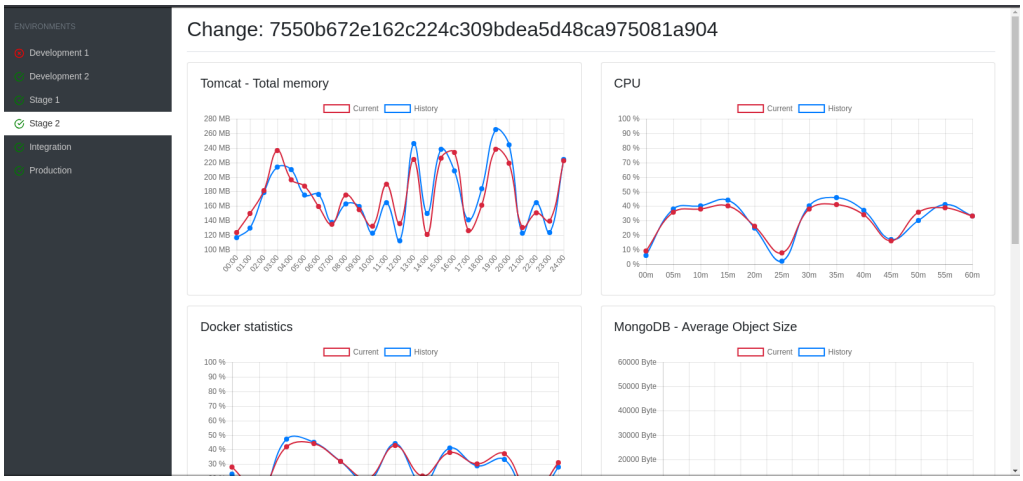
is okoz egy-egy komponens frissítése. A végfelhasználók rendkívül nagy száma és a rendelkezésre állási szabályok tovább bonyolítják azt, hogy az egyes komponensek milyen körülmények között, milyen sorrendben frissíthetőek. A különböző fejlesztőknél kiadott legfrissebb verziói egymásra gyakorolt hatása egyáltalán nem egyszerű. A különböző vendoroknál lévő DevOps pipeline-ok szinkronizálása sem megoldható. Ezért kidolgoztunk egy módszert, amelynek segítségével a multivendor pipeline-ok kezelhetőek. Létrehoztunk egy eszközt, amely a pipeline-okban elkészült artifactokhoz biztosít egy olyan időszeletet, amikor az elkészült rendszer biztonságosan beüzemelhető. Az eszköz implementációjához Neo4J gráf adatbázist használtuk fel. Ennek az eszköznek a segítségével mérhetően gyakrabban lehetett frissíteni egy-egy komponenst, mint a teljes rendszert [24].

4.3. DevOps Dashboard

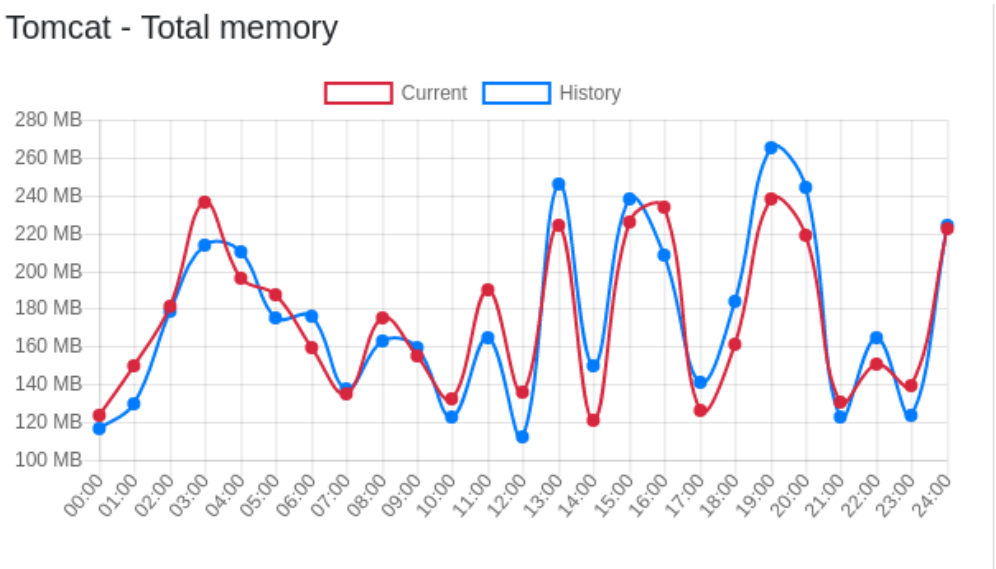
Egy olyan új DevOps eszközt terveztük meg, ami koncepcióját tekintve hasonlít a Continuous Integration rendszerekhez: azt jeleníti meg egy saját dashboard felületen, hogy a szoftver különböző verziói hogyan működnek éles környezetükben. Ellentétben azzal a CI-os megközelítéssel, hogy egy változás a szoftver forráskódjában milyen hatással van a fordítási és tesztelési folyamatokra, mi arra koncentrálunk, hogy a változás hogyan befolyásolja az éles rendszer teljesítményét, használatát. Begyűjtjük és megjelenítjük, hogy alapvető paraméterek, például a memóriahasználat, a CPU terheltségi adatok hogyan változnak meg 1-1 változás kapcsán, de jóval érdekesebb vizsgálat lehet, hogy az új funkciót megtalálják-e a végfelhasználók, tudják-e használni, át kell-e tervezni a felhasználói felületet, esetleg meg kellene szüntetni a funkciót az érdeklődés hiányában. A rendszer prototípusa elkészült, a központi dashboard felületet a 4.1 ábra mutatja be.

Az egyes service-ekkel (például Tomcat webszerver, Docker, stb.) a daemonként futó ágensek kommunikálnak. Ezek az ágensek felelősek azért, hogy a megfigyelt service adatait rendszeresen begyűjtsék, aggregálják es továbbítsák a dashboard irányába. Ilyen adatok a szoftverünk dashboardján keresztül vizualizált formában megtekinthetőek, például egy memóriafogyasztási grafikon látható a 4.2 ábrán. Az ágensek általános és specifikus tulajdonságaikat, kommunikációs mechanizmusukat definiáltuk [39].

A DevOps megközelítés tézishez tartozó publikációim: [24, 32, 33, 34, 39].



4.1. ábra. A prototípus szoftver dashboard felülete



4.2. ábra. Egy példa memóriafogyasztás vizualizációjára

5. fejezet

További kutatási témák

5.1. Statikus elemzési feladatok

5.1.1. Funktorok refaktorálása

A C++14-es szabványában megjelentek a *transzparens funktorok*. Ezek előnye, hogy a fordítóprogram paraméter dedukció segítségével definiálja a funktor `operator()` tagfüggvényének paraméterezését, nem kell a programozónak kiírnia template paraméterként ezt, illetve ezeket. A transzparens funktorok így biztonságosabban használhatóak, ugyanis nem vezetnek be hibás konverziókat, a programozók pedig megszabadulhatnak a felesleges kódduplikátumoktól [15].

Kidolgoztunk egy statikus elemzésen alapuló eszközt, amely meglévő, régebbi szabvány szerint készült kódokat refaktorál kihasználva a transzparens funktorok előnyeit. Az eszközt Clang Tidy bővítményként valósítottuk meg [15]. Ezek a refaktorálások nem végezhetőek el bizonyos vizsgálatok nélkül, de a meglévő kódok jelentős részében javítani tudja a nem kívánatos típuskonverziókkal kapcsolatos problémákat a futásidő növekedése nélkül.

5.1.2. Szabványok változásának problémája

A programozási nyelvek is fejlődnek, újabb és újabb szabványok jelennek meg, egyre több nyelvi konstrukció válik elérhetővé, illetve a felderített hibalehetőségeket csökkentik. Az az általános elképzelés, hogy a bővítések backward- vagy reverz-kompatibilisek legyenek, így a meglévő kódbázisok fordíthatósága és szemantikája, azaz működése ne változzon meg. Kirívó ellenpélda a Python 2 és 3-as verziója közötti különbség [23].

A C++ programozási nyelvnek is rendszeresen jelennek meg újabb szabványai: C++11, C++14, C++17, valamint a C++20 várható 2020-ban. A

szabványbizottság határozottan fontosnak érzi, hogy ilyenkor bővítéskor lehetőség szerint a szabványkönyvtár változzon meg, ne újabb és újabb nyelvi elemekkel bővüljön a C++. Sajnos ez is be tud vezetni olyan eseteket, ami mást jelent két különböző C++ szabvány szerint. Az alábbi kódrészlet bemutat egy ilyen esetet [9]:

```
struct S
{
  S() : i( ++counter ) {}
  static int counter;
  int i;
};

int S::counter = 0;

int main()
{
  std::vector<S> v( 5 );
  for ( std::size_t x = 0 ; x < v.size(); ++x )
  {
    std::cout << v[x].i << ' ';
  }
}
```

A kódrészlet a C++03-as szabvány szerint a `1 1 1 1 1` kimenetet írja ki, a C++11-es szabvány szerint pedig a `1 2 3 4 5` kimenetet adja [8]. A különbség oka, hogy a korábbi szabvány szerint ez a kódrészlet egyszer létrehoz egy default konstruált `S` típusú objektumot a két-paraméteres konstruktor második paraméternek alapértelmezett értékként, amit ötször bemásol a vectorba. Ezzel szemben a C++11-es szabványtól ez az egy-paraméteres konstruktort hívja meg, ami jelen kódrészlet alapján ötször meghívja az `S` default konstruktorát.

Egy Clang-alapú eszközt valósítottunk meg, amelynek segítségével korábbi szabvány szerint megírt kódok validálhatóak, hogy újabb szabvány szerint sem változik a kód jelentése [9]. Ehhez a felépített AST-ken mélységi bejárás segítségével nézzük meg, hogy az AST ugyanolyan elemekből épül-e fel mindkét szabvány szerint.

5.2. Memóriakezelési problémák C++-ban

A C/C++ nyelvekben a dinamikus memóriakezelés feladata a programozók vállát nyomja, fordítóprogramok nem foglalkoznak a memóriakezelés helyességének validálásával, ezért a memóriaszivárgás klasszikus C/C++ probléma, amit napjainkban is el lehet követni [38]. Noha a C++ nyelv behozott olyan nyelvi konstrukciókat, amelyekkel ez a hibalehetőség csökkenthető, de nem kényszeríti ezek felhasználását. A detektáló eszközök (például Clang Static Analyzer) is egyre fejlődnek, de átfogó megoldást egyik sem tud nyújtani.

Bemutattuk és kiértékeltük azon eszközöket, melyek a memóriaszivárgás elkerülésére terveztek: smart pointerek, Valgrind, Clang Static Analyzer, Hans Boehm garbage collectora [27]. Találtunk hiányzó elemet a C++ szabványkönyvtárából. Ezen túlmenően a modern smart pointerek (`unique_ptr`, `shared_ptr`) futási idejű teljesítményét is megvizsgáltuk és kiértékeltük [4].

5.3. Java kódgenerálás az Apache Flink keretrendszerhez

A Java programozási nyelvben támogatott fogalom a reflection, de jelentős negatív hatása van a futásidőre [40]. Az Apache Flink egy nyílt forráskódú keretrendszer streameken érkező adatok kötegelt feldolgozásához. Ez a keretrendszer nagyon széles körben alkalmazza a reflectiont, ami nehezen optimalizálható a JIT fordítás alatt.

A reflection helyett egy futási idejű kódgenerálási megoldást vezettünk be a szoftver hatékonyságának javítására. A FreeMarker template motor segítségével Java kódot generálunk, amit a Janino fordítóprogrammal fordítunk byte code-dá. Megnehezíti a generált byte code betöltését, hogy a Flink jellemzően több node-ból álló clusterként fut külön JVM-ekben. Ezért a kódgenerálás végén a létrehozott forráskódot juttatjuk el ahhoz a JVM-hez, amelyik majd fordítani és használni fogja. Arra is oda kellett figyelni, hogy egy node pontosan csak egyszer fordíthasson le 1-1 generált osztályt.

Komplex jobokkal elemeztünk a megoldásunk hatékonyságát, ami alapján több, mint hatszor lett gyorsabb a szerializáció teljesítménye, ami 20%-os javulást jelentett összesen a szoftver teljes egészében [18].

Kapcsolódó publikációim: [4, 8, 9, 15, 18, 27].

6. fejezet

Összefoglalás

Jelen dolgozatban bemutattam számos napjaink modern szoftverfejlesztéséhez tartozó lehetőséget és konstrukciót.

A statikus elemzés segítségével olyan hibákat is megtalálhatunk a kódban, amit a fordítóprogramunk nem fedez fel. Ez fontos lehet olyan esetekben, mint például a generikus programozási paradigmán alapuló STL könyvtár használata, ami segít leküzdeni a klasszikus C/C++ programozási hibákat, de közben bevezet számos új típusú hibalehetőséget, amit a fordítóprogramok nem tudnak detektálni. A statikus elemzést egyéb feladatokra is sikeresen alkalmaztuk. Az STL kapcsán találunk olyan esetet, amit nem lehet statikus elemzéssel detektálni, ezért nem-intruzív futási idejű megoldásokat is kidolgoztunk. A szimbolikus végrehajtás segítségével olyan statikus elemzési megoldáshoz jutunk, ami limitáltan, de a futási idejű adatokról, valamint a végrehajtási útvonalról is érvel. A Clang Static Analyzer egy ilyen eszköz C/C++/Objective-C nyelvekhez, ami ezen nyelvek fordítási modellje miatt limitált képességekkel bír. Több megoldást kidolgoztunk, hogy a Clang SA hibajelzési képességeit javítsuk.

A DevOps egy modern szoftverfejlesztési módszertan, melynek három alappillére a fejlesztés, a kódminőség biztosítása, valamint az automatizált deployment folyamatok. Sok lépésből álló, számos eszközt felhasználó DevOps pipeline-ok biztosítják, hogy a végfelhasználókhoz minél hamarabb, minél jobb minőségű szoftver jusson el. Ilyen DevOps pipeline-ok bővítésével foglalkoztunk: A/B teszteléssel a jobb végfelhasználói élményért, multivendor rendszerek hatékony frissítéséért, valamint egy olyan DevOps eszközzel, mellyel a fejlesztők precíz visszajelzést kaphatnak a staging tesztelési környezetben vagy az éles rendszerben futó szoftverrendszerről.

Ezen témákon túlmutató kutatásaimat röviden ismertettem.

Irodalomjegyzék

[1] B. Babati, G. Horváth, V. Májer, N. Pataki: *Static Analysis Toolset with Clang*, in Proc. of the 10th International Conference on Applied Informatics (ICAI 2017), pp. 23–29.

[2] B. Babati, G. Horváth, N. Pataki, A. Páter-Részeg: *On the Validated Usage of the C++ Standard Template Library*, in Proc. of the 9th Balkan Conference on Informatics (BCI' 19), pp. 23(1)–23(8), ACM Digital Library.

[3] B. Babati, N. Pataki: „Analysis of Include Dependencies in C++ Source Code", *Annals of Computer Science and Information Systems* **13**, pp. 149–156, (2017).

[4] B. Babati, N. Pataki: „Comprehensive performance analysis of C++ smart pointers", *Pollack Periodica* **12(3)**, pp. 157–166 (2017).

[5] B. Babati, N. Pataki: *Static Analysis of Functors' Mathematical Properties in C++ Source Code*, in Proc. of the International Conference on Numerical Analysis and Applied Mathematics (ICNAAM-2018), AIP Conference Proceedings **2116**, pp. 350002(1)– 350002(4).

[6] D. Bernstein: „Containers and cloud: From LXC to Docker to Kubernetes", *IEEE Cloud Computing* **1(3)**, pp. 81—84, (2014).

[7] E. Bodden: *The Secret Sauce in Efficient and Precise Static Analysis*, in ISSTA '18 Companion Proceedings for the ISSTA/ECOOP 2018 Workshops, pp. 85–93.

[8] T. Brunner, N. Pataki, Z. Porkoláb: *Tool for Detecting Standardwise Differences in C++ Legacy Code*, in Proc. of 13th International Scientific Conference on Informatics, pp. 57–62, IEEE Xplore (2015).

[9] T. Brunner, N. Pataki, Z. Porkoláb: „Backward Compatibility Violations and Their Detection in C++ Legacy Code Using Static Analysis", *Acta Electrotechnica et Informatica* **16(2)**, pp. 12–19 (2016).

[10] C. Ebert, G. Gallardo, J. Hernantes, N. Serrano: „DevOps", *IEEE Software* **33(3)**, pp. 94–100 (2016).

[11] F. Granelli, R. Bassoli: *Autonomic Mobile Virtual Network Operators for Future Generation Networks*, IEEE Network **32(5)**, pp. 76–84 (2018).

[12] Horváth G.: *Függvények modellezése globális statikus analízis céljából*, TDK Dolgozat (2014).

[13] G. Horváth, N. Pataki: „Clang matchers for verified usage of the C++ Standard Template Library", *Annales Mathematicae et Informaticae* **44**, pp. 99–109 (2015).

[14] G. Horváth, N. Pataki: *Source language representation of function summaries in static analysis*, in Proc.of the 11th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems (ICOOOLPS 2016), pp. 6(1)–6(9), ACM Digital Library.

[15] G. Horváth, N. Pataki: *Transparent functors for the C++ Standard Template Library*, in Proc. of the 11th Joint Conference on Mathematics and Computer Science (MaCS 2016), CEUR-WS **2046**, pp. 96–101.

[16] G. Horváth, N. Pataki: *Predicting bugs using symbolic execution graph*, in Proc. of the International Conference on Numerical Analysis and Applied Mathematics (ICNAAM-2018), AIP Conference Proceedings **2116**, pp. 350003(1)–350003(4).

[17] G. Horváth, N. Pataki: *Categorization of C++ Classes for Static Lifetime Analysis*, in Proc. of the 9th Balkan Conference on Informatics (BCI' 19), 21(1)–21(7), ACM Digital Library.

[18] G. Horváth, N. Pataki, M. Balassi: *Code Generation in Serializers and Comparators of Apache Flink*, in Proc.of the 12th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems (ICOOOLPS 2017), pp. 5(1)–5(6), ACM Digital Library.

[19] G. Horváth, A. Páter-Részeg, N. Pataki: *Detecting Misusages of the C++ Standard Template Library*, in Proc. of the 10th International Conference on Applied Informatics (ICAI 2017), pp. 129–136.

[20] G. Horváth, P. Szécsi, Z. Gera, D. Krupp, N. Pataki: *Challenges of Implementing Cross Translation Unit Analysis in Clang Static Analyzer*, in

Proc. of 2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation (SCAM 2018), pp. 171–176, IEEE Xplore.

[21] G. Horváth, P. Szécsi, Z. Gera, D. Krupp, N. Pataki: *Poster: Implementation and Evaluation of Cross Translation Unit Symbolic Execution for C Family Languages*, in 2018 ACM/IEEE 40th International Conference on Software Engineering: Companion Proceedings, pp. 428–429.

[22] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, J. Irwin: *Aspect-oriented Programming*, in Proc. of European Conference on Object-Oriented Programming (ECOOP' 97), Lecture Notes in Computer Science **1241**, pp. 220–242.

[23] B. A. Malloy, J. F. Power: *Quantifying the Transition from Python 2 to 3: An Empirical Study of Python Applications*, in Proc. of the 2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), pp. 314–323.

[24] M. P. Martinez, T. László, N. Pataki, Cs. Rotter, Cs. Szalai: *Multivendor Deployment Integration for Future Mobile Networks*, in Proc. of SOFSEM 2018: Theory and Practice of Computer Science : 44th International Conference on Current Trends in Theory and Practice of Computer Science, Lecture Notes in Computer Science **10706**, pp. 351–364.

[25] S. Meyers: *Effective STL*, Addison-Wesley (2001).

[26] Nyékyné Gaizler J.: *Programozási Nyelvek*, Kiskapu Kiadó (2003).

[27] D. Papp, N. Pataki: „Bypassing Memory Leak in Modern C++ Realm", *Annales Mathematicae et Informaticae* **48**, pp. 43–50 (2018).

[28] Pataki N.: „Advanced Functor Framework for C++ Standard Template Library", *Studia Universitatis Babeş-Bolyai Informatica* **LVI**, pp. 99–113 (2011).

[29] Pataki N.: *Generatív programok helyessége*, PhD Disszertáció, (2013).

[30] Pataki N.: *DevOps módszertan a modern szoftverfejlesztésben*, Az Új Nemzeti Kiválóság Program 2017/2018. tanévi pályázat nyertesei, p. 344 (2018).

[31] T. Reps, S. Horwitz, M. Sagiv: *Precise Interprocedural Dataflow Analysis via Graph Reachability*, in Proc. of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 1995, pp. 49–61.

[32] Á. Révész, N. Pataki: *Continuous A/B Testing in Containers*, in Proc. of the 2019 2nd International Conference on Geoinformatics and Data Analysis, pp. 11–14, ACM Digital Library (2019).

[33] Á. Révész, N. Pataki: *Integration Heaven of Nanoservices*, in Proc. of the 21th International Multi-Conference INFORMATION SOCIETY IS'2018, Volume G: Collaboration, Software and Services in Information Society, pp. 43–46.

[34] Á. Révész, N. Pataki: *Containerized A/B Testing*, in Proc. of the Sixth Workshop on Software Quality Analysis, Monitoring, Improvement, and Applications (SQAMIA 2017) CEUR-WS **1938**, pp. 14(1)–14(8), (2017).

[35] V. O. Savitskii, D. V. Sidorov: „Fast Analysis of Source Code in C and C++", *Programming and Computer Software* **39(1)**, pp. 49–55 (2013).

[36] J. Smeds, K. Nybom, I. Porres: *DevOps: A Definition and Perceived Adoption Impediments*, In Proc. of the Agile Processes in Software Engineering and Extreme Programming (XP 2015), Lecture Notes in Business Information Processing **212**, pp. 166–177.

[37] O. Spinczyk, D. Lohmann, M. Urban: *Advances in AOP with AspectC++*, in Proc. of the 2005 Conference on New Trends in Software Methodologies, Tools and Techniques, pp. 33–53.

[38] B. Stroustrup: *A C++ programozási nyelv*, Kiskapu Kiadó (2000).

[39] M. Török, N. Pataki: *Service Monitoring Agents for DevOps Dashboard Tool*, in Proc. of the 21th International Multi-Conference INFORMATION SOCIETY (IS'2018), vol. G: Collaboration, Software and Services in Information Society, pp. 47–50.

[40] C. Tudose, C. Odubăşteanu, R. Serban: *Java Reflection Performance Analysis Using Different Java Development*, in Advances in Intelligent Control Systems and Computer Science, Springer, pp. 439–452 (2013).