TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond
Tarkvarateaduse instituut

Anton Ašot Roolaid
980774IAPB

AVATUD LÄHTEKOODIGA VAHENDITE KOHANDAMINE MICROSOFT VISUAL C++ TARKVARALAHENDUSTE KVALITEEDI ANALÜÜSIKS SONARQUBE SERVERIS

Bakalaureusetöö

Juhendaja: Juhan-Peep Ernits
PhD

Tallinn 2019

Autorideklaratsioon

Kinnitan, et olen koostanud antud lõputöö iseseisvalt ning seda ei ole kellegi teise poolt varem kaitsmisele esitatud. Kõik töö koostamisel kasutatud teiste autorite tööd, olulised seisukohad, kirjandusallikatest ja mujalt pärinevad andmed on töös viidatud.

Autor: Anton Ašot Roolaid

21.05.2019

Annotatsioon

Käesolevas bakalaureusetöös uuritakse, kuidas on võimalik saavutada suure hulga C++ lähtekoodi kvaliteedi paranemist, kui ettevõttes kasutatakse arenduseks Microsoft Visual Studiot ning koodikaetuse ja staatilise analüüsi ülevaate saamiseks SonarQube serverit (*Community Edition*). Seejuures SonarSource'i poolt pakutava tasulise SonarCFamily for C/C++ analüsaatori (mille eelduseks on SonarQube serveri *Developer Edition*) asemel kasutatakse tasuta ja vaba alternatiivi: SonarQube C++ Community pluginat.

Analüüsivahenditena eelistatakse avatud lähtekoodiga vabu tarkvaravahendeid. Valituks osutuvad koodi kaetuse analüüsi utiliit OpenCppCoverage ja staatilise analüüsi utiliit Cppcheck. Siiski selgub, et nende utiliitide töö korraldamiseks ja väljundi sobitamiseks SonarQube Scanneri vajadustega tuleb kirjutada paar skripti: üks PowerShellis ja teine Windowsi pakkfailina. Regulaarselt ajastatud analüüside käivitamist tagab QuickBuild, mis on ka MSBuildi käivitajaks üksuste testide kompileerimisel.

Pikemalt kirjeldatakse ka Cppchecki töös ilmnenud defektide parandamist Cppchecki avatud lähtekoodis, seda nii Cppchecki peamise arendaja Daniel Marjamäki poolt kui ka käesoleva töö autori loodud ühe tõmbetaotluse aktsepteerimist Cppchecki ametlikku repositooriumisse GitHubis.

Suures plaanis täitis lahendus püstitatud eesmärgi, sest ettevõtte C++ lähtekoodis avastati ja parandati mitmeid vigu, sealhulgas kümmekond tõsist viga.

Lõputöö on kirjutatud eesti keeles ning sisaldab teksti 38 leheküljel, 5 peatükki, 4 joonist.

Abstract

Adapting Open Source Tools for Code Quality Analysis of Microsoft Visual C++ Solutions with SonarQube Server

In this bachelor thesis we investigate the possibilities for achieving a quality improvement in a large C++ source code base, given the fact that the company uses Microsoft Visual Studio for development, and SonarQube server (Community Edition) for code coverage and static source code analysis overviews. Also, instead of using SonarSource's paid SonarCFamily for C/C++ analyzer (which requires the Developer Edition of SonarQube server), the free and open source alternative is used: SonarQube C++ Community plugin.

Free and open source software is preferred when choosing the analysis tools. As the result, the code coverage analyzer OpenCppCoverage and the static code analyzer Cppcheck are chosen. However, it becomes clear that a couple of scripts, one in PowerShell and the other as a Windows batch file, need to be written in order to manage the workflow of these utilities and to transform the output according to the needs of SonarQube Scanner. QuickBuild ensures periodical execution of the scheduled analyses, it also starts MSBuild for unit tests compilation.

Bugs found in Cppcheck during its operation, fixed mostly by the Cppcheck's main developer Daniel Marjamäki, as well as a pull request made by the author of this thesis and officially accepted into Cppcheck's GitHub repository, are described in more detail.

In general, the implemented solution did fulfill the set objective, because several bugs, including about a dozen of severe ones, were discovered and fixed in the company's C++ source code.

The thesis is in Estonian and contains 38 pages of text, 5 chapters, 4 figures.

Lühendite ja mõistete sõnastik

Atribuudifail	Microsoft Visual Studio fail (`*.props`), milles on sarnaselt projektifailiga võimalik salvestada mitmesuguseid kompileerimisega seonduvaid seadistusi ja atribuute, ingl *property sheet file*.
JAR	*Java archive*, failivorming paljude Java klassifailide ja seonduvate metaandmete koondamiseks üheks tervikuks, pakitud failiks.
Lahendusfail	Microsoft Visual Studio fail (`*.sln`) projektifailide rühmitamiseks, ingl *solution*.
Pakkfail	Windowsi konsooli käskude keeles kirjutatud skriptifail (`*.cmd` või `*.bat`), ingl *batch file*.
PDB	*Program database*, failivorming laademoodulite kohta käiva silumisteabe salvestamiseks.
Projektifail	Microsoft Visual Studio fail (C++ korral on see `*.vcxproj`) lähtekoodi- ja ressursifailide rühmitamiseks ning kompilaatori ja linkuri seadistuste jm konfiguratsiooni salvestamiseks, ingl *project*.
XPath	*XML Path Language*, päringukeel XML dokumentides elementide jm osade valimiseks.

Sisukord

Jooniste loetelu

1 Sissejuhatus

Ettevõttes, kus käesoleva bakalaureusetöö autor töötab C++ tarkvaraarendajana, on kasutusel Microsoft Visual Studio arenduskeskkond. Ettevõtte ühe põhilise tarkvaratoote C++ keeles kirjutatud osa koosneb kümnekonnast tarkvaralahendusest (ingl *solution*), millest igaüks hõlmab ligi sadakonda projekti, kokku mõne tuhande lähtekoodifaili rühmitamiseks. Kogu selle C++ lähtekoodi maht on üle miljoni koodirea.

Sellise suurusega koodibaasi puhul oli vigade leidmine ilma ülevaatlike vahenditeta raskendatud. Seetõttu otsustati see probleem kesksel viisil ära lahendada.

Koodi kvaliteedi automatiseeritud analüüsiks on ettevõttes kasutusel SonarQube server (täpsemalt, selle tasuta *Community Edition*), mida on juba pikemat aega aktiivselt rakendatud tarkvaratoote Java keeles kirjutatud osa kallal. Et saavutada sarnast ülevaadet ka C++ osas, paluti autoril uurida erinevate C++ koodikaetuse ja staatilise analüüsi utiliitide, eelistatult avatud lähtekoodiga vabade tarkvaravahendite, SonarQube serveriga lõimimise võimalusi.

Niisiis on töö üldine eesmärk C++ koodi kvaliteedi tõstmine autori töökohal. Konkreetselt on aga võetud ülesandeks uurida välja, millised avatud lähtekoodiga vahendid sobivad SonarQube'iga kasutamiseks ja kuidas saab need vahendid kokku siduda automatiseeritud regulaarseks töövooks.

Järgnevalt kõigepealt vaadatakse üle ettevõttes juba kasutatavaid tarkvaralisi vahendeid (Visual Studio, SonarQube, QuickBuild, MSBuild) ja tehnoloogiaid (üksuste testimine Google Testiga, atribuudifailide kohandamine) ning mainitakse paar levinud meetodit koodi kvaliteedi automatiseeritud analüüsiks (lähtekoodi staatiline analüüs ja koodi kaetuse analüüs), vt peatükk 2.

Samuti pakutakse välja loodava lahenduse esialgne kava, vt jaotis 2.8. Seejärel jaotises 2.9 puudutatakse koodi kvaliteedi analüüsi vahendite valiku teemat, arvestades siiski konkreetse ettevõtte konteksti ja sellest tulenevaid kitsendusi.

Lõpuks üksikasjalikult kirjeldatakse valitud utiliitide (Cppcheck ja OpenCppCoverage) SonarQube'iga kasutuseks kohandamist ja kogu analüüsiprotsessi automaatset juhtimist alates QuickBuildist ja lõpetades skriptidega (põhiline Windowsi pakkfail, korrigeeriv PowerShelli skript), vt peatükk 3.

Teostatud lahendusele ja selle rakendamise tulemusele antakse hinnang peatükis 4.

2 Kasutatavad tehnoloogiad ja koodi analüüsi meetodid

Käesolevas peatükis kõigepealt tutvustatakse põgusalt meetodeid, mida soovitakse C++ tarkvara kvaliteedi parandamiseks kasutada ettevõttes, milles käesoleva bakalaureusetöö autor töötab. Ka vaadatakse üle, millised tarkvaralised vahendid on ettevõttes kasutusel ja milliseid tehnoloogiaid rakendatakse.

Seejärel pakutakse välja loodava lahenduse esialgne kava. Lõpuks põhjendatakse koodi kvaliteedi analüsaatorite valikut ja seda mõjutanud kriteeriume.

2.1 Staatiline analüüs

Tarkvara staatilise analüüsi all peetakse tavaliselt silmas lähtekoodi automatiseeritud kontrollimist selles esineda võivate vigade ja võimalike riskide tuvastamiseks. Sarnaselt koodi läbivaatusele inimeste (arendajate) poolt, nii ka tüüpilise staatilise analüüsi käigus ei käivitata analüüsitavat tarkvara, vaid kogu töötlus käib tarkvara lähtekoodi kallal.

Lähtekoodi staatilise analüüsi tööriistu leidub mitmesuguseid. Mõned neist on loodud nt eelkõige tarkvara turvalisusega seonduvate nõrkuste avastamiseks, hoopis teised aga tegelevad üksnes koodistiili ja -vorminduse korrektse järgimise kontrollimisega. Erinevad viisid, kuidas staatilised analüsaatorid saavad aidata kaasa C++ keeles kirjutatud tarkvara kvaliteedi tõstmisele, on lähemalt kirjeldatud nt Elias Penttilä magistritöös [1].

2.2 Kaetuse analüüs

Tarkvara lähtekoodi kaetuse all mõeldakse reeglina testide läbimise käigus käivitatud koodi osakaalu kogu analüüsitavas koodibaasis[1]. Mida kõrgem on koodikaetuse protsent, seda väiksem on tõenäosus, et tarkvaras leidub seni avastamata vigu (kuna testitud saab suurem osa koodist). Sellise analüüsi sooritamiseks on olemas automatiseeritud vahendid.

[1] Vikipeedia artikkel „Code coverage“, https://en.wikipedia.org/wiki/Code_coverage (ingl, 21.05.2019)

Levinuimad koodikaetuse kriteeriumid on koodiridade, koodiharude ja funktsioonide kaetus. Kõige lihtsam on lugeda kokku, mitu rida kogu lähtekoodist on kaetud ja mitu mitte, võib pidada arvet ka iga konkreetse rea oleku (läbitud või mitte) kohta eraldi; see on koodiridade kaetus. Veidi keerulisem on funktsioonide ja/või harude läbimise kohta statistika kogumine, kuna sel juhul peab analüsaator olema koodi struktuurist veidigi teadlik. Käesolevas töös on koodikaetusest edaspidi juttu ainuüksi koodiridade mõistes.

2.3 SonarQube'i ülevaade

SonarQube on avatud lähtekoodiga platvorm tarkvara kvaliteedi pidevaks ülevaatuseks[1]. SonarQube server kuvab analüüsitava tarkvara kvaliteedi peamiste näitajate kohta oma veebiliideses mitmesuguseid mõõdikuid ja diagramme. Kvaliteedinäitajate muutumiskäik on regulaarselt jälgitav, neid on võimalik perioodiliselt võrrelda eelmiste seisudega.

Hinnastamine toimub sõltuvalt analüüsitava koodibaasi suurusest. Serveri ühe eksemplari kasutamise kohta kehtib aastatasu, v.a *Community Edition*. Litsentsi soetamiseks on saadaval neli erinevat hinnapaketti[2]:

- *Community Edition* on tasuta ning sellel on 15 programmeerimiskeele (sealhulgas Java, C#, Ruby, Python, PHP, kuid mitte C ega C++) tugi.
- *Developer Edition* on mitme lisavõimalusega (nt repositooriumi arendusharude ja tõmbetaotluste analüüs) ning pakub kokku 21 programmeerimiskeele, sealhulgas nii C kui ka C++, ametlikku tuge.
- *Enterprise Edition* lisab suurte tarkvaraprojektide hierarhilise halduse võimalusi ja veel viie programmeerimiskeele toe. Aastatasu kõrgem kui *Developer Edition*il.
- *Data Center Edition* on mõeldud missioonikriitilisi tarkvaralahendusi arendavate suurettevõtete vajaduste rahuldamiseks, kõigi võimalustega ning on veelgi kallim.

Nagu on juba mainitud, on autori töökohas kasutusel *Community Edition*.

[1] SonarQube (a SonarSource Product), https://www.sonarqube.org/

[2] SonarSource Plans & Pricing, https://www.sonarsource.com/plans-and-pricing/ (ingl, 21.05.2019)

2.3.1 Mõõdikud

SonarQube serveris kuvatavatest tarkvarakvaliteedi näitajatest pakuvad ettevõttele enim huvi peamiselt järgmised:

- lähtekoodis esinevad vead ja muud probleemid, mis on avastatud koodi staatilise analüüsi käigus;
- koodi kaetuse ulatus üksuste testide poolt;
- korduvate koodilõikude (kopeeritud read, plokid või failid) osakaal koodibaasis.

SonarQube serveri veebiliidese kaudu saab vaadata nii vastavaid mõõdikuid ning võrdlusgraafikuid kui ka klõpsata linkidel, et sirvida aina enam detaile välja toovatel aruannetel, kuni konkreetsete lähtekoodiridade lugemiseni välja.

2.3.2 SonarQube Scanner

Komponent, mis analüüsib lähtekoodi (või võtab vastu väliste analüsaatorite aruandeid) ja laadib tulemused SonarQube serverisse üles, kannab nime SonarQube Scanner. See koosneb Windowsi käsureaskriptist (pakkfailist `sonar-scanner.bat`) ja ühest JAR-failist, mille see pakkfail käivitab. Konfiguratsioonifaili `sonar-project.properties` kaudu saab anda ette parameetreid [2]. Autori töökohas on kasutusel joonisel 1 näidatud parameetrid.

SonarQube serveri aadress on sätitud parameetriga `sonar.host.url`: sinna laaditakse üles analüüsi aruanded. Juurdepääsuks serverisse kasutatakse antud juhul parameetriga `sonar.login` etteantud autentimise tõendit (alternatiivina oleks võimalik kasutada ka kasutajanime ja salasõna).

Et analüüs piirduks vaid C++ keelega, kasutatakse parameetrit `sonar.language`, vastasel juhul käivitub mitmekeelne analüüs. Selle parameetri kasutamist loetakse SonarQube'i uuemates versioonides küll vananenuks, kuid aitab vältida tulemustes segadust, mis tuleneb ettevõtte C++ koodibaasi kaustades leiduvatest asjassepuutumatutest C# ja Java lähtekoodifailidest.

Analüüsitava C++ tarkvaraosa sisemine identifikaator, kasutajale kuvatav nimi, samuti versioon on seadistatud vastavalt parameetrite `sonar.projectKey`, `sonar.projectName` ja `sonar.projectVersion` kaudu.

Parameetriga `sonar.scm.disabled` lülitatakse välja Scanneri-poolne SVN-repositooriumi töötlus (sest tollal kasutusel olnud versioonil tekkisid sellega teatud probleemid).

Lähtekoodifaile sisaldavate alamkaustade nimistu on SonarQube Scannerile ette antud parameetri `sonar.sources` kaudu. Parameetri `sonar.sourceEncoding` kaudu on määratud lähtekoodifailide tärgistik, antud juhul on see Unicode'i kodeering UTF-8 (tegelikult aga konkreetses ettevõttes, kus autor töötab, kasutatakse C++ lähtekoodis ainult ASCII tärke).

```
sonar.host.url=https://sonarqube.firmanimi.corp
sonar.login=a5957266bce73b61f6a545abf0ae9e855cd2866f
sonar.language=c++
sonar.projectKey=frontend
sonar.projectName=Frontend
sonar.projectVersion=trunk
sonar.sourceEncoding=UTF-8
sonar.scm.disabled=true
sonar.sources=core,drivers,framework,graphics,net,shared,toolkit,utils
sonar.cxx.cppcheck.reportPath=build/sonar/log/cppcheck_report.xml
sonar.cxx.coverage.reportPath=build/sonar/log/coverage_report.xml
sonar.cxx.errorRecoveryEnabled=True
sonar.cxx.msbuild.reportPath=build/sonar/log/msbuild.log
sonar.cxx.msbuild.charset=UTF-8
```

Joonis 1. SonarQube Scanneri konfiguratsioonifaili sisu.

Konfiguratsiooniread, mis algavad tärgijadaga `sonar.cxx.`, puudutavad aga jaotises 2.9 mainitud SonarQube C++ Community pluginat. Nende ridade tähendus on lahti seletatud jaotises 2.7 ning alajaotistes 3.1.1 ja 3.2.2, kus on juttu valitud vahendite liidestamisest SonarQube'iga.

2.4 Lähtekoodi rühmitamine Microsoft Visual Studios

Nagu juba eelnevalt mainitud, on autori töökohas C++ arenduskeskkonnana kasutusel Microsoft Visual Studio[1]. Kuna ettevõtte koodibaas on väga suur, siis tuleb lähtekoodi temaatiliselt rühmitada. Selleks saab kasutada Visual Studio standardseid võimalusi: projektifaile ja lahendusfaile.

Kuigi selline rühmitamine oluliselt hõlbustab arendajate tööd, siis, nagu ka töö käigus selgus, mõjutab see oluliselt staatilise koodi analüüsi vahendite kasutamise omapärasid.

[1] Microsoft Visual Studio: an integrated development environment, https://visualstudio.microsoft.com/vs/

Järgnevates alajaotistes seletatakse väga lühidalt lahti, kuidas lähtekoodi rühmitamine Visual Studios üldiselt välja näeb.

2.4.1 Projektifailid

C++ lähtekoodi põhi- ja päisefailide ning ressursifailide rühmitamiseks ühe laademooduli (nt dünaamilise teegi ehk DLL-i) kompileerimiseks vajalikuks loogiliseks kogumiks saab Visual Studios kasutada nn projektifaile (`*.vcxproj`), ingl *project file*. Projektifailides salvestuvad nii lähtekoodifailide loetelu kui ka kompilaatori ja linkuri seadistused ja muu kompileerimiseks vajalik konfiguratsioon.

Projektil võib olla mitu erinevat konfiguratsiooni: tavaliselt vähemalt `Release` ja `Debug`, s.o vastavalt väljalaskeks ja silumiseks mõeldud kompileerimise variandid. Üksuste testide olemasolu korral võib kasutada nt `Test Release` ja `Test Debug` konfiguratsioone (need ei ole vaikimisi loodavad, vaid hiljem käsitsi tekitatud konfiguratsioonid).

Laademooduli kompileerimiseks saab Visual Studios siis lihtsalt klõpsata projektihalduris vastava projekti nimel ja lasta kompileerimisprotsess käima (vajadusel eelnevalt valides konfiguratsiooni). Ka automatiseeritud kompileerimisel nt MSBuildi abil saab käsureal anda ette konfiguratsiooni nime, mida soovitakse kompileerida.

2.4.2 Kohandatud atribuudifailid

Tihti võivad arendajad tabada end mõttelt, et ühtesid ja samu kompilaatori/linkuri seadistusi vms on tarvis kasutada mitmete projektide juures, samas kui mitmete teiste projektide juures tuleb kasutada teisi (kui samuti korduvaid) seadistusi. Sel juhul ei ole mõistlik sättida iga projekti atribuute käsitsi atribuudihalduris, vaid tuleks luua ja kasutada kohandatud atribuudifaile.

Microsoft Visual Studio kohandatud atribuudifailis (`*.props`), ingl *custom property sheet file* [3], salvestuvad samasugused kompilaatori/linkuri jm seadistused, atribuudid ja makrod nagu projektifailideski, kuid mitte lähtekoodifailide loetelud. Atribuudifailid võivad kaasata teisigi atribuudifaile, sedasi tekitades seadistuste hierarhiat.

Atribuudifailid on justkui mallid, mida saab algul erinevate projektitüüpide jaoks luua ja siis atribuudihalduri abil konkreetsetele projektidele lisada, et rakendada korraga kõiki vajalikke seadistusi (ilma neid käsitsi ükshaaval üle käimata). Kui koodibaasis on sadu

projekte, siis annab atribuudifailide kasutus nii ajasäästu kui ka aitab vältida juhuslikke seadistamisvigu.

2.4.3 Lahendusfailid

Kui koodibaas on suur ja mitmekesine, siis projektifailidest üksi ei piisa, kuna tarkvara erinevad komponendid võivad hõlmata mitmeid laademooduleid. Komponente või olla ka mitu (vastavalt tarkvaratoote ülesehitusele).

Selleks rühmitatakse ühe komponendi kõigi moodulite kompileerimiseks vajalikud projektifailid Visual Studio lahendusfailide (`*.sln`), ingl *solution files*, alla. Neis failides salvestub projektifailide loetelu.

Ka lahendusfailidele võib luua erinevaid konfiguratsioone (nt ülalmainitud `Release`, `Debug`, `Test Release`, `Test Debug`). See võimaldab tarkvaralahenduste kompileerimisel kasutada kas lihtsalt kõigi kaasatud projektide samu konfiguratsioone (nt `Release`) või panna lahendusfaili suvandites eraldi paika, et nt üksuste testide puhul kompileeritaks teatud (teste sisaldavad) projektid `Test Debug` konfiguratsioonis, teised projektid aga tavalises (`Debug`) konfiguratsioonis (kui neid on vaja testide toimimiseks) või jäetaks hoopiski kompileerimata (need projektid, kus teste veel ei ole).

2.5 Tarkvaraüksuste testimise korraldus Google Testi abil

Visual Studios C++ tarkvaraüksuste testimise süstematiseerimiseks on mõistagi mitmeid võimalusi, kuid ühe lihtsa lähenemisena saab soovitada just testide kompileerimiseks mõeldud spetsiaalsete lisakonfiguratsioonide loomist nii lahendusfailide kui ka projektide tasemel. Autori töökohas on harilikest konfiguratsioonidest kasutusel `Debug` ja `Release` nimelised, niisiis said testide jaoks loodud vastavalt `Test Debug` ja `Test Release` konfiguratsioonid. Nende loomisel tuli Visual Studio konfiguratsioonihalduris kõigepealt määrata ka algsete suvandite pärimine õigesti: et `Test Debug` pärib oma suvandid olemasolevalt `Debug` konfiguratsioonilt ning et `Test Release` pärib vastavalt `Release` konfiguratsioonilt. Alguses said niiviisi paika kõigi lahendusfailide konfiguratsioonid, seejärel projektide omad (ainult neis projektides, milledes oli kavas luua üksuste testid).

Konfiguratsiooni `Test Release` kasutatakse üksuste testide igakordseks automaatseks kompileerimiseks ja käivitamiseks QuickBuildis enne seda, kui SVN-i repositooriumisse

saabub järjekordne sissekanne. Kui mõni test luhtub, siis jäetakse ära ka edasine koosteprotsess (`Release` konfiguratsiooni ei asutagi kompileerima) ning repositooriumi viimatisele üleslaadijale saadetakse e-kiri veateatega.

Olgu veel mainitud, et lahendusfailide testikonfiguratsioonides kompileeritakse ainult need projektid, millel on olemas testikonfiguratsioon või mis on vajalikud testide tööks. Teised projektid jäetakse vahele. See on paika pandud konfiguratsioonihalduri abil.

Testide lähtekoodifailid on projektides paigutatud spetsiaalsesse `tests` kausta, täpsemalt on need grupeeritud `tests`-nimelise filtri alla (Visual Studio failivaatega sakis). Need failid on mitte-testikonfiguratsioonides koosteprotsessist välja jäetud (vastava suvandi määramise teel). See on oluline konfliktide vältimiseks kompileerimisel.

Üksuste testid on kirjutatud Google Test raamistikus. Testide lähtekoodifailides on põhiliselt `TEST_F` ja `TEST_P` makrodega [4] defineeritud testid. Mõne soovitud C++ klassi testimiseks tavaliselt tuletatakse vastavast klassist testklass. Seejärel luuakse Google Mock raamistikus selle testitava klassi poolt pruugitavate teiste (väliste) klasside maketid. Selleks tuletatakse välistest klassidest makett-klassid, milledesse paigutatakse vastava liidese meetodite maketid, kasutades Google Mock raamistiku `MOCK_METHOD*` ja `MOCK_CONST_METHOD*` makrosid [5]. Vajadusel kaasatakse projekti koosseisu teisedki, juba olemasolevad, maketid. Lõpuks, testitava klassi konstruktoris antakse ette makett, mitte päris objekt.

Testide lähtekoodi põhifail, mida võib nimetada näiteks `test_main.cpp`, sisaldab `main`-funktsiooni, milles Google Test raamistik algväärtustatakse ning käivitatakse kõik testid.

Harilike (`Debug` ja `Release`) konfiguratsioonide kompileerimise tulemusel tekivad autori töökohas arendataval C++ tarkvaratootel üldiselt DLL-failid (sest enamus projektidest on teegid), seevastu tarkvaraüksuste testide konfiguratsioonide (`Test Debug` ja `Test Release`) kompileerimisel on tulemuseks EXE-failid. Selline eristumine toimub kohandatud atribuudifailide abil, kus linkuri väljundi tüübiks on sätitud eraldiseisev programm, mitte teek. (Mõlema konfiguratsioonitüübi puhul on kasutusel küll üks ja seesama atribuudifail, kuid nimetatud seadistuse kitsendava tingimusena kontrollitakse, kas konfiguratsioon on `Test Debug` / `Test Release`. Uute projektifailide loomisel peavad arendajad lisama neile atribuudifaili Visual Studio atribuudihalduri abil.)

Üksuste testi laademoodul on konsooliprogramm, mis sisaldab ühte või mitut üksuste testi. Programmile on võimalik käsurea kaudu ette anda mitmesuguseid parameetreid, näiteks `--gtest_output` tekitab testide kohta XML või muus vormingus aruande [4].

2.6 QuickBuildi ülevaade

Ettevõttes, milles töötab ka käesoleva bakalaureusetöö autor, on kasutusel QuickBuild[1]. Tegu on pidevlõimimist (ingl *continuous integration*) ja tarkvara pidevat kasutuselevõttu (ingl *continuous deployment*) võimaldava serveriga. Tarkvara lähtekoodi allalaadimine repositooriumist, kompileerimine, testide käivitus jm sammud on QuickBuildi paindliku veebiliidese kaudu seadistatavad.

QuickBuildi saab rakendada programmi koosteprotsessi automatiseerimiseks, nagu on näidatud Taavi Tali bakalaureusetöös [6]. Käesoleva töö raames on QuickBuildi rolliks kvaliteedianalüüsi protsessi automaatne perioodiline käivitus.

2.7 MSBuildi liidestamine SonarQube'iga

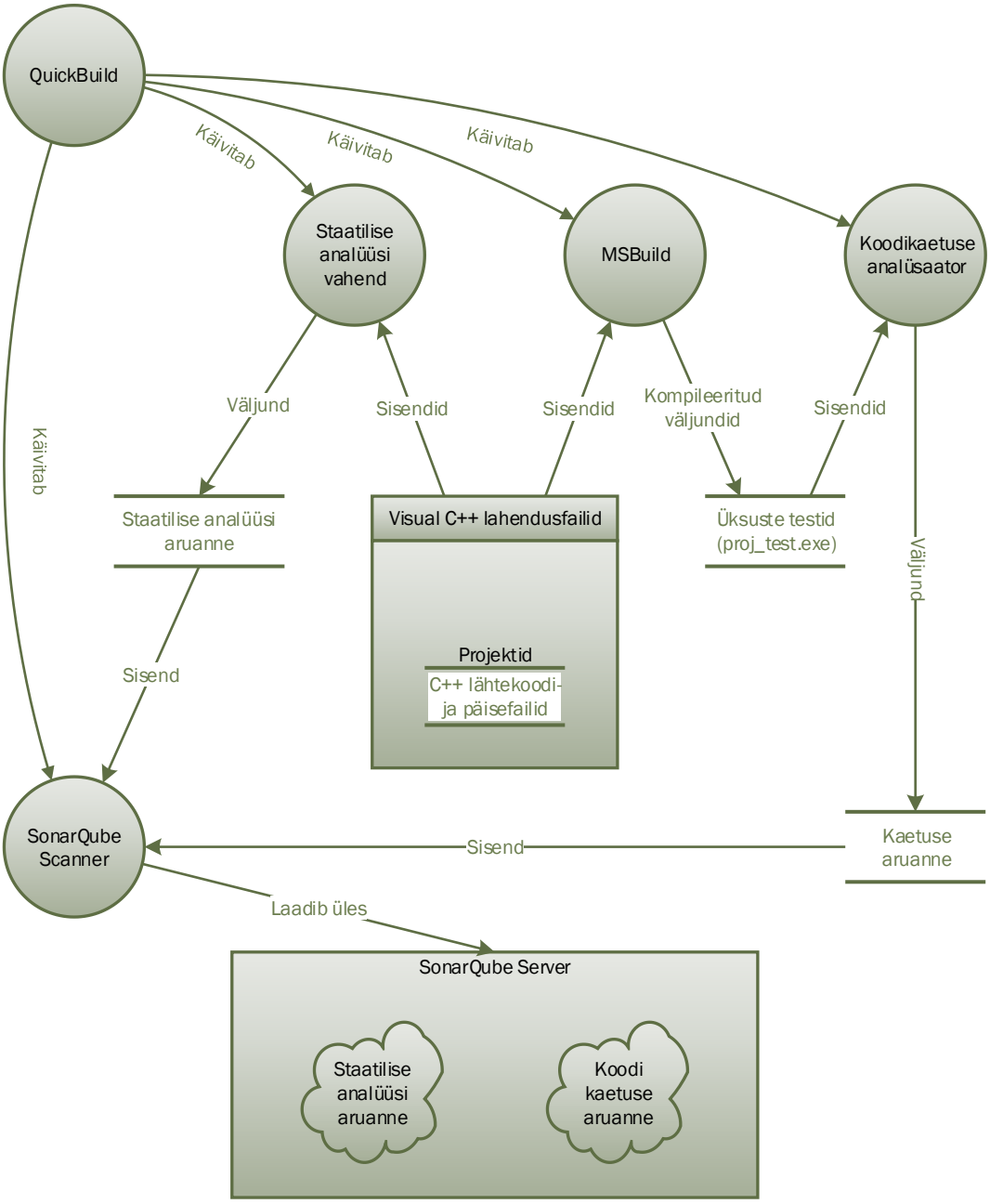
Jaotises 2.9 mainitud SonarQube C++ Community plugin võimaldab muuhulgas edastada lähtekoodi kompileerimise käigus kompilaatori poolt logifailisse väljastatavaid hoiatusi SonarQube serverisse. Seejuures plugina uuemates versioonides arvestatakse isegi (selles logifailis leiduvate) makrodefinitsioonide ja päisefailide otsinguteekondadega.

Vastavalt plugina dokumentatsioonile [7], antakse MSBuildi logifaili asukoht ette `sonar.cxx.msbuild.reportPath` nimelise konfiguratsiooniparameetriga. Joonisel 1 on näidatud selle parameetri kasutus SonarQube Scanneri konfiguratsioonifailis. Parameeter `sonar.cxx.msbuild.charset` aga määrab logifaili tärgistiku. Nagu samal joonisel näha, on antud juhul kasutusel Unicode'i kodeering UTF-8. Autori töökohas on see logifail (sarnaselt lähtekoodifailidega) tegelikult puhtas ASCII kodeeringus, sest konkreetses ettevõttes ühtegi ASCII-välist tärki ei lähtekoodis ega failinimedes ei kasutata.

[1] PMEase QuickBuild, https://www.pmease.com/quickbuild

2.8 Loodav lahendus

Esialgne kava lähtekoodi kvaliteedi vabavaraliste analüüsivahendite SonarQube'iga liidestamiseks on näidatud joonisel 2. Nagu jaotises 2.4 kirjeldatud, on kasutades Visual



Joonis 2. Loodava lahenduse lihtsustatud üldskeem.

Studiot kõik C++ lähtekoodi- ja päisefailid rühmitatud projektidesse, mis on omakorda kaasatud lahendusfailidesse. Need on sisendiks nii koodi staatilise analüüsi vahendile kui ka üksuste testide kompileerimise tarbeks MSBuildile.

Staatilise analüüsi tulemusena tekkiv aruanne läheb edasi SonarQube Scannerisse, mis laaditakse koos lähtekoodi kaetuse analüsaatori tekitatava aruandega üles SonarQube serverisse mitmesuguste mõõdikute kuvamiseks.

Nagu näidatud, on koodikaetuse analüüsi sisendiks mitte otseselt C++ lähtekoodifailid, vaid üksuste testide laademoodulid (`proj_test.exe`), mis tekivad (testide) lähtekoodi kompileerimise tulemusel ehk MSBuildi väljundina.

Kõiki neid protsesse (üksuste testide kompileerimist, koodikaetuse ja staatilise analüüsi vahendeid ning SonarQube Scannerit) võiks esialgse idee poolest käivitada kas või otse QuickBuildist. Seda eeldusel, et tegemist on lihtsal viisil käideldavate utiliitidega, mis ei vaja sisendite/väljundite eel- ega järeltöötlust ja ei eelda keerulisi käsureaparameetreid. Vastasel korral võiks osa protsessidest automatiseerida skriptide vahendusel ja lasta skript käima QuickBuildist. See võimaldaks just tarkvaraarendajatel (mitte koosteprotsessi tiimil) edaspidigi hooldada ka nimetatud keeruliste skriptide lähtekoodi, hoides seda koos muu lähtekoodiga SVN-repositooriumis, mitte aga QuickBuildi konfiguratsioonis.

Kuna konkreetsete tööriistade valik ei ole selles staadiumis veel teada (nendest tuleb juttu järgmises jaotises), on joonisel utiliitide nimede asemel kasutatud vastavate protsesside olemust kirjeldavad nimetused (staatilise analüüsi vahend, koodikaetuse analüsaator).

2.9 Lähtekoodi kvaliteedi analüüsi vahendite valik

SonarSource'i poolt pakutav ametlik C++ analüsaator[1] eeldab paketi *Developer Edition* soetamist, mille litsents on lähtekoodi suure hulga (ettevõttes, kus autor töötab, on C++ lähtekoodi kogumaht üle miljoni koodirea) puhul kõrge aastatasuga. Seetõttu otsustati seda mitte soetada.

Ametlikule tasulisele C++ analüsaatorile leidub avatud lähtekoodiga tasuta alternatiiv: SonarQube C++ Community plugin[2]. Selle plugina kasutamiseks tuleb aga valida mõni väline analüüsivahend ja lõimida see pluginaga. Ettevõte otsustas selles osas eelistada avatud lähtekoodiga vahendeid.

---

[1] SonarCFamily for C/C++, https://sonarsource.com/products/codeanalyzers/sonarcfamilyforcpp.html

[2] SonarQube C++ Community plugin, https://github.com/SonarOpenCommunity/sonar-cxx

Selguse huvides olgu mainitud, et käesolevas bakalaureusetöös kirjeldatud tegevused leidsid aset 2016. aasta lõpukuudel, C++ arendusvahendina oli ettevõttes tol ajal kasutusel Microsoft Visual Studio 2012 ning analüüsimist vajav C++ lähtekood on vähemalt teatud osas üsna Windowsi-spetsiifiline.

Kaalumisel olid järgmised lähtekoodi staatilise analüüsi vahendid.

- PVS-Studio on (C, C++, C# ja Java) lähtekoodi staatiline analüsaator[1]. Kuid see on tasuline ja selle lähtekood on kinnine.
- Coverity toodete perest (Synopsys) on tasuta saadaval Coverity Scan, kuid ainult avatud lähtekoodiga projektide analüüsimiseks[2]. Ettevõtte lähtekood on seevastu kinnine.
- Cpplint on vahend, mis võimaldab kontrollida koodi vastavust Google'i loodud C++ lähtekoodi stiilijuhistele[3]. Ettevõtte lähtekoodis aga seda stiili ei järgita, vaid on kasutusel teistsugune, ettevõtte oma stiil.
- Clang sisaldab samuti lähtekoodi staatilist analüsaatorit[4]. Tegemist on avatud lähtekoodiga vahendiga. Ettevõtte hinnangul (tolle valiku tegemise aegu) Clang aga ei olnud Windowsi platvormil kasutuselevõtuks veel piisavalt küps.
- Frama-C on avatud lähtekoodiga platvorm. Nagu selle platvormi loojad ise on kirjeldanud [8], võimaldab Frama-C arhitektuur kasutada väga mitmesuguseid pluginaid erinevat tüüpi analüüsi teostamiseks. C++ keele tugi on võimalik LLVM kompilaatorile suunatud Frama-Clang plugina vahendusel[5]. See töövahend jäeti kõrvale samal põhjusel nagu Clang.
- Infer on Facebooki arendajate poolt loodud staatiline analüsaator (kontrollib nii Java kui ka C, C++ või Objective-C lähtekoodi), kuid see ei toeta Microsoft Windowsi platvormi[6]. Ka mitmed teised utiliidid on Linuxi-põhised ja seetõttu ei sobinud.

Koodi kaetuse analüsaatoritest on tuntud Gcov, kuid selle kasutusjuhendis [9] väidetakse, et see töötab see üksnes GNU Compiler Collection (GCC) abil kompileeritud koodiga.

---

[1] PVS-Studio: Static Code Analyzer for C, C++, C# and Java, https://www.viva64.com/en/pvs-studio/

[2] Coverity Scan – Frequently Asked Questions (FAQ), https://scan.coverity.com/faq (ingl, 21.05.2019)

[3] Cpplint – static code checker for C++, https://github.com/cpplint/cpplint (ingl, seisuga 21.05.2019)

[4] Clang Static Analyzer, https://clang-analyzer.llvm.org/ (ingl, seisuga 21.05.2019)

[5] The Frama-Clang plugin, https://frama-c.com/frama-clang.html (ingl, seisuga 21.05.2019)

[6] Infer static analyzer | Support, https://fbinfer.com/support (ingl, seisuga 21.05.2019)

Töö eesmärgiks on SonarQube'iga toimiva lahenduse saamine, mitte aga niivõrd tööriista valimine. Sellised töövahendid, mida tundus olevat lihtne käima saada, on Cppcheck ja OpenCppCoverage, seetõttu need valitigi. Järgmistes alajaotistes on neist lähemalt juttu.

2.9.1 Cppcheck

Avatud lähtekoodiga vaba tarkvara Cppcheck on C ja C++ lähtekoodi staatilise analüüsi tööriist[1]. Erinevalt mitmest eelmainitud vahendist toetab Cppcheck ka Microsoft Visual Studiot. Cppcheckil on olemas nii konsoolipõhine kui ka graafiline kasutajaliides [10].

Cppchecki kasutajaskond on üsna suur, seega saab kogukond pakkuda paremat tuge kui mõne vähelevinud utiliidi puhul. Cppcheck on olemas olnud juba pikka aega (esimene versioon ilmus aastal 2007) ning seda on teiste analüsaatoritega võrreldud nii 2011. aastal ilmunud artiklis [11] C-keelses koodis defektide leidmise võimekuse kohta (artikli autorid on G. Chatzieleftheriou ja P. Katsaros) kui ka 2018. aastal avaldatud Jonathan Moermani bakalaureusetöös [12], kus osaliselt kasutati sama testikomplekti, kuid uuemat Cppchecki versiooni.

Cppchecki kasuks räägib ka tõik, et sellel utiliidil ilmuvad regulaarselt uued versioonid, mis viitab jätkuvalt aktiivsele arendusele.

2.9.2 OpenCppCoverage

Avatud lähtekoodiga vaba tarkvara OpenCppCoverage on C++ koodi kaetuse analüsaator Microsoft Windowsi platvormile[2]. Utiliit toetab Microsoft Visual Studios kompileeritud laademoodulite analüüsi, kasutades PDB (*Program database*) faile.

Tulemusena väljastab OpenCppCoverage aruande, milles on märgitud laademooduli (tüüpiliselt mõne tarkvaraüksuse testi) töö tagajärjel käivitatud (või käivitamata jäetud) koodiread, samuti on arvutatud iga lähtekoodifaili kaetuse protsent. Aruande vorming on käsureaparameetritega määratav: SonarQube'i tarvis saab kasutada Cobertura vormingut, kuid valida saab muuhulgas näitliku HTML-vormingu (sellises väljundis on lingid, sektordiagrammid jm inimloetav sisu).

---

[1] Cppcheck – A tool for static C/C++ code analysis, http://cppcheck.sourceforge.net/ (ingl, 21.05.2019)

[2] OpenCppCoverage'i repositoorium GitHubis, https://github.com/OpenCppCoverage/OpenCppCoverage

3 Valitud vahendite liidestamine SonarQube'iga

Käesolevas peatükis seletatakse põhjalikult lahti, kuidas autoril õnnestus kohandada valitud vabavaralised koodikvaliteedi analüüsi utiliidid (Cppcheck ja OpenCppCoverage) nii, et nende tulemeid saaks vastavate kvaliteedinäitajate ja -aruannete vaatamiseks SonarQube serverisse edukalt üles laadida SonarQube Scanneri vahendusel, pidades silmas valitud vabavaralise SonarQube C++ Community plugina võimalusi.

Kõigepealt kirjeldatakse OpenCppCoverage'i kasutamist kogu vajaliku C++ lähtekoodi kui terviku kaetuse analüüsimiseks. Mainitakse ka spetsiaalse PowerShelli skripti loomise vajadust OpenCppCoverage'i väljundi omapäradest tingituna ja peatutakse pikemalt selle skripti toime üksikasjadel.

Seejärel puudutatakse Cppchecki (staatilise analüüsi utiliidi) sisendite ja väljundite teemat ning pikemalt kirjeldatakse selles tööriistas endas ilmnenud vigade parandamist Cppchecki avatud lähtekoodis.

Samuti tuleb juttu QuickBuildi konfigureerimisest kogu protsessi käivitamiseks vajalikul viisil ja MSBuildi rakendamisest QuickBuildis.

Peatüki viimases alajaotises on üksikasjalikult kirjeldatud analüüsiaruannete loomise protsessi haldamisega tegeleva peamise skripti toimet ning näidatud ka kogu lahenduse skeem nii, nagu see lõpuks tegelikult teostatud sai.

3.1 OpenCppCoverage'i liidestamine SonarQube'iga

Koodikaetuse analüüsimise eelduseks on üksuste testide olemasolu. Kõik jaotises 2.5 kirjeldatud viisil loodud üksuste testid kompileeritakse lahendusfailide kaupa spetsiaalses testikonfiguratsioonis `Test Debug`. Koodikaetuse analüüsi tarbeks kasutatakse just nimelt `Test Debug` (mitte `Test Release`) konfiguratsiooni, et vältida kompilaatori poolt koodi optimeerimisest tingitud ebatäpsusi reanumbrite tuvastamisel. Nagu OpenCppCoverage'i

vikilehekülje[1] selgitatakse, võib optimeerimine teatud juhtudel muuta originaalfailis reanumbrite tuvastamise sootuks võimatuks.

Kompileerimise tulemusena tekivad igale projektile (millel on olemas `Test Debug` testikonfiguratsioon) vastavad laademoodulid. Nagu jaotises 2.5 mainitud, on iga selline laademoodul oma olemuselt konsooliprogramm, mis sisaldab ühte või mitut üksuste testi.

Käivitamist vajava järjekordse laademooduli nimi antakse OpenCppCoverage'ile ette käsureal. Täpsemalt on seda kirjeldatud alajaotises 3.4.2, kus seletatakse lahti analüüsi automatiseerimiseks loodud peamise skripti sisu.

Iga sellise testimisprogrammi käivitamisel OpenCppCoverage'i vahendusel märgib OpenCppCoverage üles iga kaasatud päise- ja lähtekoodifaili kohta kõik reaalset (käivitatavat) koodi sisaldavate koodiridade numbrid ning seda, kas vastav rida sai testimise käigus ka tegelikult käivitatud või mitte. Samuti arvutab OpenCppCoverage kaetuse protsendi igas asjassepuutuvas koodifailis.

Infot laademooduli töötamise tagajärjel kaasatud koodifailide kohta ammutab OpenCppCoverage linkuri tekitatud PDB-failidest, millele laademoodulites on viidatud.

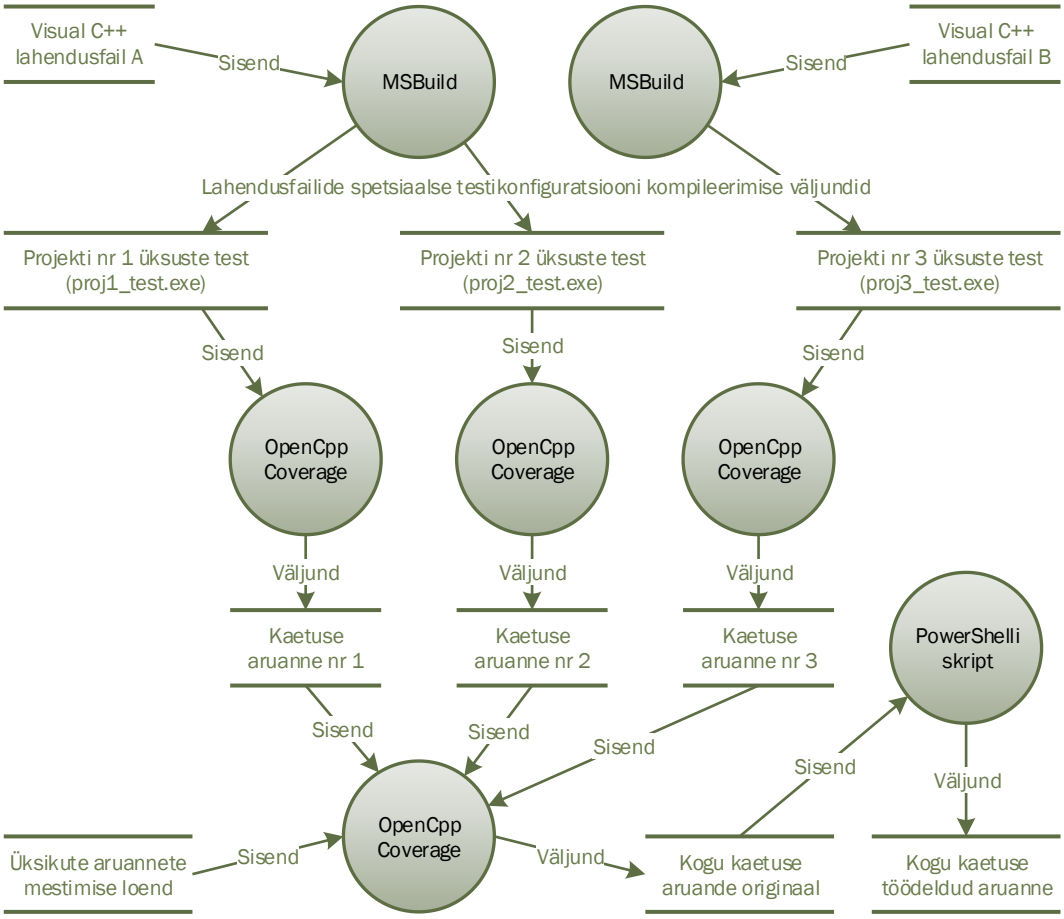
3.1.1 Üksikute väljundite koondamine tervikuks

OpenCppCoverage'i väljundiks on küll kaetuse aruanne, kuid ühe laademooduli käivitamise tulemusel ei kajasta selline üksik aruanne kogu koodibaasi kaetust. Seetõttu tuleb kõigepealt järjest käivitada OpenCppCoverage'is kõik laademoodulid ja seejärel tekkinud aruanded kokku koondada.

Kokkukoondamise juhtimiseks kasutatakse erilist sisendfaili: üksikute aruannete mestimise loendit. Vastava loendi tekitab põhiskript, mis on üksikasjalikult kirjeldatud alajaotises 3.4.2. Sellisel (viimasel) OpenCppCoverage'i käivitamisel ei tehta tegelikult muud kui lihtsalt mestitakse kõik üksikud kaetuse aruanded, tulemuseks on kõiki tulemusi sisaldav kogu kaetuse aruanne.

[1] OpenCppCoverage'i repositoorium GitHubis, vikilehe kodusektsiooni alajaotis „Release versus Debug“, https://github.com/OpenCppCoverage/OpenCppCoverage/wiki#release-versus-debug (ingl, 14.02.2019)

Joonisel 3 on näidatud koodikaetuse analüüsi kulgemine OpenCppCoverage'i kasutamisel. Viimane samm selles protsessis, mis viib SonarQube'ile sobiliku aruande tekkimiseni (kasutades PowerShelli skripti), on seletatud järgnevas alajaotises.



Joonis 3. Koodikaetuse analüüsi üksikasjalik andmevoodiagramm.

Nagu kirjeldatakse lähemalt alajaotises 3.4.2, antakse lõplik kogu kaetuse töödeldud aruanne SonarQube Scannerile ette. SonarQube Scanneri konfiguratsioonifailis on selle jaoks SonarQube C++ Community plugina parameeter `sonar.cxx.coverage.reportPath` (vt Joonis 1).

3.1.2 Väljundi teisendamine Cobertura vormingule

Üks koodikaetuse aruande vormingutest, mida SonarQube C++ Community plugin toetab oma sisendina, on Cobertura oma. See on XML-põhine vorming, mille võttis kasutusele samanimeline avatud lähtekoodiga utiliit Java koodi kaetuse analüüsimiseks. Sõna *cobertura* tähendab hispaania (ja portugali) keeles kaetust, siit tuligi utiliidi nimetamise

mõte[1]. OpenCppCoverage väljastab Coberturaga ühilduvas XML-vormingus aruannet, kui kasutada `--export_type=cobertura` käsureaparameetrit [13].

Kuid OpenCppCoverage'i aruannete SonarQube serverisse üleslaadimisega katsetamisel selgus väike probleem. Nimelt, SonarQube Scanner paistab aruandes viidatud failide otsingul eeldavat, et lähtekoodifailide teekonnad on aruandes antud ette analüüsitava lähtekoodi juurkausta suhtes, seevastu OpenCppCoverage'i poolt tekitatavas Cobertura XML aruandes on teekonnad antud ette draivi juure suhtes (ehk tegemist on sisuliselt absoluutteekondadega). Ebakõla tulemusel ei saa kaetuse aruannet SonarQube serveris üksikasjalikult sirvida, kuna vastavaid faile ei leita.

Selle probleemi lahendamiseks sai kirjutatud PowerShelli skript, mille lähtekood on toodud lisas 2. Skriptimiskeelena sai valitud PowerShell, sest vastav interpretaator on kõikides koosteprotsessi kaasatud Windowsi-põhistes virtuaalmasinates olemas ning, erinevalt Windowsi konsooli käskude keelest, võimaldab PowerShell üsna mugavat XML-failide toimetamist, vajadusel kasutades XPath päringukeelt. Järgnevalt kirjeldame üksikasjalikult skripti toimet.

Kõigepealt, ridadel nr 5 kuni 11 on määratletud skriptile etteantavad käsureaparameetrid:

- aluskausta teekond (parameeter `-BaseDir`), s.o uus teekond, mille suhtes tuleb korrigeeritud väljundis kõikide lähtekoodifailide teekonnad ümber sättida;
- sisendfaili teekond/nimi (parameeter `-InputFile`), s.o OpenCppCoverage'i algse väljundina saadud kaetuse aruande fail, mida hakatakse töötlema;
- väljundfaili teekond/nimi (parameeter `-OutputFile`), s.o kaetuse korrigeeritud aruande fail, milles on failiteekonnad uue aluskausta suhtes ümbersätitud;
- logifaili teekond/nimi (parameeter `-LogFile`), s.o fail, kuhu logitakse PowerShelli skripti kogu töö üksikasju.

Kõik need parameetrid on kohustuslikud, et oleks skripti kasutamisel ei jääks juhuslikult midagi märkamata ja et väärkasutamisest tekkida võivaid vigu oleks hõlpsam avastada

[1] Cobertura repositoorium GitHubis, vikilehe sektsioon „FAQ“, https://github.com/cobertura/cobertura/wiki/FAQ#where-does-the-name-cobertura-come-from (ingl, 8.09.2014)

(tõsi, aluskausta teekond võib olla tühi sõne, andmaks märku, et see on meelega välja jäetud).

Järgnevatel ridadel asuvad kõigepealt skripti sisemised funktsioonid (read nr 13 kuni 57), alles seejärel (alates reast nr 59 kuni koodi lõpuni) on skripti põhiplokk.

Üks sisemine funktsioon, nimega `Log` (ridadel nr 13–18), võtab parameetrina mingi teksti (näiteks veateate) ja lisab selle teksti logifaili (vt ülalmainitud parameetrit `-LogFile`) lõppu, koos ajatempliga.

Teine funktsioon, nimega `IsFullPath` (read nr 20–24), tuvastab, kas argumendina etteantud teekond on täielik (absoluutne). Kui on, siis tõeväärtusena tagastab tõese. Kui aga on tegemist suhtelise teekonnaga, siis tulemiks on väär. Kuna skripti kasutatakse ainult konkreetse utiliidi väljundi töötlemiseks, on siin tehtud üldisel juhul lubamatu lihtsustus: tegelikult kontrollib funktsioon ainult seda, kas etteantud teekonna pikkus on vähemalt kaks tärki ja kas teisel kohal asetsev tärk on koolon. Antud skripti kontekstis loetakse suhtelisteks teekondadeks ka draivi juure suhtes antud teekondi, mis ei sisalda draivi tähist (vaatamata sellele, et Microsofti terminoloogia järgi on need ka Windowsis siiski absoluutteekonnad[1]).

Põhiline funktsioon, nimega `Rebase` (read nr 26–57), teisendab esimese argumendina etteantud failiteekonna nii, et kui see oli algselt suhteline teise argumendina etteantud vana aluskausta suhtes (või oli absoluutne), siis teisenduse järel on see failiteekond suhteline kolmanda argumendina etteantud uue aluskausta suhtes (seda juhul, kui selline teisendus on võimalik). Selleks teisendatakse etteantud failiteekond kõigepealt täielikuks absoluutseks teekonnaks, kasutades vajadusel teavet vana aluskausta kohta. Seejärel aga eemaldatakse eelnevas sammus teisendatud failiteekonda tähistava sõne algusest uue aluskausta teekonda tähistav alamsõne, juhul kui võimalik (tagastatav uus teekond on seega suhteline). Kui alamsõne eemaldamine pole võimalik, tagastatakse aga absoluutne teekond. Funktsioon logib neid tegevusi, samuti loendab suhtelisi / absoluutseid teekondi statistika kogumiseks.

---

[1] Microsoft Docs, artikli „Naming Files, Paths, and Namespaces“ alajaotis „Fully Qualified vs. Relative Paths“, https://docs.microsoft.com/en-us/windows/desktop/FileIO/naming-a-file#fully-qualified-vs-relative-paths (ingl, 31.05.2018)

Edasi, skripti põhiplokis, pärast `-BaseDir` parameetri korrektsuse kontrolli, luuakse uus XML-tüüpi objekt (rida nr 84), millesse laaditakse sisendfaili sisu (rida nr 85). Siinkohal on objekti tüübiks tegelikult .NET klass nimega `System.Xml.XmlDocument`, kuid loetavuse huvides kasutatakse lihtsalt `XML` tüübilühendit [14].

Alates reast nr 87 kuni reani nr 137 toimub sisendfailis leiduvate `<source>`-elementide kontroll ja muutmine. Selleks kogu sisendit hõlmava vastloodud XML-objekti kallal rakendatakse `SelectNodes` meetodit[1], millele antakse ette `<source>`-elementide teekond XPath avaldisena. Kui vähemalt üks `<source>`-element eksisteerib (read nr 88–120), siis kontrollitakse, kas selle sisuks on aluskausta täielik teekond: kui jah, siis asendatakse see uue aluskausta teekonnaga, vastasel korral kuvatakse veateade ja skripti töö lõpetatakse. Kui `<source>`-elemente on mitu, siis peaksid need olema duplikaadid (ehk erinema vaid suur- ja väiketähtede kasutamise poolest), vastasel juhul lõpeb skript veateatega.

Kui aga ühtegi `<source>`-elementi ei eksisteeri (read nr 121–137), siis kuvatakse hoiatus, et ükski failiteekond sisendis ei tohi sel juhul olla suhteline (vastasel korral ei ole ju täielikku teekonda võimalik tuvastada), ning luuakse `<source>`-element, mille sisuks on skriptile ette antud uue aluskausta teekond. (Kui `-BaseDir` argument on tühi sõne, ehk uut aluskausta pole määratud, siis väljundisse tulevad vaid täielikud failiteekonnad; sel juhul eemaldatakse kogu `<sources>`-element, vt read nr 139–147.)

Ridadel nr 150 kuni 154 toimub kõigi sisendfailis leiduvate failiteekondade teisendamine nii, et need põhineks vana aluskausta teekonna asemel uue aluskausta teekonnal. Selleks valitakse tsüklis kõik XML-objektis leitud `filename`-atribuudid, kasutades PowerShelli `Select-Xml` käsku [15] koos vastava XPath avaldisega (vt rida nr 143), seejärel kutsutakse iga leitud `filename`-atribuudi muutmiseks funktsiooni `Rebase` (vt rida nr 153).

Lõpuks, ridadel nr 157 kuni 163 salvestatakse kogu muudetud XML-objekt väljundfaili ning ridadel nr 165–168 logitakse statistikat ja lahkutakse skriptist. Read nr 170–176 on juhuks, kui skripti töö käigus peaks tekkima mõni erind (ingl *exception*): see logitakse ja skript tagastab veakoodi.

---

[1] Microsoft Docs, .NET raamistiku dokumentatsioon, veebilehekülg „XmlNode.SelectNodes Method“, https://docs.microsoft.com/en-us/dotnet/api/system.xml.xmlnode.selectnodes (ingl, seisuga 19.05.2019)

3.2 Cppchecki liidestamine SonarQube'iga

Lähtekoodi kvaliteedi staatilise analüüsi aruannete automatiseeritud viisil saamiseks ja SonarQube serverisse üleslaadimiseks on kõigepealt vaja Cppcheck (konsoolirakenduse variant, mitte graafilise kasutajaliidese oma) käivitada, andes käsureaparameetritena ette lähtekoodi asukohta ja väljundi vormingut määravad suvandid. Selle sammu üksikasjad on välja toodud järgnevates alajaotistes 3.2.1 ja 3.2.2.

Cppchecki aruande valmimisel tuleb see edastada SonarQube Scannerile. Sellest on juttu alajaotises 3.2.2. Kogu selle protsessi juhtimiseks on loodud lisas 1 näidatud peamine skript (Windowsi pakkfail).

3.2.1 Cppchecki sisendi etteandmise viisid

Cppchecki ametlikus kasutusjuhendis [10] on kirjeldatud kaks erinevat viisi, kuidas saab staatilist analüüsi vajavate lähtekoodifailide asukohti ette näidata. Üks neist on lihtsalt käsurea lõpus, pärast kõiki suvandeid, loetleda kõikide analüüsitavate failide/kaustade teekonnad/nimed. Seejuures kausta etteandmisel analüüsitakse kõiki selles leiduvaid lähtekoodifaile. Näiteks, kui käsurea lõppu panna punkt (`.`), siis see tähendab, et analüüsi vajavad kõik töökaustas olevad C++ failid.

Sellise n-ö käsitsi etteandmise viisi korral saab käsurea kaudu suvanditega määrata ka päisefailide täiendavaid otsinguteekondi (kas üksikute `-I` suvandite kaupa eraldi või korraga kogu otsinguteekondade nimekirja sisaldava sisendfailiga `--includes-file` suvandiga), välistatud kaustu (`-i` suvandiga), makrodefinitsioone (`-D` suvandiga) ja platvormi (`--platform` suvandiga).

Teine viis, vähemalt Microsoft Visual Studio tarkvaralahenduste korral, on anda ette üksnes kõikide lahendusfailide nimed (järjest kas või mitme) `--project` suvandi abil. Sel juhul analüüsitakse vastavates lahendus- ja/või projektifailides viidatud lähtetekstifaile, võttes arvesse projektifailide atribuutides määratud otsinguteekondi, makrodefinitsioone ja platvorme. Käsitsi etteantud vastavasisulisi suvandeid `--project` suvandi olemasolul aga ignoreeritakse.

3.2.2 Cppchecki väljundi kasutamine

Lähtekoodi staatilise analüüsi aruande edasiseks automatiseeritud töötlemiseks, antud juhul SonarQube serveris, on vaja lülitada Cppcheckis sisse XML-vormingus väljundi

loomine. Selleks saab anda käsureal ette parameetrina kas `--xml` (s.o vanem vorming) või `--xml-version=2` (s.o uuem vorming). Alajaotises 3.4.2 kirjeldatud põhiskript määrab Cppchecki vastava parameetriga just uuemas XML-vormingus aruande tekitamise.

XML-väljundi saamiseks tuleb pidada meeles, et selle väljastab Cppcheck standardse veateadete voo kaudu, mitte standardse väljundvoo kaudu.

Nagu kirjeldatakse alajaotises 3.4.2, antakse staatilise analüüsi aruanne ette SonarQube Scannerile, mille konfiguratsioonifailis on selle jaoks SonarQube C++ Community plugina parameeter `sonar.cxx.cppcheck.reportPath` (vt Joonis 1).

3.3 Cppchecki töös ilmnenud defektid

Katsetades Cppchecki (versiooni 1.76.1 ehk tollal uusima) tööd sisendi etteandmisel Visual Studio lahendusfailide kaupa, kohe ilmnes märkimisväärne erinevus nii analüüsi teostamisele kuluvas ajas kui ka sellises utiliidi enda poolt raporteeritud statistilises näitajas nagu analüüsi läbinud lähtekoodifailide koguarv. Sellest sai oletada, et enamus lähtekoodifailidest mingil põhjusel ei läbigi analüüsi.

Oletuse kontrollimiseks käivitati Cppcheck spetsiaalses konfiguratsiooni kontrollimise režiimis (kasutades käsureal suvandit `--check-config`), milles lähtekoodi analüüs on välja lülitatud ja toimub vaid Cppchecki seadistamist puudutavate probleemide otsing. Tulemuseks kuvati suur hulk informatsiooni tasemel teateid päisefailide mitteleidmise kohta. Seega on oletus osutunud tõeks: lahendusfailide kaupa etteandmisel jääb enamus lähtekoodist analüüsist lihtsalt välja, kuna ei leita paljusid vajalikke päisefaile.

3.3.1 Lahendusfailide puudulik parsimine

Lihtne võimalus kontrollida, millistest kaustadest Cppcheck päisefaile otsib, on kasutada käsureal suvandit `--verbose`, siis kuvatakse ka eelprotsessori makrodefinitsioone ning päisefailide otsinguteekondi. Niiviisi talitades sai peagi selgeks, et failide mitteleidmise põhjus peitub asjaolus, et teekondade konstrueerimise käigus ei tuvastata selliseid makrodefinitsioone nagu `$(SolutionDir)` – need jäävad teekondadesse töötlemata kujul. Kasutaja poolt kohandatud atribuudifailides sisalduvad definitsioonid jäävad samuti parsimata.

Tarkvaralahenduses kasutatavate täiendavate otsinguteekondade etteandmine käsureal suvandite `-I` ja `--includes-file` kaudu pole aga võimalik, sest lahendusfailide režiimi (ehk `--project` suvandi) kasutamise puhul need sihilikult ignoreeritakse.

Autor küsis Stack Overflow kogukonnast nõu[1]. Cppchecki projekti eestvedaja ja põhiline arendaja Daniel Marjamäki isiklikult vastas[2], et lahendusfailide etteandmise režiim on liiga värske võimalus (ja seetõttu ei pruugi veel täielikult töötada). Cppchecki programmivigade jälgimise keskkonnas loodi ametlik sissekanne[3].

3.3.2 Cppchecki lähtekoodi korrigeerimine

Esimene samm programmivea korrigeerimiseks oli Cppchecki põhiarendaja palvel luua väike ja korratav testjuht. Esinedes Cppchecki programmivigade jälgimise keskkonnas varjunime „bug22“ all, lõigi autor sellise testjuhu, arvestades ettevõttes tarvitusel olevat praktikat kasutada kohandatud atribuudifailide hierarhiat.

Testjuht sisaldab tervet kaustapuud, mille juurkaustas asub üks Visual Studio lahendusfail (`solution.sln`), mis sisaldab ühte projekti. Erinevates kaustades ja erineval tasemel alamkaustades sisaldub kokku kaheksa erinevat päisefaili. Igas päisefailis on ainult järjekordse konstandi deklaratsioon.

Lisaks on projektis olemas üks lähtekoodifail: konsooliprogrammi kood, mis kaasab kõik päisefailid ja tagastab kõikide konstantide summa. Seega on kõik päisefailid vajalikud, st `main`-funktsiooni keha parsimiseks ei saa ühtegi päisefaili välja jätta. Enamus päisefailidest kaasatakse üksnes failinime järgi. Paaril juhul kaasatakse päisefail koos selle faili ülemkausta nimega, kusjuures meelega on valitud kaustapuus kahes erinevas kohas asetsevatele kaustadele sama nimi. Nii saab veenduda, et ühe otsinguteekonna luhtumisel jätkatakse otsingut järgmise etteantud otsinguteekonnaga.

---

[1] Stack Overflow kasutaja „heap underrun“ (s.o autori varjunimi) küsimus „Cppcheck doesn't expand Visual Studio project macros“, https://stackoverflow.com/q/40345379 (ingl, 31.10.2016)

[2] Stack Overflow kasutaja Daniel Marjamäki vastus küsimusele „Cppcheck doesn't expand Visual Studio project macros“, https://stackoverflow.com/a/40452354 (ingl, 6.11.2016)

[3] Cppcheck Trac, kasutaja „orbitcowboy“ sissekanne „Ticket #7791: Cppcheck does not expand Visual Studio macros“, https://trac.cppcheck.net/ticket/7791 (ingl, 1.11.2016)

Lahendusfailis on kaks konfiguratsiooni (`Debug` ja `Release`), mis on otsinguteekondade poolest identsed ja millede eraldamise ainsaks otstarbeks on veenduda, et Cppcheck kontrollib ära mõlemad konfiguratsioonid.

Projektifailis otsinguteekondade määramisel on makrodena kasutusel nii standardsed [16], nagu `$(SolutionDir)` ja `$(ProjectDir)` kui ka kasutaja poolt määratud [17] makrod `$(Custom_PerProject_IncDir)` ja `$(Custom_String)`. Projektifail kaasab kohandatud atribuudifaili `custom.props`, mille teekond on ette antud makro `$(SolutionDir)` abil (ehk suhtelisena, aluseks lahendusfaili asukoht).

Atribuudifaile on kaks. Neist esimene fail, `base.props`:

- defineerib makro `$(Custom_Substring)` lihtsa sõnena, mis sisaldab muuhulgas kurakaldkriipsu, et veenduda kaustanimede eraldaja korrektses toimimises isegi sõnede konkatenatsiooni järgselt;
- defineerib makro `$(Custom_PerProject_IncDir)`, mis mh kasutab standardset makrot `$(ProjectDir)`;
- lisab täiendava otsinguteekonnana veel ühe alamkausta, kasutades mh makrot `$(SolutionDir)`.

Teine atribuudifail – `custom.props` – kaasab faili `base.props` ning defineerib makro `$(Custom_String)`, mis omakorda kasutab alamsõnena makrot `$(Custom_Substring)`.

Niisiis ei ole ei see projektifail ega need kohandatud atribuudifailid eraldi võetuna kompileerimiseks mõeldud, küll aga sätitakse otsinguteekonnad lõplikult paika alles siis, kui alustada parsimist lahendusfailist. Just selline olukord valitseb ka autori töökohal.

Visual Studio 2012 kompileeris sellise testjuhu korrektselt, seevastu Cppcheck 1.76.1 (kasutades `--project="solution.sln"` suvandit) kõiki päisefaile ei leidnud.

Cppchecki peamine arendaja Daniel Marjamäki viis Cppchecki lähtekoodi sisse mitmed täiendused, et parandada testjuhuga esile toodud vead. Tutvunud Cppchecki uuendatud koodiga, leidis bakalaureusetöö autor ometi vajakajäämisi. Esiteks osutus, et testjuhu tarbeks spetsiaalselt loodud atribuudifaili `custom.props` nimi ja teekond sõna-sõnalt kodeeriti Cppchecki lähtekoodi sisse. Loomulikult on tegelikes projektides kasutusel hoopis erinevad atribuudifailid, ka nende asukohad võivad olla suvalised.

Teiseks, tundus ebakorrektne hakata atribuudifaile otsima / importima kogu lahendusfaili pealt ühiselt, vaid seda peaks tegema igast projektifailist eraldi. Üldjuhul kaasab iga projekt erinevaid atribuudifaile, mitte tingimata samu. Lisaks peaks Cppcheck olema võimeline analüüsima mitte ainult terveid tarkvaralahendusi, vaid ka üksikuid projekte.

Pärast repositooriumist Cppchecki värskendatud lähtekoodi allalaadimist ja oma ettevõtte reaalse tarkvaratoote analüüsimiseks katsetamist selgus, et Cppcheckis leidub ikka veel parandamata vigu, mis takistavad lahedus- ja projektifailide parsimist. Niisiis oli autor sunnitud looma veel teisegi testjuhu, demonstreerimaks teistsuguseid, seni välja toomata jäänud, viise atribuudifailide ja otsinguteekondade kasutamiseks.

Ka teine testjuht sisaldab tervet kaustapuud, kuid lahendusfail asub mitte juurkaustas, vaid eraldi kaustas. Teise kausta eraldi alamkaustades on kokku neli päisefaili.

Erinevalt esimesest testjuhust kaasatakse päisefailid kolme erinevasse lähtekoodifaili, kusjuures need lähtefailid on kolmes eraldi projektis. Niisiis tekib kompileerimisel kolm eraldi konsooliprogrammi. Nendele projektidele on omaette kaust kolme alamkaustaga (iga projekti jaoks oma).

Atribuudifaile on kolm:

- `alfa.props` (paikneb omaette kaustas) täiendab põhilist otsinguteekonda `$(IncludePath)` veel ühe alamkaustaga, kasutades mh standardset makrot `$(ProjectDir)`;
- `bravo.props` (paikneb esimese atribuudifaili kausta suhtes alamkaustas) kaasab atribuudifaili `charlie.props`, samuti standardse makro `$(ProjectDir)` kaasabil;
- `charlie.props` (paikneb hoopis ühe projekti alamkaustas) laiendab täiendavat otsinguteekonda `%(AdditionalIncludeDirectories)` veel ühe alamkaustaga, ikka makro `$(ProjectDir)` kaasamisega.

Ühes projektifailis kaasatakse atribuudifail `alfa.props`, mille teekond on ette antud makro `$(ProjectDir)` abil ehk suhtelisena, aluseks projektifaili asukoht, kusjuures kuna atribuudifail asub täiesti teises kohas, viidatakse teekonnas topeltpunktide abil kahe taseme võrra ülespool asuvale kaustale ja sealt edasi tolle alamkaustale: `$(ProjectDir)..\..\something\alfa.props`.

Teises projektifailis kaasatakse sarnasel viisil atribuudifail `bravo.props`.

Kolmandas projektifailis ühtegi kohandatud atribuudifaili ei kaasata (sest selle projekti kontekstis vajalike päisefailide asukohad ei ole atribuudifailides määratud). Selle asemel täiendatakse otsinguteekondi `$(IncludePath)` ja `%(AdditionalIncludeDirectories)` otse projektifailis, määrates õigete päisefailide teekonnad.

Pärast Cppchecki peamise arendaja järjekordseid parandusi selgus, et uuendatud Cppcheck ikka ei leia ühte päisefailidest esimesest testjuhust ja ühte ka teisest testjuhust. Värsket lähtekoodi uurides tekkisid taas mõned mõtted, mis võiks valesti olla, nimelt:

- täiendava otsinguteekonna nimistu järkjärgulisel konstrueerimisel iga leitud `<AdditionalIncludeDirectories>` elemendi väärtuse kokku kogumise teel kasutati Cppcheckis eraldajana semikooloni asemel ekslikult plussmärki, mille tõttu seda ei õnnestunud hiljem tükeldada mujal kasutatavaks sõnede loendiks;
- elemendi `<AdditionalIncludeDirectories>` töötlemiseks on Cppchecki koodis loodud vastav struktuur, kuid see oli kasutusel ainult projektifaili importimise funktsioonis, seevastu atribuudifailide laadimise funktsioonis oli seda struktuuri kasutamata leiutatud jalgratast praktiliselt samasuguse koodijupi näol;
- element `<IncludePath>` (mis esineb elemendi `<PropertyGroup>` sees) võib olla ette antud otse projektifailis, mitte ainult atribuudifailis (teises testjuhus seda testitakse kolmandas projektifailis), kuid Cppcheck sellise võimalusega ei arvestanud;
- tuleb arvestada, et element `<IncludePath>` saab olla ka kumuleeruv, kui selle väärtus sisaldab makrot `$(IncludePath)`, samamoodi nagu võib kumuleeruda ka `<AdditionalIncludeDirectories>` makro `%(AdditionalIncludeDirectories)` kasutamisel.

Autor taasavas vahepeal parandatuks märgitud ja kinni pandud programmivea sissekande, kuna vaatamata makrode töötlemisele, ei leidnud suvand `--project` mõnedel juhtudel kõiki otsinguteekondi. Ülaltoodud tähelepanekud kiitis Cppchecki arendaja heaks ja viis nende teostamiseks sisse veelgi täiendusi.

Siiski ei suutnud ka selliselt täiendatud Cppcheck leida kõiki kahes testjuhus sisalduvaid päisefaile. Selle tõiga avastamise ajal oli aga Cppchecki programmivigade raporteerimise keskkond paraku maas, seetõttu ei õnnestunud vastavat sissekannet seal enam taasavada.

Autor lõi Cppchecki ametlikus repositooriumis GitHubis tõmbetaotluse[1] (taas varjunime „bug22“ all). Nimelt, pärast koodi uurimist autor leidis ja lahendas kaks probleemi.

Üks probleem peitus üksuste määratlemise grupi tingimuse kontrolli funktsioonis. Kuna `Condition` atribuut on `<ItemDefinitionGroup>` elemendis mittekohustuslik ning selle otstarve on vaid kitsendavate tingimuste seadmine grupi kehtivusele, siis see tähendab, et kui ühtegi tingimust ei ole määratud (ehk selle atribuudi puudumise korral), peaks üksuste grupi määratlus kehtima kõikide konfiguratsioonide korral (tingimusteta). Seni tagastas Cppchecki vastav funktsioon (mis kontrollib, kas tingimus on rahuldatud) sellisel juhul tõeväärtusena väära. Seetõttu jäeti üksuste määratlemise grupid, milledel rakendumist kitsendav tingimus puudus, hoopis rakendamata. Autor muutis funktsiooni nii, et puuduva või tühja tingimuse korral tagastataks tulemina tõest. Teisisõnu, kui üksuste grupile ei ole kitsendavat tingimust seatud, siis võib seda pidada antud kontekstis sama hästi kui täidetuks, lubades vastava üksuste grupi rakendumist igal juhul, ilma kitsendusteta.

Teine probleem seisnes viisis, kuidas projektifaili importimise funktsioonis asustati failisuvandeid sisaldavat loendit. Seni paigutas kõige sisemine tsükkel loendisse iga leitud üksuste määratlemise grupi kohta eraldi failisuvandi. Autor usub, et selle asemel peaks iga failisuvand esindama projekti konfiguratsiooni kirjet. Näiteks, kui leidub mitu `Release` konfiguratsiooni kohta käivat üksuste määratlemise gruppi ja mitu `Debug` konfiguratsiooni kohta käivat, peaks Cppcheck käivitama analüüse ainult kaks korda: kord `Release` ja kord `Debug` konfiguratsiooniga (ehk eelprotsessori definitsioonidega, otsinguteekondadega jne). Seni käsitles Cppcheck iga üksuste määratlemise grupi kirjet justkui eraldi konfiguratsioonina. Võimalik, et see tundus mõttekas, kui oletati, et üksuste määratlemise grupp peab alati seadma mõne tingimuse, ja loodeti, et kõik üksused on määratletud ainsas grupis. Üldjuhul selline oletus alati paika ei pea, eriti atribuudifailide kasutamise korral.

Lühidalt, autor korraldas koodi ümber nii, et akumuleeritaks makrodefinitsioonid ja otsinguteekonnad kõikidest asjassepuutuvatest üksuste määratlemise gruppidest ning et failisuvandeid kogutaks projektis ainult tervete konfiguratsioonide kaupa kokku.

---

[1] Cppchecki repositoorium GitHubis, kasutaja „bug22“ tõmbetaotlus nr 849: „Fixed #7791 (Cppcheck does not expand Visual Studio macros)“, https://github.com/danmar/cppcheck/pull/849 (ingl, 7.12.2016)

Cppchecki arendaja Daniel Marjamäki mestis selle tõmbetaotluse ametlikku harusse. Tulemusena leidis Cppchecki versioon 1.77 üles kõik kahes testjuhus sisalduvad failid.

3.4 QuickBuildi konfiguratsioon kvaliteedianalüüsi sooritamiseks

Kvaliteedianalüüs peaks toimuma regulaarselt, näiteks iga nädala tagant pühapäeviti (nagu lahenduse juurutamisel selgus, võtab ettevõtte C++ koodibaasi kvaliteedi analüüs umbes viis tundi aega). Selleks sai analüüsikonfiguratsiooni üldiste seadistuste lehel QuickBuildis pandud paika vastav perioodiline ajakava: `0 0 12 ? * SUN` (s.o *cron*-laadses vormingus, tähendab algusega pühapäeviti lõuna ajal).

Windowsi virtuaalmasinas, kus kõik QuickBuildi poolt juhitavad protsessid töötavad, tekitatakse mäluketas ja sinna luuakse eraldi nn tööruumi (ingl *workspace*) kaust koodianalüüsi tarbeks.

Juhul, kui analüüsiprotsess peaks luhtuma, saavad nii selle algataja (käsitsi käivitamise korral) kui ka käesoleva töö autor e-kirjaga teavituse probleemist. Sedasi sai QuickBuildis määratud analüüsikonfiguratsiooni teavituste lehel.

Analüüsi käivitamisel on QuickBuildi poolt täidetavad sammud on järgmised:

- C++ kogu lähtekoodi allalaadimine SVN-repositooriumist;
- koodianalüüsi vahendite (Cppcheck, OpenCppCoverage, SonarQube Scanner) allalaadimine SVN-repositooriumist;
- MSBuildi vana logifaili kustutamine, juhul kui selline fail on eelmisest korrast veel alles (vt `sonar.cxx.msbuild.reportPath` parameetrit joonisel 1);
- MSBuildi abil kõigis lahendusfailides `Test Debug` konfiguratsiooni eelmise korra kompileerimise tulemuste puhastamine;
- MSBuildi abil kõigi lahendusfailide `Test Debug` konfiguratsiooni kompileerimine (üksikasjade kohta vt järgnevat alajaotist 3.4.1);
- peamise skripti käivitamine ja selle tulemuste äraootamine.

Kõik sammud täidetakse ülaltoodud järjekorras ning ühtegi sammu ei asuta täitma enne, kui eelmine samm on edukalt lõppenud. (Ainsaks erandiks võib olla MSBuildi logifaili kustutamine, mida võib igal juhul sooritada.)

3.4.1 MSBuildi rakendamine QuickBuildis

MSBuildi rakendamisega seotud tegevusteks on QuickBuildis spetsiaalselt selleks ette nähtud sammude tüüp (MSBuild-tüüpi sammud). Ülaltoodud loetelus jaotises 3.4 on seda tüüpi samme kaks: eelmise kompileerimise tulemusfailide puhastamine ja testide uus kompileerimine.

Kui tulemuste puhastamine käib lihtsalt (sihtmärkide parameetriks, ehk ingl *targets*, tuleb määrata puhastamine, ehk ingl *clean*, ja loomulikult ette anda konfiguratsioon `Test Debug` ja lahendusfailide nimed), siis kompileerimise sammus tuleb lisaks sihtmärgiks ehitamise (ingl *build*) määramisele anda MSBuildile ette täiendavad käsureaparameetrid (välja *Additional Switches* kaudu).

Üks selline käsureaparameeter on `/maxcpucount`, mis võimaldab mitut lahendusfailis viidatut projekti ehitada samaaegselt, kasutades arvuti mikroprotsessori erinevaid tuumi, seega kiirendades kompileerimise protsessi [18].

Teine käsureaparameeter on `/fileLogger` (väljundi logimiseks faili), mille järel tuleb kolmandas parameetris `/fileloggerparameters:` määrata järgmised logimise suvandid: `Append;Encoding=UTF-8;Verbosity=detailed;LogFile=build/sonar/log/msbuild.log` (et logifaili ei kirjutataks üle, vaid täiendataks; et kodeeringuna kasutataks UTF-8; et määrata detailne logimistase; ning et näidata ette logifaili teekond). Neist suvanditest on eriti oluline detailse logimistaseme määramine (`Verbosity=detailed`), kuna SonarQube C++ Community plugin suudab MSBuildi logi parsimisel ammutada kasulikku infot (sh otsinguteekonnad, makrodefinitsioonid) vaid piisavalt paljusõnalise taseme (*detailed* või *diagnostic*) korral [19].

3.4.2 Analüüsiaruannete tekitamine ja SonarQube Scannerile ettesöötmine

Pärast seda, kui kõik üksuste testid saavad kompileeritud, juhib kogu edasist tegevust lähtekoodi kvaliteedi analüüsi aruannete tekitamiseks ja SonarQube Scannerile ettesöötmiseks spetsiaalselt seks tarbeks kirjutatud skript (nimetagem seda põhiskriptiks või peamiseks skriptiks), mis pannakse viimase sammuna käima QuickBuildi poolt (QuickBuildi sammude loetelu vt jaotisest 3.4). Tegemist on Windowsi konsooli käskude keeles [20] kirjutatud pakkfailiga, skripti lähtetekst on toodud lisas 1. Järgnevalt kirjeldame üksikasjalikult skripti toimet.

Kõigepealt, ridadel nr 7 kuni 10, seatakse üles mõned keskkonnamuutujad (õigem oleks antud juhul öelda küll konstandid, mitte muutujad) skripti tööks vajalikele kaustadele (ja skripti logifailile) mugavaks viitamiseks. Konteksti täpsustamiseks olgu mainitud, et need teekonnad on määratud kogu analüüsitava C++ lähtekoodi SVN-tüve (`trunk`) kausta suhtes ning et põhiskript ise paikneb failis `build\sonar\bin\scan-all.cmd` (seegi teekond on `trunk`-kausta suhtes). Et minimeerida skripti kõrvaltoimet, kasutatakse real nr 4 käsku `SETLOCAL`, seega on seatavad keskkonnamuutujad lokaalsed [20].

Edasi, real nr 13, seatakse skripti töökaustaks C++ lähtekoodi SVN-tüve kaust (selleks et lihtsustada edaspidist kaustadele viitamist). Seda tehakse arvestusega, et skriptifail ise paikneb alati sellises alamkaustas, mis on kolm taset allpool SVN-tüve. Sellega välditakse absoluutteekonna teadmisest sõltumist. Esiteks on asi selles, et kogu QuickBuildi poolt juhitav protsess töötab virtuaalmasinas, ja koodianalüüsi tarbeks seal ajutiselt luuakse eraldi mäluketas (mille draivi tähtnime võidakse otsustada muuta), kuhu luuakse nn tööruumi kaust (mille nime võidakse samuti muuta). Teiseks võidakse tulevikus otsustada analüüsida mitte ainult tüve, vaid ka mõnd versiooniharu repositooriumis. Seetõttu pole absoluutteekonna sissekodeerimine mõistlik. Käsu `CD` võti `/D` võimaldab vahetada korraga nii töökausta kui -draivi [20] (niisiis töötab skript korrektselt ka siis, kui seda käivitada nt käsitsi käsurealt, isegi kui konsooli töödraiviks on parajasti mõni teine draiv).

Real nr 17 lisatakse otsiteekonda (ehk keskkonnamuutujasse `PATH`) analüüsivahendite Cppcheck ja OpenCppCoverage ning Sonar Scanneri asukohad (ka neid vahendeid hoitakse repositooriumis ja laaditakse enne analüüsi käivitamist alla).

Viimased ettevalmistavad tegevused enne koodianalüüsi algust on ridadel nr 22 kuni 28 vanade logi- ja aruandefailide kustutamine (mõnel juhul koos vastavate kaustadega). See võimaldab soovi korral käivitada sama skripti mitu korda järjest (näiteks pärast mõne konfiguratsioonimuudatuse tegemist), alustades iga kord n-ö puhtalt lehelt.

C++ lähtekoodi staatiline analüüs käivitatakse ridadele nr 32 kuni 54 laotatud pika käsuga (katusemärk `^` võimaldab murda pikki käsuridu). Siin käivitatakse Cppcheck, andes ette Windowsi 32-bitisele platvormile (ja Windowsi API kasutamisega) kirjutatud C++ lähtekoodi analüüsiks tüüpilisi suvandeid. Punkt 52. real on käsurea viimane parameeter, tähistades analüüsitavate kaustade loetelu: kuna kaustanimeks on punkt, siis tähendab see kogu töökausta (antud juhul `trunk`) rekursiivset analüüsi. Ridadel nr 53 ja 54 on aga juba

mitte Cppchecki parameetrid, vaid toimub Cppchecki protsessi väljundite (vastavalt standardse väljundvoo ja standardse veateadete voo) ümbersuunamine konsooli asemel failidesse. Cppchecki standardne väljundvoog kannab üldisi teateid analüüsi kulgemise kohta (logimiseks suunatakse see faili `cppcheck_run.log`), seevastu veateadete voo kaudu edastabki Cppcheck analüüsi aruande XML-vormingus. Ümbersuunamise tulemusel salvestub see aruanne `cppcheck_report.xml` faili.

Kui Cppcheck suudaks laitmatult parsida ka keerukaid Visual Studio projekte ja lahendusfaile, siis saaks skriptis kasutada mõnevõrra lihtsamat käsurida, nimelt platvormi (`--platform`), makrodefinitsioonide (`-D`), välistatud kaustade (`-i`), otsinguteekondade loetelufaili (`--includes-file`) ja analüüsitavate failide/kaustade loetelu (`.`) asemel (ehk ridade nr 36–46 ja 52 asemel) saaks anda ette üksnes kõikide lahendusfailide nimed (järjest mitme `--project` suvandi abil).

Edasi toimub koodi kaetuse analüüsi ettevalmistamine. Üksikute koodi kaetuse aruannete kokkukoondamiseks saab vastavate aruannete failinimesid OpenCppCoverage'ile ette anda kas `--input_coverage` parameetri abil käsurealt või `input_coverage` (ehk lihsalt ilma kriipsudeta) parameetri kaudu konfiguratsioonifailis [13] (ehk mestimisloendis, kui see fail koosnebki üksnes `input_coverage` nimekirjast). Protsessi automatiseerimiseks kogub põhiskript kokku üksikute aruannete mestimise loendi ja söödab selle loendi (käsureal parameetri `--config_file` abil) OpenCppCoverage'ile ette.

Niisiis, et leida üles kõik üksuste teste sisaldavad laademoodulid, otsitakse ridadel nr 59 kuni 79 paiknevas tsüklis üles kõik sõnega `_test.exe` lõppevad failinimed. Samas tsüklis need failid nii käivitatakse OpenCppCoverage'i vahendusel (kogu käsk ridadel nr 63–77, käivitatava laademooduli nimi real nr 74) kui ka lisatakse mestimisloendisse (rida nr 61) vastavad failinimed (kuid EXE asemel COV-faililaiendiga, kuna kaetuse aruanded tekivad just sellise laiendiga, nagu määratud real nr 66). Real nr 75 etteantav parameeter `--gtest_output` kuulub juba laademooduli (mitte OpenCppCoverage'i) parameetrite hulka ja tekitab käivitatud testide kohta XML vormingus aruande (see Google Testi võimalus on mainitud jaotises 2.5). Ridadel nr 76 ja 77 toimub OpenCppCoverage'i väljundi ja veateadete ümbersuunamine faili `coverage_run.log` (logimiseks).

Ridadel nr 83–91 toimub üksikute koodi kaetuse aruannete mestimine ühte tervikusse (real nr 88 määratud `coverage_report_original.xml` faili). Nagu juba mainitud, antakse

OpenCppCoverage'ile käsureal ette mestimisloendi failinimi parameetri `--config_file` abil (vt rida nr 85). Ka siin (ridadel nr 90 ja 91) toimub OpenCppCoverage'i väljundi ja veateadete lisamine samasse logifaili (`coverage_run.log`).

Seejärel toimub tervikus kaetuse aruandes sisalduvate failiteekondade teisendamine SonarQube Scannerile sobivale kujule, nagu on kirjeldatud alajaotises 3.1.2, spetsiaalse PowerShelli skripti (`tweak-cobertura-xml.ps1`) abil: ridadel nr 97–106 paikneva käsuga käivitatakse PowerShell ning antakse (parameetri `-Command "& …"` abil) ette käivitamist vajava skriptifaili teekond koos skripti parameetritega (kõik ühes sõnes). Kuna tegemist on loetavuse huvides üle mitme eraldi rea laotud käsuga, ja kuna see sisaldab jutumärke, kasutatakse real nr 100 üht trikki, mis võimaldab murda Windowsi pakkfailides jutumärke sisaldavaid ridu: parsimise järel `!"=!` osa kaob[1] ja katusemärk saab korrektselt toimida. Read 96 (`SETLOCAL ENABLEDELAYEDEXPANSION`) ja 107 (`ENDLOCAL`) on vajalikud üksnes triki toimimiseks.

PowerShelli skripti sisendiks on fail `coverage_report_original.xml`, väljundiks aga korrigeeritud aruanne faili `coverage_report.xml` näol. Samuti logib PowerShelli skript kõik tehtud muudatused oma logifaili (`tweak-cobertura-xml.log`).

Viimaks käivitatakse SonarQube Scanner (read nr 118–122), mis laadib valminud aruanded (failid `coverage_report.xml` ja `cppcheck_report.xml`) SonarQube serverisse üles. Käsureaparameetrina antakse ette Scanneri konfiguratsioonifaili asukoht (vt rida nr 120). Kõiki vea- ja muid teateid logitakse (`scanner_run.log`). Ridadel nr 114–117 määratakse keskkonnamuutuja SonarQube Scanneri suvanditega, mida kasutatakse serveriga krüptitud turvalise ühenduse loomiseks (sest nagu alajaotises 0 näidatud, on `sonar.host.url` parameetri väärtuseks HTTPS-protokolli kasutav aadress).

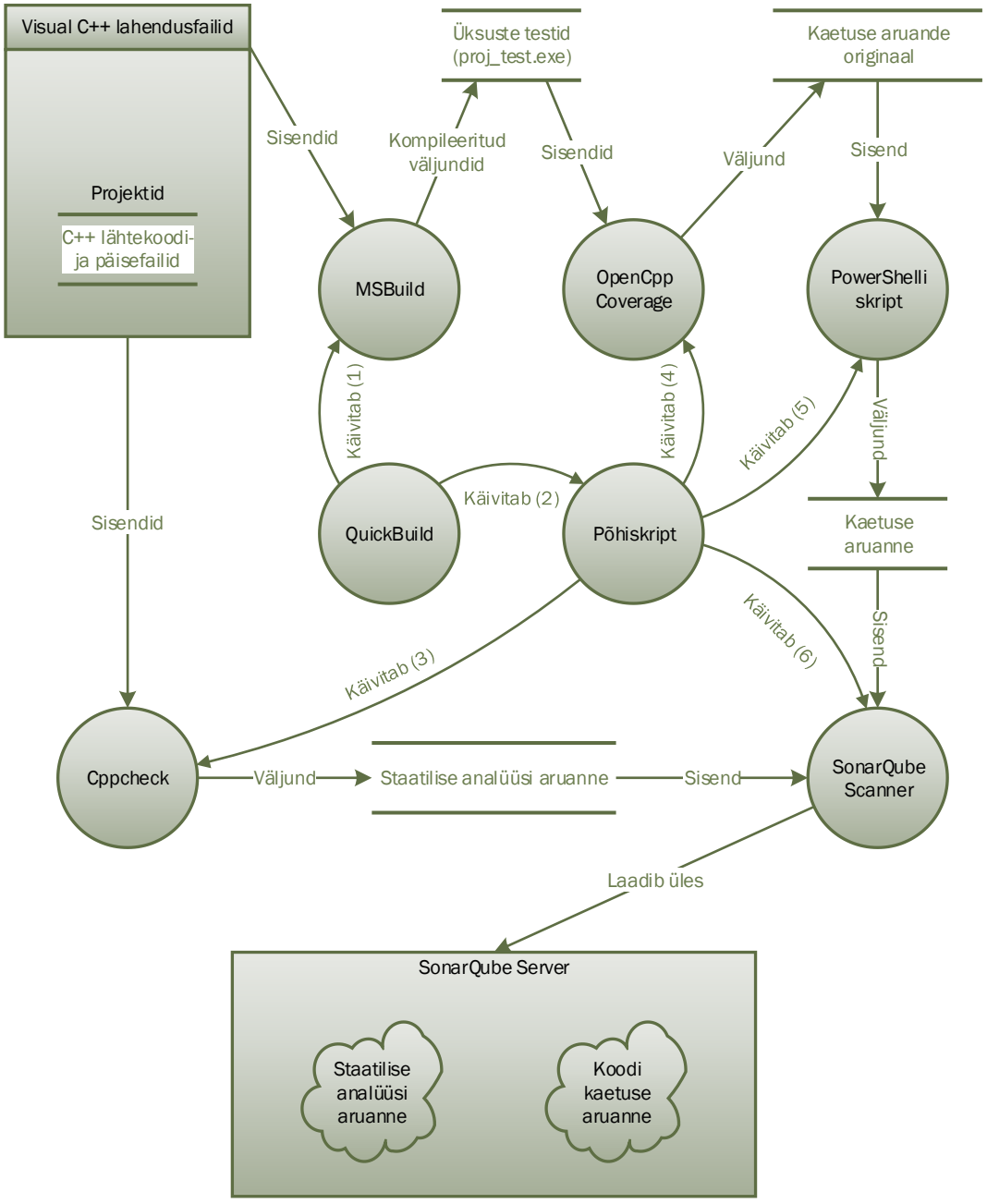
Kuna SonarQube Scanner käivitatakse samuti pakkfaili kaudu, nagu põhiskript isegi, on kõrvaltoimete vältimiseks võetud kasutusele read nr 113 ja 123 (`SETLOCAL` / `ENDLOCAL`), millega on kindlustatud ka Scanneri-skripti sisemiste keskkonnamuutujate lokaalsus.

Kogu põhiskripti tööd logitakse läbivalt faili `scan-all.log` koos tegevuste ajatemplitega.

---

[1] Stack Overflow kasutaja „jeb“ vastus küsimusele „How to split double quoted line into multiple lines in Windows batch file?“ (kolmas meetod), https://stackoverflow.com/a/4645113 (ingl, 12.04.2012)

Üldisema konteksti juurde tagasi tulles võib mainida, et kui joonisel 2 oli näidatud analüüsiprotsessi automatiseerimise esialgne kava, siis joonisel 4 on näidatud tegeliku lahenduse üldine skeem nii, nagu see katsetamiste käigus ilmnenud uute asjaolude ja vajaduste valguses lõpuks teostatud sai.



Joonis 4. Kogu lahenduse skeem.

Kui kätte jõuab järjekordse perioodilise kvaliteedianalüüsi aeg, siis kõigepealt käivitab QuickBuild (mis on konfigureeritud nii, nagu jaotises 3.4 kirjeldatud) MSBuildi protsessi, mille sisenditeks on Visual C++ lahendusfailid, selleks et kompileerida kõik üksuste testid (kasutades `Test Debug` konfiguratsiooni). Kui kõik üksuste testid on kompileeritud ja

vastavad laademoodulid valmis, käivitab QuickBuild lisas 1 toodud peamise skripti ja ootab ära skripti eduka lõpu. Sellega QuickBuildi roll piirdub.

Nüüd käivitab peamine skript Cppchecki, mille sisendiks on needsamad Visual C++ lahendusfailid (või, kui selliselt esineb ikka veel failide leidmisega seotud tõrkeid, siis on sisendiks hoopis analüüsitavate lähtekoodifailide kaust ja päisefailide otsinguteekondade loetelu). Cppchecki väljundiks on staatilise analüüsi aruanne XML-vormingus.

Kui Cppchecki protsess jõuab oma analüüsiga valmis, käivitab peamine skript iga üksuste testi laademooduli kallal eraldi OpenCppCoverage'i, siis käivitab OpenCppCoverage'i veel viimast korda, et koondada üksikud kaetuse aruanded üheks tervikuks: see on koodikaetuse aruande originaal.

Seejärel käivitab peamine skript PowerShelli skripti, mis teeb algupärasest koodikaetuse aruandest veidi teistsuguse, korrigeeritud aruande: selles on failiteekonnad määratud nii, et SonarQube Scanner need leiaks.

Nüüd, kui mõlemad aruanded (nii staatilise analüüsi kui ka koodi kaetuse oma) on valmis, käivitab peamine skript SonarQube Scanneri, mis on konfigureeritud nende aruannete leidmiseks, SonarQube C++ Community plugina vahendusel parsimiseks ning lõpuks ka üleslaadimiseks SonarQube serverisse.

Kui SonarQube Scanner teeb oma töö ära, lõpeb ka peamise skripti töö, ning mõne aja möödudes saab SonarQube serveris koodi kvaliteedi analüüsi tulemusi sirvida ja eelmiste perioodide tulemustega võrrelda.

4 Hinnang lahendusele

Käesolevas töös kirjeldatud tegevused lähtekoodi analüüsi vahendite liidestamiseks SonarQube'iga võeti ette kõigepealt selleks, et saada aimu ettevõtte C++ keeles kirjutatud tarkvara kvaliteedist, konkreetsemalt just staatilistest analüüsimeetoditest lähtudes. Kogu selle lõimimise tulemusena õnnestuski saada ülevaade nii koodi kaetuse ulatusest üksuste testide poolt kui ka lähtekoodis esinevatest vigadest ja muudest probleemidest. Ettevõtte SonarQube serveris tekkisid nüüd mitmesugused piltlikud ja mugavalt käsitletavad kvaliteediaruanded ka C++ lähtekoodi kohta, sarnaselt juba varemgi Java lähtekoodi kohta avaldatutele.

Lähtekoodi aruanne näitas C++ tarkvaratoote osas kokku paar tuhat väidetavat probleemi. Leitud loetelust parandati sadakond viga. Muuhulgas said parandatud mitmed sellised defektid, nagu:

- muutujale uue väärtuse omistamine enne vana väärtuse kasutamist (põhiliselt olid nendeks ülekirjutatud väärtusteks tulemikoodid);
- algväärtustamata liikmesmuutuja;
- jõudluse vähenemine (näiteks juhtudel, kus optimaalse jõudluse saavutamiseks tuleks mõni argument ette anda konstantse viite kaudu, või kui mõni meetod tuleks muuta konstantseks või staatiliseks);
- kasutamata muutuja;
- muutuja liiga ulatuslik kehtivuspiirkond (ja see tuleks tegeliku kasutuspiirkonnani ahendada).

Kuna üldiseks eesmärgiks on mõistagi koodi kvaliteedi parandamine, siis võib öelda, et lahendus täitis oma eesmärgi, sest tänu sellele avastati ja parandati samas loetelus välja toodud lisaks ülalmainitud vähemolulistele probleemidele ka kümmekond tõsist viga, nende hulgas:

- mälulekked puhvritest;
- ressursileke.

Tuhatkonna vähemolulise avaldatud probleemiga jäi tarkvaraarendajatel ajapuudusel tegelemata (ignoreeritud soovitused olid valdavalt seotud muutujate kehtivuspiirkonna ahendamisega ja meetodite konstantseks määramisega).

Puudusena võib seevastu mainida, et lähtekoodi aruandes esines tuhatkond valepositiivset probleemileidu. Valdavalt olid need teated väidetavalt kasutamata jäetud funktsioonidest. Kontrollimisel osutusid need teated tõele mittevastavaks müraks. Sellise müra peamiseks põhjuseks näib asjaolu, et kasutatud vahendid endiselt ei suuda leida päris kõiki vajalikke lähtekoodi päisefaile.

Koodikaetuse üldine aruanne näitas, et kogu C++ lähtekood oli üksuste testite poolt kaetud 12% ulatuses. Samuti oli SonarQube'is võimalik tuvastada näiteks kõige vähem testide poolt kaetud piirkonnad lähtekoodis.

Kuna koodianalüüsi tegum on QuickBuildis ajastatud käivituma kord nädalas (igal pühapäeval), tekib SonarQube'is mugav iganädalane ülevaade ning kvaliteedinäitajate muutusi on võimalik võrrelda eelmiste seisudega.

Tulevikus võiks kaaluda korraga mitme analüüsivahendi kasutamist, näiteks lisaks käesolevas töös käsitletutele võiks kasutada Clangi. See aitaks tõsta nii koodi kvaliteeti kui ka standardile vastavust.

5 Kokkuvõte

Töö peamiseks eesmärgiks oli C++ koodi kvaliteedi tõstmine autori töökohal, kus arenduseks kasutatakse Microsoft Visual Studiot. Selle saavutamiseks tuli kõigepealt tagada regulaarne ülevaade koodi kvaliteedi peamistest näitajatest SonarQube serveri vahendusel.

Kuna SonarSource'i poolt pakutavat ametlikku C++ analüsaatorit otsustati kõrge hinna tõttu mitte soetada, hakkas analüüsivahendite edasist valikut dikteerima selle analüsaatori tasuta ja vaba alternatiiv: SonarQube C++ Community plugin.

Esialgse kava kohaselt pidi QuickBuild otse käivitama nii analüüsivahendeid kui ka SonarQube Scannerit, mis omakorda laadiks tekkivaid analüüsiaruandeid üles SonarQube serverisse.

Ettevõtte huvidest lähtuvalt otsustati piirduda avatud lähtekoodiga vabade tarkvaraliste analüüsivahenditega. Valituks osutusid staatilise analüüsi utiliit Cppcheck ja koodi kaetuse analüüsi utiliit OpenCppCoverage. Nende utiliitide käimapanekul ja sobitamisel SonarQube Scanneri vajadustega ilmnesid aga mõned ootamatud komistuskivid.

OpenCppCoverage väljastab küll kaetuse aruande, kuid ühe laademooduli käivitamise tulemusel ei kajasta selline üksik aruanne kogu koodibaasi kaetust. Seetõttu tuli kõigepealt järjest käivitada OpenCppCoverage'is kõik laademoodulid ja seejärel tekkinud aruanded kokku koondada. Samuti selgus, et OpenCppCoverage'i loodav Cobertura vormingus aruanne oma algsel kujul ei sobi SonarQube Scannerile ettesöötmiseks, kuna failiteekonnad peavad olema antud ette konkreetse aluskausta, antud juhul repositooriumi tüve suhtes, mitte aga draivi juure suhtes. Aruande korrigeerimiseks sai kirjutatud spetsiaalne PowerShelli skript.

Cppcheck esialgu ei suutnud Visual Studio täiemahulisi ja keerukaid lahendus- ja projektifaile korrektselt parsida. Käesoleva töö autori loodud testjuhtude abil parandas Cppchecki peamine arendaja mõned vead. Ka töö autor ise lõi Cppchecki ametlikus

repositooriumis GitHubis tõmbetaotluse paari vea parandamiseks; see tõmbetaotlus rahuldati ja liideti Cppchecki versiooni 1.77 koosseisu.

Erinevalt esialgselt kavandatud lahendusest, milles kõikide analüüsiga seonduvate protsesside haldajaks oleks QuickBuild, kirjutas töö autor enamuse nende protsesside juhtimiseks Windowsi pakkfaili. Tõsi, QuickBuild jäi selle pakkfaili ja MSBuildi käimatõmbajaks.

Töö tulemile kriitilise hinnangu andmisel autor tõdeb, et arenguruumi kahtlemata veel on. Kuid kuna teostatud lahenduse abil avastati ja parandati ettevõtte C++ lähtekoodis mitmeid vigu, sealhulgas kümmekond tõsist, siis võib peamist eesmärki pidada täidetuks.

Kasutatud kirjandus

[1] E. Penttilä, "Improving C++ Software Quality with Static Code Analysis," *Master's thesis, Aalto University, Finland,* 16 May 2014.

[2] SonarSource S.A, Switzerland, "SonarQube Scanner Analysis Parameters," 2019. [Online]. Available: https://docs.sonarqube.org/latest/analysis/analysis-parameters/. [Accessed 14 May 2019].

[3] Microsoft Docs, "Share or reuse Visual Studio project settings," 27 March 2019. [Online]. Available: https://docs.microsoft.com/en-us/cpp/build/create-reusable-property-configurations. [Accessed 21 May 2019].

[4] Google, "Advanced Google Test Topics," GitHub, 30 March 2019. [Online]. Available: https://github.com/google/googletest/blob/master/googletest/docs/advanced.md. [Accessed 11 May 2019].

[5] Google, "Google Mock for Dummies," GitHub, 4 September 2018. [Online]. Available: https://github.com/google/googletest/blob/master/googlemock/docs/ForDummies.md. [Accessed 11 May 2019].

[6] T. Tali, „Programmi koosteprotsessi automatiseerimine QuickBuildi abil," *Bakalaureusetöö, Tallinna Tehnikaülikool,* 15 June 2015.

[7] SonarQube C++ Community Plugin, "Supported configuration properties," GitHub, 8 February 2019. [Online]. Available: https://github.com/SonarOpenCommunity/sonar-cxx/wiki/Supported-configuration-properties. [Accessed 14 May 2019].

[8] F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles and B. Yakobowski, "Frama-C: A software analysis perspective," *Formal Aspects of Computing,* vol. 27, no. 3, p. 573–609, May 2015.

[9] Free Software Foundation, Inc., "Using the GNU Compiler Collection (GCC): 10.1 Introduction to gcov," 2019. [Online]. Available: https://gcc.gnu.org/onlinedocs/gcc/Gcov-Intro.html. [Accessed 21 May 2019].

[10] Cppcheck, "Cppcheck manual," 9 February 2019. [Online]. Available: http://cppcheck.sourceforge.net/manual.pdf. [Accessed 19 May 2019].

[11] G. Chatzieleftheriou and P. Katsaros, "Test-driving static analysis tools in search of C code vulnerabilities," in *2011 IEEE 35th Annual Computer Software and Applications Conference Workshops*, 2011.

[12] J. Moerman, "Evaluating the performance of open source static analysis tools," *Bachelor thesis, Radboud University, The Netherlands,* 24 June 2018.

[13] OpenCppCoverage, "Command line reference," GitHub, 15 February 2018. [Online]. Available: https://github.com/OpenCppCoverage/OpenCppCoverage/wiki/Command-line-reference. [Accessed 17 May 2019].

[14] L. Holmes, "Working with the .NET Framework," in *Windows PowerShell Quick Reference*, O'Reilly Media, Inc, 2006, p. 38.

[15] Microsoft Docs, "Select-Xml," in Powershell Reference, Microsoft.PowerShell.Utility module, 27 August 2018. [Online]. Available: https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.utility/select-xml. [Accessed 19 May 2019].

[16] Microsoft Docs, "Common macros for MSBuild commands and properties," 20 March 2019. [Online]. Available: https://docs.microsoft.com/en-us/cpp/build/reference/common-macros-for-build-commands-and-properties. [Accessed 12 April 2019].

[17] Microsoft Docs, "Set compiler and build properties," 27 March 2019. [Online]. Available: https://docs.microsoft.com/en-us/cpp/build/working-with-project-properties. [Accessed 12 April 2019].

[18] Microsoft Docs, "MSBuild command-line reference," 4 November 2016. [Online]. Available: https://docs.microsoft.com/en-us/visualstudio/msbuild/msbuild-command-line-reference. [Accessed 19 May 2019].

[19] SonarQube C++ Community Plugin, "Compilers | Retrieving compiler information from buildlog," GitHub, 11 March 2019. [Online]. Available: https://github.com/SonarOpenCommunity/sonar-cxx/wiki/Compilers. [Accessed 19 May 2019].

[20] Microsoft Docs, "Windows Commands," 22 May 2018. [Online]. Available: https://docs.microsoft.com/en-us/windows-server/administration/windows-commands/windows-commands. [Accessed 16 May 2019].

Lisa 1 – Peamine skript analüüsi automatiseerimiseks pakkfailiga

```
REM This batch script runs necessary tools to perform static code and coverage analysis of C++
REM sources in all solutions, and uploads results to the SonarQube server. Prerequisite: PATH
REM environment variable includes the PowerShell location, and JAVA_HOME is properly set.
SETLOCAL

REM Setting up local environment constants for convenience
SET BINDIR=build\sonar\bin
SET CONFDIR=build\sonar\conf
SET LOGDIR=build\sonar\log
SET LOGFILE="%LOGDIR%\scan-all.log"

REM Changing the current directory to the root folder of the frontend
CD /D "%~dp0..\..\.."
ECHO %DATE% %TIME% Changed current directory to "%CD%" > %LOGFILE%

REM Updating the PATH with Cppcheck, OpenCppCoverage, Sonar Scanner folders in our Build Machine
SET PATH=%CD%\.tools\Cppcheck;%CD%\.tools\OpenCppCoverage;%CD%\.tools\sonar-scanner\bin;%PATH%
ECHO %DATE% %TIME% Updated local PATH=%PATH% > %LOGFILE%

REM Cleanup, deleting old logs and reports
ECHO %DATE% %TIME% Cleanup, deleting old logs and reports >> %LOGFILE%
RMDIR /S /Q "%LOGDIR%\coverage_report_html"
RMDIR /S /Q "%LOGDIR%\coverage_reports"
RMDIR /S /Q "%LOGDIR%\gtest_reports"
DEL "%LOGDIR%\coverage_*"
DEL "%LOGDIR%\cppcheck_*"
DEL "%LOGDIR%\scanner_run.log"
DEL "%LOGDIR%\tweak-cobertura-xml.log"

REM Invoking Cppcheck
ECHO %DATE% %TIME% Invoking Cppcheck >> %LOGFILE%
cppcheck ^
 --xml-version=2 ^
 --library=microsoft_sal.cfg ^
 --library=windows.cfg ^
 --platform=win32W ^
 -D_WIN32=1 ^
 -DWIN32=1 ^
 -D_UNICODE ^
 -DUNICODE ^
 -DNDEBUG=1 ^
 -D_MSC_VER=1900 ^
 -D__cplusplus=201703 ^
 -i "externals" ^
 -i "customers" ^
 --includes-file="%CONFDIR%\cppcheck_includes.txt" ^
 --suppressions-list="%CONFDIR%\cppcheck_suppressions.txt" ^
 --inline-suppr ^
 --verbose ^
 --enable=all ^
 --inconclusive ^
 . ^
 1> "%LOGDIR%\cppcheck_run.log" ^
 2> "%LOGDIR%\cppcheck_report.xml"
ECHO %DATE% %TIME% Cppcheck done, result %ERRORLEVEL% >> %LOGFILE%

REM Executing unit tests and invoking OpenCppCoverage
ECHO %DATE% %TIME% Searching for unit test executables >> %LOGFILE%
```

```
FOR %%C IN ("bin\debug\*_test.exe") DO (
REM Updating the merge list for coverage aggregation
ECHO input_coverage=%LOGDIR%\coverage_reports\%%~nC.cov>> "%LOGDIR%\coverage_merge_list.txt"
ECHO %DATE% %TIME% Invoking OpenCppCoverage with the unit test "%%~fC" >> %LOGFILE%
OpenCppCoverage ^
 --verbose ^
 --continue_after_cpp_exception ^
 --export_type=binary:"%LOGDIR%\coverage_reports\%%~nC.cov" ^
 --sources="%CD%" ^
 --excluded_sources="externals" ^
 --excluded_sources="customers" ^
 --modules="%CD%" ^
 --excluded_modules="externals" ^
 --excluded_modules="customers" ^
 -- ^
 "%%~fC" ^
 --gtest_output=xml:"%LOGDIR%\gtest_reports/" ^
 1>> "%LOGDIR%\coverage_run.log" ^
 2>>&1
ECHO %DATE% %TIME% OpenCppCoverage for "%%~nC" done, result %ERRORLEVEL% >> %LOGFILE%
)

REM Aggregating coverage results
ECHO %DATE% %TIME% Aggregating coverage results >> %LOGFILE%
OpenCppCoverage ^
 --verbose ^
 --config_file="%LOGDIR%\coverage_merge_list.txt" ^
 --export_type=binary:"%LOGDIR%\coverage_report.cov" ^
 --export_type=html:"%LOGDIR%\coverage_report_html" ^
 --export_type=cobertura:"%LOGDIR%\coverage_report_original.xml" ^
 -- ^
 1>> "%LOGDIR%\coverage_run.log" ^
 2>>&1
ECHO %DATE% %TIME% Coverage results aggregated, OpenCppCoverage result %ERRORLEVEL% >> %LOGFILE%

REM Changing paths in the coverage XML report so that SonarQube Scanner will locate source files
ECHO %DATE% %TIME% Providing proper relative paths in the coverage XML report >> %LOGFILE%
SETLOCAL ENABLEDELAYEDEXPANSION
powershell ^
 -NoProfile ^
 -ExecutionPolicy Bypass ^
 -Command "& !"=! ^
 '%BINDIR%\tweak-cobertura-xml.ps1' ^
 -BaseDir '%CD%' ^
 -InputFile '%LOGDIR%\coverage_report_original.xml' ^
 -OutputFile '%LOGDIR%\coverage_report.xml' ^
 -LogFile '%LOGDIR%\tweak-cobertura-xml.log'; ^
 exit $LASTEXITCODE"
ENDLOCAL
ECHO %DATE% %TIME% Paths in the coverage report updated, script result %ERRORLEVEL% >> %LOGFILE%

REM Invoking SonarQube Scanner
ECHO %DATE% %TIME% SonarQube Scanner will use JAVA_HOME=%JAVA_HOME% >> %LOGFILE%
ECHO %DATE% %TIME% Invoking SonarQube Scanner >> %LOGFILE%
SETLOCAL
SET SONAR_SCANNER_OPTS=-Djavax.net.ssl.trustStore="%CONFDIR%\truststore.p12" ^
 -Djavax.net.ssl.trustStoreType=PKCS12 ^
 -Djavax.net.ssl.trustStorePassword=changeit ^
 -Dcom.sun.security.enableAIAcaIssuers=true
CALL sonar-scanner ^
 -X ^
 -Dproject.settings="%CONFDIR%\sonar-project.properties" ^
 1> "%LOGDIR%\scanner_run.log" ^
 2>&1
ENDLOCAL
ECHO %DATE% %TIME% SonarQube Scanner done, result %ERRORLEVEL% >> %LOGFILE%
ECHO %DATE% %TIME% All done >> %LOGFILE%
ENDLOCAL
```

Lisa 2 – PowerShelli skript Cobertura vormingu teisendamiseks

```
# This PowerShell script parses OpenCppCoverage code coverage report in a Cobertura XML file and
# transforms path attributes found there into relative paths so that SonarQube Scanner is able to
# locate C++ source files.

param
(
    [Parameter(Mandatory=$true)][AllowEmptyString()][string]$BaseDir,
    [Parameter(Mandatory=$true)][string]$InputFile,
    [Parameter(Mandatory=$true)][string]$OutputFile,
    [Parameter(Mandatory=$true)][string]$LogFile
)

function Log
{
    param ([string]$text)
    Add-Content -ErrorAction Continue -Path $LogFile -PassThru -Value $(
        (Get-Date).ToString('yyyy-MM-dd HH\:mm\:ss.fff') + ' ' + $text)
}

function IsFullPath
{
    param ([string]$path)
    $path.Length -ge 2 -and $path.Substring(1, 1) -eq ':'
}

function Rebase
{
    param ([string]$Path, [string]$OldBase, [string]$NewBase)

    $result = $Path

    if (!(IsFullPath $result))
    {
        if (!$OldBase)
        {
            $null = Log 'Error: no original base path for sources, '
            $null = Log "and not a fully-qualified filename: `"$Path`""
            $null = Log 'Exiting!'
            exit 51
        }
        $result = $OldBase + $result.TrimStart('\/')
    }

    if ($NewBase -and $result.StartsWith($NewBase, [System.StringComparison]::OrdinalIgnoreCase))
    {
        $result = $result.Remove(0, $NewBase.Length)
        ++$Script:numRelativePaths
        $null = Log "Successfully transformed filename: `"$Path`" => `"$result`""
    }
    else
    {
        ++$Script:numAbsolutePaths
        $null = Log "Cannot rebase filename: `"$Path`" => `"$result`""
    }

    $result
}

```

```
$ErrorActionPreference = 'Stop'

$numAbsolutePaths = 0
$numRelativePaths = 0

try
{
    Log "Started with log file `"$LogFile`", input `"$InputFile`", output `"$OutputFile`""
    Log "New base path for sources will be `"$BaseDir`""

    $normOriginalBaseDir = ''
    $normNewBaseDir = $BaseDir

    if ($normNewBaseDir)
    {
        $normNewBaseDir = $normNewBaseDir.TrimEnd('\/') + '\'
        if (!(IsFullPath $normNewBaseDir))
        {
            Log 'Unsupported: the desired new base path for sources is not fully-qualified'
            Log 'Exiting!'
            exit 11
        }
    }

    Log "Loading the original XML from file `"$InputFile`""
    $CoberturaXML = New-Object -TypeName XML
    $CoberturaXML.Load($InputFile)

    $SourceElementsList = $CoberturaXML.SelectNodes('/coverage/sources/source')
    if ($SourceElementsList.Count)
    {
        foreach ($SourceElement in $SourceElementsList)
        {
            if (!$normOriginalBaseDir.Length)
            {
                $normOriginalBaseDir = $SourceElement.InnerText
                Log "The input XML has a <source> element containing `"$normOriginalBaseDir`""
                $normOriginalBaseDir = $normOriginalBaseDir.TrimEnd('\/') + '\'
                if (!(IsFullPath $normOriginalBaseDir))
                {
                    Log 'Unsupported: the input sets not a fully-qualified base path for sources'
                    Log 'Exiting!'
                    exit 12
                }
                if ($normNewBaseDir)
                {
                    $SourceElement.InnerText = $normNewBaseDir
                }
            }
            elseif ($SourceElement.InnerText.TrimEnd('\/') + '\' -ieq $normOriginalBaseDir)
            {
                Log 'The input has a duplicate <source> element; discarding it from the output'
                $SourceElement.ParentNode.RemoveChild($SourceElement) | Out-Null
            }
            else
            {
                Log 'Unsupported: the input XML provides multiple different <source> elements'
                Log 'Exiting!'
                exit 13
            }
        }
    }
    else
    {
        Log 'Warning: no <source> element in the input; assuming there are no relative paths!'
        if ($normNewBaseDir)
        {
            $SourcesElement = $CoberturaXML.SelectSingleNode('/coverage/sources')
```

```
127             if ($SourcesElement -eq $null)
128             {
129                 Log 'Creating a <sources> element for the output XML'
130                 $SourcesElement = $CoberturaXML.SelectSingleNode('/coverage').AppendChild(
131                     $CoberturaXML.CreateElement('sources'))
132             }
133             Log 'Creating a <source> element for the output XML'
134             $SourcesElement.AppendChild($CoberturaXML.CreateElement('source')).AppendChild(
135                 $CoberturaXML.CreateTextNode($normNewBaseDir))
136         }
137     }
138
139     if (!$normNewBaseDir)
140     {
141         Log 'The desired new base path for sources is empty, '
142         Log 'thus the resulting XML will contain fully-qualified paths only'
143         foreach ($item in (Select-Xml -Xml $CoberturaXML -XPath '/coverage/sources'))
144         {
145             $item.Node.ParentNode.RemoveChild($item.Node) | Out-Null
146         }
147     }
148
149     Log 'Changing the filename attributes in the XML'
150     foreach ($FileNameAttribute in (Select-Xml -Xml $CoberturaXML -XPath '/coverage//@filename'))
151     {
152         $FileNameAttribute.Node.Value =
153 Rebase -Path $FileNameAttribute.Node.Value -OldBase $normOriginalBaseDir -NewBase $normNewBaseDir
154     }
155
156     Log "Saving the updated XML into file `"$OutputFile`""
157     $OutputWriterSettings = New-Object System.Xml.XmlWriterSettings
158     $OutputWriterSettings.Encoding = New-Object System.Text.UTF8Encoding($false)
159     $OutputWriterSettings.Indent = $true
160     $OutputWriter = [System.Xml.XmlWriter]::Create($OutputFile, $OutputWriterSettings)
161     $CoberturaXML.Save($OutputWriter)
162     $OutputWriter.WriteWhitespace($OutputWriterSettings.NewLineChars)
163     $OutputWriter.Close()
164
165     Log "XML transformation completed"
166     Log "Yielded $numAbsolutePaths absolute, $numRelativePaths relative paths"
167     Log 'All done'
168     exit 0
169 }
170 catch
171 {
172     Log $('Encountered an error at line #' + $_.InvocationInfo.ScriptLineNumber + ', column #' +
173         $_.InvocationInfo.OffsetInLine)
174     Log $_
175     exit 101
176 }
```