

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, Integrovaný testování, Úroveň integrity bezpečnosti, Bezpečnostně-kritický systém, C++, Vestavný systém, Železniční infrastruktura, Železniční automatizace

**Vedoucí:** Ing. Joel Matějka  
Katedra řídicí 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

<b>1 Úvod</b>	<b>1</b>	3.3 Obecnost .....	19
<b>2 Definice a teoretický základ</b>	<b>5</b>	3.4 Minimální požadavky na uživatele	19
2.1 Multikomponentní systém .....	5	<b>4 Návrh testovacího prostředí</b>	<b>21</b>
2.1.1 Specifika komponenty .....	7	4.1 Integrace .....	22
2.2 Real-time systém .....	8	4.1.1 Plné nahrazení systému .....	22
2.3 Testování multikomponentního systému .....	8	4.1.2 Plná integrace se systémem .	23
2.3.1 Unit level .....	10	4.1.3 Částečné nahrazení systému .	24
2.3.2 Component level .....	10	4.2 Real time .....	24
2.3.3 System level .....	10	4.2.1 Reálný a virtuální čas .....	25
2.3.4 Integrační testování .....	11	4.3 Testovací strategie .....	25
2.3.5 Požadavky normy EN 50128:2011 .....	11	4.4 Struktura .....	26
2.3.6 Mock .....	14	4.4.1 Mock .....	27
<b>3 Požadavky na testovací prostředí</b>	<b>17</b>	4.4.2 Mock knihovna .....	27
3.1 Možnost softwarové integrace...	17	4.4.3 Akce .....	27
3.2 Provázání požadavek-test .....	18	4.4.4 Paměťová struktura .....	28
		4.5 Grafické rozhraní .....	28
		4.5.1 Dělení na testcasy .....	29
		4.6 Běh .....	29

4.6.1 Registrace . . . . .	30	5.4.3 Simulace na HW . . . . .	38
4.6.2 Inicializace . . . . .	30	<b>6 Diskuse</b>	<b>39</b>
4.6.3 Hlavní cyklus . . . . .	30	6.1 Plánované využití . . . . .	39
4.6.4 Inkrementace času . . . . .	31	6.2 Další možnosti využití . . . . .	39
<b>5 Ověření návrhu řešení</b>	<b>33</b>	6.2.1 Další projekty . . . . .	40
5.1 Jak software použít . . . . .	33	6.2.2 Využití v průběhu vývoje . . .	40
5.1.1 Příprava . . . . .	33	6.3 Budoucí vývoj . . . . .	41
5.2 Omezení . . . . .	35	6.3.1 GUI . . . . .	41
5.2.1 Typ komponenty . . . . .	35	6.3.2 Příprava prostředí . . . . .	41
5.2.2 Výpočetní kapacita . . . . .	35	6.3.3 Dokumentace . . . . .	42
5.2.3 Paměť . . . . .	36	<b>7 Závěr</b>	<b>43</b>
5.3 Srovnání s využitím unit test prostředí . . . . .	36	<b>Literatura</b>	<b>45</b>
5.3.1 Příprava . . . . .	36	<b>Zadání práce</b>	<b>47</b>
5.3.2 Tvorba testovacích sad . . . . .	37		
5.4 Prostředí . . . . .	37		
5.4.1 Grafické prostředí . . . . .	38		
5.4.2 Simulace na PC . . . . .	38		

## Obrázky

2.1 Dělení multikomponentního systému [CSVC10] .....	6
2.2 Požadavky na komponenty podle [EN511]. R - Recommended, HR - Highly recommended, M - mandatory	7
2.3 Úrovně testování.....	9
2.4 Požadavky na test softwaru podle [EN511]. R - Recommended, HR - Highly recommended, M - mandatory .....	12
2.5 Požadavky pro měření pokrytí podle [EN511]. R - Recommended, HR - Highly recommended, M - mandatory .....	13
3.1 Vložení testovací vrstvy do zbytku systému .....	18
3.2 Rozdělení práce na testování softwaru .....	20
4.1 Nahrazení systému .....	23
4.2 Integrace do systému .....	24
4.3 Architektura testovacího prostředí	26
4.4 Běh testovacího prostředí .....	29

5.1 Grafické rozhraní .....	35
-----------------------------	----

# Tabulky







# Kapitola 1

## Úvod

Při vývoji bezpečnostně kritického softwaru je kladen velký důraz na celý vývojový proces, jehož významnou součástí také testování, ověřující korektní chování vyvinutého softwaru.

Celý vývojový proces je pro bezpečnostně kritická zařízení zároveň regulován požadavky konkrétních norem, které popisují celý proces vývoje a kladou specifické požadavky, jakým způsobem mají být jednotlivé části 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 verifikační činnost při vývoji zařízení pro železniční automatizaci EN 50128:2011 [EN511].

Pro testování projektů svázaných požadavky norem je využití existujících a všeobecně rozšířených testovacích prostředí jako je GoogleTest [Goo19] či CppUTest [GV20] možné pouze za splnění dodatečných podmínek.

Tato prostředí zároveň nejsou uzpůsobena pro testování za mandatorních podmínek stanovených normami, což přináší dodatečnou implementační náročnost pro testera.

Cílem této práce je proto navržení testovacího prostředí pro testování na úrovni komponent v plné softwarové integraci, které splňuje podmínky pro testování jakéhokoliv softwaru vyvíjeného v souladu s podmínkami pro nejvyšší úroveň bezpečnosti SIL4 (Safety integrity level) definované normou EN 50128:2011 [EN511].

Výstupy této práce je však možné využít pro jakýkoliv systém splňující obdobné regulatorní požadavky.

Z důvodu požadavků na plnou softwarovou integraci vyplývajících ze zmi-

ňované normy a na komplexitu jednotlivých komponent nejsou standardně užívaná prostředí jako CppUTest [GV20] či GoogleTest [Goo19] bez významné vlastní nadstavby pro tuto činnost vhodná.

Významná část této nadstavby je ovšem generalizovatelná. V rámci návrhu testovacího prostředí v této práci je právě tato generalizovatelná část implementována.

Prostředí je navrženo pro testování multikomponentního systému, jehož jednotlivé komponenty jsou vrstveny vertikálně, a je tedy ve vztahu dvou komponent vždy jedna vyšší (aplikační) a jedna nižší (základová).

Navržené prostředí je vytvořeno s ohledem na real-time aplikaci testovaného zařízení.

Je očekáváno, že testovaná komponenta je volána z nižší části systému v předdefinovaných časových intervalech a pro její běh je vyhrazena pouze omezená doba. Proto má navržené testovací prostředí možnost sledovat rovněž reálný čas a na jeho základě je schopné přerušit svou činnost s předdefinovanou periodou, aby nebyl v integraci přesažen čas vyhrazený testované komponentě. Následně je prostředí schopné na svou činnost navázat ve chvíli, kdy opět nastane čas pro jeho běh.

Důraz je dále kladen na možnost běhu testů v integraci s cílovým hardwarem, neboť nejnižší komponenty systému mohou být na hardwaru závislé. Z tohoto důvodu je prostředí optimalizováno pro minimalizaci nároků na paměťovou i výpočetní kapacitu.

Pro dosažení maximálního přiblížení průběhu testů výsledné aplikaci je kladen důraz na možnost využití stejných kompilátorů a kompilačních konfigurací, jaké využívá finální software a navržené prostředí je z tohoto důvodu zcela nezávislé na externích knihovnách včetně standardních.

Dalším významným specifickým navrženého prostředí je optimalizace lidských zdrojů v rámci vývoje produktu.

Testovací prostředí je proto navrženo tak, aby samotné testování sestávalo ze dvou fází.

První fází je příprava testovacího prostředí v závislosti na specifických parametrech konkrétní testované softwarové komponenty. Tato fáze vyžaduje práci člověka seznámeného s problematikou cílového softwaru.

Druhou fází je poté tvorba samotných testovacích sad a jejich specifikací. Ta již probíhá výhradně přes grafické rozhraní testovacího prostředí, čímž jsou redukovány požadavky na testera z hlediska hlubšího porozumění technickým specifickým projektů.

Navržené testovací prostředí je v rámci práce implementováno a funkcionality navrženého konceptu je ověřena při testování zařízení z řady xCM společnosti SIEMENS, splňující bezpečnostní stupeň SIL4 normy EN 50128:2011 [EN511],

kteřé se zabývá železniční automatizací.

Diplomová práce je po úvodu členěna následovně:

Ve druhé kapitole jsou představeny pojmy nezbytné pro pochopení práce.

Třetí kapitola obsahuje požadavky na navržené testovací prostředí, které vyplývají ze zadání práce a z požadavků normy EN 50128:2011 [EN511].

Čtvrtá kapitola obsahuje popis navrženého testovacího prostředí a vysvětluje jeho specifika.

Pátá kapitola je věnována přímé implementaci navrženého testovacího prostředí, srovnává ho se standardně využívanými prostředími a vysvětlují se v ní omezení dané implementace.

Šestá kapitola rozebírá praktické využití implementovaného testovacího prostředí, ukazuje plánované využití v rámci vývoje zařízení pro železniční automatizaci firmy SIEMENS a rovněž poskytuje náhled na možnost využití na dalších projektech. V rámci této kapitoly je rovněž ukázán směr, kterým se budoucí vývoj bude ubírat.





## Kapitola 2

### Definice a teoretický základ

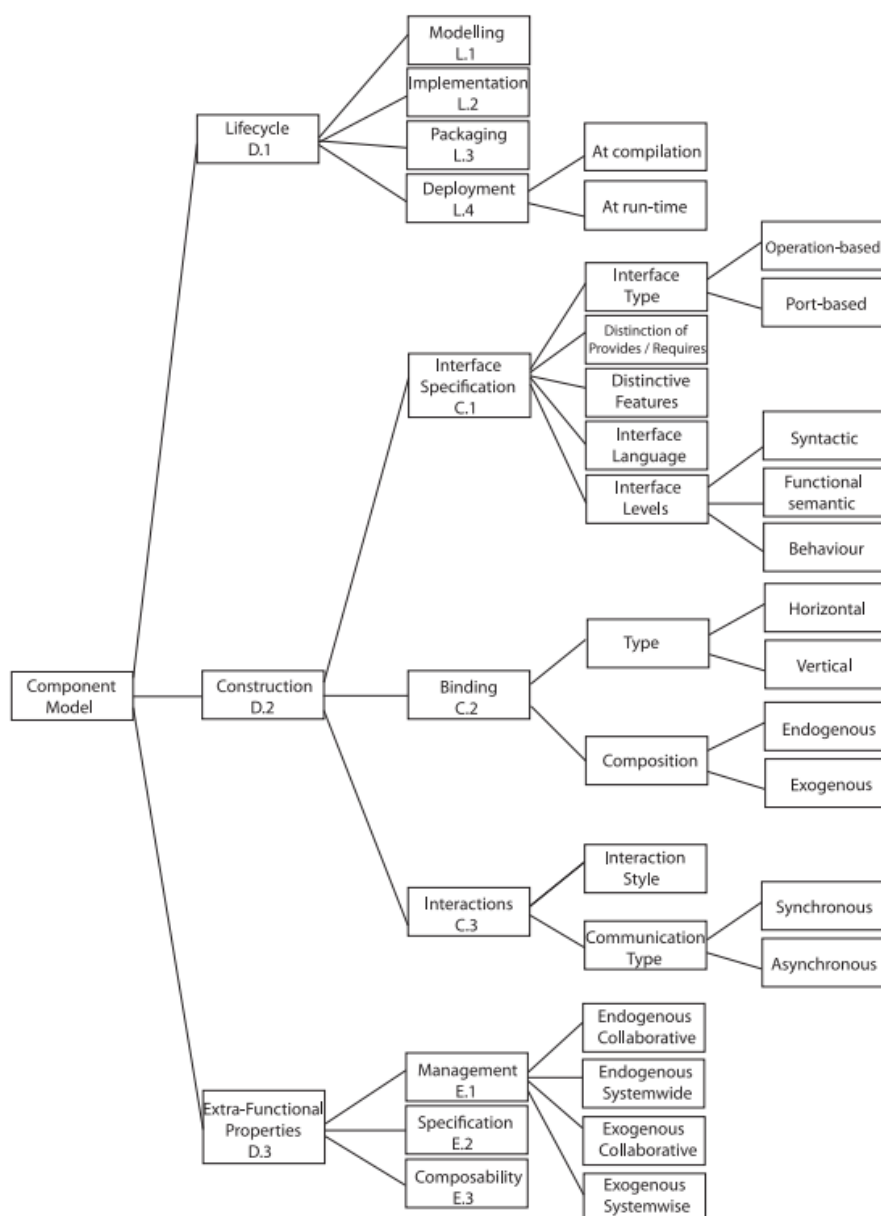


#### 2.1 Multikomponentní systém

Multikomponentní systém je takový, který se skládá z většího množství na sobě nezávislých komponent. Tyto komponenty mezi sebou komunikují prostřednictvím přesně definovaných rozhraní a vzájemně si poskytují služby jim určené.

Příklon k multikomponentnímu systému je z pohledu vývoje žádoucí, neboť jednotlivé komponenty jsou znovupoužitelné v jiných projektech a jejich vývoj je na sobě (vyjma zadaného rozhraní) zcela nezávislý [LW07].

Obecný multikomponentní systém lze dělit na mnoho kategorií, na základě skládání jednotlivých komponent, jejich vzájemné komunikaci a dalších parametrech, které znázorňuje obrázek 2.1 [CSVC10].



**Obrázek 2.1:** Dělení multikomponentního systému [CSVC10]

Z pohledu normy, na niž má brát práce ohled, je však definice komponenty výrazně skoupější. Norma EN 50128:2011 [EN511] ve volném překladu definuje komponentu následovně:

komponenta je část softwaru, která má přesně zdefinovaná rozhraní a své chování s ohledem na softwarovou architekturu a design a splňuje následující kritéria:

- je navržena podle 2.2
- pokrývá určitou podmnožinu požadavků na software
- je přesně identifikovatelná a je nezávislá, a nebo je součástí skupiny komponent, která je nezávislá.

Table A.20 – Components

TECHNIQUE/MEASURE	Ref	SIL 0	SIL 1	SIL 2	SIL 3	SIL 4
1. Information Hiding	D.33	-	-	-	-	-
2. Information Encapsulation	D.33	R	HR	HR	HR	HR
3. Parameter Number Limit	D.38	R	R	R	R	R
4. Fully Defined Interface	D.38	R	HR	HR	M	M
Requirement: 1) Information Hiding and encapsulation are only highly recommended if there is no general strategy for data access.						
NOTE Technique/measure 4 is for Internal Interfaces.						

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

Z důvodu nepřesné všeobecně uznávané definice komponenty [BDH<sup>+</sup>98], [WPC01], [EN511] se práce bude zabývat pouze specifickým typem komponent. Multikomponentní systém v našem chápání bude takový, jehož komponenty splňují specifika nastíněná v kapitole 2.1.1.

Tyto komponenty budou v rámci testovaného systému vrstvené, kdy ve vztahu dvou komponent je vždy jedna nižší (základová) a jedna vyšší (aplikační), a tedy budou reprezentovat vertikální typ skládání systému na 2.1 [CSVC10].

### 2.1.1 Specifika komponenty

- Komponenta reprezentuje jeden logický či organizační proces
- Dělení na komponenty je hrubší než na jednotlivé třídy. Může se skládat z více vzájemně provázaných tříd.
- Komponenta se může skládat z více (sub) komponent.

- Komponenta komunikuje s ostatními komponentami prostřednictvím přesně definovaného rozhraní.
- Komponenta je nezávislá na verzi projektu. Může být samostatně vylepšována na novou verzi.
- Je navržena podle požadavků na SIL4 projekt znázorněnými v 2.2

Tato specifikace volně vychází z parametrů definovaných Antonem Deimelem [BDH<sup>+</sup>98] a je doplněna o specifické požadavky normy EN 50128:2011 [EN511].

## 2.2 Real-time systém

Za real-time systém považujeme jakýkoliv systém, který musí interagovat s vnějším světem s určitou periodou či reagovat určitou rychlostí na externí vlivy. Zkráceně lze tedy říci, že je to jakýkoliv systém, který pro své řízení využívá reálný čas a garantuje jeho dodržení [LMW95], [BG92].

V našem systému definovaném v 2.1 je specifikace aplikace reálného času dosaženo prostřednictvím periodického volání určité funkce (cyclicFunction) ze základové komponenty, která je vstupní branou do komponenty aplikační. Tato aplikační komponenta musí v definovaném čase vrátit řízení běhu programu zpět komponentě základové, tedy dodržet časový slot vyhrazený pro její běh.

## 2.3 Testování multikomponentního systému

Testování požadavků se skládá ze třech základních stupňů. Jedná se o Unit test 2.3.1, Component test 2.3.2 a System test 2.3.3.

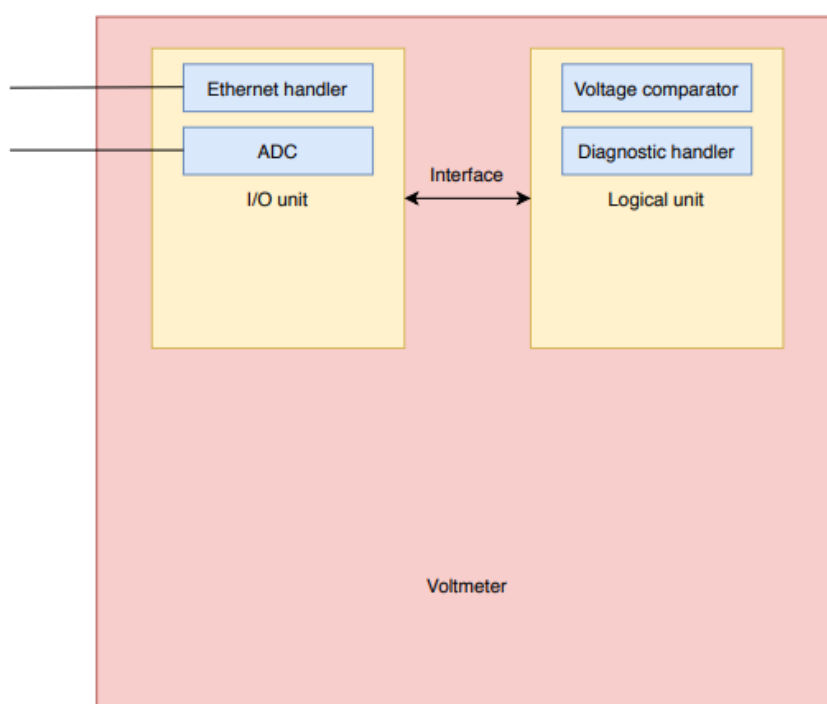
Každá z úrovní přináší rozdílnou míru integrace a tím detekce potenciálních problémů spojených s interakcí mezi jednotlivými částmi programu. Opačným směrem proti míře integrace pak jde samotná testovatelnost programu. Čím je testovaná část programu ucelenější, tím je složitější otestovat všechny kombinace vnitřních stavů, a rovněž může být nemožné pro účel testu upravovat hodnoty vnitřních (zapouzdřených) proměnných.

Všechny úrovně ovšem spojuje snaha o co nejbližší přiblížení se reálné aplikaci,



kterého lze dosáhnout na dané úrovni testování. Toho lze dosáhnout několika cestami, kde mezi nejvýznamější patří testování již předkompilovaných objektů testovaného softwaru, aby se zamezilo ovlivnění kompilace způsobené čistě testovacím prostředím.

Na obrázku 2.3 jsou znázorněny jednotlivé úrovně programu a na jaké úrovni jsou poté testovány. Modře je úroveň Unit testu, žlutě Component testu a červeně poté System test.



**Obrázek 2.3:** Úrovně testování

Systém znázorněný na obrázku 2.3 znázorňuje aplikaci voltmetru, složeného ze dvou komponent:

- **Vstupně-výstupní jednotka**, která čte hodnoty z AD převodníku a prostřednictvím předdefinovaného rozhraní tyto hodnoty poskytuje Logické jednotce. Dále se stará o ethernetovou komunikaci, prostřednictvím níž posílá datové telegramy, které připraví Logická jednotka a poskytne je opět prostřednictvím rozhraní.

- **Logická jednotka** přijímá data AD převodníku od Vstupně-výstupní a vyhodnocuje převod. Následně připraví diagnostickou zprávu a poskytne ji Vstupně-výstupní jednotce pro odeslání po ethernetu.

### ■ 2.3.1 Unit level

Unit level testování je nejjemnější standardně využívanou úrovní testů. Sleduje reakce dílčí části programu (třídy či ekvivalentu) na vstupní parametry funkcí, popřípadě na návratové hodnoty externích funkcí.

V této úrovni je tester schopen simulovat téměř libovolné chování volaných funkcí, a rovněž modifikovat všechny vnitřní hodnoty testované části programu.

Tato úroveň testování je široce podporována mnohými testovacími prostředími, jelikož je téměř nezávislá na konkrétním projektu. Existuje řada vysoce optimalizovaných prostředí pro tutu úroveň, jmenovitě GoogleTest [Goo19], CppUtest [GV20] a mnoho dalších.

### ■ 2.3.2 Component level

Component level testuje jednotlivé komponenty, které byly zdefinovány v 2.1.1. Sleduje chování jedné uzavřené softwarové části programu.

Pro simulaci externích události je využíváno striktně její externí rozhraní. Tester již nemá možnost zasahovat do vnitřních struktur samotné komponenty, ani možnost přistupovat k jednotlivým subkomponentám, vnitřním proměnným a podobně.

Tato úroveň nabízí již významnou míru integrace, neboť je komponenta navržena tak, aby byla v maximální míře uzavřená sama do sebe, a díky tomu je na této úrovni testována většina vzájemných závislostí v rámci komponenty.

### ■ 2.3.3 System level

System level nabízí nejvyšší úroveň integrace, na této úrovni se již testuje finální produkt (systém) a sleduje se pouze jeho vnější interakce se světem.

Kontroluje se tu integrace jednotlivých komponent a jejich vzájemné ovlivnění.

Testovaná funkcionální se částečně překrývá i s testy na úrovni komponent, avšak v této úrovni testy nejdou do takové hloubky, neboť z důvodu přístupu čistě přes vnější rozhraní není možné kontrolovat veškeré vnitřní stavy.

#### ■ 2.3.4 Integrovaní testování

Částečně nezávisle na předchozích třech úrovních testování je dále test integrovaní. Integrací se rozumí kombinace co největšího množství reálně využívané aplikace při běhu testů aby byl omezen případný nežádoucí vzájemný vliv jednotlivých softwarových modulů. Tento vliv se může objevit například při správě paměti, popřípadě při využívání externích zdrojů (kupříkladu omezená kapacita komunikační linky).

#### ■ 2.3.5 Požadavky normy EN 50128:2011

Pro testovaný software s ohledem na normu EN 50128:2011 [EN511] musí být mimo jiné splněny požadavky vyplývající z 2.4 a 2.5.

Table A.5 – Verification and Testing (6.2 and 7.3)

TECHNIQUE/MEASURE	Ref	SIL 0	SIL 1	SIL 2	SIL 3	SIL 4
1. Formal Proof	D.29	-	R	R	HR	HR
2. Static Analysis	Table A.19	-	HR	HR	HR	HR
3. Dynamic Analysis and Testing	Table A.13	-	HR	HR	HR	HR
4. Metrics	D.37	-	R	R	R	R
5. Traceability	D.58	R	HR	HR	M	M
6. Software Error Effect Analysis	D.25	-	R	R	HR	HR
7. Test Coverage for code	Table A.21	R	HR	HR	HR	HR
8. Functional/ Black-box Testing	Table A.14	HR	HR	HR	M	M
9. Performance Testing	Table A.18	-	HR	HR	HR	HR
10. Interface Testing	D.34	HR	HR	HR	HR	HR
Requirements:						
1) For software Safety Integrity Levels 3 and 4, the approved combination of techniques is 3, 5, 7, 8 and one from 1, 2 or 6.						
2) For Software Safety Integrity Level 1 and 2, the approved combinations of techniques is 5 together with one from 2, 3 or 8.						
NOTE 1 Techniques/measures 1, 2, 4, 5, 6 and 7 are for verification activities.						
NOTE 2 Techniques/measures 3, 8, 9 and 10 are for testing activities.						

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

Table A.21 – Test Coverage for Code

Test coverage criterion	Ref	SIL 0	SIL 1	SIL 2	SIL 3	SIL 4
1. Statement	D.50	R	HR	HR	HR	HR
2. Branch	D.50	-	R	R	HR	HR
3. Compound Condition	D.50	-	R	R	HR	HR
4. Data flow	D.50	-	R	R	HR	HR
5. Path	D.50	-	R	R	HR	HR

Requirements:

- For every SIL, a quantified measure of coverage shall be developed for the test undertaken. This can support the judgment on the confidence gained in testing and the necessity for additional techniques.
- For SIL 3 or 4 test coverage at component level should be measured according to the following:
  - 2 and 3; or
  - 2 and 4; or
  - 5
 or test coverage at integration level should be measured according to one or more of 2, 3, 4 or 5.
- Other test coverage criteria can be used, given that this can be justified. These criteria depend on the software architecture (see Table A.3) and the programming language (see Table A.15 and Table A.16).
- Any code which it is not practicable to test shall be demonstrated to be correct using a suitable technique, e.g. static analysis from Table A.19.

NOTE 1 Statement coverage is automatically achieved by items 2 to 5.

NOTE 2 The test coverage criteria in this table are used for structure-based (code-based, white box) testing. Techniques/measures for functional (specification-based, black box) testing are given in Table A.14.

NOTE 3 A high percentage of coverage is usually difficult to achieve. The use of test case execution from boundary values (D.4) and equivalence classes and input partition testing (D.18) can enable a sufficient coverage to be achieved with a smaller number of tests.

NOTE 4 The difference between 2 and 3 depends in practice on the level of the programming language and the use of compound conditions. When single conditions are used only, for example as a result of compilation, 2 and 3 are considered identical.

**Obrázek 2.5:** Požadavky pro měření pokrytí podle [EN511]. R - Recommended, HR - Highly recommended, M - mandatory

Požadavky v tabulce 2.4 zahrnují rovněž verifikační činnost, která nespadá pod přímou činnost testera a není vyhodnocována v rámci běhu samotných testů.

V rámci testování projektu na úrovni SIL4 je z pohledu testera zapotřebí být schopen měřit pokrytí softwaru v rámci běhu testů, a to na základě 2.5. Tohoto požadavku je mimo jiné dosaženo měřením *Branch coverage* v rámci běhu integračních softwarových testů, tedy jednotlivé komponenty musí být pro měření pokrytí testovány společně se všemi ostatními komponentami systému a test musí projít každým příkazem skoku v programu. Dalším mandatorním požadavkem na testera je běh testů ve formě black-box pro test funkcionality komponent.

### ■ 2.3.6 Mock

Souhrnně značí mock implementaci funkcionality netestovaných částí systému, respektive implementaci funkcí rozhraní, které je zdefinované pro komunikaci s danou částí systému, pro simulaci interakce s testovanou částí softwaru (funkce, třída, komponenta, ...).

Tato implementace ostatních částí systému je zpravidla výrazně jednodušší a méně komplexní, než je implementace reálných komponent, neboť má tester nad mocky plnou kontrolu a může vyloučit například chybové chování.

Mock navíc poskytuje přístup k hodnotám, s nimiž jsou jednotlivé funkce rozhraní z testované části softwaru volány a zároveň poskytující možnost upravovat návratové hodnoty, díky čemuž je tester schopen nejen sledovat chování testované komponenty, ale rovněž simulovat chování zbytku systému.

### ■ Unit test mock

Pro Unit test úroveň je takřka výhradně využíváno funkcionality Mock implementace [TS06b], [TS06a].

Unit test prostředí jako CppUTest [GV20] či GoogleTest [Goo19] nabízejí pro implementaci mocků rozsáhlou podporu, kdy je možné prostřednictvím předdefinovaných maker sledovat, kdy byla jaká funkce volána a s jakými parametry, a zároveň nastavovat návratové hodnoty.

Nastavení návratových hodnot však musí být vždy explicitně definováno, neboť jsou prostředí určena primárně pro Unit testování, kdy se neočekává delší běh testované části softwaru a mnohonásobné volání funkcí rozhraní.

### ■ Component test mock

Implementace mocků je však využitelná i pro testování na úrovni komponent. Z důvodu vyšší integrace je však již nutné dbát na detailnější implementaci mocků.

V rámci testů komponent již není sledováno pouze volání jedné funkce a reakce testovaného softwaru. Jedná se již o komplexnější funkcionality, kdy sledujeme reakce na větší množství přístupů do ostatních komponent, a tedy mock implementace musí být schopna si předávat stavy rovněž mezi jednotlivými voláními.

Jednoduchým příkladem nezbytnosti hlubší implementace mocků budiž následující příklad:

Nižší (netestovaná) komponenta přijímá data po ethernetové lince a ukládá si je do bufferů, mapované na základě MAC adresy odesílatele.

Vyšší vrstva poté využije funkci rozhraní `void writeEth(MAC_address, data)`. Následně využije opět funkci rozhraní `data getEthResponse()`.

V této chvíli testovaná komponenta již očekává, že nižší vrstva si pamatuje na jakou adresu byla odeslána poslední zpráva a tedy odkud je odpověď očekávána.





## Kapitola 3

### Požadavky na testovací prostředí

Aby byly splněny požadavky normy [EN511] a zároveň byla zachována praktická využitelnost testovacího prostředí, je nutné zajistit schopnost prostředí měřit pokrytí testované komponenty v rámci plné softwarové integrace a zajistit jednoznačnou identifikaci samotných testů pro jejich provázání se specifikací a s požadavky. Zároveň je mandatorní i možnost garantovat schopnost opakovat testovací proces.

#### 3.1 Možnost softwarové integrace

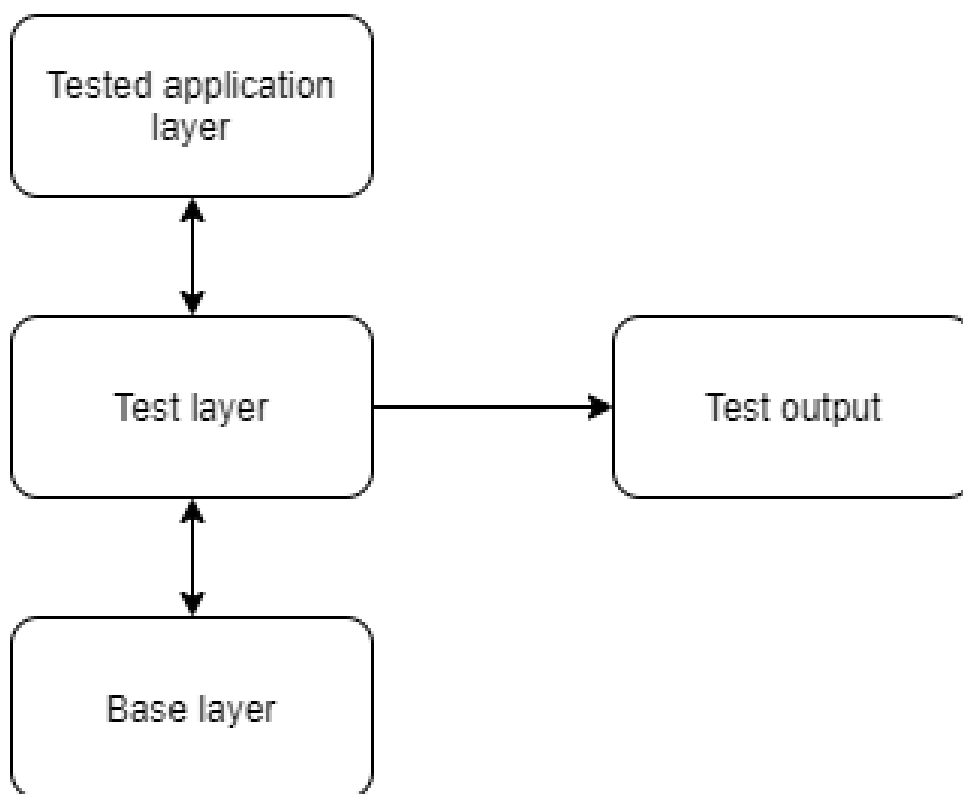
Pro splnění podmínek 2.3.5 je v rámci testovacího procesu nutné docílit softwarové integrace, v rámci které bude poté probíhat měření pokrytí testované komponenty testy.

Softwarové integrace je dosaženo ve chvíli, kdy jsou všechny komponenty výsledného systému zahrnuty do testovacího procesu. Z důvodu, že nejnižší komponenty systému mohou být zároveň závislé na cílovém hardware, existuje možnost, že softwarová integrace je rovněž podmíněna integrací softwaru s hardwarem.

Jelikož vycházíme z 2.1, očekáváme vertikálně vrstvený multikomponentní systém, tedy v rámci vztahu dvou komponent vždy jednu nižší (základovou) a jednu vyšší (aplikační). Tato podmínka nám povoluje mezi nižší a vyšší komponentu vložit další vrstvu, kterou je právě testovací prostředí.

Ve chvíli, kdy je testovací vrstva vložena mezi základovou a aplikační vrstvu (3.1), je testovací prostředí přebírat kompletní kontrolu nad během testované

komponenty a zároveň je docíleno softwarové integrace.



**Obrázek 3.1:** Vložení testovací vrstvy do zbytku systému

## 3.2 Provázání požadavek-test

Testovací proces musí být přesně specifikovatelný a zdokumentovatelný. Z tohoto důvodu musí být možné provázat jednotlivé testy s požadavky, které jsou na testovaný software kladeny. Z tohoto důvodu musí být jednotlivé testy přesně identifikovány.

V rámci testovacího prostředí je proto nutné umožnit dělit běh testů na jednotlivé *testcases*, které je poté možné v rámci dokumentace přímo provázat s konkrétními požadavky.

Proto musí být v rámci přípravy testovacích sad vygenerován soubor obsahující specifikace jednotlivých *testcasů*, na základě kterého je možné testovací proces identifikovat a zopakovat. K tomuto nemusí nutně sloužit přímo zdrojové kódy testů, neboť obsahují nadbytečné množství informací. Pro tuto funkci je dostačující a žádoucí pouze soubor schraňující jednotlivé události, které jsou vykonané testerem nad rámec standardního běhu systému, a to ve formě čitelné běžnému programátorovi, tedy například ve formě *.json* souboru.

## 3.3 Obecnost

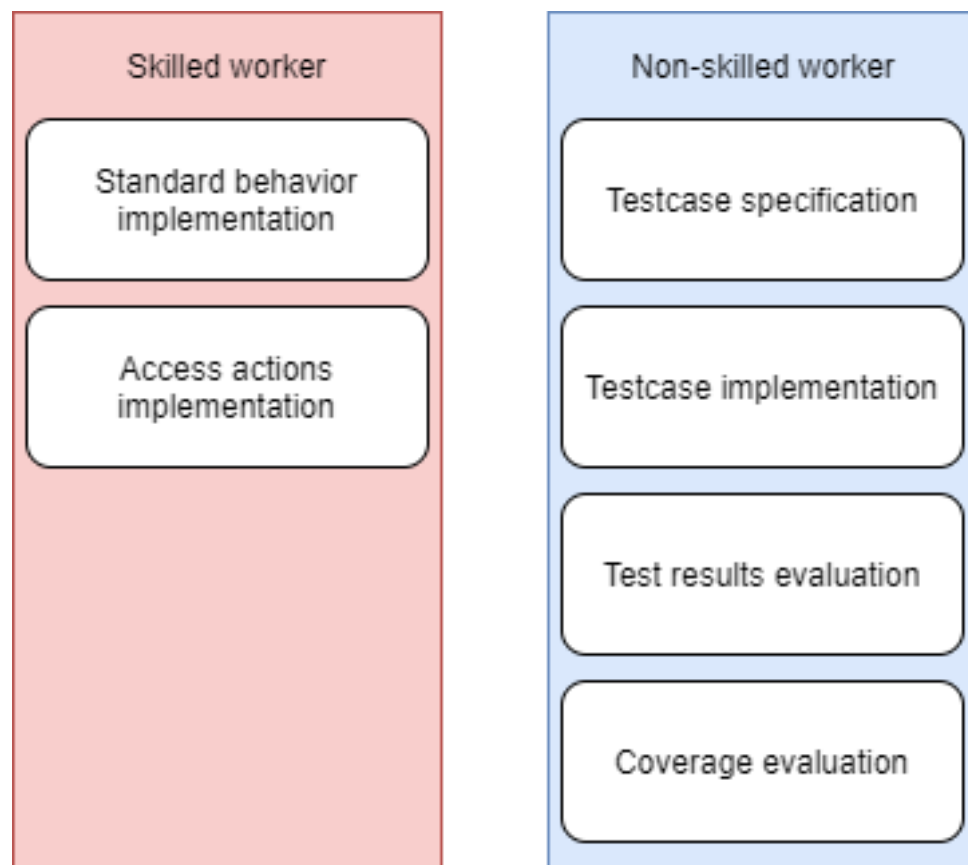
Testovací prostředí musí být obecné pro možnost využití s jakýmkoliv C++ kompilátorem a na jakémkoliv vertikálně vrstveném multikomponentním systému. Z tohoto důvodu je nutné aby prostředí nebylo závislé na externích knihovnách včetně standardních, neboť standardní knihovny nemusejí být v testované komponentě využívány a jejich definice mohou kolidovat s definicemi v rámci testované komponenty. Prostředí si proto musí implementovat vlastní typy (jako je například `std::string`) pro zachování maximální obecnosti prostředí.

## 3.4 Minimální požadavky na uživatele

Testovací prostředí na sebe musí přebírat část požadavků na specializaci uživatele. Proto musí být testovací prostředí rozděleno na dva separátní úkony. Prvním je příprava samotného prostředí pro testování specifické projektu a druhým tvorba samotných testovacích sad.

První úkon musí být vykonáván testerem s hlubokou znalostí projektu. Nicméně úkon druhý již musí být uživatelsky přívětivý natolik, aby tvorba testovacích sad mohla být vykonávána již testerem s výrazně nižší znalostí problematiky.

Příprava testovacího prostředí tedy musí implementovat standardní funkcionalitu systému a nabídnout možnost do běhu zasahovat. Samotné zásahy již mohou být využívány v rámci grafického rozhraní a konkrétní činnost již tvůrce testovacích sad nemusí implementovat, ani zevrubně znát. Rozdělení práce je znázorněno na 3.2



**Obrázek 3.2:** Rozdělení práce na testování softwaru

## Kapitola 4

### Návrh testovacího prostředí

Cílem práce bylo sestavit testovací prostředí pro možnost testování softwarových komponent na úrovni softwarové integrace.

Tato úroveň se již významně blíží systémové úrovni testů, neboť je již, na rozdíl od Unit testů, popřípadě Component testů (bez softwarové integrace), významně závislá na konkrétním projektu.

Z tohoto důvodu není možné obecně zadefinovat prostředí pro všechny typy komponent znázorněné na 2.1. Je ovšem možné připravit testovací prostředí s určitými omezeními na konkrétní typ komponent.

V této práci se zabýváme pouze vertikálně vrstvenými komponentami, což je již dostatečná specifikace pro tvorbu testovacího prostředí pro testování takto strukturovaného programu.

Pro testování na úrovni softwarové integrace je možné částečně využít všeobecně známá a optimalizovaná unit test prostředí jako je GoogleTest [Goo19], CppUTest [GV20]. Nad jejich funkcionalitu je ovšem zapotřebí naimplementovat další hlubší funkcionalitu mocků, než jakou tato prostředí podporují.

Navržené prostředí vytváří obecnou nadstavbu nad libovolné unit test prostředí, které je vytvořeno přímo pro integrační testování na úrovni komponent v multikomponentním systému podle 2.1.

Prostředí vyvinuté v rámci této práce se stará o validní běh testované komponenty na volitelné úrovni integrace se zbytkem systému a poskytuje možnost v libovolných časech do onoho běhu zasahovat.

Prostředí rovněž dělí celou testerskou činnost na dvě části. Činnost implementační, kdy je prostředí připravováno zkušeným testerem s hlubší znalostí projektu, a poté činnost čisté tvorby testovacích sad, která probíhá výhradně přes grafické rozhraní a tedy není zapotřebí hlubší znalost samotného projektu.

Tímto je docíleno efektivního využití lidských zdrojů v rámci vývoje projektu,

díky přesnému rozdělení na implementační a testerskou činnost je možné využít zkušeného zaměstnance pouze pro skutečně nezbytnou práci. Pro méně náročnou testerskou činnost je již možné využít ne zcela znalých zaměstnanců.

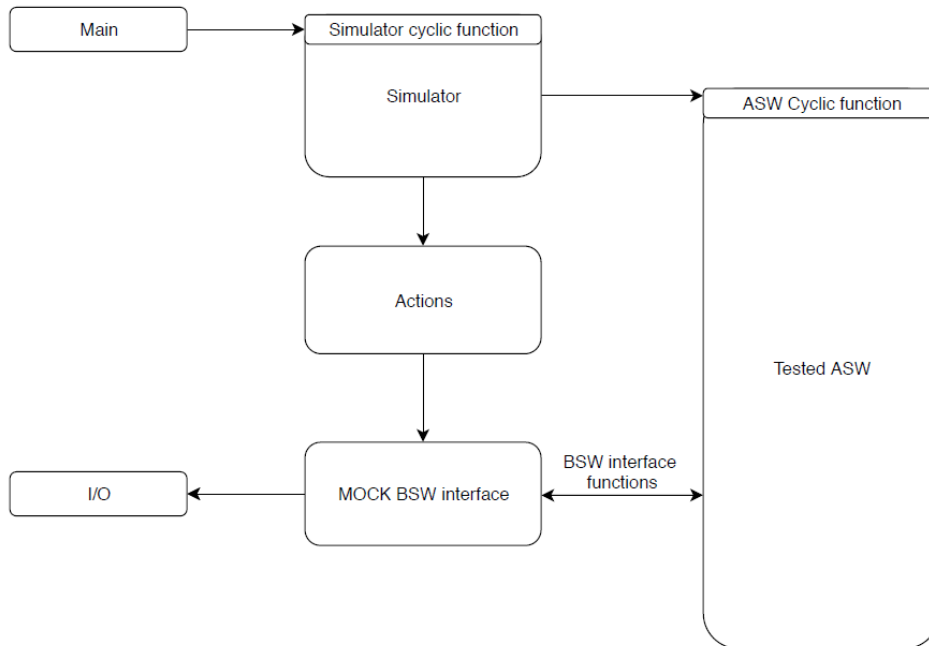
## 4.1 Integrace

Jednotlivé komponenty mezi sebou komunikují prostřednictvím rozhraní (interface), jak je zadefinováno v 2.1.1. Všechny testy a úpravy vnitřních stavů komponenty proto probíhají výhradně prostřednictvím těchto rozhraní. Přístup, jakým je toho docíleno, je však možné rozdělit do dvou kategorií.

### 4.1.1 Plné nahrazení systému

Pro nejjednodušší formu testování na úrovni komponent je tedy možné vytvořit vlastní implementaci funkcí rozhraní, se kterými má testovaná komponenta komunikovat. Tím jsme schopni vyčítat parametry, se kterými jsou funkce volány a simulovat funkcionalitu ostatních komponent.

Toto řešení je však při testování komplexnějších komponent nedostatečné, neboť mnohá volání přicházejí ve specifických okamžicích, až když se testovaná komponenta dostane do určitého vnitřního stavu. Aby bylo možné to otestovat touto metodou, musí být vlastní implementace funkcí velmi komplexní a de facto implementovat celou komponentu, s níž má testovaná komponenta komunikovat. Toto má za následek výraznou implementační zátěž pro testera. Na druhou stranu tato metoda přináší i určité výhody. Hlavní z nich je možnost uplatnit tuto formu testování komponenty v době, kdy je tato komponenta oproti zbytku systému napřed, a tedy neexistuje jiná implementace ostatních komponent. Rovněž se díky této metodě můžeme zcela odklonit od cílového hardwaru, se kterým mohou být ostatní komponenty spojeny, a všechnu jejich funkcionalitu pouze simulovat. Tím můžeme docílit testování komponenty nezávisle na cílovém hardwaru, a tedy testovat s vyšší výpočetní kapacitou.



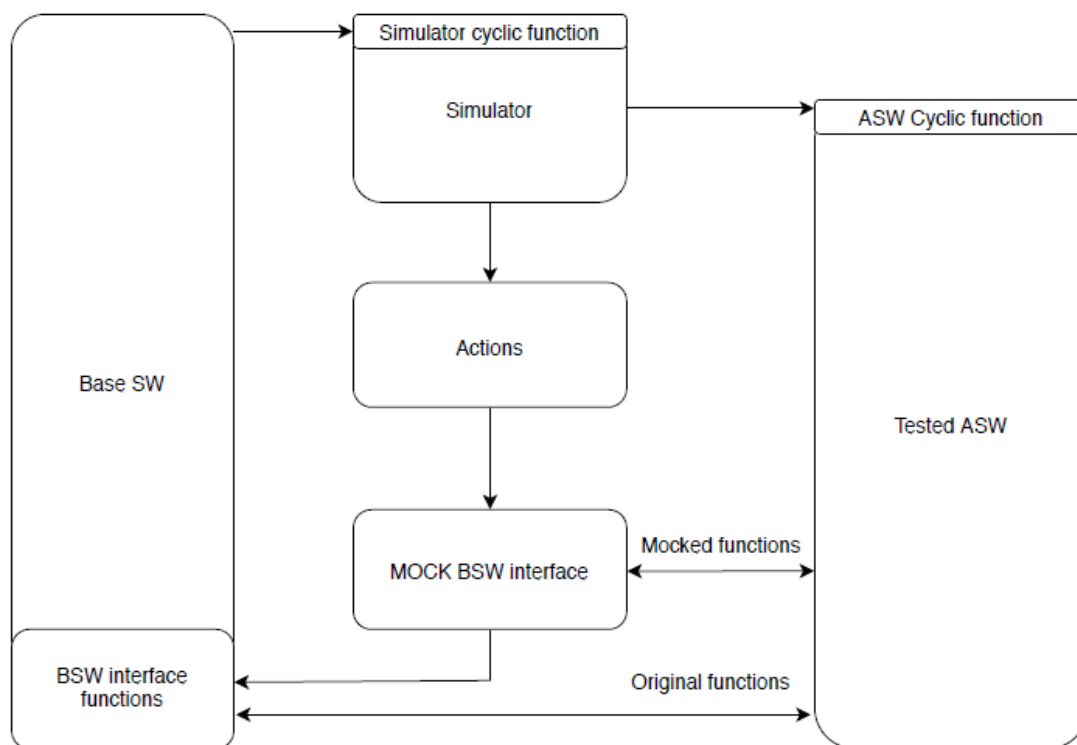
Obrázek 4.1: Nahrazení systému

#### ■ 4.1.2 Plná integrace se systémem

Obecnějším řešením je ponechat i ostatní komponenty systému (tedy nejen komponentu testovanou), a přesměrovávat pouze volání funkcí, které nás v daném testu zajímají. Toho lze docílit vložení vrstvy testovacího prostředí mezi vrstvu testovanou a zbytek systému. Testovací prostředí pouze přesměrovává všechna volání do ostatních komponent, a v kýženém okamžiku sleduje jedno konkrétní, které v sobě nese informaci nezbytnou pro test.

Tímto je docíleno nejen hlubší integrace, neboť již na úrovni komponent testů rovněž kontrolujeme část systémové integrace, ale rovněž se výrazně sníží implementační náročnost pro testera.

Nevýhodou této metody je přímá závislost na více komponentách a fázi jejich vývoje.



Obrázek 4.2: Integrace do systému

### ■ 4.1.3 Částečné nahrazení systému

Jelikož oba přístupy mají své výhody i nevýhody, je v testovacím prostředí možné volit libovolnou kombinaci mezi těmito přístupy v závislosti na specifických požadavcích konkrétního projektu.

Je tedy možné využít plné náhrady některé systémové komponenty, zatímco jiné nechat originální. Rovněž je možné implementovat pouze část funkcionality komponenty a zbytek nechat originální, a tím dosáhnout integrace na maximální možné míře, zachovat plnou testovatelnost a zároveň omezit implementační náročnost pro testera.

## ■ 4.2 Real time

Při běhu na systému reálného času bylo nutné zvážit i další aspekty, mezi něž patří zejména doba vyhrazená pro běh jednotlivých komponent. Externí



hardwarové watchdogy na cílovém zařízení kontrolují, zda byla včas předána kontrola zpět základové vrstvě programu. Proto je nutné, aby testovací vrstva vždy dodržela časový rámec, který je určený pro vrstvu vyšší.

#### ■ 4.2.1 Reálný a virtuální čas

Testovací prostředí odděluje čas reálný a čas simulace. Čas simulace je virtuální a je závislý na výpočetní kapacitě zařízení, na které testovací prostředí běží. Vždy však je garantováno, že čas simulace poběží rychleji než čas reálný. V opačném případě by totiž samotný hardware nebyl schopen výpočetně stihnout standardní běh programu. Obecně lze tvrdit, že ve většině případů bude čas simulace výrazně rychlejší než čas reálný, a tím se celý testovací proces urychluje.

Reálný čas je využíván k pozastavení simulace s definovanou periodou a navrácení kontroly nad programem zpět nižší vrstvě, a poté k navázání dalšího běhu po opětovném předání kontroly vyšší vrstvě.

### ■ 4.3 Testovací strategie

Standardní způsob výkonu testů je způsobem restart-událost-kontrola. Tento způsob se nejen nevyhne častému kopírování jednotlivých testů, ale rovněž musí při každém testu v první řadě dostávat komponentu do určitého stavu. Navrhované testovací prostředí se řídí odlišným paradigmatem, konstantně nechává komponentu běžet a v předem definovaných časech simulace provede předdefinovanou akci.

Při využití logiky zvolené v našem testovacím prostředí tedy můžeme jednoho přechodu do určitého stavu komponenty využít pro více samotných testů, čímž je urychlen samotný testovací proces.

Rovněž tato struktura umožňuje znovupoužití jednotlivých akcí, které jsou jednou zdefinovány, a poté umožňuje jejich vícenásobné využití, čímž výrazně snižuje náročnost samotné implementace.

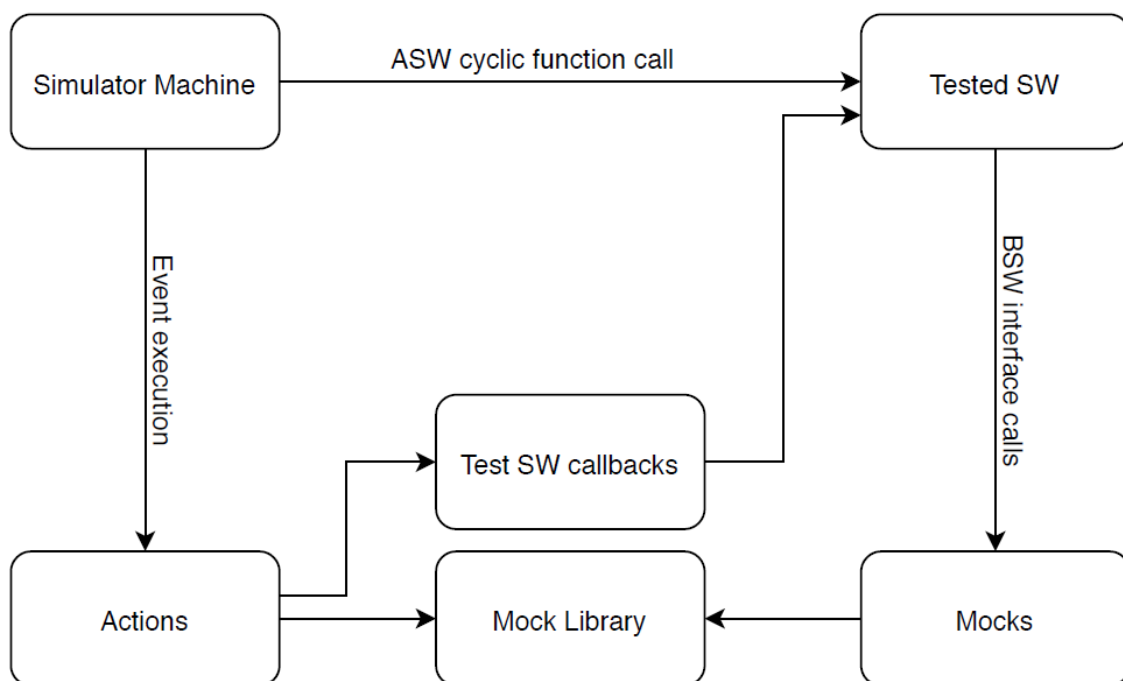
## 4.4 Struktura

Testovací prostředí je strukturováno jako samostatná uzavřená komponenta, která je vložena jako vrstva mezi testovanou komponentu a zbytek systému. Tímto fakticky přebírá kontrolu nad během testované komponenty. Z pohledu testované komponenty prostředí vystupuje celý zbytek systému, z pohledu zbytku systému vystupuje jako testovaná komponenta.

Pro komunikaci s testovanou komponentou využívá svou implementaci jednotlivých funkcí rozhraní (Mock). Tyto funkce poté ukládají parametry, se kterými jsou volány, do Mock knihovny a rovněž z této knihovny berou data pro odpověď na tato volání.

Tester si rovněž definuje Akce, za pomoci kterých upravuje, popřípadě vyčítá Mock knihovnu, a tím simuluje chování externích komponent systému a sleduje chování komponenty testované.

Struktura jednotlivých bloků je znázorněna na obrázku 4.3.



Obrázek 4.3: Architektura testovacího prostředí

### ■ 4.4.1 Mock

Mock jsou funkce, které implementují funkcionalitu rozhraní. Není nutné zachovat jejich kompletní funkcionalitu, byť je žádoucí se jí přiblížit co nejvíce. Jsou implementovány samotným testerem, neboť jsou přímo závislé na konkrétním testovaném projektu a rozsahu testů, a poté jsou využívány jako náhražka implementace rozhraní u komponent, které nejsou v případě částečné, popřípadě plné integrace se systémem jinak zdefinovány.

Tyto funkce plní v rámci testovacího prostředí trojí roli.

První rolí je, že udržují systém ve standardním běhu, tedy navrací očekávané hodnoty na volání jednotlivých funkcí.

Druhou rolí je úprava vnitřního stavu testované komponenty, kdy právě využitím úpravy návratových hodnot dostáváme komponentu do žádoucího stavu.

Třetí rolí je poté sledování parametrů, se kterými jsou jednotlivé funkce volány, čímž umožňují testerovi kontrolovat, že se komponenta chová podle očekávání.

### ■ 4.4.2 Mock knihovna

Mock knihovna slouží jako rozhraní mezi testerem a mockovanými funkcemi. Do této knihovny se ukládají jednotlivé parametry se kterými byly funkce volány. Na základě toho poté mockované funkce volí návratové hodnoty, aby věrně simulovaly funkci ostatních komponent systému.

Prostřednictvím této knihovny může tester ovlivňovat chování jednotlivých mockovaných funkcí, a tím simulovat určité nestandardní chování ostatních systémových částí.

Zároveň je možné z této knihovny vyčítat jednotlivé hodnoty, díky čemuž má tester možnost sledovat chování samotné testované komponenty.

### ■ 4.4.3 Akce

Akce jsou předdefinované události, které může tester libovolně spouštět v jakýchkoliv časech.

Tester si akce definuje sám, neboť jsou přímo závislé na konkrétním projektu, a stejně tak i na testovaných událostech.

Účel Akcí je určen primárně k přípravování a vyčítání dat z knihovny Mocků, čímž aktivně upravuje (respektive sleduje) chování testované komponenty.

Další funkcionalitou Akcí je i ovládání samotné simulace za využití předregistrovaných callback funkcí (například *restart*, *stop simulation* a dalších). Teoreticky v sobě mohou však Akce nést i další události, neboť se ve výsledku jedná o funkci, která je vykonána ve zvoleném čase simulace. Žádná omezení na konkrétní využití nebyla zavedena, neboť se očekává, že tester bude mít k případnému jinému využití oprávněné důvody.

Pro snížení implementační náročnosti jednotlivých Akcí mohou (a nemusí) být Akce široce parametrizovány. Proto z pohledu testera stačí zadefinovat například `SendTelegram`, kde již zvolenými parametry určí komunikační linku, data i další náležitosti, vyplývající z jeho konkrétních potřeb.

#### ■ 4.4.4 Paměťová struktura

Z důvodu běhu na cílovém hardwaru, u kterého nelze očekávat souborový systém, jsou jednotlivé akce nahrávány do programové paměti formou automaticky vygenerované funkce. Ta je v rámci inicializace volána a všechny akce jsou poté uloženy na haldu v paměťově optimalizované formě.

### ■ 4.5 Grafické rozhraní

Součástí navrženého testovacího prostředí je rovněž grafické rozhraní. Rozhraní slouží k samotné tvorbě testovacích sad, kdy si uživatel vybírá, které akce budou kdy provedeny.

Důvodem k tvorbě tohoto rozhraní je snížení náročnosti samotné implementace, a především diverzifikace práce, kdy člověk sestavující testy prostřednictvím grafického rozhraní již nemusí být blízce seznámen s testovaným softwarem, ani nemusí mít hluboké znalosti programovacího jazyka.

Grafické rozhraní nabízí možnost třídít jednotlivé akce pod skupiny *testcase*, čímž umožňuje jejich logické rozdělení pro možnost pokrývat konkrétní požadavky konkrétním *testcasem*.

Grafické rozhraní na základě výběru akcí vygeneruje *json* soubor, popisující jednotlivé zvolené akce a jejich parametry, z něhož je následně vytvořen zdrojový *.cpp* soubor pro kompilaci se samotným testovacím prostředím.

Na rozdíl od unit test prostředí je zde tedy hlavním prvkem samotné grafické prostředí a nikoliv využívání předdefinovaných maker ve zdrojových souborech, jako je tomu například u `CppUTest` [GV20], či `GoogleTest` [Goo19].

Vygenerovaný *json* soubor zároveň slouží pro přímou identifikovatelnost jednotlivých testů vůči jejich specifikaci.

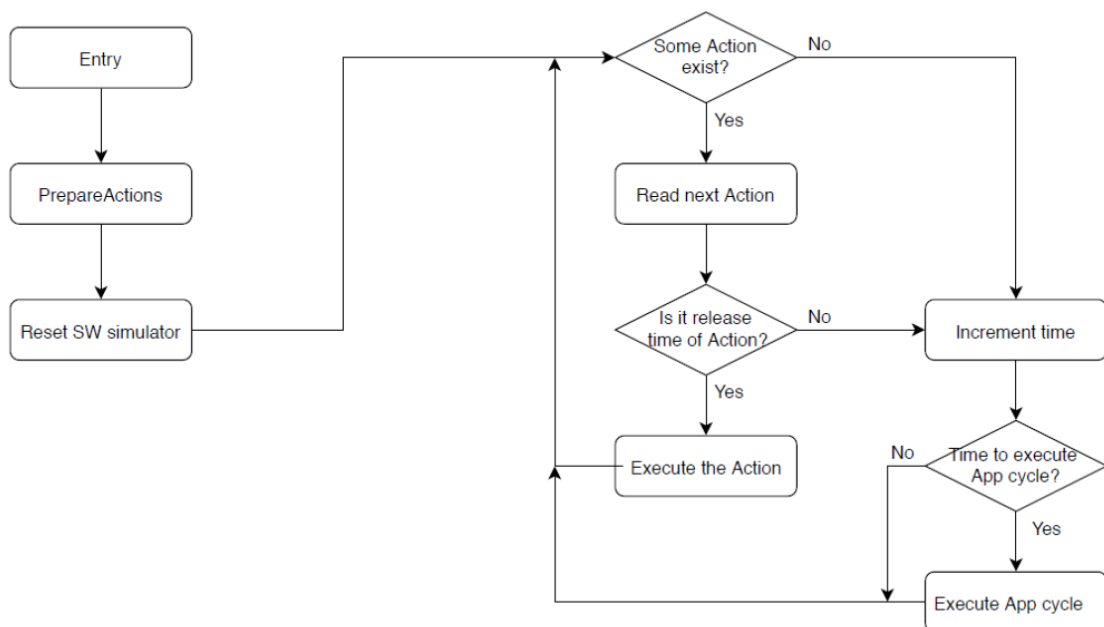
Tento soubor může být kdykoliv znovu použit pro opakování testovacího procesu.

#### ■ 4.5.1 Dělení na testcasy

Aby bylo možné jednoznačně identifikovat jednotlivé testy vůči jejich dokumentované specifikaci a vůči testovaným požadavkům, je každá Akce přiřazena konkrétnímu testcasu, čímž je zajištěno, že soubor Akcí dohromady vytváří jeden přesně identifikovatelný testcase.

## ■ 4.6 Běh

Běh samotného programu se řídí podle stavového diagramu znázorněného na obrázku 4.4



Obrázek 4.4: Běh testovacího prostředí

### ■ 4.6.1 Registrace

Prvním krokem, který testovací prostředí provede, je že zaregistruje svou vlastní cyklickou funkci do nižší vrstvy (pokud existuje) popřípadě do timeru, kterým má být testovaný software buzen. Tímto krokem pro zbytek systému začíná přebírat místo, které v reálném produktu zastává testovaný software. Po tomto kroku již je veškerá další činnost testovacího prostředí závislá na zavolání této cyklické funkce, a v rámci ní vykonává veškerou činnost.

### ■ 4.6.2 Inicializace

Inicializace je prvotní stav automatu, kterým se testovací prostředí řídí. Poté vrátí kontrolu zpět nižší vrstvě a čeká na příští zavolání cyklické funkce, neboť následující stav může být časově náročný.

Tím stavem je čtení jednotlivých akcí, které byly předdefinovány uživatelem prostřednictvím grafického rozhraní. Ve chvíli, kdy jsou akce načteny a uloženy do dynamické paměti v prioritní frontě, se cyklus opět ukončuje a čeká se na další zavolání cyklické funkce.

Dalším krokem je restart a inicializace samotné testované komponenty. Toho je docíleno zavoláním uživatelem předdefinovaných funkcí `test_sw::deInit()` a následně `test_sw::init()`. V rámci těchto funkcí by rovněž měla být reinitializována i knihovna mocků, čímž je simulován restart všech ostatních komponent, se kterými má testovaná komponenta prostřednictvím rozhraní komunikovat.

Následujícím stavem už je *normalOperation*, který obsahuje svůj vlastní vnitřní stavový automat.

### ■ 4.6.3 Hlavní cyklus

V rámci hlavního cyklu již testovací prostředí začne využívat výpočetní kapacitu na maximum. V rámci něj je kontrolován aktuálně uběhnutý čas v rámci tohoto zavolání cyklické funkce testovacího prostředí. Kontrola nižší vrstvě se vrací až v době, kdy je dosažena maximální doba, kterou má v systému komponenta povoleno využít.

V rámci tohoto stavu je periodicky volán vnitřní stavový automat.

### ■ Načtení akce

V rámci tohoto stavu se načte z fronty další akce, která má být prostřednictvím testovacího prostředí vykonána jako další.

### ■ Vykonání akce

V případě, že simulovaný čas dosáhl času spuštění akce, je akce vykonána. Po jejím vykonání se příští stav nastavuje opět na čtení, aby akce, které mají být vykonány simultánně skutečně mohly být z hlediska virtuálního času simulace provedeny paralelně.

### ■ 4.6.4 Inkrementace času

V případě, že akce ještě neměla být vykonána, přichází na řadu inkrementace času. V rámci té se inkrementuje jak virtuální čas simulace, tak se aktualizuje i čas reálný, který je používán pro přerušení simulace





# Kapitola 5

## Ověření návrhu řešení

Na základě návrhu představeném v kapitole 4 bylo pro ověření konceptu implementováno testovací prostředí a jeho funkcionalita ověřena pro testování zařízení plňujícího požadavky na SIL4 zařízení podle normy EN 50128:2011 [EN511].

### 5.1 Jak software použít

Pro využití navrženého prostředí je nutné v první řadě provést určité kroky pro jeho fungování se specifickým projektem.

#### 5.1.1 Příprava

Pro funkční běh programu jsou zapotřebí tyto softwarové moduly

- **Testovaný software** - softwarová komponenta, jež má být testována. Tato komponenta může být vložena jak formou statické knihovny, tak formou zdrojových souborů. Využití statické knihovny grantuje jistotu, že

testované binární soubory jsou shodné s výsledným produktem. Zdrojové soubory jsou poté potřebné v případě potřeby úpravy zdrojových souborů pro potřeby testů, například přidáním maker pro kontrolu pokrytí testy (branch coverage).

- **Deklarace komunikačních rozhraní mezi komponentami** - header soubory, obsahující deklarace funkcí rozhraní pro komunikaci mezi komponentami
- **Mock knihovna** - testerem definovaná funkcionalita jednotlivých funkcí rozhraní
- **Ostatní komponenty** - v případě, že není potřebné mockovat funkcionalitu nějakého rozhraní, je možné využít originální komponentu
- **Definice akcí** - definice jednotlivých událostí, které budou využívány v průběhu testování
- **Testovací prostředí** - prostředí je implementováno v rámci této práce, během přípravy je však nutné zadefinovat několik projektově specifických funkcí (například funkce pro restart testovaného softwaru a podobně)

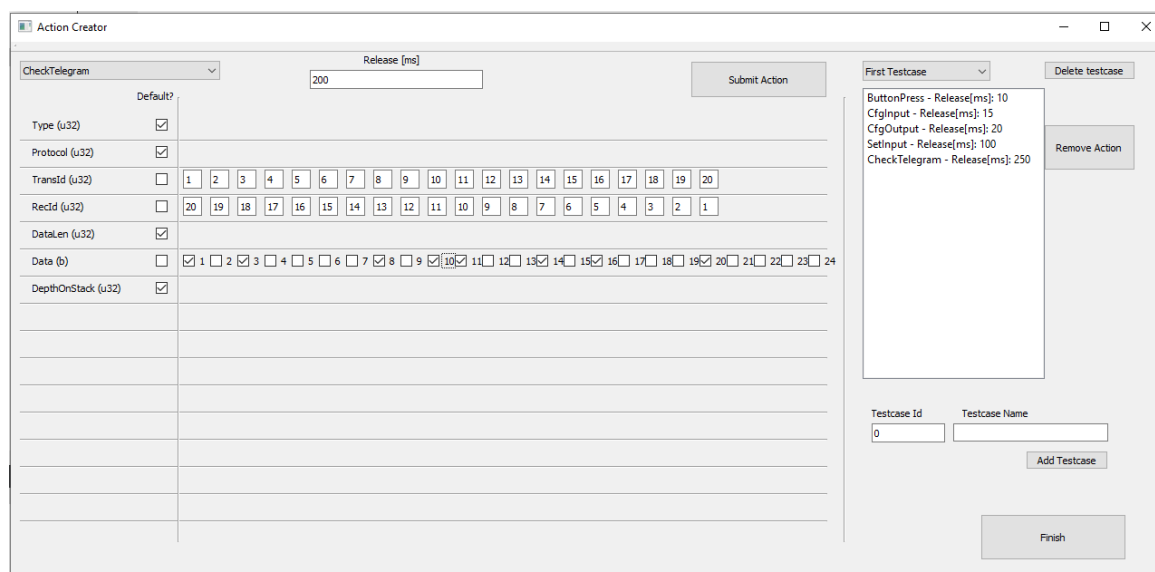
Pro přípravu je tedy z pohledu testera zapotřebí pouze zadefinovat Mock knihovnu, zadefinovat jednotlivé akce a nakonec zadefinovat několik projektově specifických funkcí.

Ve chvíli, kdy jsou tyto části zadefinovány, je možné spustit podpůrný skript, který připraví `.json` soubor, který obsahuje specifikace jednotlivých akcí, který bude poté využívat grafické rozhraní.

## ■ Užívání

Ve chvíli, kdy je vše připraveno, je možné jednoduše spustit grafické rozhraní. Zde si tester vybere, které akce, v jakém čase běhu a s jakými parametry mají být vykonány.

Po jejich vytvoření je poté možné pouze zkompileovat prostředí společně s částmi definovanými v 5.1.1.



Obrázek 5.1: Grafické rozhraní

## 5.2 Omezení

Při využití navrženého testovacího prostředí je nutné zvážit omezení vyplývající ze samotné implementace.

### 5.2.1 Typ komponenty

Samotné testovací prostředí je možné aplikovat na libovolnou komponentu, která splňuje vertikální strukturu komponent a je buzena v pevných časových intervalech nižší vrstvou prostřednictvím cyklické funkce.

### 5.2.2 Výpočetní kapacita

Testovaná komponenta rovněž nesmí využívat čas pro ni určený naplno, neboť testovací prostředí přidává dodatečnou zátěž v řádu desítek operací v každém aplikačním cyklu. Vykonání akce poté závisí na výpočetní náročnosti

implementace samotného uživatele, ovšem lze očekávat další přidanou zátěž. Proto musí být před využitím zvážena výpočetní kapacita cílového hardwaru a její využití při běhu bez testů.

Testovací prostředí je připraveno s očekáváním výrazně vyšší výpočetní kapacity než jsou požadavky testované komponenty a optimalizuje běh právě za těchto podmínek.

### ■ 5.2.3 Paměť

Stejně jako v případě výpočetní kapacity, i z hlediska paměťové náročnosti využívá testovací prostředí část paměti zařízení. Naprostou majoritu paměti využívají jednotlivé zadané akce. V případě velkých testovacích sad může jít o nezanedbatelné množství. Testovaná komponenta proto nesmí vyplňovat jí určený paměťový prostor na maximum. V případě velmi malého nevyužitého prostoru je nutné rozdělit běh velkých testovacích sad na několik menších.

## ■ 5.3 Srovnání s využitím unit test prostředí

V této kapitole je zanalyzováno využití navrhovaného testovacího prostředí v kontrastu proti standardně užívanému unit test prostředí CppUTest [GV20] pro integrační testování softwarových komponent.

### ■ 5.3.1 Příprava

Z hlediska přípravy je pro navržené testovací prostředí naimplementovat jednotlivé mock funkce s jejich hlubší funkcionalitou. Tato činnost je shodná i pro CppUTest, neboť v případě vyšší funkcionality na úrovni softwarové integrace podpora je podpora mock definovaná v prostředí CppUTest nedostatečná.

Další implementační činností v rámci navrženého testovacího prostředí je definice jednotlivých akcí, kterými je poté v rámci testů upravována a vyčítána mock knihovna. Definice samotných akcí není v rámci prostředí CppUTest přímo dělána. Ovšem za předpokladu, že je CppUTest prostředí využíváno

efektivně, je shodná funkcionalita implementována v rámci funkcí přístupové knihovny pro vyčítání a úpravu mocků a obsahuje mírně nižší implementační náročnost.

Z hlediska přípravy samotného prostředí je tedy implementační náročnost téměř shodná s mírnou převahou unit test prostředí, způsobené tvorbou funkcionality přímou implementací funkcí, kde oproti tomu navržené testovací prostředí implementuje potomky třídy Action pro shodnou funkcionalitu.

### ■ 5.3.2 Tvorba testovacích sad

Následnou činností je poté tvorba samotných testovacích sad. V této fázi se naplno projevují výhody navrženého testovacího prostředí.

V rámci této činnosti, která je objemem řádově náročnější než samotná příprava prostředí, již pracuje navržené testovací prostředí pouze s grafickým rozhraním, které poté automaticky generuje zdrojový kód. Díky mezikroku ve formě *.json* souboru je rovněž možné akce definovat i přímo zde.

Navržené testovací prostředí se stará o standardní běh komponenty a stará se o korektní časovou souslednost. Proto je možné jednotlivé akce vykonávat již pouze volbou parametru *Release* pro jednotlivé akce.

CppUTest naopak vyžaduje přímou implementaci jednotlivých testovacích sad. V rámci této implementace je mimo jiné nutné dbát na přesnou časovou souslednost událostí, pro každý test měřit čas a sledovat, po kolika voláních cyklické funkce má být jaká činnost vykonána.

V rámci tvorby testovacích sad se tedy projevuje výhoda navrženého testovacího prostředí oproti pouhému využití unit test prostředí. Hlavní výhodou je výrazně nižší znalostní úroveň potřebná pro implementaci samotných testovacích sad, které jsou z hlediska práce testera majoritní částí práce. Toto je zajištěno tím, že samotná komponenta již běží korektně, a jediné, co je nutné, je vygenerovat událost. V případě využití unit test prostředí je nutné znát konkrétní software mnohem zevrubněji a vždy se dostávat do konkrétního stavu, kdy má být daná událost vykonána.

## ■ 5.4 Prostředí

V rámci implementace návrhu testovacího prostředí bylo využito následujících vývojových prostředí.

### ■ 5.4.1 Grafické prostředí

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 prostředí na základě specifik v kapitole 4 bylo implementováno a verifikováno při testování zařízení z řady xCM firmy SIEMENS.

### 6.1 Plánované využití

V rámci testování bylo ověřeno splnění podmínek stanovených normou EN 50128:2011 [EN511] pro nejvyšší SIL4 úroveň bezpečnosti vývoje. Prostředí však v době vydání práce neprošlo oficiálním validačním procesem. Jeho využití je plánováno jako oficiální verifikační prostředí pro nadcházející verze zařízení z řady xCM společnosti SIEMENS.

### 6.2 Další možnosti využití

Navržené testovací prostředí je připraveno pro maximální generalizaci testovacího procesu. Díky možným konfiguracím v rámci přípravy prostředí pro daný projekt, jako je volitelná hloubka dosažené softwarové integrace či ne-restriktivní povaha definovaných Akcí, které sice mají specifikované plánované využití, avšak toto využití není mandatorní.

V průběhu přípravy prostředí zkušeným 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<sup>+</sup>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



dotělováním opoždřených (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 uživatelem 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<sup>+</sup>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<sup>+</sup>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, *Efficient 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.

## I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Steidl** Jméno: **Richard** Osobní číslo: **420389**  
Fakulta/ústav: **Fakulta elektrotechnická**  
Zadávací 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.

\_\_\_\_\_  
Datum převzetí zadání

\_\_\_\_\_  
Podpis studenta