ZÁPADOČESKÁ UNIVERZITA V PLZNI
FAKULTA ELEKTROTECHNICKÁ

Katedra aplikované elektroniky a telekomunikací

DIPLOMOVÁ PRÁCE

Statická analýza kódu v souladu s požadavky normy EN 50128

Václav Průcha **2017/2018**

ZÁPADOČESKÁ UNIVERZITA V PLZNI
Fakulta elektrotechnická
Akademický rok: **2017/2018**

ZADÁNÍ DIPLOMOVÉ PRÁCE

(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení:	**Bc. Václav PRŮCHA**
Osobní číslo:	**E16N0079P**
Studijní program:	**N2612 Elektrotechnika a informatika**
Studijní obor:	**Dopravní elektroinženýrství a autoelektronika**
Název tématu:	**Statická analýza kódu v souladu s požadavky normy EN50128**
Zadávající katedra:	**Katedra aplikované elektroniky a telekomunikací**

Zásady pro vypracování:

1. Uveďte přehled metodik statické analýzy kódu a navrhněte metodiku vhodnou pro nasazení v prostředí společnosti ŠKODA - oddělení vývoje řízení kolejových vozidel.
2. Implementuje navržené řešení přímo v produkčním prostředí na jednom projektu a to včetně provázání s již existujícími nástroji (Mercurial, IssueTracker).
3. Připravte systém reportingu, jak pro účely schvalování SW, tak pro účely sledování postupu prací na projektu.
4. Zhodnoťte navržené zařízení (případě uveďte možnosti dalšího vylepšení či rozšíření).

Rozsah grafických prací:	**podle doporučení vedoucího**
Rozsah kvalifikační práce:	**40 - 60 stran**
Forma zpracování diplomové práce:	**tištěná/elektronická**
Seznam odborné literatury:	

Norma EN50128

Vedoucí diplomové práce:	**Ing. Petr Weissar, Ph.D.** Katedra aplikované elektroniky a telekomunikací
Datum zadání diplomové práce:	**10. října 2017**
Termín odevzdání diplomové práce:	**24. května 2018**

Doc. Ing. Jiří Hammerbauer, Ph.D.
děkan

L.S.

Doc. Dr. Ing. Vjačeslav Georgiev
vedoucí katedry

V Plzni dne 10. října 2017

Abstrakt

Předkládaná diplomová práce se věnuje problematice zajištění kvality SW u bezpečnostně relevantních zařízení – konkrétně SW řídicího počítače lokomotivy. Zvláštní důraz je kladen na metodiku statické analýzy kódu v souladu s požadavky normy EN 50128. Práce analyzuje vybrané metody statické analýzy kódu, jejich začlenění do realizačního procesu a využitelné SW nástroje (SonarQube, Cppcheck a vývojové prostředí Microsoft Visual Studio 2017). Cílem práce je vytvořit užitečný SW nástroj fungující v konkrétním podnikovém prostředí (Škoda Transportation a.s.).

V úvodní – teoretické části se seznámíme s obecným přehledem problematiky zajištění jakosti SW a poté jsou uvedeny specifické informace sloužící k porozumění implementační sekce této práce, která popisuje reálný proces statické analýzy kódu a kontrolu implementace SW dle pravidel kódovacího standardu.

Práce se snaží dát do souvislosti množství technických termínů s cílem vysvětlit složitý proces zajišťování kvality softwaru.

Klíčová slova

Statická, analýza, kód, SW, SCC, SonarQube, Cppcheck, norma, Škoda

Abstract

This thesis investigates how and where static code analysis can be integrated into a development process of safety relevant SW – namely SW of Train Control and Monitoring System of a locomotive. Since development of SW for traction vehicles is subject of assessment according to the norm EN 50128 this thesis focuses on adjustment and modification of existing generic SW tools (SonarQube, Cppcheck and Microsoft Visual Studio 2017).

First, a theoretical study of SW quality assurance is presented. Secondly, specific data and implementation process of static code analysis is shown in real industrial environment.

The aim of this thesis is implementation of static code analysis tools and processes within real SW project. The resulting product should extend tools and processes utilized in SKODA TRANSPORTATION for SW quality assurance.

Key words

Static, analysis, code, SW, SCC, SonarQube, Cppcheck, norm, Skoda

Prohlášení

Prohlašuji, že jsem tuto diplomovou/bakalářskou práci vypracoval samostatně, s použitím odborné literatury a pramenů uvedených v seznamu, který je součástí této diplomové práce.

Dále prohlašuji, že veškerý software, použitý při řešení této bakalářské/diplomové práce, je legální.

.....

podpis

V Plzni dne 2. 5. 2018 Václav Průcha

Poděkování

Tímto bych rád poděkoval vedoucímu diplomové práce Ing. Petrovi Weissarovi Ph.D., Ing. Ondřeji Borusíkovi Ph.D a Ing. Jiřímu Langhammerovi za cenné profesionální rady, připomínky a metodické vedení práce.

Obsah

Seznam obrázků

Seznam tabulek

Seznam symbolů a zkratek

ACG	Application Controll Graphic – Součást TDD. Neobsahuje grafický výstup, ale zajišťuje konfiguraci, sdílenou paměť, rozhraní pro negenerickou část a vyhodnocení detekce stisknutí obrazovky.
ADG	Application Displaying Graphics – Součást TDD. Zobrazovací část, zpracovává konfigurační soubory, sdílenou paměť, grafické objekty a grafické rozhraní.
ATO	Automatic Train Operation – Jednotka realizující centrální regulátor vozidla zadávající tažnou a brzdnou sílu pro pohon.
ČSN	Československá norma – Chráněné označení českých technických norem. Úřad pro technickou normalizaci, metrologii a státní zkušebnictví má na starost vydávání a tvorbu.
DMAIC	Define, Measure, Analyze, Improve, Control – Metoda pro zvyšování úrovně kvality, bezpečnosti, ochrany životního prostředí. Jedná se o zdokonalený PDCA cyklus.
EMC	Electromagnetic Compatibility – Vlastnost všech elektrických nebo magnetických přístrojů neovlivňovat nebo nebýt ovlivňován jiným zařízením.
FTA	Fault Tree Analysis – Analýza stromu poruchových stavů. Analytická technika pro vyhodnocení pravděpodobnosti selhání, respektive spolehlivosti.
FURPS	Functionality, Usability, Reliability, Performance, Supportability – Metoda na ověření a poznání kvality dodávaného software.
HW	Hardware – Soubor všech fyzicky existujících prvků. Technická výbava lokomotivy.
ISO	International Organization for Standardization – Světová federace národních normalizačních organizací.
MISRA	Motor Industry Software Realiability Association – Standard pro vývoj softwaru v programovacím jazyce C/C++.
NŘ	Nadřazené řízení – Oddělení vývoje SW ve Škoda Transportation.
OS	Operating system – Operační systém
PDCA	Plan, Do, Check, Act – Metoda založená na na čtyřech základních krocích pro neustálé zdokonalování procesů, kvality, výrobků, služeb atd.
PDF	Portable Document Format – Formát vytvořený firmou Adobe pro nezávislé používání dokumentů. Může obsahovat text i obrázky. Hlavní výhodou je, že se na všech zařízeních zobrazí stejně.
RNG	Regular Language for XML Next Generation – Specifické schéma pro strukturu a obsah XML dokumentu.
SIL	Safety Integrity Level – Stupeň integrity bezpečnosti. Relativní úroveň rizika zajištěného bezpečnostní funkcí.
SW	Software – Součást počítačového systému, složený z kódovaných informací nebo strojových instrukcí.
ŠT (ŠTRN)	Škoda Transportation a.s.
TCU	Traction control unit – Samostatný řídicí systém pro řízení trakčních měničů a jejich ochran.
TDD	Technology Diagnostic Display – Displej pro drážní vozidla
TSI	Technical Specifications for Interoperability – Směrnice o interoperabilitě pro vysokorychlostní a konvenční železnice.
TÜV	Technischer Überwachungsverein – Nadnárodní organizace, která se

	zaměřuje na technickou inspekci a certifikaci.
VCU	Vehicle control unit – Redundantní řídicí počítač lokomotivy složený ze tří procesorových karet (LCC+ATO+DCC).
XML	eXtensible Markup Language – Rozšiřitelný značkovací jazyk.

1. Úvod – postupy pro zajištění kvality SW

1.1 Statická analýza kódu

Statická analýza kódu je jednou z nejvýznamnějších metod pro vyhledání a odstranění chyb SW. Na základě dlouhodobých zkušeností lze konstatovat, že kvalitní statickou analýzou zdrojového kódu je možno odhalit až 20% chyb, které se v kódu vyskytují.

Převážná většina chyb v SW má technologický charakter (chybná implementace zadání nebo správná implementace chybného zadání). Takovéto chyby musí být odhaleny během testování a zejména během validace SW. Ve vyhledávání chyb tohoto druhu není statická analýza příliš úspěšná.

Statická analýza kódu se zaměřuje na chyby SW, které mohou být potencionálně nebezpečné pro běh programu (např. zápis na chybné místo paměti) nebo na takové, které se nemusí za normálních podmínek projevit. Časté jsou také tzv. kopírovací chyby nebo chyby pramenící z nedodržení normy kódování SW.

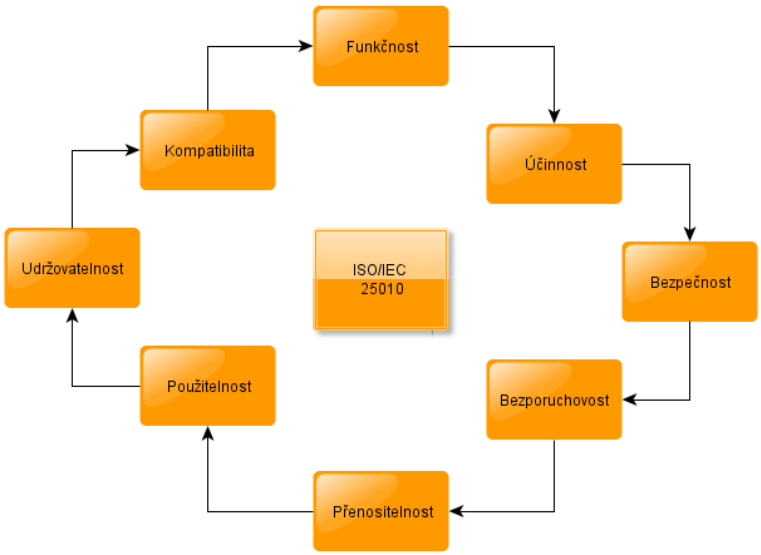
1.2 Testování a validace SW

Pomocí testování se snažíme zjistit, jestli software splňuje dané funkční požadavky, které jsou na něj kladeny a objevovat odchylky od stavu, který je požadován. Stručně řečeno, hledáme chyby ve funkci softwaru. Testeři kontrolují splnění samotných specifikovaných požadavků, ale také jestli požadavky splňují představy zákazníka. Požadavky nám určují, jaký má daný software být a jaké má mít vlastnosti.

Zatímco testování SW bývá často prováděno v laboratoři za použití simulace vstupních signálů, validace je prováděna na reálném zařízení při použití skutečných technologických procesů.

Existují modely, které obecně kategorizují softwarové charakteristiky, například mezinárodní norma ISO/IEC 25010 nebo model FURPS.

ISO/IEC 25010 nám udává osm charakteristik a pro každou z nich několik subcharakteristik.



Obrázek 1 - Kritéria kvality dané normou ISO/IEC 25010

FURPS je složený z počátečních písmen podstatných znaků kvality: Funkčnost (funkcionality), Použitelnost (Usability), Spolehlivost (Reliability), Výkonnost (Performance), Rozšiřitelnost (Supportability).

1.3 Řízení/kontrola kvality (QC)

Na kvalitu výsledného SW má zásadní vliv celý proces jeho tvorby. Kontrola kvality se proto neomezuje jen na testování výsledného SW, ale zabývá se všemi požadovanými a použitými postupy. Kontroluje se kvalita všech faktorů, které jsou různými způsoby propojeny s vývojem softwaru, například dodržování domluvených pravidel, procesů, předpisů, porovnávání testů z různých verzí SW atd. a poskytuje zpětnou vazbu o příčinách problémů s kvalitou SW. Řízení/kontrola kvality je specifikována dle řady norem ISO 9000.

Kontrola kvality z pohledu procesu ověřuje splnění výstupů dané etapy a existenci nutných vstupů pro etapu následující. Systematickým přístupem se snaží identifikovat a eliminovat případné nedostatky.

1.4 Zajištění/prokazování kvality (QA)

Zajišťování kvality – Quality Assurance (QA) je proces systematického sledování a hodnocení různých aspektů projektu, služby nebo zařízení s cílem splnění norem kvality.

Proces zajišťování kvality maximalizuje kvalitu ve všech procesech životního cyklu produktu od specifikace požadavků až po implementace služeb a technickou podporu. Jde vlastně o včasnou detekci systematických poruch a rizik v průběhu celého životního cyklu

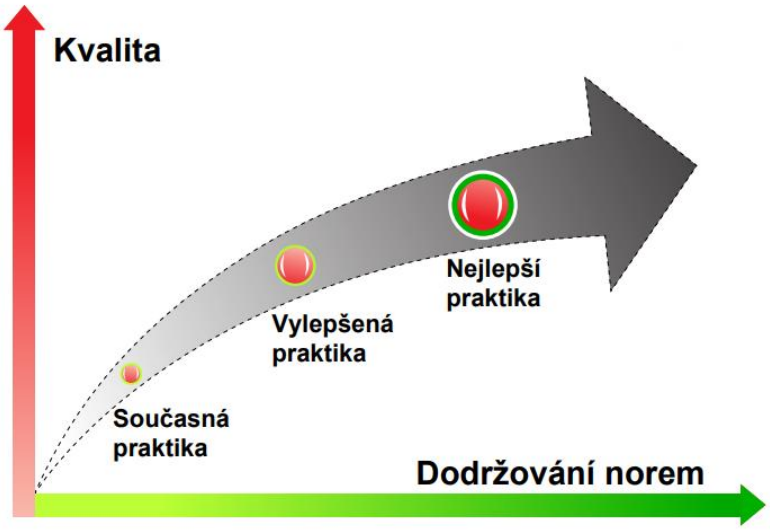
vývoje softwaru. Dalším faktorem, které QA musí splňovat, je vyhodnocování provozu a v případě potřeby spouštět změnové řízení, které zamezí důsledkům odhalených chyb. Tím docílíme snížení nákladů projektů, vyšší kvality a spolehlivosti produktu.

1.5 Podnikové prostředí

S rozvojem techniky rostou požadavky na zajišťování kvality a bezpečnosti produkce. Současně s tím rostou i požadavky na prokazování toho, že metody QA jsou důsledně používány. Správně zavedené a používané metody QA zvyšují konkurenceschopnost firem v tržním prostředí a s tím spojený zisk. Existují různé všeobecné návody a postupy pro Quality Management (QM) jako například PDCA – Plan, Do, Check, Act (plánování, provedení, kontrola, akce) nebo projektové metody Six Sigma (SS). SS je strategie řízení, klade si za cíl identifikovat a odstranit příčiny defektů a chyb v procesech výroby a obchodu, k čemuž využívá metodiku DMAIC: Define (Definice), Measure (Měření), Analyze (Analýza), Improve (Zlepšení), Control (Kontrola).

Implementace postupů pro zajištění kvality je také předmětem hodnocení a certifikace ze strany globálně fungujících organizací či institucí. Jednou z nejznámějších institucí je TÜV NORD. Získání certifikátu o zajišťování kvality je dnes již nutnou podmínkou pro účast v soutěžích a tenderech na významné zakázky.

V dnešní době je již zavádění QA do stávajících procesů neodmyslitelné, stává se z ní běžná součást rozvoje podniků. Důležité je, aby zavedení QA nebylo samoúčelné, ale aby skutečně vedlo ke zvýšení a udržení kvality produkce. V takovém případě se dostaví žádoucí efekt ve formě vyšších zisků. Dříve bylo běžné, že si velké podniky stanovovaly požadavky na kvalitu svých produktů samy (podnikové normy). V současné době dochází k postupné synchronizaci podnikových norem s obecně platnými standardy.



Obrázek 2 - Závislost dodržování norem na kvalitě [4]

Ve světě již dnes existuje řada mezinárodních organizací pro normalizaci, například:

- International Organization for Standardization (ISO).
- International Eletrotechnical Commision (IEC).
- Institute of Electrical and Electronics Engineers (IEEE).

2. Normy a bezpečnost

2.1 Norma

Co je vlastně norma? Využijeme-li formální definici od ISO, tak se dozvíme, že: Norma je dokument ustanovený na základě dohody a schválený uznávaným subjektem. Pro obecné a opakované použití poskytuje pravidla, směrnice nebo charakteristiky ve vztahu s činností a jejich výsledky. Usiluje o dosažení optimálního uspořádání v dané souvislosti.

Norma popisuje sjednaný, opakovatelný způsob provedení úkonu. Poskytuje návod pro navrhování, používání nebo vlastnosti materiálů, výrobků, procesů, systémů, služeb a osob. Je to technická specifikace s přesnými kritérii, pravidly, pokyny, definicemi pro QA. Ustanovení normy je podloženo znalostí správného postupu k dosažení cíle.

Další definice, která je převzata říká: Norma, také někdy standard, je vyjádřením požadavků na to, aby výrobek, proces nebo služba byly za specifických podmínek vhodné pro daný účel. Stanoví základní požadavky na kvalitu a bezpečnost, slučitelnost, zaměnitelnost, ochranu zdraví a životního prostředí. Usnadňuje volný pohyb zboží v mezinárodním obchodu, snaží se, aby výroba byla racionální, aby se ochrana životního prostředí a konkurenceschopnost vzájemně podporovaly, aby na vnitřním trhu byli spotřebitelé dostatečně chráněni [6].

Normy můžeme nejobecněji rozdělit na absolutní (normativní, descendentní), které předepisují a vyžadují, jsou ustanoveny nějakou autoritou a relativní (popisné, ascendentní), které popisují normalitu jako to, co je běžné a obvyklé. Další způsobem se dají rozdělit jako oficiální normy a praktické zásady. Oficiální normy mohou být právní anebo dobrovolné, bez donucení zákonem.

2.2 Organizace pro normalizaci

Smyslem organizací pro normalizaci je vytvářet a uveřejňovat standardy pro rozličná odvětví lidské činnosti. Dodržováním předepsaných standardů (norem) je zajištěna určitá kvalita a bezpečnost produktů či služeb. Dodržování norem je zároveň předmětem kontrol ze strany legitimních kontrolních orgánů, což má opět za cíl zajistit dostatečnou míru kvality a bezpečnosti. Dále jsou pro příklad uvedeny nejznámější organizace zabývající se normalizací.

2.2.1 Mezinárodní organizace

- **ISO** – Světová federace národních normalizačních organizací se sídlem v Ženevě, čítá

163 členů.

Obrázek 3 - logo ISO

- **IEC** – Mezinárodní elektrotechnická komise, která vypracovává mezinárodní normy pro elektrotechniku, elektroniku, sdělovací techniku a příbuzné odvětví, čítá 82 členských států.

Obrázek 4 - logo IEC

- **IEEE** – Institut pro elektrotechnické a elektronické inženýrství, která usiluje o vzestup technologie související s elektrotechnikou, čítá nejvíce členů technické profese na světě, a to přes 360 000 ve 175 zemích.

Obrázek 5 - logo IEEE

2.2.2 Regionální organizace

- **CEN** – Evropský výbor pro normalizaci, jejímž posláním je podpora evropské ekonomiky v globálním obchodu, poskytováním základního kamene při rozvoji,

údržbě a šíření ucelených souborů norem.

Obrázek 6 - logo CEN

- **CENELEC** – Evropský výbor pro normalizaci v elektrotechnice se stejným posláním jako CEN.

Obrázek 7 - logo CENELEC

- **ETSI** – Evropský ústav pro telekomunikační normy, stará se o standardizaci v telekomunikačním průmyslu, nejvýznamnější výsledek skupiny ETSI je standardizace mobilní sítě GSM.

Obrázek 8 - logo ETSI

- **MISRA**

Motor Industry Software Reliability Association (MISRA) je organizace, která vydává standardy pro vývoj softwaru. Cílem standardu je zlepšení bezpečnosti kódu, spolehlivosti a

přenositelnosti v embedded systémech. MISRA vydává dva standardy:

- MISRA C – standard pro vývoj softwaru v programovacím jazyce C.
- MISRA C++ - standard pro vývoj softwaru v programovacím jazyce C++.

MISRA byla vytvořena převážně pro automobilový průmysl, ale v dnešní době se stále více vývojářů i mimo automobilový průmysl odráží právě od těchto standardů. „MISRA se vyvinula v široce přijímaný soubor nejlepších praktik vedoucími vývojáři v oblastech aerospace, telekomunikačních, zdravotnických zařízení, obranných, železničních a dalších.“**[5]** První vydání standardu MISRA C bylo vydáno v roce 1998 a druhé vydání (momentálně aktuální) v roce 2004. Standard MISRA C++ byl vytvořen na základě standardu MISRA C v roce 2008.

2.2.3 Národní organizace

- **ÚNMZ** – Úřad pro technickou normalizaci, metrologii a státní zkušebnictví, který má na starost produkovat české technické normy ČSN.

Obrázek 9 - logo ÚNMZ

- **ANSI** – Americký národní standardizační institut, vytváří průmyslové standardy v USA, členem IEC a ISO.

Obrázek 10 - logo ANSI

- **DIN** – Německý ústav pro průmyslovou organizaci.

Obrázek 11 - logo DIN

2.3 Normy pro vývoj SW kolejových vozidel

Pro tvorbu SW kolejových vozidel a ostatních bezpečnostně relevantních zařízení je základem norma ČSN EN 50128. Její první verze vyšla v roce 2002 jako překlad normy EN 50128:2001. V současnosti platí revize této normy z roku 2011. V současné době je v návrhu také mezinárodní standard EN 50657, který navazuje na normu EN 50128 a upřesňuje její požadavky pro použití v kolejových vozidlech.

3. Životní cyklus systému

Životní cyklus systému (vyjádřený např. V-diagramem) je podobný nebo shodný pro vývoj všech systémů lokomotivy:

- Mechanické části.
- Brzdové systémy.
- Silovou elektrickou výzbroj.
- Řídicí obvody.
- **Software.**

K tvorbě dokumentů životního cyklu je možno přistupovat různými způsoby, ale v každém případě musí být zmíněny a komentovány všechny požadavky norem EN 50 126, **EN 50 128** a EN 50 129 pro danou etapu. V případě, že není možné požadavek splnit, musí být uveden důvod (např. odlišná koncepce, irelevantnost,…).

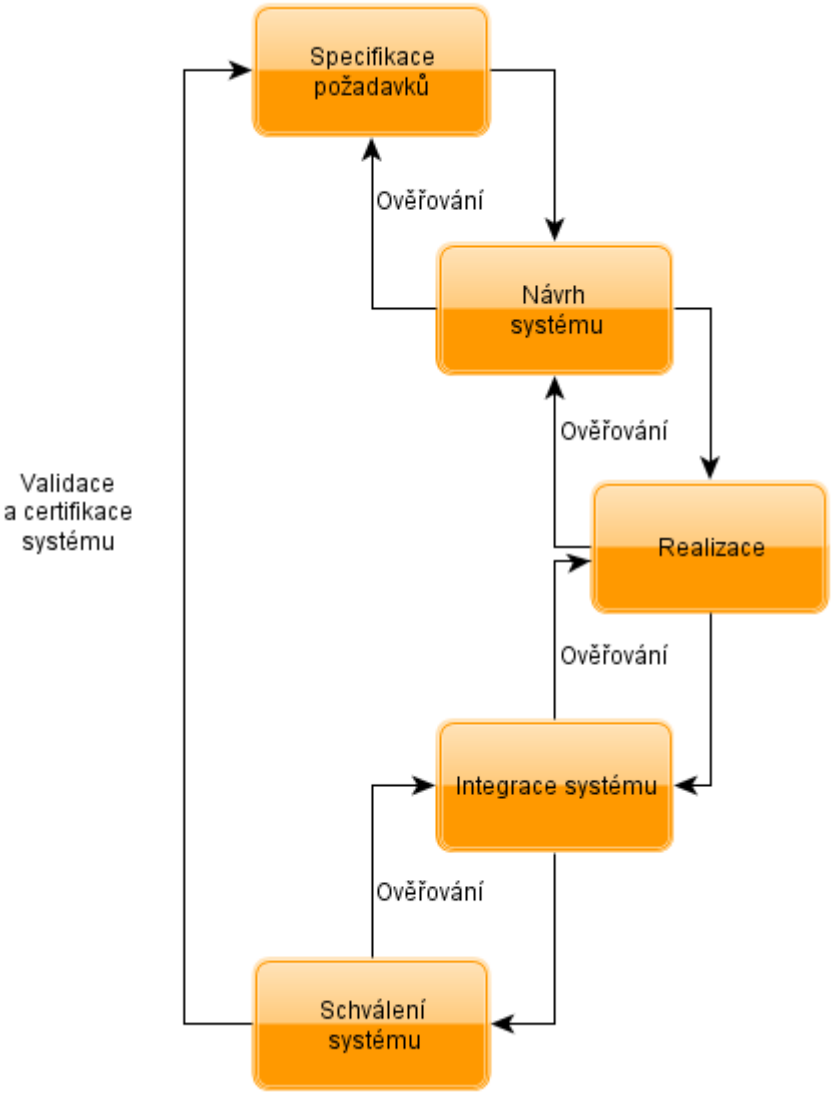
Životní cyklus často popisuje vývoj nového produktu metodou „shoda dolů“. Tato metoda by měla zajistit správnou implementaci požadavků zákazníka a legislativy.

3.1 Fáze životního cyklu SW

Činnosti, které jsou prováděny během životního cyklu SW lze rozdělit do určitých fází:

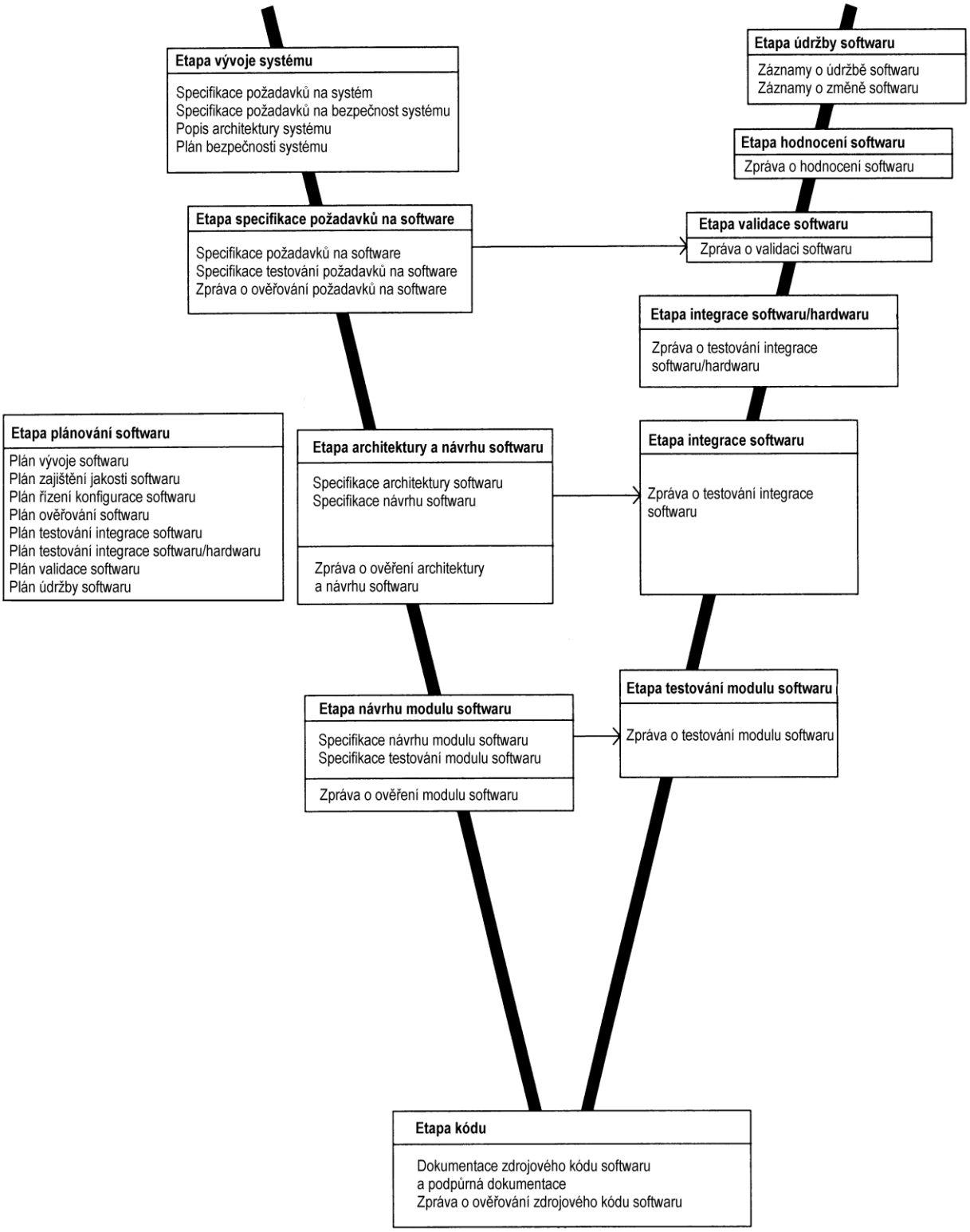
- Etapa specifikace požadavků na systém.
- Etapa specifikace požadavků na software.
- Etapa architektury a návrhu software.
- Etapa návrhu komponent softwaru.
- Etapa implementace a testování komponent.
- Etapa integrace.
- Etapa celkového testování softwaru / závěrečná validace.
- Etapa údržby SW.

Jednotlivé etapy na sebe navzájem navazují – výstupy jedné tvoří vstupy té následující. Pro každou etapu jsou definovány činnosti a dokumenty, které musí být vytvořeny. Po ukončení každé etapy musí být provedena verifikace splnění požadavků definovaných pro danou etapu. Výsledkem verifikace je ověřovací zpráva, která shrnuje návaznost na předchozí etapu a připravenost výstupů pro etapu další. V případě rozporu s předchozí etapou musí být předmět rozporu buď řádně odůvodněn, nebo musí být specifikace přepracována do souladu s požadavky předchozí etapy.



Obrázek 12 - Ověřování jednotlivých etap systému

3.2 V - diagram a rozdělení do etap



Obrázek 13 - V-diagram [2]

V-diagram je jeden z modelů popisujících postupy a vazby aplikované během celého životního cyklu software. Jednotlivé etapy jsou vykonávány postupně, tak jak symbolizuje písmeno V (shora dolů a pak zdola nahoru). Tvar písmene V umožňuje vyjádřit vazby, které se uplatňují mezi určitými procesy. Například integrace software se provádí v souladu s etapou architektura a návrh software. Výstupy jednotlivých etap tvoří základ pro realizaci etap následujících. Levá strana písmena V znamená rozklad požadavků a vytváření specifikace systému. Pravá strana písmena V zastupuje integraci dílů a jejich validaci.

Norma ČSN EN 50 128 popisuje podrobně jednotlivé etapy vývoje software, jejich vstupy i výstupy. V následujících odstavcích je uvedena jejich stručná charakteristika.

3.2.1 Etapa specifikace požadavků na systém

Cíle

Poskytnout popis systému, z vyšších úrovní abstrakce dolů až po podrobné zjemňování s cílem vytvořit takovou strukturu systému, která bude zárukou funkčnosti a bezpečnosti a vytvoří vhodné podmínky pro budoucí činnosti údržby.

Požadavky

Vytvořit pro software dokumenty dle rozsahu požadované úrovně integrity bezpečnosti.

3.2.2 Etapa specifikace požadavků na software

Cíle

Popsat kompletní sadu požadavků na software, splňující všechny požadavky na systém a bezpečnost a poskytnout úplnou sadu dokumentů pro každou následující etapu. Popsat specifikaci celkového testování softwaru.

Vstupní dokumenty

- Specifikace požadavků na systém.
- Specifikace požadavků na bezpečnost systému.
- Popis architektury systému.
- Specifikace externích rozhraní.
- Plán zajištění kvality softwaru.
- Plán validace softwaru.

Výstupní dokumenty

- Specifikace požadavků na software.
- Specifikace celkového testování softwaru.

- Zpráva z verifikace požadavků na software.

Požadavky

Specifikace požadavků na software musí vyjadřovat požadované vlastnosti softwaru, který je vyvíjen. Tyto vlastnosti, které jsou všechny (mimo bezpečnost) definovány souborem norem ISO/IEC 9126, musí zahrnovat:

- Funkcionalitu (včetně výkonnosti a doby odezev).
- Robustnost a udržovatelnost.
- Bezpečnost (včetně bezpečnostních funkcí a jejich přiřazených úrovní integrity bezpečnost softwaru).
- Efektivitu.
- Použitelnost.
- Přenositelnost.

3.2.3 Etapa architektury a návrhu software

Cíle

- Vyvinout architekturu softwaru, která splňuje požadavky na software.
- Identifikovat a vyhodnotit důležitost interakcí mezi HW a SW z pohledu bezpečnosti.
- Vybrat metodu návrhu.
- Navrhnout SW s definovanou úrovní SIL dle vstupních dokumentů.
- Zajistit testovatelnost systému od jeho počátku, jelikož verifikace a testování budou rozhodujícím prvkem validace.

Vstupní dokumenty

- Specifikace požadavků na software.

Výstupní dokumenty

- Specifikace architektury softwaru.
- Specifikace návrhu softwaru.
- Specifikace rozhraní softwaru.
- Specifikace testů integrace softwaru.
- Specifikace testů integrace softwaru/hardwaru.
- Zpráva z verifikace architektury a návrhu softwaru.

Požadavky

Navržená architektura softwaru musí být uvedena a podrobně popsána ve specifikaci architektury softwaru.

Specifikace architektury musí brát v úvahu dosažitelnost specifikace požadavků na software při požadované úrovni integrity SIL.

Specifikace architektury softwaru musí identifikovat všechny SW komponenty a pro tyto komponenty identifikovat:

- Jestli tyto komponenty jsou nové nebo již existující.
- Jestli byly komponenty dříve validovány.
- Úroveň SIL.

Komponenty softwaru musí:

- Pokrýt definovanou podmnožinu požadavků na SW.
- Být jasně identifikované a nezávisle verzované uvnitř systému řízení konfigurace.

3.2.4 Etapa návrhu komponent softwaru

Cíle

Vytvořit návrh komponent softwaru, který splňuje požadavky specifikace návrhu softwaru v rozsahu požadovaném úrovní integrity bezpečnosti softwaru.

Vytvořit specifikaci testů komponent softwaru, které splňují požadavky specifikace návrhu komponent softwaru v rozsahu požadovaném úrovní integrity bezpečnosti softwaru.

Vstupní dokumenty

- Specifikace návrhu softwaru.

Výstupní dokumenty

- Specifikace návrhu komponent softwaru.
- Specifikace testů komponent softwaru.
- Zpráva z verifikace návrhu komponent softwaru.

Požadavky

Pro každou komponentu softwaru musí být dostupné následující informace:

- Autor.
- Historie konfigurace.
- Stručný popis.

Specifikace návrhu SW komponenty musí určit:

- Identifikaci všech nejnižších SW jednotek (např. subrutiny, metody, procedury) sledované zpět do vyšších úrovní.
- Jejich detailní rozhranní s prostředím a dalšími komponentami s podrobně popsanými vstupy a výstupy.
- Jejich úroveň integrity bezpečnosti bez dalšího dělení uvnitř samotné komponenty.
- Podrobně popsané algoritmy a datové struktury.

3.2.5 Etapa implementace a testování komponent

Cíle

Dosáhnout softwaru, který je analyzovatelný, testovatelný, verifikovatelný a udržovatelný. V této etapě je zahrnuto i testování komponent.

Vstupní dokumenty

- Specifikace návrhu komponenty softwaru.
- Specifikace testů komponenty softwaru.

Výstupní dokumenty

- Zdrojový kód softwaru a podpůrná dokumentace.
- Zpráva z testů komponenty softwaru.
- Zpráva z verifikace zdrojového kódu softwaru.

Požadavky

Velikost a složitost vyvíjeného zdrojového kódu musí být vyvážená. Zdrojový kód musí být čitelný, pochopitelný a testovatelný. Zdrojový kód musí být jednoznačně identifikován před zahájením dokumentovaného testování.

Zpráva z testů komponenty softwaru musí zahrnout následující:

- Údaje o výsledcích testů a o tom, zda každá komponenta splnila požadavky uvedené ve své specifikaci návrhu komponenty softwaru.
- Údaje o pokrytí kódu testem musí být poskytnuty pro každou komponentu a prokázat, že požadovaný stupeň pokrytí kódu testem byl dosažen pro všechna požadovaná kritéria.

3.2.6 Etapa integrace

Cíle

Provést integraci softwaru a integraci softwaru/hardwaru.

Ukázat, že software a hardware spolupracují správně a provádějí zamýšlené funkce.

Vstupní dokumenty

- Specifikace testů integrace softwaru/hardwaru.
- Specifikace testů integrace softwaru.

Výstupní dokumenty

- Zpráva z testů integrace softwaru.
- Zpráva z testů integrace softwaru/hardwaru.
- Zpráva z verifikace integrace softwaru.

Požadavky

Integrace SW komponent musí být proces postupného slučování jednotlivých a dříve testovaných komponent do složeného celku za tím účelem, aby mohly být rozhraní komponent a sestaveného softwaru dostatečně ověřené před provedením systémové integrace a systémových testů.

Během integrace softwaru/hardwaru musí být jakákoliv změna integrovaného systému podrobena studii dopadů, která identifikuje všechny ovlivněné komponenty a nezbytné opětovné verifikační činnosti.

Zpráva z verifikace integrace softwaru musí být napsána v souladu s obecnými požadavky stanovenými pro zprávu z verifikace.

3.2.7 Etapa celkového testování softwaru / závěrečná validace

Cíle

Analyzovat a testovat integrovaný software a hardware pro zaručení shody se specifikací požadavků na software s důrazem na funkční a bezpečnostní aspekty v souladu s úrovní integrity bezpečnosti a pro kontrolu zda je vhodný pro zamýšlené použití.

Vstupní dokumenty

- Specifikace požadavků na software.
- Specifikace celkového testování softwaru.
- Plán verifikace softwaru.

- Plán validace softwaru.
- Veškerá dokumentace hardwaru a softwaru včetně výsledků z částečných verifikací.
- Specifikace požadavků na bezpečnost systému.

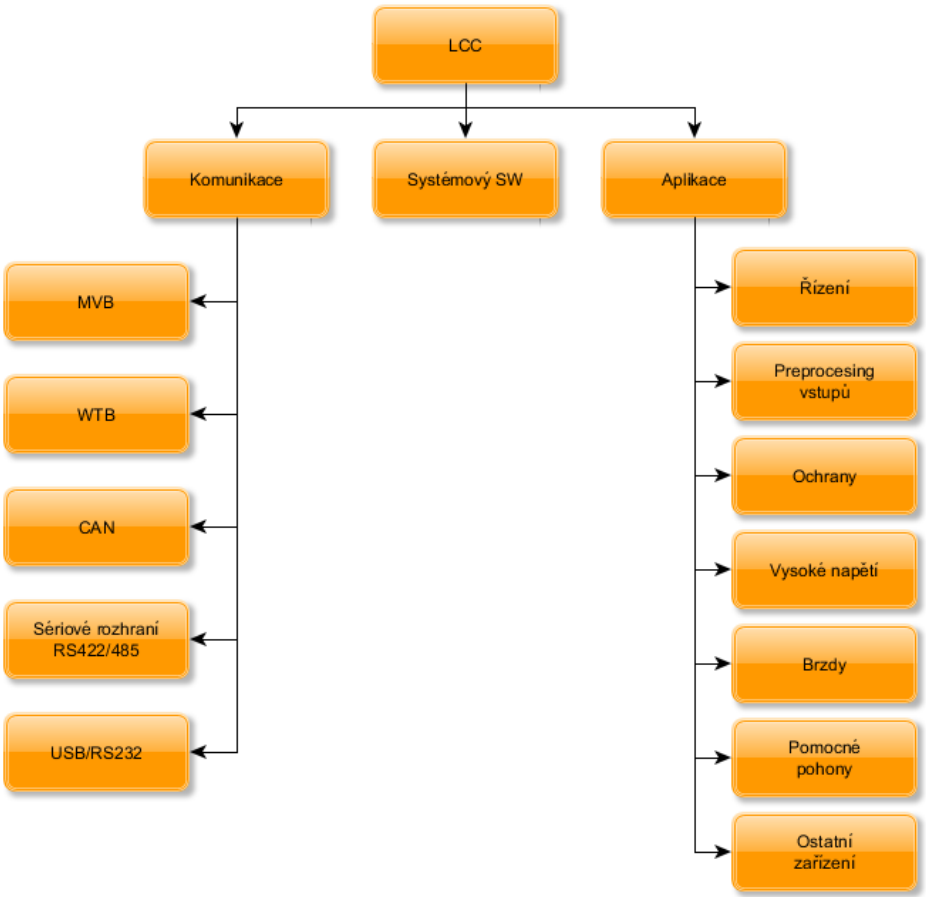
Výstupní dokumenty

- Zpráva z celkového testování softwaru.
- Zpráva z validace softwaru.
- Poznámky k vydání.

Požadavky

Zpráva z celkového testování SW musí být napsána na základě specifikace celkového testování SW. Validátor musí specifikovat a provést doplňkové testy podle svého uvážení nebo je nechat provést testerem. Zatímco celkové testy SW vycházejí především ze struktury specifikace požadavků na SW, validátor může jejich hodnotu zvýšit o testy, které zatíží systém komplexními scénáři odrážejícími aktuální potřeby uživatele.

Výsledky všech testů a analýz musí být zaznamenány ve zprávě z celkového testování softwaru.



Obrázek 14 - Rozdělení SW na prvky

3.2.8 Bezpečnost SW

Při specifikaci požadavků na systém je zpracována také tzv. riziková analýza. V této analýze jsou uvedena všechna možná bezpečnostní rizika systému, jejich závažnost a možná souvislost se softwarem. Částem SW, které mají za úkol eliminovat některé z rizik, se říká bezpečnostní funkce. Pro každou bezpečnostní funkci musí být určena úroveň integrity bezpečnosti (SIL).

Specifikace požadavků bezpečnosti SW musí být dostatečně podrobná, aby návrh a jeho realizace umožňovaly dosažení požadované integrity bezpečnosti i provedení odhadu funkční bezpečnosti. Stupeň podrobnosti se může měnit v závislosti na složitosti konkrétní aplikace.

Na základě analýzy rizik a architektury systému a subsystému bude přiřazena hodnota SIL pro jednotlivé komponenty SW zajišťující bezpečnostní funkce dle EN 50128, ale také s ohledem na požadavky drážních úřadů.

Úroveň integrity bezpečnosti softwaru	**Popis úrovně SIL**
4	Velmi vysoká
3	Vysoká
2	Střední
1	Nízká
0	Nevztahuje se k nebezpečí

Tabulka 1- Popis úrovní SIL

Úroveň integrity bezpečnosti SW (SIL), kterou musí příslušná komponenta SW splňovat, má na celý proces vývoje SW podstatný vliv. Definuje požadavky na personální obsazení, nezávislost členů vývojového týmu, použité metody pro vývoj a testování SW a řadu dalších.

4. Projekt statické analýzy SW v podmínkách ŠT

4.1.1 Hlavní úkoly diplomové práce

Tato kapitola popisuje hlavní přínos mé diplomové práce – vytvoření funkčního prostředí s výkonnými SW nástroji pro statickou analýzu kódu vytvořeného ve Škoda Transportation a.s.

Komplexní nástroj pro statickou analýzu SW v podmínkách ŠT jsem realizoval tak, aby poskytoval následující služby:

1. **Návaznost na úložiště SW** - konkrétně bylo nutno připravit programové skripty pro ovládání systému Mercurial fungujícího na síťovém disku ŠT.
2. **Kontrola kódu dle vlastní specifikace** - bylo nutno převzít a podstatně rozšířit (doprogramovat) funkčnost SW nástroje „Skoda Code Checker“, který vznikl za účelem cílené kontroly plnění normy kódování v ŠT.
3. **Kontrola kódu dle všeobecně uznávaných pravidel** - celý proces byl doplněn o kontrolu kódu pomocí všeobecně rozšířeného nástroje – konkrétně Cppcheck, který je dostupný pod licencí GNU General Public License.
4. **Možnost ručního prověření nálezů** – byl vybrán a pro tento účel vhodně nakonfigurován SW nástroj Sonar scaner.
5. **Prezentace výsledků celé analýzy** – pro prezentaci výsledků jsem použil a nakonfiguroval SW nástroj SonarQube.

4.1.2 Použité nástroje a postupy

Statická analýza souhrnně označuje techniky pro automatické nebo manuální hledání chyb ve zdrojovém kódu, aniž by tento kód byl spuštěn. Zkušenosti ukazují, že při dynamickém testování, ačkoliv se snažíme pokrýt testy co největší část kódu (tzv. code coverage) a použít co nejrozmanitější vstupy, zůstávají některé chyby neodhaleny. Tuto mezeru v kontrole kódu se snaží vyplnit statická analýza, která má potenciál k odhalení problémů, které se vyskytují při všech možných variantách běhu, aniž by bylo nutné kód spouštět. Další výhodou je to, že analýza může být prováděna už během psaní kódu, takže lze chyby opravovat přímo při jejich vzniku, kdy ještě nemusí být program spustitelný. Je proto výhodou, zařadit takovouto analýzu přímo jako součást vývojového cyklu a její přínosy zužitkovat přímo při samotné tvorbě aplikací. Navíc je tak programátorům poskytována určitá forma vzdělání – je pravděpodobné, že po opakovaném upozornění na chyby dojde k omezení frekvence jejich

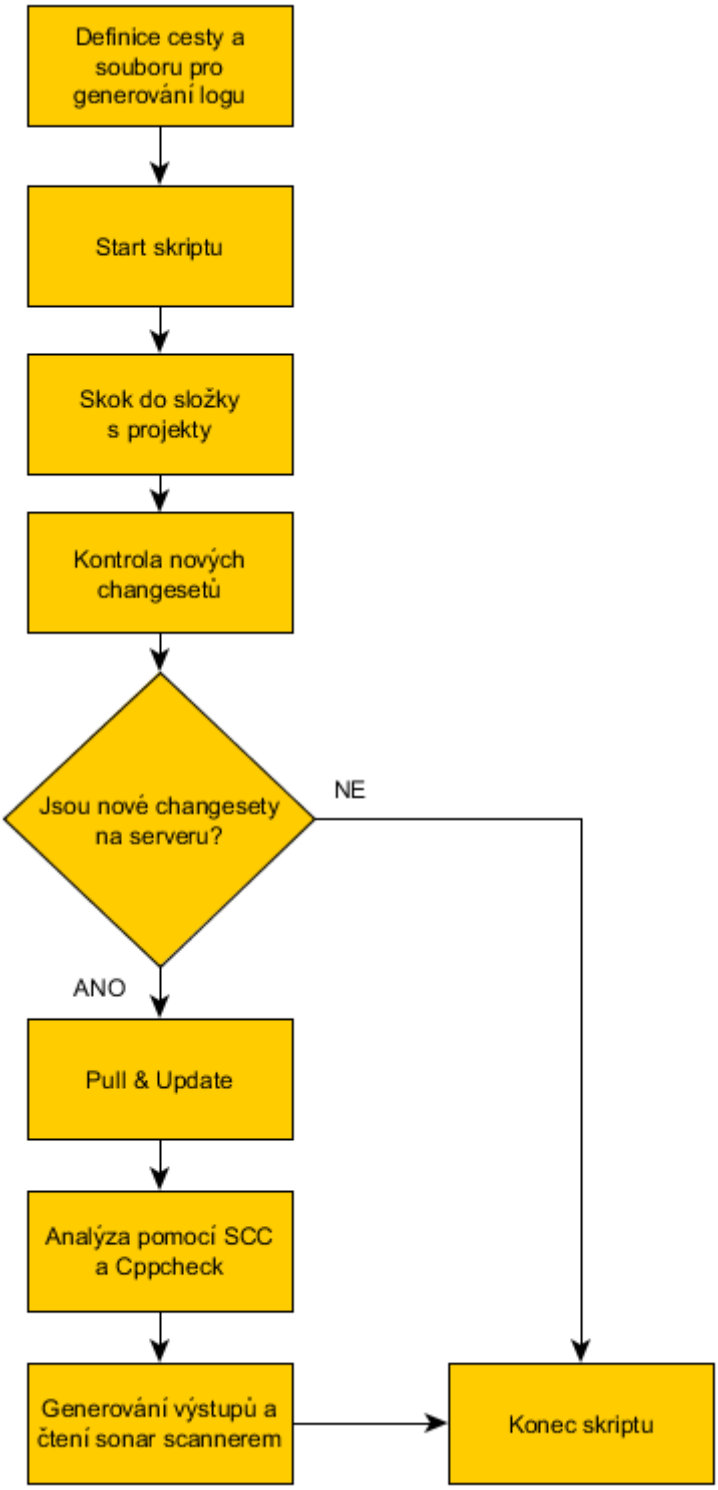
výskytu.

Účinnost statické analýzy kódu zvyšuje kombinované použití několika různých technik:

- **Manuální kontrola zdrojového kódu:** Tato metoda je velice efektivní v režimu údržby SW – kontrola se zaměřuje pouze na provedené změny SW. Dále je tato metoda nutným doplňkem strojových kontrol, které upozorní na potenciálně chybný nebo rizikový kód, ale toto upozornění je nutno manuálně prověřit.
- **Kontrola kódu vlastním „na míru psaným“ programem:** Tato metoda kontroluje zejména dodržování norem kódování, která mohou obsahovat pravidla specifická pro konkrétní SW. Dále tato metoda vyhledává potenciálně rizikové operace a překládá je k manuálnímu prověření. Kontrola může být zaměřena také na další specifické operace:
    - Kontrola zpracování vstupních signálů.
    - Kontrola plnění výstupních signálů.
    - Kontrola zápisu do jedné globální proměnné pouze na jednom místě.
    - Kontrola inicializace všech globálních proměnných a polí.
- **Kontrola kódu volně šiřitelným SW:** Tyto programy jsou již dnes velmi kvalitní a dokážou vyhodnotit testovaný SW opravdu globálně a v souvislostech. Zpravidla však generují také řadu falešných chybových hlášek, takže je nutné vše manuálně prověřit. Příkladem, který bude využit i v této diplomové práci je CppCheck.

Pro co největší efektivitu statické analýzy kódu je vhodné výše uvedené metody vzájemně zkombinovat. Vhodně navázat na verzovací systém SW a vyřešit propojení s interní politikou vývoje SW ve společnosti Škoda.

Pro provázání a postupné provedení jednotlivých operací jsem naprogramoval skript, jehož zjednodušený vývojový diagram je na následujícím obrázku.



Obrázek 15 - Algoritmus procesu analýzy kódu

4.2 Návaznost na síťové úložiště SW ve ŠT

4.2.1 Mercurial

Multiplatformní, verzovací nástroj pro vývoj softwaru. Kompatibilní s operačním systémem Windows, ale také se systémy typu Unix (Linux, Mac OS, atd.). Mercurial je určen primárně pro použití v příkazovém řádku, dostupné jsou ale i grafická uživatelská rozhraní. Veškeré operace nástroje Mercurial jsou volány v podobě parametrů hlavního příkazu *hg.* (chemická značka rtuti).

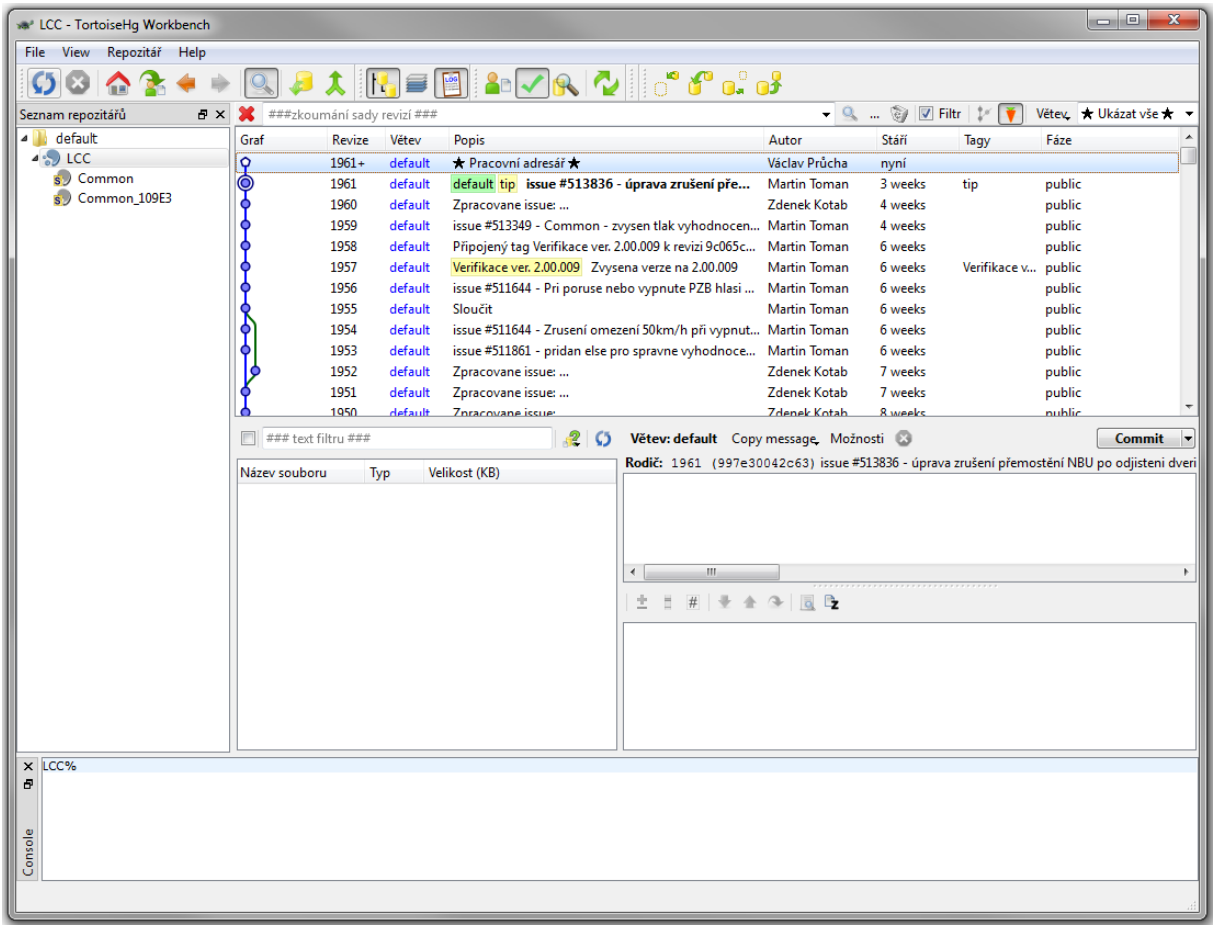
Obrázek 16 - Logo Mercurial

Hlavní přednosti Mercurialu:

- Vysoká výkonnost a škálovatelnost.
- Plně distribuovaný týmový vývoj SW.
- Možnost pokročilého větvení a slévání (merge).

4.2.2 TortoiseHg

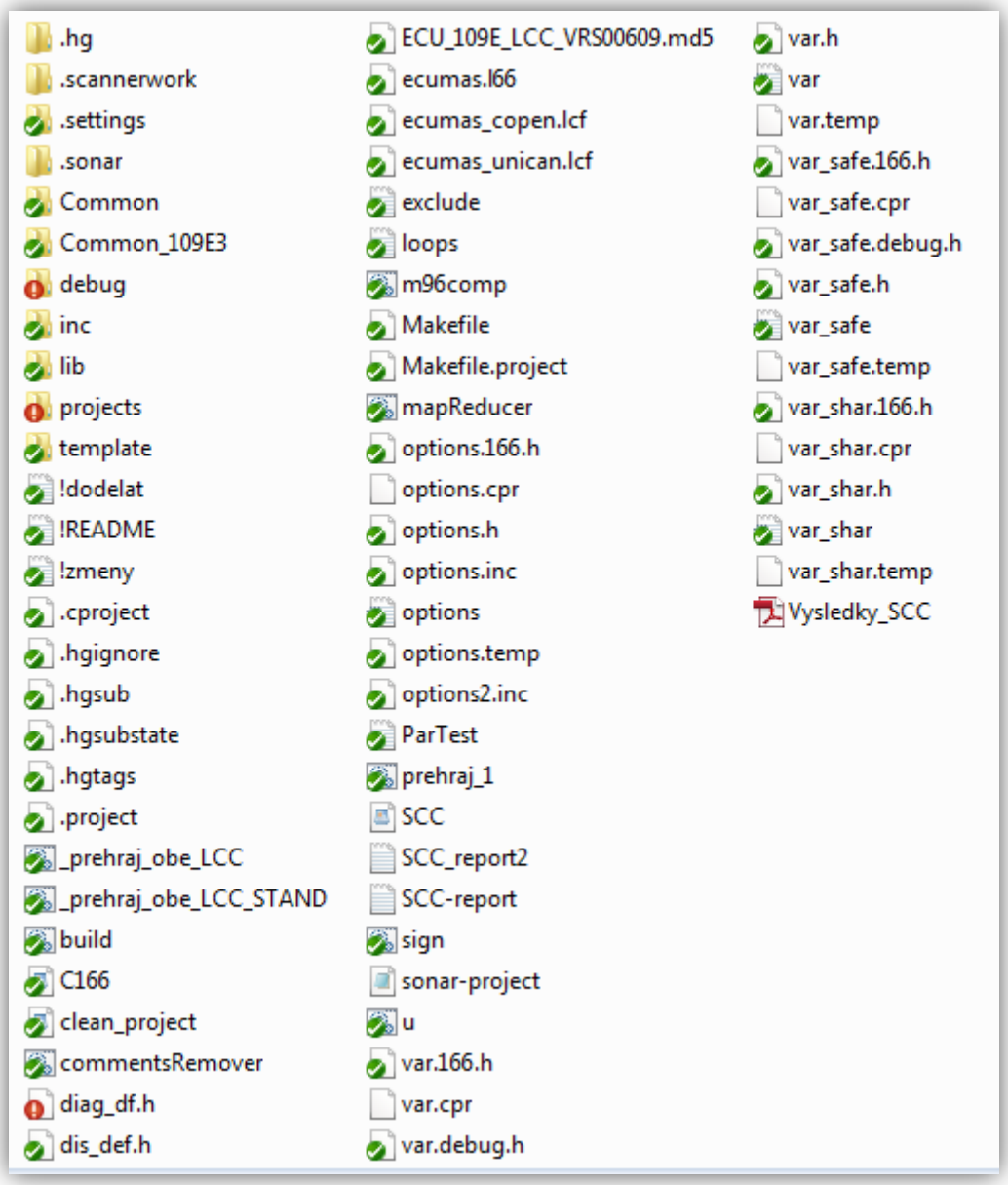
TortoiseHg je volně dostupný program, který obsahuje sadu grafických nástrojů a extenzí uživatelského rozhraní programu Mercurial pro distribuovanou správu verzí.



Obrázek 17 – Struktura revizí SW projektu LCC

4.2.3 Propojení Mercurialu s procesem analýzy

Prvním krokem celého procesu analýzy je stáhnout a aktualizovat SW projekt ze síťového na lokální disk pro další zpracování. Při každém procesu analýzy se zjistí, zda došlo k nějakým úpravám či změnám na vybraných projektech. Pokud ano, tak pomocí Mercurial příkazů s prefixem hg dokážeme stáhnout změny SW projektu ze síťového na lokální disk. Při úspěšné aktualizaci SW projektu se proces dostane na další krok a tím je analýza pomocí vlastní, na míru psané aplikace Škoda Code Checker a volně šiřitelným SW nástrojem Cppcheck.



Obrázek 18 - Struktura adresáře SW projektu LCC

Propojení SonarQube s programem TortoiseHg je řešeno pomocí skriptu psaném ve windows powershell (příloha A). Řešení je znázorněno ve vývojovém diagramu (obrázek č. 15). V prvním kroku se nejdříve definuje formát a cesta pro vygenerování logovacího souboru (příloha B), ten obsahuje datum, čas a celý průběh skriptu. Je to děláno pro případ neúspěšné analýzy, pro dohledání chyb. Dalším krokem se spustí samotný skript. V první řadě se musíme dostat do složky, kde jsou uložené projekty, které chceme analyzovat. Pomocí příkazů s prefixem hg. dokážeme zkontrolovat nově příchozí changesety. Pokud jsou na serveru opravdu nové changesety, tak pomocí příkazů pull a update stáhneme a aktualizujeme kontrolovaný projekt. Pokud vše proběhlo v pořádku (povedl se pull i update), spustí se

analýza pomocí aplikace SCC a dále pomocí Cppchecku. Oba nástroje na analýzu kódu vygenerují výstupy ve formátu XML, který sonar scanner dokáže rozparsovat a převést do výchozí podoby pro webovou aplikaci SonarQube.

4.3 Statická analýza kódu pomocí vlastního SW nástroje

4.3.1 Škoda Code Checker

Vlastní aplikace pro statickou analýzu kódu psaná v jazyce C# se zaměřením na plnění požadavků normy kódování v ŠT. Jako vývojové prostředí bylo použito MS Visual Studio 2017. Celá aplikace je postavená na využití regulárních výrazů (zkracováno Regex). Regulární výraz je ve své podstatě řetězec popisující větší množinu řetězců. Umožňují efektivně hledat v řetězcích pomocí vlastního vzoru, takzvaného patternu. Ke konstrukci regulárních výrazů se používají tzv. metaznaky. Pří prohledávání textu se pomocí regulárních výrazů testuje shoda metaznaků s prohledávaným textem. Dalším základním kamenem aplikace SCC je generování XML výstupů, které jsou nejdůležitějším prostředníkem mezi aplikací SCC a webovou aplikací SonarQube. XML výstup musí mít přesně daný schéma. Sonar scanner, který parsuje XML výstup má jasně dané RNG schéma (obrázek č. 19).

```
<element name="results" xmlns="http://relaxng.org/ns/structure/1.0">
 <zeroOrMore>
  <element name="error">
   <attribute name="file"/>
   <attribute name="msg"/>
   <attribute name="id"/>
   <attribute name="line">
    <data type="integer" datatypeLibrary="http://www.w3.org/2001/XMLSchema-datatypes" />
   </attribute>
   <text/>
  </element>
 </zeroOrMore>
</element>
 </rule>
```

Obrázek 19 - RNG schéma

Pro tvorbu XML souborů v jazyce C# je knihovna System.Xml.Linq, která obsahuje třídy pro technologie LINQ to XML. To je programovací rozhraní v paměti XML, které nám umožnilo snadno a efektivně pracovat s XML soubory. Ve výsledku funkce pro tvorbu XML v daném RNG schématu vypadá takto:

```
public void Export(IEnumerable<CompleteFile> codeFiles) {
            Console.WriteLine("Exporting to {0}.", XmlReportFile);

try {
        new XDocument(
               new XElement("results",
                     codeFiles
                         .Where(file => file.Bugs.Count > 0)
                         .Select(file => file.Bugs.Select(
                             bug => new XElement("error",
                                    new XAttribute("file",
      file.InFile.FullName.Replace(Environment.CurrentDirectory + @"\", "")),
                                    new XAttribute("msg",  bug.Description),
                                    new XAttribute("line", bug.Line),
                                    new XAttribute("id",   "SCC_" + bug.Type)
                                )
                            ))
                    )
               )
               .Save(XmlReportFile);
            }
            catch(Exception e) {
                Console.WriteLine("Nepovedlo se vytvořit xml report. Podrobnosti:
{0}", e.Message);
            }
```

Výsledný soubor SCC-export.xml:

```
<results>
        <error
           file="projects\109mrp\procs\000_DiagKom\CAN_Diag_Kom.c"
           msg="Výskyt hvězdičky: &quot;  U8 *LifetimeOld, &quot;"
           line="23"
           id="SCC_Asterisk" />
        <error
           file="projects\109mrp\procs\000_DiagKom\CAN_Diag_Kom.c"
           msg="Výskyt hvězdičky: &quot; U8 *ChybnePrubehy, &quot;"
           line="24"
           id="SCC_Asterisk" />
        <error
           file="projects\109mrp\procs\000_DiagKom\MVB_Diag_Kom.c"
           msg="Řádek delší než 150 znaků. Celkem 163 znaků."
           line="332"
           id="SCC_LongLine" />
</results>
```

Z důvodu obsáhlosti reportu, který obsahuje přes tři tisíce varovných hlášení, není možné vše vypsat a proto je tady jen příklad výpisu. Pro zobrazení a přehled všech chyb a varování v projektu je právě využit SonarQube, který si celý report převede pomocí Sonar-scanneru do grafického, přehledného zobrazení ve webové aplikaci.

Další vlastností aplikace SCC je konfigurační soubor, který se vkládá do kontrolovaného projektu. V konfiguračním souboru je možnost nastavit mnohé specifikace analýzy kódu. Například nastavení hledání chyb v projektu, nastavení cest adresářů, ve kterých se kontrolují soubory, přípony souborů, typy vyhledávaných chyb atd. Viz obrázek č. 20.

```
# Nastavení hledání chyb v projektu
# Argumenty zadané při spuštění programu mají vyšší prioritu
# Cesty lze zadat relativně k adresáři projektu nebo absolutně
# Hodnoty zadávat vždy na 1 řádek, oddělovač -> čárka

# === Adresáře, ve kterých se kontrolují soubory ===
# Příklad:
#   Includes = src\APL, src\GEN
Includes = projects\109mrp\procs\

# === Přípony souborů, které jsou zahrnuty do kontroly ===
# Defaultně: cpp, cc, c
# Příklad:
#   Extensions = cpp, cc

# === Typy vyhledávaných chyb ===
# Defaultně: všechny
# Možnosti: MultipleVarAssignment, ...
# Příklad:
#   Search = MultipleVarAssignment, LongLine

# === Export chyb ===
# Defaultně: Console
# Možnosti: Xml, Console, Excel
# Příklad:
#   Export = Xml, Excel
Export=Xml, Pdf

# === Název (adresa) xml reportu ===
# Defaultně: SCC-report.xml
# Příklad:
#   Export-xml = chyby.xml

# === Název (adresa) xml reportu ===
# Defaultně: SCC-report2.xml
# Příklad:
#   Export-xml = chyby.xml

# === Soubor/y s názvy proměnných ===
# Příklad:
#   Var-defs = deklar1.h, src\deklar_special.h
Var-defs = projects\109mrp\deklar.h, projects\109mrp\deklar_NoKeyNoVal.h

# === Maska/y definice proměnné v souborech Var-defs (viz. výše) ===
# Jde o regulérní výraz/y, kde {var} nahrazuje jméno proměnné
# Příklad:
#   Var-def-masks = db_Put{var}\s*\(, CAN_Put_{var}\s*\(
Var-def-masks = \s+(huge|ndata|sdata)\s+{var}\s+; \/\/\s

# === Maska/y přiřazení hodnoty do proměnní ===
# Jde o regulérní výraz/y , kde {var} nahrazuje jméno proměnné
# Defaultně: {var}\s*([\+\-\*\/\%\|\&\^]|>>|<<)?=[^=], (\+\+|\-\-){var}, {var}(\+\+|\-\-)
#            ^^vyhledá klasický zápis do proměnné pomocí =, *= aj. nebo ++/--
# Příklad:
#   Var-assignment-masks = Put\({var}\)
```

Obrázek 20 - SCC konfigurátor

Obrázek 21 - logo aplikace SCC

4.4 Statická analýza kódu pomocí SW nástroje Cppcheck

4.4.1 Cppcheck

Open source nástroj pro statickou analýzu C++ zdrojového kódu. Na rozdíl od kompilátorů C/C++ a mnoha dalších analytických nástrojů neumí Cppcheck detekovat chyby syntaxe. Cppcheck naopak úspěšně detekuje chyby, které kompilátory obvykle nedokáží rozpoznat. Umí kontrolovat například úniky paměti, použití reference pro dealokace, chyby mezí, s cílem hlášení skutečné chyby bez false-positive. Další kontroly, u nichž je známo false-positive hlášení, např. chyby přetečení a indexování pole mimo rozsah, mohou být dodatečně povoleny. Další možností je povolení detekce kontroly podmínek a kontroly nedosažitelného kódu ve formě nepoužitých funkcí.

Jak bylo uvedeno v kapitole č. 7, Cppcheck bude využit jako vhodný doplněk aplikace SCC pro komplexní statickou analýzu kódu.

4.4.2 Automaticky generovaný výstup

Jak už bylo dříve zmíněno, musí mít aplikace možnost XML výstupu v předem uvedeném RNG schématu. Cppcheck umí generovat výstup v XML souboru. V dřívějších verzích Cppcheck měl možnost výběru ve formátu souboru. V nynější verzi se standardizoval formát 2. Formát 2 je výstup, který obsahuje dva základní elementy. <Error> element, který obsahuje id, severity (vážnost chyby), message (krátký popis chyby) a verbose (popis chyby v dlouhém formátu). <Location> element obsahuje, file (cesta k souboru), line (číslo řádku, kde se chyba vyskytuje) a případně ještě info (krátká zpráva o chybě).

```
<?xml version="1.0" encoding="UTF-8"?>
<results version="2">
  <cppcheck version="1.82"/>
  <errors>
    <error
          id="preprocessorErrorDirective"
          severity="error"
          msg="Syntax error in #elif"
          verbose="Syntax error in #elif">
    <location
          file="NIM\LCC\inc\regs_cc770.h"
          line="70"/>
  </error>
  <error
          id="zerodiv"
          severity="error"
          msg="Division by zero."
          verbose="Division by zero.">
    <location
          file="NIM\LCC\projects\109mrp\procs\017_InputValuesCheck\InputValues.c"
          line="23"
          info="Division by zero"/>
    </error>
  </errors>
</results>
```

4.5 Sonar scanner

Sonar scanner je nezbytnou součástí procesu statické analýzy pro SonarQube. Funguje jako defaultní spouštěč pro analýzu projektu. V sonarscanner konfigurátoru se nastavují výchozí parametry, jako například adresa SonarQube serveru, kódování zdrojového kódu a globální nastavení databází. Nezbytnou součástí je konfigurační soubor, který musí být vložen do každého kontrolovaného projektu. Nastavují se konkrétní parametry kontrolovaného projektu.

Zásadní parametry jsou:

- Sonar.projectKey – unikátní klíč, pomocí něhož scanner přiřadí analýzu ke správnému projektu v SQ.
- Sonar.projectName – jméno projektu, které bude zobrazeno ve webové aplikaci SQ.
- Sonar.sources – relativní cesta ke složce se zdrojovými soubory.

Sonar scanner má spousty dalších možností, které nemá smysl rozepisovat, protože v projektu analýzy nejsou využity a s jejich budoucím využitím se nepočítá.

Další funkcí sonar scanneru je zpracování generovaných výstupů z analyzátorů kódu (SCC, Cpp). Oba dva XML soubory rozparsuje a převede na čitelný formát pro webovou aplikaci SonarQube.

4.6 Prezentace výsledků automatické analýzy kódu

Z výše uvedeného přehledu i z praktických zkušeností je zřejmé, že programy pro automatickou analýzu kódu generují obrovské množství chybových a varovných hlášení.
Přitom jen velmi malá část těchto hlášení je skutečně relevantní a vyžaduje provedení opravy zdrojového kódu.

Proto je nezbytné, aby systém statické analýzy maximálně podporoval a usnadňoval přezkoumání výsledků automatických kontrol.
Musí být podporovány minimálně následující funkce:

- Rychlý náhled do zdrojového kódu v místě chyby.
- Možnost skrytí všech výskytů určitého druhu chyby.
- Možnost trvalé kvitace konkrétního výskytu chyby nebo skupiny chyb.
- Report přetříděných výsledků.

4.6.1 SonarQube

SonarQube je open source platforma pro průběžnou kontrolu kvality kódu, která provádí automatické přezkoumání se statickou analýzou kódu pro detekci chyb, code smells a bezpečnostní zranitelnost pro více než 20 programovacích jazyků. Například Java, C#, PHP, JavaScript, C/C++ a mnoho dalších. SonarQube dále nabízí zprávy o duplicitě kódu, kódovací standardy, unit testy, code coverage, komentáře a chyby.

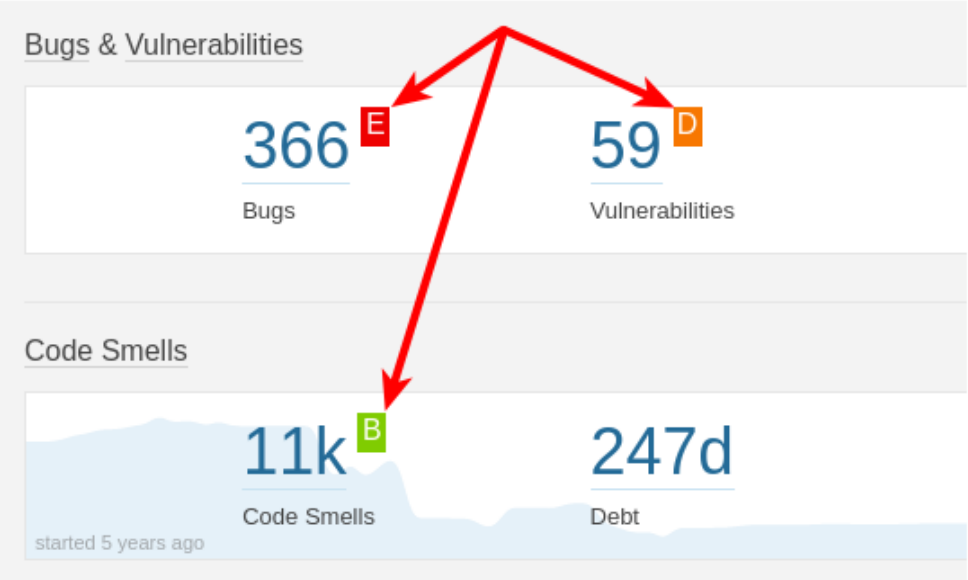
Obrázek 22 - Logo SonarQube

4.6.2 Model kvality SonarQube

Pro zhodnocení a rozlišení toho co je pro nás důležité se v SQ vytvořil model kvality, který využívá to nejlepší ze softwaru SQALE (Obecná metoda vyhodnocení zdrojového kódu). Je založen na následujících základních bodech:

- Model kvality by měl být snadno použitelný.
- Skutečné chyby a bezpečnostní rizika by se neměly ztrácet v množství nepodstatných hlášek a varování.
- Přítomnost závažných chyb nebo zranitelných míst v projektu by měly být vidět.
- Otázka udržitelnosti je stále důležitá a neměla by být ignorována.
- Výpočet nákladů na nápravu je stále důležitý.

Aby se splnila daná kritéria, začaly se rozlišovat problémy spolehlivosti (bugs) a zabezpečení (vulnerabilities) do vlastních kategorií. V další kategorii se upevnilo to, co zůstalo v otázkách udržovatelnosti (code smell). Díky modelu kvality SQALE vznikly tři jednoduché kategorie.



Obrázek 23 - Kategorizace na základě SQALE[1]

- **Software Bugs -** Software Bug (dále už jen SB) je chyba nebo selhání v počítačovém programu či systému. Způsobuje, že může dojít k nesprávnému nebo neočekávanému výsledku nebo se může chovat nezamýšleným způsobem. Proces opravy SB je označován jako ladění (debugging) a často používá formální techniky nebo nástroje k určení chyb. Už od padesátých let byly některé počítačové systémy navržené tak, aby během operací odradily, detekovaly nebo opravovaly různé SB. Nejvíce SB pochází z chyb ve zdrojovém kódu nebo jeho designu. Mohou být také způsobené kompilátory produkující nesprávný kód. Chyby mohou způsobit různé problémy. Jsou SB s jemným efektem, který není tak invazivní jako SB co zmrazí celý systém nebo bezpečnostní SB, které mohou způsobit napadení ze třetí strany. Pro rozdělení a řešení chyb máme tzv. Bug Management, který zahrnuje proces dokumentace, kategorizace, přiřazování, reprodukce, opravy a uvolnění opraveného kódu.

Obrázek 24 - Software bugs

- **Vulnerability** - Vulnerabilita neboli zranitelnost je slabina, která umožňuje útočníkovi omezit zabezpečení informací systému. Zranitelnost je průnikem tří elementů: systémová citlivost, schopnost útočníka zneužít chyby a přístup útočníka k chybě. Za účelem zranitelnosti musí mít útočník alespoň jeden použitelný nástroj nebo techniku, která může využít slabosti systému. V tomto rámci je zranitelnost také známá jako útoková plocha. Řízení zranitelnosti neboli Vulnerability management je cyklickou praxí při identifikaci, klasifikaci a snižování zranitelných míst.

Obrázek 25 – Vulnerabilities

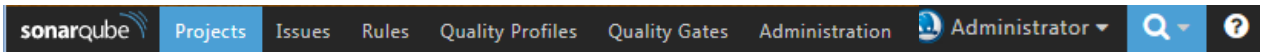
- **Code Smells** - Code Smell (dále už jen CS) ve světě počítačového programování indikuje symptom ve zdrojovém kódu, který většinou značí hlubší problém, neboli podle Martina Fowlera: „Code Smell je povrchová indikace, která obvykle odpovídá hlubšímu problému v systému.“. Další cestou jak se na CS dívat je z pohledu jakosti a kvality. CS jsou určitými strukturami kódu, které naznačují narušení základních zásad návrhu a negativně ovlivňují kvalitu návrhu. CS obvykle nejsou chyby, nejsou technicky nesprávné a v současné době nezabraňují fungování programu. Namísto toho naznačují slabiny v návrhu, které mohou v budoucnu zpomalit vývoj nebo zvýšit riziko chyb nebo selhání. CS mohou být ukazatelem faktorů, které přispívají k tzv. technickému zadlužení (= odráží implicitní náklady na další přepracování způsobené výběrem snadného řešení, a nikoliv použitím lepšího přístupu, který bude trvat déle). Určování toho co je a co není CS je subjektivní, převážně záleží na použitém programovacím jazyku, vývojáři a vývojové metodě, která je použita.

Obrázek 26 - Code Smells

4.6.3 Webová aplikace SonarQube

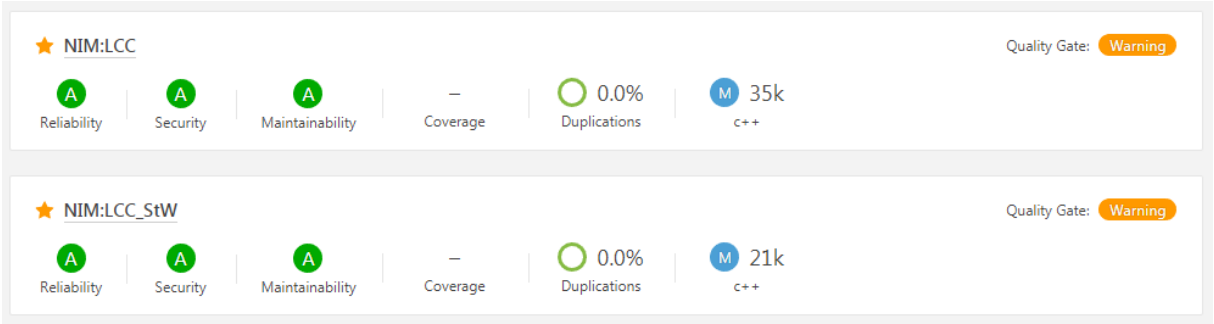
Přehledné grafické rozhraní zobrazující souhrnné výsledky analýz u daných projektů. Web běžící na serveru škoda nebo lokálně na místním počítači. Přístup do aplikace SQ má každý uživatel ŠTRN pomocí svého uživatelského jména a hesla. Ověřování uživatelů je založena na LDAP protokolu.

- **Hlavní menu**



Obrázek 27 - Náhled hlavního menu

- **Projects** – zobrazení všech analyzovaných projektů + obecný souhrn výsledků a vlastností projektů.



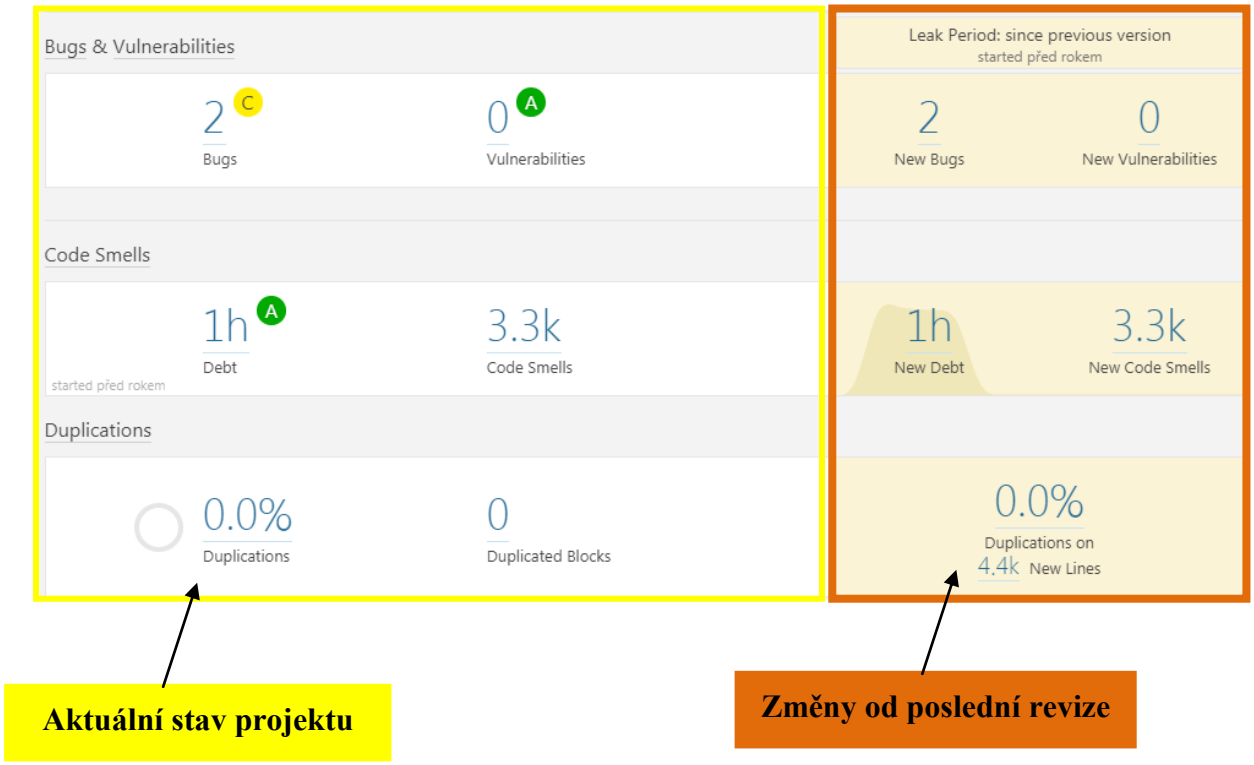
Obrázek 28 - Obecný souhrn výsledků projektů

- **Issues** - Záložka pro zobrazení všech přiřazených issue přihlášeného uživatele. Administrátor má možnost vidět všechna issue u všech projektů.
- **Rules** – Všechna aktivovaná i neaktivovaná pravidla dostupná pro C++ jazyk. Uživatel si může projít pravidla pro případné nesrovnalosti nebo porozumění daného pravidla, které hledá. Administrátor má možnost pravidla přidávat.
- **Quality Profiles** – Definuje naše požadavky definováním sady pravidel. Jde vytvořit různé profily kvality pro jednotlivé projekty. V našem případě využijeme jen jeden vytvořený profil a to SCC, kde máme definovanou sadu našich pravidel pro jazyk C++. Kdykoliv jde přidat a odebrat konkrétní pravidla.
- **Quality Gates** – Quality Gates je nejlepší způsob jak nastavit politiku kvality naší organizace. Je to odpověď na jednu otázku: Mohu dodat svůj projekt dnes nebo ne? Quality gates je definovaný soubor booleovských podmínek založených na prahových hodnotách měření, podle kterých jsou projekty měřeny.
- **Administration** – Základní konfigurace viditelná jen pro přihlášené uživatele jako administrátory. Je zde možné nastavit mnohé konfigurace, například externí

pravidla SCC, emailovou notifikaci, zakládání účtů uživatelů, nastavování pravomocí, management projektů a background tasků. Navíc obsahuje ještě update center, kde je možné aktualizovat používané pluginy, či nainstalovat nové.

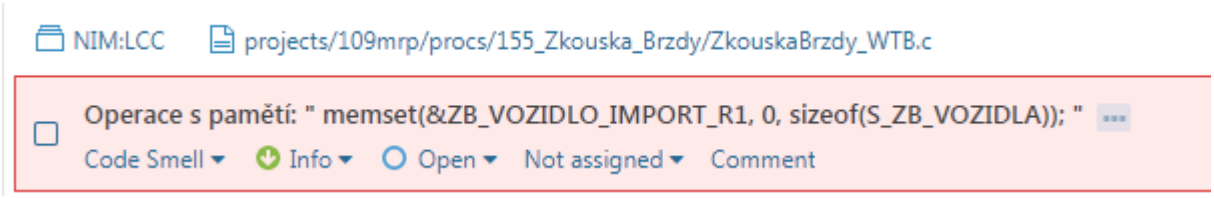
4.6.4 Náhled do zdrojového kódu v místě chyby

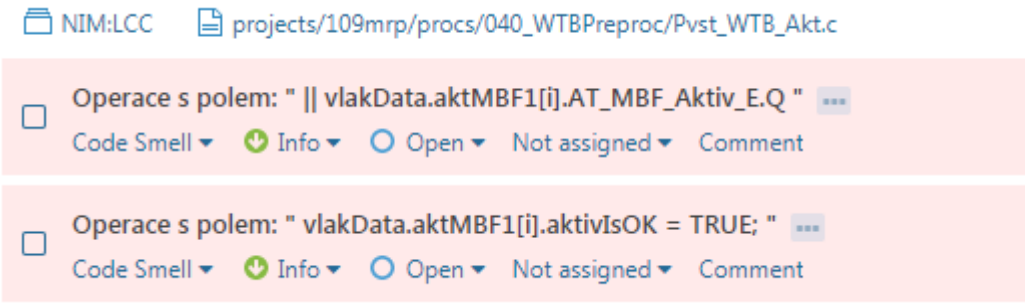
Pro zobrazení chyb si musíme zvolit konkrétní projekt, kde se zobrazí dvě tabulky. První je s aktuálním stavem projektu a druhá zobrazuje změny od poslední revize.



Obrázek 29 - Stav vybraného SW projektu

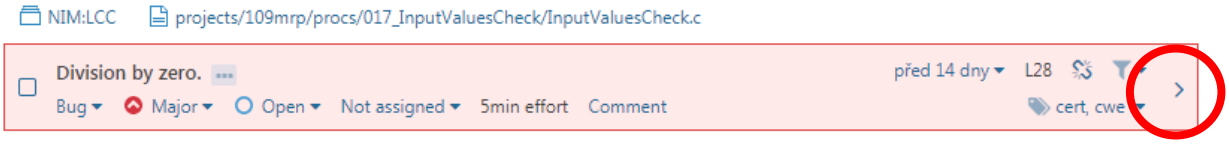
Při bližším zkoumání se dostaneme až na konkrétní chyby, které byly nalezeny ve zkoumaném projektu. Zobrazí se celý seznam nevyřešených výskytů, v našem případě přesně 3340 výskytů.



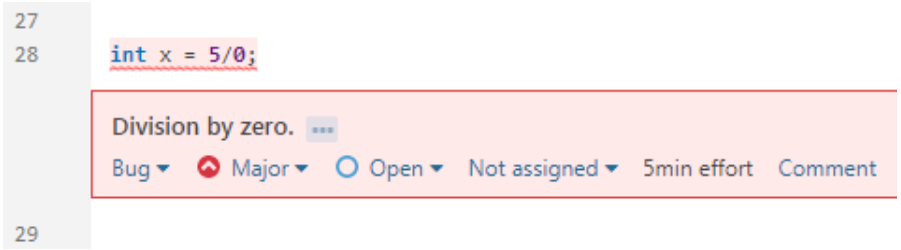


Obrázek 30 - Výpis konkrétních nálezů

Jedna z funkcí je rychlé zobrazení zdrojového kódu v místě chyby. Při vybrání určitého výskytu se verifikátor může velice snadno jedním kliknutím dostat na místo výskytu chyby, či upozornění ve zdrojovém kódu.



Obrázek 31- Ukázka tlačítka na zobrazení umístění výskytu v kódu



Obrázek 32 - Zobrazení výskytu v kódu

Pro verifikátora by takovéto zobrazení znamenalo procházení výskytů v řádech dnů. Pro nalezení a zobrazení skutečně závažných chyb si verifikátor může zvolit libovolný filtr pro zobrazení výskytů určitého druhu.

4.6.5 Možnost skrytí výskytů vybraného druhu – filtry

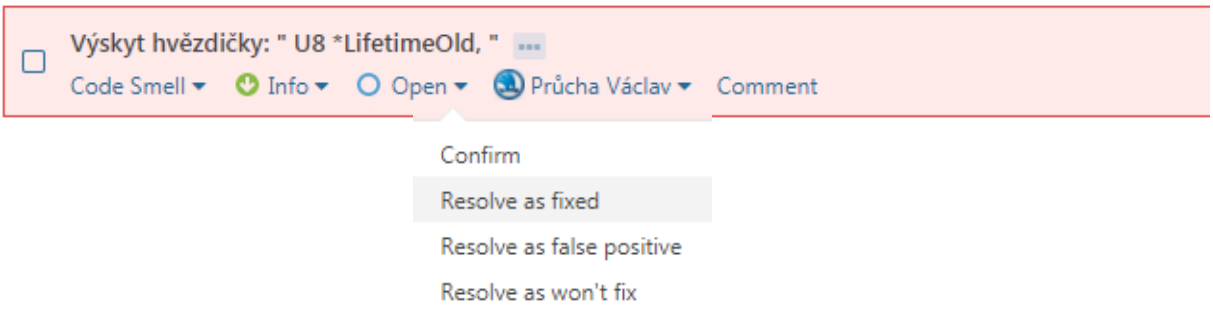
Pro co nejlepší možné skrytí, či zobrazení vybraných druhů výskytů je SonarQube vybaven širokou škálou filtrů. Pro představu si je všechny stručně představíme.

- **Type** – Filtr, který filtruje výskyty dle typu problému.
- **Resolution** – Filtr, který nám umožňuje zobrazit jen nevyřešené, vyřešené, odstraněné, neopravitelné a false positive výskyty nebo jejich kombinace.
- **Severity** – Filtr dle závažnosti výskytu.

- **Status –** Podobný filtr jako resolution, který filtruje výskyty dle jejich stavu.
- **Creation Date –** Filtr pro řazení výskytů dle času nalezení.
- **Rule –** Jeden z nejvýznamnějších filtrů. Řazení výskytů dle pravidel.
- **Tag –** Každé pravidlo lze otagovat, například pravidla aplikace SCC otagovány jako „škoda“.
- **Module –** Filtr pro řazení dle projektů.
- **Directory –** Filtrace dle kontrolovaných složek v adresáři projektu.
- **File –** Filtrace dle kontrolovaných souborů v adresáři projektu.
- **Author –** Filtrace dle autora hlášeného výskytu ve zdrojovém kódu.
- **Language –** Možnost filtrace dle kódovacího jazyku. Výskyt jenom C/C++.

4.6.6 Možnost trvalé kvitace konkrétního výskytu chyby nebo skupiny chyb

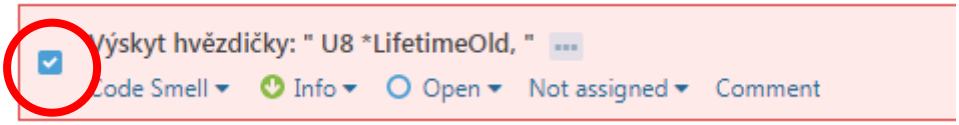
Verifikátor v prostředí SonarQube má možnost kvitovat výskyty chyb postupně po jednom nebo pro celou skupinu. Pokud se verifikátor dostane do stavu, kdy daný výskyt chce uzavřít, má na výběr vícero možností.



Obrázek 33 - Možnosti kvitace výskytu

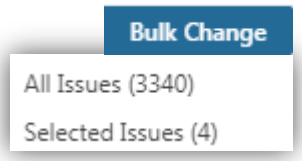
- Confirm – Po přezkoumání issue potvrdíme, že se s tím něco musí udělat.
- Resolve as fixed – Issue bylo opraveno v kódu, ale čeká se na další analýzu, než se uzavře, nebo se znovu otevře po neúplném nebo špatném opravení.
- Resolve as false positive – Tento problém lze ignorovat, protože je způsoben omezením analytického nástroje. Nebude počítáno.
- Resolve as won´t fix – Tento problém lze také ignorovat, protože pravidlo v tomto kontextu není relevantní. Nebude počítáno.

Další možností je skupinová kvitace konkrétního výskytu chyb. Verifikátor má více možností na výběr. Za prvé může postupně označit výskyty a za druhé může využít dříve zmíněných filtrů a uzavřít celou skupinu.



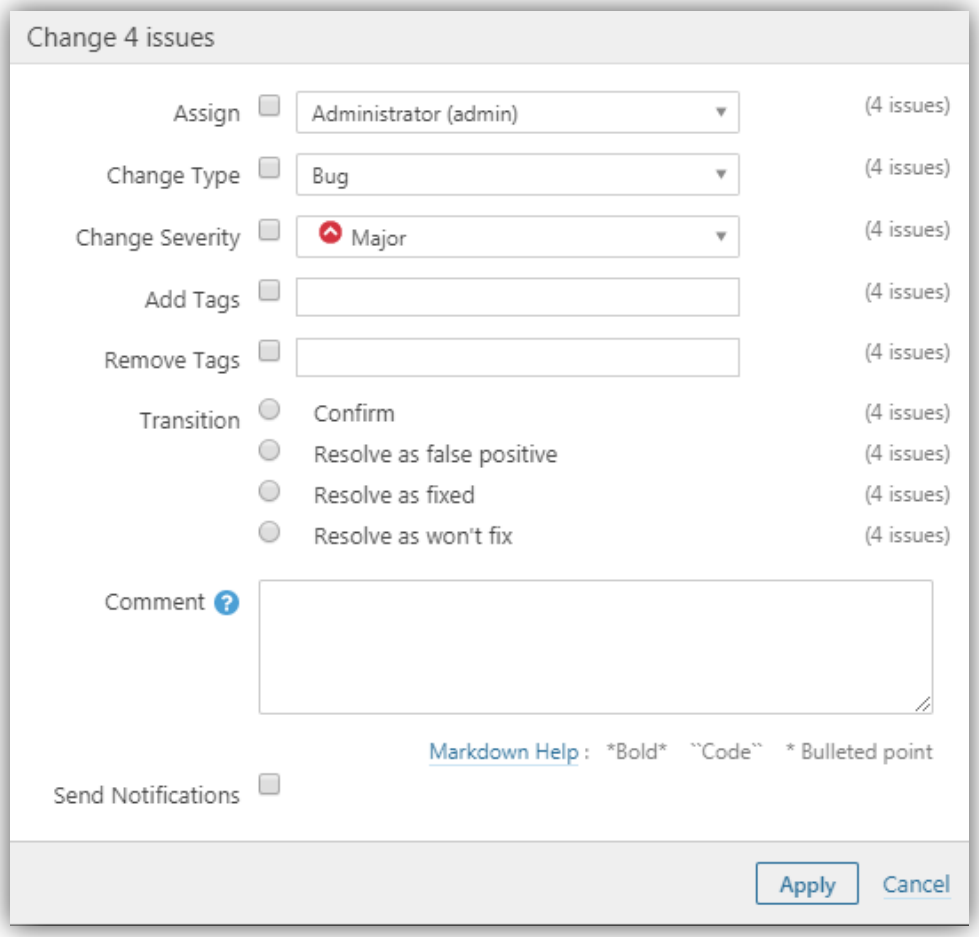
Obrázek 34 - Výběr výskytu

Funkce se jmenuje Bulk Change. Při rozkliknutí nám dává na výběr mezi námi vybranými výskyty nebo celou skupinou.



Obrázek 35 - Funkce Bulk Change

Po zvolení jedné z možností se zobrazí okno s různými řešeními, co můžeme s danými výskyty udělat. Například přiřadit uživateli, změnit závažnost, otagovat, vyřešit nebo okomentovat.



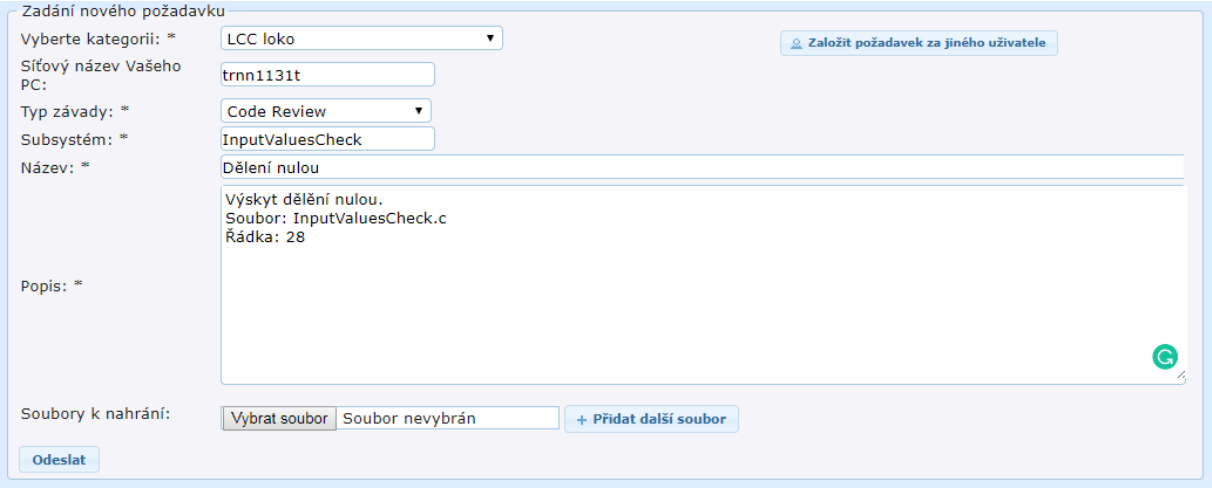
Obrázek 36 - Možnosti funkce Bulk Change

4.6.7 Report přetříděných výsledků

Celkovým výsledkem bude přetříděný projekt s vyznačenými výskyty chyb, které jsou opravdu závažné a s těmi, které jsou bezvýznamné. Výsledným reportem bude PDF soubor příloha C), který obsahuje zprávu o kvalitě projektu s nejdůležitějšími informacemi z webového rozhraní SonarQube. Cílem zprávy je poskytnout souhrnnou informaci o výsledku kontroly SW projektu, která bude podkladem pro řešení nalezených neshod a na kterou se budou odkazovat verifikační dokumenty tvořené v rámci projektové dokumentace SW. Zpráva obsahuje:

- Název projektu.
- Verzi SW projektu.
- Datum generování souboru.
- Celkový počet řádků SW projektu.
- Poměr komentářů k/ke zdrojovému kódu.
- Počet přetříděných výskytů.
- Nejvíce vyskytované chyby.
- Soubory s nejvíce výskyty.
- Nejsložitější třídy.

4.6.8 Řešení nalezených problémů



Obrázek 37 - Založení nového požadavku v IssueTrackeru

Poslední fází je prokazatelné odstranění nalezených problémů. Toho se dosáhne napojením na již fungující proces - ŠKODA IssueTracker. Na základě reportu přetříděných výsledků provede verifikátor v IssueTrackeru zápis požadavku na odstranění nalezené chyby.

Požadavek je verifikátorem adresován na konkrétního autora SW nebo na skupinu autorů.

Důležitá je přesná specifikace nalezeného problému a požadované opravy. Před odesláním požadavku na opravu SW zpravidla ještě verifikátor zkonzultuje s autorem SW nalezený problém a domluví se na vhodném řešení. Autor SW poté provede požadovanou opravu a doplní záznam do IssueTrackeru.

Každý požadavek v IssueTrackeru má své jedinečné číslo (ID). Toto číslo uvede autor SW do popisu příslušného commitu v Mercurialu.

5. Normy kódování SW a kontrola jejich plnění

Definice normy kódování SW dle EN 50 128: „*Normy kódování musí stanovit ověřené programovací zásady, zakázat nebezpečné charakteristiky jazyka a popsat procedury pro dokumentaci zdrojového kódu*“. [2]

V následujících kapitolách jsou popsána jednotlivá pravidla. U každého pravidla je uvedeno, zda je při statické analýze kódu testováno automaticky (Cppcheck, SCC) nebo ručně.

5.1 Normy kódování ve společnosti Škoda Transportation

Prakticky veškerý SW ve ŠT je psaný v jazyce C/C++. Proto i normy kódování vychází z doporučení MISRA, která jsou všeobecně uznávaným standardem pro tento jazyk. Programy psané v ŠT jsou do určité míry specifické a lze je rozdělit na dvě základní skupiny – aplikační a systémové. Aplikační programy jsou z programátorského hlediska jednoduché programy využívající omezenou množinu instrukcí a metod jazyka C/C++. Programy jsou zaměřené na logické zpracování vstupních informací a jednoznačné generování výstupních signálů. Jednoduchost programátorská je vyvážena složitostí technologickou a velkým objemem zdrojového kódu. Systémový SW psaný v ŠT představuje objemově menší část, z hlediska programátorského je však výrazně náročnější. Normy kódování pro SW ŠT jsou zaměřené zejména na tvorbu aplikačního SW. Proto obsahují jen určitou množinu z doporučení MISRA, která je relevantní pro tento druh SW. Norma je naopak doplněná o řadu požadavků formulovaných cíleně pro ŠT. Norma je vydána a udržována ve formě interní směrnice, která je sepsána v souladu s požadavky normy EN 50 128. Účelem této směrnice je stanovení pravidel psaní software ve společnosti ŠT. Odpovědnost za udržování této směrnice v aktuálním stavu, provádění změn a výklad jejího obsahu má na starosti vedoucí oddělení Nadřazené řízení (dále jen NŘ).

5.1.1 Základní pojmy

Pro vysvětlení základní pojmů této směrnice:

- **Verzovací systém**

  Systém sloužící pro ukládání změn, správu změna a porovnávání výsledků změn v daném SW projektu. Typický a aktuálně využívaný je systém Mercurial.

- **Metoda**

  Část programu volatelná z jiných částí programu.

- **Funkce**

  Metoda s vyjádřenou návratovou hodnotou.

- **Procedura**

  Metoda bez návratové hodnoty

- **Makro**

  Definice nahrazovaná na úrovni preprocesoru jazyka C konkrétním obsahem

- **Zdrojový soubor**

  Soubor obsahující zdrojový kód v příslušném programovacím jazyce. Jeho překladem vzniká slinkovatelný a následně i spustitelný nebo zaveditelný kód. Zdrojový soubor obsahuje jeden nebo více SW modulů. Seznam a funkce zdrojových souborů v rámci SW projektu je popsán v architektuře SW dané komponenty

- **Hlavičkový soubor**

  Zvláštní druh zdrojového souboru, který obsahuje deklarace a definice. Hlavičkové soubory se nepřekládají samostatně, ale jako součást programových a zdrojových souborů.

- **Projektový adresář**

  Složka obsahující zdrojové kódy, nastavení vývojového prostředí a překladače, linkovací předpisy, výsledné soubory a případně další SW využívaný při překladu.

- **SW komponenta**

  Nedílná součást SW, která má jasně definované rozhraní a chování s ohledem na architekturu softwaru a jeho návrh. Upřesnění spočívá v definování několika úrovní komponent. Komponenta Level 1 – SW modul (nejmenší samostatně identifikovatelná část SW), Level 2 – SW soubor (zdrojový soubor SW), Level 3 – SW projekt (samostatný SW projekt), Level 4 – Zařízení (samostatně fungující komplet HW a SW).

- **SW projekt**

  Realizuje definovanou skupinu funkcí s využitím jedné nebo více SW komponent (LCC, ACG, ADG atd.). SW projekt se skládá ze zdrojových souborů. Seznam SW projektů je popsán v architektuře SW daného zařízení.

- **SW modul**

  Realizuje jeden požadavek na SW. V případě větší složitosti se SW modul může skládat z metod, funkcí, procedur podle potřeby. Z hlediska struktury programu je třída rovnocenná se SW modulem.

- **Blok příkazů**

  Skupina příkazů nebo podmínek, které spolu logicky nebo funkčně souvisí v rámci SW modulu.

5.1.2 Klasifikace pravidel

Pro každé pravidlo musí být uvedena závaznost pro jeho plnění.

- **Povinná pravidla neboli Mandatory („M“)**
  - Není-li řečeno jinak, jsou příslušné odstavce závazné a normativní.
  - Nově psaný SW musí povinná pravidla splňovat.
- **Velmi doporučená pravidla neboli Highly Recommended („HR“)**
  - Dodržování takto označených pravidel je **velmi doporučováno.**
  - Pokud toto pravidlo není dodrženo, musí být v dokumentu „Návrh SW“ nebo ve zdrojovém kódu uvedeno zdůvodnění.
- **Doporučená pravidla neboli Recommended („R“)**
  - Dodržování pravidel je **doporučeno.**
  - Nižší úroveň doporučení „HR“.
  - Zpravidla mají za cíl postupné sjednocení vizuální stránky SW a ustálení postupů.
  - Doporučená pravidla nesmějí být ignorována.
  - Tato pravidla musí každý účastník vývoje SW znát a v maximální možné míře dodržovat.

5.1.3 Struktura softwarového projektu

- **Založení softwarového projektu ve verzovacím systému**
  - Struktura ukládání dat ve verzovacím systému (Mercurial) musí být popsána v příslušném dokumentu.
- **Ukládání změn, jejich identifikace a verzování**
  - [M] SW musí být v každé etapě svého životního cyklu jednoznačně identifikován, pravidelně zálohován a archivován.
  - [HR] Změny ve struktuře a obsahu zdrojových souborů se pravidelně ukládají s příslušným komentářem do verzovacího systému.
  - [HR] Frekvence ukládání změn do verzovacího systému je alespoň jednou týdně.
  - [R] Frekvence ukládání změn do verzovacího systému je alespoň jednou denně.
  - [HR] SW projekt ukládaný do verzovacího systému musí být aktualizovaný (vzhledem ke změnám ostatních uživatelů), přeložitelný a funkční do té míry, aby umožnil vývoj a testování ostatním uživatelům.

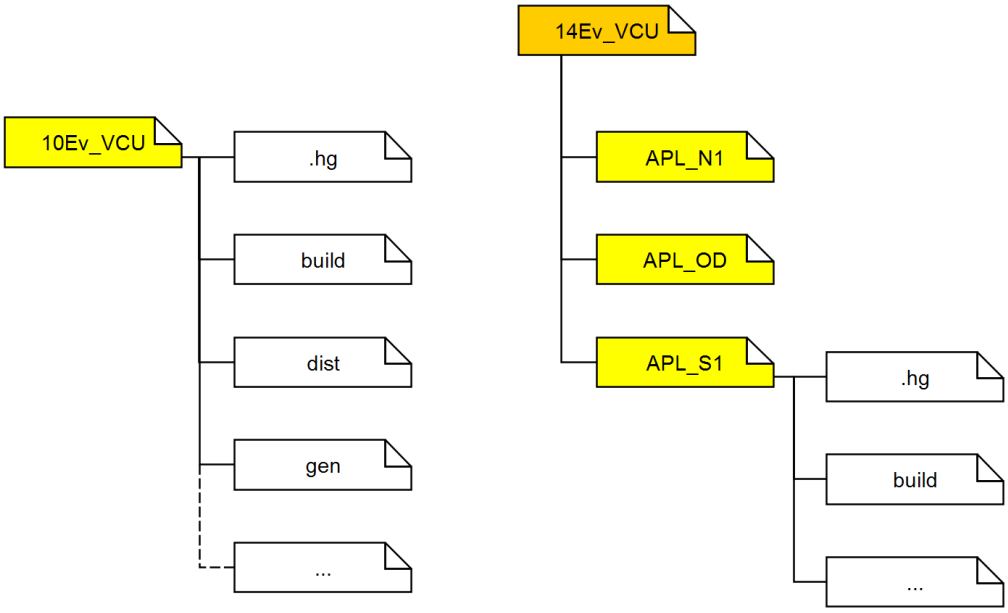
- **Cpp: -**
- **SCC: -**
- **Man:** Kontrola verzí a vnitřních směrnic pro zálohování a archivaci SW a jejich dodržování.

5.1.4 Identifikace požadavků na SW

- [HR] Vývoj SW je prováděn na základě Specifikace požadavků na SW, Specifikace architektury SW, specifikace návrhu SW a specifikace SW komponent. Požadavky na SW jsou identifikovány pomocí řetězce znaků – tzv. ID požadavku, který musí být sledovatelný od Specifikace požadavků na SW až do zdrojového kódu SW. Za tímto účelem je ID požadavku uváděno ve zdrojovém kódu SW před tou částí SW, která daný požadavek realizuje.
  - **Cpp: -**
  - **SCC:** Cílený test na výskyt identifikátorů požadavků na SW.
  - **Man:** Prověření výsledků cíleného testu.

5.1.5 Základní adresářová struktura

- [HR] Názvy adresářů se skládají z maximálně 31 znaků. Pokud se název adresáře skládá z několika částí, používá se k jejich oddělení velké písmeno nebo podtržítko. Mimo písmena anglické abecedy, podtržítka a číslice nejsou žádné jiné znaky povoleny. V systémech založených na OS Linux je povoleno používání pomlček. Konfigurační adresáře projektu smí začínat tečkou (např. .hg, .settings).
  - **Cpp: -**
  - **SCC:** Cílený test na správný formát názvů adresářů.
  - **Man:** Prověření výsledků cíleného testu.
- [R] Pokud projekt zahrnuje více souvisejících aplikací, jsou jejich projektové adresáře umístěny pohromadě v adresáři s názvem obsahujících pojmenování projektu. Jinak je název obsahující pojmenování projektu přímo názvem projektového adresáře.



Obrázek 38 - Příklady struktur adresářů projektu s jedním procesorem/aplikací (vlevo) a více aplikacemi (vpravo)

- **Cpp:** -
- **SCC:** -
- **Man:** Manuální kontrola struktury adresářů.

➢ [R] Každý projektový adresář obsahuje následující základní podadresáře:

Název	Popis
	Adresáře verzovacího systému (např. adresář .hg pro systém Mercurial).
Build	Adresář s mezivýsledky kompilace zdrojových kódů (typicky soubory *.o, *.obj).
Dist	Adresář se soubory určenými k distribuci (typicky *.lib, *.hex apod.).
Gen	Adresář s generátory pro danou aplikaci (typicky generátor TROX, generátor diagnostických hlášení, generátory proměnných (např. TDGú, apod.). Součástí adresáře mohou být také vstupní soubory generátorů (např. *.tdg, *.tsv, *.h).
Lib	Obsahuje adresáře se soubory knihovního/generického charakteru (typicky adresáře CARMEN, ORIS, TROL, GASS nebo soubory *.lib).
Test	Adresář se soubory nastavení testů, testovacím zdrojovým kódem, testovacími projekty a výsledky testů.
Project	Adresář s nastavením projektu (typicky *.ld, oxygen.cfg apod.).
Src	Adresář se zdrojovými soubory (typicky *.c, *.cpp, *.h).

Tabulka 2 - Základní podadresáře v projektovém adresáři

- **Cpp:** -
- **SCC:** -
- **Man:** Manuální kontrola struktury adresářů.

5.1.6 Zdrojové soubory v jazyce C/C++

- [HR] Názvy souborů se skládají z maximálně 31 znaků. Pokud se název souboru skládá z několika částí, používá se k jejich oddělení velké písmeno nebo podtržítko. Mimo písmena anglické abecedy, podtržítka a číslice nejsou žádné jiné znaky povoleny. V systémech založených na OS Linux je povoleno používání pomlček. Konfigurační soubory projektu smí začínat tečkou (např. .cproject, .hgsub).
  - **Cpp:** -
  - **SCC:** Cílený test na správný formát názvů souborů.
  - **Man:** Prověření výsledků cíleného testu.
- [R] Zdrojový kód je dělen do souborů s následujícími příponami:

Přípona	Popis
***.c**	Zdrojový soubor programu v jazyce „C“
***.cpp** ***.cc**	Zdrojový soubor programu v jazyce „C++“
***.h**	Hlavičkový soubor pro „C/C++“, soubor s definicemi proměnných a symbolických konstant, deklaracemi tříd, procedur a funkcí.
***.hpp** ***.hh**	Hlavičkový soubor pro „C++“, používá se v případě, kdy má překladač problém s rozlišením souborů v jazyce C a C++.
***.lib**	Soubor knihovny s předkompilovanými zdrojovými soubory
	Další přípony specifické pro konkrétní vývojové prostředí

Tabulka 3 - Přípony zdrojových souborů

  - **Cpp:** -
  - **SCC:** Cílený test na správný formát názvů souborů v souvislosti s danou platformou.
  - **Man:** Prověření výsledků cíleného testu.

5.2 Pravidla pro vytváření zdrojového kódu

5.2.1 Obecná pravidla

- [M] SW musí plnit požadované funkce
  - **Cpp:** Nalezení a odstranění programátorských chyb a podezřelých částí kódu.
  - **SCC:** Cílené testy na zpracování všech vstupních a výstupních signálů a spouštění všech definovaných funkcí.
  - **Man:** Prověření automaticky generovaných nálezů.
- [M] SW nesmí negativně ovlivnit jinou část SW ani činnost SW jako celku.

  Příklady negativního působení SW:
  - Zápis do oblasti paměti, která tomuto SW nepřísluší.
  - Zacyklení programu.
  - Nepřiměřená časová náročnost programu.

- **Cpp:** Nalezení a odstranění programátorských chyb a podezřelých částí kódu.
- **SCC:** Nalezení potenciálně nebezpečných operací (dělení, práce s pointery a poli, programové smyčky, …).
- **Man:** Prověření automaticky generovaných nálezů.

- [HR] Zdrojový kód musí být přehledný a srozumitelný. Musí obsahovat dostatečné množství komentářů.
  - **Cpp: -**
  - **SCC:** Cílený test na poměr délky čistého kódu a kódu s komentáři.
  - **Man:** Prohlédnutí a posouzení kódu.
- [HR] SW moduly musí být možno otestovat a to jak v laboratorních podmínkách, tak i po nasazení v cílovém systému.
  - **Cpp**: **-**
  - **SCC**: Cílený test na přístup k proměnným přes makro „get_“, které umožňuje podkládání proměnných.
  - **Man**: Prověření výsledků cíleného testu.
- [HR] Pro každý SW modul musí být zajištěna sledovatelnost v celém životním cyklu SW (uvedením identifikačního čísla požadavku v hlavičce SW modulu).
  - **Cpp: -**
  - **SCC:** Cílený test na výskyt identifikátorů požadavků na SW.
  - **Man:** Prověření výsledků cíleného testu.
- [R] Maximální délka řádků, včetně komentářů, je stanovena na 150 znaků.
  - **Cpp: -**
  - **SCC:** Cílený test na maximální délku řádků.
  - **Man:** Prověření výsledků cíleného testu.
- [R] Maximální počet řádků zdrojového souboru nesmí při základním návrhu SW překročit 500 řádků. V průběhu údržby nebo dovývoje může být postupně zvýšen až na 1000 řádků. Hodnota je uvedena včetně záhlaví, komentářů a prázdných řádků. Výjimkou z tohoto pravidla jsou soubory, které obsahují nějakou databázi, typicky to bývají generované soubory. Další výjimku tvoří všechny zcela nebo částečně generované soubory a soubory jediné třídy, kdy by bylo komplikované a nepřehledné umělé dělení do dílčích souborů.
  - **Cpp: -**
  - **SCC:** Cílený test na maximální počet řádků souboru.

- **Man:** Prověření výsledků cíleného testu.

- [R] Maximální počet řádků SW modulu nesmí při základním návrhu SW překročit 150 řádků. V průběhu údržby nebo dalšího vývoje může být postupně zvýšen až na 300 řádků. Hodnota je uvedena bez záhlaví SW modulu.
  - **Cpp:** -
  - **SCC:** Cílený test na maximální délku řádků SW modulu.
  - **Man:** Prověření výsledků cíleného testu.
- [HR] Zdrojový kód a komentáře se nesmí odsazovat tabelátory. Doporučeno je nastavit vývojové prostředí tak, aby na místo tabelátorů vkládalo 2 mezerníky.
  - **Cpp:** -
  - **SCC:** Cílený test na výskyt tabelátorů.
  - **Man:** Prověření výsledků cíleného testu.
- [HR] Je zakázáno používat v program tzv. "kouzelná čísla". Číselné konstanty musí být definovány ve formě symbolických konstant nebo konstant a to vždy s uvedením výstižného a jednoznačného komentáře. Výjimkou jsou čísla, u nichž je význam zcela jednoznačně určen jejich hodnotu a obecnými zvyklostmi (např. 0, 1, 0xFFFF, 0xF00, ">>8" apod.).
  - **Cpp:** -
  - **SCC:** Cílený test na výskyt „kouzelných čísel" v SW.
  - **Man:** Prověření výsledků cíleného testu.
- [R] Prázdné řádky se vkládají podle vlastního uvážení kdekoliv, kde mohou zlepšit čitelnost kódu.
  - **Cpp:** -
  - **SCC:** -
  - **Man:** Manuální posouzení čitelnosti a přehlednosti SW.
- [HR] Žádná část zdrojového kódu nesmí být trvale zakomentování. Pokud je nutno určitou část kódu dočasně zakomentovat (typicky nedokončené nebo chybné části kódu), označí se dané místo znaky "TODO" a v komentáři se uvede důvod zakomentování této části kódu. Nejpozději před verifikací/validací, je nutno zakomentování části kódu dokončit, opravit nebo vymazat.

```
static void SW_Ochrany_Stridacu_U21U22_M2 (void)
{  B_M2_U21_OCH = (                        // Vyhodnocení podmínek funkce trakčního
střídače U21 M2
         get_BX_M2_UD21_MAX                // přepětí filtru
    //|| get_BX_M2_UD21_MIN)               // podpětí filtru  TODO JL nutno ověřit podmínku
      || get_BX_M2_I21SW_MAX)              // nadproud střídače
      || get_BX_M2_ITRDCSW_MAX             // Nadproud troleje
      || get_BX_M2_ITRACSW_MAX);           // Nadproud troleje AC
}
```

- **Cpp:** -
- **SCC:** Cílený test na výskyt zdrojového kódu v komentáři.
- **Man:** Manuální kontrola, zejména v etapě údržby. Prověření výsledků cíleného testu.

➢ [HR] Klíčové slovo **extern** smí být použito pouze ve hlavičkových souborech.

- **Cpp:** -
- **SCC:** Cílený test na výskyt příkazu extern mimo hlavičkové soubory.
- **Man:** Prověření výsledků cíleného testu.

➢ [HR] Musí být zřejmé a snadno kontrolovatelné, že všechny proměnné mají před prvním čtením definovanou hodnotu (např. Generovaná inicializace všech globálních proměnných).

- **Cpp:** Kontrola u lokálních proměnných.
- **SCC:** -
- **Man:** Manuální kontrola hromadné inicializace globálních proměnných.

➢ [HR] Při přiřazování dat ze širšího typu do menšího typu, se nesmí spoléhat na implicitní typovou konverzi. V takových případech je nutno provést explicitní typovou konverzi, aby byl zřejmý záměr programátora.

```
/**
* Příklad implicitní a explicitní konverze
*/
public static void main(){
  byte i;  // 1-Bytová proměnná
  word j;  // 2-Bytová proměnná

  i = j;         // Chybný zápis (implicitní konverze)
  i = (byte)j;   // Správný zápis (explicitní konverze)

}
```

- **Cpp:** Kontrola typových konverzí.
- **SCC:** -
- **Man:** Prověření výsledků Cpp, manuální kontrola kódu.

➢ [HR] Pro tvorbu nových typů, proměnných typu struktura, se musí vždy použít deklarace typu pomocí **typedef**.

```
typedef struct _TyNode {
  long lData;
  struct _TyNode *ptyNext;
} TyNode, *pTyNode;
```

- **Cpp:** -

- **SCC:** -
- **Man:** Manuální kontrola kódu.

➢ [R] Struktury je třeba definovat tak, aby nedocházelo k implicitnímu zarovnávání překladačem a to zvláště u struktur určených pro komunikaci. Pro dorovnání slova je nutno datovou strukturu doplnit pomocí vloženého prvku příslušné velikosti.

```
typedef struct can_obj {
  U32   msg_ctl;      // Message Control
  U32   arbitr;       // Arbitration
  U32   msg_cfg;      // Message Configuration
  U8    msg_data[7]; // Message Data 0 .. 7 byte
  U8    dummy;        // Reserved Byte
};
```

- **Cpp:** -
- **SCC:** -
- **Man:** Manuální kontrola kódu.

➢ [HR] Dynamická alokace paměti není povolena.

Případné výjimky vyžadují explicitní kladné vyjádření verifikátora!

- **Cpp:** -
- **SCC:** Cílená kontrola na použití příkazu malloc(), free(), calloc(), realloc() a new[].
- **Man:** Manuální kontrola kódu, prověření výsledků SCC.

5.2.2 Popisy a komentáře

➢ [HR] Pokud to dané vývojové prostředí umožňuje, označují se jednořádkové komentáře znakem "//". V opačném případě je povoleno používat blokové ohraničení komentářů.

- **Cpp:** -
- **SCC:** Cílená kontrola na použití blokových komentářů na jedné řádce.
- **Man:** Manuální kontrola kódu, prověření výsledků cílené kontroly.

➢ [HR] Uvnitř víceřádkového komentáře se nesmí vyskytovat sekvence znaků "/*".

- **Cpp:** -
- **SCC:** Cílená kontrola na použití sekvence „/*“ uvnitř blokových komentářů.
- **Man:** Prověření výsledků cílené kontroly.

➢ [R] Komentáře se píší přednostně v češtině bez diakritiky. Přípustné jsou komentáře v češtině s diakritikou nebo v angličtině.

- **Cpp:** -
- **SCC:** -

- **Man:** Manuální kontrola.

- [HR] V komentářích se nesmí používat hanlivé nebo jinak nevhodné výrazy.
  - **Cpp:** -
  - **SCC:** -
  - **Man:** Manuální kontrola.
- [R] Jednořádkové komentáře umístěné na řádku společně s kódem musí být zarovnány s ohledem na přehlednost. Pokud se nevejdou na jeden řádek, začíná další část se stejným odsazením.

```
for (long lI = 0; lI < DUMMY_CONST; lI++){
  DummyFunction();                  // together with the source code
                                    // this did not fit the first line
}
```

  - **Cpp:** -
  - **SCC:** -
  - **Man:** Manuální kontrola.
- [HR] Pro záhlaví zdrojových souborů a popisy modulů se používán formátování a příkazy doxygen.
  - **Cpp:** -
  - **SCC:** Cílená kontrola na výskyt řídicích řetězců oxygen.
  - **Man:** Manuální kontrola, prověření výsledků cílené kontroly.
- [HR] Všechna čísla požadavků na SW jsou uvedena za příkazem **@requirement**.
  - **Cpp:** -
  - **SCC:** Cílená kontrola na výskyt příkazů **@requirement**.
  - **Man:** Manuální kontrola, prověření výsledků cílené kontroly.
- [HR] Všechny parametry procedur a funkcí musí být popsány pomocí příkazu **@param**.
  - **Cpp:** -
  - **SCC:** Cílená kontrola na výskyt příkazů **@param**.
  - **Man:** Manuální kontrola, prověření výsledků cílené kontroly.
- [HR] Návratová hodnota funkce musí být popsána pomocí příkazu **@return**.
  - **Cpp:** -
  - **SCC:** Cílená kontrola na výskyt příkazů **@return**.
  - **Man:** Manuální kontrola, prověření výsledků cílené kontroly.
- [HR] Na začátku každého zdrojového souboru je umístěno záhlaví v níže uvedeném formátu. Obdobný popis se umístí i do hlavičkového souboru.

```
/**
* @file
* @brief                Modul zajistujici zpracovani signalu z/pro LIM
* @par "Archive:"       109E3 LCC
* @par "Copyright:"     Skoda Transportation, a.s.
* @author               Jmeno autora
* @par "Created:"       DD.MM.YYYY
*/
```

- **Cpp:** -
- **SCC:** Cílená kontrola na výskyt záhlaví ve zdrojovém souboru.
- **Man:** Manuální kontrola, prověření výsledků cílené kontroly.

➢ [HR] Generované zdrojové soubory mají odlišný formát záhlaví. Obsahuje informaci, že se jedná o generovaný kód, identifikaci makra, datum a čas generování souboru.

```
/**
* @file
* @brief                Automaticky (VBA makrem) vygenerovany kod
* @brief                Modul plneni portu disleje v ridicim pocitaci
* @par "Archive:"       109E3 LCC
* @par "Copyright:"     Skoda Transportation, a.s.
* @author               Vesely
* @par "Makro:"         VBA makro „Generuj_zdrojove_kody_LCC" ver. 1.001.23
* @par "Created:"       22.9.2014, 9:19:04
*/
```

- **Cpp:** -
- **SCC:** Cílená kontrola na výskyt záhlaví ve zdrojovém souboru.
- **Man:** Manuální kontrola, prověření výsledků cílené kontroly.

➢ [HR] Záhlaví SW modulu obsahuje stručný popis funkce a číslo požadavku.

```
/* ------------------------------------------------------------ */
/**
 * @brief FOLOWER OK
 *
 * Souhrnna informace o tom, že řízená lokomotiva je připravena poslouchat povely z
řídící lokomotivy. \n
 * Pro nahlášení připravenosti musí být splněny následující podmínky: \n
 * a)       WTB je inagurována a gateway komunikuje správnì, \n
 * b)       na lokomotivě je nastaven režim řízená lokomotiva přepínaèem S137, \n
 * c)       byla úspěšně ověřena kompatibilita SW,  \n
 * d)       není porucha řídícího počítače, \n
 * e)       není porucha CMOS; \n
 *
 * @requirement L5500-4.1
 *
 */
static void Folower_OK (void){
```

- **Cpp:** -
- **SCC:** Cílená kontrola na výskyt záhlaví ve zdrojovém souboru.
- **Man:** Manuální kontrola, prověření výsledků cílené kontroly.

➢ [HR] Popisy proměnných musejí být na místě, které lze v kódu jednoduše nalézt (pomocí názvu proměnné). Doporučeno je uvést popis u deklarace v hlavičkovém souboru nebo u definice, pokud proměnná nemá samostatnou deklaraci (např. Lokální proměnná). Výjimkou mohou být proměnné, které mají v místě definice jednoznačně určený účel (např. iterační proměnné cyklů).

- **Cpp:** -
- **SCC:** -
- **Man:** Manuální kontrola.

➢ [HR] U definice každé proměnné musí být uveden v komentáři její význam

```
byte   ndata  BI_BSAC_S1_Y;    // Napětí na segmentu SAC1 napájení HVAC
```

- **Cpp:** -
- **SCC:** -
- **Man:** Manuální kontrola.

➢ [HR] U definice každé vícehodnotové proměnné musí být uveden v komentáři význam jednotlivých hodnot. Komentář může mít vice řádků.

```
byte   ndata  DI_Q02_ACHV;    // Stav hlavního vypínače AC   0 - mezipoloha,   1 - vypnuto,
                              // Stav hlavního vypínače AC   2 - zapnuto,    255 - porucha

nebo

byte   ndata  DI_Q02_ACHV;    // Stav hlavního vypínače AC

// Stavy hlavního vypínače AC (DI_Q02_ACHV)
ACHV_MEZI        0;  // mezipoloha
ACHV_VYP         1;  // vypnuto
ACHV_ZAP         2;  // zapnuto
ACHV_ERR       255;  // porucha
```

- **Cpp:** -
- **SCC:** -
- **Man:** Manuální kontrola.

➢ [HR] V případě proměnných nebo konstant, které reprezentují fyzikální veličinu, musí být v komentáři uveden rozměr a fyzikálně přípustné hodnoty.

```
word   ndata  AI_B108_THV;     // Tlak hlavního vzduchojemu [mbar] 0-10000
```

- **Cpp:** -
- **SCC:** -
- **Man:** Manuální kontrola.

5.2.3 Formátování zdrojového kódu

➢ [R] Odsazení začátku řádku odpovídá hloubce vnoření. S každým dalším vnořením se odsazení o minimálně dva znaky zvětšuje. V jednom souboru je vždy používáno stejné odsazení.

```
  // VLASTNI TEST
  if(AT_LIM_SlfTst > 0){
    // Prvni krok - testovani akualne nastavene zeme
    switch (get_LIM_STEP){
      case 1:                                 // Narodni volba pro prvni krok testu se meni v zavislosti na aktualnivolbe
        D_LIM_SlfTst_RQ = get_DY_DSP_NV;       // Nastavi aktualni zemi pro test LIM
        D_LIM_TEST = Testuj_zemi(get_D_LIM_SlfTst_RQ);
        if (get_D_LIM_TEST == get_D_LIM_SlfTst_RQ){
          LIM_STEP = DK_NV_PL;                 // Nastavi dalsi krok
        }
        break;
```

- **Cpp:** -
- **SCC:** -
- **Man:** Manuální kontrola.

➢ [HR] Pokud příkaz pokračuje na další řádce, je jeho pokračování zarovnáno v logické návaznosti na začátek výrazu.

- **Cpp:** -
- **SCC:** -
- **Man:** Manuální kontrola.

➢ [R] Na jedné řádce je umístěna pouze jedna deklarace nebo příkaz. Výjimku tvoří příkazy, které spolu logicky souvisí (např. podmínka její přemostění).

```
BTF_R2E_BNS_SET =                  // Nouzove vypnuti foloweru
  ( ( get_B_EMERG_STOP_Y           // Použití záchranné brzdy
      ||
      get_B_TLAC_STOP_Y            // Použití tlačítka centrální stop (žluté tlačítko)
      ||
      get_B_X01_ADD_OCH            // Zasah ADD sberace X01
      ||
      get_B_X02_ADD_OCH            // Zasah ADD sberace X02
      ||
     (get_BX_M1_ITV_MAX && !get_BY_M1_MSK_V)  // Nadproud topeni vlaku z neodpojene
skrine
      ||
     (get BX M2 ITV MAX && !get BY M2 MSK V)  // Nadproud topeni vlaku z neodpojene
skrine
      ||
      get_B_ITVMAX_SYST_OCH        // Neni zasah ochrany topeni vlaku
    )
    &&
    get B WTBI NOUZSTOP EN         // Povoleni nouzove vypnuti
  );
```

- **Cpp:** -
- **SCC:** -
- **Man:** Manuální kontrola.

➢ [HR] Direktivy preprocesoru začínají na začátku řádku.

```
// zarovnani struktur na bajty
#pragma pack(1)
// velikost portu v bajtech
#define VELIKOST_PORTU   32
```

- **Cpp:** -
- **SCC:** -
- **Man:** Manuální kontrola.

➢ [HR] Začátek složené závorky se u příkazů “if”, “else”, “for”, “while”, “do”, “switch”, umístění na konci řádku s příkazem.

```
if (get_BY_DSP_LIM_TEST_EN) {                    // Povoleni testovani LIMu
  AT_LIM_SlfTst = AK_LIM_ST_TO;                  // Nastaveni timeoutu selftestu
cca 5min
  LIM_STEP = 1;                                  // Nastavi prvni krok testu
}
```

- **Cpp:** -
- **SCC:** -

- **Man:** Manuální kontrola.

➢ [HR] Složené závorky se u příkazů “if”, “else”, “for”, “while”, “do”, “switch”, píší vždy. Tam, kde to přispěje k přehlednosti a čitelnosti kódu, je přípustná variant krátký jednořádkový podmíněný příkaz. V takovém případě není použití složených závorek vyžadováno.

```
if (AT_LIM_STS > 0) AT_LIM_STS--;                    // Dekrementace casovace
```

- **Cpp:** -
- **SCC:** -
- **Man:** Manuální kontrola.

➢ [R] Pro zvýšení čitelnosti výrazů (např. pro správnou interpretaci priority operátorů) se ve vhodné míře používají závorky ().

```
// Příklad nevhodné konstrukce:
if (aaa && bbb || ccc && ddd) …
// Příklad správného provedení:
if ( (aaa && bbb)
      ||
     (ccc && ddd))
…
```

- **Cpp:** -
- **SCC:** -
- **Man:** Manuální kontrola.

➢ [R] Konstrukce výrazů se, pokud možno, vyhýbat řetězení negací. Ztěžuje pochopení výsledku výrazu.

```
  B_M2_U21BLOK_OCH =                                    // Zablokování trakčního
střídače U21 M2
    !((!get_BX_M2_UD21_MAX)                             // přepětí filtru
    &&(!get_BX_M2_UD21_MIN)                             // podpětí filtru
    &&(!get_BX_M2_I21SW_MAX)                            // nadproud střídače
    &&(!get_BX_M2_ITRDCSW_MAX)                          // Nadproud troleje
    &&(!get_BX_M2_ITRACSW_MAX))                         // Nadproud troleje AC
```

- **Cpp:** -
- **SCC:** -
- **Man:** Manuální kontrola.

➢ [HR] Při použití makra s parametry, se jednotlivé argument a elementární výrazy uzavírají do kulatých závorek.

```
#define mabs(wX)              (((wX) & 0x8000) ? -(wX) : (wX))
```

- **Cpp:** -
- **SCC:** -
- **Man:** Manuální kontrola.

5.2.4 Řídicí struktury, tok programu

➢ [HR] Výrazy definované v plovoucí řádové čárce nesmí být testovány na rovnost či nerovnost. Výjimkou může být, pokud do nich programátor úmyslně vložil hodnotu v nějaké podmínce, na kterou pak proměnnou testuje.

- **Cpp:** -
- **SCC:** Cílený test na kontrolu testování rovnosti či nerovnosti výrazu definovaného v plovoucí řádové čárce.
- **Man:** Manuální kontrola.

- [HR] Řídicí výraz cyklu "for" nesmí obsahovat objekt definovaný v plovoucí řádové čárce.
  - **Cpp:** -
  - **SCC:** Cílený test na kontrolu použití plovoucí řádky v cyklu for.
  - **Man:** Manuální kontrola.
- [HR] Řídicí proměnná cyklu "for" nesmí být v těle cyklu modifikována.
  - **Cpp:** Nalezení modifikace řídící proměnné v cyklu for.
  - **SCC:** -
  - **Man:** Manuální kontrola výsledků Cpp.
- [R] Nesmí být použity výrazy logického typu, které jsou vždy "true" nebo vždy „false", pokud to není vynuceno požadovanou funkcí programu.
  - **Cpp:** Nalezení výrazů logického typu, které jsou vždy „true" nebo „false".
  - **SCC:** -
  - **Man:** Manuální kontrola výsledků Cpp.
- [HR] Nesmí se vyskytovat nedosažitelný kód.
  - **Cpp:** Nalezení nedosažitelných částí kódu.
  - **SCC:** -
  - **Man:** Manuální kontrola výsledků Cpp.
- [HR] Příkazy "goto" se nesmí používat.
  - **Cpp:** -
  - **SCC:** Cílený test na výskyt příkazů goto.
  - **Man:** Prověření výsledků cíleného testu.
- [R] Konstrukce if…else if musí být vždy zakončena částí else, i kdyby za ní následoval pouze prázdný složený příkaz.
  - **Cpp:** -
  - **SCC:** Cílený test na kontrolu konstrukce if-else.
  - **Man:** Prověření výsledků cíleného testu.
- [HR] Každá sekce příkazu "switch", která obsahuje alespoň jeden příkaz, musí být ukončena příkazem "break" nebo v odůvodněných případech příkazem "return". Pokud není break ani return použit úmyslně, je třeba toto řádně v kódu

okomentovat.

- **Cpp:** -
- **SCC:** Cílený test na kontrolu konstrukce příkazu switch.
- **Man:** Prověření výsledků cíleného testu.

- [R] Každý příkaz "switch" musí být zakončen sekcí "default" (třeba I prázdnou) nebo musí být v komentáři popsáno chování program v případě, že není splněna žádná z podmínek.
  - **Cpp:** -
  - **SCC:** Cílený test na kontrolu konstrukce příkazu switch a zakončení sekcí default.
  - **Man:** Prověření výsledků cíleného testu.

5.2.5 Funkce

- [HR] Funkční prototypy musí být úplné, tzn. včetně argument a jejich typů a typu návratové hodnoty.
  - **Cpp:** Nalezení a odstranění programátorských chyb a podezřelých částí kódu.
  - **SCC:** -
  - **Man:** Prověření automaticky generovaných nálezů.
- [R] Funkce nesmí být definovány s proměnným počtem argumentů. Případné výjimky musí být odůvodněny v komentářích kódu.
  - **Cpp:** -
  - **SCC:** -
  - **Man:** Manuální kontrola.
- [R] Rekurzivní volání funkcí není povoleno. Případné výjimky musí být explicitně odůvodněny v komentářích.
  - **Cpp:** Nalezení rekurzivního volání funkcí.
  - **SCC:** -
  - **Man:** Manuální kontrola automaticky generovaných výsledků Cpp.
- [R] Metody bez parametrů, resp. nevracející žádnou hodnotu, jsou deklarovány a definovány s parametrem void, resp. návratovou hodnotou void.
  - **Cpp:** -
  - **SCC:** -
  - **Man:** Manuální kontrola.

5.2.6 Pole, struktury a ukazatele

- [HR] Aritmetické operace s ukazateli smí být prováděny pouze v případě, že tyto ukazatele adresují pole nebo ukazují na prvky polí.
  - **Cpp:** Nalezení aritmetických operací s ukazateli.
  - **SCC:** -
  - **Man:** Prověření automaticky generovaných výsledků.
- [HR] Matematické operace „rozdíl dvou ukazatelů“ a „porovnání dvou ukazatelů“ mohou být aplikovány, pouze pokud oba ukazatele ukazují na prvky téhož pole nebo struktury.
  - **Cpp:** Nalezení matematických operací rozdíl a porovnání dvou ukazatelů.
  - **SCC:** -
  - **Man:** Prověření automaticky generovaných výsledků.
- [HR] Při zápisu do paměti a čtení z paměti pomocí indexace v rámci pole musí být zaručeno, že index nepřekročí meze pole.
  - **Cpp:** Nalezení výskytů překročení mezí pole.
  - **SCC:** -
  - **Man:** Prověření automaticky generovaných výsledků.
- [HR] Při použití funkcí memset či memcpy musí být zaručeno, že nedojde k přepsání paměti mimo požadovanou oblast. Určení velikosti paměti k přepsání se provádí pomocí funkce “sizeof” nad cílovým polem / strukturou (nebo jejím typem).

```
memcpy(&RealTimeSet_T, &datum_cas_write, sizeof(RealTimeSet_T));
memcpy(&datum_cas_read, &RealTimeAct_T,  sizeof(datum_cas_read))
hmemcpy(pDataDestination, pDataSource, sizeof(S_VYSTUPY_RCB));     // pDataDestination
a pDataSource jsoupointery
```

  - **Cpp:** -
  - **SCC:** Cílený test na výskyt funkcí memset a memcpy.
  - **Man:** Manuální prověření nalezených výskytů.
- [R] Při práci s polem používáme pro určení “MAX-1” stejnou symbolickou konstantu, jaké bylo použito při definici pole.

```
#define  DEL_CNT  20
short            DelkyPriNacteni[DEL_CNT];
for( i = 0; i < DEL_CNT; i++){
     DelkyPriNacteni[i] = …
    }
```

  - **Cpp:** -
  - **SCC:** -
  - **Man:** Manuální kontrola.

5.3 Značení proměnných, konstant, funkcí a typů v jazyce C/C++

- [HR] Vytváření jmen proměnných se řídí samostatným interním dokumentem.

5.3.1 Značení datových typů v jazyce C/C++

Značení datových typů závisí na vývojové platformě příslušného SW. V rámci jednoho SW musí být použit jednotný systém značení datových typů. Datový typ „int“ může mít velikost 16 nebo 32 bitů. Z důvodu této nejednoznačnosti je nutno místo něho používat datové typy s definovanou velikostí – short (16 bitů) nebo long (32 bitů).

V rámci společnosti byly zavedeny následující ekvivalenty pro značení datových typů. Použití jiného způsobu značení není povoleno. Výjimka může být vynucena použitím nové vývojové platformy nebo převzetím části SW od externího dodavatele. Je preferováno používat **červeně** označené typy na platformě Škoda Electric a **modře** označené typy na platformě AMiT.

Význam	Ekvivalenty značení datového typu						
Logická proměnná s významem TRUE/ FALSE	**BOOL**	BOOL	bool	**Bool**			
Neznaménková 8bit proměnná	**uchar**	BYTE	byte	**U8**		unsigned char	
neznaménková 16-bit proměnná	**ushort**	WORD	word	**U16**		unsigned short	
neznaménková 32-bit proměnná	**ulong**	DWORD	dword	**U32**		unsigned long	
neznaménková 64-bit proměnná	**ullong**	DDWORD	ddword	**U64**			
znaménková 8-bit proměnná	**char**	BYTE		**I8**		signed char	
znaménková 16-bit proměnná	**short**	WORD		**I16**		signed short	
znaménková 32-bit proměnná	**long**	LONG	long	**I32**		long	
znaménková 64-bit proměnná	**llong**	LLONG	llong	**I64**			

Tabulka 4 - Značení datových typů

Pro tvorbu nových typů proměnných typu struktura se musí vždy použít deklarace typu pomocí *typedef*. Názvy typů jsou tvořeny písmeny, jednotlivé části jsou oddělené velkými písmeny, případně podtržítkem.

```
typedef struct _TyNode {
  long lData;
  struct _TyNode *ptyNext;
} TyNode, *pTyNode;
```

Při definici struktury je třeba brát v úvahu zarovnávání položek dle velikosti slova procesoru. To je důležité při předávání dat mezi různými procesory. Pro dorovnání slova je

nutno datovou strukturu doplnit pomocí vloženého prvku typu BYTE resp. WORD.

```
typedef struct can_obj {
  WORD  msg_ctl;     // Message Control
  DWORD arbitr;      // Arbitration
  BYTE msg_cfg;      // Message Configuration
  BYTE msg_data[8]; // Message Data 0 .. 7 byte
  BYTE dummy;        // Reserved Byte
};
```

5.3.2 Značení funkcí v jazyce C/C++

Jména funkcí/procedur mohou obsahovat pouze malá a velká písmena anglické abecedy, podtržítka a číslice. Jméno funkce začíná vždy písmenem. Jméno funkce začíná vždy písmenem. Jména funkcí nesmí být záměnná s konstantami a musí být v rámci SW projektu jedinečná (nesmí být definovány různé funkce se stejným jménem, ani nesmí být definována stejná funkce na dvou místech v SW projektu).

```
static void SW_Ochrany_Stridacu_U21U22_M2 (void){
}
```

5.3.3 Značení symbolických konstant v jazyce C/C++

Názvy symbolických konstant (maker) jsou tvořeny velkými písmeny. Jednotlivé části jsou oddělené podtržítkem.

```
#define NORM_X10   1023                          //Rozsah dilku + (pro 10bit A/D prevodnik)
```

6. Závěr

První část práce vysvětluje postupy uplatňované při vývoji SW pro zajištění jeho požadované kvality. Byly zmíněny metody používané výhradně pro SW (statická analýza, testování a validace SW) ale i metody obecnější – pod souhrnným označením QA (Quality Assurance). Práce také zmiňuje proces normalizace a představuje hlavní organizace, které se jím zabývají.

Ve druhé části práce je popsán celý životní cyklus vývoje SW projektu se všemi náležitostmi. Jsou popsány jednotlivé etapy, jejich vstupy a výstupy. Je vysvětlen pojem V-diagram, jako základní model životního cyklu SW uplatňovaný ve ŠKODA Transportation.

Další část diplomové práce je již praktická – obsahuje analýzu konkrétního prostředí a výběr vhodných SW nástrojů k realizaci celého systému statické analýzy kódu. Byl představen nástroj dostupný pod licencí GNU (General Public License) – Cppcheck a nástroj SCC vytvořený ve společnosti ŠT, který jsem výrazně upravil a doplnil. Nedílnou částí mé práce bylo napojení těchto nástrojů na verzovací systém Mercurial a vyřešení prezentace a třídění výsledků pomocí webového rozhraní SonarQube. Průběh a výsledky statické analýzy ilustruje řada obrázků a dokumentů v příloze.

Poslední část práce obsahuje normu kódování SW společnosti ŠTRN. U každého z pravidel normy je uvedeno, jakým způsobem jej lze ověřit v navrženém systému statické analýzy kódu. Mnoho pravidel lze již dnes ověřit pomocí automatických nástrojů (Cppcheck nebo SCC). Hodně práce však stále zůstává na manuální kontrole zkušeným verifikátorem.

Hlavní cíl diplomové práce byl splněn. Navržený systém statické analýzy kódu je funkční a s velkou pravděpodobností bude používán při vývoji SW nových zakázek. Bude nutné provést proškolení jeho budoucích uživatelů a provádět průběžnou údržbu systému. Předpokládá se také postupné vylepšování a doplňování testů prováděných programem SCC.

7. Seznam literatury a informačních zdrojů

[1] *Doc SonarQube* [online]. Switzerland: SonarSource, 2008 [cit. 2018-01-22]. Dostupné z: https://docs.sonarqube.org

[2] *Drážní zařízení - Sdělovací a zabezpečovací systémy a systémy zpracování dat - Software pro drážní řídicí a ochranné systémy: ČSN EN 50128*. Ed. 2. Praha: Úřad pro technickou normalizaci, metrologii a státní zkušebnictví, 2014.

[3] *Norma kódování SW NŘ*. ŠKODA TRANSPORTATION, 2018.

[4] Kwon, W. (2009), Software Testing ISO Standard 29119, STA Consulting Inc.

[5] MISRA. *MISRA* [online]. Warwickshire, UK: MIRA Limited, 2008 [cit. 2018-03-28]. Dostupné z: https://www.misra.org.uk/

[6] Co je to technická norma?. *Úřad pro technickou normalizaci, metrologii a státní zkušebnictví* [online]. Praha: ÚNMZ, 2018 [cit. 2018-04-23]. Dostupné z: http://www.unmz.cz

8. Přílohy

Příloha A – Skript pro analýzu projektů

```
$logPath = split-path -parent $MyInvocation.MyCommand.Definition
$logPath += "\Logy\analyza_"
$logPath += Get-Date -Format "%yMMdd_HH-mm"
$logPath += ".log"
Start-Transcript -path $logPath #výpis logu, | Out-Default

pushd C:\Staticka_analyza\Projekty
$projekty = "NIM\LCC", "NIM\LCC_STW"

foreach($projekt in $projekty) {

    pushd $projekt
"
======== Projekt: $projekt ========
"

    "-- Kontrola nových changesetů: --"
    hg incoming --noninteractive 2>&1 | Out-Default
   if($lastExitCode -eq 0) { # pokud jsou na serveru nové changesety
        "-- Pull & update: --"
        hg pull   --noninteractive 2>&1 | Out-Default
        hg update --noninteractive 2>&1 | Out-Default
    }

    popd

    if ($lastExitCode -eq 0) { # pokud na serveru byly nové changesety a povedl se
pull i update
        "-- Analýza kódu pomocí SCC: --"
        ..\Code_checkers\SCC\SCC.exe $projekt 2>&1 | Out-Default

    if ($projekt -eq "NIM\LCC") {
       "-- Analýza kódu pomocí Cppcheck pro LCC: --"
       ..\Cppcheck\cppcheck.exe --enable=all --xml --verbose --inconclusive --
inline-suppr --quiet --force $projekt --output-file=NIM\LCC\cppcheck.xml
    }
    if ($projekt -eq "NIM\LCC_STW") {
       "-- Analýza kódu pomocí Cppcheck pro LCC_STW: --"
       ..\Cppcheck\cppcheck.exe --enable=all --xml --verbose --inconclusive --
inline-suppr --quiet --force $projekt --output-file=NIM\LCC_STW\cppcheck.xml
    }

        pushd $projekt
        "-- Sonar-scanner (spuštění analýzy pro SonarQube): --"
        ..\..\..\sonar-scanner-3\bin\sonar-scanner.bat 2>&1 | Out-Default
        popd
    }
}

popd

Stop-Transcript
```

Příloha B – Vygenerovaný report

```
**********************
Windows PowerShell transcript start
Start time: 20180403094339
Username: SKODA\vaclav.prucha
RunAs User: SKODA\vaclav.prucha
Machine: SRV-DATARAIL1 (Microsoft Windows NT 6.2.9200.0)
Host Application: C:\Windows\system32\WindowsPowerShell\v1.0\PowerShell_ISE.exe
Process ID: 7540
**********************
Transcript started, output file is C:\Staticka_analyza\Logy\analyza_180403_09-43.log

======== Projekt: NIM\LCC ========

-- Kontrola nových changesetů: --
comparing with T:\TRN\Mercurial\Projects\NIM_express\LCC
searching for changes
no changes found
-- Analýza kódu pomocí SCC: --
Exporting to SCC-report.xml.
Exporting to SCC_report2.xml.
-- Analýza kódu pomocí Cppcheck pro LCC: --
-- Sonar-scanner (spuštění analýzy pro SonarQube): --
INFO: Scanner configuration file: C:\Staticka_analyza\sonar-scanner-3\bin\..\conf\sonar-scanner.pro
perties
INFO: Project root configuration file: C:\Staticka_analyza\Projekty\NIM\LCC\sonar-project.propertie
s
INFO: SonarQube Scanner 3.0.3.778
INFO: Java 1.8.0_121 Oracle Corporation (64-bit)
INFO: Windows Server 2012 R2 6.3 amd64
INFO: User cache: C:\Users\vaclav.prucha\.sonar\cache
INFO: Load global repositories
INFO: Load global repositories (done) | time=1094ms
INFO: User cache: C:\Users\vaclav.prucha\.sonar\cache
INFO: Load plugins index
INFO: Load plugins index (done) | time=140ms
INFO: SonarQube server 6.2
INFO: Default locale: "en_US", source code encoding: "UTF-8"
INFO: Process project properties
INFO: Load project repositories
INFO: Load project repositories (done) | time=1237ms
INFO: Execute project builders
INFO: Execute project builders (done) | time=16ms
INFO: Load quality profiles
INFO: Load quality profiles (done) | time=515ms
INFO: Load active rules
INFO: Load active rules (done) | time=890ms
INFO: Publish mode
INFO: ------------  Scan NIM:LCC
INFO: Load server rules
INFO: Load server rules (done) | time=281ms
INFO: Initializer GenericCoverageSensor
INFO: Initializer GenericCoverageSensor (done) | time=0ms
INFO: Base dir: C:\Staticka_analyza\Projekty\NIM\LCC
INFO: Working dir: C:\Staticka_analyza\Projekty\NIM\LCC\.scannerwork
INFO: Source paths: projects/109mrp/procs
INFO: Source encoding: UTF-8, default locale: en_US
INFO: Index files
INFO: 277 files indexed
```

```
INFO: Quality profile for c++: SCC
INFO: Sensor Lines Sensor
INFO: Sensor Lines Sensor (done) | time=47ms
INFO: Sensor SCM Sensor
INFO: SCM provider for this project is: hg
INFO: 3 files to be analyzed
INFO: 2/3 files analyzed
INFO: Searching reports by relative path with basedir 'C:\Staticka_analyza\Projekty\NIM\LCC' and se
arch prop ''
INFO: Undefined report path value for key ''
INFO: Issues processed = 0
INFO: Sensor CxxCoverageSensor (done) | time=0ms
INFO: Sensor CxxCppCheckSensor
INFO: Searching reports by relative path with basedir 'C:\Staticka_analyza\Projekty\NIM\LCC' and se
arch prop 'sonar.cxx.cppcheck.reportPath'
INFO: Scanner found '1' report files
INFO: Parser will parse '1' report files
INFO: Processing report 'C:\Staticka_analyza\Projekty\NIM\LCC\cppcheck.xml'
INFO: Added report 'C:\Staticka_analyza\Projekty\NIM\LCC\cppcheck.xml' (parsed by: CppcheckParserV2
)
INFO: CppCheck Errors processed = 1
INFO: Sensor CxxCppCheckSensor (done) | time=4141ms
INFO: Sensor CxxPCLintSensor
INFO: Searching reports by relative path with basedir 'C:\Staticka_analyza\Projekty\NIM\LCC' and se
arch prop 'sonar.cxx.pclint.reportPath'
INFO: Undefined report path value for key 'sonar.cxx.pclint.reportPath'
INFO: PC-Lint errors processed = 0
INFO: Sensor CxxPCLintSensor (done) | time=0ms
INFO: Sensor CxxDrMemorySensor
INFO: Searching reports by relative path with basedir 'C:\Staticka_analyza\Projekty\NIM\LCC' and se
arch prop 'sonar.cxx.drmemory.reportPath'
INFO: Undefined report path value for key 'sonar.cxx.drmemory.reportPath'
INFO: Dr Memory errors processed = 0
INFO: Sensor CxxDrMemorySensor (done) | time=0ms
INFO: Sensor CxxCompilerSensor
INFO: Searching reports by relative path with basedir 'C:\Staticka_analyza\Projekty\NIM\LCC' and se
arch prop 'sonar.cxx.compiler.reportPath'
INFO: Undefined report path value for key 'sonar.cxx.compiler.reportPath'
INFO: C++ compiler Warnings processed = 0
INFO: Sensor CxxCompilerSensor (done) | time=0ms
INFO: Sensor CxxVeraxxSensor
INFO: Searching reports by relative path with basedir 'C:\Staticka_analyza\Projekty\NIM\LCC' and se
arch prop 'sonar.cxx.vera.reportPath'
INFO: Undefined report path value for key 'sonar.cxx.vera.reportPath'
INFO: Vera++ rule violations processed = 0
INFO: Sensor CxxVeraxxSensor (done) | time=0ms
INFO: Sensor CxxValgrindSensor
INFO: Searching reports by relative path with basedir 'C:\Staticka_analyza\Projekty\NIM\LCC' and se
arch prop 'sonar.cxx.valgrind.reportPath'
INFO: Undefined report path value for key 'sonar.cxx.valgrind.reportPath'
INFO: Valgrind errors processed = 0
INFO: Sensor CxxValgrindSensor (done) | time=0ms
INFO: Sensor CxxExternalRulesSensor
INFO: Searching reports by relative path with basedir 'C:\Staticka_analyza\Projekty\NIM\LCC' and se
arch prop 'sonar.cxx.other.reportPath'
INFO: Scanner found '1' report files
INFO: Parser will parse '1' report files
INFO: Processing report 'C:\Staticka_analyza\Projekty\NIM\LCC\SCC-report.xml'
INFO: External C++ rules violations processed = 3182
INFO: Sensor CxxExternalRulesSensor (done) | time=4843ms
```

```
INFO: Sensor Unit Test Results Import
INFO: Sensor Unit Test Results Import (done) | time=0ms
INFO: Sensor Zero Coverage Sensor
INFO: Sensor Zero Coverage Sensor (done) | time=125ms
INFO: Sensor Code Colorizer Sensor
INFO: Sensor Code Colorizer Sensor (done) | time=47ms
INFO: Sensor CPD Block Indexer
INFO: DefaultCpdBlockIndexer is used for c++
INFO: Sensor CPD Block Indexer (done) | time=0ms
INFO: Calculating CPD for 231 files
INFO: CPD calculation finished
INFO: Analysis report generated in 3296ms, dir size=3 MB
INFO: Analysis reports compressed in 1812ms, zip size=1 MB
INFO: Analysis report uploaded in 641ms
INFO: ANALYSIS SUCCESSFUL, you can browse http://10.42.1.69:4950/dashboard/index/NIM:LCC
INFO: Note that you will be able to access the updated dashboard once the server has processed the
submitted analysis report
INFO: More about the report processing at http://10.42.1.69:4950/api/ce/task?id=AWKKf-1RiX2RafB4w9S
0
INFO: Task total time: 53.047 s
INFO: ------------------------------------------------------------------------
INFO: EXECUTION SUCCESS
INFO: ------------------------------------------------------------------------
INFO: Total time: 57.984s
INFO: Final Memory: 44M/233M
INFO: ------------------------------------------------------------------------
```

Příloha C – PDF výstup

NIM:LCC
0.1

2018-04-17

sonar	Sonar PDF Report	NIM:LCC

Table of contents

sonar	Sonar PDF Report	NIM:LCC

1. NIM:LCC

This chapter presents an overview of the project measures. This dashboard shows the most important measures related to project quality, and it provides a good starting point for identifying problems in source code.

1.1. Report Overview

Static Analysis

Lines of code
34,209
N/A packages
20 classes
1,264 methods
0.0% duplicated lines

Comments
43.8%
26,666 comment lines

Complexity
10.9
0.0 /class
13,805 decision points

Dynamic Analysis

Code Coverage
N/A
N/A tests

Test Success
N/A
N/A failures
N/A errors

Coding Rules Issues

Technical Debt
0

Issues
83

1.2. Issues Analysis

Most violated rules	
Cykly	83

Most violated files	
Diag_ZB_Vyhodnoceni2.c	20
MVB_Diag_Kom.c	13
Diag_ZB_Vyhodnoceni4.c	8
Diag_ZB_Vyhodnoceni1.c	7

sonar	Sonar PDF Report	NIM:LCC

Pvst_WTB_MVB.c	6

Most complex files	
MVB_get_data_TDD.c	492
Data_WTB_R1.c	342
Pvst_ATO.c	303
Data_WTB_R2.c	264
Ovl_ZkouskaBrzdy_Prubeh.c	255

Most duplicated files
No duplications

1.3. Issues Details

Rule	Cykly
File	Line
MVB_Diag_Kom.c	327, 337, 413, 354, 404, 188, 306, 516, 316, 344, 510, 218, 296
MVB_get_data_TDD.c	72
Pvst_WTB_Akt.c	114, 343, 277, 191
Pvst_WTB_MVB.c	179, 100