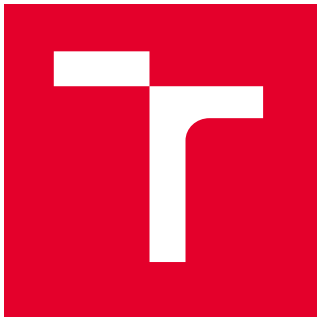


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

ÚSTAV AUTOMATIZACE A MĚŘICÍ TECHNIKY

DEPARTMENT OF CONTROL AND INSTRUMENTATION

APLIKACE PRO FORMÁLNÍ TESTOVÁNÍ PLC PROGRAMU

SOFTWARE FOR FORMAL TESTING OF PLC PROGRAM

BAKALÁŘSKÁ PRÁCE

BACHELOR'S THESIS

AUTOR PRÁCE — **Ondřej Sýkora**

AUTHOR

VEDOUCÍ PRÁCE — **Ing. Jakub Arm, Ph.D.**

SUPERVISOR

BRNO 2022

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

Bakalářská práce

bakalářský studijní program **Automatizační a měřicí technika**

Ústav automatizace a měřicí techniky

Student: Ondřej Sýkora ***ID:*** 221017

Ročník: 3 ***Akademický rok:*** 2021/22

NÁZEV TÉMATU:

Aplikace pro formální testování PLC programu

POKYNY PRO VYPRACOVÁNÍ:

Úkolem této práce je vytvořit aplikaci pro formální testování PLC programu pro Siemens. K tomu bude potřeba implementovat rozhraní TIA Openness. Předmětem testování je formální úprava a funkčnost kódu dle zadaného předpisu.

1) Proveďte rešerši v oblasti formálního testování PLC kódu.
2) Navrhněte aplikaci pro testování PLC kódu.
3) Realizujte testovací aplikaci.
4) Vytvořte demonstrační testovací scénáře a vyhodnoťte výsledky testování.

DOPORUČENÁ LITERATURA:

Blake, M., Idris, F. Hacks To Crush PLC Program Fast & Efficiently Everytime: Coding, Simulating & Testing PLC from Beginning with Examples. 2021.

Termín zadání: 7.2.2022 ***Termín odevzdání:*** 23.5.2022

Vedoucí práce: Ing. Jakub Arm, Ph.D.

doc. Ing. Václav Jirsík, CSc.
předseda rady studijního programu

ABSTRAKT

Tato bakalářská práce rozebírá testovací techniky, používané v klasickém IT odvětví a analogicky je aplikuje na průmyslové odvětví - konkrétně programování PLC. V práci je navržena a realizována testovací aplikace, jejímž účelem je najít chyby vytvořené PLC programátorem nebo generátorem PLC kódu v projektu prostředí TIA Portal. Aplikace dokáže najít a rozpoznat až 10 různých chyb. Jedná se o windows form aplikaci, vyvinutou v jazyce C#, využívající otevřené rozhraní TIA Portal Openness. Následně je otestována její funkčnost na malém testovacím projektu se známým množstvím chyb a na velkém firemním projektu s neznámým množstvím chyb.

KLÍČOVÁ SLOVA

Testování správnosti PLC kódu, Windows form aplikace, C#, XML, TIA Portal, TIA Portal Openness

ABSTRACT

This bachelor's thesis analyzes the testing techniques used in the IT industry and applies them analogously to the machine industry - specifically PLC programming. In this work, a test application is designed and implemented. It's purpose is to find errors created by a PLC programmer or PLC code generator in the TIA Portal environment project. The application can find and detect up to 10 different errors. It is a windows form application, developed in C#, using the open interface of TIA Portal Openness. Subsequently, its functionality is tested on a small test project with a known number of errors and also on a large company project with an unknown number of errors.

KEYWORDS

PLC code correctness testing, Windows application form, C#, XML, TIA Portal, TIA Portal Openness

Vysázeno pomocí balíčku `thesis` verze 4.03; `http://latex.feec.vutbr.cz`

SÝKORA, Ondřej. *Aplikace pro formální testování PLC programu*. Brno: Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, Ústav automatizace a měřicí techniky, 2022, 91 s. Bakalářská práce. Vedoucí práce: Ing. Jakub Arm, PhD.

Prohlášení autora o původnosti díla

Jméno a příjmení autora:	Ondřej Sýkora
VUT ID autora:	221017
Typ práce:	Bakalářská práce
Akademický rok:	3
Téma závěrečné práce:	Aplikace pro formální testování PLC programu

Prohlašuji, že svou závěrečnou práci jsem vypracoval samostatně pod vedením vedoucí/ho závěrečné práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou všechny citovány v práci a uvedeny v seznamu literatury na konci práce.

Jako autor uvedené závěrečné práce dále prohlašuji, že v souvislosti s vytvořením této závěrečné práce jsem neporušil autorská práva třetích osob, zejména jsem nezasáhl nedovoleným způsobem do cizích autorských práv osobnostních a/nebo majetkových a jsem si plně vědom následků porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon), ve znění pozdějších předpisů, včetně možných trestněprávních důsledků vyplývajících z ustanovení části druhé, hlavy VI. díl 4 Trestního zákoníku č. 40/2009 Sb.

Brno 23.5.2022 .....

.....

podpis autora*

*Autor podepisuje pouze v tištěné verzi.

PODĚKOVÁNÍ

Rád bych poděkoval vedoucímu bakalářské práce panu Ing. Jakubovi Armovi, Ph.D. za odborné vedení, konzultace, trpělivost a podnětné návrhy k práci. Dále děkuji konzultantu z firmy ICE Industrial Services a.s. panu Bc. Zdeňku Ondráčkovi za odbornou konzultaci bakalářské práce.
V Brně dne: 23.května 2022

Obsah

Úvod **19**

1 Formální testování PLC kódu **21**
1.1 Testování softwaru . . . 21
1.2 Statické testovací techniky . . . 21
1.2.1 Statické White box testování . . . 22
1.3 Dynamické testovací techniky . . . 23
1.3.1 Funkční testování - Black box testing . . . 23
1.3.2 Strukturální testování - White box testing . . . 25
1.4 Úrovně testování . . . 28
1.4.1 Jednotkové testy (Unit tests) . . . 28
1.4.2 Integrační testy (Integration testing) . . . 29
1.4.3 Systémové testy (System testing) . . . 29
1.4.4 Akceptační testy (Acceptance testing) . . . 29
1.5 Testování PLC kódu . . . 29
1.5.1 Manuální a automatické testování . . . 29
1.5.2 Formální testování . . . 30

2 Návrh aplikace pro testování PLC kódu **31**
2.1 Cíl aplikace . . . 31
2.2 Časté chyby v PLC projektu (TIA Portal) . . . 31
2.2.1 Klasifikace chyb . . . 31
2.2.2 Chyby v hardwarové konfiguraci projektu . . . 32
2.2.3 Chyby ve funkcích a funkčních blocích . . . 33
2.3 Návrh aplikace . . . 35
2.3.1 Vývoj aplikace . . . 35
2.3.2 Požadavky na aplikaci . . . 36
2.3.3 Důležité pojmy spojené s jazykem C# . . . 36
2.3.4 Soubory XML . . . 38
2.3.5 TIA Portal Openness . . . 38
2.3.6 Hlavní problém Opennessu v rámci použití v mé aplikaci . . . 40
2.3.7 Návrh vzhledu aplikace . . . 44
2.3.8 Interakce aplikace s uživatelem . . . 46

3 Realizace aplikace pro testování PLC kódu **47**
3.1 Členění aplikace . . . 47
3.1.1 Atributy a seznamy hlavní třídy *App_Init_Screen* . . . 47

3.1.2 Metody hlavní třídy *App_Init_Screen* 48
3.1.3 Vytvořené enumy 50
3.1.4 Třída *ProjectBlocks* 51
3.1.5 Třída *Errors* 52
3.2 Základní funkcionality aplikace 54
3.3 Proces testování PLC projektu 56
3.3.1 Kompilace kódu pomocí TIA Openness 56
3.3.2 Vymazání adresáře *XMLHelpExportFolder* 56
3.3.3 Rekurzivní export XML předpisů PLC bloků do adresáře *XML-HelpExportFolder* 57
3.3.4 Načtení vyexportovaných bloků do třídy *ProjectBlocks* 57
3.3.5 Testovací algoritmy 59
3.3.6 Zobrazení chyb 63
3.3.7 Uložení chyb do textového souboru 64

4 Testovací scénáře v TIA Portal projektech a výsledky testování 65
4.1 Testovací projekt 65
4.1.1 Rozmístěné chyby 65
4.1.2 Výsledky testování 72
4.2 Firemní projekt 74
4.2.1 Velikost firemního projektu 74
4.2.2 Výsledky testování 77
4.3 Nevýhody aplikace a plány vylepšení aplikace do budoucna 78

Závěr 79

Literatura 81

Seznam symbolů a zkratek 83

Seznam příloh 85

A Přiložené obrázky 87
A.1 Přehled hlavní třídy App_Init_Screen 87
A.2 Flow diagram procesu testování 88
A.3 Chyby firemního projektu v aplikaci po testování 89

B Obsah přiloženého CD 91

Seznam obrázků

1.1 Příklad pro Branch coverage 27
1.2 Testovací úrovně v čase - názorný obrázek 28
2.1 XML předpis FB, zobrazený pomocí XML editoru 41
2.2 *Accessy*, *Cally*, *Party* a *Wiry* v PLC kódu 42
2.3 Třída Network a její atributy - diagram 43
2.4 Diagram tříd InterfaceOB, InterfaceFB a InterfaceFC a jejich atributů 43
2.5 Základní vzhled aplikace 44
3.1 Diagram tříd pro OB, FB a FC 52
3.2 Diagram třídy *Errors* 53
3.3 Diagram tříd pro chyby v hardwarové konfiguraci 53
3.4 Diagram tříd pro chyby v PLC blocích 54
3.5 Vývojový diagram základních funkcionalit aplikace 55
4.1 Záměrně umístěná nepojmenovaná zařízení v hardwarové konfiguraci 66
4.2 Nepřiřazené porty na zařízeních v rámci PN topologie 66
4.3 Nepojmenovaný network (Station110, network 2) 67
4.4 2 PLC tagy a 1 proměnná z cizí stanice (120) ve stanici 110, network 1 68
4.5 Příklad výskytu chyby typu čtení neinicializované TEMP proměnné v testovacím projektu (Station110, network 2) 69
4.6 Přiřazení do proměnné na 3 cívkách v bloku *Station110* 70
4.7 Stejná proměnná *SS_TONFltrMsg[1]* na 3 časovačích TON (Station130, networky 5 a 6) 71
4.8 *STDComm_0* merker použit v bloku stanice 110, network 5 72
A.1 Atributy 87
A.2 Flow 88
A.3 Automat 89

Seznam tabulek

1.1 Rozhodovací tabulka pro interlock přízdvihu 25
4.1 Nepojmenované networky - výskyt v projektu 67
4.2 Chybná symbolika - výskyt chyby v projektu 67
4.3 Výskyt zápisu do TEMP proměnné před jejím čtením 68
4.4 Výskyt proměnné na více cívkách (coil) 70
4.5 Výskyt nepoužitých proměnných 71
4.6 Doby testování testovacího TIA Portal projektu 73
4.7 Firemní projekt - počet networků jednotlivých PLC bloků - 1.část . . 74
4.8 Firemní projekt - počet networků jednotlivých PLC bloků - 2.část . . 75
4.9 Firemní projekt - počet networků jednotlivých PLC bloků - 3.část . . 76
4.10 Firemní projekt - počet networků jednotlivých PLC bloků - 4.část . . 77
4.11 Doby testování firemního TIA Portal projektu 78

Úvod

Testování správnosti a kvality kódu je v dnešní době nezbytně nutnou částí vývoje softwaru, vyvíjeného jak pro klasické počítače, tak pro programovatelné logické automaty. Jedná se o techniky formální verifikace kódu a opravy chyb programátora.

Tato práce se zabývá testováním správnosti PLC kódu a softwaru obecně. Jsou zde vysvětleny testovací techniky, často používané v rámci klasického programování - programování desktopových aplikací, webových aplikací, počítačových her, mobilních aplikací či vývoje speciálního softwaru pro konkrétní firmy. Většina těchto technik je však přímo aplikovatelná i na vývoj řídicího softwaru pro PLC.

Pokud programujete PLC a pracujete s vývojovým prostředím Siemens TIA Portal, určitě používáte jazyk ladder. Ve vašem kódu (v ladderu) se může vyskytovat určité množství chyb, aniž byste o nich měli vůbec ponětí. S přibývajícím kódem přibývá i množství chyb, které se později projeví při testování vašeho kódu. Pokud máte pouze průměrně výkonný počítač, TIA Portal je znám i tím, že otevírání a "scrollování" kódu není příliš rychlé, což je další z faktorů, který prodlužuje vaše testování (a vaši trpělivost). Pokud je váš kód delší či složitější, testování a ladění kódu vám může zabrat i hodiny, ne-li celé dny. Po takové době pak může poklesnout i vaše soustředěnost a hledání chyb v kódu se může stát méně efektivním. I s takovými problémy se mohou potýkat PLC programátoři (nejen) z firmy ICE Industrial Services a proto bude aplikace určena pro praktické použití v této firmě a zde využívána PLC programátory pro kontrolu chyb v projektech vývojového prostředí TIA Portal.

Cílem práce je tedy navrhnout aplikaci pro formální testování PLC kódu a odhalování častých chyb, která může ušetřit i hodiny času stráveného testováním. Tyto chyby mohou být například: zařízení s výchozím jménem (nepřejmenovaná zařízení), nepřiřazené porty zařízení v rámci PN topologie, nepojmenované networky, čtení neinicializovaných proměnných typu Temp, přepisování proměnné více cívkami, nepoužité proměnné či použité stejné názvy pro více instancí funkčních bloků, čítačů a časovačů. Firma Siemens zatím neimplementovala kontrolu těchto chyb do programu TIA Portal. Ten umí provést kompilaci kódu a najde takové chyby jako kontakt či cívka bez proměnné, ale již nedokáže najít chyby, které vyhledává tato aplikace.

1 Formální testování PLC kódu

1.1 Testování softwaru

Testováním softwaru se rozumí odhalování chyb v programech. Je to postup, při kterém se testuje program na možný výskyt chyb. Tyto chyby mohou být pouze formálního charakteru (chyby v komentářích, neadekvátní typy proměnných, chyby znepřehledňující kód, atd.), ale také mohou být fatální (chyby v bezpečnostní části programu, chyby způsobující kolaps programu, chyby znemožňující správný chod programu či aplikace, atd.). Aby se zejména fatálním chybám předcházelo, zavedlo se testování softwaru, konkrétně testování správnosti jeho zdrojového kódu. Díky testování software (dále jen SW) lze objektivně posoudit rizika nasazení programu, protože testy nám mohou odhalit chyby, kterých se programátor dopustil, nebo části kódu, které zapomenul přidat. Některé chyby se mohou vyskytnout i při pozdějších úpravách kódu z důvodu dalšího vývoje SW, či z důvodu opravy jiné chyby a kvůli této změně se může stát nefunkční jiná část programu. Čím dříve dojde k odhalení chyb, tím lépe a rychleji bude možné program uvést do provozu nebo prodat zákazníkovi. [1][2]

Testování správnosti kódu (ať už kódu pro vývoj desktopových aplikací nebo PLC kódu) je důležitá činnost, jejíž podceňování může vést ke zvýšení nákladů na vývoj. Pokud není software testován, je velká šance, že se při jeho dalším působení v blízké budoucnosti vyskytne chyba. V lepším případě se problém vyskytne ještě ve fázi vývoje, v horším případě u zákazníka, který si závady či chyby všimne a bude požadovat opravu, případně vrácení peněz.[4]

"Obecně se dá říci, že špatné nebo žádné testování vede k finančním ztrátám, ztrátám reputace a dokonce i ke ztrátám lidských životů."[2]

Níže popsané testovací techniky neplatí jen na klasické programování (desktopové aplikace, hry, databáze, programy, atd.), ale právě i na programování PLC. Problematika je občas popisována na textových programovacích jazycích, je však analogicky platná i pro jazyky grafické (networky u PLC programování).

1.2 Statické testovací techniky

Zde se jedná o kontrolu kódu jiným člověkem, který může návrh kódu vizuálně zkontrolovat a zjistit, jaké chyby v kódu vznikají ještě před samotným spuštěním. Statické techniky tedy nevyžadují funkční program - testovaný SW nemusí být ani nutně spustitelný. Statické techniky se využívají v počátečních fázích vývoje. Toto testování tak slouží k prevenci chyb v testovaném kódu. Mimo vizuální kontrolu kódu se zde kontroluje, jestli již v samotném dokumentu projektu není chyba, která

by vedla na nechtěný vznik chybného kódu. Dokument projektu bývá diskutován týmem, hodnotí se technický návrh. Všichni na projektu účinkující programátoři by měli pochopit výsledný dokument a poté opravovat chyby přímo ve vývojovém prostředí. Mezi statické techniky patří neformální kontrola, walkthrough, technická kontrola a statická analýza.[1][3]

Chyby nemusí nutně způsobovat programátor, v případě programování PLC často vznikají chybným generováním PLC kódu pomocí firemních generátorů. Tyto generátory vygenerují často celý projekt, případně do vytvořeného projektu je možné generovat konkrétní funkce, funkční bloky, datové bloky, vizualizační faceplaty, PLC tagy, HMI tagy, zařízení, atd. Tyto prvky pak obsahují chyby různého charakteru, které musí PLC programátor opravovat.

1.2.1 Statické White box testování

White box testování může patřit do statických i dynamických testování, proto je možné jej naleznout i v kapitole 1.3.2. Zde jsou popisovány statické techniky White box testování (kdy se testují postupy, případně kód pouze vizuálně, bez spouštění či provozu).

Desk checking Statickou kontrolu provádějí programátoři před kompilací nebo spuštěním kódu.[5]

Kontrola kódu zkušenými programátory Skupina vysoce postavených technických zaměstnanců, kteří často disponují vysokými a dlouholetými znalostmi a zkušenostmi, kladou otázky autorovi kódu, který jim vysvětlí svůj postup a logiku a společně pak naleznou a opraví chyby [5]. Jedná se často o senior programátory, vedoucí týmů programátorů a product owners.

Formální kontrola Formální kontrola se sestává z několika na sebe navazujících fází:

1. **Plánování** - Ve fázi plánování se zařídí vhodné místo a čas schůzky účastníků kontroly.[5]
2. **Přehled** - Všem účastníkům kontroly je předána detailní dokumentace celkového návrhu kódu.[5]
3. **Příprava** - Na základě předchozích zkušeností s chybami v kódu (např. z předchozích projektů) se účastníci pokusí odhalit možné chyby v kódu.[5]
4. **Kontrola** - Nalezené chyby jsou na setkání předloženy autorovi kódu.[5]
5. **Oprava chyb** - Autor na základě těchto podnětů od účastníků kontroly opraví chyby.[5]

6. **Další kroky** - Moderátor schůzky provede případné další kroky k tomu, aby zajistil, že všechny nalezené chyby a závady byly odstraněny.[5]

1.3 Dynamické testovací techniky

U dynamických testovacích technik je potřeba spustit testovaný kód či aplikaci. Kód je spuštěn a tester (osoba zodpovědná za testování správnosti kódu) ověří funkčnost programu z hlediska toho, jaké chyby se v něm vyskytují a jestli jde program vůbec spustit. Dynamické testování zachycuje a opravuje vzniklé chyby, které nedokázalo podchytit testování statické. Nejedná se tedy už o prevenci vzniku chyb, ale o samotné řešení případně vzniklých chyb po spuštění kódu.[1]

Pro kapitolu dynamických testovacích technik mi sloužil jako hlavní zdroj a inspirace odborný článek o testovacích technikách s názvem *Black Box and White Box Testing Techniques - A Literature Review*[5] od autorů Srinivas Nidhra (Švédsko) a Jagruthi Dondeti (Indie). Autoři v tomto článku dokázali pokrýt celou škálu testovacích technik a jednoduše každou popsat.

Do dynamických technik patří testování typu Black box a White box.[3]

1.3.1 Funkční testování - Black box testing

Black box (neboli "černá skříňka") znamená, že tester vůbec nemusí rozumět tomu, jak testovaný kód programu funguje, aby mohl provádět svoji činnost - systém je pro něho "černou skříňkou", do které jsou zaváděny vstupní data a ze které jsou vyvedeny výstupní data. Testerovi stačí, aby znal účel programu a jak má program fungovat. Testeři shromažďují během celého vývoje softwaru požadavky zákazníků. Ty poté analyzují a na základě nich připravují testovací data, vymýšlejí hraniční příklady, testovací sekvence, jednotkové testy, atd. Kromě toho, že tester nemusí mít žádné znalosti programování, spočívá výhoda těchto typů testovacích technik v tom, že testování je vlastně prováděno z uživatelského (tzn. často i zákaznického) pohledu a pomáhá tak odstranit veškeré nejasnosti či nesrovnalosti v požadavcích zákazníků. [1][5]

Testování ekvivalence (Ekvivalence partitioning)

Testování spočívá v zadávání různých hodnot do programu. Tyto hodnoty by měly být ve správném rozsahu i mimo něj, aby se otestovala správná reakce programu i na špatné hodnoty. Typicky by měl program upozornit uživatele a navést jej ke správně zadaným údajům, nebo rovnou zajistit, aby jiné hodnoty uživatel nemohl vůbec zadat. Úspěšnost takového přístupu závisí na tom, zda-li je tester schopen

správně rozpoznat rozdíly mezi částí dat validních a částí dat invalidních (jestli je správně, včas a pravidelně informovaný o změnách).[5]

Abych tento typ testování vysvětlil na příkladu, představte si, že máte PLC spojené se systémem MES, do kterého má vedoucí výroby od vedení firmy přikázáno zadat plánovanou denní výrobu, aby PLC mohlo řídit produkci stroje či výrobní linky. Maximum může být až 2000 kusů (konkrétní výrobek zde není podstatný). Vstupní data se tak rozdělí na 1 validní interval a 2 invalidní intervaly:

Validní interval - interval od 0 do 2000 - program by měl přijmout zadaný údaj a přizpůsobit výrobu.

Invalidní interval I - interval od $-\infty$ do 0 - program by v tomto případě měl uživatele upozornit, že zadal mínusovou hodnotu a vysvětlit operátorovi, že taková hodnota je nesmyslná. Program by neměl zkolabovat nebo s takovým číslem počítat.

Invalidní interval II - interval od 2000 do $+\infty$ - ve druhém případě by měl program upozornit uživatele, že překročil dovolený rozsah a číslo zaokrouhlit na maximální úroveň, tedy 2000. Lepší program by se ještě například otázal na zvýšení limitu a odeslal by požadavek MES systému, kde by ho vedení schválilo a limit by se navýšil například na 2500. Program by opět neměl skončit neznámou chybou či kolapsem.

Analýza limitních hodnot (Boundary value analysis)

Tento typ funkčního testování je podobný testování ekvivalence (předchozí typ). Místo různých hodnot se zde ale testuje reakce systému na předem známé limitní hodnoty.[5]

Jako příklad bych uvedl registraci bankovního účtu. Obvykle v takové registraci uživatel vytváří své přihlašovací údaje, k nimž patří i heslo. Délka hesla bývá většinou omezena dolní hranicí z důvodu bezpečnosti. Uvažme možnost, že banka zavádí dolní hranici minimálně 8 znaků a horní hranici maximálně 20 znaků. Jako testovací data nám poslouží 4 hesla. 2 hesla s délkami 7 a 8 znaků pro otestování dolní hranice a 2 hesla s délkami 20 a 21 znaků pro otestování horní hranice. Testuje se zde jaká bude reakce programu, když budou data ještě v rozsahu a jak program zareaguje, když se dostaneme mimo rozsah.

Rozhodovací tabulka (decision table)

Rozhodovací tabulka je vlastně pravdivostní tabulka pro různé případy, které mohou nastat. Pro její vytvoření je třeba nejdříve stanovit vstupní požadavky, zjistit počet

možných kombinací a zapsat je do tabulky. Poté odlišit případy, kdy se má něco skutečně stát, akce pro tyto případy blíže určit a vytvořit pro ně specifické testovací případy.[5]

Abych to opět vysvětlil na příkladu souvisejícím s průmyslovou automatizací, představme si, že máme přízdvih trubky, který má za úkol vysunout trubku do pracovní pozice profukovací stanice. Tento přízdvih je však zajištěn interlocky - podmínkami, při jejichž splnění je možné s přízdvihem hýbat. Tyto podmínky jsou: 1) Dopravník stojí, 2) Chapadlo pro uchopení trubky otevřeno, 3) Tryska zasunuta

Tab. 1.1: Rozhodovací tabulka pro interlock přízdvihu

Dopravník stojí	Chapadlo otevřeno	Tryska zasunuta	**Je možné hýbat?**
NE	NE	NE	**NE**
NE	NE	ANO	**NE**
NE	ANO	NE	**NE**
NE	ANO	ANO	**NE**
ANO	NE	NE	**NE**
ANO	NE	ANO	**NE**
ANO	ANO	NE	**NE**
ANO	ANO	ANO	**ANO**

Strategie testování ortogonálního pole

Testování pomocí ortogonálního pole spočívá ve správném určení kombinací, pomocí nichž lze otestovat celý program a to bez nutnosti procházení všech kombinací. Hlavním cílem je z těchto kombinací složit ortogonální pole. Zatím však neexistuje žádná efektivní metoda či algoritmus, pomocí kterého by se dalo takové pole sestavit. Tester, který tuto techniku zvolí, se musí spoléhat na již předem napsané katalogy.[5][6]

1.3.2 Strukturální testování - White box testing

Strukturální testování softwaru (také známé jako White box testing technique) se používá hlavně pro detekování logických chyb v kódu programu. Soustředí se tedy hlavně na vnitřní logiku a strukturu kódu. Používá se pro debugování kódu, hledání náhodných typografických chyb a objevování chybných predikcí programátora. Testování touto technikou se provádí na nízké úrovni. Tester vidí zdrojový kód - většinou to bývá přímo vývojář, který rozumí vnitřní struktuře programu. Výhoda

tohoto typu testování je, že má tester dobrý přehled o tom, které části kódu již byly otestovány. Techniky White box testování jsou statické a strukturální.[1][5]

Statické metody již byly vysvětleny v kapitole 1.2.1, zde budou vysvětleny strukturální.

Vytvoříme-li podmínky pro testování a program necháme za těchto podmínek otestovat, kolik kódu jsme vlastně otestovali? **Pokrytí** je právě parametr uvádějící, kolik kódu jsme za daných podmínek dokázali otestovat na chyby. Pro otestování zbytku kódu musíme upravit testovací sekvenci a znovu spustit program.

Statement coverage (pokrytí řádků) - V tomto testování se testují řádky kódu.

$$P_R = \frac{R}{R_C} \cdot 100\ [\%] \tag{1.1}$$

kde:

P_R...pokrytí (řádky),

R...počet zkontrolovaných řádků programu,

R_C...celkový počet řádků

(Písmena použitá v této rovnici nejsou nijak oficiálně určená, slouží pouze pro lepší přehlednost rovnice.)

Výsledek se udává v procentech.[5][7]

Pokud existuje například kód, ve kterém dochází k rozvětvení pomocí IF podmínky a výraz podmínku splňuje, zkontrolují se tak příkazy uvnitř tělíčka podmínky IF, ale už ne příkazy uvnitř tělíčka podmínky ELSE, která za IF následuje. Pokryjeme tak jen určitý počet řádků.[7]

Branch coverage (pokrytí rozvětvení) - Míra pokrytí se u tohoto testování měří podle zkontrolovaných rozvětvení.

$$P_V = \frac{V}{V_C} \cdot 100\ [\%] \tag{1.2}$$

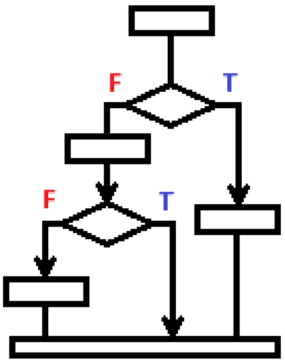
,kde:

P_V...pokrytí (rozvětvení),

V...počet zkontrolovaných větví programu,

V_C...celkový počet větví

Výsledek se udává v procentech.[5][8]



Obr. 1.1: Příklad pro Branch coverage

Pro lepší vysvětlení problematiky jsem nakreslil obrázek 1.1. Jedná se o malý zjednodušený vývojový diagram. Šipky znázorňují jednotlivá rozvětvení, v diagramu se tak nachází 4 rozvětvení. Pokud se například stanoví podmínky pro testování tak, že první podmínka IF bude splněna (vydáme se cestou napravo v diagramu), otestuje se pouze jedna větev ze čtyř, což znamená pokrytí 25 %.[8]

Condition coverage (pokrytí stavů) Zde jde o otestování případů, které mohou být otestovány správně a zároveň obsahují příkaz, který může způsobovat v určitých případech problémy. Jedná se například operování s nepřiřazenými proměnnými nebo dělení nulou ve výrazu. Je třeba tedy vzít při testování v úvahu i specifické případy, pokusit se je otestovat a na základě vzniklých chyb navrhnout řešení. Je třeba otestovat všechny výrazy podmínek v každé větvi.[5][9]

Testování hlavní cesty (Basic path testing) Testování hlavní cesty se provádí pomocí tokových grafů, které obsahují uzly (v grafu kolečka), spojovací šipky, oblasti přechodu mezi šipkami a uzly (regiony) a podmínkové uzly (obsahující podmínku).[5]

Testování kvality kódu na základě cyklomatické složitosti

Cyklomatická složitost (CYC) "Cyklomatická složitost programu je parametr, který udává kvantitativní stupeň logické obtížnosti programu."[5] Dokumentace Microsoftu má jednodušší popis: "Složitost cyklomatická se definuje jako měření 'míry rozhodovací logiky ve funkci zdrojového kódu'. Jednoduše řečeno: Více rozhodnutí, která je nutné provést v kódu, znamená složitější kód."[10]

$$\mathbf{V(G) = e - n + 2} \tag{1.3}$$

V(G)...cyklomatická složitost kódu

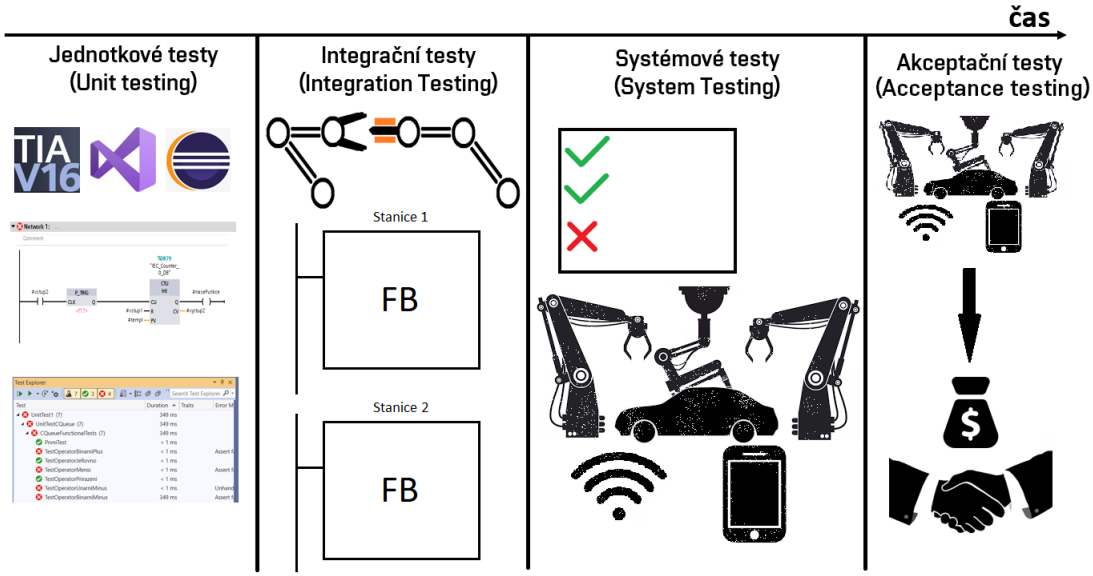
e...počet šipek v diagramu programu

n...počet uzlů v diagramu programu

Obecně platí, že čím je menší cyklomatická složitost kódu, tím je menší šance na výskyt chyb v programu, program je přehlednější a méně složitý.[5]

1.4 Úrovně testování

Celé testování prochází čtyřmi úrovněmi, aby se testováním pokrylo pokud možno nejvíce různých případů a opravilo se největší množství chyb před prodejem zákazníkovi či zavedením softwaru do provozu.[1][11]



Obr. 1.2: Testovací úrovně v čase - názorný obrázek

1.4.1 Jednotkové testy (Unit tests)

Jednotkové testy jsou nejnižší úrovní testování. Testují se zde individuálně jednotlivé komponenty programu: funkce, třídy objektově orientovaného programování, metody, skripty, atributy tříd, konstruktory, destruktory.[1][11]

U PLC programování se úroveň jednotkových testů analogicky týká testování funkcí, funkčních bloků a jednotlivých networků.

Tyto testy jsou nejdůležitější pro celý vývoj SW.

1.4.2 Integrační testy (Integration testing)

Integrační testy ověřují, zdali části programu mezi sebou komunikují, předávají si data a vzájemně reagují. Ověřují, jestli je část programu správně integrována.[1][11]

Například pokud je linka, která se sestává z několika různých stanic, testuje se, jestli mezi sebou funkční bloky těchto stanic správně reagují, zejména v automatickém režimu.

1.4.3 Systémové testy (System testing)

Systémové testy slouží k otestování kompletně integrovaného systému. Testují se zde často využívané části programu z pohledu uživatele, nejlepší skupinou testovacích technik je tu Black box testování, kdy program testují na chyby lidé, kteří nerozumí fungování samotného kódu. Systémovými testy se ověřuje, zdali systém splňuje veškeré požadavky zákazníka.[1][11]

Pokud si opět představíme výrobní linku, testujeme ji jako celek - funguje-li správně ve všech režimech, splňuje-li nároky na celkový takt, komunikuje-li s nadřazenými systémy, atd.

1.4.4 Akceptační testy (Acceptance testing)

Akceptační testy jsou testy správnosti či přijatelnosti (acceptance) softwaru ze strany zákazníka. Dochází tak k testování ze strany zákazníka a účelem je vyhodnocení soulad systému s obchodními podmínkami a požadavky. Vyhodnocuje se tak, jestli je program pro zákazníka přijatelný a může být zákazníkovi plně předán.[1][11]

Naposledy se podíváme na příklad výrobní linky. Ta byla v případě akceptačních testů poslána zákazníkovi a začlenila do výroby druhé firmy, která v zakázce figurovala jako zákazník či zadavatel. Druhá firma (zadavatel) nyní testuje program z pohledu začlenění celé objednané linky do výrobního řetězce a hlavně z hlediska návratnosti této investice.

1.5 Testování PLC kódu

Tato kapitola pojednává o testovacích technikách PLC kódu.

1.5.1 Manuální a automatické testování

V této kapitole jsou přiblíženy techniky manuálního a automatického testování PLC kódu.

Manuální testování

Manuální testování PLC kódu jednoduše spočívá ve vizuální kontrole jeho chování. Vyžaduje, aby měl tester k dispozici příslušné vývojové prostředí, ve kterém byl PLC program vyvinut. Toto prostředí je propojeno s PLC, kde program běží. Je tak možné například sledovat stav proměnných přímo v kódu. Tester má k dispozici katalog instrukcí, podle kterých se řídí a otestuje tak všechny možné případy a zapíše k nim odpovídající výsledky.[12]

Jak si můžeme všimnout, jedná se o dynamické strukturální testování kódu (White box). Tester nemusí být přímo PLC programátor, ale musí PLC kódu rozumět, kontroluje jeho strukuturu. Manuální testování spadá do úrovně jednotkových a integračních testů.

Automatické testování

Automatické testování PLC kódu má tendenci nahrazovat testera a samostatně posílat testovací příkazy do PLC pomocí testovacích skriptů. Toto mohou typicky provádět systémy SCADA. Automatické testování spadá do úrovně systémových testů (testuje se až když celý systém funguje jako celek). [12]

1.5.2 Formální testování

V této kapitole jsou přiblíženy techniky formálního testování PLC kódu.

Axiomatické testování

Tato formální metoda spočívá v kontrole pomocí trasování změn stavů od počátečních až po konečné podmínky. Jedná se o metodu podobnou testování pomocí stavových grafů. Testeři či inženýři si nakreslí program jako stavový automat a na základě přechodových podmínek testují všechny stavy programu.[12]

Algoritmické testování

Algoritmické formální testování je založeno na kontrole modelu systému. Není tedy testovaný samotný reálný systém, ale je testovaný jeho model vůči formálním podmínkám. V průmyslové praxi se využívá právě tohoto typu testování. Takto modelovat systémy (např. pomocí stavových grafů) a testovat jejich chování lze například pomocí programu UPPAAL.[12]

2 Návrh aplikace pro testování PLC kódu

Tato kapitola pojednává o požadavcích na aplikaci, jaké konkrétní chyby má za úkol aplikace nalézat, případně jaké problémy bylo potřeba při návrhu vyřešit.

2.1 Cíl aplikace

Hlavním cílem aplikace pro testování PLC kódu je kontrola častých chyb v projektu, upozornění programátora na tyto chyby a možnost opravy těchto chyb. Chyby jsou blíže popsány v následující podkapitole. Aplikace bude vyvíjena pro použití ve firmě ICE Industrial Services, kde ji budou využívat PLC programátoři jako pomůcku při kontrole PLC kódu v prostředí TIA Portal V16. Aplikace by měla umět najít otevřený projekt v TIA portalu, najít a klasifikovat chyby v projektu, upozornit vývojáře na tyto chyby a umět výpis chyb vyexportovat do textového souboru. Aplikace by měla být přehledná.

2.2 Časté chyby v PLC projektu (TIA Portal)

Chyby, které v projektu prostředí TIA Portal vznikají, bývají nejčastěji způsobeny generátory PLC kódů, které jsou v dnešní době často využívány pro počáteční vygenerování programu, který se pak musí opravovat podle potřeb daného projektu. Opravy a úpravy vygenerovaného kódu již musí provádět programátor. Vygenerování projektu slouží pouze jako počáteční odhad kostry programu na základě zkušeností z minulých projektů a na základě předložené projektové dokumentace. Chyby pak může programátor přehlédnout, nebo je sám vytvořit při úpravě programu. Chyby také mohou vzniknout na základě nových změn v projektu (přejmenování prvků, změny požadavků zákazníka, nové prvky v projektu, atd.).

2.2.1 Klasifikace chyb

U chyb, které popisuji v kapitolách 2.2.2 a 2.2.3 udávám jejich klasifikaci - rozdělení na Warning a Error. Toto rozdělení by bylo vhodné vysvětlit.

Warning Pokud se jedná o warning, chyba není tolik závažná jako error. Jedná se zpravidla o chybu estetickou, která nemá na chod programu ani komunikaci mezi zařízeními za žádných okolností žádný vliv. Může však zmást vývojáře nebo způsobit nesrovnalosti.

Error Chyba typu Error je závažnější než warning, protože může způsobovat nesprávný chod programu, způsobovat nesprávný sled instrukcí nebo způsobovat chyby v komunikaci mezi zařízeními. Tato chyba není chyba estetická, ale chyba funkční. Má tedy větší závažnost než warning.

2.2.2 Chyby v hardwarové konfiguraci projektu

V této kapitole jsou shrnuty všechny chyby, které se týkají hardwarové konfigurace projektu v TIA Portalu.

Výchozí pojmenování komponent

Výchozí pojmenování komponent je chyba obvykle způsobena počátečním generováním projektu. Nebo může být způsobena tím, že programátor přidal do projektu zařízení bez pojmenování pouze na otestování a později se rozhodl zařízení v HW konfiguraci ponechat (nebo jej zapomněl odstranit). Může být ale způsobena i změnou oficiálního názvu zařízení v dokumentaci projektu. Například pokud přidám do projektu nové zařízení typu PLC, vždy se mu přiřadí název ve tvaru PLC_**X**, kde X je číslo podle toho, kolik se takových PLC s výchozím názvem v projektu nachází. Takto by se zařízení nemělo jmenovat - mělo by být přejmenováno podle firemní dokumentace daného projektu. Nesprávné či nejednoznačné pojmenování zařízení je snadno opravitelná chyba, která nezpůsobuje větší problémy s fungováním programu a řízení výrobních procesů. Může ale zmást vývojáře, nebo způsobit neshody ve vedení týmu, který má projekt na starosti.

- Klasifikace chyby: **Warning**
- Exkluzivní pouze pro firemní standard: **NE**

Nepřiřazená zařízení v rámci PN topologie

Nepřiřazené zařízení opět mohou být způsobeny generátorem kódu i přehlédnutím této chyby vývojářem. Samotný vývojář většinou nezapomene přiřadit prvky v topologii, avšak se to může stát. Zařízení mezi sebou nebudou správně komunikovat, pokud nejsou správně nakonfigurovány, propojeny a nejsou ve správné podsíti. Pokud mezi sebou nemohou zařízení komunikovat, rozhodně nemůže projekt fungovat. Chybu lze opravit správným přiřazením zařízení v rámci Topology view v hardwarové konfiguraci projektu.

- Klasifikace chyby: **Error**
- Exkluzivní pouze pro firemní standard: **NE**

2.2.3 Chyby ve funkcích a funkčních blocích

Funkční blok (dále jen FB) se od funkce (dále jen FC) liší v tom, že na rozdíl od ní uchovává data v datových blocích (dále jen DB) či ve vnitřních proměnných typu *Static* a může je použít v další iteraci programu či v dalším zavolání stejného FB. FC mohou vracet hodnoty, ale nemohou si ukládat data jako FB.

Nepojmenovaný network

Příčinou bývá chybné generování kódu pomocí firemních generátorů, přehlédnutí nepojmenovaného networku vývojářem, založení a nepojmenování networku vývojářem v rámci jeho myšlenky či změněných informací v dokumentaci projektu. Pojmenování networků slouží pouze pro snadnou orientaci v rámci FC či FB, nepojmenované FC/FB tak nejsou nijak ovlivněny, avšak ztrácí se přehlednost. FB často obsahují stovky networků a každý z nich plní svůj úkol. Špatné či žádné názvy networků nevedou tedy k funkční chybě, ale vedou k velkému chaosu v komunikaci mezi vývojáři, v úpravách a opravách kódu, v hledání v kódu, atd.

- Klasifikace chyby: **Warning**
- Exkluzivní pouze pro firemní standard: **NE**

Nesprávně použité symboly/adresy ve vztahu k dané stanici

Příčinou bývá chybné generování kódu pomocí firemních generátorů, přehlédnutí chybného použití proměnné vývojářem. Nejčastěji dochází ke špatnému vygenerování názvu snímačů: Mějme například chybu, kdy ve stanici 220 může být použita proměnná *"STATUS_240"."UH10-BG01"*, nejspíše určena pro stanici 240. Správně by měla být použita proměnná *"STATUS_220"."UH10-BG01"*. Nesprávně používané proměnné vedou k chybnému chování programu. V lepším případě mohou pouze zmást vývojáře při kontrole vstupů a výstupů, v horším případě mohou například povolit interlock u jiné stanice a může pak vzniknout i ztráta, zničení či znehodnocení vyráběného materiálu, skřípnutí něčí ruky, zranění, atp. Ne vždy však je použitá proměnná chybně - někdy například skutečně je zapotřebí signál z cizí stanice. Tuto chybu musí vždy vývojář samostatně vyhodnotit.

- Klasifikace chyby: **Error**
- Exkluzivní pouze pro firemní standard: **ANO**

Zapisování do TEMP (dočasných) proměnných až po jejich čtení

Příčinou bývá chybný sled kódu. V případě čtení proměnných Temp před jejich přepisem bude docházet k čtení prázdných hodnot a program nepřečte validní informaci. Proměnné TEMP jsou při každém cyklu programu vymazány.

- Klasifikace chyby: **Error**
- Exkluzivní pouze pro firemní standard: **NE**

Používaná přiřazení do proměnných v rámci FB/FC na více místech

Častými příčinami jsou chybné generování kódu, nezkontrolování chyb, nechtěné vytvoření chyby vývojářem. Typicky se jedná o 2 cívky (coil), přepisující stejnou proměnnou v programu. Chyba může způsobit přepisování hodnot v rámci jednoho průchodu programem. To způsobuje nesprávnou funkčnost programu a navíc zmatení programátora, kterému na první pohled nemusí dojít, proč se program chová jinak než je jeho předpoklad.

- Klasifikace chyby: **Error**
- Exkluzivní pouze pro firemní standard: **NE**

V bloku není použita deklarovaná proměnná

Chyba může vzniknout vygenerováním proměnné navíc v rámci předchozí myšlenky v dokumentu projektu, vymazáním části kódu, který se stal nepotřebným a v kterém byla proměnná použita nebo založením a opomenutím nové proměnné programátorem. Proměnná zbytečně zabírá paměťové místo v PLC, ale nijak neškodí ani neovlivňuje chod programu. Při větším počtu mohou nepoužívané proměnné způsobovat chaos a mást vývojáře. Kompilace programu v samotném TIA Portalu tuto chybu neumí rozpoznat.

- Klasifikace chyby: **Warning**
- Exkluzivní pouze pro firemní standard: **NE**

Pro více instancí často používaných funkcí použita stejná proměnná (instance)

Zde opět může být několik příčin jako nesprávné generování kódu, kopírování FB bez přepisu použitého DB a bez přepisu vstupně výstupních proměnných, atd. Chyba je vztažena zejména na nejčastěji používané FB, jako je timer, counter, p_trig, r_trig. Pokud máme FB jednoho typu na více místech a zapomeneme přepsat jemu příslušný DB, bude docházet k přepisování hodnot mezi těmito FB. Nemusí to být ale pouze FB, existují základní instrukce, které také využívají instancí. Právě nejčastěji bývá tato chyba zastoupena u časovačů a čítačů.

- Klasifikace chyby: **Error**
- Exkluzivní pouze pro firemní standard: **NE**

Příliš velký počet instrukcí v rámci jednoho networku

Nejčastěji je velký počet instrukcí pravděpodobně zapříčiněn experimentováním s programem nebo vymýšlení zbytečně složitého programu. To má za následek zdržení se programu na jednom networku déle než na jiných, nepřehlednost a zvýšení šance na vznik chyby. Zároveň vývojář musí scrollovat - posouvat obrazovku, aby viděl kód (ať už vertikálně či horizontálně). Ve firmě se většinou pracuje na noteboocích s běžným rozlišením FullHD (1920x1080 pixelů) a proto je třeba na tuto velikost navrhnout maximální horizontální i vertikální délku networku. Velký počet instrukcí v rámci jednoho networku jinak není nic, co by narušovalo správný chod programu, pokud v networku nejsou jiné chyby. Jedná se tedy warning.

- Klasifikace chyby: **Warning**
- Exkluzivní pouze pro firemní standard: **NE**

V programu jsou použité M merkery s pojmenováním STDComm

Ve firmě se používají M merkery - globální proměnné (např. M100.0). M merkery mohou být použity ve firemních FC/FB, označených jako STD - standardní. Jedná se o FC či FB vytvořené firmou a používané napříč projekty pro usnadnění práce programátorů. Většinou jsou vždy označeny jako STD + název. Firma požaduje, aby aplikace označila použití M merkeru s názvy začínajícími "STDComm" jako chybné.

- Klasifikace chyby: **Error**
- Exkluzivní pouze pro firemní standard: **ANO**

2.3 Návrh aplikace

V této kapitole je pojednáno o zvoleném vývojovém prostředí a jazyce, práci s XML soubory a návrh struktury a vzhledu aplikace.

2.3.1 Vývoj aplikace

Aplikace je vyvíjena v prostředí Visual Studio a jedná se o Windows form application. Požadavky na aplikaci totiž jsou, aby uměla přehledně a názorně zobrazit všechny nalezené chyby a byla snadno ovladatelná. Windows form application je v tomto ohledu nejlepší volbou ze dvou důvodů. První důvod je ten, že s vývojem aplikace tímto způsobem již mám nějaké zkušenosti z minulosti a druhý důvod je, že skutečně splňuje požadavek na přehlednost. Zároveň se zde snadno pracuje s rozhraním TIA Portal Openness. Běžně se v tomto prostředí vytvářejí například generátory PLC kódu.

Aplikace je vyvinuta v jazyce C#, který vychází z jazyků C a C++, je čistě objektově orientovaný a má přístup ke třídám knihovny .NET Framework [14].

Veškeré jmenné prostory, třídy, metody a objekty nesou anglický název, vystihující jejich funkcionalitu. Názvy tlačítek a nápisů na vzhledu aplikace jsou také v anglickém jazyce.

2.3.2 Požadavky na aplikaci

Aplikace musí splňovat základní funkcionalitu - umět vyhledat veškeré chyby, popsané v kapitole 2.2. Aplikace by měla být přehledná a umožnit chyby označit. Aplikace by se měla po spuštění umět připojit k otevřenému projektu v TIA Portalu, případně umožnit projekt zavřít, či z ní otevřít jiný projekt. Měla by umět uživatele upozornit na to v jaké fázi se nachází testování projektu na chyby - například ukazatelem průběhu testování - progress barem (testování velkých projektů může probíhat v řádu minut až desítek minut).

2.3.3 Důležité pojmy spojené s jazykem C#

V této kapitole jsou shrnuty některé pojmy z prostředí objektově orientovaného programování v jazyce C#, které se vyskytují v dalších částech této práce.

Jmenný prostor - *namespace*

Jmenné prostory mohou sdružovat třídy, metody, ale i podprostory, či podřízené jmenné prostory. Zahrnutí jmenného prostoru do souborů aplikace lze provést klíčovým slovem `using`. [14]

Třída - *Class*

Třída `class` je v C# velmi podobná struktuře `struct`, ale hlavní rozdíl mezi třídou strukturou spočívá v jejich prvcích. Prvky struktury bývají obyčejné datové členy, zatímco prvky třídy jsou atributy a metody (a samozřejmě i datové členy jako u struktury). Dalším rozdílem je, že ve třídě je možné definovat k jednotlivým objektům přístupová práva - povolení ostatním třídám k přístupu k jejím objektům a metodám. Pokud je metoda či objekt označen jako *public*, může být použit i mimo svoji třídu. Pokud je ale označen jako *private*, může být použit pouze v rámci své třídy. Ve výchozím nastavení jsou všechny členové třídy privátní.[16]

Třídy nejčastěji reprezentují objekty. Například, pokud máme třídu, která reprezentuje objekt automobil, máme v ní objekty, které jsou s ní spojené - dveře, volant, motor apod. [14]

Metoda - *Method*

Metody jsou v C# obdobou funkcí, na rozdíl od nich však náleží své nadřazené třídě. Metody mohou být různých datových typů a stejně jako funkce vrací výsledky (kromě metody typu *void*). Metody se volají s parametry a mohou být privátní (*private*), veřejné *public* či chráněné *protected*.[16]

Objekty - Instance tříd

Třídy jsou vlastně předpisem pro vytvoření jejich instancí. Tyto instance nazýváme objekty. Ve WinForms je typickým objektem například tlačítko. Můžeme jich mít libovolné množství a podle toho, kolik jich máme, vznikne tolik objektů typu tlačítko (odlišených od sebe názvem objektu). Třídy jsou tak pouze předpisem pro vznik objektů. [14]

Atributy objektu

Každý objekt nese své atributy, tedy vlastnosti. Například u tlačítka je to jeho barva, barva písma, barva okraje, text písma či vlastnosti písma. Atribut může být obyčejný prvek - `field`, nebo lépe `property` - vlastnost objektu. Hlavní rozdíl mezi nimi je v tom, že `field` je obyčejná proměnná, zatímco `property` je člen objektu, který poskytuje flexibilní mechanismus čtení, zápisu či výpočtu proměnné [13]. Protože se `property` používá u většiny tříd, které v C# používáme, budu tuto konvenci dodržovat a také místo `field` používat `property`.

```
public int cislo { get; set; }
```

Listy - seznamy

Seznamy jsou v C# podobné polím či vektorům v C++, poskytují různé metody na operaci s nimi a s jejich prvky a k jednotlivým prvkům lze přistupovat stejně jako k prvkům pole pomocí indexace.

Cyklus foreach

Jedná se o jednodušší zápis for cyklu, kdy není nutno použít indexaci prvků. Slouží pro procházení instancí tříd, ať už v kompozicích či seznamech - listech (viz kapitola 2.3.3). Cyklus `foreach` prochází všechny prvky v určité kolekci prvků (pole, seznam, kompozice prvků, atd.). Pokud existuje ještě další element v kolekci, přepne na jeho instanci a vykoná se pro tuto instanci tělo cyklu. Pokud v kolekci již prvek není, z `foreach` smyčky se vychází.[15]

2.3.4 Soubory XML

Soubory XML jsou napsané v jazyce XML (eXtensible Markup Language). Jedná se o značkovací jazyk odvozený od SGML (Standard Generalized Markup Language), vyvinut komunitou World Wide Web Consortium (W3C). Dokument, vytvořený v jazyce XML je strukturovaný dokument. Prezentace XML předpisu je oddělena od jeho skutečného obsahu. Proto je pro počítač snáze čitelný a XML soubor může být přečten libovolným XML prohlížečem. Díky takto strukturovanému dokumentu je pak jednoduché přistupovat k jeho složkám velmi jednoduše. Jazyk XML je univerzální a tudíž velmi rozšířený jazyk, používaný například u webů. Charakteristická vlastnost XML je datová nezávislost aplikací, webových ale i klasických aplikací a databází.[14]

Při tvorbě XML souboru se musí nejdříve nadefinovat o jaký typ dokumentu se jedná, v jaké verzi XML a v jakém kódování znaků je uložen. Tomu se říká hlavička XML souboru. Poté následuje tělíčko, které je členěno na elementy označené jako *<NazevElementu>* a ukončené jako *</NazevElementu>* Pokud je element bez dalšího popisu, představuje prázdnou složku. Pokud je však ve tvaru *<Nazev Hodnota = "5"Popis = "Neznamy"*, bude se jednat o složku obsahující položky *Hodnota* a *Popis*. Vnořené elementy představují složky ve složkách.[14]

Práce s XML souborem

V první řadě je vhodné mít v referencích zahrnutou knihovnu `System.Xml`. Pro správnou implementaci v rámci projektu je třeba ji ještě zahrnout pomocí:

```
using System.Xml;
```

Načítání dat lze provést různými způsoby. Například pomocí série příkazů:

```
XmlDocument MyXMLDoc = new XmlDocument();
MyXMLDoc.Load(file);
```

Proměnná *MyXMLDoc* je typu XmlDocument a jsou v ní strukturovaně uloženy data z XML souboru *file*. K jednotlivým prvkům nyní lze snadno přistupovat jako k prvkům struktury.

2.3.5 TIA Portal Openness

TIA Portal Openness představuje otevřené rozhraní pro TIA Portal. Znamená to, že pomocí tohoto rozhraní můžeme propojit TIA Portal s externími programy. Díky tomu lze automatizovat část práce PLC programátorů, lze generovat PLC kód, lze kontrolovat a upravovat PLC kód. Openness je určen primárně pro editování PLC

projektu, knihoven, PLC kontrolérů a HMI v TIA Portalu pomocí externích aplikací, nejčastěji vyvinutých ve Visual studiu v jazyce C#.[17][18]

Funkční možnosti TIA Portal Openness

Rozhraní Openness umožňuje spouštět prostředí TIA Portal s uživatelským rozhraním i bez uživatelského rozhraní, kdy TIA Portal běží pouze na pozadí. Umožňuje zavírání TIA Portalu. Lze se k TIA Portalu připojovat a odpojovat. Openness umožňuje práci s projektem - můžeme přes vyvinutou aplikaci vytvořit nový projekt, otevřít projekt nebo jej ukládat a kontrolovat v něm změny. Z projektu lze vyčíst hardwarovou konfiguraci a modifikovat jednotlivé hardwarové prvky v ní (PLC kontroléry, HMI panely, průmyslové počítače, motory, servopohony, IO link senzoriku, řízení pneumatiky, apod.). V projektu lze vytvářet či mazat složky.[17][18]

Další funkční možností rozhraní Openness je práce s knihovnami. Lze otevírat a zavírat globální knihovnu TIA Portalu a zkontrolovat aktualizace instancí. Je možné aktualizovat projektové knihovny a s nimi spojené prvky v projektu (použité FC, FB, vizualizační prvky - faceplate), prvky lze v knihovně vyhledat a odstranit. Openness možňuje kopírování knihovních funkcí v projektu.[17][18]

Nejdůležitější funkční možností rozhraní Openness je možnost manipulace se složkami řídících kontroléru - PLC a se složkami vizualizace - HMI. Pomocí Openness lze konfigurovat spojení, připojení a odpojení TIA Portalu k PLC. Bloky, PLC tagy, proměnné a konstanty lze kompilovat, exportovat do XML a importovat z XML. Přes Openness lze pracovat s obrazovkami HMI, měnit grafiku. Je možné vytvářet spojení (Connections) mezi PLC a HMI. Dále lze vytvářet, exportovat, importovat či mazat proměnné, skripty, cykly, text a graphics listy. [17][18]

Jak implementovat Openness do projektu ve Visual studiu

Aby bylo možné vytvořit program, který bude pomocí Openness pracovat s TIA Portalem, je v první řadě vhodné mít standardizovanou strukturu projektu a možnost výměny dat mezi inženýrskými softwary (např. TIA Selection tool, EPLAN, apod.). Poté již lze vytvářet aplikace, které s TIA Portalem mohou interagovat. Pro programování TIA Openness ve Visual studiu je třeba implementovat knihovny *Siemens.Engineering* a *Siemens.Engineering.Hmi* do referencí projektu. Visual studio rozpozná tyto knihovny jako jmenné prostory (`namespace`). Pokud je potřeba použít některou ze tříd z cizího jmenného prostoru, je třeba jmenný prostor zařadit do kódu použitím klíčového slova `using`.[17]

Možnosti přímého využití TIA Portal Openness v mojí aplikaci

Pomocí TIA Openness je možné přímo načíst data otevřeného projektu. Je možné pracovat se zařízeními v projektu, přistupovat k jejich hardwarové konfiguraci a jejich názvům. To znamená že je přímo možné najít obě "hardwarové" chyby, které byly specifikovány v kapitole 2.2.2. Například použitím podprostoru

```
using Siemens.Engineering.HW.Features;
```

lze používat třídu `NetworkPort`, která je důležitá pro detekci chyby typu nepojmenované zařízení.

2.3.6 Hlavní problém Openness v rámci použití v mé aplikaci

Rozhraní TIA Portal Openness umí přistupovat k většině informacím o projektu a jeho hardwarové konfiguraci, ale pro potřeby testovací aplikace neumí to nejdůležitější - přistupovat k jednotlivým networkům a proměnným FC a FB a k obsahu networků. Tento problém se zdál na začátku vývoje aplikace vysoce limitujícím a tak jsem musel najít alternativní řešení. Toto alternativní řešení spočívá v exportu bloků do XML předpisu a následné načtení tohoto předpisu do třídy `ProjectBlocks`, kterou jsem pro to speciálně vytvořil.

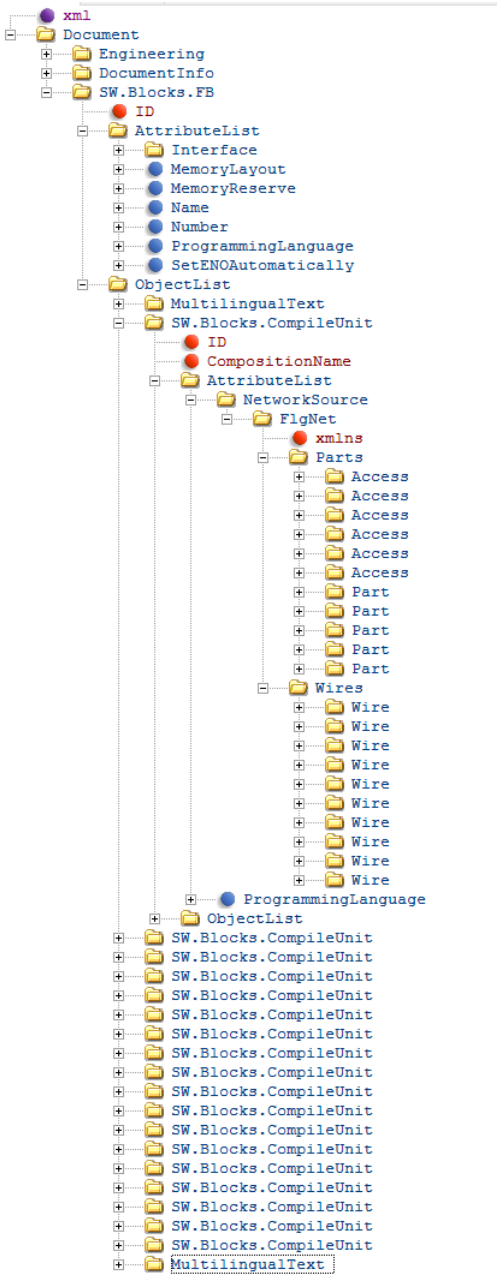
XML předpis PLC bloků

TIA Portal Openness sice neumí přistoupit k networkům, ale umí přistoupit k alespoň základním informacím o PLC blocích jako je jejich název, typ, číslo, atp. Zároveň umí exportovat jednotlivé bloky pomocí funkce *Export* do XML souboru. V tomto XML souboru se už nachází informace o všech networcích. Elementy *Engineering* a *DocumentInfo* jsou pro potřeby aplikace nepotřebné. Element *SW.Blocks.FB* (může být OB, FB nebo FC v závislosti na tom, o jaký typ bloku se jedná) je nejpodstatnějším prvkem. Obsahuje totiž další elementy *AttributeList* a *ObjectList*. *AttributeList* obsahuje veškeré informace o proměnných bloku (představuje interface bloku) a element *ObjectList* obsahuje veškeré networky, jejich informace a především jejich obsah.

Na obrázku 2.1 je možné vidět element *Interface*, jehož obsah by se do obrázku již nevešel. Obsahuje však elementy *Sections*, které představují jednotlivé typy proměnných PLC bloku - *Input*, *Output*, *InOut*, *Temp*, *Static*, *Constant* či *Return*. V těchto elementech se nachází podelementy *Members*, které představují již jednotlivé proměnné PLC bloku a drží veškeré informace o nich.

Dále je možné na obrázku 2.1 vidět elementy *SW.Blocks.CompileUnit*. Přesně ty představují jednotlivé networky. V elementu *FlgNet* se poté nachází obsah networku.

Jedná se o elementy *Accesses*, *Calls*, *Parts* a *Wires*.



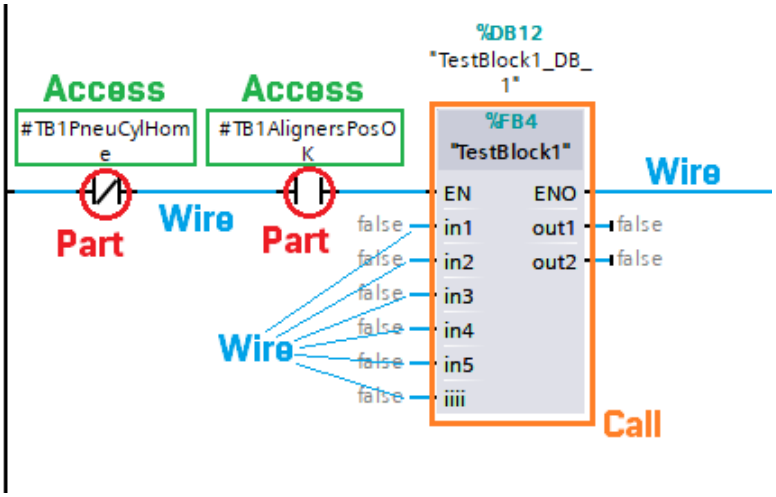
Obr. 2.1: XML předpis FB, zobrazený pomocí XML editoru

Access představuje proměnnou, která je obsažena v rámci networku. Nese informaci o jejím typu, jestli je globální či lokální (lokální znamená proměnnou tohoto bloku - z interface), jestli je to proměnná či konstanta a hlavně informaci o jejím unikátním ID.

Part představuje použitou instrukci. Může se jednat o kontakt, cívku, move, timer, counter, matematickou operaci, atd.

Call je volání FC či FB uvnitř logiky networku. Call obsahuje v případě FB element *Instance* - přiřazenou proměnnou - instanci bloku a v případě FC i FB elementy *Parameters*, obsahující název, typ proměnné a informaci, zdali se jedná o vstup či výstup.

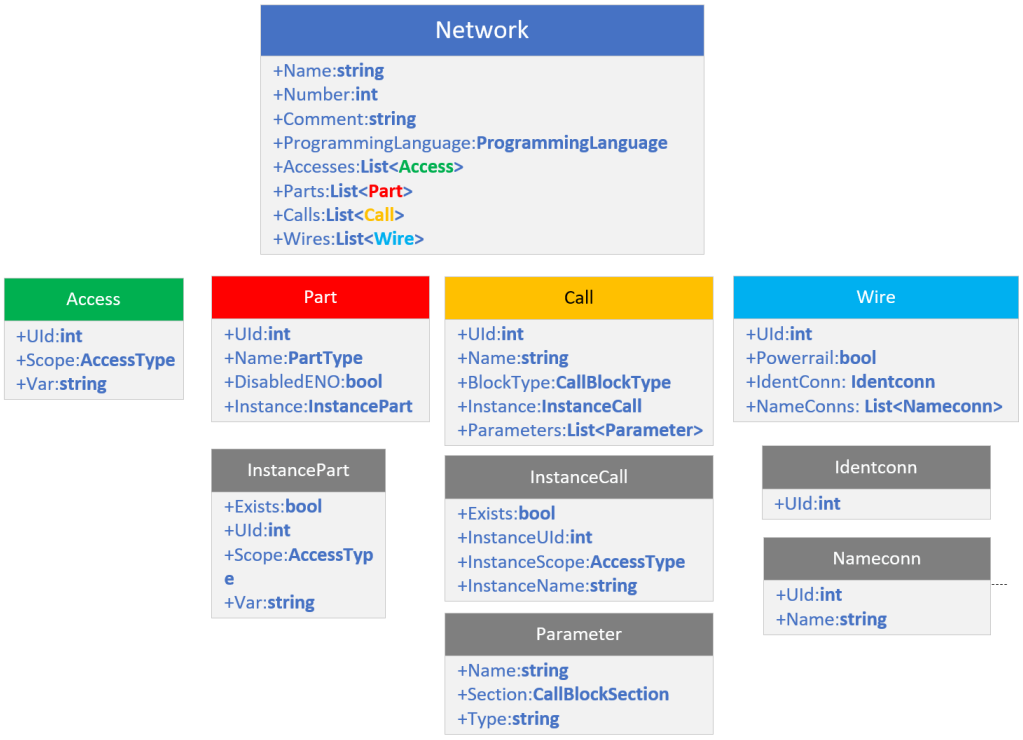
Wire přináší smysl a posloupnost mezi *Accessy*, *Cally* a *Party*. Obsahuje totiž informace na základě kterých je možno jednoznačně určit propojení jednotlivých instrukcí a funkčních volání mezi sebou a jaké proměnné (*Accessy*) jsou k nim přiřazeny a to pomocí UId napojení, která jsou shodná s UId *Partů*, *Callů* a *Accessů*. Jsou 2 typy napojení. Nameconn je napojení na *Part* či *Call*, Identconn je napojení na *Access*. Pokud *Wire* propojuje dvě instrukce, má dvě napojení typu *Nameconn*.



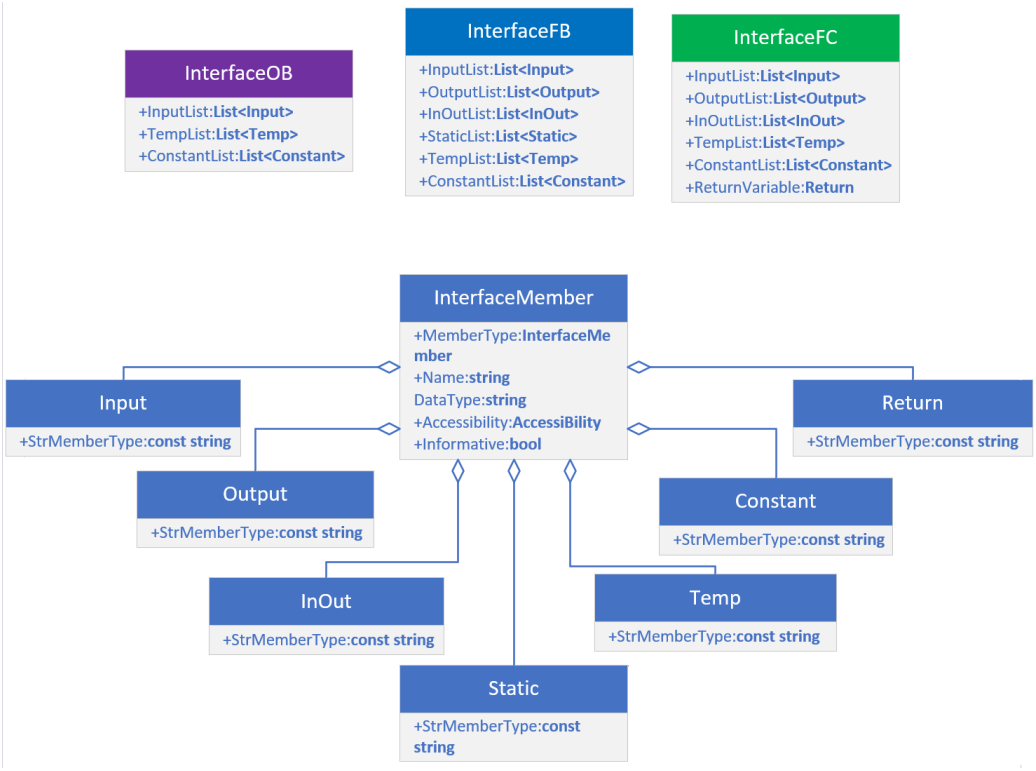
Obr. 2.2: *Accessy*, *Cally*, *Party* a *Wiry* v PLC kódu

Třída ProjectBlocks

Vyčítaná data by bylo vhodné organizovaně ukládat. Proto byla vytvořena třída `ProjectBlocks`, která má za úkol sdružovat veškeré informace získané z XML předpisu PLC bloku. Obsahuje seznamy všech OB, FB a FC nalezených v projektu. Tyto bloky jsou opět představeny třídami s názvy `OrganisationBlock`, `FunctionBlock` a `Function`. Ty dále obsahují další třídy - `Interface` a `Networks`, obsahující seznamy proměnných a networků. Podrobnější vysvětlení členění je zobrazeno na následujících diagramech tříd.

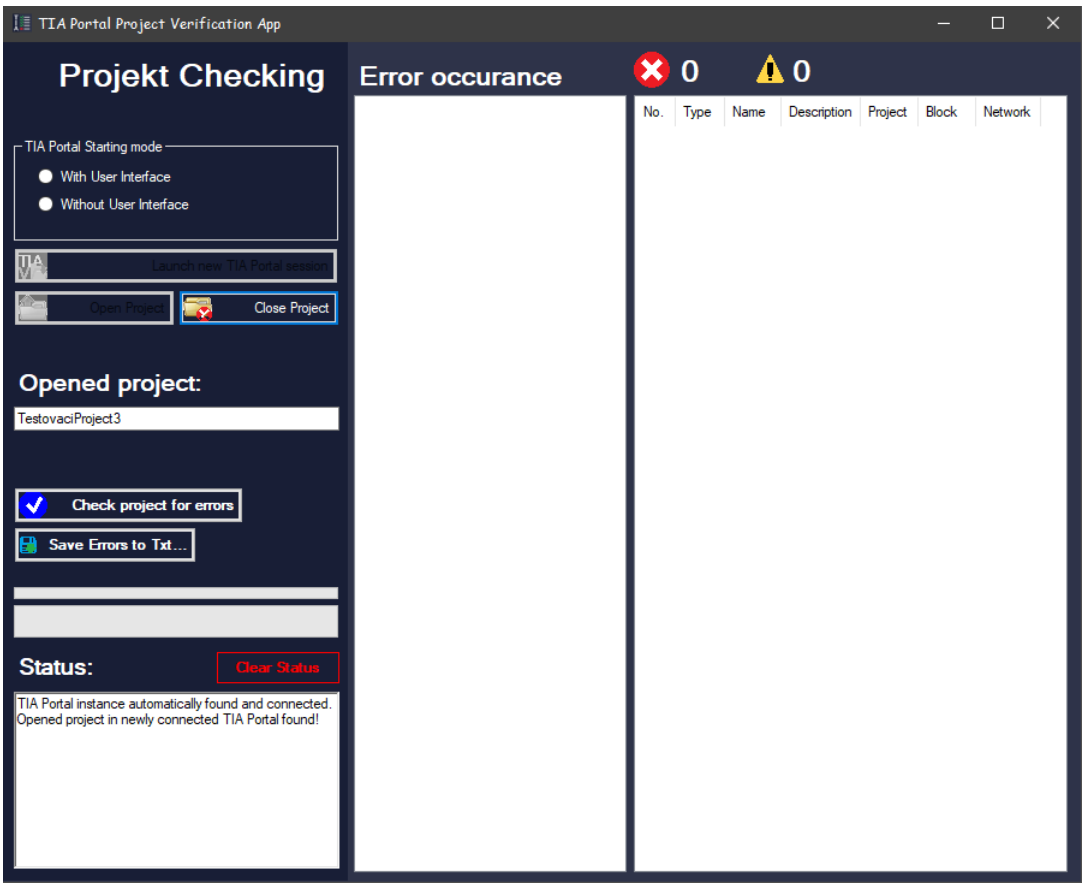


Obr. 2.3: Třída Network a její atributy - diagram



Obr. 2.4: Diagram tříd InterfaceOB, InterfaceFB a InterfaceFC a jejich atributů

2.3.7 Návrh vzhledu aplikace



Obr. 2.5: Základní vzhled aplikace

Vzhled aplikace je zobrazen na obr. 2.5. Aplikace má své logo a název *TIA Portal Project Verification App*.

Prvky aplikace

V této podkapitole budou shrnuty prvky, které je možné na obrázku vidět.

Sekce TIA Portal Starting mode umožní uživateli volbu zapnutí TIA Portalu z aplikace. Uživatel může TIA Portal spouštět s uživatelským rozhraním nebo bez něj (TIA Portal nebude vidět, ale spustí se na pozadí).

Tlačítko *Launch new TIA Portal session* umožní uživateli spustit TIA Portal přímo z aplikace.

Tlačítko *Open Project* umožní uživateli otevřít projekt v otevřeném TIA Portalu. Pokud není TIA Portal spuštěn, aplikace jej spustí až poté se objeví file dialog, aby mohl uživatel najít projekt v adresáři.

Tlačítko *Close Project* umožní uživateli zavřít otevřený projekt.

Tlačítko Check project for errors je tlačítko, pod kterým se nachází celý testovací proces aplikace. Zde dochází k exportu OB,FB,FC (a DB) na pozadí, následně jejich načítání do třídy `ProjectBlocks`.

Tlačítko *Save Errors to Txt...* umožní uživateli uložit chyby do textového souboru tak jak budou zobrazeny ve výčtu chyb. Otevře se file dialog a uživatel bude moci zvolit destinaci uložení textového souboru na svém počítači.

Ukazatele průběhu (anglicky progress bary) mají za úkol informovat uživatele o průběžném stavu kontroly PLC kódu - jak dlouho kontrola ještě bude přibližně trvat. Udávají pouze orientační postup v testování. Jedná se pouze o estetické prvky.

Status je komunikačním prostředkem mezi uživatelem a aplikací. Aplikace informuje uživatele skrze status, v jakém stavu je nebo co dělá, případně co se povedlo či nepovedlo, nebo co by měl uživatel udělat.

Tlačítko *Clear status* umožní vymazat status aplikace.

Výskyt chyb je zobrazen v *treeview* - ve stromovém zobrazení. V tomto zobrazení se načte projekt a v něm všechna zařízení a PLC bloky a to včetně jejich networků. Část programu, která obsahuje chybu, bude zvýrazněna červeně (error) či oranžově (warning).

Výčet chyb informuje uživatele o nalezených chybách. Chyby jsou setřízeny podle jejich názvu. Políčko *Type* udává informaci, zdali se jedná o warning či o error. Políčko *Name* představuje název chyby, políčko *Description* její popis. Dále je zde název projektu, název PLC bloku, ve kterém se nachází, v případě HW chyby je zde uvedeno, že se jedná o chybu v Devices&Networks (hardwarová konfigurace projektu). Nakonec je zde v případě chyby v PLC blocích uveden network, ve kterém se chyba nachází (pokud se v něm má nacházet).

2.3.8 Interakce aplikace s uživatelem

Po načtení projektu do aplikace (dále bude vysvětleno v kapitole 3.2, je možné celý projekt otestovat pomocí tlačítka *Check project for errors*. Doba testování může zabrat i několik desítek minut, pokud bude projekt obrovský, pokud však bude menšího charakteru, měl by být otestován v řádu sekund, maximálně minut.

Aplikace průběžně komunikuje s uživatelem skrze status, po nálezu všech chyb je klasifikuje na errory a warningy a zobrazí je. Aplikace umožňuje uživateli status smazat a označovat nalezené chyby.

Zároveň aplikace umožňuje uživateli vybrat místo, kam uloží textový soubor s výpisem chyb.

3 Realizace aplikace pro testování PLC kódu

Tato třetí a hlavní část pojednává o tom, jakým způsobem byla aplikace realizována. Jsou zde vysvětleny základní funkcionality aplikace a celý proces testování. Ten probíhá tak, že nejdříve proběhne základní inicializace (vymazání všech seznamů - listů chyb), kompilace kódu v TIA Portalu a vymazání složky s názvem *XMLHelpExportFolder*, která se nachází v kořenovém adresáři aplikace. Následně se exportují nové bloky pomocí TIA Openness a metody `Export` a z těchto bloků se načtou data do třídy `ProjectBlocks`. Poté se spustí samotné vyhledávací algoritmy chyb a po jejich dokončení jsou chyby vypsány na aplikaci.

3.1 Členění aplikace

Aplikace je členěna do několika jmenných prostorů. Ve jmenném prostoru *AppForTestingCodePLC* se nachází hlavní třída aplikace, třída `App\_Init\_Screen`, třída `Errors` a třída `ProjectBlocks`, zároveň obsahuje podprostor *AnotherHelpClasses*. V tomto podprostoru se nachází pouze podpůrné velmi malé třídy, určené pouze pro vyřešení určité části problému u některých testovacích algoritmů. Druhým důležitým jmenným prostorem je *PlayApp*, který obsahuje celou strukturu třídy `ProjectBlocks`. Obsahuje různé menší podprostory s různými třídami.

3.1.1 Atributy a seznamy hlavní třídy *App_Init_Screen*

Zde jsou vypsány veškeré používané atributy třídy *App_Init_Screen*.

Popis:

NumberOfOpennedTiaProjects udává počet otevřených TIA Portal projektů.

TiaProcess je pouze pomocný TiaProcess.

TIA_Portal je puštěný a do aplikace načtený TIA Portal session.

TIA_Project představuje TIA Portalový projekt, načtený do tohoto atributu.

GeneralPlcBlocksNames je seznam názvů PLC bloků přečtených pomocí Opennessu z projektu.

GeneralPlcBlocks je seznam PLC bloků přečtených pomocí Opennessu z projektu.

TIA_ProjectBlocks jsou bloky z TIA projektu, načtené z vygenerovaných XML předpisů do speciálně strukturované třídy `ProjectBlocks`.

XmlHelpPath je stringová cesta ke složce aplikace.

TIA_ProjectErrors představuje seznamy veškerých chyb, nalezených při testování projektu.

Výpis:

```
int NumberOfOpennedTiaProjects = 0;
private static TiaPortalProcess TiaProcess { get; set; }
public TiaPortal TIA_Portal{get; set;}
public Project TIA_Project {get; set;}
List<string> GeneralPlcBlocksNames { get; set; } = new List<string>();
List<PlcBlock> GeneralPlcBlocks { get; set; } = new List<PlcBlock>();
ProjectBlocks TIA_ProjectBlocks { get; set; } = new ProjectBlocks();
string XmlHelpPath { get; set; } =
    Directory.GetParent(Environment.CurrentDirectory).Parent.FullName;
Errors TIA_ProjectErrors { get; set; } = new Errors();
```

3.1.2 Metody hlavní třídy *App_Init_Screen*

Zde jsou vypsány některé metody třídy *App_Init_Screen*. Nejsou zde vypisovány metody stisknutí tlačítek či jiné událostní metody, pouze mnou vytvořené metody.

Metoda *PVA_Init_Screen_Load* určuje, co se má vykonat hned po spuštění aplikace - kontroluje, jestli už je projekt otevřený, či TIA Portal spuštěný.

```
private void PVA_Init_Screen_Load(object sender, EventArgs e)
```

Metody *TreeViewShow*, *ColorFindInterface* a *ColorFindNetwork* zobrazí stromové rozřazení OB, FB a FC v projektu a barevně vyznačí přesně na jakých networcích se nachází errory a warningy. *ColorFindInterface* a *ColorFindNetwork* jsou pomocné metody.

```
private void TreeViewShow(Project aTIA_Project){...}
private Color ColorFindInterface(dynamic Block){...}
private Color ColorFindNetwork(dynamic Block, int NetworkNr){...}
```

Metoda *ErrorChecking* postupně volá jednotlivé algoritmy pro nalezení daných chyb.

```
private void ErrorChecking(){...}
```

Metoda *CompileDevices* zkompiluje všechna PLC zařízení v projektu tedy i jejich bloky - kvůli konzistenci kódu.

```
private void CompileDevices(){...}
```

Metody *ShowErrors*, *ErrTypeColor*, *BlockColor* a *ErrorListView_ItemCheck* jsou metody, které mají za úkol vypsat všechny nalezené chyby ve výčtu chyb (`ListView` *ErrorListView*), zvýraznit červeně errory, zvýraznit oranžově warningy, zvýraznit fialově OB, modře FB a zeleně FC. Umožnit uživateli zaškrtnout chybu a zvýraznit ji tak bledě modře pro lepší orientaci.

```
private void ShowErrors(){...}
private Color ErrTypeColor(dynamic aError){...}
private Color BlockColor(dynamic aError){...}
private void ErrorListView_ItemCheck(object sender,
    System.Windows.Forms.ItemCheckedEventArgs e)
```

Metoda *DeleteAllXMLBlockFiles* má za úkol rekurzivně vymazat všechny stávající XML soubory ze složky *XmlHelpExportFolder*.

```
private void DeleteAllXMLBlockFiles(){...}
```

Metody *ExportAllBlocks* a *RecursiveExport* vyexportují veškeré PLC bloky z PLC projektu do složky *XmlHelpExportFolder*.

```
private void ExportAllBlocks(){...}
private void RecursiveExport(PlcBlockUserGroup aPlcBlockUserGroup){...}
```

Metody *UnpackAllBlocksFromXMLExport*, *DetectBlockFromXML*, *FindPlcBlock*, *OB_Unpacking*, *FB_Unpacking*, *FC_Unpacking* a *StructingRecursive* tvoří souhrn složitějších metod, jejichž hlavním úkolem je načíst základní data ze všech PLC bloků, načtených pomocí Openness a načíst všechny potřebné informace, spojené s interfacem a networky PLC bloku. Jedná se o importovací algoritmy.

```
private void UnpackAllBlocksFromXMLExport(){...}
private blockType DetectBlockFromXML(XmlNode BlockType){...}
private PlcBlock FindPlcBlock(string aBlockName){...}
private OrganisationBlock OB_Unpacking(string filePath){...}
private FunctionBlock FB_Unpacking(string filePath){...}
private Function FC_Unpacking(string filePath){...}
```

```
private InterfaceMember StructingRecursive(XmlNode
    aMember,InterfaceMember aInterfaceMember){...}
```

Metoda *SaveTxt_Click* je metoda na stisknutí tlačítka, která bude vysvětlena v kapitole 3.3.7.

```
private void SaveTxt_Click(object sender, EventArgs e){...}
```

Poslední metoda je *DelStat_Click_1*, jejíž účel je vymazat status, kterým aplikace dává vědět uživateli o svém stavu.

3.1.3 Vytvořené enumy

Nacházejí se v podprostoru *TIA_Enums* jmenného prostoru *PlayApp*. Vytvořené enumy usnadňují tvorbu aplikace a zpřehledňují ji. Výpis:

```
enum PLCProgrammingLanguage{ LAD,SCL,FBD,STL,GRAPH,PRODIAG }
enum AccessiBility{ Public,Private,Protected }
enum InterfaceMemberType{ Input, Output, InOut, Temp, Static, Constant,
   Return }
enum CallBlockType{ FB,FC }
enum CallBlockSection{ Input,Output,InOut }
enum ConnTEMPTypeEnum{ None,Read,Write }
enum blockType{ OB, FB, FC, DB, non_defined }
enum AccessType{ LocalConstant, LiteralConstant, LocalVariable,
   GlobalVariable, TypedConstant, GlobalConstant }
enum PartType
{
    //bez instanci
    Contact,ContactNegated, PContact, NContact, Sr, Rs,
    Coil,CoilNegated,SCoil, RCoil, PCoil, SBitfield, RBitfield,
    PBox, NBox, NCoil,CoilTP,CoilTON,CoilTOF,CoilTONR,
       ResetIECTimerCoil,PtCoil,
    Eq,Ne,Ge,Le,Gt,Lt,InRange,OutRange,OK,NOK,
    Move, Serialize, Deserialize, MoveBlockI, MoveBlockII,
    Convert,Scale_X,Normalize,
    And,Or,Xor,Invert,
    Shr,Shl,Ror,Rol,
    //s instancemi
    R_TRIG, F_TRIG,TP,TON,TOF,TONR,CTU,CTD,CTUD,
    //jine bloky - v pripade potreby doplnime
```

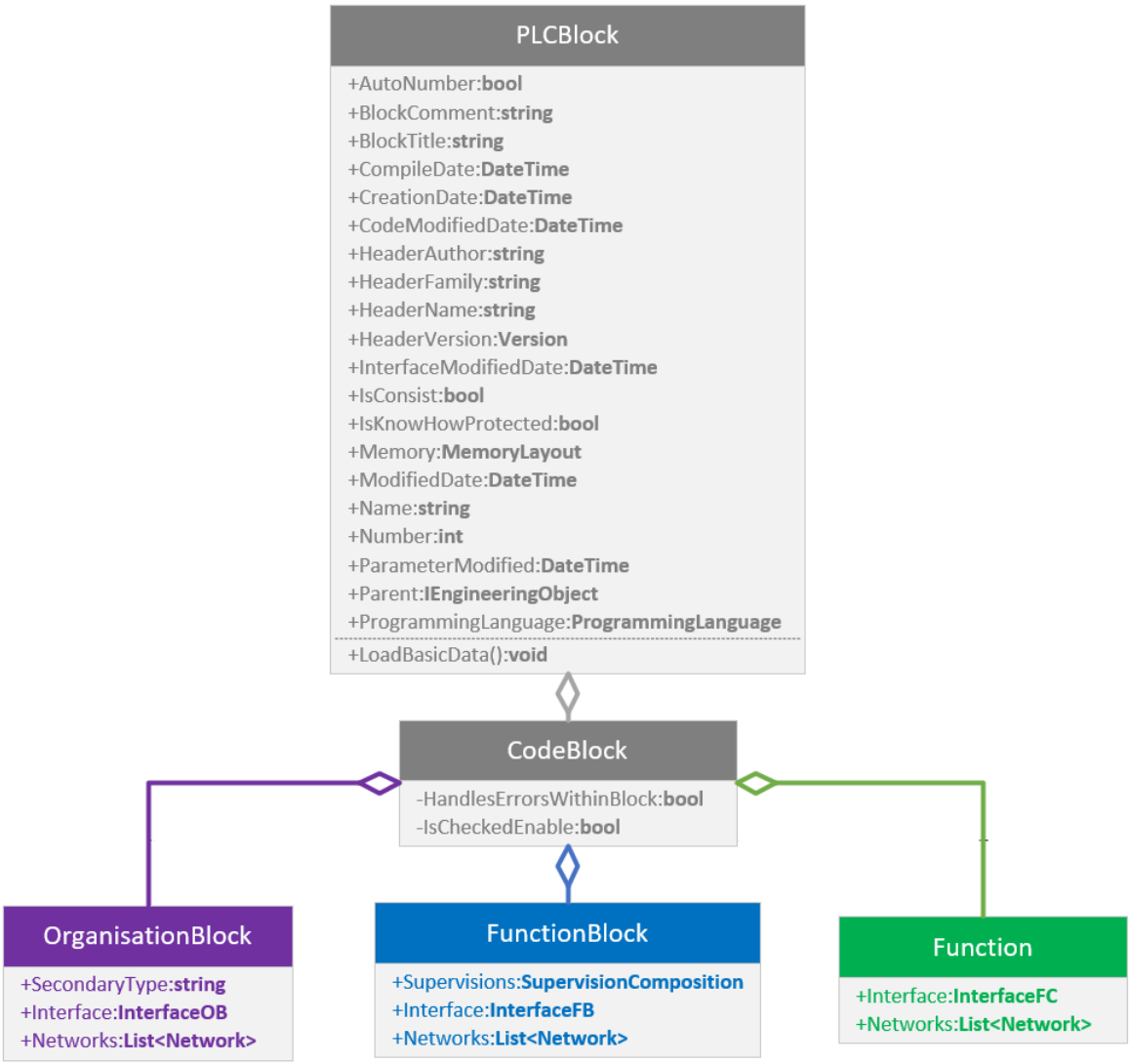
```
    OTHERS
}
enum PartCall{ Part,Call }
enum ErrorTypeEnum{ Error,Warning }
```

3.1.4 Třída *ProjectBlocks*

Tato třída se nachází v podprostoru *BlockClasses* jmenného prostoru *AppForTestingCodePLC* a jedná se o třídu, která združuje všechny PLC bloky z projektu. Tato třída uchovává veškeré potřebné informace o PLC blocích, ať už základní informace získané pomocí Openness u či informace o interface a networcích, získané pomocí importovacích algoritmů. Obsahuje 3 seznamy:

```
public List<OrganisationBlock> OrganisationBlocks { get; set; } = new
    List<OrganisationBlock>();
public List<FunctionBlock> FunctionBlocks { get; set; } = new
    List<FunctionBlock>();
public List<Function> Functions { get; set; } = new List<Function>();
```

Členění OB, FB a FC se nachází na obrázku 3.1. Třídy `InterfaceOB`, `InterfaceFB`, `InterfaceFC`, `Network` jsou stejné jako na obrázcích 2.3 a 2.4, popsané v kapitole *Návrh aplikace pro testování PLC kódu*.

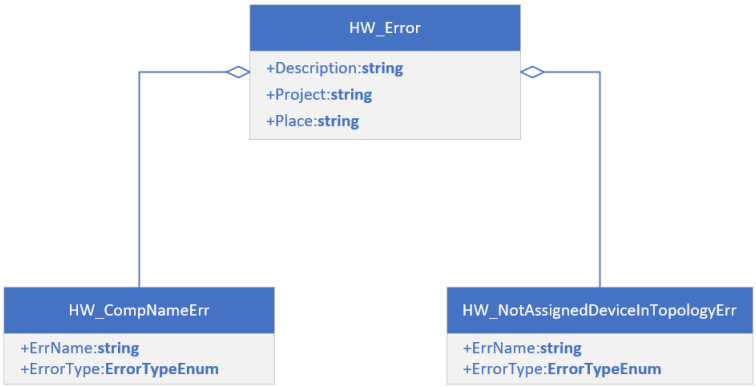


Obr. 3.1: Diagram tříd pro OB, FB a FC

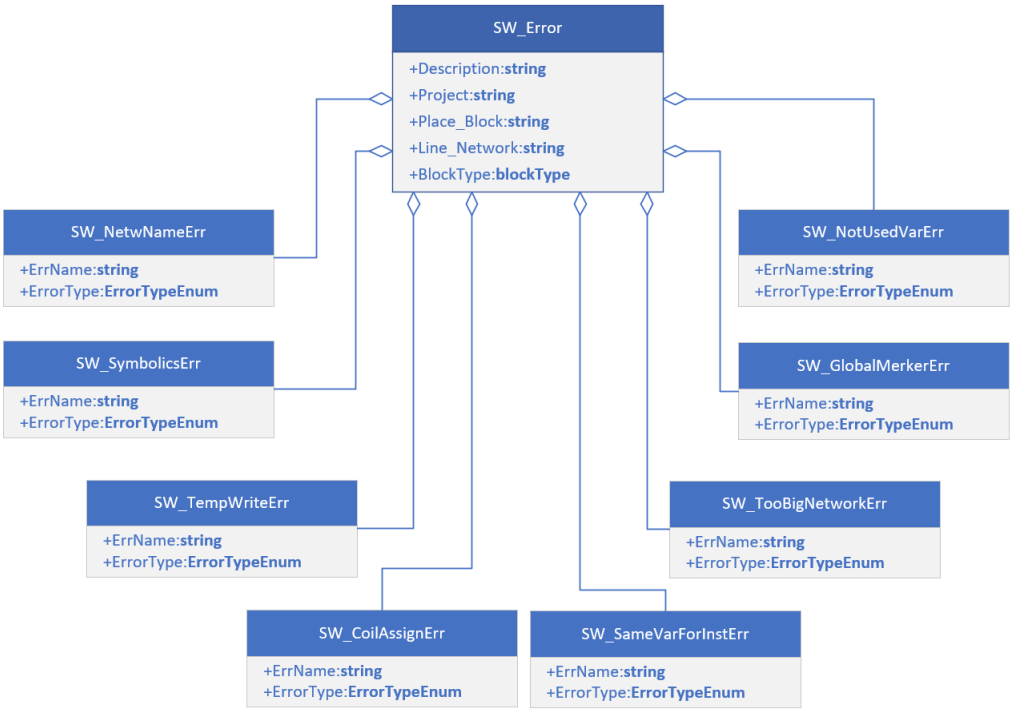
3.1.5 Třída *Errors*

Třída Errors je velice důležitá třída v aplikaci. Sdružuje totiž nejen nalezené chyby, ale i veškeré vyhledávací algoritmy chyb. Nachází se v souboru *Errors.cs*. V souboru *ErrorClasses.cs* jsou pak obsaženy třídy všech chyb, které testuji. Atributy a metody třídy `Errors` se nachází na obrázku 3.2. Třídy chyb v hardwarové konfiguraci se nachází na obrázku 3.3 a třídy chyb v PLC blocích (SW chyb) na obrázku 3.4. Třída `Errors` je v projektu zastoupena instancí *TIA_ProjectErrors*.

Obr. 3.2: Diagram třídy *Errors*



Obr. 3.3: Diagram tříd pro chyby v hardwarové konfiguraci



Obr. 3.4: Diagram tříd pro chyby v PLC blocích

3.2 Základní funkcionality aplikace

Základní funkcionality zahrnují schopnost aplikace zapnout TIA Portal (který nebyl doposud zapnut), otevřít projekt v TIA Portalu, zavřít projekt v TIA Portalu, či se přímo po zapnutí k otevřenému projektu připojit.

Na obrázku 3.5 je vidět vývojový diagram, ve kterém se nachází základní funkcionality aplikace - tlačítka *Open Project* (otevřít projekt), *Close projekt* (zavřít projekt), *Launch new TIA Portal session* (otevřít TIA Portal) a *Check project for errors* (spustit hledání chyb).

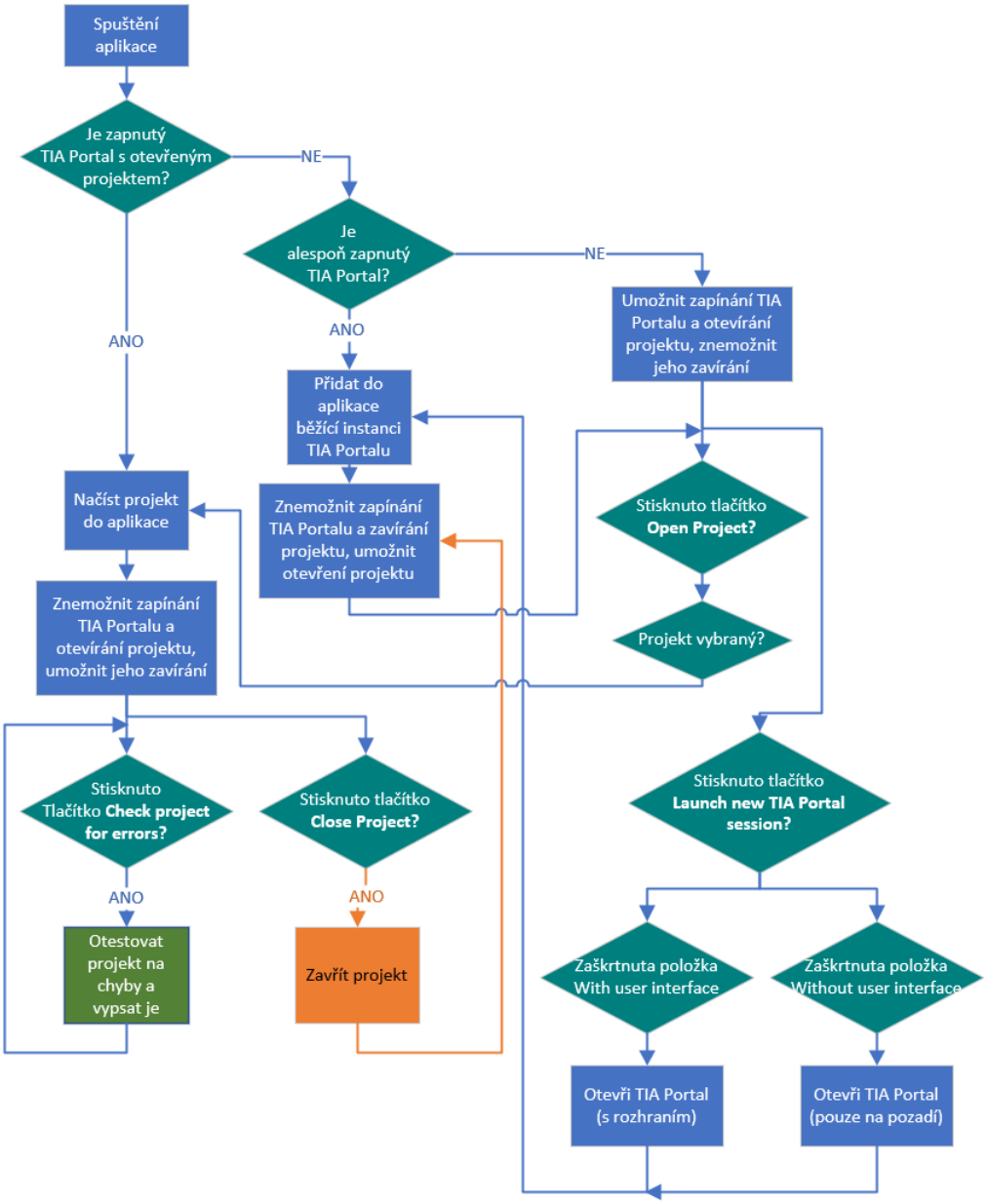
Pokud uživatel aplikaci zapne a nemá spuštěný TIA Portal, je mu umožněno jej spustit přímo z aplikace, nebo přímo otevřít projekt. Po spuštění TIA Portalu z aplikace se zároveň načte instance projektu do aplikace - konkrétně do instance *TIA_Portal* třídy `TiaPortal` (třída z Openness knihovny). Pokud uživatel rovnou otevře projekt, nejenže se načte instance třídy `TiaPortal`, ale načte se i instance *TIA_Project* třídy `Project` (opět z Openness knihovny).

Pokud uživatel spustil aplikaci a již měl na pozadí spuštěný TIA Portal, aplikaci si automaticky načte spuštěný session TIA Portalu opět do instance *TIA_Portal* třídy `TiaPortal`.

Pokud uživatel spustil aplikaci a již měl otevřený projekt, aplikace si automaticky načte spuštěný session TIA Portalu a zároveň si načte projekt.

Po otestování projektu je možné uložit aplikaci do textového dokumentu. V textovém dokumentu budou jednotlivé chyby zapsány formátu: "Číslo | Typ chyby | Název chyby | Popis chyby | Projekt | Blok | Network |".

Přímý název aplikace je *TIA Portal Project Verification App*. Hlavní třída celé aplikace nese název *App_Init_Screen*, dědí ze třídy `Form`, která představuje okno aplikace a nachází se v prostoru *AppForTestingCodePLC*.



Obr. 3.5: Vývojový diagram základních funkcionalit aplikace

3.3 Proces testování PLC projektu

Ještě před samotným testováním se vymažou seznamy OB, FB a FC z instance třídy *TIA_ProjectBlocks*, seznamy *GeneralPlcBlocksNames*, *GeneralPlcBlocks* a veškeré chybové seznamy z instance *TIA_ProjectErrors*.

3.3.1 Kompilace kódu pomocí TIA Openness

Provádí se pomocí metody *CompileDevices*. Po vymazání seznamů je zapotřebí přes Openness zkompilovat PLC bloky, které budou následně exportované. Pokud nebudou konzistentní (parametr *IsConsistent*), Openness je nedovolí vyexportovat a aplikace by vyvolala výjimku. Pomocí cyklu `foreach` prochází aplikace všechny zařízení třídy `Device` z projektu, z každého zařízení se pak opět pomocí vnořeného cyklu `foreach` prochází všechny položky třídy `DeviceItem`, kde se z každé položky pomocí příkazu

```
SoftwareContainer softwareContainer =
    deviceItem.GetService<SoftwareContainer>();
```

načte instance třídy `SoftwareContainer`, která se zkontroluje, jestli nepředstavuje nulový odkaz (dále jen `null`) a jestli je její položka `Software` třídy `PlcSoftware`. Pokud ano, opět se zkontroluje jestli není `null` a v případě že ne, dojde ke kompilaci kódu pro dané PLC zařízení.

3.3.2 Vymazání adresáře *XMLHelpExportFolder*

Provádí se pomocí metody *DeleteAllXMLBlockFiles*. Následující kód vymaže veškerý obsah adresáře a všechny případné subadresáře tohoto adresáře - druhý `foreach`. Ten by v kódu již nemusel být, protože aplikace žádné subadresáře ve adresáři *XMLHelpExportFolder* již negeneruje, ale pro správnost mazání složky jsem jej ponechal.

```
string slePath = XmlHelpPath + "\\XmlHelpExportFolder";
System.IO.DirectoryInfo directoryToClear = new DirectoryInfo(slePath);
foreach (FileInfo fileInfo in directoryToClear.GetFiles())
{
   file.Delete();
}
foreach (DirectoryInfo directoryInfo in directoryToClear.GetDirectories())
{
   dir.Delete(true);
}
```

Kód byl převzat a implementován do aplikace ze stránky [19].

3.3.3 Rekurzivní export XML předpisů PLC bloků do adresáře *XMLHelpExportFolder*

Provádí se pomocí metod *ExportAllBlocks* a *RecursiveExport*. V první metodě se procházejí podobně jako u kapitoly 3.3.1 všechny položky `Software` třídy `PlcSoftware`. Dále se však dalším vnořeným `foreachem` prochází všechny PLC bloky pomocí:

```
foreach (PlcBlock plcBlock in PLCSoft.BlockGroup.Blocks){...}
```

kde PLCSoft je instancí aktuálně procházené třídy `PlcSoftware`. Pomocí `ifu` se zkontroluje konzistence kódu a to, jestli náhodou blok není uzamčený (parametr *IsKnowHowProtected*) - takové totiž nelze exportovat, export by opět skončil vyvoláním příslušné výjimky. Také je třeba si dát pozor na export bloku s názvem, obsahující znak "/", který je třeba nahradit jiným znakem (v mém případě "_"). Před exportem si jméno a blok uložím do seznamů *GeneralPlcBlocksNames* a *GeneralPlcBlocks*, již zmíněných v kapitole 3.1.1 a provedu export PLC bloku příkazem:

```
plcBlock.Export(new FileInfo(string.Format(XmlHelpPath +
    "\\XmlHelpExportFolder\\{0}.xml", plcBlock.Name)),
    ExportOptions.None);
```

Po exportu všech bloků z projektu je však ještě třeba exportovat PLC bloky, obsažené v uživatelských složkách - `UserGroups`. Na každou tuto složku je zavolána druhá metoda *RecursiveExport*, ve které se úplně stejným způsobem vyexportují všechny bloky, které složka obsahuje a pokud se v ní nachází další podsložky, opět se z této metody rekurzivně metoda sama zavolá. Parametrem této metody je třída `PlcBlockUserGroup`.

Pozn.: Adresář *XMLHelpExportFolder* je adresář, který patří pod aplikaci, uživatel jej nemusí nikde vytvářet.

3.3.4 Načtení vyexportovaných bloků do třídy *ProjectBlocks*

Po vyexportování XML předpisů následuje jejich načtení do instance *TIA_ ProjectBlocks* třídy `ProjectBlocks`. Hlavní importovací metodou je *UnpackAllBlocksFromXMLExport*. V ní se nachází pouze jeden `foreach`, který prochází všechny názvy souborů z adresáře *XMLHelpExportFolder* a pomocí metody *DetectBlockFromXML* zjišťuje, zdali se jedná o OB, FB či FC. Metodě je předán příslušný a správně načtený parametr `XmlNode` *BlockType* a na základě názvu potomku tohoto XML uzlu (XML

Node) vrátí hodnotu výroku `enum blockType`. Zpět ve funkci *UnpackAllBlocksFromXMLExport* se pak zavolá příslušná funkce *XX_Unpacking*, kde XX v názvu může být OB, FB či FC. Metody vrací vytvořenou instanci tříd `OrganisationBlock`, `FunctionBlock` či `Function` a tyto instance jsou vkládány přímo do příslušných seznamů instance *TIA_ProjectBlocks* třídy `ProjectBlocks`, popsané v kapitolách 2.3.6 a 3.1.4. Všechny tři metody typu *XX_Unpacking* fungují velmi podobně, rozdíl plyne pouze z jejich podstaty - OB má pouze proměnné bloku *Input*,*Temp* a *Constant*, FB má *Input*, *Output*, *InOut*, *Static*, *Temp* a *Constant* a FC má *Input*, *Output*, *InOut*, *Temp*, *Constant* a *Return*.

Vnitřní fungování metod *OB_Unpacking*, *FB_Unpacking* a *FC_Unpacking*

Všechny tři metody mají parametr `string` *filePath*, představující název cesty ke XML souboru daného bloku. Ten je načten do instance *MyXMLDoc* třídy `XmlDocument` pomocí metody `Load`.

Po načtení XML souboru se nejdříve načtou základní data z instancí třídy `PlcBlock` (z Openness), které jsem v kapitole 3.3.3 ukládal do seznamů *GeneralPlcBlocksNames* a *GeneralPlcBlocks* a to pomocí metod *FindPlcBlock* a *LoadBasicData*. Metoda *FindPlcBlock* pomocí cyklu `foreach` najde v seznamu *GeneralPlcBlocks* příslušný blok podle jména na základě nalezeného názvu v XML předpisu a vrátí jej jako instanci a to přímo do argumentu metody *LoadBasicData*. Tato metoda už ale nepatří hlavní třídě `App\_Init\_Screen`, nýbrž třídě `PLCBlock`, konkrétně instancím *organisationBlock* třídy `OrganisationBlock`, *functionBlock* třídy `FunctionBlock` či *function* třídy `Function` a jedná se o jednoduchou kopii dat z příslušného PLC bloku. Základní data se nacházejí v mateřské třídě `PLCBlock` (stejně jako metoda *LoadBasicData*) a představují všechna data původní Openness třídy `PlcBlock`.

Po načtení základních dat následuje načtení všech proměnných interface bloku. Nejdříve se najde XML uzel s názvem *Interface* ("/Document/SW.Blocks.FB/AttributeList/Interface") a v něm uzel *Sections* ("SW.Blocks.FB" značí konkrétně FB, může tam být například "SW.Blocks.OB" nebo "SW.Blocks.FC"). V uzlu *Sections* se nacházejí jednotlivé elementy *Section*, představující typ proměnné (atribut *Name*). Každý tento element obsahuje subelementy *Member*, které představují jednotlivé proměnné a jejich údaje - názvy, hodnoty, komentáře a tyto údaje jsou načítány do nově vytvářených instancí třídy `Input`, `Output`, `InOut`, atd. (pro všechny typy proměnných funguje algoritmus naprosto shodně). Pokud je detekována struktura, volá se metoda s názvem *StructingRecursive*. Jejími parametry jsou XML subelement typu *Member* a třída `InterfaceMember`. Algoritmus strukturu projde a pokud v ní nalezne další strukturu, opět se rekurzivně zavolá. Aplikace umí i detekovat a

rozlišit dva druhy datových typů *Array*, tedy pole. První typ je ve tvaru "Array[A..Z] of NázevDatovéhoTypu", kde A je počáteční a Z koncová hodnota. Druhý typ je ve tvaru "Array[*] of NázevDatovéhoTypu". Veškeré prvky, ať už struktur nebo polí, aplikace pro daný typ proměnné seřadí do seznamů a uloží všechny tyto seznamy jednotlivých typů proměnných (Input, Output,...) do seznamů ve třídách `InterfaceOB`, `InterfaceFB` a `InterfaceFC` (popsaných v kapitole 2.3.6 na obrázcích obrázcích 2.3 a 2.4.

Jakmile dojde k načtení všech proměnných interface bloku, je na řadě načtení networků a veškerých jejich potřebných dat. Nejdříve se musí najít v XML předpisu bloku element s názvem *ObjectList* (Cesta: "/Document/SW.Blocks.FB/ObjectList"), obsahující subelementy s názvem *SW.Blocks.CompileUnit* a přesně tento subelement představuje network. V jednom cyklu `foreach` se pro každý tento subelement (tedy pro každý network) vytvoří instance *network* třídy `Network`, načtou se z XML předpisu jeho nejpodstatnější informace - název, komentář, programovací jazyk. Poté se pomocí druhého vnořeného cyklu `foreach` prochází jednotlivé Party (subelement *Parts*) a klasifikují se na `Accessy`, `Party` a `Cally`. `Accessy` jsou načteny do seznamu *Accesses* v instanci *network*. `Accessy` se dále klasifikují na lokální constanty, literály, lokální proměnné, globální proměnné, typové konstanty a globální konstanty. Algoritmus umí opět rozpoznat pole a struktury a zapsat proměnné v nich tak, aby byly dále porovnávatelné při vyhledávání chyb. U `Partů` se načte jejich UId - číselné označení a Název - *part.Name*, který je datového typu `enum PartType`. String z XML předpisu je zde převeden na tento enum. Název může být *Contact*,*ContactNegated*,*Coil*,*Move*,*Eq*, atd. Také se načte informace o tom, jestli má `Part` instanci a pokud ano, tak se načte její UId a název. U `Callu`, tedy volaného bloku se načte jeho typ (FB/FC), v případě FB jeho instanci a všechny vstupně-výstupní parametery s jejich veškerými daty. Po načtení Partů se načtou `Wiry`, které nesou informaci o jejich UId, Nameconny a Identconny s jejich názvy a UId, představující napojené `Accessy`, `Party` a `Cally`.

Po načtení všech `Accessů`, `Partů`, `Callů` a `Wirů` (s jejich daty) do networku se network uloží také do seznamu a po tomto načtení všech networků je OB/FB/FC připraveno a vráceno pomocí příkazu `return` zpět do funkce *UnpackAllBlocksFromXMLExport*, konkrétně bude přidaný nový načtený blok do seznamů OB/FB/FC instance *TIA_ProjectBlocks* třídy `ProjectBlocks`.

3.3.5 Testovací algoritmy

PLC bloky jsou nyní uloženy v instanci *TIA_ProjectBlocks* třídy `ProjectBlocks`, což znamená, že jsou veškerá potřebná data načtena do aplikace a tudíž přicházejí na řadu jednotlivé testovací algoritmy. Nalézané chyby se ukládají do jednotlivých

seznamů v instanci *TIA_ProjectErrors* třídy `Errors`. V následujících kapitolách budou testovací algoritmy pouze stručně popsány.

Algoritmus hledání nepřejmenovaných zařízení

Algoritmus funguje tak, že pomocí cyklů `foreach` projde všechny zařízení `Devices` v projektu a v nich všechny `DeviceItemy`. Každý `DeviceItem` pak otestuje, zda-li jeho název není ve tvaru "Nazev_X", kde X je jednociferné číslo. Nevýhoda tedy je, že aplikace nedokáže rozpoznat více než 9 stejných zařízení se stejnou chybou. Tento nedostatek je v plánu odstranit. Algoritmus je však schopen otestovat celou řadu zařízení. Pokud je chyba nalezena, je přidána do seznamu *hW_CompNameErrs*.

Algoritmus hledání nepřiřazených zařízení v rámci PN topologie

Tento algoritmus už je složitější, protože musí procházet nejen všechny instance třídy `Device`, ale i instance tříd `DeviceUserGroup` a `Device` ze skupiny *UngroupedDevicesGroup* - tedy veškerá zařízení obsažena v projektu. U všech zařízení se musí cykly `foreach` projít až na patřičnou úroveň - kde se vyskytují názvy subzařízení jako "PROFINET", "interface", "Interface" či "PN-IO" a u nich ještě zkontrolovat veškeré porty. Ty se musí nejdříve získat příkazem:

```
NetworkPort port =
    ((IEngineeringServiceProvider)PortDeviceItem).GetService<NetworkPort>();
```

a u každé instance *port* zkontrolovat, zda se počet připojených zařízení nerovná nule. Pokud se nule rovná, k portu není připojeno žádné zařízení a tudíž algoritmus nalezl chybu a zapsal ji do seznamu *hW_NotAssignedDeviceInTopologyErrs*.

Algoritmus hledání nepojmenovaných networků

Jednoduchý algoritmus, procházející všechny PLC bloky (OB, FB a FC) předané z instance *TIA_ProjectBlocks*. V každém bloku projde všechny networky (opět samozřejmě cyklus `foreach`) a pokud délka stringu jejich názvu je nulová, jedná se o nepojmenovaný network a chyba je zapsána do seznamu *sW_NetwNameErrs*.

Algoritmus hledání chybně použité symboliky

U tohoto hledání stačí procházet pouze funkční bloky, protože firma používá bloky s názvy "StationXXX" (kde XXX je trojciferné číslo) pouze jako FB. Pomocí vnořených `foreach` se projde každý `Access` v každém networku. Algoritmus využívá pomocnou funkci *BlockNameNumber*, který zjistí číslo XXX stanice a také funkci *FindNumber*, která se používá u každého procházeného `Accessu` a ta má za úkol z

jeho názvu vyčíst číslo. Pokud se jedná o vstupně-výstupní (I/O) tag, proměnnou začínající názvem "STATUS" či proměnnou začínající názvem "HMI", funkce vrátí celočíselnou (`int`) hodnotu - číslo, signalizující stanici XXX, ke které proměnná patří. Pokud je toto číslo jiné než to, které bylo získáno pomocí funkce *BlockNameNumber*, jedná se o chybu a ta se zapíše do seznamu *sW_SymbolicsErrs*.

Algoritmus hledání chyb typu čtení neinicializovaných TEMP proměnných

Prochází všechny OB, FB a FC (pro všechny PLC bloky je volána metoda *BadTEMP_FindingAlgorithm*) a v každém bloku prochází všechny proměnné typu TEMP. Pokud se jedná o strukturu, zavolá si pomocnou rekurzivní funkci *RecursiveReadingInterfaceVars*, která funguje tak, že si vytvoří pomocný list a do ní dává její názvy (`string`) jednotlivých proměnných. Tento list poté vrací jako výpis všech proměnných ve struktuře. Pokud se opět jedná o strukturu, rekurzivně se volá znovu. Konečná návratová hodnota je seznam obsahující všechny prvky struktury. Takto se projde každá instance třídy `Temp` a zapíše se do pomocného seznamu *tempName*.

Po naplnění listu *tempName* se `foreachem` projde každý `Access` ve všech networcích bloku a pomocí pomocné funkce *IsTemp* se zjistí, zda-li se jedná o Temp proměnnou z tohoto bloku a pokud ano, přidá se `Accessu` informace *ConnTEMPType* - informace o tom, zdali je `Access` čten, či je do něho zapisováno. Tato informace je typu `enum ConnTEMPTypeEnum`.

Nakonec se takto informačně upravené `Accessy` znovu projdou a pokud daný `Access` nebyl nalezen podruhé a zároveň je jeho parametr *ConnTEMPType* klasifikován jako "Read", je prohlášen za chybu. Tato chyba je pak připsána do seznamu *sW_TempWriteErrs*

Algoritmus hledání přiřazení do stejné proměnné na více cívkách

Chyba je hledána opět ve všech PLC blocích (pro všechny PLC bloky je volána metoda *MultipleCoilAssignmentFindingAlgorithm*). V ní se nejdříve založí list *FoundedCoils* pomocné třídy typu `CoilInBlock`, definované v podprostoru *AnotherHelpClasses*. Cykly `foreach` se projde každý `Part` typu *Coil* nebo *CoilNegated* (cívka, která není typu S/R). Pokud je takový `Part` nalezen, projde se každý `Wire` a pokud se UId jeho parametru *Nameconn* shoduje s UId nalezeného `Partu` a zároveň má druhý parametr *Identconn* (připojení k `Accessu`) s nenulovým UId, podívá se na toto UId Identconnu a podle něho nalezne `Access` (jehož UId je stejné). Takto nalezené cívce se poté přiřadí číslo networku, ve kterém se nachází, UId `Partu` a název připojeného `Accessu` (cívka bude novou instancí třídy `CoilInBlock`) a ta se zařadí do seznamu *FoundedCoils*.

Poté se projde tento seznam a každý prvek se porovná s každým (kromě sebe samotného) a pokud má připojený `Access` stejné jméno a stejná chyba není v seznamu *sW_CoilAssignErrs* ještě evidována, zapíše se do tohoto seznamu.

Algoritmus hledání nepoužitých proměnných

Podobně jako u kapitoly 3.3.5 jsou všechny proměnné bloku (nejenom Temp) uloženy do seznamu, který nese název *AllVarsInInterface* a jedná se o list stringů (v případě struktur se opět volá rekurzivní funkce *RecursiveReadingInterfaceVars*).

Pomocí cyklu `foreach` se tento list prochází po jednotlivých `string` prvcích a pro každý network se prochází každý `Access`, každá instance `Partů` a každá instance `Callů`. Pokud se jejich názvy shodují s aktuálním procházeným prvkem seznamu *AllVarsInInterface*, je ta proměnná považována za používanou a tudíž se nejedná o chybu a přechází se na další proměnnou (jinými slovy: pokud se vyskytuje kdekoliv v bloku, je používána). Pokud nikde však nalezená proměnná nebude, booleovská proměná *NotUsed* zůstane true a chyba se zapíše do seznamu *sW_NotUsedVarErrs*.

Algoritmus je použit na všechny PLC bloky (OB,FB i FC).

Algoritmus hledání stejných proměnných pro více instancí instrukcí typu čítače, časovače, R_TRIG, F_TRIG

Opět se zde prochází všechny OB, FB a FC (pro všechny PLC bloky je tentokrát volána metoda *MultipleInstancesFindingAlgorithm*). V metodě je založen seznam *FoundedInstances* pomocné třídy typu `MultipleInstancesErrClass`. Do ní se načte vše, co má nějakou instanci. To znamená veškeré volané `Cally` typu FB a `Party` typu *R_TRIG*, *F_TRIG*, všechny druhy čítačů a časovačů. V seznamu instancí se pak pomocí dvou vnořených `for` cyklů vyhledají duplicity - chyby a ty se zapíší do seznamu *sW_SameVarForInstsErrs*.

Algoritmus hledání příliš dlouhých networků

Stejně jako v předchozích algoritmech se i zde prochází všechny OB,FB a FC, přičemž na všechny PLC bloky se volá metoda *TooManyInstErrFindingAlgorithm*, ve které je cyklus `foreach` procházející všechny networky bloku. V tomto cyklu se inicializují 2 proměnné - *horLength* a *vertLength*, které představují počítanou horizontální a vertikální délku networku. Pokud není network prázdný, tak za každé rozbočení se rekurzivně počítá jeho velikost (vertikální). V okamžiku kdy větev již další rozbočení nemá, vrací se velikost takové součástky, která na ní byla největší. V každém rozbočení se pak velikost větví sčítají až k původnímu rozvětvení. Celkový součet je vertikální délkou networku. Podobným způsobem se počítá i horizontální délka networku. Ta je však na výpočet jednodušší, protože se počítají pouze `Party`

a `Cally`, které jdou za sebou. U vertikální délky networku je třeba každému `Callu` přiřadit číslo na základě toho, kolik má vstupů a výstupů. Protože jsou vstupy a výstupy na stejné úrovni a vertikální velikost součástky zvětšují pouze připojení (vstupy/výstupy), kterých je více, je třeba počítat právě jen s těmi připojeními, kterých je na jedné straně volaného `Callu` více. Za každé připojení u `Callu` (ve třídě `ProjectBlocks` nazvané jako "parameter") se vertikální velikosti `Callu` přičítá +1 v metodě *CallVertLength*, která slouží k výpočtu velikosti `Callu`. Za každý `Part` se přičítá +2, stejně tak za samotný `Call` (kdyby byl bez vstupů a výstupů).

V tomto algoritmu jsou zde použity malé metody *FindWire*, *FindPart* a *FindCall*, pomocí kterých si algoritmus hledá následující "součástku" (`Part`/`Call`), umístěnou na jednom "drátě"(`Wire`).

Jakmile jsou finální čísla *horLength* a *vertLength* vypočítána, bylo třeba experimentálně stanovit meze, kdy bude network prohlášen za horizontálně či vertikálně příliš velký. Pokud celková vertikální velikost přesáhne 12, je network prohlášen za vertikálně příliš dlouhý. Pokud celková horizontální velikost přesáhne 6, je network prohlášen za horizontálně příliš dlouhý. Pokud celková vertikální velikost přesáhne 12 a zároveň celková horizontální velikost přesáhne 6, je network prohlášen za vertikálně i horizontálně příliš dlouhý. Tato čísla jsou zjištěna tak, aby se network s průměrně velkými názvy `Accesů` vešel na obrazovku s rozlišením 1920 na 1080 pixelů (FullHD) u průměrného programátorského notebooku. Vyhodnocené chyby se zapisují do seznamu *sW_TooBigNetworkErrs*.

Poz.: Tento výpočet by se ještě mohl zpřesnit, kdyby se do něho zahrnuly právě i délky názvů `Accesů`. Vzhledem k tomu, že se však jedná pouze o chybu estetického charakteru (tedy o warning), může však toto méně přesné řešení postačovat.

Algoritmus hledání použití M merkerů typu STDComm

Poslední algoritmus testuje, zdali se v projektu nevyskytuje proměnná, která by obsahovala řetězec "STDComm"či "Comm". Algoritmus projde ve všech networcích všechny `Accessy`, ujistí se, že se jedná o globální proměnné (tak bývají M merkery označeny) a pokud ano, zeptá se, jestli v něm výše zmíněné řetězce nejsou obsaženy. Pokud ano, vyhodnocená chyba se zapíše do seznamu *sW_GlobalMerkersErrs*. Globální proměnné nemusí být jenom M merkery, jedná se pouze o ujištění, že algoritmus hledá ve správné kategorii.

3.3.6 Zobrazení chyb

Tato kapitola se zabývá zobrazováním chyb v okně aplikace. Barevně rozlišuji errory (červená) a warningy (oranžová). Dále barevně ve výpisu chyb rozlišuji OB (fialová), FB (modrá) a FC (zelená). Metoda *ShowErrors* prochází všechny chyby,

které byly zaznamenány do instance *TIA_ProjectErrors* třídy `Errors`. Postupně vypisuje všechny obsažené chyby podle toho, jak byly do seznamů zařazeny. Každý nový `ListViewItem` představuje další vypisovanou chybu. Jeho `SubItemy` obsahují další informace jako typ chyby, název chyby, popis chyby, otevřený projekt, místo výskytu a network (kde se chyba vyskytuje). Každý řádek má zaškrtávací políčko. Metoda *ErrTypeColor* mění barvu položek typ chyby a *BlockColor* mění barvu položek místo výskytu, pokud se jedná o chybu v PLC blocích. Metoda s názvem *ErrorListView_ItemCheck* podbarvuje daný řádek, pokud někdo klikne na jeho políčko.

Stromové zobrazení networků, obsahujících chybu

Pro zobrazení networků, které obsahují chybu, používám v aplikaci třídu `TreeView`, která je zastoupena instancí *ProjectErrorTree*. Jedná se o třídu, nacházející se v podprostoru: "System.Windows.Forms.TreeView". Zde jednoduše vytvářím uzly stromové struktury. Hlavní uzel je název projektu, větví se na uzly *Devices&Networks* a *SW Blocks*.

Pokud se v projektu nachází chyby typu nepřiřazená zařízení v rámci PN topologie (error), je podbarvení uzlu *Devices&Networks* červené , pokud se tam nachází chyba typu nepřejmenované zařízení (warning), podbarvení je oranžové.

V uzlu *SW Blocks* se nacházejí 3 uzly: *OBs*, *FBs* a *FCs*. V každém z těchto uzlů se nacházejí další uzly a to jsou již názvy příslušných PLC bloků a nakonec jejich interface a networky. Network svítí červeně, pokud obsahuje error a oranžově, obsahuje-li warning. Pokud obsahuje oboje, svítí červeně (error má přednost před warningem).

Metody *ColorFindInterface* a *ColorFindNetwork* jsou pomocnými metodami, které zjišťují na základě obsažených chyb, jakou barvou má uzel mít.

Celé stromové zobrazení obstarává metoda *TreeViewShow*.

3.3.7 Uložení chyb do textového souboru

Tuto funkci zajišťuje metoda *SaveTxt_Click*, která se vykoná při kliknutí na tlačítko s nápisem *Save Errors to Txt...*. Uložení chyb funguje tak, že se nejdříve založí nová instance třídy `SaveFileDialog`, který se zobrazí metodou *ShowDialog*. Uživatel zvolí název souboru a stiskne OK. Pokud není název souboru - *filename* (string) prázdný, použije se třída `StreamWriter`, která do toto stejného souboru (*filename*) začne zapisovat. První `foreach` se stará o výpis řádků a vnořený `foreach` o výpis jednotlivých sloupců - položek typ chyby, název chyby, popis chyby, otevřený projekt, místo výskytu a network. Odděluje je znakem "|".

4 Testovací scénáře v TIA Portal projektech a výsledky testování

V této poslední kapitole je pojednáno o vytvoření různých testovacích scénářů v projektu, vytvořeném v programu TIA Portal. Zároveň je zde vyhodnoceno testování většího firemního projektu. Předmětem testování je zejména úspěšnost - zdali se podařilo správně implementovat všechny algoritmy, vyhledávající chyby v projektu, které byly popsány v kapitole 3 a dále rychlost testování.

4.1 Testovací projekt

Pro doložení výsledků testování jsem vytvořil v programu TIA Portal V16 testovací projekt s názvem *TestingProject_ErrorScenarios*, na kterém byla aplikace testována. Jsou v něm obsaženy úplně všechny chyby, které byly popsány v kapitole 2.2 v různých variantách. Projekt není určen ke splnění žádného automatizačního úkolu. Součástky a proměnné jsou v něm rozmístěny buď náhodně nebo tak, aby vytvořili nějakou chybu, kterou testuji. Testovací projekt obsahuje 1 hlavní organizační blok (Main OB1), 5 funkčních bloků, 2 funkce a 13 datových bloků (DB). Zařízení v projektu je 1 PLC typu Simatic 1516-3 PN/DP, 1 HMI typu KTP700 Basic PN a 1 průmyslový počítač (v projektu PC-System) typu IPC127E 3xPN/IE.

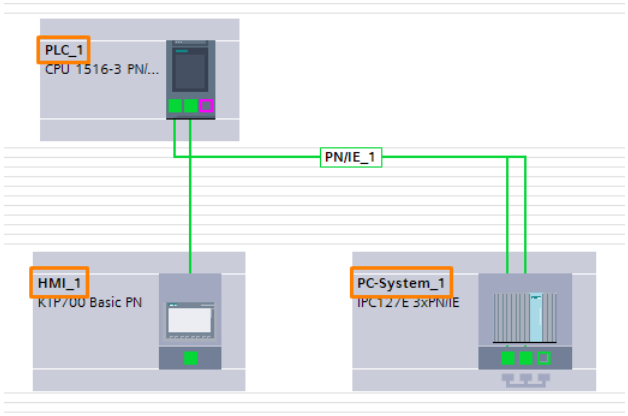
Testovací projekt bude dodán k bakalářské práci jako příloha.

4.1.1 Rozmístěné chyby

V projektu bylo záměrně rozmístěno celkově 60 chyb typu error a 112 chyb typu warning. V tomto počtu je zastoupena každá chyba, kterou má za úkol aplikace opravit.

Nepřejmenovaná zařízení

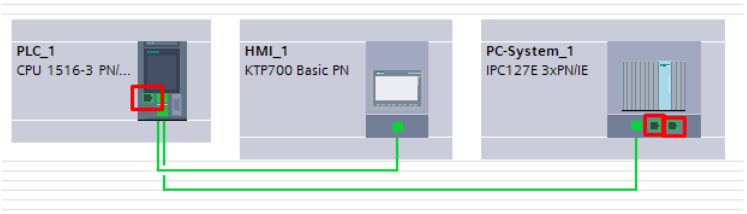
Záměrně jsem nepojmenoval ani jedno z umístěných zařízení, která jsou 3. Tato chyba je typu warning a tak by korektním výsledkem testování této chyby měly být 3 nalezené warningy, které jsou vidět na obrázku 4.1.



Obr. 4.1: Záměrně umístěná nepojmenovaná zařízení v hardwarové konfiguraci

Nepřiřazené porty zařízení v rámci PN topologie (nepřiřazená zařízení)

V záložce topology view hardwarové konfigurace jsem spojil pouze některá zařízení. Není přiřazen Port_1 na PROFINET interface_2 u řídícího PLC a na průmyslovém počítači nejsou přiřazeny 2 porty.



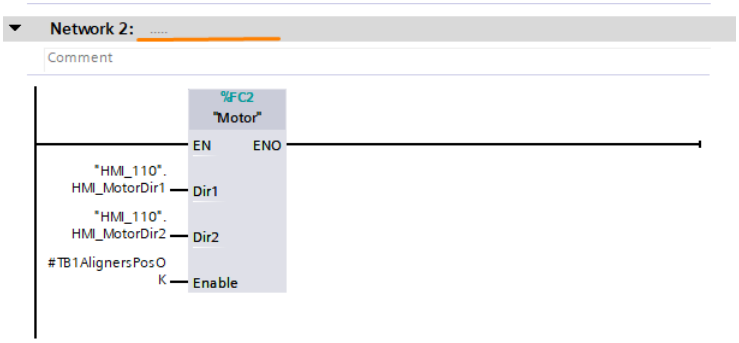
Obr. 4.2: Nepřiřazené porty na zařízeních v rámci PN topologie

Nepojmenované networky

Těchto chyb je v projektu celkem 24 na různých místech. Na obrázku 4.3 jsem vybral jeden exemplární příklad nepojmenovaného networku. Nepojmenované networky jsou vypsané v tabulce 4.1.

Tab. 4.1: Nepojmenované networky - výskyt v projektu

Blok	Nepojmenované networky
Main (OB1)	1,2,3 a 4
Station110 (FB1)	2,4,18,19 a 20
Station120 (FB2)	7 a 8
Station130 (FB3)	1,2,3,4,5,6 a 7
TestBlock1 (FB4)	1
TestBlock2 (FB5)	1
Func (FC1)	1 a 2
Motor (FC2)	1



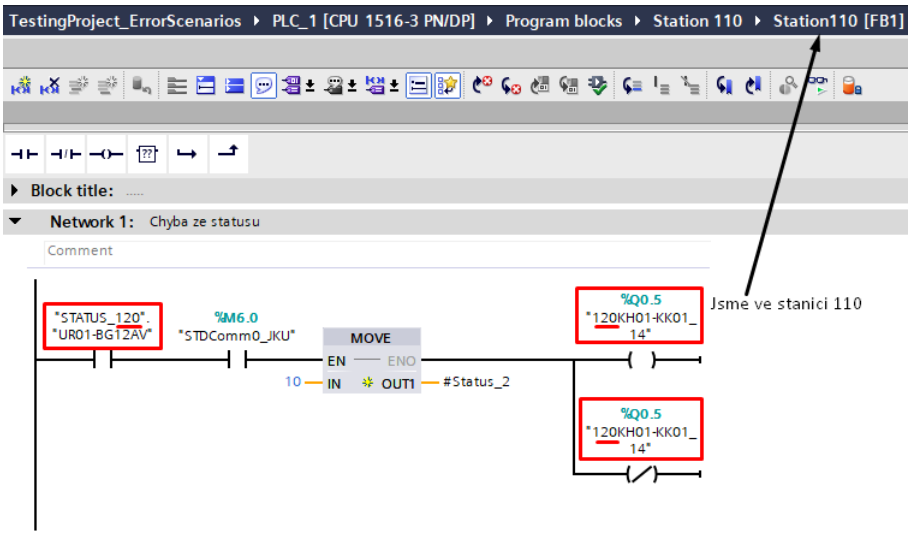
Obr. 4.3: Nepojmenovaný network (Station110, network 2)

Výskyt proměnných či tagů z cizí stanice (chybná symbolika)

Ukázka této chyby v testovacím projektu se nachází na obrázku 4.4. Na tomto obrázku jsou vidět 3 takové chyby - konkrétně jde o použití proměnné začínající "STATUS_**CisloStanice**" a PLC tagu, který zpravidla začíná číslem stanice. Těchto chyb se v testovacím projektu vyskytuje 30. Výskyty této chyby jsou zaznamenány v tabulce 4.2.

Tab. 4.2: Chybná symbolika - výskyt chyby v projektu

Blok	Networky obsahující chybnou symboliku
Station110 (FB1)	1,3,5,9,12,13,14,16 a 17
Station120 (FB2)	7
Station130 (FB3)	1,2,3,4,5,6



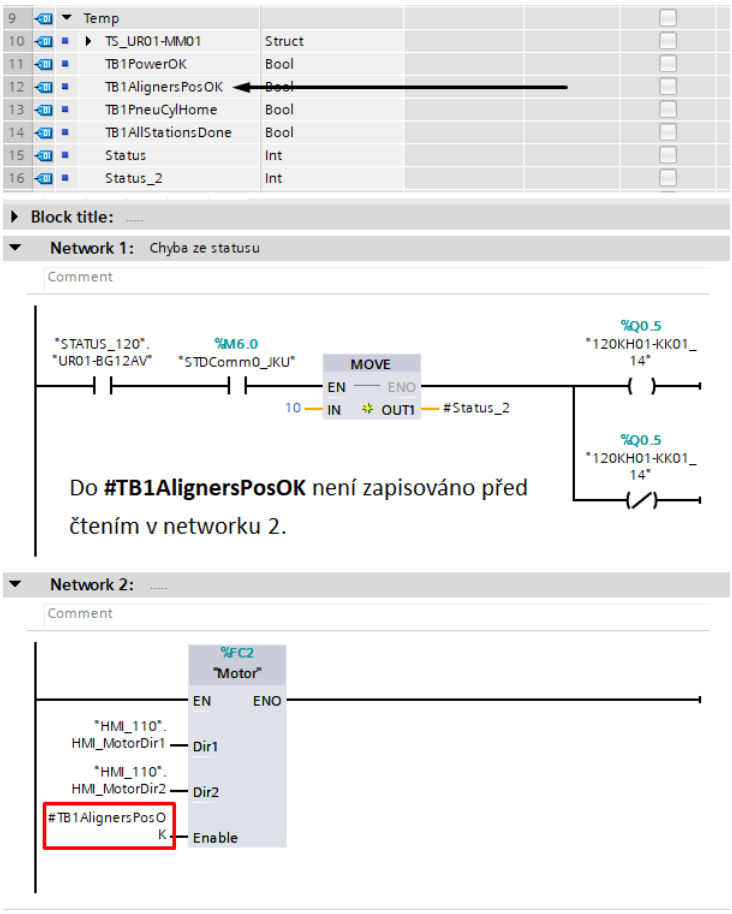
Obr. 4.4: 2 PLC tagy a 1 proměnná z cizí stanice (120) ve stanici 110, network 1

Čtení TEMP proměnných před jejich zápisem

Počet výskytů této chyby v testovacím projektu je 11. Výskyty této chyby jsou udány v tabulce 4.3. Příklad výskytu chyby v testovacím projektu je na obrázku 4.5.

Tab. 4.3: Výskyt zápisu do TEMP proměnné před jejím čtením

Blok	Networky obsahující předčasně čtenou TEMP proměnnou
Station110 (FB1)	2 a 3
Station120 (FB2)	6



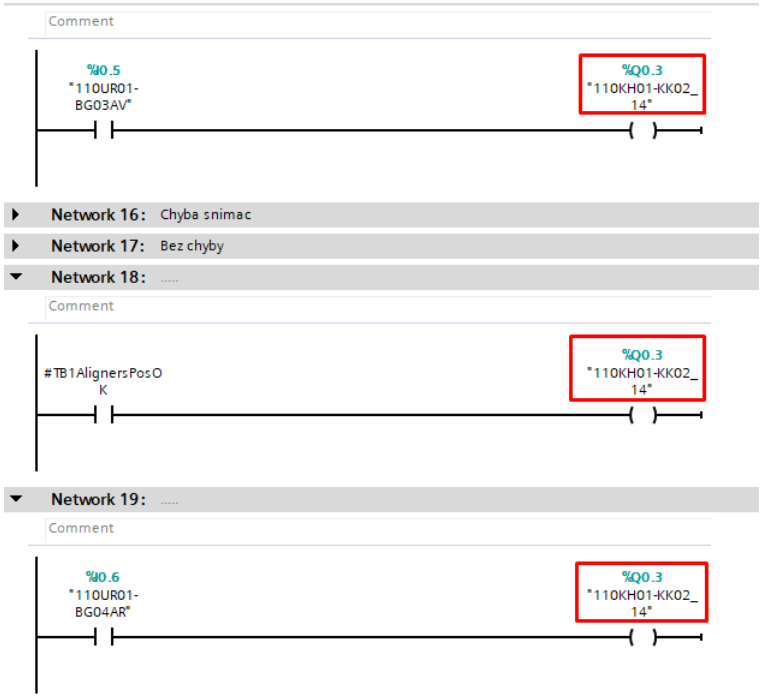
Obr. 4.5: Příklad výskytu chyby typu čtení neinicializované TEMP proměnné v testovacím projektu (Station110, network 2)

Použití běžných cívek na jednu proměnnou v rámci PLC bloku

Chyba, kdy je použito přiřazení do proměnné (BOOL) na více místech (s výjimkou cívek typu Set a Reset) je zastoupena v testovacím projektu osmkrát. Je zde ale problém, jestli tuto chybu vnímat pro například 4 takové přiřazení 4x a nebo pouze jednou jako jednu chybu. Moje aplikace počítá s tím, že zobrazí každý výskyt tohoto přiřazení, aby bylo snadnější chybu najít a odstranit. Výskyt této chyby je zobrazen v tabulce 4.4 a příklad na obrázku 4.6.

Tab. 4.4: Výskyt proměnné na více cívkách (coil)

Blok	Networky obsahující cívky se stejnou proměnnou
Station110 (FB1)	1,5,13,14,15,18 a 19
Station130 (FB3)	6



Obr. 4.6: Přiřazení do proměnné na 3 cívkách v bloku *Station110*

Nepoužité proměnné, deklarované v interface bloku vyskytující se v projektu

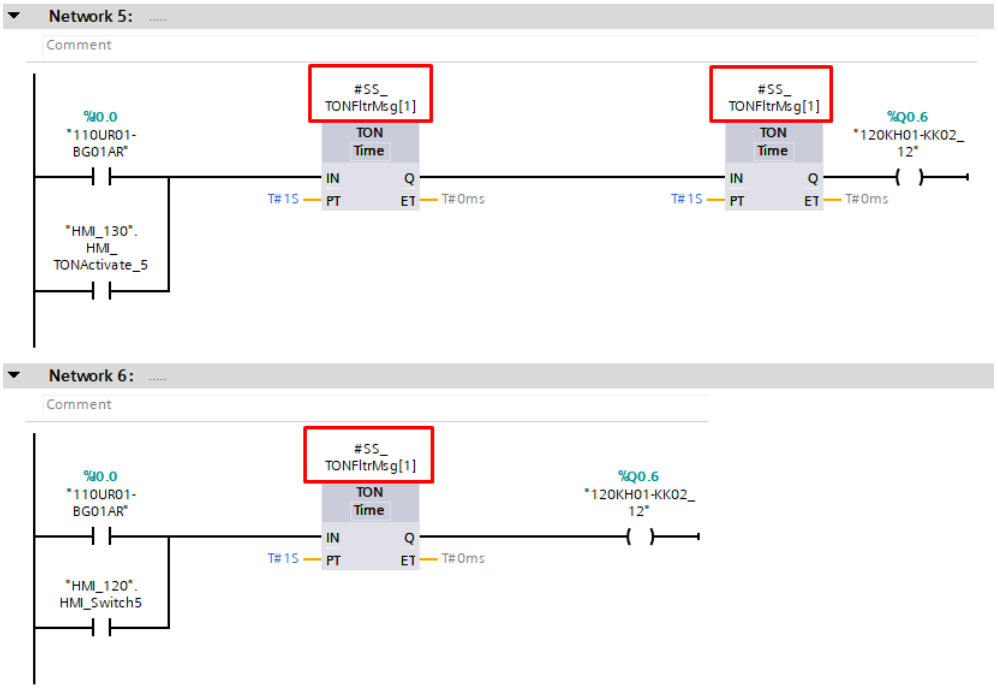
Jedná se o warning, který má v testovacím projektu největší zastoupení - těchto proměnných jsem v projektu umístil celkem 83. Nutno podotknout, že za tuto chybu se považují všechny proměnné ve strukturách a všechny prvky polí. Proto je tento počet tak vysoký. Výskyty této chyby jsou zaznamenány v tabulce 4.5.

Tab. 4.5: Výskyt nepoužitých proměnných

Blok	Počet výskytů
Main (OB1)	2
Station110 (FB1)	2
Station120 (FB2)	16
Station130 (FB3)	37
TestBlock1 (FB4)	8
TestBlock2 (FB5)	9
Func (FC1)	6
Motor (FC2)	3

Instance FB se stejnou proměnnou

V bloku *Station130* na networcích 1,2,3,5 a 6 je záměrně použita stejná instance časovače TON (stejná proměnná). Celkem se chyba vyskytuje 6x.

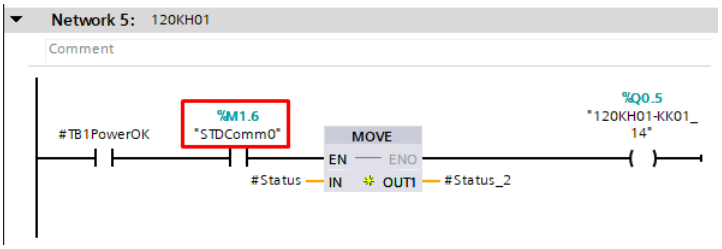


Obr. 4.7: Stejná proměnná *SS_TONFltrMsg[1]* na 3 časovačích TON (Station130, networky 5 a 6)

Pozn.: Aplikace umí pracovat i s poli různých datových typů (v tomto případě Timerů).

STDComm merkery

Globální proměnná typu merker se nachází v bloku pro stanici 110 na networcích 1 a 5 a v bloku pro stanici 120 na networku 7.



Obr. 4.8: *STDComm_0* merker použit v bloku stanice 110, network 5

Výskyt příliš velkých networků v testovacím projektu

V testovacím projektu se nachází celkem 2 networky, které lze klasifikovat jako zbytečně dlouhé. Network 3 ve stanici 110 je příliš dlouhý vertikálně, network 4 (ve stejné stanici) je naopak příliš dlouhý horizontálně (obsahuje 19 instrukcí typu MOVE, které jsou horizontálně zřetězeny).

4.1.2 Výsledky testování

Aplikace nalezla celkem 60 errorů a 113 warningů. Celkem našla 3 nepojmenovaná zařízení, 3 nepřipojené porty (bez partnera) v rámci topologie hardwarové konfigurace, 24 nepojmenovaných networků, 30 výskytů proměnných či tagů z cizí stanice, 11 čtení neinicializovaných TEMP proměnných a 8 přepisování booleovských proměnných pomocí několika cívek. Dále aplikace nalezla 83 nepoužitých proměnných a 5 použití stejných instancí pro časovač TON a 3 použití M merkeru s pojmenováním "STDComm".

U chyby příliš velký network byl zbytečně navíc nalezen network, který sice obsahuje značné množství kontaktů a vývodů u volaného FB, ale není příliš velký - vejde se ještě na obrazovku. To je způsobeno tím, že do výpočtu velikosti networku se nepočítají i velikosti názvů tagů a proměnných, přiřazených ke vstupům a výstupům do FC a FB a také tím, že každému `Partu` přiřazuji stejnou velikost, přičemž například instrukce `Move` zabírá více místa než instrukce `Contact`. Jedná se však o chybu estetickou (warning) a tudíž není tolik důležitá. Oprava tohoto problému by spočívala v zahrnutí velikostí přiřazených názvů proměnných ke vstupům a výstupům volaných FB a FC a stanovenou horizontální délkou pro každý typ instrukce zvlášť.

Ostatní nalezené chyby se však shodují s počtem umístěných chyb a na základě testování tohoto projektu lze označit aplikaci za funkční.

Doba testování

Doba testování velmi záleží na tom, jak výkonný počítač je a kolik na něm běží současně programů. Testovací projekt jsem za sebou otestoval pro porovnání desetkrát, abych mohl vypočítat průměrnou dobu jeho otestování. Z dob testování je jasně znát rozdíl mezi prvním a dalšími testováními, kdy první trvalo $\approx 50{,}93$ sekund, další se již pohybovali kolem 10 - 20 sekund. Měření bylo realizováno pomocí instance třídy `Stopwatch`, která se nachází v podprostoru `System.Diagnostics` a umožňuje měřit dobu vykonávání určitého úseku kódu. Příklad použití:

```
Stopwatch stopwatch = new Stopwatch();
stopwatch.Start();
//mereny kod
//mereny kod
//mereny kod
stopwatch.Stop();
```

Tabulka xx udává doby jednotlivých měření. Výsledky jsou zaokrouhleny na 2 desetinná místa.

Tab. 4.6: Doby testování testovacího TIA Portal projektu

č.m.	**Doba testování** [s]
1	50,93
2	13,34
3	12,45
4	11,99
5	12,26
6	11,87
7	21,23
8	22,10
9	19,95
10	26,00

Průměr: $\bar{t} = \frac{1}{n} \cdot \sum_{i=1}^{n} t_i = \frac{50{,}93+13{,}34+12{,}45+...+26{,}00}{10} = 20,21$ s
n...počet vzorků
i...i-tý vzorek
$\bar{t}$...průměr testovacích dob

t...jednotlivé testovací doby

Průměr bez prvního dlouhého měření: $\bar{t} = \frac{1}{n-1} \cdot \sum_{i=2}^{n} t_i = \frac{13{,}34+12{,}45+...+26{,}00}{9} = 16,80$ s

4.2 Firemní projekt

Na otestování mi byl firmou ICE Industrial Services poskytnut jeden firemní projekt. Tento projekt není finálním projektem, jednalo se o rannou verzi a proto by měl být výskyt chyb hojnější. Obecně platí, že čím větší projekt se testuje, tím více chyb se v něm najde. Také však záleží na tom, v jaké fázi vývoje se PLC projekt nachází. Je zřejmé, že na začátku vývoje bude mít projekt mnohem více chyb než v pozdější fázi vývoje. Dodaný testovací projekt má oproti jiným projektům nadprůměrnou velikost, nejedná se však o projekt velkého charakteru, který by například zahrnoval více PLC.

4.2.1 Velikost firemního projektu

Projekt obsahuje celkem 8 organizačních bloků, 78 funkčních bloků a 16 funkcí, přičemž jeden z OB je netestovatelný z důvodu ochrany "know-how". Celkový počet testovatelných networků je **2865**. V následující tabulce jsou vypsány všechny testovatelné PLC bloky a počet jejich networků. Pokud mají bloky jiný jazyk než ladder, nejsou v tabulce zahrnuty. Počet chyb obsažených ve firemním projektu **není známý**.

Tab. 4.7: Firemní projekt - počet networků jednotlivých PLC bloků - 1.část

Název PLC bloku	**Typ bloku**	**Počet networků**
naseFunkce	FC	2
Block_1	FB	1
Inputs	FC	12
Par	FB	0
SafetyAux	FB	20
STD_System	FB	16
MatCard	FB	20
Comm	FB	2
Comm_Mauss_Marca	FB	8

Tab. 4.8: Firemní projekt - počet networků jednotlivých PLC bloků - 2.část

Název PLC bloku	**Typ bloku**	**Počet networků**
Comm_Mauss_VDM	FB	8
Main	OB	4
MC-Interpolator	OB	0
MC-Servo	OB	0
OB_CompleteRestart	OB	3
OB_DiagnosticErrorInterrupt	OB	1
OB_ProgrammingError	OB	1
OB_RackOrStationFailure	OB	1
Main_Safety_RTG1	FB	1
F-Input	FB	50
F-Output	FB	16
IO_Module	FB	12
Logic	FB	18
MainSafety	FB	12
F-InComm	FC	1
F-OutComm	FC	0
F-OutCommInit	FC	0
FtoS_Interface	FC	59
StoF_Interface	FC	16
STD_DeviceModes	FC	9
STD_LSHigh	FC	3
STD_LSLow	FC	3
STD_Positioning	FC	8
STD_AI_Mon	FB	13
STD_BalluffBTL7-V50T	FB	23
STD_CameraKeyence_CV-X400	FB	68
STD_CH_AI	FB	14
STD_CH_AO	FB	13
STD_CH_DI	FB	6
STD_CodeReadKeyence_SR-1000	FB	26
STD_DI_Mon	FB	12
STD_DoorEuchner_MGB2	FB	28
STD_Drive1	FB	22
STD_Drive2	FB	30
STD_Integrator	FB	7
STD_Motor	FB	22
STD_MotorFESTO_CMMT	FB	36
STD_MotorSEW_MC07B	FB	34

Tab. 4.9: Firemní projekt - počet networků jednotlivých PLC bloků - 3.část

Název PLC bloku	**Typ bloku**	**Počet networků**
STD_MotorSEW-C_PosAxis	FB	76
STD_MotorSEWIpos	FB	49
STD_OpTrig	FB	9
STD_PartCounter	FB	16
STD_PrcsMsg	FB	43
STD_PT1Filter	FB	6
STD_SftySwchEuchner2_MBM	FB	18
STD_Station	FB	43
STD_Valve2WB	FB	38
STD_Valve2WB16V	FB	38
STD_Zone	FB	51
STD_Station	FB	43
STD_Valve2WB	FB	38
STD_Valve2WB16V	FB	38
STD_Zone	FB	51
STD_MotorFESTO_EMCA	FB	44
Zone000	FB	51
IM_000+UH01-KF11	FB	15
Station000	FB	8
Zone100	FB	286
IM_101+UR01-KF01	FB	5
IM_102+UR01-KF10	FB	4
IM_105+UR01-KF10	FB	2
IM___125+UR01-KF10	FB	1
IM_140+UR01-KF01	FB	2
IM_155+UR01-KF10	FB	1
IM_170+UR01-KF10	FB	1
IM_175+UR01-KF10	FB	1
IM_180+UR01-KF10	FB	1
IM_185+UR01-KF01	FB	2
IM_199+UR01-KF10	FB	2
Station101	FB	96
Station102	FB	96
Station105	FB	70
Station120	FB	56
Station125	FB	80
Station130	FB	66
Station140	FB	89

Tab. 4.10: Firemní projekt - počet networků jednotlivých PLC bloků - 4.část

Název PLC bloku	Typ bloku	Počet networků
Station145	FB	20
Station155	FB	86
Station160	FB	55
Station170	FB	92
Station175	FB	62
Station180	FB	111
Station185	FB	87
Station190	FB	18
Station197	FB	66
Station199	FB	71

4.2.2 Výsledky testování

Aplikace ve firemním projektu nalezla celkem 308 errorů a 1981 warningů (celkem tedy 2289 chyb). Celkem bylo nalezeno 20 nepřiřazených portů v rámci PN topologie, 30 nepojmenovaných networků, 176 výskytů proměnných či tagů z cizí stanice, 15 čtení neinicializovaných TEMP proměnných, 47 přepisování booleovských proměnných pomocí více cívek (které nejsou typu S/R), 1477 nepoužitých proměnných, 2 výskyty stejné proměnné pro timer na jednom networku - považováno za jednu chybu stejné proměnné pro instanci FB, 48 použití M merkerů typu "STDComm" a 474 příliš (vertikálně či horizontálně) dlouhých networků.

Doba testování

Stejně jako u testování testovacího projektu i zde jsem firemní projekt za sebou otestoval desetkrát a vypočítal průměrnou dobu jednoho otestování. Firemní projekt je mnohem větší než testovací a proto trvá mnohem delší dobu jej otestovat. Pro porovnání: U testovacího projektu se doby testování pohybovaly v řádech sekund až desítek sekund. Zde se doby testování pohybují v řádech minut až desítek minut.

Tab. 4.11: Doby testování firemního TIA Portal projektu

č.m.	**Doba testování**
1	13 m 9,31 s
2	8 m 8,49 s
3	8 m 35,57 s
4	9 m 32,15 s
5	10 m 13,18 s
6	9 m 13,96 s
7	8 m 47,98 s
8	8 m 40,16 s
9	11 m 59,03 s
10	8 m 23,61 s

Průměr: $\bar{t} = \frac{1}{n} \cdot \sum_{i=1}^{n} t_i = \frac{13{,}1552+13{,}1415+59{,}2833+...+8{,}3935}{10} = 9{,}6724$ m ≈ 9 m 40 s

Průměrná doba otestování firemního projektu byla tedy 9 minut a 40 sekund. Obsah firemního projektu je tajností firmy a tudíž jej nemůžu přiložit k práci jako testovací projekt. V příloze se však nachází obrázek, na kterém je firemní projekt otestovaný (A.3).

4.3 Nevýhody aplikace a plány vylepšení aplikace do budoucna

Velkou nevýhodu aplikace spatřuji v tom, že není schopna kontrolovat i použité UDT - user defined types. Je to z toho důvodu, že některé UDT neumožňují v XML předpisu přístup k jednotlivým členům. Všechny ostatní datové typy je aplikace schopna rozpoznávat. Další slabinou aplikace je již zmiňovaný výpočet horizontální a vertikální velikosti networku, který sice staví na dobrém základu (rekurzivní vracení největšího prvku ve větvi), ale nezahrnuje do výpočtu například velikost `Accessů`, nebo například to, že každý `Part` je horizontálně jinak dlouhý.

V plánu je opravit druhou zmíněnou nevýhodu a pokusit se nalézt řešení pro tu první. Další vylepšení mohou být například, aby uživatel měl možnost otestovat pouze jeden vybraný blok a nemusel by čekat na otestování celého projektu, umět chyby exportovat nejen do textového souboru, ale například i do Excelu, nebo zrealizovat algoritmy pro opravu chyb. Tyto návrhy nejsou firmou v rámci bakalářské práce požadovány, jedná se pouze o moje potenciální tipy pro vylepšení aplikace do budoucna.

Závěr

Práce shrnuje testovací techniky a úrovně testování a poté popisuje návrh a vývoj samotné aplikace a otestování její funkčnosti na dvou projektech.

Hlavním cílem této práce bylo vyvinutí testovací aplikace, která bude schopna najít chyby v prostředí TIA Portal. Pro vývoj aplikace byl zvolen jazyk C#, vývojové prostředí Visual Studio a typ aplikace windows form. Využívá otevřeného rozhraní TIA Portal Openness pro načítání projektu, jeho dat a export PLC bloků ve formátu XML. Aplikace byla vyvinuta pro použití ve firmě ICE Industrial Services.

Byla vytvořena třída `ProjectBlocks`, která strukturovaně sdružuje data, načtená z XML předpisů PLC bloků. Jedná se zejména o proměnné (interface) bloku a jeho networky, k čemuž samotný Openness nedokáže přistupovat. Díky tomu je poté možné PLC kód testovat vyvinutými vyhledávacími testovacími algoritmy. Po otestování aplikace chyby zobrazí, přehledně popíše jejich výskyt a klasifikuje.

Aplikace dokázala správně rozpoznat všechny chyby v testovacím projektu. Průměrná doba testování je 16,8 sekund. Průměrná doba otestování firemního projektu byla 9 minut a 40 sekund. Aplikace by měla být schopna otestovat jakýkoliv projekt pro TIA Portal V16, nicméně je třeba ji ve firmě nechat testovat různé projekty, protože jsem nemusel podchytit všechny možné případy, které mohou na cizích počítačích a projektech nastat. Aplikace tak může například vyhodit nějakou výjimku. Ve snaze není se výjimkám obcházet, ale spíše se vyhnout situacím, za kterých mohou nastat. Proto například v aplikaci téměř vůbec není použit příkaz *try-catch*. Vyladění aplikace tak, aby perfektně a zaručeně fungovala na všech počítačových konfiguracích a byla schopna otestovat všechny možné projekty bude následovat v dalších měsících a to od chvíle, kdy začne být aplikace používána. Zároveň tak získám i zpětnou vazbu ohledně různých vylepšeních aplikace ze strany uživatelů - programátorů. Aplikace umí otestovat pouze bloky, které jsou napsány v jazyce ladder.

Vybrané chyby jsou součástí zadání od firmy ICE Industrial Services, jejíž programátoři musí kód kontrolovat na tyto chyby vizuálně, protože je kompilace PLC kódu samotného TIA Portalu nedokáže detekovat - firma Siemens kompilaci těchto chyb v TIA Portalu (ještě) neimplementovala.

Literatura

[1] PRAX, Jan. *Automatizované testování PLC kódu pro TwinCAT 3 PLC* [online]. Brno, 2018 [cit. 2022-05-17]. Dostupné z: https://www.vut.cz/studenti/zav-prace/detail/112423. Bakalářská práce. Vysoké učení technické v Brně, Fakulta strojního inženýrství, Ústav automatizace a informatiky. Vedoucí práce doc. Ing. Jan Roupec, Ph.D.

[2] Co je testování softwaru? *Kitner* [online]. 2021 [cit. 2022-05-15]. Dostupné z: https://kitner.cz/testovani_softwaru/co-je-testovani-softwaru/

[3] Přehled testovacích technik. *Kitner* [online]. 2021 [cit. 2022-05-15]. Dostupné z: https://kitner.cz/testovani_softwaru/prehled-testovacich-technik/

[4] SHARMA, Lakshay. Software Testing. *ToolsQA* [online]. 2013 [cit. 2022-05-15]. Dostupné z: https://toolsqa.com/software-testing/software-testing/

[5] NIDHRA, Srinivas a Jagruthi DONDETI. Black Box and White Box Testing Techniques - A Literature Review. *International Journal of Embedded Systems and Applications* [online]. 2012, **2**(2) [cit. 2022-05-15]. DOI: 10.5121/ijesa.2012.2204. Dostupné z: https://www.researchgate.net/publication/276198111_Black_Box_and_White_Box_Testing_Techniques_-_A_Literature_Review

[6] Testovací strategie ortogonálních polí. *Algoritmy.net* [online]. 2016 [cit. 2022-05-15]. Dostupné z: https://www.algoritmy.net/article/24654/Ortogonalni-pole

[7] Udacity, 2015, *Statement Coverage - Georgia Tech - Software Development Process*, YouTube video. [cit. 2022-05-15] Dostupné z: https://www.youtube.com/watch?v=9PSrhH2gtkU&t=51s

[8] Udacity, 2015, *Branch Coverage - Georgia Tech - Software Development Process*, YouTube video. [cit. 2022-05-15] Dostupné z: https://www.youtube.com/watch?v=JkJFxPy08rk&t=168s

[9] Udacity, 2015, *Condition Coverage - Georgia Tech - Software Development Process*, YouTube video. [cit. 2022-05-15] Dostupné z: https://www.youtube.com/watch?v=ZnPmJd5aqyw&t=6s

[10] Code metrics - Cyclomatic complexity. *Microsoft* [online]. [cit. 2022-05-15]. Dostupné z: https://docs.microsoft.com/en-us/visualstudio/code-quality/code-metrics-cyclomatic-complexity?view=vs-2022

[11] Software Testing Levels. *Software testing fundamentals* [online]. [cit. 2022-05-15]. Dostupné z: https://softwaretestingfundamentals.com/software-testing-levels/

[12] FERNÁNDEZ, Borja, Enrique BLANCO a Alexey MEREZHIN'S. Testing & verification of PLC code for process control. *Proc. of the 14th Int. Conf.* [online]. 2013 (October 2013), San Francisco, USA. [cit. 2022-05-17]. Dostupné z: https://accelconf.web.cern.ch/ICALEPCS2013/papers/thppc080.pdf.

[13] Difference Between Field and Property in C#. *DifferenceBetween* [online]. [cit. 2022-05-15]. Dostupné z: https://www.differencebetween.com/difference-between-field-and-vs-property-in-c/

[14] DEITEL, Paul a Harvey DEITEL. *C# 2010 For programmers*. Fourth edition. RR Donnelley in Crawfordsville, Indiana: Pearson Education, 2011. ISBN 978-0-13261820-5. [cit. 2022-05-17].

[15] C# foreach loop. *Programiz* [online]. [cit. 2022-05-15]. Dostupné z: https://www.programiz.com/csharp-programming/foreach-loop

[16] RICHTER, Miloslav, et al. *Praktické programování v C++* [online]. Brno, 2004, [cit. 2022-05-17]. Dostupné z: http://vision.uamt.feec.vutbr.cz/PPC/others/prednes.pdf. Elektronické texty.

[17] KVÁČ, Josef. Jak na openness?. *Siemens* [online]. [cit. 2022-05-15]. Dostupné z: https://cz.webinar.siemens.com/jak-na-openness/room

[18] KVÁČ, Josef. TIA Portal Openness Generování projektu. *Docplayer* [online]. Praha, 2016 [cit. 2022-05-15]. Dostupné z: https://docplayer.cz/30075547-Tia-portal-openness-generovani-projektu-https-workspace-automation-html

[19] How to delete all files and folders in a directory?. *StackOverflow* [online]. [cit. 2022-05-15]. Dostupné z: https://stackoverflow.com/questions/1288718/how-to-delete-all-files-and-folders-in-a-directory

Seznam symbolů a zkratek

SW	Software
HW	Hardware
HW konfigurace	Hardwarová konfigurace
PLC	Programmable logic controller (Programovatelný logický automat)
MES	Manufacturing Execution System (Výrobní informační systém)
CYC	Cyclomatic complexity (Cyklomatická složitost)
SCADA	Supervisory Control And Data Acquisition
TIA Portal	Totally integrated automation Portal (vývojové prostředí)
OB	Organizační blok
DB	Datový blok
FB	Funkční blok
FC	Funkce
WinForms	Windows form application (okenní formulářová aplikace ve Visual studiu
WPF	Windows presentation foundation
GUI	grafiké uživatelské rozhraní
XML	eXtensible Markup Language

Seznam příloh

A Přiložené obrázky **87**

A.1 Přehled hlavní třídy App_Init_Screen 87

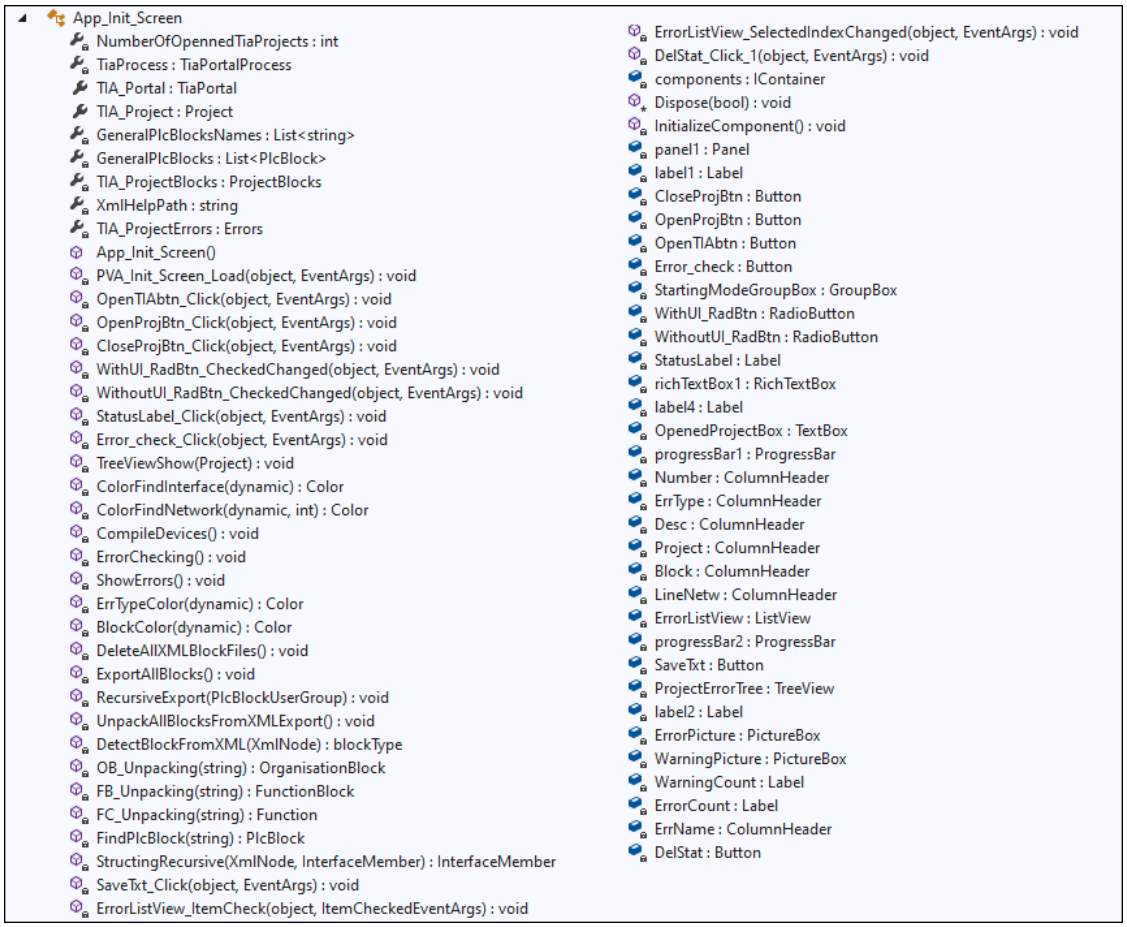
A.2 Flow diagram procesu testování 88

A.3 Chyby firemního projektu v aplikaci po testování 89

B Obsah přiloženého CD **91**

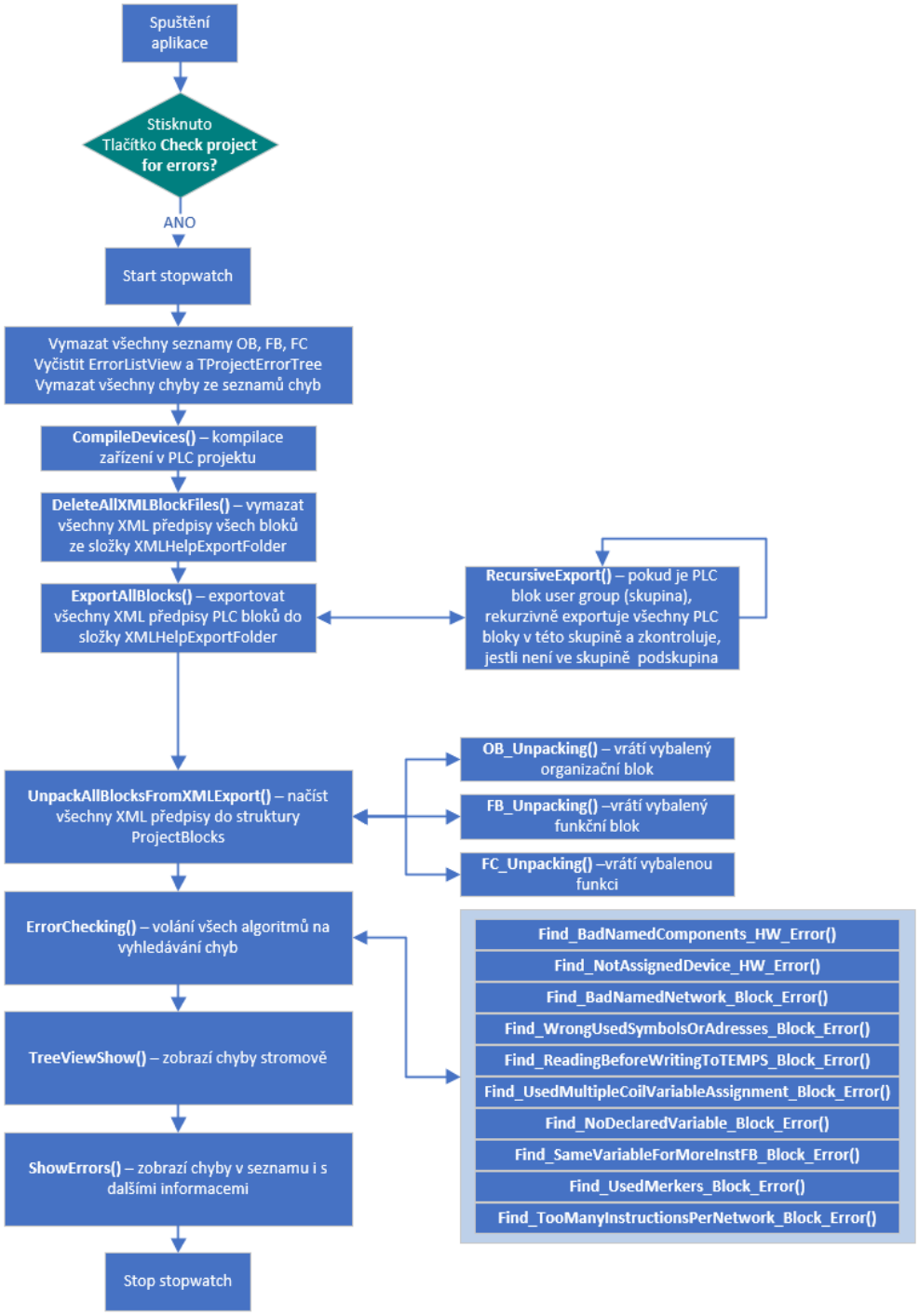
A Přiložené obrázky

A.1 Přehled hlavní třídy App_Init_Screen



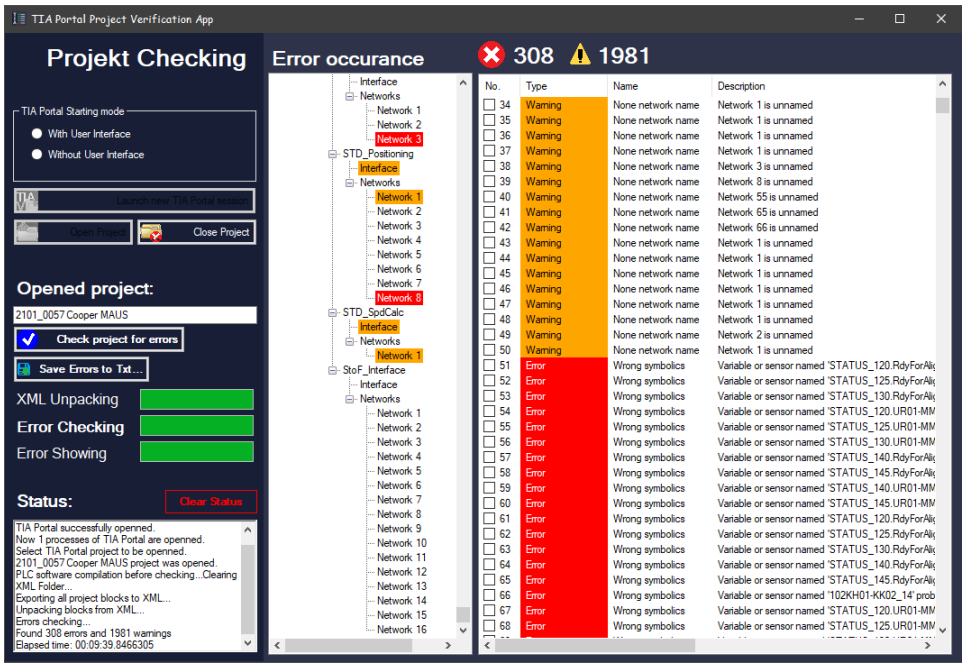
Obr. A.1: Přehled atributů a metod hlavní třídy *App_Init_Screen*

A.2 Flow diagram procesu testování



Obr. A.2: Flow diagram nejpodstatnějších metod v procesu testování

A.3 Chyby firemního projektu v aplikaci po testování



Obr. A.3: Chyby ve firemním projektu - po tom co je aplikace nalezla

B Obsah přiloženého CD

`PRILOHY_BP_SYKORA_ONDREJ` ..... Hlavní adresář přiloženého CD
- `Sykora_BP.pdf` ..... Vlastní text bakalářské práce
- `AppForTestingCodePLC` ..... Adresář aplikace (Visual Studio)
- `TestingProject_ErrorScenarios` ..... Adresář testovacího projektu (TIA Portal)