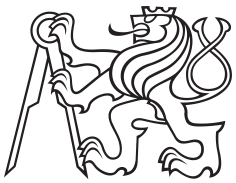
Diplomová práce



České
vysoké
učení technické
v Praze

F3

Fakulta elektrotechnická
Katedra měření

Testovací prostředí pro integrační testování real-time software složeného z více komponent

Bc. Richard Steidl

Vedoucí: Ing. Joel Matějka
Obor: Počítačové inženýrství
Studijní program: Otevřená informatika
Srpen 2020

Poděkování

Děkuji svému vedoucímu práce za jeho vedení, rady i za vřelý přístup.
Dále děkuji své rodině za podporu i za odborné debaty v průběhu práce.
V neposlední řadě děkuji společnosti Siemens Mobility GmbH za poskytnutí zázemí při implementaci a ověřování konceptu.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

V Praze dne 09. srpna 2020

.....
podpis autora práce

Abstrakt

Práce analyzuje možnost tvorby obecného nástroje pro zjednodušení testování softwaru na úrovni komponent v rámci multikomponentního systému za plné softwarové integrace.
Práce rovněž obsahuje konkrétní návrh řešení, který je následně i implementován pro ověření navrženého konceptu při testování softwarové komponenty splňující bezpečnostní úroveň SIL 4 (Safety integrity level) na základě normy EN 50128:2011 [EN511].
Navržené testovací prostředí snižuje implementační náročnost pro testera a rovněž dělí testovací činnost do dvou částí, na přípravu testovacího prostředí pro konkrétní projekt a na samotnou tvorbu testovacích sad.
Z těchto dvou částí vyžaduje pouze příprava testovacího prostředí hlubší seznámení s testovaným projektem. Pro tvorbu samotných testovacích sad na sebe již navržené testovací prostředí přebírá významnou část nutné funkcionality a tím výrazně snižuje požadavky na specializaci uživatele.

Klíčová slova: Testování softwaru, Aplikace reálného času, Komponenta, Testování komponent, Integrační testování, Úroveň integrity bezpečnosti, Bezpečnostně-kritický systém, C++, Vestavný systém, Železniční infrastruktura, Železniční automatizace

Vedoucí: Ing. Joel Matějka
Katedra řídící techniky,
Karlovo náměstí 13,
Praha 2

Abstract

The thesis analyses possibility of creation of generic tool for testing vertical multi-component system. The testing is done on component level, while full software integration is achieved.
The thesis also contains proposal of test framework. To prove the concept, the proposed solution is implemented and used while testing software component that complies with SIL 4 (Safety integrity level) limitations that are defined in norm EN 50128:2011 [EN511].
The proposed framework reduces the implementation complexity for tester and also divides the test implementation to two parts: to preparation of the framework for specific project, and to testcase creation itself.
Out of these two parts, only the preparation of the framework requires deeper knowledge of the tested project. For the testcase creation, the framework takes major part of necessary functionality on itself and therefore the required tester specialisation for this part is greatly reduced.

Keywords: Software testing, Real-time, Component, Component testing, Integration testing, Safety integrity level, Safety-critical system, C++, Embedded, Railway infrastructure, Railway automation

Title translation: Test environment for integration testing of component-based real-time software

Obsah

Obrázky

Tabulky

Kapitola 1

Úvod

P°i vývoji bezpe£nostn¥ kritického softwaru je kladen velký d·raz na celý vývojový proces, jeho º významnou sou£ástí také testování, ov¥°ující korektní chování vyvinutého softwaru.
Celý vývojový proces je pro bezpe£nostn¥ kritická za°ízení zárove¬ regulován poºadavky konkrétních norem, které popisují celý proces vývoje a kladou speci cké poºadavky, jakým zp·sobem mají být jednotlivé £ásti procesu vykonávány a dokumentovány.
V rámci této práce se ve spolupráci se Siemens Mobility GmbH (dále v práci pouze SIEMENS) zabýváme konkrétní normou stanovující podmínky pro veri- ka£ní £innost p°i vývoji za°ízení pro ºelezni£ní automatizaci EN 50128:2011 [EN511].

Pro testování projekt· svázaných poºadavky norem je vyuºití existujících a v²eobecn¥ roz²í°ených testovacích prost°edí jako je GoogleTest [Goo19] £i CppUTest [GV20] moºné pouze za spln¥ní dodate£ných podmínek.
Tato prost°edí zárove¬ nejsou uzp·sobena pro testování za mandatorních podmínek stanovených normami, coº p°iná²í dodate£nou implementa£ní náro£nost pro testera.

Cílem této práce je proto navrºení testovacího prost°edí pro testování na úrovni komponent v plné softwarové integraci, které spl¬uje podmínky pro testování jakéhokoliv softwaru vyvíjeného v souladu s podmínkami pro nejvy²²í úrove¬ bezpe£nosti SIL4 (Safety integrity level) de novené normou EN 50128:2011 [EN511].
Výstupy této práce je v²ak moºné vyuºít pro jakýkoliv systém spl¬ující obdobné regulatorní poºadavky.

Z d·vodu poºadavk· na plnou softwarovou integraci vyplývajících ze zmi-

¬ované normy a na komplexitu jednotlivých komponent nejsou standardn¥ uºívaná prost°edí jako CppUTest [GV20] £i GoogleTest [Goo19] bez významné vlastní nadstavby pro tuto £innost vhodná.
Významná £ást této nadstavby je ov²em generalizovatelná. V rámci návrhu testovacího prost°edí v této práci je práv¥ tato generalizovatelná £ást implementována.

Prost°edí je navrºeno pro testování multikomponentního systému, jehoº jednotlivé komponenty jsou vrstveny vertikáln¥, a je tedy ve vztahu dvou komponent vºdy jedna vy²²í (aplika£ní) a jedna ni²í (základová).

Navrºené prost°edí je vytvo°eno s ohledem na real-time aplikaci testovaného za°ízení.
Je o£ekáváno, ºe testovaná komponenta je volána z ni²í £ásti systému v p°edde novaných £asových intervalech a pro její b¥h je vyhrazena pouze omezená doba. Proto má navrºené testovací prost°edí moºnost sledovat rovn¥º reálný £as a na jeho základ¥ je schopné p°eru²it svou £innost s p°edde novanou periodou, aby nebyl v integraci p°esaºen £as vyhrazený testované komponent¥. Následn¥ je prost°edí schopné na svou £innost navázat ve chvíli, kdy op¥t nastane £as pro jeho b¥h.

D·raz je dále kladen na moºnost b¥hu test· v integraci s cílovým hardwarem, nebo´ nejni²í komponenty systému mohou být na hardwaru závislé. Z tohoto d·vodu je prost°edí optimalizováno pro minimalizaci nárok· na pam¥´ovou i výpo£etní kapacitu.

Pro dosaºení maximálního p°iblíºení pr·b¥hu test· výsledné aplikaci je kladen d·raz na moºnost vyuºití stejných kompilátor· a kompila£ních kon gurací, jaké vyuºívá nální software a navrºené prost°edí je z tohoto d·vodu zcela nezávislé na externích knihovnách v£etn¥ standardních.

Dal²ím významným speci kem navrºeného prost°edí je optimalizace lidských zdroj· v rámci vývoje produktu.
Testovací prost°edí je proto navrºeno tak, aby samotné testování sestávalo ze dvou fází.
První fází je p°íprava testovacího prost°edí v závislosti na speci ckých parametrech konkrétní testované softwarové komponenty. Tato fáze vyºaduje práci £lov¥ka seznámeného s problematikou cílového softwaru.
Druhou fází je poté tvorba samotných testovacích sad a jejich speci kací. Ta jiº probíhá výhradn¥ p°es gra cké rozhraní testovacího prost°edí, £ímº jsou redukovány poºadavky na testera z hlediska hlub²ího porozumn¥ní technickým speci k·m projektu.

Navrºené testovací prost°edí je v rámci práce implementováno a funkcionalita navrºeného konceptu je ov¥°ena p°i testování za°ízení z °ady xCM spole£nosti SIEMENS, spl¬ující bezpe£nostní stupe¬ SIL4 normy EN 50128:2011 [EN511],

které se zabývá ºelezni£ní automatizací.

Diplomová práce je po úvodu £len¥na následovn¥:
Ve druhé kapitole jsou p°edstaveny pojmy nezbytné pro pochopení práce.
T°etí kapitola obsahuje poºadavky na navrºené testovací prost°edí, které vyplývají ze zadání práce a z poºadavk· normy EN 50128:2011 [EN511].
ƒtvrtá kapitola obsahuje popis navrºeného testovacího prost°edí a vysv¥tluje jeho speci ka.
Pátá kapitola je v¥nována p°ímé implementaci navrºeného testovacího prost°edí, srovnává ho se standardn¥ vyuºívanými prost°edími a vysv¥tlují se v ní omezení dané implementace.
’está kapitola rozebírá praktické vyuºití implementovaného testovacího prost°edí, ukazuje plánované vyuºití v rámci vývoje za°ízení pro ºelezni£ní automatizaci rmy SIEMENS a rovn¥º poskytuje náhled na moºnost vyuºití na dal²ích projektech. V rámci této kapitoly je rovn¥º ukázán sm¥r, kterým se budoucí vývoj bude ubírat.

Kapitola 2

De nice a teoretický základ

2.1 Multikomponentní systém

Multikomponentní systém je takový, který se skládá z v¥t²ího mnoºství na sob¥ nezávislých komponent. Tyto komponenty mezi sebou komunikují prost°ednictvím p°esn¥ de novaných rozhraní a vzájemn¥ si poskytují sluºby jim ur£ené.
P°íklon k multikomponentnímu systému je z pohledu vývoje ºádoucí, nebo´ jednotlivé komponenty jsou znovupouºitelné v jiných projektech a jejich vývoj je na sob¥ (vyjma zade novaného rozhraní) zcela nezávislý [LW07].
Obecný multikomponentní systém lze d¥lit na mnoho kategorií, na základ¥ skládání jednotlivých komponent, jejich vzájemné komunikaci a dal²ích parametrech, které znázor¬uje obrázek 2.1 [CSVC10].

Obrázek 2.1: D¥lení multikomponentního systému [CSVC10]

Z pohledu normy, na ni^{o} má brát práce ohled, je v²ak de nice komponenty výrazn¥ skoup¥j²í. Norma EN 50128:2011Ł[N511] ve volném p°ekladu de - nuje komponentu následovn¥:
komponenta je £ást softwaru, která má p°esn¥ zade novaná rozhraní a své chování s ohledem na softwarovou architekturu a design a spl¬uje následující kritéria:

- je navrºena podle 2.2
- pokrývá ur£itou podmnoºinu poºadavk· na software
- je p°esn¥ identi kovatelná a je nezávislá, a nebo je sou£ástí skupiny komponent, která je nezávislá.

Obrázek 2.2: Poºadavky na komponenty podle [EN511]. R - Recommended, HR - Highly recommended, M - mandatory

Z d·vodu nep°esné v²eobecn¥ uznávané de nice komponenty [BDH+98], [WPC01], [EN511] se práce bude zabývat pouze speci ckým typem komponent. Multikomponentní systém v na²em chápání bude takový, jehoº komponenty spl¬ují speci ka nastín¥ná v kapitole 2.1.1.

Tyto komponenty budou v rámci testovaného systému vrstvené, kdy ve vztahu dvou komponent je vºdy jedna niº²í (základová) a jedna vy²²í (aplika£ní), a tedy budou reprezentovat vertikální typ skládání systému na 2.1 [CSVC10].

2.1.1 Speci ka komponenty

- Komponenta reprezentuje jeden logický £i organiza£ní proces
- D¥lení na komponenty je hrub²í neº na jednotlivé t°ídy. M·ºe se skládat z více vzájemn¥ provázaných t°íd.
- Komponenta se m·ºe skládat z více (sub) komponent.

- Komponenta komunikuje s ostatními komponentami prost°ednictvím p°esn¥ de novaného rozhraní.
- Komponenta je nezávislá na verzi projektu. M·ºe být samostatn¥ vylep-²ována na novou verzi.
- Je navrºena podle poºadavk· na SIL4 projekt znázorn¥nými v 2.2

Tato speci kace voln¥ vychází z parametr· de novaných Antonem Deimelem [BDH+98] a je dopln¥na o speci cké poºadavky normy EN 50128:2011 [EN511].

2.2 Real-time systém

Za real-time systém povaºujeme jakýkoliv systém, který musí interagovat s vn¥j²ím sv¥tem s ur£itou periodou £i reagovat ur£itou rychlostí na externí vlivy. Zkrácen¥ lze tedy °íci, ºe je to jakýkoliv systém, který pro své °ízení vyuºívá reálný £as a garantuje jeho dodrºení [LMW95], [BG92].
V na²em systému de novaném v 2.1 je speci kace aplikace reálného £asu dosaºeno prost°ednictvím periodického volání ur£ité funkce (cyclicFunction) ze základové komponenty, která je vstupní branou do komponenty aplika£ní. Tato aplika£ní komponenta musí v de novaném £ase vrátit °ízení b¥hu programu zp¥t komponent¥ základové, tedy dodrºet £asový slot vyhrazený pro její b¥h.

2.3 Testování multikomponentního systému

Testování poºadavk· se skládá ze t°ech základních stup¬·. Jedná se o Unit test 2.3.1, Component test 2.3.2 a System test 2.3.3.
Kaºdá z úrovní p°iná²í rozdílnou míru integrace a tím detekce potenciálních problém· spojených s interakcí mezi jednotlivými £ástmi programu. Opa£ným sm¥rem proti mí°e integrace pak jde samotná testovatelnost programu. *Ƒ*ím je testovaná £ást programu ucelen¥j²í, tím je sloºit¥j²í otestovat v²echny kombinace vnit°ních stav·, a rovn¥º m·ºe být nemoºné pro ú£el testu upravovat hodnoty vnit°ních (zapouzd°ených) prom¥nných.
V²echny úrovn¥ ov²em spojuje snaha o co nejbliº²í p°iblíºení se reálné aplikaci,

kterého lze dosáhnout na dané úrovni testování. Toho lze dosáhnout n¥kolika cestami, kde mezi nejvýznam¥j²í pat°í testování ji º p°edkompilovaných objekt· testovaného softwaru, aby se zamezilo ovlivn¥ní kompilace zp·sobené £ist¥ testovacím prost°edím.
Na obrázku 2.3 jsou znázorn¥ny jednotlivé úrovn¥ programu a na jaké úrovni jsou poté testovány. Mod°e je úrove¬ Unit testu, ºlut¥ Component testu a £erven¥ poté System test.

Obrázek 2.3: Úrovn¥ testování

Systém znázorn¥ný na obrázku 2.3 znázor¬uje aplikaci voltmetru, sloºeného ze dvou komponent:

- Vstupn¥-výstupní jednotka , která £te hodnoty z AD p°evodníku a prost°ednictvím p°edde novaného rozhraní tyto hodnoty poskytuje Logické jednotce.
  Dále se stará o ethernetovou komunikaci, prost°ednictvím níº posílá datové telegramy, které p°ipraví Logická jednotka a poskytne je op¥t prost°ednictvím rozhraní.

- Logická jednotka p°ijímá data AD p°evodníku od Vstupn¥-výstupní a vyhodnocuje p°evod. Následn¥ p°ipraví diagnostickou zprávu a poskytne ji Vstupn¥-výstupní jednotce pro odeslání po ethernetu.

2.3.1 Unit level

Unit level testování je nejjemn¥j²í standardn¥ vyuºívanou úrovní test·. Sleduje reakce díl£í £ásti programu (t°ídy £i ekvivalentu) na vstupní parametry funkcí, pop°ípad¥ na návratové hodnoty externích funkcí.
V této úrovni je tester schopen simulovat tém¥° libovolné chování volaných funkcí, a rovn¥º modi kovat v²echny vnit°ní hodnoty testované £ásti programu.
Tato úrove¬ testování je ²iroce podporována mnohými testovacími prost°edími, jelikoº je tém¥° nezávislá na konkrétním projektu. Existuje °ada vysoce optimalizovaných prost°edí pro tutu úrove¬, jmenovit¥ GoogleTest [Goo19], CppUtest [GV20] a mnoho dal²ích.

2.3.2 Component level

Component level testuje jednotlivé komponenty, které byly zade novány v 2.1.1. Sleduje chování jedné uzav°ené softwarové £ásti programu.
Pro simulaci externích událostí je vyuºíváno striktn¥ její externí rozhraní. Tester jiº nemá moºnost zasahovat do vnit°ních struktur samotné komponenty, ani moºnost p°istupovat k jednotlivým subkomponentám, vnit°ním prom¥nným a podobn¥.
Tato úrove¬ nabízí jiº významnou míru integrace, nebo´ je komponenta navrºena tak, aby byla v maximální mí°e uzav°ená sama do sebe, a díky tomu je na této útovni testována v¥t²ina vzájemných závislostí v rámci komponenty.

2.3.3 System level

System level nabízí nejvy²²í úrove¬ integrace, na této úrovni se jiº testuje nální produkt (systém) a sleduje se pouze jeho vn¥j²í interakce se sv¥tem.

Kontroluje se tu integrace jednotlivých komponent a jejich vzájemné ovliv-
n¥ní.
Testovaná funkcionalita se £áste£n¥ p°ekrývá i s testy na úrovni komponent,
av²ak v této úrovní testy nejdou do takové hloubky, nebo´ z d·vodu p°ístupu
£ist¥ p°es vn¥j²í rozhraní není mo°né kontrolovat ve²keré vnit°ní stavy.

2.3.4 Integra£ní testování

ƒáste£n¥ nezávisle na p°edchozích t°ech úrovních testování je dále test inte-
gra£ní. Integrací se rozumí kombinace co nejv¥t²ího mno°ství reáln¥ vyu°ivané
aplikace p°i b¥hu test· aby byl omezen p°ípadný ne°ádoucí vzájemný vliv
jednotlivých softwarových modul·. Tento vliv se m·°e objevit nap°íklad p°i
správ¥ pam¥ti, pop°ípad¥ p°i vyu°ívání externích zdroj· (kup°íkladu omezená
kapacita komunika£ní linky).

2.3.5 Po°adavky normy EN 50128:2011

Pro testovaný software s ohledem na normu EN 50128:2011 [N511] musí být
mimo jiné spln¥ny po°adavky vyplývající z 2.4 a 2.5.

Obrázek 2.4: Poºadavky na test softwaru podle [EN511]. R - Recommended, HR - Highly recommended, M - mandatory

Obrázek 2.5: Poºadavky pro m¥°ení pokrytí podle [EN511]. R - Recommended, HR - Highly recommended, M - mandatory

Poºadavky v tabulce 2.4 zahrnují rovn¥º veri ka£ní £innost, která nespadá pod p°ímou £innost testera a není vyhodnocována v rámci b¥hu samotných test·.

V rámci testování projektu na úrovni SIL4 je z pohledu testera zapot°ebí být schopen m¥°it pokrytí softwaru v rámci b¥hu test·, a to na základ¥ 2.5. Tohoto poºadavku je mimo jiné dosaºeno m¥°enímBranch coveragev rámci b¥hu integra£ních softwarových test·, tedy jednotlivé komponenty musí být pro m¥°ení pokrytí testovány spole£n¥ se v²emi ostatními komponentami systému a test musí projít kaºdým p°íkazem skoku v programu.

Dal²ím mandatorním poºadavkem na testera je b¥h test· ve form¥ black-box pro test funkcionality komponent.

2.3.6 Mock

Souhrnn¥ zna£í mock implementaci funkcionality netestovaných £ástí systému, respektive implementaci funkcí rozhraní, které je zade novené pro komunikaci s danou £ástí systému, pro simulaci interakce s testovanou £ástí softwaru (funkce, t°ída, komponenta, ...).
Tato implementace ostatních £ástí systému je zpravidla výrazn¥ jednodu²í a mén¥ komplexní, ne º je implementace reálných komponent, nebo´ má tester nad mocky plnou kontrolu a m·ºe vylou£it nap°íklad chybové chování.

Mock navíc poskytuje p°ístup k hodnotám, s nimi º jsou jednotlivé funkce rozhraní z testované £ásti softwaru volány a zárove¬ poskytující moºnost upravovat návratové hodnoty, díky £emu º je tester schopen nejen sledovat chování testované komponenty, ale rovn¥º simulovat chování zbytku systému.

Unit test mock

Pro Unit test úrove¬ je tak°ka výhradn¥ vyuºíváno funkcionality Mock implementace [TS06b], [TS06a].
Unit test prost°edí jako CppUTest [GV20] £i GoogleTest [Goo19] nabízejí pro implementaci mock· rozsáhlou podporu, kdy je moºné prost°ednictvím p°edde novaných maker sledovat, kdy byla jaká funkce volána a s jakými parametry, a zárove¬ nastavovat návratové hodnoty.
Nastavení návratových hodnot v²ak musí být vºdy explicitn¥ de nováno, nebo´ jsou prost°edí ur£ena primárn¥ pro Unit testování, kdy se neo£ekává del²í b¥h testované £ásti softwaru a mnohonásobné volání funkcí rozhraní.

Component test mock

Implementace mock· je v²ak vyuºitelná i pro testování na úrovni komponent. Z d·vodu vy²²í integrace je v²ak jiº nutné dbát na detailn¥j²í implementaci mock·.
V rámci test· komponent jiº není sledováno pouze volání jedné funkce a reakce testovaného softwaru. Jedná se jiº o komplexn¥j²í funkcionalitu, kdy sledujeme reakce na v¥t²í mnoºství p°ístup· do ostatních komponent, a tedy mock implementace musí být schopna si p°edávat stavy rovn¥º mezi jednotlivými voláními.

Jednoduchým p°íkladem nezbytnosti hlub²í implementace mock· budiº následující p°íklad:
Niº²í (netestovaná) komponenta p°ijímá data po ethernetové lince a ukládá si je do bu er·, mapované na základ¥ MAC adresy odesílatele.
Vy²²í vrstva poté vyuºije funkci rozhraní void writeEth (MAC_address, data).
Následn¥ vyuºije op¥t funkci rozhraní data getEthResponse ().
V této chvíli testovaná komponenta jiº o£ekává, ºe niº²í vrstva si pamatuje na jakou adresu byla odeslána poslední zpráva a tedy odkud je odpov¥¤ o£ekávána.

Kapitola 3

Poºadavky na testovací prost°edí

Aby byly spln¥ny poºadavky normy [EN511] a zárove¬ byla zachována praktická vyuºitelnost testovacího prost°edí, je nutné zajistit schopnost prost°edí m¥°it pokrytí testované komponenty v rámci plné softwarové integrace a zajistit jednozna£nou identi kaci samotných test· pro jejich provázání se speci kací a s poºadavky. Zárove¬ je mandatorní i moºnost garantovat schopnost opakovat testovací proces.

3.1 Moºnost softwarové integrace

Pro spln¥ní podmínek 2.3.5 je v rámci testovacího procesu nutné docílit softwarové integrace, v rámci které bude poté probíhat m¥°ení pokrytí testované komponenty testy.
Softwarové integrace je dosaºeno ve chvíli, kdy jsou v²echny komponenty výsledného systému zahrnuty do testovacího procesu. Z d·vodu, ºe nejni²í komponenty systému mohou být zárove¬ závislé na cílovém hardware, existuje moºnost, ºe softwarová integrace je rovn¥º podmín¥na integrací softwaru s hardwarem.
Jelikoº vycházíme z 2.1, o£ekáváme vertikáln¥ vrstvený multikomponentní systém, tedy v rámci vztahu dvou komponent vºdy jednu ni²í (základovou) a jednu vy²í (aplika£ní). Tato podmínka nám povoluje mezi ni²í a vy²í komponentu vloºit dal²í vrstvu, kterou je práv¥ testovací prost°edí.
Ve chvíli, kdy je testovací vrstva vloºena mezi základovou a aplika£ní vrstvu (3.1), je testovací prost°edí p°ebrat kompletní kontrolu nad b¥hem testované

komponenty a zárove¬ je docíleno softwarové integrace.

Obrázek 3.1: Vloºení testovací vrstvy do zbytku systému

3.2 Provázání poºadavek-test

Testovací proces musí být p°esn¥ speci kovatelný a zdokumentovatelný. Z tohoto d·vodu musí být moºné provázat jednotlivé testy s poºadavky, které jsou na testovaný software kladeny. Z tohoto d·vodu musí být jednotlivé testy p°esn¥ identi kovány.
V rámci testovacího prost°edí je proto nutné umoºnit d¥lit b¥h test· na jednotlivé testcaseș které je poté moºné v rámci dokumentace p°ímo provázat s konkrétními poºadavky.
Proto musí být v rámci p°ípravy testovacích sad vygenerován soubor obsahující speci kace jednotlivých testcas·, na základ¥ kterého je moºné testovací proces identi kovat a zopakovat. K tomuto nemusí nutn¥ slouºit p°ímo zdrojové kódy test·, nebo´ obsahují nadbyte£né mnoºství informací. Pro tuto funkci je dosta£ující a ºádoucí pouze soubor schra¬ující jednotlivé události, které jsou vykonané testerem nad rámec standardního b¥hu systému, a to ve form¥ £itelné b¥ºnému programátorovi, tedy nap°íklad ve form¥.json souboru.

3.3 Obecnost

Testovací prost°edí musí být obecné pro mo^{o}nost vyu^{o}ití s jakýmkoliv C++ kompilátorem a na jakémkoliv vertikáln¥ vrstveném multikomponentním systému. Z tohoto d·vodu je nutné aby prost°edí nebylo závislé na externích knihovnách v£etn¥ standardních, nebo´ standardní knihovny nemusejí být v testované komponent¥ vyu^{o}ívány a jejich de nice mohou kolidovat s de nicemi v rámci testované komponenty. Prost°edí si proto musí implementovat vlastní typy (jako je nap°íklad std::string) pro zachování maximální obecnosti prost°edí.

3.4 Minimální po^{o}adavky na u^{o}ivatele

Testovací prost°edí na sebe musí p°ebírat £ást po^{o}adavk· na specializaci u^{o}ivatele. Proto musí být testovací prost°edí rozd¥leno na dva separátní úkony. Prvním je p°íprava samotného prost°edí pro testování speci cké projektu a druhým tvorba samotných testovacích sad.
První úkon musí být vykonáván testerem s hlubokou znalostí projektu. Nicmén¥ úkon druhý ji^{o} musí být u^{o}ivatelsky p°ív¥tivý natolik, aby tvorba testovacích sad mohla být vykonávána ji^{o} testerem s výrazn¥ ni^{o2}í znalostí problematiky.
P°íprava testovacího prost°edí tedy musí implementovat standardní funkcionalitu systému a nabídnout mo^{o}nost do b¥hu zasahovat. Samotné zásahy ji^{o} mohou být vyu^{o}ívány v rámci gra ckého rozhraní a konkrétní £innost ji^{o} tv·rce testovacích sad nemusí implementovat, ani zevrubn¥ znát. Rozd¥lení práce je znázorn¥no na 3.2

Obrázek 3.2: Rozd¥lení práce na testování softwaru

Kapitola 4

Návrh testovacího prost°edí

Cílem práce bylo sestrojit testovací prost°edí pro mo^{o}nost testování softwarových komponent na úrovni softwarové integrace.
Tato úrove¬ se ji^{o} významn¥ blí^{o}í systémové úrovni test·, nebo´ je ji^{o}, na rozdíl od Unit test·, pop°ípad¥ Component test· (bez softwarové integrace), významn¥ závislá na konkrétním projektu.
Z tohoto d·vodu není mo^{o}né obecn¥ zade novat prost°edí pro v²echny typy komponent znázorn¥né na 2.1. Je ov²em mo^{o}né p°ipravit testovací prost°edí s ur£itými omezeními na konkrétní typ komponent.
V této práci se zabýváme pouze vertikáln¥ vrstvenými komponentami, co^{o} je ji^{o} dostate£ná speci kace pro tvorbu testovacího prost°edí pro testování takto strukturovaného programu.

Pro testování na úrovni softwarové integrace je mo^{o}né £áste£n¥ vyu^{o}ít v²eobecn¥ známá a optimalizovaná unit test prost°edí jako je GoogleTest [Goo19], CppUTest [GV20]. Nad jejich funkcionalitu je ov²em zapot°ebí naimplementovat dal²í hlub²í funcionalitu mock·, ne^{o} jakou tato prost°edí podporují.
Navr^{o}ené prost°edí vytvá°í obecnou nadstavbu nad libovolné unit test prost°edí, které je vytvo°eno p°ímo pro integra£ní testování na úrovni komponent v multikomponentním systému podle 2.1.
Prost°edí vyvinuté v rámci této práce se stará o validní b¥h testované komponenty na volitelné úrovni integrace se zbytkem systému a poskytuje mo^{o}nost v libovolných £asech do onoho b¥hu zasahovat.
Prost°edí rovn¥^{o} d¥lí celou testerskou £innost na dv¥ £ásti. ƒinnost implementa£ní, kdy je prost°edí p°ipravováno zku²eným testerem s hlub²í znalostí projektu, a poté £innost £isté tvorby testovacích sad, která probíhá výhradn¥ p°es gra cké rozhraní a tedy není zapot°ebí hlub²í znalost samotného projektu.
Tímto je docíleno efektivního vyu^{o}ití lidských zdroj· v rámci vývoje projektu,

díky p°esnému rozd¥lení na implementa£ní a testerskou £innost je mo°né vyu°ít zku²eného zam¥stnance pouze pro skute£n¥ nezbytnou práci. Pro mén¥ náro£nou testerskou £innost je ji° mo°né vyu°ít ne zcela znalých zam¥stnanc·.

4.1 Integrace

Jednotlivé komponenty mezi sebou komunikují prost°ednictvím rozhraní (interface), jak je zade nováno v 2.1.1. V²echny testy a úpravy vnit°ních stav· komponenty proto probíhají výhradn¥ prost°ednictvím t¥chto rozhraní. P°ístup, jakým je toho docíleno, je v²ak mo°né rozd¥lit do dvou kategorií.

4.1.1 Plné nahrazení systému

Pro nejjednodu²²í formu testování na úrovni komponent je tedy mo°né vytvo°it vlastní implementaci funkcí rozhraní, se kterými má testovaná komponenta komunikovat. Tím jsme schopni vy£ítat parametry, se kterými jsou funkce volány a simulovat funkcionalitu ostatních komponent.
Toto °e²ení je v²ak p°i testování komplexn¥j²ích komponent nedostate£né, nebo´ mnohá volání p°icházejí ve speci ckých okam°icích, a° kdy° se testovaná komponenta dostane do ur£itého vnit°ního stavu. Aby bylo mo°né to otestovat touto metodou, musí být vlastní implementace funkcí velmi komplexní a de facto implementovat celou komponentu, s ní° má testovaná komponenta komunikovat. Toto má za následek výraznou implementa£ní zát¥° pro testera. Na druhou stranu tato metoda p°iná²í i ur£ité výhody. Hlavní z nich je mo°nost uplatnit tuto formu testování komponenty v dob¥, kdy je tato komponenta oproti zbytku systému nap°ed, a tedy neexistuje jiná implementace ostatních komponent. Rovn¥° se díky této metod¥ m·°eme zcela odklonit od cílového hardwaru, se kterým mohou být ostatní komponenty spojeny, a v²echnu jejich funkcionalitu pouze simulovat. Tím m·°eme docílit testování komponenty nezávisle na cílovém hardwaru, a tedy testovat s vy²²í výpo£etní kapacitou.

Obrázek 4.1: Nahrazení systému

4.1.2 Plná integrace se systémem

Obecn¥j²ím °e²ením je ponechat i ostatní komponenty systému (tedy nejen komponentu testovanou), a p°esm¥rovávat pouze volání funkcí, které nás v daném testu zajímají. Toho lze docílit vlo°ením vrstvy testovacího prost°edí mezi vrstvu testovanou a zbytek systému. Testovací prost°edí pouze p°esm¥rovává v²echna volání do ostatních komponent, a v ký°eném okam°iku sleduje jedno konkrétní, které v sob¥ nese informaci nezbytnou pro test.
Tímto je docíleno nejen hlub²í integrace, nebo´ ji° na úrovni komponent test· rovn¥° kontrolujeme £ást systémové integrace, ale rovn¥° se výrazn¥ sní°í implementa£ní náro£nost pro testera.
Nevýhodou této metody je p°ímá závislost na více komponentách a fázi jejich vývoje.

Obrázek 4.2: Integrace do systému

4.1.3 ƒáste£né nahrazení systému

Jeliko^{o} oba p°ístupy mají své výhody i nevýhody, je v testovacím prost°edí mo^{o}né volit libovolnou kombinaci mezi t¥mito p°ístupy v závislosti na specických po^{o}adavcích konkrétního projektu.
Je tedy mo^{o}né vyu^{o}ít plné náhrady n¥které systémové komponenty, zatímco jiné nechat originální. Rovn¥^{o} je mo^{o}né implementovat pouze £ást funkcionality komponenty a zbytek nechat originální, a tím dosáhnout integrace na maximální mo^{o}né mí°e, zachovat plnou testovatelnost a zárove¬ omezit implementa£ní náro£nost pro testera.

4.2 Real time

P°i b¥hu na systému reálného £asu bylo nutné zvá^{o}it i dal²í aspekty, mezi ne^{o} pat°í zejména doba vyhrazená pro b¥h jednotlivých komponent. Externí

hardwarové watchdogy na cílovém za°ízení kontrolují, zda byla v£as p°edána kontrola zp¥t základové vrstv¥ programu. Proto je nutné, aby testovací vrstva v°dy dodr°ela £asový rámec, který je ur£ený pro vrstvu vy²²í.

4.2.1 Reálný a virtuální £as

Testovací prost°edí odd¥luje £as reálný a £as simulace. *ƒ* as simulace je virtuální a je závislý na výpo£etní kapacit¥ za°ízení, na které testovací prost°edí b¥°í. V°dy v²ak je garantováno, °e £as simulace pob¥°í rychleji ne° £as reálný. V opa£ném p°ípad¥ by toti° samotný hardware nebyl schopen výpo£etn¥ stihnout standardní b¥h programu. Obecn¥ lze tvrdit, °e ve v¥t²in¥ p°ípad· bude £as simulace výrazn¥ rychlej²í ne° £as reálný, a tím se celý testovací proces urychluje.
Reálný £as je vyu°íván k pozastavení simulace s de novanou periodou a navrácení kontroly nad programem zp¥t ni°²í vrstv¥, a poté k navázání dal²ího b¥hu po op¥tovném p°edání kontroly vy²²í vrstv¥.

4.3 Testovací strategie

Standardní zp·sob výkonu test· je zp·sobem restart-událost-kontrola. Tento zp·sob se nejen nevyhne £astému kopírování jednotlivých test·, ale rovn¥° musí p°i ka°dém testu v první °ad¥ dostávat komponentu do ur£itého stavu. Navrhované testovací prost°edí se °ídí odli²ným paradigmatem, konstantn¥ nechává komponentu b¥°et a v p°edem de novaných £asech simulace provede p°edde novanou akci.
P°i vyu°ití logiky zvolené v na²em testovacím prost°edí tedy m·°eme jednoho p°echodu do ur£itého stavu komponenty vyu°ít pro více samotných test·, £ím° je urychlen samotný testovací proces.
Rovn¥° tato struktura umo°¬uje znovupou°ití jednotlivých akcí, které jsou jednou zade novány, a poté umo°¬uje jejich vícenásobné vyu°ití, £ím° výrazn¥ sni°uje náro£nost samotné implementace.

4.4 Struktura

Testovací prost°edí je strukturováno jako samostatná uzav°ená komponenta, která je vlo^{o}ena jako vrstva mezi testovanou komponentu a zbytek systému. Tímto fakticky p°ebírá kontrolu nad b¥hem testované komponenty. Z pohledu testované komponenty prost°edí vystupuje celý zbytek systému, z pohledu zbytku systému vystupuje jako testovaná komponenta.
Pro komunikaci s testovanou komponentou vyu^{o}ívá svou implementaci jednotlivých funkcí rozhraní (Mock). Tyto funkce poté ukládají parametry, se kterými jsou volány, do Mock knihovny a rovn¥^{o} z této knihovny berou data pro odpov¥¤ na tato volání.
Tester si rov¥^{o} de nuje Akce, za pomoci kterých upravuje, pop°ípad¥ vy£ítá Mock knihovnu, a tím simuluje chování externích komponent systému a sleduje chování komponenty testované.
Struktura jednotlivých blok· je znázorn¥na na obrázku 4.3.

Obrázek 4.3: Architektura testovacího prost°edí

4.4.1 Mock

Mock jsou funkce, které implementují funkcionalitu rozhraní. Není nutné zachovat jejich kompletní funkcionalitu, by´ je ºádoucí se jí p°iblíºit co nejvíce. Jsou implementovány samotným testerem, nebo´ jsou p°ímo závislé na konkrétním testovaném projektu a rozsahu test·, a poté jsou vyuºívány jako náhraºka implementace rozhraní u komponent, které nejsou v p°ípad¥ £áste£né, pop°ípad¥ plné integrace se systémem jinak zade novány.
Tyto funkce plní v rámci testovacího prost°edí trojí roli.
První rolí je, ºe udrºují systém ve standardním b¥hu, tedy navrací o£ekávané hodnoty na volání jednotlivých funkcí.
Druhou rolí je úprava vnit°ního stavu testované komponenty, kdy práv¥ vyuºitím úpravy návratových hodnot dostáváme komponentu do ºádoucího stavu.
T°etí rolí je poté sledování parametr·, se kterými jsou jednotlivé funkce volány, £ímº umoº¬ují testerovi kontrolovat, ºe se komponenta chová podle o£ekávání.

4.4.2 Mock knihovna

Mock knihovna slouºí jako rozhraní mezi testerem a mockovanými funkcemi. Do této knihovny se ukládají jednotlivé parametry se kterými byly funkce volány. Na základ¥ toho poté mockované funkce volí návratové hodnoty, aby v¥rn¥ simulovaly funkci ostatních komponent systému.
Prost°ednictvím této knihovny m·ºe tester ovliv¬ovat chování jednotlivých mockovaných funkcí, a tím simulovat ur£ité nestandardní chování ostatních systémových £ástí.
Zárove¬ je moºné z této knihovny vy£ítat jednotlivé hodnoty, díky £emuº má tester moºnost sledovat chování samotné testované komponenty.

4.4.3 Akce

Akce jsou p°edde nované události, které m·ºe tester libovoln¥ spou²t¥t v jakýchkoliv £asech.
Tester si akce de nuje sám, nebo´ jsou p°ímo závislé na konkrétním projektu, a stejn¥ tak i na testovaných událostech.
Ú£el Akcí je ur£en primárn¥ k p°ipravování a vy£ítání dat z knihovny Mock·, £ímº aktivn¥ upravuje (respektive sleduje) chování testované komponenty.

Dal²í funkcionalitou Akcí je i ovládání samotné simulace za vyuºíté p°edregistrovaných callback funkcí (nap°íklad restart, stop simulation a dal²ích). Teoreticky v sob¥ mohou v²ak Akce nést i dal²í události, nebo´ se ve výsledku jedná o funkci, která je vykonána ve zvoleném £ase simulace. šádná omezení na konkrétní vyuºití nebyla zavedena, nebo´ se o£ekává, ºe tester bude mít k p°ípadnému jinému vyuºití oprávn¥né d·vody.
Pro sníºení implementa£ní náro£nosti jednotlivých Akcí mohou (a nemusí) být Akce ²iroce parametrizovány. Proto z pohledu testera sta£í zade novat nap°íklad SendTelegram, kde jiº zvolenými parametry ur£í komunika£ní linku, data i dal²í náleºitosti, vyplývající z jeho konkrétních pot°eb.

4.4.4 Pam¥´ová struktura

Z d·vodu b¥hu na cílovém hardwaru, u kterého nelze o£ekávat souborový systém, jsou jednotlivé akce nahrávány do programové pam¥ti formou automaticky vygenerované funkce. Ta je v rámci inicializace volána a v²echny akce jsou poté uloºeny na haldu v pam¥´ov¥ optimalizované form¥.

4.5 Gra cké rozhraní

Sou£ástí navrºeného testovacího prost°edí je rovn¥º gra cké rozhraní. Rozhraní slouºí k samotné tvorb¥ testovacích sad, kdy si uºivatel vybírá, které akce budou kdy provedeny.
D·vodem k tvorb¥ tohoto rozhraní je sníºení náro£nosti samotné implementace, a p°edev²ím diverzi kace práce, kdy £lov¥k sestavující testy prost°ednictvím gra ckého rozhraní jiº nemusí být blízce seznámen s testovaným softwarem, ani nemusí mít hluboké znalosti programovacího jazyka.
Gra cké rozhraní nabízí moºnost t°ídit jednotlivé akce pod skupiny testcase, £ímº umoº¬uje jejich logické rozd¥lení pro moºnost pokrývat konkrétní poºadavky konkrétním testcasem.
Gra cké rozhraní na základ¥ výb¥ru akcí vygeneruje json soubor, popisující jednotlivé zvolené akce a jejich parametry, z n¥hoº je následn¥ vytvo°en zdrojový .cpp soubor pro kompilaci se samotným testovacím prost°edím.
Na rozdíl od unit test prost°edí je zde tedy hlavním prvkem samotné gra cké prost°edí a nikoliv vyuºívání p°edde novaných maker ve zdrojových souborech, jako je tomu nap°íklad u CppUTest [GV20], £i GoogleTest [Goo19].
Vygenerovaný json soubor zárove¬ slouºí pro p°ímou identi kovatelnost jednotlivých test· v·£i jejich speci kaci.

Tento soubor m·ºe být kdykoliv znovu pouºit pro opakování testovacího procesu.

4.5.1 D¥lení na testcasy

Aby bylo moºné jednozna£n¥ identi kovat jednotlivé testy v·£i jejich dokumentované speci kaci a v·£i testovaným poºadavk·m, je kaºdá Akce p°i°azena konkrétnímu testcasu, £ímº je zaji²t¥no, ºe soubor Akcí dohromady vytvá°í jeden p°esn¥ identi kovatelný testcase.

4.6 B¥h

B¥h samotného programu se °ídí podle stavového diagramu znázorn¥ného na obrázku 4.4

Obrázek 4.4: B¥h testovacího prost°edí

4.6.1 Registrace

Prvním krokem, který testovací prost°edí provede, je ºe zaregistruje svou vlastní cyklickou funkci do ni²í vrstvy (pokud existuje) pop°ípad¥ do timeru, kterým má být testovaný software buzen. Tímto krokem pro zbytek systému za£íná p°ebírat místo, které v reálném produktu zastává testovaný software. Po tomto kroku ji º je ve²kerá dal²í £innost testovacího prost°edí závislá na zavolání této cyklické funkce, a v rámci ní vykonává ve²kerou £innost.

4.6.2 Inicializace

Inicializace je prvotní stav automatu, kterým se testovací prost°edí °ídí. Poté vrátí kontrolu zp¥t ni²í vrstv¥ a £eká na p°í²tí zavolání cyklické funkce, nebo´ následující stav m·ºe být £asov¥ náro£ný.
Tím stavem je £tení jednotlivých akcí, které byly p°edde novány uºivatelem prost°ednictvím gra ckého rozhraní. Ve chvíli, kdy jsou akce na£teny a uloºeny do dynamické pam¥ti v prioritní front¥, se cyklus op¥t ukon£uje a £eká se na dal²í zavolání cyklické funkce.
Dal²ím krokem je restart a inicializace samotné testované komponenty. Toho je docíleno zavoláním uºivatelem p°edde novaných funkcí test_sw::deInit() a následn¥ test_sw::init(). V rámci t¥chto funkcí by rovn¥º m¥la být reinicializována i knihovna mock·, £íºje simulován restart v²ech ostatních komponent, se kterými má testovaná komponenta prost°ednictvím rozhraní komunikovat.
Následujícím stavem uº je normalOperation, který obsahuje sv·j vlastní vnit°ní stavový automat.

4.6.3 Hlavní cyklus

V rámci hlavního cyklu ji º testovací prost°edí za£ne vyuºívat výpo£etní kapacitu na maximum. V rámci n¥j je kontrolován aktuáln¥ ub¥hnutý £as v rámci tohoto zavolání cyklické funkce testovacího prost°edí. Kontrola ni²í vrstv¥ se vrací aº v dob¥, kdy je dosaºena maximální doba, kterou má v systému komponenta povoleno vyuºít.
V rámci tohoto stavu je periodicky volán vnit°ní stavový automat.

Na£tení akce

V rámci tohoto stavu se na£te z fronty dal²í akce, která má být prost°ednictvím testovacího prost°edí vykonána jako dal²í.

Vykonání akce

V p°ípad¥, ºe simulovaný £as dosáhl £asu spu²t¥ní akce, je akce vykonána. Po jejím vykonání se p°í²tí stav nastavuje op¥ na £tení, aby akce, které mají být vykonány simultánn¥ skute£n¥ mohly být z hlediska virtuálního £asu simulace provedeny paraleln¥.

4.6.4 Inkrementace £asu

V p°ípad¥, ºe akce je²t¥ nem¥la být vykonána, p°ichází na °adu inkrementace £asu. V rámci té se inkrementuje jak virtuální £as simulace, tak se aktualizuje i £as reálný, který je pouºíván pro p°eru²ení simulace

Kapitola 5

Ov¥°ení návrhu °e²ení

Na základ¥ návrhu p°edstaveném v kapitole 4 bylo pro ov¥°ení konceptu implementováno testovací prost°edí a jeho funkcionalita ov¥°ena pro testování za°ízení pl¬ujícího po²adavky na SIL4 za°ízení podle normy EN 50128:2011 [EN511].

5.1 Jak software pou²ít

Pro vyu²ití navr²eného prost°edí je nutné v první °ad¥ provést ur£it¥ kroky pro jeho fungování se speci ckým projektem.

5.1.1 P°íprava

Pro funk£ní b¥h programu jsou zapot°ebí tyto softwarové moduly

- Testovaný software - softwarová komponenta, je² má být testována. Tato komponenta m·²e být vlo²ena jak formou statické knihovny, tak formou zdrojových soubor·. Vyu²ití statické knihovny grantuje jistotu, ²e

testované binární soubory jsou shodné s výsledným produktem. Zdrojové soubory jsou poté pot°ebné v p°ípad¥ pot°eby úpravy zdrojových soubor· pro pot°eby test·, nap°íklad p°idáním maker pro kontrolu pokrytí testy (branch coverage).

- Deklarace komunika£ních rozhraní mezi komponentami - hea- der soubory, obsahující deklarace funkcí rozhraní pro komunikaci mezi komponentami

- Mock knihovna - testerem de novaná funkcionalita jednotlivých funkcí rozhraní

- Ostatní komponenty - v p°ípad¥, ºe není pot°ebné mockovat funkcio- nalitu n¥jakého rozhraní, je moºné vyuºít originální komponentu

- De nice akcí - de nice jednotlivých událostí, které budou vyuºívány v pr·b¥hu testování

- Testovací prost°edí - prost°edí je implementováno v rámci této práce, b¥hem p°ípravy je v²ak nutné zade novat n¥kolik projektov¥ speci ckých funkcí (nap°íklad funkce pro restart testovaného softwaru a podobn¥)

Pro p°ípravu je tedy z pohledu testera zapot°ebí pouze zade novat Mock knihovnu, zade novat jednotlivé akce a nakonec zade novat n¥kolik projektov¥ speci ckých funkcí.
Ve chvíli, kdy jsou tyto £ásti zade novány, je moºné spustit podp·rný skript, který p°ipraví .json soubor, který obsahuje speci kace jednotlivých akcí, který bude poté vyuºívat gra cké rozhraní.

UºívánÍ

Ve chvíli, kdy je v²e p°ipraveno, je moºné jednodu²e spustit gra cké rozhraní. Zde si tester vybere, které akce, v jakém £ase b¥hu a s jakými parametry mají být vykonány.
Po jejich vytvo°ení je poté moºné pouze zkompilovat prost°edí spole£n¥ s £ástmi de novanými v 5.1.1.

Obrázek 5.1: Gra cké rozhraní

5.2 Omezení

P°i vyuºití navrºeného testovacího prost°edí je nutné zváºit omezení vyplývající ze samotné implementace.

5.2.1 Typ komponenty

Samotné testovací prost°edí je moºné aplikovat na libovolnou komponentu, která spl¬uje vertikální strukturu komponent a je buzena v pevných £asových intervalech niº²í vrstvou prost°ednictvím cyklické funkce.

5.2.2 Výpo£etní kapacita

Testovaná komponenta rovn¥º nesmí vyuºívat £as pro ni ur£ený naplno, nebo´ testovací prost°edí p°idává dodate£nou zát¥º v °ádu desítek operací v kaºdém aplika£ním cyklu. Vykonání akce poté závisí na výpo£etní náro£nosti

implementace samotného u^{o}ivatele, ov²em lze o£ekávat dal²í p°idanou zát¥^{o}. Proto musí být p°ed vyu^{o}itím zvá^{o}ena výpo£etní kapacita cílového hardwaru a její vyu^{o}ití p°i b¥hu bez test·.
Testovací prost°edí je p°ipraveno s o£ekáváním výrazn¥ vy²²í výpo£etní kapacity ne^{o} jsou po^{o}adavky testované komponenty a optimalizuje b¥h prám¥ za t¥chto podmínek.

5.2.3 Pam¥´

Stejn¥ jako v p°ípad¥ výpo£etní kapacity, i z hlediska pam¥´ové náro£nosti vyu^{o}ívá testovací prost°edí £ást pam¥ti za°ízení. Naprostou majoritu pam¥ti vyu^{o}ívají jednotlivé zade novné akce. V p°ípad¥ velkých testovacích sad m·^{o}e jít o nezanedbatelné mno^{o}ství. Testovaná komponenta proto nesmí vypl¬ovat jí ur£ený pam¥´ový prostor na maximum. V p°ípad¥ velmi malého nevyu^{o}itého prostoru je nutné rozd¥lit b¥h velkých testovacích sad na n¥kolik men²ích.

5.3 Srovnání s vyu^{o}itím unit test prost°edí

V této kapitole je zanalyzováno vyu^{o}ití navrhovaného testovacího prost°edí v kontrastu proti standardn¥ u^{o}ívanému unit test prost°edí CppUTest [GV20] pro integra£ní testování softwarových komponent.

5.3.1 P°íprava

Z hlediska p°ípravy je pro navr^{o}ené testovací prost°edí naimplementovat jednotlivé mock funkce s jejich hlub²í funkcionalitou. Tato £innost je shodná i pro CppUTest, nebo´ v p°ípad¥ vy²²í funkcionality na úrovni softwarové integrace podpora je podpora mock de novaná v prost°edí CppUTest nedostate£ná.

Dal²í implementa£ní £inností v rámci navr^{o}eného testovacího prost°edí je denice jednotlivých akcí, kterými je poté v rámci test· upravována a vy£ítána mock knihovna. De nice samotných akcí není v rámci prost°edí CppUTest p°ímo d¥lána. Ov²em za p°edpokladu, ^{o}e je CppUTest prost°edí vyu^{o}íváno

efektivn¥, je shodná funkcionalita implementována v rámci funkcí p°ístupové knihovny pro vy£ítání a úpravu mock· a obsahuje mírn¥ ni²í implementa£ní náro£nost.

Z hlediska p°ípravy samotného prost°edí je tedy implementa£ní náro£nost tém¥° shodná s mírnou p°evahou unit test prost°edí, zp·sobené tvorbou funkcionality p°ímou implementací funkcí, kde oproti tomu navr°ené testovací prost°edí implementuje potomky t°ídy Action pro shodnou funkcionalitu.

5.3.2 Tvorba testovacích sad

Následnou £inností je poté tvorba samotných testovacích sad. V této fázi se naplno projevují výhody navr°eného testovacího prost°edí.
V rámci této £innosti, která je objemem °ádov¥ náro£n¥j²í ne° samotná p°íprava prost°edí, ji° pracuje navr°ené testovací prost°edí pouze s gra ckým rozhraním, které poté automaticky generuje zdrojový kód. Díky mezikroku ve form¥ .json souboru je rovn¥° mo°né akce de novat i p°ímo zde.
Navr°ené testovací prost°edí se stará o standardní b¥h komponenty a stará se o korektní £asovou souslednost. Proto je mo°né jednotlivé akce vykonávat ji° pouze volbou parametru Releasepro jednotlivé akce.
CppUTest naopak vy°aduje p°ímou implementaci jednotlivých testovacích sad. V rámci této implementace je mimo jiné nutné dbát na p°esnou £asovou souslednost událostí, pro ka°dý test m¥°it £as a sledovat, po kolika voláních cyklické funkce má být jaká £innost vykonána.

V rámci tvorby testovacích sad se tedy projevuje výhoda navr°eného testovacího prost°edí oproti pouhého vyu°ití unit test prost°edí. Hlavní výhodou je výrazn¥ ni²í znalostní úrove¬ pot°ebná pro implementaci samotných testovacích sad, které jsou z hlediska práce testera majoritní £ástí práce. Toto je zaji²t¥no tím, °e samotná komponenta ji° b¥°í korektn¥, a jediné, co je nutné, je vygenerovat událost. V p°ípad¥ vyu°ití unit test prost°edí je nutné znát konkrétní software mnohem zevrubn¥ji a v°dy se dostávat do konkrétního stavu, kdy má být daná událost vykonána.

5.4 Prost°edí

V rámci implementace návrhu testovacího prost°edí bylo vyu°ito následujících vývojových prost°edí.

5.4.1 Gra cké prost°edí

IDE: QT Creator 4.11.0
Kompilátor: MinGW 7.3.0 - 64bit for C++

5.4.2 Simulace na PC

IDE: Eclipse 2019-06 (4.12.0), build 20190614-1200
Kompilátor: Gcc 4.7.2

5.4.3 Simulace na HW

MCU: STM32F417ZG
IDE: IAR Embedded workbench IDE 7.20.5.7624
Kompilátor: IAR ANSI C/C++ Compiler V7.20.5.7591/W32 for ARM
Linker: IAR ELF Linker V7.20.5.7591/W32 for ARM

Kapitola 6

Diskuse

Testovací prost°edí na základ¥ speci k v kapitole 4 bylo implementováno a veri kováno p°i testování za°ízení z °ady xCM rmy SIEMENS.

6.1 Plánované vyu^{o}ití

V rámci testování bylo ov¥°eno spln¥ní podmínek stanovených normou EN 50128:2011 [EN511] pro nejvy^{22}í SIL4 úrove« bezpe£nosti vývoje.
Prost°edí v^{2}ak v dob¥ vydání práce nepro^{2}lo o ciálním valida£ním procesem. Jeho vyu^{o}ití je plánováno jako o ciální veri ka£ní prost°edí pro nadcházející verze za°ízení z °ady xCM spole£nosti SIEMENS.

6.2 Dal^{2}í mo^{o}nosti vyu^{o}ití

Navr^{o}ené testovací prost°edí je p°ipraveno pro maximální generalizaci testovacího procesu. Díky mo^{o}ným kon guracím v rámci p°ípravy prost°edí pro daný projekt, jako je volitelná hloubka dosa^{o}ené softwarové integrace £i nerestriktivní povaha de novaných Akcí, které sice mají speci kované plánované vyu^{o}ití, av^{2}ak toto vyu^{o}ití není mandatorní.
V pr·b¥hu p°ípravy prost°edí zku^{2}eným testerem se znalostí problematiky

daného projektu tak není tento tester omezován nad nezbytnou míru, čímž je schopen implementovat i funkcionalitu, která nebyla v průběhu navrhování testovacího prostředí předpokládána.

6.2.1 Další projekty

Obecně lze říci, že navržené testovací prostředí není vhodné pouze pro projekty svázané normou EN 50128:2011 [EN511], ale je využitelné pro většinu projektů, kde je využit vertikálně vrstvený multikomponentní systém a zároveň je aplikováno testování na úrovni komponent v rámci (ať už plné či jen částečné) softwarové integrace, a to na základě limitace normou či na základě žádoucího zvýšení garance funkcionality výsledného zařízení a odhalení chyby dříve než v produkci [BRD+16].
Pro testování softwaru, na nějž nejsou kladeny požadavky z hlediska softwarové integrace je rovněž možné navržené prostředí využít, nicméně nelze obecně garantovat výhodnost oproti využití vlastní minimalistické nadstavby nad Unit test prostředí. Toto je nutné zvážit pro každý specifický projekt samostatně.

6.2.2 Využití v průběhu vývoje

Z principu běhu samotného testovacího prostředí je možné využití nejen v rámci verifikačního procesu, ale rovněž již v průběhu vývoje.
Jelikož je prostředí schopné plně nahradit z pohledu testované komponenty zbytek systému, je prostředí rovněž vhodné pro testování implementované funkcionality již samotnými vývojáři, a to i v době, kdy zbytek (či část) systému ještě v rámci vývoje není schopný korektní interakce.
Stejně tak je prostředí možné využít v době, kdy cílový hardware ještě není připraven a tak vývojáři nemají možnost testovat funkcionalitu své komponenty, neboť prostředí nabízí možnost integrovat pouze části softwaru, které přímo na hardware závislé nejsou, a nahradit pouze komponenty přímo na hardware závislé.

Z obou zmíněných důvodů zároveň prostředí nabízí přípravu testovacích sad i pro samotné testery v době, kdy ještě zbytek systému (software či hardware) není připraven pro integraci. Vykonání těchto testů samozřejmě není z pohledu normy EN 50128:2011 [EN511] validní, nicméně specifikace a definice jednotlivých testů již může být díky tomu připravována paralelně s

dodělávaním opožděných (proti testované komponentě) částí systému a tím je docíleno výrazného urychlení celého testovacího procesu.

6.3 Budoucí vývoj

V rámci práce byl implementací ověřen koncept navrženého testovacího prostředí, avšak samotná implementace v této fázi ještě plně nevyužívá svůj potenciál, především z hlediska uživatelské přívětivosti.

6.3.1 GUI

Grafické rozhraní plní veškerou požadovanou funkcionalitu, avšak momentálně neobsahuje podchycení chybových stavů a tak vyžaduje pouze korektní chování uživatele.
Rovněž momentálně neobsahuje možnost jednoduchého nahrávání již dříve vygenerovaných souborů se specifikací testovacích sad, o kterou bude rozhraní rozšířeno.

6.3.2 Příprava prostředí

Prostředí bylo navrženo s extensivním využitím batch skriptů pro generování zdrojových souborů na základě uživetelem definovaných specifik. Tento přístup byl zvolen pro zajištění možnosti využití na jakémkoliv zařízení.
V současnosti se ovšem jeví jako zbytečně složitý a proto je v plánu přikročit k uživatelsky přívětivější formě přípravy prostředí pro využití na daném projektu.

6.3.3 Dokumentace

V momentálním stavu neexistuje oficiální dokumentace k implementovanému prostředí, neboť ještě není plně připraveno pro aktivní využívání.
V rámci budoucí práce je ovšem nutné vytvořit uživatelský manuál, který musí být připraven pro splnění podmínek verifikačního procesu.

Kapitola 7

Závěr

V rámci práce bylo dokázáno, že při stanovení určitých podmínek na strukturu testovaného softwaru lze vytvořit obecné testovací prostředí pro integrační testování na úrovni komponent.
Bylo vytvořeno testovací prostředí na základě specifik nastíněných v samotné práci, které ověřilo funkčnost konceptu při testování zařízení firmy SIEMENS, spadající pod bezpečnostní úroveň SIL4 normy EN 50128:2011 [EN511].
Bylo ověřeno, že testovací prostředí podporuje všechny požadavky vyplývající ze zmíněné normy a je tedy vhodné pro využití na projektech s obdobnými regulatorními požadavky.
Rovněž bylo ukázáno, že využití navrženého testovacího prostředí je z hlediska implementace i náročnosti na kvalifikaci uživatele (testera) jednodušší, než využití vlastní nadstavby nad standardně užívané unit test knihovny.

Literatura

[BDH+98] Manfred Broy, Anton Deimel, Juergen Henn, Kai Koskimies, František Plášil, Gustav Pomberger, Wolfgang Pree, Michael Stal, and Clemens Szyperski, *What characterizes a (software) component?*, Software-Concepts & Tools **19** (1998), no. 1, 49–56.

[BG92] T. E. Bihari and P. Gopinath, *Object-oriented real-time systems: concepts and examples*, Computer **25** (1992), no. 12, 25–32.

[BRD+16] Miroslav Bureš, Miroslav Renda, Michal Doležel, et al., *Efektivní testování softwaru: klíčové otázky pro efektivitu testovacího procesu*, Grada Publishing as, 2016.

[CSVC10] Ivica Crnkovic, Severine Sentilles, Aneta Vulgarakis, and Michel RV Chaudron, *A classification framework for software component models*, IEEE Transactions on Software Engineering **37** (2010), no. 5, 593–615.

[EN511] *Railway applications - Communication, signalling and processing systems - Software for railway control and protection systems*, Standard, European Committee for Electrotechnical Standardization, Avenue Mamix 17,B - 1000 Brussels, June 2011.

[Goo19] Google, *Googletest - Google Testing and Mocking Framework*, October 2019.

[GV20] James Grenning and Bas Vodde, *CppUTest unit testing and mocking framework for C/C++*, May 2020.

[LMW95] Y. . S. Li, S. Malik, and A. Wolfe, *E cient microarchitecture modeling and path analysis for real-time software*, Proceedings 16th IEEE Real-Time Systems Symposium, 1995, pp. 298–307.

[LW07] Kung-Kiu Lau and Zheng Wang, *Software component models*, IEEE Transactions on software engineering **33** (2007), no. 10, 709–724.

[TS06a] N. Tillmann and W. Schulte, *Mock-object generation with behavior*, 21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06), 2006, pp. 365–368.

[TS06b] _____, *Unit tests reloaded: parameterized unit testing with symbolic execution*, IEEE Software **23** (2006), no. 4, 38–47.

[WPC01] Ye Wu, Dai Pan, and Mei-Hwa Chen, *Techniques for testing component-based software*, Proceedings Seventh IEEE International Conference on Engineering of Complex Computer Systems, IEEE, 2001, pp. 222–232.

ZADÁNÍ DIPLOMOVÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Steidl** Jméno: **Richard** Osobní číslo: **420389**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra měření**

Studijní program: **Otevřená informatika**

Specializace: **Počítačové inženýrství**

II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce:

Testovací prostředí pro integrační testování real-time software složeného z více komponent

Název diplomové práce anglicky:

Test environment for integration testing of component-based real-time software

Pokyny pro vypracování:

1. Seznamte se s integračním testováním real-time software sestaveného z více modulů především s ohledem na normu EN 50128. Seznamte se s konkrétní aplikací složené z aplikační a základové vrstvy.
2. Navrhněte a implementujte testovací prostředí pro testování aplikační vrstvy a simulující základovou vrstvu.
3. Součástí testovacího prostředí bude GUI aplikace pro ovládání simulátoru, simulátor a knihovna pro komunikaci s aplikační vrstvou.
4. Zdokumentujte vytvořený software a ověřte funkčnost na zařízeních řady xCM společnosti Siemens AG určených pro železniční automatizaci a splňujících bezpečnostní stupeň SIL 4.

Seznam doporučené literatury:

[1] EN 50128: Railway applications - Communication, signalling and processing systems. 2011.
[2] Bures, Renda, Dolezel: Efektivní testování softwaru. Grada 2016
[3] Ye Wu, Dai Pan, Mei-Hwa Chen: Techniques for testing component-based software. IEEE 2001

Jméno a pracoviště vedoucí(ho) diplomové práce:

Ing. Joel Matějka, katedra řídicí techniky FEL

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) diplomové práce:

Datum zadání diplomové práce: **20.09.2019** Termín odevzdání diplomové práce: _____

Platnost zadání diplomové práce:
do konce letního semestru 2020/2021

Ing. Joel Matějka podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	prof. Mgr. Petr Páta, Ph.D. podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Diplomant bere na vědomí, že je povinen vypracovat diplomovou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací. Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v diplomové práci.

. _____ Podpis studenta

Datum převzetí zadání