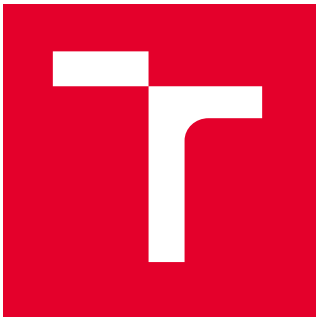


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

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

NÁSTROJ PRO TVORBU AUTOMATICKÉ DOKUMENTACE SW PRO PLC

AUTOMATIC DOCUMENTATION TOOL FOR PLC SOFTWARE

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — **Bc. Jan Navrátil**
AUTHOR

VEDOUCÍ PRÁCE — **Ing. Petr Petyovský, Ph.D.**
SUPERVISOR

BRNO 2024

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

Diplomová práce

magisterský navazující studijní program **Kybernetika, automatizace a měření**

Ústav automatizace a měřicí techniky

Student: Bc. Jan Navrátil ***ID:*** 221006

Ročník: 2 ***Akademický rok:*** 2023/24

NÁZEV TÉMATU:

Nástroj pro tvorbu automatické dokumentace SW pro PLC

POKYNY PRO VYPRACOVÁNÍ:

1. Nastudujte problematiku a teorii pro tvorbu nástroje pro automatické generování UML diagramů ze zdrojových textů programu.
2. Zvolte konkrétní programové prostředí využívané v oblasti průmyslové automatizace. Popište formát používaný k reprezentaci diagramů dokumentujících strukturu programu.
3. Popište vlastnosti jazyka Structured text konkrétní zvolené rodiny PLC a navrhněte vhodnou reprezentaci tohoto jazyka pomocí UML.
4. Navrhněte a implementujte analyzátor pro jazyk Structured text.
5. Vytvořte generátor výstupu ve formátu, který podporují dokumentační nástroje zvoleného prog. prostředí.
6. Zvolte demonstrační úlohu využitelnou v laboratoři číslicové řídicí techniky na UAMT ověřující funkčnost realizovaného generátoru.
7. Realizujte navrženou demonstrační úlohu a prezentujte její automaticky vygenerovanou dokumentaci.
8. Zhodnoťte dosažené výsledky a navrhněte další možná rozšíření.

DOPORUČENÁ LITERATURA:

[1] FOWLER, M. Destilované UML. Praha: Grada, 2009, 173 s. ISBN 978-80-247-2062-3.

[2] TM246 - Structured Text [online]. B&R Strasse 1 5142 Eggelsberg Rakousko: B&R Industrial Automation, 2023 [cit. 2023-09-11]. Dostupné z: <https://www.br-automation.com/cs/ke-stazeni/automation-academy/training-modules/control-technology/tm246-structured-text-st/>.

Termín zadání: 5.2.2024 ***Termín odevzdání:*** 15.5.2024

Vedoucí práce: Ing. Petr Petyovský, Ph.D.

Konzultant: Ing. Pavel Vávra, B+R automatizace

doc. Ing. Petr Fiedler, Ph.D.
předseda rady studijního programu

ABSTRAKT

Tato práce se zabývá implementací nástroje pro automatické generování diagramů z anotovaného zdrojového kódu v jazyce Structured Text pro PLC aplikace. Cílem je zefektivnit proces tvorby dokumentace softwarových projektů tím, že umožní vývojářům automaticky generovat diagramy, které popisují strukturu a chování jejich aplikací. Práce se dále zaměřuje na průzkum existujících nástrojů pro tvorbu diagramů a jejich možností. Na základě tohoto průzkumu je navržen a implementován překladač, který extrahuje relevantní informace z kódu a generuje diagramy ve standardizovaném grafickém jazyce UML. Výsledkem je implementovaný nástroj, který umožňuje softwarovým vývojářům snadno vytvářet a aktualizovat diagramy jejich aplikací, což přispívá k rychlejšímu a přesnějšímu procesu dokumentace.

KLÍČOVÁ SLOVA

automatická dokumentace kódu, vývojový diagram, Structured Text, PlantUML, Doxygen, Flex, Bison

ABSTRACT

This thesis focuses on implementing a tool for automatic generation of diagrams from annotated source code written in Structured Text for PLC applications. The aim is to streamline the documentation process of software projects by enabling developers to automatically generate diagrams that describe the structure and behavior of their applications. The thesis further explores existing tools for diagram creation and their capabilities. Based on this exploration, a compiler is designed and implemented, which extracts relevant information from the code and generates diagrams in the standardized graphical language UML. The result is an implemented tool that allows developers to easily create and update diagrams of their applications, contributing to a faster and more accurate documentation process.

KEYWORDS

automatic code documentation, flowchart, Structured Text, PlantUML, Doxygen, Flex, Bison

NAVRÁTIL, Jan. *Nástroj pro tvorbu automatické dokumentace SW pro PLC*. Diplomová práce. Brno: Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, Ústav automatizace a měřicí techniky, 2024. Vedoucí práce: Ing. Petr Petyovský, PhD.

Prohlášení autora o původnosti díla

Jméno a příjmení autora:	Bc. Jan Navrátil
VUT ID autora:	221006
Typ práce:	Diplomová práce
Akademický rok:	2023/24
Téma závěrečné práce:	Nástroj pro tvorbu automatické dokumentace SW pro PLC

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 ....

podpis autora*

*Autor podepisuje pouze v tištěné verzi.

PODĚKOVÁNÍ

Děkuji vedoucímu diplomové práce Ing. Petrovi Petyovskému Ph. D. za pedagogickou a odbornou pomoc a další cenné rady při konzultování mé semestrální práce. Dále také mým konzultantům Pavlu Vávrovi a Michalu Vavríkovi ze společnosti B&R Industrial Automation za podnětné rady a poskytnutý vzorový projekt. Nakonec bych chtěl poděkovat mé rodině, která se mnou měla trpělivost při psaní této práce.

Obsah

Seznam obrázků

Seznam tabulek

Seznam výpisů

Úvod

Při vývoji aplikací nejen pro PLC se běžně vytváří dokumentace sofwarového projektu. Pro názornost a přehlednost je vhodné vytvářet diagramy, aby bylo možné rychleji pochopit strukturu a chování aplikace. Rozvoj metodik konstruování diagramů používaných v softwarovém inženýrství vedl k vytvoření standardizovaného grafického jazyka UML, který nabízí konkrétní typy diagramů určených pro vizualizaci daného řešení. Jelikož se jedná rovněž o jazyk, nabízí se možnost vytvoření překladače mezi UML a textovými zdrojovými kódy. Takové překladače umožňují šetřit práci po fázi návrhu aplikace vygenerováním kostry aplikace nebo po fázi implementace vygenerováním dokumentace ze zdrojového kódu.

Tato práce si bere jako hlavní cíl implementaci řešení, které z anotovaného zdrojového kódu v jazyce Structured Text umožní vytvořit vhodný diagram tak, aby byl graficky popsán algoritmus konkrétního programu v PLC aplikaci. Pro splnění tohoto cíle bude potřeba provést průzkum konkrétních nástrojů pro ruční nebo automatizovanou tvorbu a vizualizaci diagramů a jejich možnosti rozšíření. Aby bylo možné tvorbu diagramů automatizovat, je potřeba najít vhodnou reprezentaci diagramů, která bude zvoleným vizualizačním nástrojem podporována. Dále bude potřeba navrhnout a vytvořit překladač, který z jazyka Structured Text extrahuje relevantní informace a vygeneruje diagram ve zvolené cílové reprezentaci. Nakonec je potřeba otestovat správnost vytvořeného diagramu na demonstračních úlohách a srovnat vygenerovanou dokumentaci s ručně vytvořenou.

Implementované řešení by mělo přispět k zefektivnění procesu dokumentace průmyslových aplikací, kdy je vývojáři umožněno generovat diagramy automaticky z již existujícího zdrojového kódu. Tímto způsobem by bylo možné vytvářet dokumentaci rychleji a s menším množstvím chyb.

1 Aplikace pro tvorbu diagramů

V této kapitole jsou představeny během práce zvažované aplikace specializované na generování a vizualizaci diagramů. U každé z nich je rozebrán způsob, jak by bylo možné ji propojit s implementovaným překladačem, případně jak propojit překladač se souborovým formátem, se kterým operují.

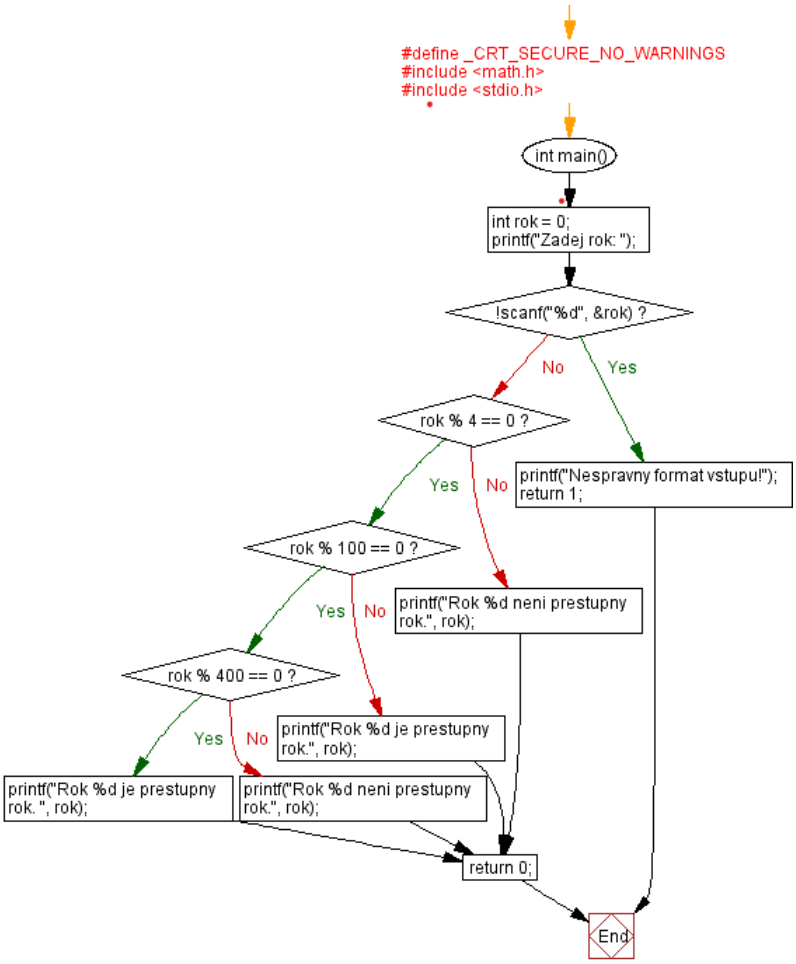
1.1 Aplikace Microsoft Visio

Microsoft Visio je aplikace určená k tvorbě grafů od společnosti Microsoft. První verze byla vydána již v roce 1992 společností ShapeWare. Později v roce 2000 ji odkoupila společnost Microsoft a integrovala jej do kancelářského balíku nástrojů Office. [3] Dostupné funkce aplikace, jako například možnosti ukládání souborů v konkrétních formátech, se odvíjejí od zvolené edice. V rámci předplatného Office 365 pro komerční použití lze Microsoft Visio použít k vytváření nejběžnějších typů diagramů přes webové rozhraní. Levnější roční plán předplatného přináší hlavně větší počet šablon, mezi nimiž je i UML. Dražší plán předplatného má kromě dalších, pokročilejších typů diagramů také možnost automatického generování diagramů z dat. Konkrétněji se ovšem jedná o data z databází. Microsoft Visio tedy umožňuje vytvářet vývojové diagramy, ale neumožňuje je generovat ze zdrojového kódu.

Verze Visio z roku 2013 začala pracovat s novým souborovým formátem `.vsdx`, který oproti staršímu binárnímu formátu `.vsd` má formu archivu ve formátu kompatibilním s formátem `.zip`. V tomto archivu je dokument uložen ve formě XML souborů a příloh, které nelze v XML formátu reprezentovat. [4] Entity ve Visio dokumentu lze tak číst a upravovat i pomocí jiných aplikací, než je Microsoft Visio. Je zde tudíž možnost takový dokument vygenerovat programově. Další možností je vytvořit rozšíření pro Office pomocí oficiálního aplikačního rozhraní v jazyce Javascript. [5]

1.2 Program Code Visual to Flowchart

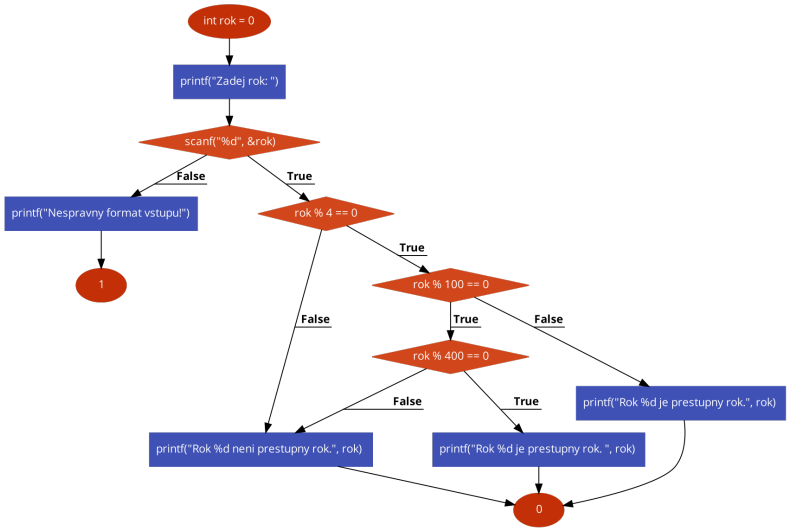
Code Visual to Flowchart je placený a proprietární program od společnosti Fatesoft. Podle webových stránek, kde je ke stažení demoverze programu [6], je možné generovat vývojový diagram jako objekt přímo do nástrojů z balíčku Office, včetně Microsoft Visio nebo do formátu HTML. Při praktickém otestování lze z nabídky podporovaných jazyků zvolit také Structured Text, ovšem při generování diagramu se zobrazí zpráva, že tento jazyk bude podporován v budoucích verzích. Navíc podle zastaralosti prostředí programu, navrženého pro Windows XP, lze usuzovat, že již nebude nadále vyvíjen. Jelikož zdrojový kód je proprietární, není zde jiný způsob, jak pro tento program implementovat vlastní analyzátor zdrojového kódu.



Obr. 1.1: Diagram vygenerovaný v programu Visutin

1.3 Program Visutin

Visutin je rovněž placený a propritetární program od společnosti Aivosto z roku 2003. [7]. Oproti předchozímu nástroji analýza kódu funguje daleko lépe. Na obrázku č. 1.1 je vygnenerovaný diagram jednoduchého algoritmu v jazyce C na výpočet přestupního roku zadaného z příkazového řádku. Diagram lze dále ručně v editoru upravovat a v plné verzi ukládat do vlastního formátu, dokumentu Word a také do běžných obrazových formátů. Program nativně nepodporuje generování ze Structured Textu a jelikož je proprietární, opět neumožňuje integrování vlastního analyzátoru.



Obr. 1.2: Diagram vygenerovaný v nástroji Code2flow.com

1.4 Aplikace Code2flow.com

Code2flow je placená webová internetová aplikace pro generování diagramů pouze pomocí vlastního jazyka. Jelikož je tento jazyk podobný vyšším programovacím jazykům, nabízí se možnost vytvoření překladače z jazyka Structured Text. Vygenerovaný diagram ovšem již nelze později upravovat a ani pomocí instrukcí nelze pevně nastavit rozložení prvků v diagramu. Volně přístupná demoverze je omezena také maximálním počtem 50 prvků v jednom diagramu. [10]

1.5 Aplikace Dia a komponenta python-dia

Dia je svobodná aplikace s otevřeným kódem pro kreslení diagramů. Původně byla vyvinuta pro grafické prostředí GNOME v roce 1998. V současné době existuje i verze pro operační systém Windows, i když poslední aktualizace je z roku 2012. Tato aplikace je inspirována funkcemi Microsoft Visio. [8] Umožňuje kreslit objekty definované pomocí XML souborů. Již předinstalované jsou objekty pro kreslení vývojových a UML diagramů a také elektrotechnických, chemických a procesních schémat. Vytvořený diagram lze exportovat jak do grafických, tak do XML formátů. Samotný program používá vlastní XML formát s příponou `.dia`. I u tohoto nástroje se tedy nabízí vytvořit překladač do tohoto formátu.

Aplikace Dia obsahuje volitelnou komponentu `python-dia`, pomocí kterého lze vytvářet rozšíření psaná v jazyce Python. [9] Aplikaci lze tak ovládat skrze programovací rozhraní, které je ovšem pro pouhé vytváření diagramů člověkem neintuitivní a náročné. Nabízí se však opět možnost generovat tento kód programově.

1.6 Nástroj diagrams.net

Diagrams.net, také známější pod svým původním názvem draw.io, je multiplatformní nástroj pro vytváření diagramů napsaný v jazyce `HTML5` a JavaScript od společnosti JGraph Ltd. Existuje jak verze pro webový prohlížeč, tak téměř rovnocenná offline desktopová verze pro operační systémy Linux, macOS, Chrome OS a Windows. Diagrams.net lze také jako nástroj integrovat do vývojářských kolaborativních prostředí Google Workspace, Microsoft 365, Confluence Jira, Notion a NextCloud. Jako doplněk ho lze spustit i z populárního prostředí Visual Studio Code. Zároveň umožňuje přímé ukládání do cloudových uložišť Dropbox, Google Drive, OneDrive a repozitářů Github a Gitlab.com. [11]

Části zdrojového kódu tohoto nástroje jsou otevřené a vydány pod licencí Apache 2. Placená verze pro podniky je dostupná jako aplikace pro cloudové nástroje společnosti Atlassian. V této verzi jsou dostupné funkce pro simultánní editování více uživateli a historie souborů. [12]

Verze pro individuální uživatele je zdarma, ovšem poskytuje všechny funkcionality očekávatelné od moderního diagramového nástroje. Umožňuje seskupovat vybrané objekty v diagramu, zarovnávat je nebo upravit jejich rozložení na stránce. Každý objekt může obsahovat metadata, jako komentáře nebo hypertextové odkazy. Textová pole můžou být vícejazyčná, tudíž lze udržovat jeden stejný diagram pro vícero jazyků. Do výkresu lze vkládat vlastní obrázky a rukou kreslené obrazce. V rámci jednoho dokumentu lze mít více pojmenovaných stránek.

Vytvořený dokument je ukládán v XML formátu s příponou `.drawio`. Jednotlivé stránky lze exportovat jako rastrový nebo vektorový obrázek. Celý dokument lze exportovat do HTML nebo PDF formátu. V současnosti je ve vývoji i export do formátu `vsdx` používaným nástrojem Microsoft Visio. Jednodušší diagramy lze rovněž nasdílet jako odkaz na internetovou aplikaci, kde celý graf je uložen v parametrech odkazu.[16] Každý z těchto formátu podporuje různé funkcionality. Do některých lze volitelně uložit i kopii diagramu, takže lze takto exportovaný diagram znovu otevřít v editoru a upravit jej.

Diagrams.net obsahuje knihovny symbolů pro technickou dokumentaci, UML diagramy a další softwarové, síťové a databázové diagramy mimo normu UML. Krom symbolů obsahuje rovněž grafické elementy pro uživatelská rozhraní, které například pocházejí z populární sady kaskádových stylů Bootstrap. Spolu v kombinaci s křížovými odkazy na objektech a s exportem do PDF nebo HTML lze takto jednoduše vytvořit interaktivní dokumentaci s moderním vzhledem.

Kromě ručního vytváření a spojování jednotlivých objektů lze diagram vytvořit podle zvoleného rozvržení pouhým nadefinováním uzlů a hranami mezi nimi, jejichž tvar se vytvoří automaticky. Obdobně lze vygenerovat konkrétní typy dia-

gramů pomocí textových značkovacích jazyků podobajících se jazyku Markdown. Podporované jsou konkrétně syntaxe `PlantUML` a `Mermaid`. Diagrams.net používá i svůj vlastní pseudokód pro vytvoření jednoduchých diagramů a složitější grafy umí vytvořit z údajů v tabulkových CSV souborech. V neposlední řadě lze vytvořit diagram entitně relačního databázového modelu přímo z SQL příkazů. [15]

1.7 Nástroj Doxygen a Graphviz

Doxygen je jeden z populárních nástrojů pro generování dokumentace ze zdrojových kódů pod volnou licencí GNU GPL. Sám o sobě tedy nepředstavuje grafický nástroj pro tvorbu diagramů, ovšem umožňuje generovat grafickou dokumentaci pomocí integrované sady nástrojů Graphviz.

Samotný nástroj Graphviz je šířený pod open-source licencí EPL a obsahuje řadu nástrojů pro manipulaci a analýzu grafů včetně nástroje `dot`, který umožňuje ze zdrojového souboru v jazyce DOT vytvořit grafickou reprezentaci grafu v běžně používaných formátech včetně PostScript. Jazyk DOT popisuje tři základní objekty, a to grafy, uzly a hrany. S použitím dalších nástrojů jako například Graphviz Visual Editor lze grafy interaktivně upravovat. [13]

Doxygen dokáže generovat textovou i grafickou dokumentaci z komentářů a zdrojového kódu v jazycích `C`, `C++`, `C#`, `Python`, `VHDL`, `Fortran`, `Java` a dalších. Uživatelsky ho lze rozšířit o podporu dalších jazyků použitím předzpracujících filtrů, které zdrojový kód upraví do požadované podoby. . Typicky se tedy jedná o transpilátory překládající do podporovaných jazyků. Pomocí nástroje Graphviz umožnňuje Doxygen přímo z textu generovat grafy závislostí, grafy dědičnosti tříd a grafy volání. Rovněž umožňuje vykreslování vlastních grafů pomocí značkovacího jazyka `PlantUML`.

Generování dokumentace se provede z příkazové řádky příkazem `doxygen` a je řízeno hlavně podle konfiguračního souboru `Doxyfile`. V něm jsou jednotlivé parametry uloženy pod pojmenovanými tagy. Soubor lze jednoduše editovat pomocí nástroje Doxywizard s grafickým uživatelským rozhraním. [14]

1.8 Srovnání uvedených aplikací

V této kapitole jsem představil jednotlivé dostupné nástroje pro tvorbu diagramů. Aplikace Code Visual to Flowchart a Visutin sice jsou přímo stvořeny za účelem generování diagramů ze zdrojových textů, ovšem neumožňují implementaci rozšíření. Microsoft Visio rozšíření umožňuje, ale musel bych nastudovat jimi používané

Tab. 1.1: Přehled řešení představených v kapitole

Název	Licence	Poslední stabilní verze	Možnosti automatizovaného generování diagramů
Microsoft Visio	Komerční	14. 11. 2023	úprava `.vsdx` souborů
Code Visual to Flowchart	Komerční	22. 4. 2009	Není
Visutin	Komerční	2. 2. 2023	Není
Code2flow.com	Komerční	2022	Vlastní jazyk
Dia	GNU GPL	5. 9. 2014	python-dia, úprava XML souborů
Diagrams.net	Apache	15. 12. 2023	Viz str. 16
Doxygen	GNU GPL	25. 12. 2023	předzpracování zdrojových souborů

knihovny. Code2flow.com používá ke generování vlastní jazyk, ale jedná se o placený nástroj, obdobně jako Microsoft Visio. Aplikace Dia je svobodná a rozšířitelná pomocí knihovny `python-dia`. Při pohledu na poslední datum aktualizace lze však usoudit, že se do budoucna aplikace nebude dále vyvíjet.

Zvoleným programovým prostředím pro řešení diplomové práce byla původně zvolena aplikace diagrams.net, jelikož je v současnosti nadále vyvíjená, je multiplatformní, její kód je otevřený a umožňuje generování diagramů z několika již existujících typů pseudokódů, které jsou rozebrány v následující kapitole. Při rešerži těchto pseudokódů jsem ovšem zjistil, že většina z nich při použití s touto aplikací nesplňuje možnost vytváření hypertextových odkazů. To pak neumožňuje vytvářet interaktivní dokumentaci, která by na softwarový projekt nahlížela jako na hiearchii, kde jednotlivé komponenty jsou provázány a lze mezi nimi snadno přecházet právě pomocí hypertextových odkazů.

Proto pro řešení diplomové práce namísto aplikace diagrams.net používám dokumentační nástroj Doxygen, který oproti diagrams.net je přímo uzpůsoben pro vytváření dokumentace k softwarovým projektům a nikoliv pouze ke generování a editaci diagramů. Provádí tudíž nezanedbatelnou část požadované funkcionality (například automatické vytváření jednotlivých stránek) v práci implementovaného nástroje sám o sobě. Navíc se jedná o svobodný a oproti aplikaci diagrams.net daleko podrobněji zdokumentovaný nástroj, tudíž implementace rozšíření bude pro něj jednodušší.

2 Reprezentace diagramů v aplikaci diagrams.net

V následující kapitole jsou představeny různé jazyky, jenž slouží k textové reprezentaci diagramů ve zvolené aplikaci diagrams.net. Tyto jazyky nedefinují přesné rozmístění objektů v diagramu, jelikož tento proces provádí samotná aplikace. Proto vynechávám z této práce formát XML, ve kterém je výsledný dokument uložen a obsahuje přesné rozmístění objektů. Překlad do formátu XML by obnášel nutnost implementovat v překladači také algoritmus řešící rozmístění prvků a to by, když tuto funkci aplikace diagrams.net poskytuje, bylo zbytečné. Kódy ke generování diagramů lze vkládat přes grafické uživatelské rozhraní v nabídce `Arrange->Insert->Advanced`.

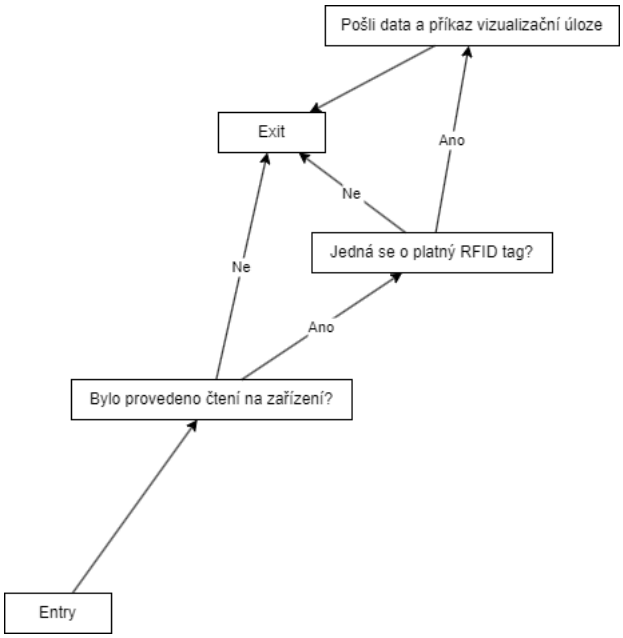
2.1 Textový popis

Aplikace diagrams.net používá vlastní textový pseudokód, pomocí kterého lze vytvořit jednoduché diagramy. Na každém řádku kódu je definováno jedno propojení mezi dvěma pojmenovanými uzly pomocí operátoru šipka. Volitelně lze vložit text i na propojení s použitím dvou operátorů šipka. Příklad tohoto kódu je ve výpise 2.1. V nastavení generátoru lze zvolit styl rozložení uzlů v diagramu. Pro ilustraci uvádím na obrázku 2.1 diagram vygenerovaný z kódu při implicitním nastavení a na obrázku 2.2 diagram při volbě rozvržení svislý tok.

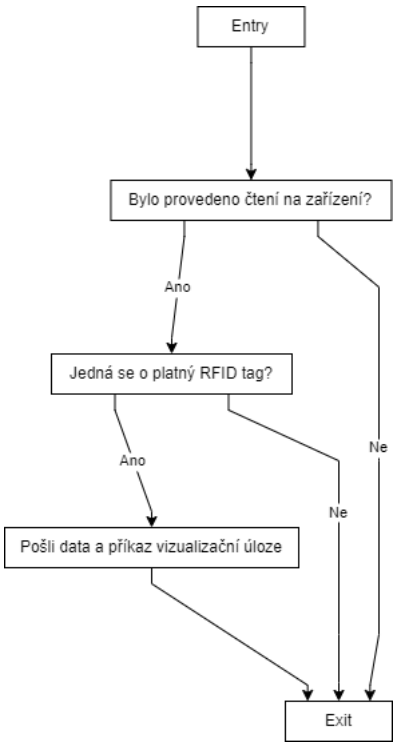
Výpis 2.1: Příklad pseudokódu aplikace diagrams.net popisující diagramy v obrázcích 2.1 a 2.2

```
1 ;Komentáře nelze vložit za užitečný kód na stejný řádek
2 Entry->Bylo provedeno čtení na zařízení?
3 Bylo provedeno čtení na zařízení?->Ano->Jedná se o platný
      RFID tag?
4 Jedná se o platný RFID tag?->Ano->Pošli data a příkaz
     vizualizační úloze
5 Pošli data a příkaz vizualizační úloze->Exit
6 Bylo provedeno čtení na zařízení?->Ne->Exit
7 Jedná se o platný RFID tag?->Ne->Exit
```

Jak lze na příkladu vidět, textový popis je velmi jednoduchý na pochopení, ovšem lze pomocí něj vytvořit pouze jednoduché stromové nebo hierarchické typy diagramů. Dokumentace [17] k aplikaci nezmiňuje možnost definovat tvary uzlů a hran. (Zmiňuje pouze syntaxi, podle které lze vytvořit v jednom grafu najednou více objektů



Obr. 2.1: Diagram vygenerovaný podle pseudokódu ve výpise 2.1 při volbě implicitního nastavení



Obr. 2.2: Diagram vygenerovaný podle pseudokódu ve výpise 2.1 při volbě rozvržení svislého toku

pro diagram tříd bez jejich vzájemného propojení.) Proto je tento typ pseudokódu pro generování UML diagramů nedostačující.

2.2 Tabulková data

Tato možnost reprezentace diagramu je vhodná pro diagramy vizualizující informace uložené v databázových tabulkách. Hodí se tedy například pro hierarchické diagramy nebo diagramy komponent. Obecně je však její použití výhodné pro jakékoliv diagramy s uzly a hranami.

Textový soubor pro generování diagramu se skládá ze dvou částí. První je preambule, jejíž každý řádek začíná symbolem # a poskytuje informace o typech tvarů uzlů, hran, rozložení diagramu a způsob, jak mají být data z tabulky interpretována. Druhá část jsou již samotná data v CSV (anglicky Comma-separated values, česky hodnoty oddělené čárkami) formátu, kde každý uzel diagramu je reprezentován jedním řádkem. [18] V následujících odstavcích uvádím příklady některých těchto příkazů. Z důvodu úspory místa jsou některé části kódu zkráceny.

V preambuli lze na samostatné řádky psát komentáře pomocí dvou symbolů #. Jednotlivé příkazy se píší ve formě `#příkaz: parameter1,parametr2 ...`. Příkazy definující názvy atributů s hodnotami jsou ve formě: `#příkaz: {"Název atributu1" : "hodnota1","název atributu2" : "hodnota2" ... }`.

Příkaz `labels` definuje formátování textu v uzlu diagramu pomocí HTML kódu. Název proměnné, který se má v textu rozvinout, se uvozuje mezi znaky %. Pakliže je použito vícero formátování, je nutné mít v CSV souboru pro každý uzel sloupec, ve kterém je uložen název formátování, který daný uzel používá. Jméno tohoto sloupce se definuje pomocí příkazu `labelname`.

```
# labels: {"Text" : "%comment%<br><em>%variable%</em>",
"NoText" : ""}
# labelname: labeltype
```

Příkaz `styles` definuje styly jednotlivých uzlů v diagramu (Například použitý tvar). Tyto styly lze přečíst v editoru při vyvolání kontextové nabídky nad objektem a zvolení menu `Change style...`. Obdobně jako u formátování se název sloupce s názvy stylů nastavuje pomocí příkazu `stylename`.

```
# styles: {"if" : "<zkráceno>",\
#             "action" : "<Zkráceno>",\
#             "entry" : "ellipse;fillColor=strokeColor;",\
#             "exit" : "<Zkráceno>",\
#             "join" : "<Zkráceno>;"}
# stylename: styletype
```

Příkazy `connect` definují typy propojení mezi uzly. Za účelem operování s vícera propojeními odlišujícími se stylem čáry a textem přidruženého popisku je potřeba použít příkaz opakovaně. Jeho povinné parametry jsou `to` definující název sloupce, ve kterém se nachází unikátní klíč pro daný uzel a `from` definující název sloupce, ve kterém je žádný, jeden nebo více těchto unikatních klíčů korespondující s uzly, se kterými má být uzel v daném řádku propojen. Je možné, aby řádek odkazoval sám na sebe a stejné uzly byly propjeny stejnými typy propojení vícekrát.

```
# connect: {"from": "ref", "to": "id",
    "invert": true,"style": "<zkráceno>"}
# connect: {"from": "refyes", "to": "id",
    "invert": true,"style": "<zkráceno>","label" : "Ano"}
# connect: {"from": "refno", "to": "id",
    "invert": true,"style": "<Zkráceno>","label" : "Ne"}
```

Příkaz `ignore` má v parametrech seznam řádků sloupců, jejichž hodnoty nemají být do diagramu exportovány jako vlastnosti objektu nebo nemají být použity jako názvy proměnné v příkazu `label`.

```
# ignore: id,labeltype,styletype,ref,refyes,refno
```

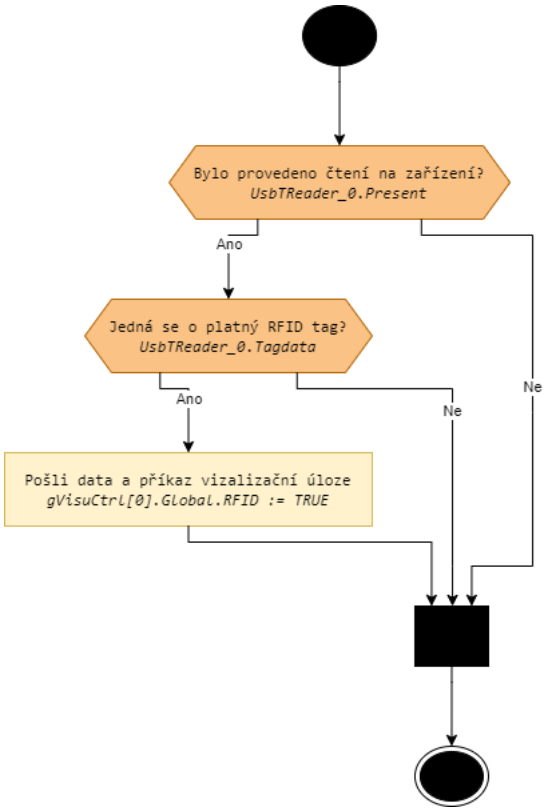
Příkaz `namespace` definuje předponu názvů objektů ve výsledném diagramu. Toto slouží jako prevence konfliktních názvů objektů.

```
# namespace: csvimport-
```

S těmito jednotlivými příkazy, s daty a s dalšími příkazy uvedenými ve výpise 2.2, které souvisí s rozložením uzlů, byl vygenerován diagram na obrázku 2.3.

Výpis 2.2: Příklad CSV souboru generující diagramy na obrázku 2.2

```
1 # nodespacing: 60
2 # layout: verticalflow
3 id,labeltype,styletype,comment,variable,ref,refyes,refno
4 1,NoText,entry,,,,,
5 2,Text,if,"Bylo␣provedeno␣čtení␣na␣zařízení?","
    UsbTReader_0.Present",1,,
6 3,Text,if,"Jedná␣se␣o␣platný␣RFID␣tag?","UsbTReader_0.
    Tagdata",,2,
7 4,Text,action,"Pošli␣data␣a␣příkaz␣vizalizační␣úloze","
    gVisuCtrl[0].Global.RFID␣:=␣TRUE",,3,
8 5,NoText,join,,,4,,"2,3"
9 6,NoText,exit,,,5,,
```



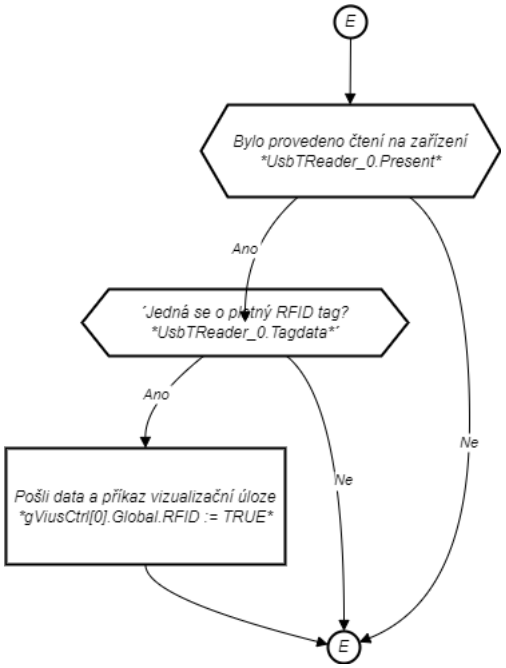
Obr. 2.3: Diagram vygenerovaný z CSV souboru ve výpise 2.2

2.3 Nástroj Mermaid

`Mermaid` je nástroj s otevřeným zdrojovým kódem v jazyce JavaSript sloužící k vytváření diagramů pomocí textového popisu vycházejícího z jazyka Markdown. Umožňuje vytváření nejběžnějších typů diagramů, z nichž některé patří pod UML. V experimentální fázi vývoje je mimo jiné generování časové osy Git repozitáře nebo C4 modelů [19]. `Mermaid` lze vyzkoušet v internetovém editoru `Mermaid Live`, ve kterém se rovnou graf při psaní vykresluje. V rámci služby `Mermaid Chart` lze diagramy ukládat i v cloudu a lze získat `Mermaid` rozšíření pro další textové a prezentační editory jako je Visual Studio Code nebo Microsoft PowerPoint. Verze zdarma umožňuje pouze 5 uložených diagramů. Placené verze pak umožňují pokročilejší typické funkce kolaborativních vývojářských nástrojů jako například verzování. [20] Samotný nástroj s kódem lze však bezplatně používat v HTML dokumentu, kde se dá propojit s ostatními prvky, například s CSS třídami.

Aplikace diagrams.net umí zpracovat pouze syntaxi `Mermaid` bez direktiv pro internetovou aplikaci a bez vazeb na HTML kód, což použití `Mermaid` omezuje. Nelze tak například volit mezi různými způsoby vykreslování hran grafu. Navíc, jak lze vidět na obrázku 2.4, aplikace diagrams.net vykresluje všechny prvky ve výchozím

stylu, i když jsou ve výpise 2.3 styly definované a jednotlivým prvkům přiřazené. Textová pole v uzlech rovněž nelze formátovat podle syntaxe Markdown při vkládání přes aplikaci diagrams.net. Toto neplatí, pokud objekt do diagramu vložíme jako obrázek za cenu ztráty možnosti vygenerované prvky diagramu upravovat.



Obr. 2.4: Diagram vygenerovaný z příkladu ve výpise 2.3 v jazyce `Mermaid`

Každý soubor v jazyce `Mermaid` začíná typem grafu. V rámci jednoho diagramu může být pouze jeden. Následuje parametr určující směr toku objektů v diagramu. V tomto případě `TD` (anglicky Top Down) značí směr od shora dolů. Pro každý typ diagramu platí jiná syntaxe a proto vysvětlím pouze syntaxi vztaženou k vývojovému diagramu.

Příkaz `classDef` definuje název třídy a její styl. Uzel se definuje svým názvem, tvarem a textem, který má uzel zobrazovat. Tvar je definován typem znaků, které text uvozují. Například rovnoběžník vykreslují dvojité složené závorky a obdélník hranaté. Text může být formátovaný podle jazyka Markdown, pokud je textový řetězec uvozen znaky `"‘ ‘ "`. Zalomení řádku lze pak vynutit novým řádkem v kódu. Příslušnost uzlu ke třídě stylu se přiřazuje pomocí operátoru `:::`.

Spoje mezi uzly se v tomto příkladu definují pomocí operátoru `-->` nebo `-- Popisek -->`, pokud mají obsahovat popisek. Existují však i jiné styly propojení, které mají své příslušné operátory, obdobně jako tvary uzlu se definují uvozujícími znaky. [21]

Výpis 2.3: Příklad kódu v jazyce `Mermaid` generující diagram na obrázku 2.4

```
flowchart TD
    classDef if fill:#FAC385,stroke:#b46504,stroke-width
       :1px;
    classDef act fill:#fff2cc,stroke:#d6b656,stroke-width
       :1px;
    classDef bdot fill:#000000,stroke:#000000,stroke-
       width:1px;
    Entry((E)):::bdot --> A{{"`Bylo␣provedeno␣čtení␣na␣za
       řízení
␣␣␣␣*UsbTReader_0.Present*`"}}:::if -- Ano --> B{{" Jedn
   á␣se␣o␣platný␣RFID␣tag?
␣␣␣␣*UsbTReader_0.Tagdata*  "}}:::if -- Ano --> C["`Pošli
   ␣data␣a␣příkaz␣vizualizační␣úloze
␣␣␣␣*gViusCtrl[0].Global.RFID␣:=␣TRUE*`"]:::act --> Exit
   ((E)):::bdot
     A -- Ne --> Exit
     B -- Ne --> Exit %%koment
```

2.4 Nástroj PlantUML

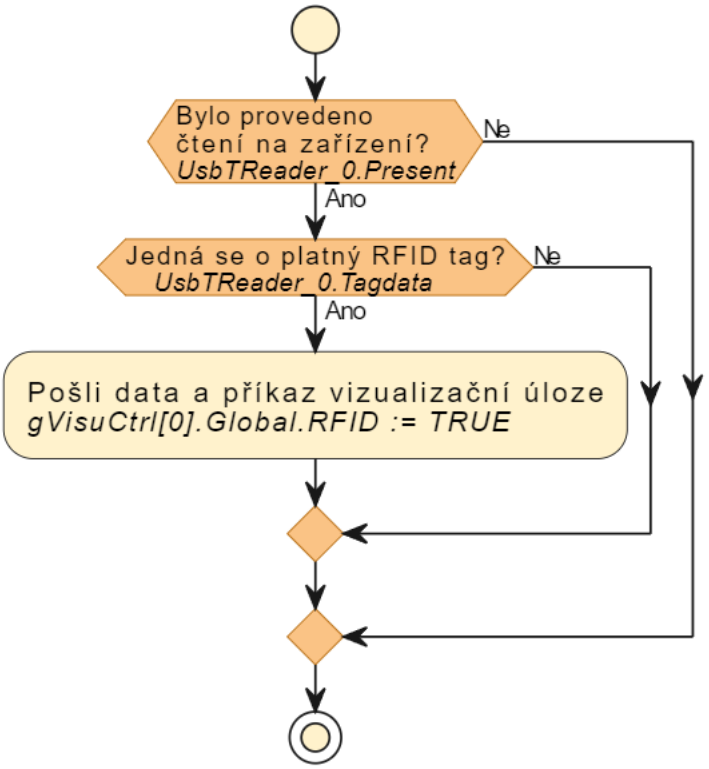
`PlantUML` je svobodný nástroj pro tvorbu diagramů vyvíjený od roku 2009 a napsaný v jazyce Java. Umožňuje vykreslení nejpoužívanějších UML diagramů a dalších typů používaných v softwarovém inženýrství. K tomu používá vlastní jazyk, ovšem dokáže v kódu operovat i se syntaxí jazyků AsciiMath, Creole, DOT a LaTeX. Umožňuje vykreslovat do formátů PNG, SVG, LaTeX a ASCII art. Pro tuto činnost používá sadu nástrojů `Graphviz`. `PlantUML` lze spouštět jako nástroj na lokálním zařízení, nebo ho lze integrovat do vývojářských prostředí. Rovněž ho lze vkládat do HTML dokumentů, čehož využívá také například populární dokumentační nástroj `Doxygen`. Na vyzkoušení nástroje bez instalace lze použít přímo stránky projektu. [22]

Nástroj diagrams.net umí operovat s `PlantUML` pouze v aplikaci přes webový prohlížeč, protože ke generování používá vlastní servery. Poskytuje však obraz pro platformu Docker, která umožňuje veškeré funkce aplikace provozovat bez vazeb na servery diagrams.net. [23] Příklad `PlantUML` kódu pro diagram na obrázku 2.5 je ve výpise 2.4. Diagram lze vkládat pouze jako obrázek vektorový nebo rastrový. Jednotlivé elementy tudíž nelze nadále upravovat.

Syntaxe `PlantUML` vychází z tvarů používaných ve specifikaci UML. `PlantUML` dokument se uvozuje mezi příkazy `@startuml` a `@stopuml`. V hlavičce lze definovat

Výpis 2.4: Příklad kódu v jazyce `PlantUML` generující diagram na obrázku 2.5

```
@startuml
<style>
activityDiagram {
  BackgroundColor #fff2cc
  BorderColor #d6b656
  FontColor #0
  FontName arial
  diamond {
    BackgroundColor #FAC385
    LineColor #b46504
    FontColor black
    FontName arial
  }
}
document {
   BackgroundColor transparent
}
</style>
start
if (Bylo provedeno čtení na zařízení?
//UsbTReader_0.Present//) then (Ano)
  if (Jedná se o platný RFID tag?
  //UsbTReader_0.Tagdata//) then (Ano)
    :Pošli data a příkaz vizualizační úloze
    //gVisuCtrl[0].Global.RFID := TRUE//;
  else (Ne)
  endif
else (Ne)
endif
stop
@enduml
```



Obr. 2.5: Diagram vygenerovaný z příkladu ve výpise 2.3 v jazyce `PlantUML`

styly diagramu a použitých tvarů pomocí bohaté škály direktiv a značek. Obsah diagramu se poté píše velmi podobně jako program ve vyšším programovacím jazyce. V diagramu se objevují jednotlivé objekty v sekvenčním pořadí tak, jak jsou napsány a to včetně větvení programu. Text lze formátovat kdekoliv v diagramu.

2.5 Srovnání jazyků pro reprezentaci diagramů

V této kapitole jsem představil jednotlivé textové reprezentace diagramů, které podporuje nástroj diagrams.net. Přehledné srovnání uvádím v tabulce 2.1

Jak již bylo zmíněno, prostý textový popis je pro účely tvorby UML diagramů nedostačující. Popis v tabulkovém CSV formátu postačuje pro tvorbu jakéhokoliv typu diagramů s uzly a hranami. Lze nastavovat způsob rozložení a styly objektů, nikoliv ovšem proporce tvarů. Vygenerovaný diagram po stránce vzhledu vypadá obstojně a lze ho případně nadále upravit. Text může obsahovat hypertextové odkazy, které jsou ve vygenerovaném grafu použitelné. Tabulková reprezentace diagramu je ovšem obtížně lidsky čitelná.

Jazyk `Mermaid` je relativně nový a používá šablony pro různé typy diagramů,

Tab. 2.1: Přehled v kapitole představených jazyků pro tvorbu diagramů použitelných v aplikaci diagrams.net

Název	Přehlednost	Možnosti úprav v textovém popisu	Možnosti úprav vygenerovaného diagramu
Textový popis	Dobrá	Pouze jednoduché diagramy	Lze plně upravovat
Tabulková data	Dobrá	Nelze volit velikost tvarů, jinak nejvíce konfigurovatelný	Lze plně upravovat
`Mermaid`	Nevyhovující	Omezeně konfigurovatelný	Jako objekt ano, jako obrázek ne
`PlantUML`	Nejlepší	Velmi konfigurovatelný	Pouze obrázek - nelze upravovat

z nichž některé jsou stále ve vývoji. Kvůli nedostupné možnosti vkládat konfigurační direktivy do aplikace diagrams.net není možné měnit například parametry vykreslování čár. Při vložení diagramu jako tvarů v aplikaci diagrams.net se také nevykreslí styly obrazců a formátování textu, vkládání hypertextových odkazů ovšem funguje. Vygenerovaný diagram po vizuální stránce obsahuje chyby a vyžadoval by další manuální úpravy. Diagram vložený jako obrázek vypadá dobře, obsahuje styly i formátování, ovšem nelze ho dále upravovat a také neobsahuje hypertextové odkazy.

`PlantUML` ze všech zmíněných jazyků podporuje největší výběr typů diagramů. V běžné desktopové verzi aplikace diagrams.net není dostupný a vygenerované diagramy lze vkládat pouze jako obrázky a ani v případě formátu SVG nelze vkládat hypertextové odkazy. Diagram tudíž není dále editovatelný, ovšem po vizuální stránce je ze všech variant nejpřehlednější.

Pro popis programu jako algoritmu se z UML specifikace [24] nejlépe dá použít diagram aktivit, který je velmi podobný rovněž používanému vývojovému diagramu mimo specifikaci UML. Kromě textového popisu aplikace diagrams.net, lze pro vytvoření diagramu aktivit nebo vývojového diagramu použít jakýkoliv ze zmíněných nástrojů.

Pro reprezentaci diagramů v práci volím nástroj `PlantUML` z důvodu jeho snadno čitelné syntaxe a přehlednosti vygenerovaných diagramů, která je shodná s UML specifikací. Jeho současné použití v dlouhodobě udržovaných projektech, jako je dokumentační nástroj Doxygen z něj činí používaný a udržovaný nástroj. Rovněž jej volím z důvodu možnosti případných budoucích rozšíření mého dokumentačního nástroje o další typy diagramů, které `PlantUML` již v současnosti podporuje.

3 Jazyk Structured Text v PLC systémech společnosti B&R Industrial Automation

V této kapitole je představeno vývojářské prostředí Automation Studio od společnosti B&R Industial Automation a programátorský jazyk Structured Text, v němž psané programy budou automaticky dokumentovány mým překladačem.

3.1 Společnost B&R Industrial Automation

Společnost B&R Industrial Automation (dále jen B&R) byla založena v roce 1979 v Rakouském městě Eggelsberg, kde dodnes sídlí její centrála. V současnosti působí po celém světě včetně Česka, kde je její hlavní pobočka sídlí v Brně pod názvem B+R Automatizace. Od roku 2017 spadá společnost pod skupinu firmy ABB Robotics & Discrete Automation. Zaměřuje na návrh a řešení systému pro automatizaci strojů a továren. Nabízí širokou nabídku systémů od kontrolérů, CNC, PLC, HMI, robotiky až po safety řešení, které lze všechny programovat z jejich vlastního vývojového prostředí Automation Studio. [25]

3.2 Vývojové prostředí Automation Studio

Automation studio integruje do jednotného vývojového prostředí nástroje pro tvorbu, konfiguraci a programování automatizačních systémů s produkty od společnosti B&R. Lze tak v jednom prostředí programovat, vytvářet vizualizaci, konfigurovat a simulovat systém. V rámci jednoho projektu lze programovat stejný zdrojový kód pro více hardwarových konfigurací. Automation Studio je vydáváno pod různými typy licencí. [26] V práci používám verzi 4.12 se studentskou licencí.

Projekt v Automation Studiu je reprezentován XML souborem s metadaty. Mimo něj jsou pak soubory ve složkách, které svým uspořádáním odpovídají jednotlivým objektům v hierarchii projektu. Soubory nejsou nijak zapouzdřeny, a tak je možné k nim přistupovat i přes programy třetích stran.

Samotné programy pro PLC lze psát v různých jazycích a v rámci jednoho projektu jich používat více. Mezi tyto jazyky patří všechny obsažené v mezinárodní normě IEC EN 61131-3, jmenovitě grafické jazyky Ladder Diagram (LD), Function Block Diagram (FBD), Sequential Function Diagram (SFD) a textové jazyky Instruction List (IL) a Structured Text (ST). Navíc lze používat textový jazyk Automation Basic od B&R podobný jazyku Structured Text s přidanými funkcionalitami vysokoúrovňových programovacích jazyků jako strukturami a ukazately. Dalším přidaným grafickým jazykem je Continuos Function Chart (CFC) podobající se zároveň

Ladder Diagramu a Function Block Diagramu. Volitelně lze využít také standardně používané jazyky `ANSI C++` a `ANSI C`. [27]

3.3 Jazyk Structured Text

V této podkapitole jsou představeny konstrukce jazyka Structured Text uvedené v specializované tréninkové příručce společnosti B&R. [2]

Každá programová jednotka obsahuje soubory se zdrojovým kódem a tabulky s lokálními datovými typy a lokálními proměnnými. Samotný program obsahuje 3 části, které můžou být v oddělených souborech (Pro každý tak lze použít různý programovací jazyk.) nebo lze mít všechny v jednom souboru. Část uvozená mezi konstrukce `PROGRAM _INIT` a `END_PROGRAM` se vykonává při prvním běhu PLC při přechodu do stavu `RUN`. V bloku `PROGRAM _CYCLIC` a `END_PROGRAM` je cyklický program, který se vykonává při běhu PLC. V konstrukci `PROGRAM _EXIT` a `END_PROGRAM` se vykoná program při posledním běhu PLC při přechodu ze stavu `RUN` do stavu `STOP`. Tento program se vykoná například při nahrání nové verze programu do PLC.

Největší výhodou používání jazyka Structured Text nad ostatními grafickými jazyky je možnost snadného vytváření výrazů tvořených vícero operandy, které propojují operátory. Operand může být proměnná, konstantnebo funkční volání. Každý výraz je uzavřen středníkem.

```
SIN(a) + cos(b);
```

Výrazy s více operátory jsou vyhodnoceny podle priority operátorů. Operátory se stejnou prioritou jsou vyhodnocovány zleva doprava tak, jak jsou napsány ve výrazu. Výčet operátorů seřazených od nejvyšší po nejnižší prioritu je uveden v tabulce 3.1.

Přiřazení se provede pomocí cílové proměnné na levé straně, operátoru `:=` a výrazem na pravé straně. Structured Text umožňuje přístup k jednotlivým bitům proměnné základních datových typů pomocí operátoru tečky. Zároveň umožňuje vytváření polí datových proměnných a složených datových typů. Názvy proměnných se rozlišují i podle velikosti jednotlivých písmen a nesmí být pojmenované jako klíčová slova nebo standardní knihovní funkce a funkční bloky.

```
Result := Status.7;
```

Komentáře ve zdrojovém kódu uvozují znaky `(*` a `*)` a můžou být v bloku na vícero řádcích. Oproti normě IEC 61131-3 operuje Automation Studio také s jednořádkovými komentáři za sekvencí znaků `//`.

Větvení programu se provádí pomocí běžných konstrukcí známých z vyšších programovacích jazyků. Následují ukázky přesných syntaxí těchto konstrukcí.

Konstrukce `IF` slouží k větvení programu na základě podmínky.

Tab. 3.1: Přehled operandů v jazyce Structured Text seřazený dle priority od nejvyšší po nejnižší

Operátor	Syntaxe
Závorky	`()`
Funkční volání	`Funkce(Argument)`
Exponent	`**`
Negace	`NOT`
Násobení, dělení, modulo	`*`,`/`,`MOD`,
Sčítání, odečítání	`+`,`-`,
Porovnání	`<`,`>`,`<=`,`=>`
Rovnost, Nerovnost	`=`,`<>`
Logický součin	`AND`
Logický exkluzivní součet	`XOR`
Logický součet	`OR`

```
IF výraz1 THEN
    příkaz1;
ELSIF výraz1 THEN
    příkaz2;
ELSE
    příkazN;
END_IF;
```

Konstrukce `CASE` slouží k větvení programu na základě srovnání hodnot s proměnnou celočíselného typu, kterou lze nadefinovat jako datový typ výčtu. Narozdíl od jazyka `C` je během cyklu programu provedena pouze jedna z větví konstrukce `CASE`. Velmi často se používá k implementaci stavového automatu.

```
CASE Celočíselná_Proměnná OF
    konstanta1,konstanta2:
        příkaz1;
    konstanta3:
        příkaz2;
ELSE
    příkazN;
END_CASE;
```

Konstrukce `WHILE` umožňuje provádět opakující se části kódu. Samotné programy PLC se provádějí cyklicky v určité časové cykly. Pokud provádění cyklu by trvalo

dlouho, může dojít k nedokončení programu. Proto musí být vždy definována ukončovací podmínka. Provádění jakéhokoliv typu cyklu lze ukončit příkazem `EXIT;`.

```
WHILE podmínka DO
    příkazy;
END_WHILE;
```

Konstrukce `REPEAT` je typ cyklu, kdy ukončovací podmínka se kontroluje až po jejím provedení. Provede se taky vždy minimálně jednou.

```
REPEAT
    příkazy;
    UNTIL podmínka
END_REPEAT;
```

`FOR` cyklus se používá k provedení omezeného počtu opakování programu. Používá k tomu indexovou proměnou, která musí být stejného datového typu jako počáteční a koncová hodnota. Klíčové slovo `BY` a hodnota, o kterou se inkrementuje je nepovinná.

```
FOR index:=PocatecniHodnota TO KoncovaHodnota BY Krok DO
    příkazy;
END_FOR;
```

Funkce v programu jsou podobné podprogramům, které vrací hodnotu při svém volání. Lze je volat v rámci výrazu s parametry (také nazvané argumenty) oddělenými čárkami.

```
Result := brwcscat(pDestination,pSource);
```

Funkční bloky se od funkcí odlišují více výstupními proměnnými a možností provádění během více jak jednoho cyklu programu. Vyžadují deklaraci instance proměnné typu korespondujícího funkčnímu bloku. Instanci funkčního bloku, předání a vyčítání parametrů lze provést dvěma způsoby.

V této variantě jsou všechny parametry předány při volání instance.

```
Timer1(IN := diButton, PT := T#42s);
doBell := Timer1.Q;  //vyčtení parametrů
```

V této variantě jsou jednotlivé parametry přiřazeny před jejím voláním.

```
Timer1.IN := diButton;
Timer1.PT := T#42s;
Timer1();    //volání funkčního bloku
doBell := Timer1.Q; //vyčtení parametrů
```

Structured Text od B&R je rozšířen mimo normu IEC 61131-3 o dynamickou alokaci proměnných za běhu aplikaci pomocí ukazatelů a referencí. Ukazatel lze deklarovat pomocí klíčového slova `REFERENCE`.

```
Pointer : REFERENCE TO INT;
```

Adresa, na kterou ukazatel bude odkazovat se přířadí pomocí klíčového slova `ACCESS`.

```
Ukazatel ACCESS ADR(proměnná);
//Ukazatel nyní ukazuje na adresu proměnné.
```

Další funkcionalitou mimo normu IEC 61131-3 je možnost používaní direktiv preprocesoru, které se velmi podobají preprocesoru `ANSII C`. Používají se pro podmíněnou kompilací programů nebo jinačí ovlivnění průběhu kompilace. Popis a kompletní seznam všech dostupných příkazů lze nalézt v nápovědě Automation Studia.

4 Návrh analyzátoru

V této kapitole jsou představeny základní pojmy používané v teorii výstavby překladačů a nástroje pro tvorbu analyzátorů. Dále je zde navržena syntaxe analyzátoru, kterou bude implementovaný nástroj přijímat.

4.1 Struktura překladačů

Informace uvedené v této podkapitole byly převzaty z knihy *Compilers, principles, techniques, and tools* [28].

Překladač je program, který převádí zdrojový kód napsaný ve zdrojovém programovacím jazyce do ekvivalentního kódu v cílovém jazyce. Existuje několik druhů překladačů. Nejčastěji jsou používané kompilátory překládající celý zdrojový kód na strojový pro přímé provádění procesorem a interprety provádějící překlad za běhu programu. Tato práce se bude dále zaměřovat na kompilátory.

Stadndardní kompilátor se skládá ze dvou hlavních částí a to na analýzu zdrojového kódu, kterou provádí analyzátor, a syntézu cílového programu. Analyzátor má za úkol zpracovat zdrojový kód a vytvořit dle jeho struktury abstraktní reprezentaci pro další použití. Nad kódem jsou provedeny 3 typy analýz, které můžou proběhnout postupně nebo v jednom průchodu zároveň. Syntéza se skládá rovněž ze tří kroků.

První krok se nazývá **Lexikální analýza** nebo také skenování. (Tuto část provádí nástroj v angličitně nazvaný jako lexer, tokenizer nebo scanner). Vstupem lexikálního analyzátoru je řetězec znaků tvořící zdrojový kód, který je rozdělen na sekvence nazývané lexémy. Pro každý z něj vytváří takzvané tokeny skládající se z názvu a hodnoty a ty ukládá do tabulky symbolů. Běžně se také zabývá zjednodušením vstupního kódu, kdy odstraňuje zbytečné bílé znaky a komentáře.

K řešení lexikální analýzy se používají nejčastěji dva způsoby reprezentací. Prvním jsou regulární výrazy, které jsou formálním nástrojem pro popis sady řetězců tvořící konkrétní vzor. Používají operace konkatenace, spojení a iterace. Druhým způsobem reprezentace jsou konečné automaty, které přijímají znak a na základě pravidel gramatiky jazyka provádějí přechody mezi svými stavy. Pokud stavový automat úspěšně skončí v definovaném stavu, je lexém přijat. V opačném případě nastává chyba.

Druhý krok se nazývá **Syntaktická analýza**. (Tuto činnost provádí nástroj v angličtině zvaný jako parser). Vstupem této analýzy je proud tokenů a výstupem je přechodná struktura znázorňující vztahy mezi lexémy. Nejčastěji se používá derivační strom (anglicky parse tree nebo také česky abstraktní syntaktický strom). Konstruuje se podle gramatiky s konečným počtem pravidel, podle které se kontroluje správnost syntaxe. Existují dva způsob konstrukce stromu a to zdola nahoru a

shora dolů. Vyšší počet programovacích jazyků dokáže obsáhnout způsob zdola nahoru, kdy jednotlivé listy se dle pravidel postupně redukují na kořen. Respektive dle opačného přístupu tato analýza hledá takový strom, který se rozvine z počátečního symbolu (kořene stromu) přes neterminální symboly popisující struktury v jazyce na terminální symboly odpovídající analyzovanému textu. Vztahy mezi symboly jsou popsány pomocí derivačních (nebo také přepisovacích) pravidel.

Třetí krok se nazývá **Sémantická analýza** zaměřující se na význam a smysl vytvořeného abstraktního syntaktického stromu. Kontroluje například správnost datových typů proměnných a provede případné implicitní přetypování. Kontroluje také zda názvy proměnných a funkcí jsou unikátní, zda jsou správně deklarovány a použity v příslušejícím bloku kódu, případně zde dochází k jejich přetěžování.

Čtvrtý krok překladu se nazývá **Generování vnitřního kódu**. Tento krok transformuje anotovaný syntaktický strom do obecného jazyka nebo jazyků. Podoba jazyka se volí s ohledem na náročnost vytvoření ze syntatktického stromu, gramatiku cílového jazyka a algoritmus optimalizace v následujícím kroku. Obecnost jazyka rovněž umožňuje případná budoucí rozšíření překladače o podporu dalších jazyků.

Pátý,volitelný krok překladu se nazývá **Optimalizace**. Jeho cílem je strukturu vnitřního kódu optimalizovat za účelem kratšího zápisu, rychlejšího provádění, menší paměťové náročnosti nebo jejich kombinací. Výsledkem optimalizace je vylepšení původního kódu při zachování stejného významu. Optimalizovat lze například odstraněním mrtvého kódu, který se při běhu programu nikdy neprovede a výpočtů s konstantami.

Šestý, poslední krok překladu je **Generování cílového kódu**. Vnitřní kód je přeložen do cílového jazyka, nejčastěji do strojových instrukcí. Zde je hlavním kritériem, aby výsledný kód funkčně odpovídal zdrojovému a byl ve správné syntaxi cílového jazyka.

Běžné kompilátory slouží k překladu vyššího programovacího jazyka do jazyka s nižší úrovní abstrakce, nejčastěji do strojového kódu, případně do jazyka Assembler. Kompilátor v této práci má překládat do jazyka `PlantUML`, což je jazyk na vyšší úrovni abstrakce, tudíž se terminologicky správně jedná o transpilátor (anglicky transpiler). Takový kompilátor se od běžných kompilátorů odlišuje hlavně v postupech uplatňovaných v části syntézy programu. Naopak analýza může probíhat stejně jako u běžných kompilátorů.

4.2 Nástroje pro tvorbu překladačů

V oblasti vývoje překladačů není vždy nezbytné konstruovat překladač od samotného počátku. Programátoři mohou využít různé nástroje, které usnadňují proces

sestavení jednotlivých fází překladače. Například generátory syntaktických analyzátorů poskytují možnost vytvářet zdrojový kód syntaktického analyzátoru na základě formální specifikace jazyka. Tento vygenerovaný kód lze následně přeložit, čímž vznikne spustitelný syntaktický analyzátor nebo překladač. V rámci této sekce budou popsány nástroje ulehčující konstrukci analyzátoru nebo celého překladače.

4.2.1 Generátor analyzátorů ANTLR

Nástroj `ANTLR` (akronym ANother Tool for Language Recognition) je nástroj pro tvorbu analyzátorů lexikálních, syntaktických a sémantických napsaný v jazyce `Java`. Současná verze 4.13.1 umožňuje generování analyzátoru v jazycích `C#`, `C++`, `JavaScript`, `Go`, `Dart`, `PHP` nebo `Swift`. [32] Zápis gramatiky je podobný jako v nástroji `Bison` při definování syntaktického analyzátoru. Generátor lexikálního analyzátoru je v nástroji `ANTLR` zahrnut a psát se nemusí. Oproti nástroji `Bison` je `ANTLR` náročnější na paměť i výpočetní čas. [33]

4.2.2 Generátory lexikálních analyzátorů Lex a Flex

`Lex` je nástroj pro tvorbu lexikálního analyzátoru pomocí regulárních výrazů ve stejnojmenném jazyce `Lex`. Zdrojový kód je pak přeložen do jazyka C. Byl vytvořen již v roce 1975 pod proprietární licencí. [29] Populární svobodná alternativa s otevřeným kódem nazvaná `Flex`, byla publikována v roce 1987 [30] a její součástí je i balíček `Flex++` umožňující generovat analyzátor v jazyce `C++`.

4.2.3 Generátory syntaktických analyzátorů Yacc a Bison

`Yacc` (zkráceně Yet Another Compiler-Compiler) je nástroj pro výstavbu syntaktického analyzátoru podle popisu formální gramatiky. Byl publikován rovněž v roce 1975 pod `MIT` licencí. Zpětně kompatibilní svobodná alternativa s otevřeným kódem se jmenuje `Bison` a byla publíkována v roce 1985. [30]. Oproti nástroji `Yacc` obsahuje několi rozšíření, jako například lokalizaci ve zdrojovém kódu a přeložitelné chybové zprávy. Nejčastěji je tento nástroj používán s lexikálním analyzátorem `Flex`.

4.3 Návrh syntaxe zdrojových souborů příjmané implementovaným analyzátorem

Jak již bylo zmíněno v podkapitole 2.5, algoritmus PLC programu bude popsán pomocí diagramu aktivit. Pro tento typ diagramu jsou ve struktuře programu relevantní pouze větvení, cykly, příkazy a komentáře. Proto se implementovaný pře-

kladač nebude zabývat vyhodnocením korektnosti jednotlivých výrazů. K tomuto účelu bude tedy postačovat analyzátor vygenerovaný programy `Flex` a `Bison`, které jsou dobře zdokumentované a generují zdrojové soubory v jazyce `C`, v kterém umím programovat.

Tyto nástroje lze jednoduše nainstalovat na operačním systému Linux přes balíkovací systém `apt` pomocí příkazu `apt-get install flex bison`. Pro operační systém Windows je potřeba najít vhodné adaptované verze těchto nástrojů. Pro tyto účely existuje vícero softwarových balíčků jako například populární, ale již neudržovaný `GnuWin`. Z důvodu jednodušší instalace jsem překladač vyvíjel v systému Linux.

Proto pro implementovaný překladač bude dostačující, když ze zdrojového kódu extrahuje bloky větvení (resp. cyklů), které v diagramové reprezentaci budou doplněné o výraz uvedený v podmínce (resp. v ukončovací podmínce). Volitelně pak bude možné k podmínkám a cyklům přidat komentáře. Jednotlivé příkazy zdrojového kódu se v diagramu nevykreslí, pokud nebudou doprovázeny komentářem pro překladač.

Syntaxí komentářů pro překladač jsem se inspiroval u dokumentačního nástroje Doxygen. Konec blokového komentáře se musí nacházet přesně o řádek výše než je daný blok programu nebo příkaz a musí být uvozen ve znacích `(**` a `**)`. Protože jazyk Structured Text v Automation Studio umožňuje také jednořádkové komentáře, lze také komentář pro překladač umístit na stejný řádek za klíčové slovo `THEN`, resp. `OF` nebo `DO` a musí mu předcházet sekvence `//*`.

Ukázka možných umístění komentářů je uvedena ve výpise 4.1. Tento kód odpovídá diagramu na obrázku 2.5 a kódu v jazyce `PlantUML` ve výpise 2.4. Komentář na 3. a 4. řádku se vztahuje k větvení programu na 5. řádku. V případě dalšího větvení je komentář na stejném řádku. Obdobně je tomu tak u příkazu na 7. řádku. Všimněme si, že příkaz na 8. řádku v diagramu není zobrazen, protože se k němu nevztahuje žádný komentář akceptovaný překladačem.

Výpis 4.1: Ukázka možných umístění komentářů

```
1 PROGRAM _CYCLIC
2
3 (** Bylo provedeno
4 čtení na zařízení? **)
5 IF UsbTReader_0.Present THEN
6     IF UsbTReader_0.Tagdata THEN    //*Jedná se o platný RFID
          tag?
7         gVisuCtrl[0].Global.RFID := TRUE;   //*Pošli data
              vizualizační úloze
```

```
8           gVisuCtrl[0].Global.LogByRFID := TRUE; //Pošli data
                do záznamu událostí
9       END_IF;
10  END_IF;
11  END_PROGRAM
```

4.4 Implementace lexikálního analyzátoru v nástroji Flex

Jako první krok ve výstavbě překladače je logické začít od lexikálního analyzátoru. Proto je potřeba při implementaci lexikálních pravidel nejprve ověřit, že tokeny ve zdrojovém textu jsou správně rozpoznávány.

4.4.1 Struktura zdrojového souboru pro nástroj Flex

Zdrojový soubor pro nástroj `Lex` má příponu `.l` a skládá se ze tří částí oddělených znaky `%%`. V kterékoliv části lze vložit bloky kódu v jazyce `C` mezi znaky `%{...}%`, které se překopírují do výgenerovaného zdrojového kódu lexikálního analyzátoru. Tohoto lze využít například pro zahrnutí potřebných hlavičkových souborů pro akce prováděné v druhé části. [34]

První část obsahuje definice ve formátu odsazeného páru na jednom řádku `Název Definice`, které umožňují v kódu programu `Flex` používat zástupné symboly podobně, jako lze používáat makra v jazyce `C`. Případně lze zde nastavit konfigurace nástroje `Flex`, jako například `caseless`, která umožňuje definovat pravidla bez rozlišování velikosti písmen.

Výpis 4.2: Příklad definiční části programu v jazyce `Flex` a příklad vloženého bloku

```
%option case    less
%{
        #include <stdio.h>
        #include "lexer.h"
%}
```

Druhá část obsahuje pravidla v páru `Vzor Akce`, kde vzor odpovídá regulárnímu výrazu a akce bloku programu mezi znaky `{...}`, který se má provést při nalezení daného výrazu v překládaném textu. Tato pravidla jsou seřazena dle priority od nejvyššího po nejnižší. Při vyhodnocení porovnávání zdrojového textu s regulárním

výrazem se provede akce, jejíž vzor zahrnuje nejvíce znaků v porovnávaném výrazu. V případě, že vícero vzorů naráz odpovídá zdrojovému textu a všechny vzory mají stejnou délku, provede se pouze akce pravidla s nejvyšší prioritou. Pokud požadované akci náleží vícero po sobě následujících vzorů, lze do části akce vložit znak `|`, který odpovídá akci definované v následujícím pravidle. Pokud analyzovaný text neodpovídá ani jednomu pravidlu, je zkopírován do výstupní proměnné analyzátoru `yyout`. Obdobně lze i definovat prázdnou akci, kdy při nalezení daného výrazu není nic provedeno.

Výpis 4.3: Příklad části s pravidly analyzátoru v jazyce `Flex`

```
\(\*\* {return COMMENT_START;}
\*\*\) {return COMMENT_END;}
\/\/\* {return SINGLE_LINE_COMMENT_START;}
then    {return THEN;}
if      {return IF;}
end_if; {return END_IF;}
\n|\r\n {return NEWLINE;}
```

Třetí část obsahuje uživatelský kód, který je do vygenerovaného programu zkopírován pod vygenerovaný překladač. Zde je tedy možné operovat s vnitřními proměnnými analyzátoru a je zde potřeba nadefinovat funkci `yywrap`, která je vyvolána při dosažení ukončovacího znaku `EOF` v analyzovaném textu. Pokud vrací nenulovou hodnotu, analyzátor ukončuje svou činnost.

4.4.2 Funkce a proměnné

V definiční části je tedy nejprve nadefinován výčet jednotlivých typů tokenů, jelikož samotná lexikální analýza se provádí voláním funkce `yylex`, jejíž návratová hodnota je celé číslo odpovídající typu rozpoznaného tokenu. Další zpracování tokenu se zpravidla provádí mimo funkci `yylex`. Navrátová hodnota musí být kladná, jelikož nekladné hodnoty signalizují, že funkce dosáhla konce čteného souboru. Pokud token reprezenentuje konkrétní hodnotu nebo jinou reprezentaci, má být tato hodnota uložena do externí proměnné `yylval`. Samozřejmě lze použít i vlastní nadefinované globální proměnné. Text odpovídající právě zpracovanému tokenu je uložen v externí proměnné `yytext`. Délka tohoto řetězce je pak uložena v proměnné `yyleng`. Při specifikování nastavení `%option yylineno` v definiční části, je dostupná stejnojmenná proměnná s číslem právě zpracovávaného řádku zdrojového textu.

Nástroj `Flex` poskytuje také další užitečné funkce, které jsem při vytváření překladače využil. Jmenovitě se jedná o funkci `yymore()`, která při následující shodě

při porovnávání nepřepíše řetězec v proměnné `yytext`, jak tomu běžně je, ale připojí ho za něj.

Opačně pak funguje funkce `yyless(n)`, která z proměnné `yytext` vrátí všechny znaky, kromě `n` prvních zpět na vstup, aby bylo možné je znovu porovnat. Obdobně funguje funkce `unput(c)`, která na vstup vloží konkrétní znak `c`. Tohoto lze využít v momentě, kdy ukončení konkrétního typu tokenu je definováno jiným tokenem, který ovšem také sám o sobě nese informaci, která je potřeba analyzovat.

Naopak funkce `int input()` vrací první nepřečtený znak ze vstupu. Tohoto je využito například pro přeskočení běžných komentářů, které pro nástroj nemají význam.

Během prvotního testování jsem rovněž využil makro `ECHO`, které zkopíruje současný `yytext` do výstupu analyzátoru. [36]

4.4.3 Použití nástroje Flex

Zavoláním příkazu `flex` je vygenerován soubor `lex.yy.c`, který lze pak zkompilovat například překladačem GCC.

Pro první testování nástroje `flex` jsem implenementoval analyzátor, který byl schopný správně rozpoznat tokeny zdrojového textu ve výpise 4.1. Nejprve jsem implementoval funkce pro operaci se soubory, poté funkci `processFile`, která sestává ze smyčky, ve které se volá funkce `yylex`. Funkčnost analyzátoru byla ověřena vypsáním hodnot v `yytext` a typů nalezených tokenů v `yylval` do výstupního souboru.

Po ověření správnosti detekovaných tokenů jsem již pokračoval zkušební implementací syntaktického analyzátoru popsané v podkapitole 4.5 v nástroji `Bison`.

4.4.4 Stavy lexikálního analyzátoru

Při implementaci a ověřování syntaktického analyzátoru jsem nejprve narazil na problém se zpracováním komentářů, kdy bylo nutné, aby lexikální analyzátor nedetekoval klíčová slova v nich napsaná. Nástroj `Flex` je za tímto účelem vybaven možností podmíněného vyhodnocování shod textu se vzory na základě stavu, ve kterém se současně nachází. [37]

Stav lze vytvořit v definiční části přidáním jednoho z těchto řádků, kde `x` označuje název exkluzivního stavu a `s` inkluzivního:

```
%x comment_scope
%s ignored_scope
```

V části s pravidly je podmíněný vzor uvozen názvy příslušejících stavů oddělených čárkami ve špičatých závorkách. V případě, že stavu připadá vícero podmíněných

vzorů současně, lze za špičaté závorky vložit celý blok pravidel a akcí ve složených závorkách. Stavy lze i vnořovat.

Výpis 4.4: Příklad zápisu podmíněných pravidel

```
<comment_scope,content_scope>vzor {akce;}
<ignored_scope>{
    \*\)     {BEGIN(INITIAL);}
    <<EOF>> {fprintf(stderr,"Unclosed␣comment\n");
             return *yytext;}
    .        ;
    }
```

Implicitně se lexikální analyzátor nachází ve stavu `INITIAL`, do které náleží všechny vzory, které před sebou nemají název podmínky. Přechod do stavu se provede příkazem `BEGIN(názevStavu);`. Pakliže je podmíněný stav exkluzivní, budou se porovnávat jen vzory stavu příslušející. V případě inkluzivního stavu se porovnávají všechny příslušející stavy a stavy, které žádnému stavu nepřísluší. Při implementaci mi pro přehlednot přišlo čitelnější a intuitivnější používat pouze exkluzivní stavy.

4.5 Implementace syntaktického analyzátoru v nástroji Bison

Po prvotní implementaci lexikálního analyzátoru jsem se pokusil sepsat odpovídající syntaktický analyzátor, který dokázal správně rozpoznat jednotlivé konstrukce nacházející se v programu. Až v momentě, kdy tento nástroj plnohodnotně fungoval na příkladu 4.1 a generoval pro něj odpovídající `PlantUML` kód, jsem oba dva analyzátory rozšiřoval o další konstrukce jazyka Structured Text.

4.5.1 Struktura zdrojového souboru pro nástroj Bison

Zdrojový gramatický soubor pro nástroj `Bison` má příponu `.y` a svou strukturou se podobá souboru pro nástroj `Flex`. Opět je rozdělen do tří částí. Zpravidla je ovšem před první částí blok kódu mezi znaky `%{...}%` obsahující hlavičkové soubory a deklarace lexikálního analyzátoru `yylex` a dalších využívaných globálních proměnných a funkcí. [35]

První část obsahuje deklaraci názvu a vlastností jednotlivých symbolů používaných v gramatice jazyka. Každá deklarace začíná znakem `%`. Je zde nutné deklarovat

typy tokenů, které lexikální analýzátor rozlišuje. Všechny symboly, které zde nejsou deklarovány a jsou použity v následující části s pravidly, jsou považovány za neterminální. Před jméno symbolu lze také vložit do špičatých závorek jeho odpovídající datový typ, který musí být předtím deklarován uvnitř bloku `union`. Pro jednotlivé symboly lze také definovat jejich vlastnosti jako asociativita a precedence.

Výpis 4.5: Příklad deklarační části parsesru v jazyce `Bison`

```
%union {
  int ival;
  float fval;
}
%token<ival> T_INT /*Tokeny jsou terminály*/
%token<fval> T_FLOAT
%token T_PLUS T_MINUS T_MULTIPLY T_DIVIDE T_LEFT T_RIGHT
%type<ival> expression /*Typy jsou neterminály*/
%type<fval> mixed_expression
%start body /*defnuje počáteční symbol*/
```

Druhá část gramatického souboru obsahuje gramatická pravidla v Backusove-Naurově formě (zkráceně BNF), která má obecnou podobu `výsledek : komponenty ... akce`, kde výsledek je název neterminálního symbolu a komponenty se skládají z jednotlivých symbolů, které dohromady tento výsledek tvoří. Opět obdobně jak vsouborech nástroje `Flex` lze jednotlivé komponenty pro stejný výsledek oddělit znakem `|`. I přesto nástroj `Bison` s nimi operuje jakoby se jednalo o různá pravidla. Až na speciální případy bude při konstrukci analyzátoru v pravidlech použita rekurze. Pro optimální fungování nástroje `Bison` je potřeba, aby se jednalo o levostrannou rekurzi. Za daným pravidlem následuje akce v jazyce `C` nejčastěji provádějící sémantické operace nad symboly.

K proměnné reprezentované symbolem lze přistupovat pomocí identifikátoru `$n`, kde n odpovídá n-tému prvku komponent pravidla. K hodnotě výsledného symbolu pak lze přistupovat pomocí identifikátoru `$$`. Alternativně lze k hodnotám symbolu přistupovat pomocí identifikátoru `$` následovaným názvem použitého symbolu. Identifikátor symbolu ve výrazu dle jeho pořadí ve větě lze taky deklarovat explicitně bezprostředně za symbolem ve hranatých závorkách, což je nutné u pravidel, kde se stejný symbol vyskytuje vícekrát.

Všechny hodnoty jsou uloženy ve struktuře jazyka C typu `union`, která se vyznačuje tím, že dokáže uchovávat vícero datových typů na jednom paměťovém místě. Tento přístup při implementaci dokumentačního nástroje, kdy operuji s řetězci, je-

Výpis 4.6: Příklad části s gramatickými pravidly parsesru v jazyce `Bison`

```
exp[result]: exp[left] '+' exp[right]
  { $result = $left + $right; } /*explicitní identifiká
     tory*/
exp: expression T_DIVIDE expression
    { $$ = $1 / (float)$3; } /*identifikátory dle pořadí
       */
```

jichž hodnota se musí uchovávat i mimo danou sémantickou akci mi připadal neintuitivní. Proto jsem tuto funkcionalitu nástroje `Bison` nevyužil a místo toho jsem operoval s globálními proměnnými, které nejsou omezeny strukturou `union`.

Třetí část obdobně jako u nástroje `Flex` je přímo zkopírována do vygenerovaného zdrojového souboru analyzátoru. Zde je tedy možné definovat funkci `yyerror(const char* s)`, která obstarává hlášení chyb během analýzy uživateli. řetězec předávaný do této funkce obsahuje textový popis chyby, která její vyvolání způsobila. Pro detailnější popis chyby je potřeba v definiční části nastavit možnost `define parse.error verbose`.

4.5.2 Zpracování chyb

Bez mechanismu ošetření chyb skončí analýza při nalezení první syntaktické chyby. Tomu se dá zabránit, pokud do gramatiky jazyka zahrneme speciální předdefinovaný terminální symbol `error`, který je vrácen při syntéze derivačního stromu vždy, když není nalezeno žádné příslušné derivační pravidlo (Je nalezena syntaktická chyba). Pro dané pravidlo se symbolem `error` lze samozřejmě definovat i příslušnou akci, která ovšem může být i prázdná. Anaylýza se pak pokusí zpracovat další symboly, přičemž dočasně potlačí tisk chybových zpráv. Toto potlačení lze zrušit v akci pomocí instrukce `yyerrok;`. Počet detekovaných chyb je uložen v globální proměnné `yyners`.

4.5.3 Použití nástroje Bison

V implementovaném programu je pro spuštění celé analýzy textu zavolat funkci `int yyparse()`, která pak sama dle potřeby volá funkci `yylex` pro lexikální analýzu. Provádění funkce je ukončeno v momentě, kdy je dosaženo konce souboru a funkce `yywrap()` vrátí jedničku. V tomto případě `yyparse` vrací nulu. Když analýza narazí na syntaktickou chybu, ze které se nedokáže zotavit, vrátí tato funkce jedničku. V případě chyby způsobené vyčerpáním paměti vrací dvojku. [38]

Zavoláním příkazu `bison` je vytvořen zdrojový soubor `parser.tab.c`. S přepínačem `-d` je rovněž vygenerován stejnojmenný hlavičkový soubor, obsahující definice jednotlivých tokenů, které očekává od lexikálního analyzátoru. Zde se tedy oba nástroje z hlediska programátora propojují.

Konflikty gramatických pravidel

Vytvořený analyzátor lze vizualizovat přeložením souboru s přepínačem `-graph` a názvem výstupního souboru s příponou `.gv` v jazyce DOT, který lze převést do grafické podoby nástrojem `dot`. Tato vizualizace vykresluje implementovaná gramatická pravidla a pomáhá odhalit problémy v nich obsažené, jako například nadbytečné symboly nebo konflikty pravidel.

Jedním z konfliktů pravidel gramatiky je shift/reduce koflikt, kdy je daný symbol v pravidle možné jak přesunout (shift) daný token na zásobník pro pozdější zpracování nebo symbol redukovat (reduce) na jiný neterminální symbol podle jiného pravidla. Pro příklad uvažujme následující pravidla:

Výpis 4.7: Příklad konfliktních pravidel

```
    if_stmt:
  "if" expr "then" stmt
| "if" expr "then" stmt "else" stmt
;
```

V momentě, kdy ve zdrojovém kódu je vícero podmínek vnořených do sebe, je při dosažení klíčového slova `else` možné jak redukovat symboly na symbol `if_stmt` podle prvního pravidla, tak vložit tento symbol na zásobník podle druhého pravidla. V prvním případě by větev příkazů za `else` příslušela podmínce v první úrovni vnoření. V druhém případě by připadala v nejspodnější úrovni vnoření. Jelikož ve většině jazyků je žádoucí možnost druhá, nástroj `Bison` vždycky preferuje tuto variantu. Jelikož je ovšem tato gramatika nejednoznačná, nástroj při překladu na tuto skutečnost upozorňuje. Detailní výpis je možno získat pomocí přepínače `-Wcounterexamples`. Aby byl tento konflikt vyřešen, je nutno explicitně označit precedenci symbolu při jeho definici nebo specifikovat jeho asociativitu. [39]

Další konflikt, co se může vyskytovat v gramatice, je reduce/reduce konflikt, který se vyskytuje v momentě, kdy stejnou sekvenci symbolů lze redukovat vícero pravidly. Tato chyba většinou představuje hrubou chybu v gramatice. I když nezáleží, které pravidlo je pro redukci použito, závisí, která akce je provedena. V případě tohoto konfliktu provede nástroj `Bison` akci, která se ve zdrojovém gramatickém souboru nachází jako první. [40]

Použití nástroje `Bison` pro implementaci dokumentačního nástroje zpětně hodnotím jako možná nadbytečné, jelikož nástroj neprovádí kompletní syntaktickou analýzu v jazyce Structured Text. Překlad zdrojových textů mimo víceřádkové komentáře probíhá po jednotlivých řádcích a proto kontrola syntaktické korektnosti se omezuje pouze na skladbu konstrukcí, co se vyskytují na jednom řádku. Není tudíž ověřeno, že podmínky a cykly jsou v textu správně ukončeny. V tomto ohledu nástroj spoléhá na korektnost textu, kterou lze ověřit překladem v prostředí Automation Studio. Přesto kvůli klíčovému slovu `ELSE`, které ovlivňuje výsledek překladu, bylo potřeba vytvořit dva zásobníky, které udržují informace o typu bloku, ve kterém se překladač právě nachází a jestli se v něm klíčové slovo `ELSE` vyskytlo. Tudíž správnost ukončnení bloků do jisté míry je ověřena.

5 Propojení analyzátoru s nástrojem Doxygen

Veškerá doposud popsaná implementace probíhala na jednotlivých souborech. Některé z nich byly čistě syntetické, a obsahovaly postupně s nárustem implementovaných pravidel všechny konstrukce v jazyce Structured Text. Ostatní pak byly jednotlivé soubory ze zvolených demonstračních úloh. Až teprve tedy po implementaci analyzátoru jsem začal pracovat na způsobu jak jej propojit s nástrojem Doxygen.

Aby bylo možné pomocí nástroje Doxygen vykreslovat diagramy, je nutné mít nainstalován balíček nástrojů Graphviz. Dále je mít potřeba nainstalovaný Java Runtime Enviroment, jelikož binární soubor pro vykreslování diagramů v jazyce `PlantUML` je napsán v jazyce Java. Cestu k těmto souborům je možné nastavit v souboru `Doxyfile` ve značce `DOT_PATH`, respektive `PLANTUML_JAR_PATH`. Aby mohly diagramy obsahovat hypertextové odkazy, je potřeba značku `DOT_IMAGE_FORMAT` nastavit na hodnotu `svg` a značku `INTERACTIVE_SVG` na hodnotu `YES`.

5.1 Vstupní filtry v nástroji Doxygen

Jak již bylo zmíněno v kapitole 1.7, Doxygen umožňuje před analýzou každého zdrojového souboru nad ním provést předzpracování. Umístění programu, který tuto úlohu provede, lze specifikovat v souboru `Doxyfile` buď pomocí značky `INPUT_FILTER`, který předzpracování provede nad každým zpracovávaným souborem, nebo lze ve značce `FILTER_PATTERNS` specifikovat tento program a vzor názvu souborů, pro které má být předzpracování aplikováno. [41] Jelikož projekt v Automation Studio může obsahovat i kód v jazyce `C`, který lze dokumentovat, je praktické použít značku `FILTER_PATTERNS` a v ní specifikovat všechny soubory s příponou `st`, které nástroj bude zpracovávat.

Doxygen má vnitřní pravidla pro volbu vlastních dokumentačních analyzátorů specializovaných na různé jazyky dle přípony. Tato pravidla lze přepsat nebo přidat vlastní ve značce `EXTENSION_MAPPING`. Odpovídající přípony pro tato pravidla musí být také specifikována ve značce `FILE_PATTERNS`, ve které jsou zahrnuty veškeré soubory, které mají být zpracovány. Naopak umístění, která mají být z těchto souborů vyloučena, lze specifikovat ve značce `EXCLUDE`. Typicky například není žádoucí dokumentovat složku `Libraries` obsahující v projektu použité knihovní funkce.

Variantu, kdy by bylo použito vstupních filtrů, jsem zrealizoval mapováním výstupu z mého překladače na analyzátor jazyka `Markdown` nástroje Doxygen. Výsledná dokumentace je pak vykreslena jako běžný `Markdown` dokument. Můj překladač bylo

pouze potřeba modifikovat, aby zpracovaný soubor vypsal na standardní výstup, jelikož toto je od vstupních filtrů očekáváno.

Varianta s použitím vstupních filtrů je funkční, ovšem mezi jednotlivými zdrojovými soubory nelze snadno vytvářet vazby. Takto vytvořená dokumentace postrádá smysl. Proto jsem přikročil k variantě, kdy můj dokumentační program nejprve vytvoří `Markdown` dokumenty pro všechny relevantní zdrojové soubory a teprve ty jsou pak zpracovány nástrojem Doxygen.

5.2 Dvouprůchodová analýza projektu

Aby bylo možné mezi soubory vytvořit vazby, je nejprve nutné projít všechny podadresáře projektu a v nich najít soubory s příponou `.st`. Takový program potřebuje funkce pro procházení souborovým systémem, který je ovšem závislý na operačním systému. Můj program je momentálně implementovaný pro systém Linux a používá proto knihovnu `dirent.h`. Vzhledem k budoucímu použití s prostředím Automation Studio bude ovšem nunté vytvořit i variantu pro systém Windows. Jelikož celý průchod adresáři je prováděn jednou funkcí, neměla by adaptace na systém Windows být obtížná. Během průchodu jsou názvy jednotlivých nalezených souborů ukládány do pole `gProcessedFiles`, aby při druhém průchodu nebylo nutné je v adresářích znovu vyhledávat.

První průchod má za úkol pouze identifikovat a uložit názvy jednotlivých akcí, funkcí a funkčních bloků. Pro tento účel bylo nejjednodušší vytvořit další lexikální analyzátor ve zdrojovém souboru `prescanner.l` pouze k tomuto účelu. Po přeložení nástrojem `flex` by však vznikly dva různé lexikální analyzátory volané stejnou funkcí `yylex()`. Proto lze do definiční části lexikálního zdrojového souboru vložit nastavení `%option prefix="symbs"`, která všem identifikátorům funkcí a proměnných analyzátoru dá předponu specifikovanou tímto nastavením. Takto z programu místo funkce `yylex()` je volaná funkce `symbslex()` a nedojde tak ke konfiktům definic při kompilaci programu. Rovněž je třeba poznamenat, že oba lexikální analyzátory je potřeba nástrojem `flex` přeložit samostatně, aby byly vytvořeny dva různé soubory v jazyce `C`.

Samotné názvy akcí, funkcí a funkčních bloků [1] jsou ukládány do tabulky symbolů, kterou jsem implementoval v souboru `TSymbolTable.c`. Spolu s názvem symbolu je uložen rovněž název souboru, ve kterém se tento symbol vyskytuje. Název souboru je ovšem předtím zpracován tak, aby rovnou odpovídal názvu stránky, kterou nástroj Doxygen pro tento soubor vygeneruje. Jelikož se jedná o nástroj s ote-

[1] Dále v práci souhrně označuji jako podprogramy kvůli neexistenci jiného zastřešujícího pojmenování. Navíc Automation Studio tento termín nepoužívá oproti například systémům od Rockwell Automation.

vřeným kódem, bylo možné najít funkce realizující tuto činnost v jeho zdrojových textech v souboru `util.cpp`. Tyto funkce jsou napsané v jazyce `C++`. Proto jsem je adaptoval do jazyka `C`. Jelikož používali pro pokročilejší práci s řetězci třídu `QCString`, vytvořil jsem také pro vlastní potřeby strukturu `SizedString`, která mi umožnila jednoduššeji spravovat dynamicky alokovanou paměť při práci s řetězci, a následně jsem ji použil pro téměř všechny řetězce ve zbytku programu. Správnost implementace jak tabulky symbolů, tak struktury `SizedString`, jsem ověřil v testovacích programech `TSymbolTableTest.c`, respektive `TSizedStringTest.c`. Korektní správu paměti jsem ověřil nástrojem `Valgrind`. [42]

Aby názvy souborů v odkazech ve vygenerovaných diagramech odpovídaly názvům vygenerovaným nástrojem Doxygen, je potřeba v konfiguračním souboru `Doxyfile` dodržet následující nastavení:

- `CREATE_SUBDIRS = NO` - Všechny vygenerované soubory jsou uloženy do jednoho adresáře.
- `SHORT_NAMES = NO` - Povoluje generovat delší a čitelné názvy souborů.

Zároveň ale není optimální, aby názvy zdrojových souborů byly delší než 128 znaků, protože nástroj Doxygen by je mohl i přes nastavení `SHORT_NAMES` zkrátit. Ostatní nastavení v souboru `Doxyfile` jsem ponechal výchozí. Adresář, ve kterém má Doxygen vyhledávat zdrojové soubory musí být nastaven na stejné umístění, ze kterého je nástroj Doxygen spouštěn. Typicky se v rámci projektu Automation Studia jedná o složku `Logical`.

Druhý průchod provádí již implementovanou analýzu uvedenou v předchozích kapitole 4.4. Ta byla pouze rozšířena o funkci porovnávání zpracovaného textu s vytvořenou tabulkou symbolů a vložením odkazu do vygenerovaného textu v případě nalezení shody.

5.3 Generování diagramu komponent

Vytvořené `Markdown` dokumenty jsou ve vygenerované dokumentaci automaticky uvedeny v seznamu `Related Pages` a nejsou uspořádáné do hiearchie ani podle adresářů. Jelikož takto nelze rozlišit, zda-li soubor obsahuje program nebo podprogram, bylo potřeba najít způsob, jak vložit na hlavní stránku dokumentace rozcestník podobný hlavní straně ve vzorové dokumentaci.

Struktura projektu v Automation Studio je reprezentovaná `XML` soubory s příponou `.pkg`, které jsou přítomny v adresářích projektu. Při prvním průchodu analýzy projektu jsou tyto soubory otevřeny a pomocí funkcí knihovny `libxml` jsou z nich vyčteny hodnoty všech elementů typu `Program`, které v Automation Studiu reprezentují programový balíček.

Dále je také v adresáři projektu hledán soubor s názvem `mainpage`. Pokud obsahuje příkaz `\mainpage`, který pro nástroj Doxygen označuje soubor, který má být zobrazen jako hlavní stránka dokumentace, bude jeho obsah vložen před vygenerovaný graf. Výsledný soubor `index.md` se bude nacházet ve stejné složce. Pokud příkaz `\mainpage` není nalezen, je nový soubor obsahující tento příkaz vytvořen v podadresáři `Documentation`.

V druhém průchodu analýzy jsou názvy nalezených programů porovnány s názvy nalezených zdrojových souborů a jsou vytvořeny odkazy na tyto soubory. Proto je nutné, aby názvy programů korespondovaly s názvy zdrojových souborů, ve kterých se tyto programy vyskytují. Následně je vygnenerován diagram pomocí `PlantUML` a je přidán do souboru `index.md`. Syntaxe vygenerovaného diagramu komponent je uvedena ve výpise 5.1.

Výpis 5.1: Příklad kódu v jazyce `PlantUML` generující diagram komponent

```
1 @startuml
2 component Program
3 url of DbComm is [[./odkaz_na_program.html]]
4 @enduml
```

Vygenerován je jeden komponent s textem s popiskem `Program` a hypertextovým odkazem specifikovaným v dvojitých hranatých závorkách

5.4 Použití implementovaného překladače

Implementovaný překladač jsem pojmenoval `stdoc`. Je volán z terminálu s následujícími možnými parametry:

- <vstupní soubor1> <vstupní soubor2> ... <vstupní souborN> - Zpracuje jednotlivé soubory bez jejich vzájemného propojení a uloží je do současného pracovního adresáře.
- -r <adresář> - Zpracuje všechny nalezené soubory v zadaném adresáři a jeho podadresářích a uloží je do umístění, kde byly nalezeny. Dále vytvoří soubor `index.md` s odkazy na vygenerované soubory.

Pro správné vygenerování odkazů je nutné specifikovat adresář `Logical` v projektu Automation Studia. Překladač při zpracování na terminál vypisuje informace o průběhu a případných chybách. Po dokončení překladu je potřeba ručně zavolat příkaz `doxygen` pro vygenerování dokumentace. Soubor `Doxyfile` musí být umístěn v adresáři `Logical`, aby bylo možné specifikovat adresář se zdrojovými soubory jako současný pracovní adresář. (V případě nástroje `Doxywizard` je potřeba v nastavení

`Source code directory` úvést znak tečky.) Vygenerovaná dokumentace je uložena do libovolného adresáře nastaveného ve značce `OUTPUT_DIRECTORY`. Hlavní strana dokumentace se nachází v souboru `index.html`.

5.4.1 Syntaxe příjmaná překladačem

Implementace návrhu syntaxe z kapitoly 4.3 byla funkční, ovšem při aplikaci na demonstrační úloze byly vygenerované diagramy velmi podobné zdrojovému kódu a nebyly vůbec přehledné pro účely dokumentace. Proto byl překladač přepracován tak, aby akceptoval pouze okomentované podmínky, cykly a příkazy. Výjimku z tohoto pravidla jsou výrazy obsahující názvy volaných podprogramů.

Komentáře určené pro překladač, které se nacházejí ve zdrojovém souboru mimo bloky programů jsou zkopírovány do dokumentace jako text. Lze používat i příkazy pro nástroj Doxygen, které jsou běžně podporované v souborech `Markdown`, jako například `\todo` nebo `\warning`. Uvnitř bloku programu je nelze použít.

Komentář pro označení konstrukce, která se má objevit v diagramu, může být i prázdný a je tak zkopírován pouze výraz v této konstrukci. Naopak lze také použít i prázdný příkaz, který do diagramu vloží pouze text komentáře. Dále platí, že text v diagramech se automaticky nezalamuje a proto každý řádek, který blokový komentář obsahuje, je zkopírován do textu v diagramu.

Výpis 5.2: Ukázka použití prázdných komentářů a prázdných příkazů

```
1 IF podmínka_bez_komentáře THEN //*
2     ;(**Víceřádkový komentář
3     bez příkazu**)
4 END_IF;
```

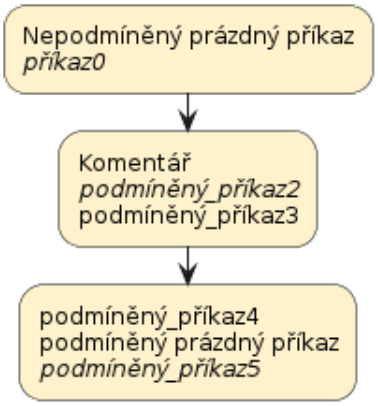
Obr. 5.1: Část diagramu prezentující použití prázdných komentářů a prázdných příkazů vygenerovaný ze zdrojového kódu ve výpise 5.2

Vícero po sobě jdoucích příkazů lze pro přehlednost spojit do jednoho uzlu diagramu. Je proto potřeba zdokumentovat pouze jeden příkaz a všechny bezprostředně

následující budou k němu připojené. Toto je přerušeno vložením prázdného řádku nebo počátkem jakékoliv jiné konstrukce než je příkaz zakončený středníkem. Této funkcionality lze využít pro zlepšení přehlednosti, kdy program nastavuje vícero proměnných ve stejném bloku programu. Toto pravidlo platí i pro prázdné příkazy.

Výpis 5.3: Ukázka hromadné dokumentace vícera příkazů v programu

```
1  ;(**Nepodmíněný prázdný příkaz**)
2  příkaz0;
3  IF nepodstatná_podmínka THEN
4      podmíněný_příkaz1;
5      podmíněný_příkaz2;    //*Komentář
6      podmíněný_příkaz3;    //*
7
8      podmíněný_příkaz4;     //*
9      ;(**podmíněný prázdný příkaz**)
10     podmíněný_příkaz5;
11 END_IF;
12 příkaz6;
```



Obr. 5.2: Část diagramu prezentující možnost hromadného dokumentování vícera příkazů vygenerovaný ze zdrojového kódu ve výpise 5.3. Všimněme si, že jelikož podmínka není zadokumentovaná, není v diagramu vyobrazena. Obdobně je tak tomu u příkazů 1 a 6, kterým nepředchází žádný zadokumentovaný příkaz v daném bloku.

V bloku podmínky může být ignorován její první výraz a do dokumentace lze vložit teprve tu podmínku, která následuje za slovem `ELSIF`. Specifikováním komentáře za slovem `ELSE` lze nahradit výchozí text `No` na hraně přechodu. V tomto případě však blok odpovídající podmínky již musí být zadokumentován.

Výpis 5.4: Ukázka dokumentace podmínek v programu

```
1 IF nevypsaná_podmínka THEN
2     příkaz1;
3 ELSIF vypsaná_podmínka THEN //*komentář podmínky
4     příkaz2;
5 ELSE (**Alternativní
6 větev**)
7     příkazN; //*
8 END_IF;
```

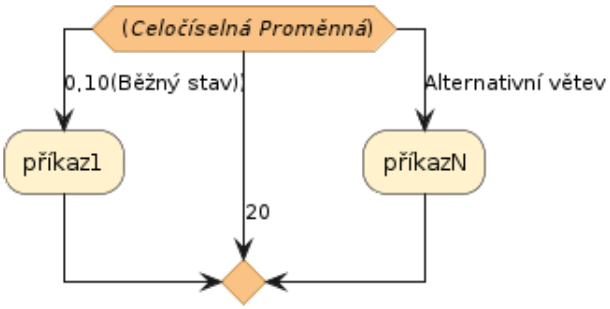
Obr. 5.3: Část diagramu prezentující možnosti dokumentace podmínek vygenerovaný ze zdrojového kódu ve výpise 5.4

Obdobně pak funguje blok `CASE`, kde ovšem musí být okomentovaný začátek tohoto bloku. Automaticky jsou pak v grafu zobrazena všechna možná větvení. `PlantUML` ovšem u tohoto bloku nepodporuje víceřádkový text.

Výpis 5.5: Ukázka dokumentace bloku `CASE` v programu

```
1 CASE Celočíselná _Proměnná OF //*
2     0,10 : //*Běžný stav
3         příkaz1;    //*
4     20:
5         příkaz2;
6     ELSE    //* Alternativní větev
7         příkazN;    //*
8 END_CASE;
```

Pro bloky cyklů `WHILE`, `REPEAT` a `FOR` jsou komentáře přidány na hranu přechodu. U cyklu `REPEAT` se chová klíčové slovo `UNTIL` obdobně jako podmínka `IF`, tudíž komentář se zobrazí pod výrazem podmínky. Příkaz `EXIT` je automaticky vložen do zdokumentovaného cyklu.

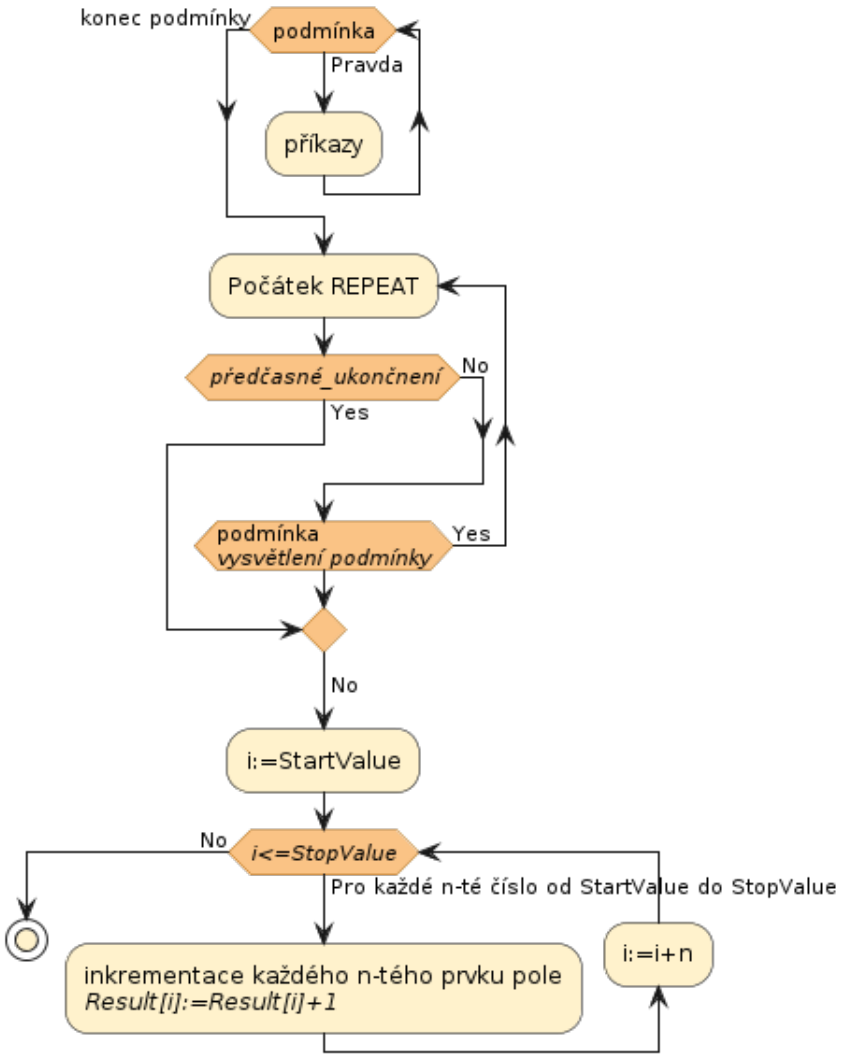


Obr. 5.4: Část diagramu prezentující možnosti dokumentace bloku `CASE` ze zdrojového kódu ve výpise 5.5

.

Výpis 5.6: Ukázka dokumentace cyklů v programu

```
1  WHILE podmínka DO   //*Pravda
2      příkazy; //*
3  END_WHILE; //*konec podmínky
4  REPEAT  //*Počátek REPEAT
5      příkazy; //
6      IF předčasné_ukončnení THEN //*
7              EXIT;
8      END_IF;
9      UNTIL podmínka //*vysvětlení podmínky
10 END_REPEAT;
11 FOR i:= StartValue TO StopValue BY n DO //*Pro každé n-té čí
       slo od StartValue do StopValue
12     Result[i] := Result[i] + 1;   //*inkrementace každého n-t
           ého prvku pole
13 END_FOR;
```



Obr. 5.5: Část diagramu prezentující možnosti dokumentace bloku cyklů vytvořená ze zdrojového kódu ve výpise 5.6

6 Volba demonstrační úlohy

Podle zadání měla být vybrána demonstrační úloha využitelná v laboratoři číslicové řídicí techniky na UAMT pro demonstraci funkčnosti implementovaného dokumentačního programu. Jelikož však v době psaní práce neexistuje v této laboratoři žádná úloha, která by ve svém projektu využívala jazyk Structured Text, bylo nutné zvolit jinou úlohu.

Pro účel práce byl zvolen projekt s názvem `edgemachine`, který pro splnění této práce poskytla společnost B&R. Hlavními funkcemi projektu je zaznamenávání celkové efektivnosti zařízení (anglicky OEE) a komunikace těchto dat s databází. Nejedná se tedy o projekt, který by v poskytnutém stavu implementoval řízení konkrétního stroje. Není předmětem firemní zakázky, ale je využíván v rámci společnosti B&R. Programy (respektive úlohy) obsažené v projektu mají za úkol sledovat hodnoty vstupních a výstupních karet, zpracovávané objednávky a výrobní časy těchto objednávek nebo jednotlivých kusů. Dále také zaznamenávají časy a důvody prostojů a další informace, které jsou následně zpracovány a poskytnuty vizualizačním úlohám.

Tento projekt je vhodný pro demonstraci implementovaného dokumentačního programu, jelikož obsahuje vícero úloh a podprogramů, na kterých lze demonstrovat generování vazeb mezi jednotlivými soubory. Dále k tomuto projektu také existuje vzorová ručně vytvořená dokumentace v nástroji diagrams.net, která může být použita pro srovnání s dokumentací vygenerovanou nástrojem Doxygen a podle níž byl implementovaný dokumentační program upravován tak, aby bylo dosaženo co nejpodobnějšího výsledku.

7 Srovnání a demonstrace dokumentace

V této kapitole je nejprve představena poskytnutá vzorová dokumentace k demonstračním úlohám a následně je provedeno srovnání s dokumentací automaticky vygenerovanou implementovaným dokumentačním programem a nástrojem Doxygen.

7.1 Vzorová dokumentace

Vzorová dokumentace byla poskytnuta jak ve formátu `.drawio`, který je možné otevřít v nástroji diagrams.net, tak ve formátu HTML dokumentu. Oba soubory je možné najít v příloze ve složce `Documentation`. V případě `.drawio` souboru se jedná o upravitelnou předlohu tvořenou jednotlivými stránkami dokumentace, ze které je pak vytvořen HTML dokument v jednom souboru. V něm je možné přepínat mezi jednotlivými stránkami, jakoby se jednalo o PDF dokument. Mimo hlavní stránku se lze vrátit zpět na „nadřazenou“ stránku pomocí odkazu umístěném na symbolu šipky zpět v horním levém rohu.

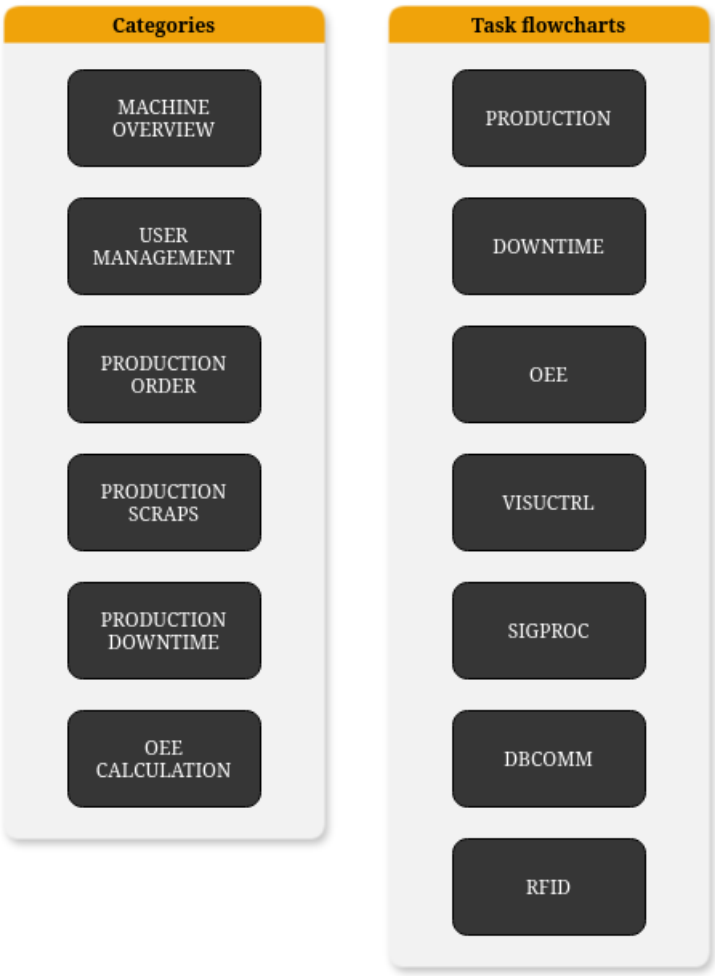
Hlavní stránka obsahuje údaje o verzích softwarových komponent použitých v projektu a položky kategorie (Categories) a vývojové diagramy úloh (Task Flowcharts).

Členění projektu na základě kategorií není určeno na základě žádného konkrétního pravidla, nýbrž vzniklo pouze podle logického členění při návrhu celé aplikace. V každé kategorii se nachází výčet úloh, které se dané kategorie týkají včetně odkazu na vývojové diagramy těchto úloh. Dále jsou zde tabulky obsahující datové typy globálních proměnných aplikace, které daná úloha v kategorii obsluhuje. Dále se v dokumentaci vyskytují další doplňující informace jako například stavové diagramy. Jelikož tato část dokumentace nemá přímou podobnost jak v programovém kódu, tak ani ve struktuře projektu a ani nebyl tento typ dokumentace předmětem této práce, není tato část v automaticky generované dokumentaci přítomna.

V položce vývojových diagramů úloh jsou vykresleny tyto diagramy pro jednotlivé úlohy. Pokud je v těchto úlohách volána akce s existujícím diagramem, je možné na ni odtud přejít. Jednotlivé uzly v diagramech jsou barevně rozlišeny dle příslušnosti uzlu k dané kategorii (Viz předchozí odstavec.) Opět tedy z důvodu, že tyto kategorie byly vytvořeny podle logického členění při návrhu aplikace, které automaticky generovaná dokumentace nemůže rozeznat, jsou vygenerované dokumentaci barevně rozlišeny pouze příkazy a bloky podmínek a cyklů.

Vzorová dokumentace obsahuje také nedokončené části označené textem `TODO` a není tudíž kompletní. Část s vývojovými diagramy obsahuje ovšem všechny části projektu, které jsou v kódu implementovány a jedinná část označená textem `TODO`

není momentálně implementovaná ani ve zdrojových kódech. Ruční vypracování dokumentace trvalo jednomu člověku, který je zároveň autorem zdrojových kódů v projektu, přes 20 hodin.



Versions	
Component	Version
Automation Studio	V4.10.3.60
Automation Runtime	B4.91
mapp View	5.18.1
mapp Services	5.18.0

Obr. 7.1: Hlavní strana vzorové dokumentace

7.2 Vygenerovaná dokumentace

Demonstrační projekt a z něj vygenerovaná dokumentace byly z důvodu úspory místa v přílohách omezeny pouze na úlohy `RFID` a `VisuCtrl`. Jelikož jsou v úloze `VisuCtrl` volány akce, a to v několika úrovních vnoření, měla by tak být dostatečně prezentována schopnost vytváření vazeb mezi programy a akcemi.

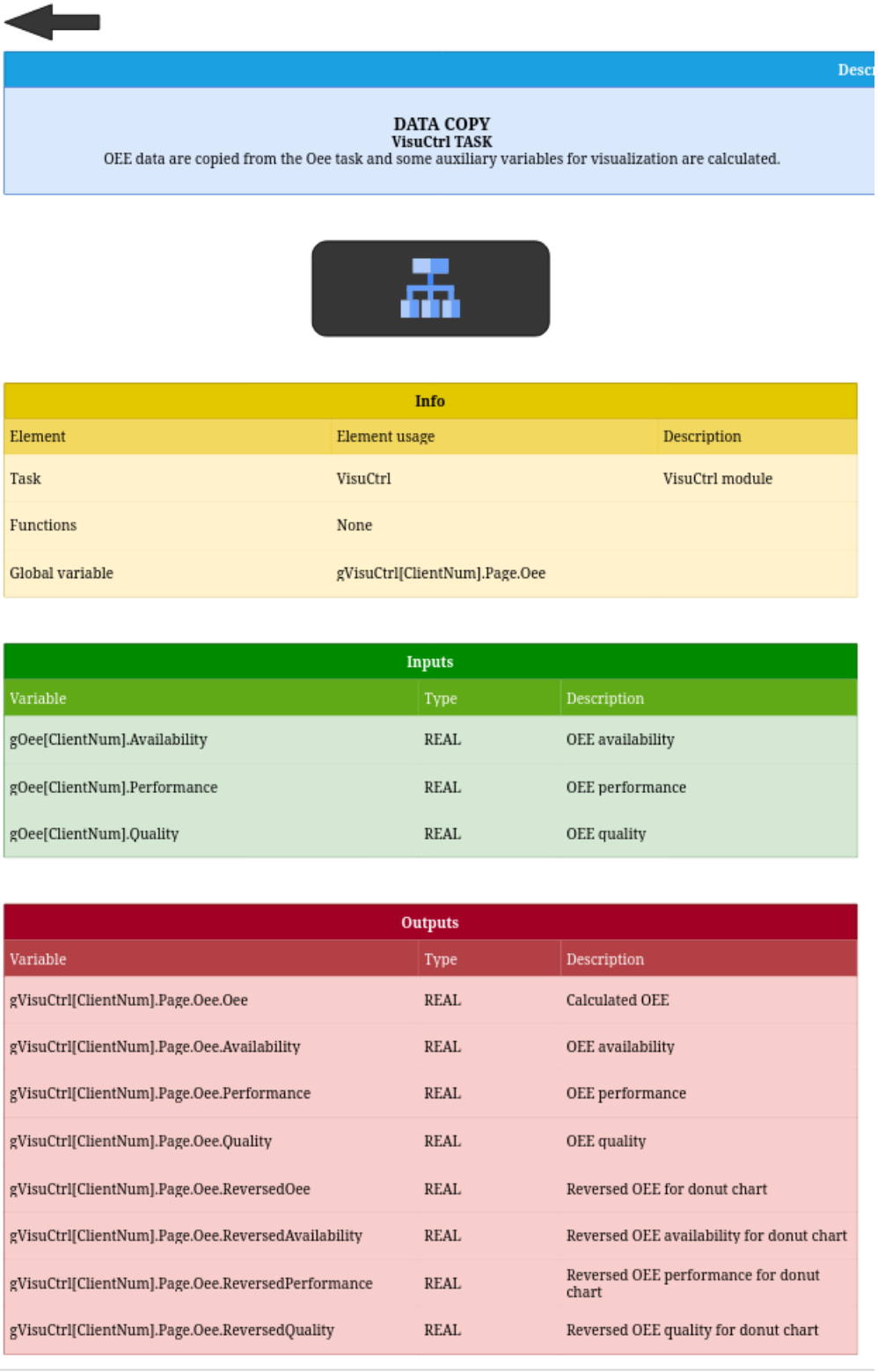
Obsah vygenerované dokumentace je nástrojem `Doxygen` uložen do podadresáře `html` v adresáři `AutoDoc` v příloze. K hlavní stránce se přistupuje z tohoto adresáře pomocí souboru `index.html`.

V automaticky vygenerované dokumentaci se nachází přes Doxygen konfigurovatelné navigační menu `Main Page`, `Related Pages`, `Namespaces`, `Classes` a `Files` (viz obrázek 7.5 na straně 61). `Main Page` odkazuje na hlavní stránku. `Related Pages` obsahuje seznam všech `Markdown` dokumentů nacházejících se v projektu. V případě vzorové dokumentace jsou zde všechny úlohy, akce a další `Markdown` dokumenty vyskytující se v projektu. Menu `Namespaces` a `Classes` obsahuje zdokumentované jmenné prostory a třídy, pokud jsou v projektu obsaženy ve zdrojových souborech v jazycích, které nástroj `Doxygen` nativně podporuje. Ve vygenerované dokumentaci k demonstračnímu projektu je možné zde nalézt knihovní funkce v jazyce `C` a skripty pro vizualizaci v jazyce `Python`. V menu `Files` se nachází seznam všech zdokumentovaných souborů.

Procházení částí dokumentace, která je implementovaná v rámci této práce, je však možné i bez těchto navigačních prvků pouze pomocí odkazů a navigačních prvků prohlížeče, které kvůli kompaktnímu řešení ve vzorové dokumentaci nelze použít.

Na hlavní stránce dokumentace je, jak již bylo zmíněno v kapitole 5.3, vložen obsah souboru `mainpage` a diagram komponent, které reprezentují jednotlivé úlohy vyskytující se v projektu. Na nich se nachází odkazy na vývojové diagramy jednotlivých úloh.

Aby vývojové diagramy programů ve vygenerované dokumentaci odpovídaly diagramům v dokumentaci vzorové, bylo potřeba všechny zdrojové soubory patřičně okomentovat podle pravidel uvedených v kapitole 5.4.1. Touto činností jsem strávil přibližně 4 hodiny.



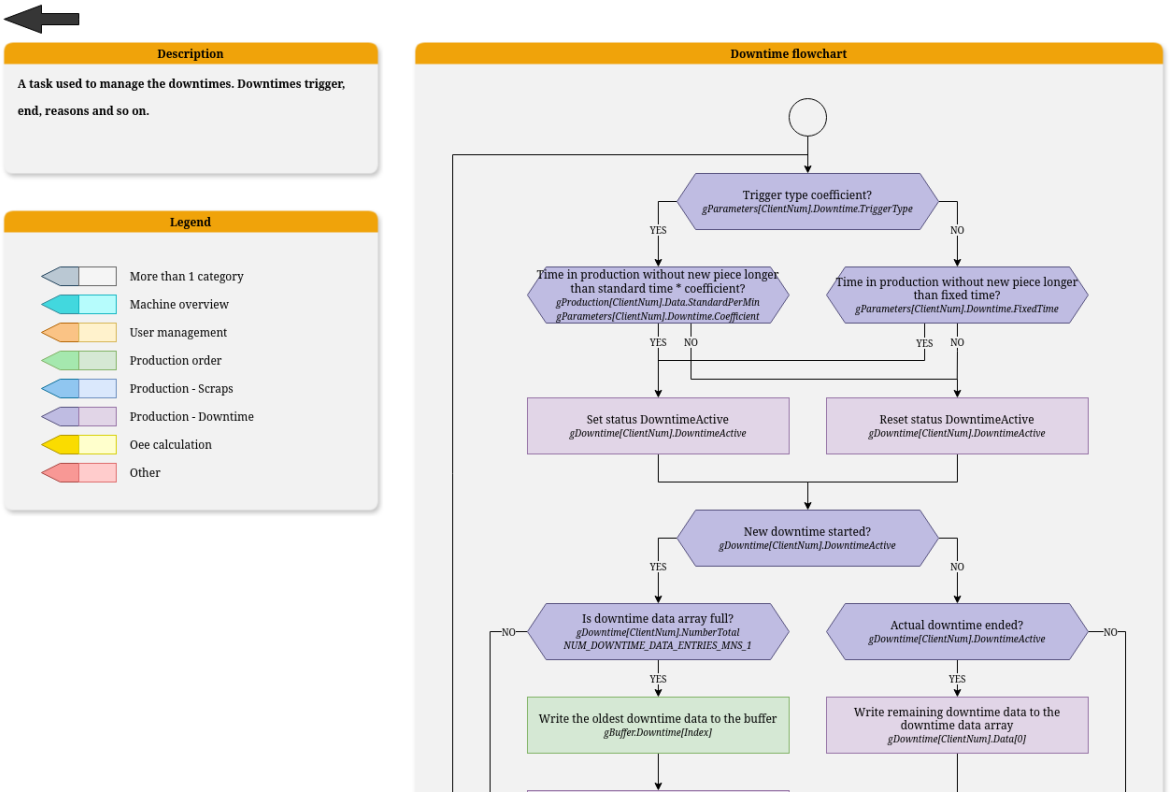
DATA COPY
VisuCtrl TASK
OEE data are copied from the Oee task and some auxiliary variables for visualization are calculated.

Info		
Element	Element usage	Description
Task	VisuCtrl	VisuCtrl module
Functions	None	
Global variable	gVisuCtrl[ClientNum].Page.Oee	

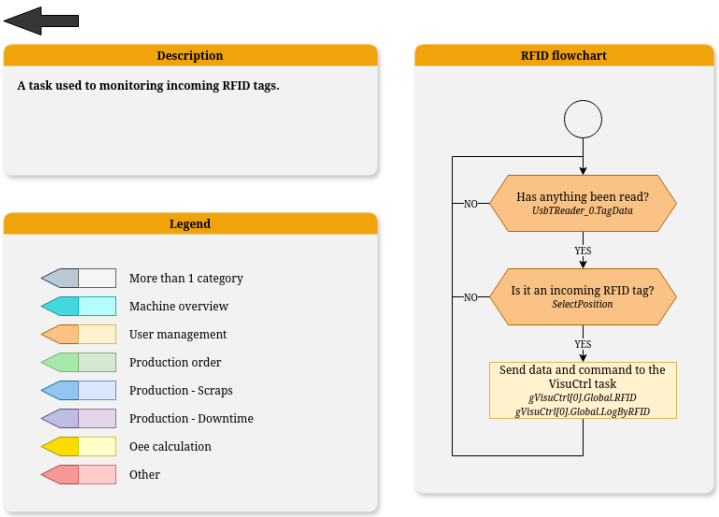
Inputs		
Variable	Type	Description
gOee[ClientNum].Availability	REAL	OEE availability
gOee[ClientNum].Performance	REAL	OEE performance
gOee[ClientNum].Quality	REAL	OEE quality

Outputs		
Variable	Type	Description
gVisuCtrl[ClientNum].Page.Oee.Oee	REAL	Calculated OEE
gVisuCtrl[ClientNum].Page.Oee.Availability	REAL	OEE availability
gVisuCtrl[ClientNum].Page.Oee.Performance	REAL	OEE performance
gVisuCtrl[ClientNum].Page.Oee.Quality	REAL	OEE quality
gVisuCtrl[ClientNum].Page.Oee.ReversedOee	REAL	Reversed OEE for donut chart
gVisuCtrl[ClientNum].Page.Oee.ReversedAvailability	REAL	Reversed OEE availability for donut chart
gVisuCtrl[ClientNum].Page.Oee.ReversedPerformance	REAL	Reversed OEE performance for donut chart
gVisuCtrl[ClientNum].Page.Oee.ReversedQuality	REAL	Reversed OEE quality for donut chart

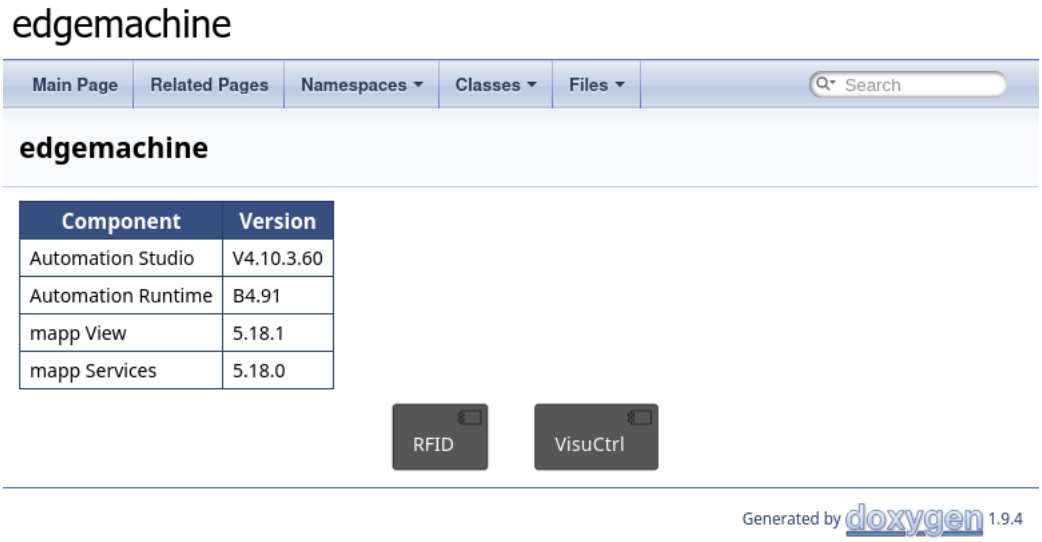
Obr. 7.2: Výstřižek stránky kategorie `OEE Calculation` ve vzorové dokumentaci. V rámci této kategorie je také druhá úloha, která na výstřižku chybí z důvodu úspory místa.



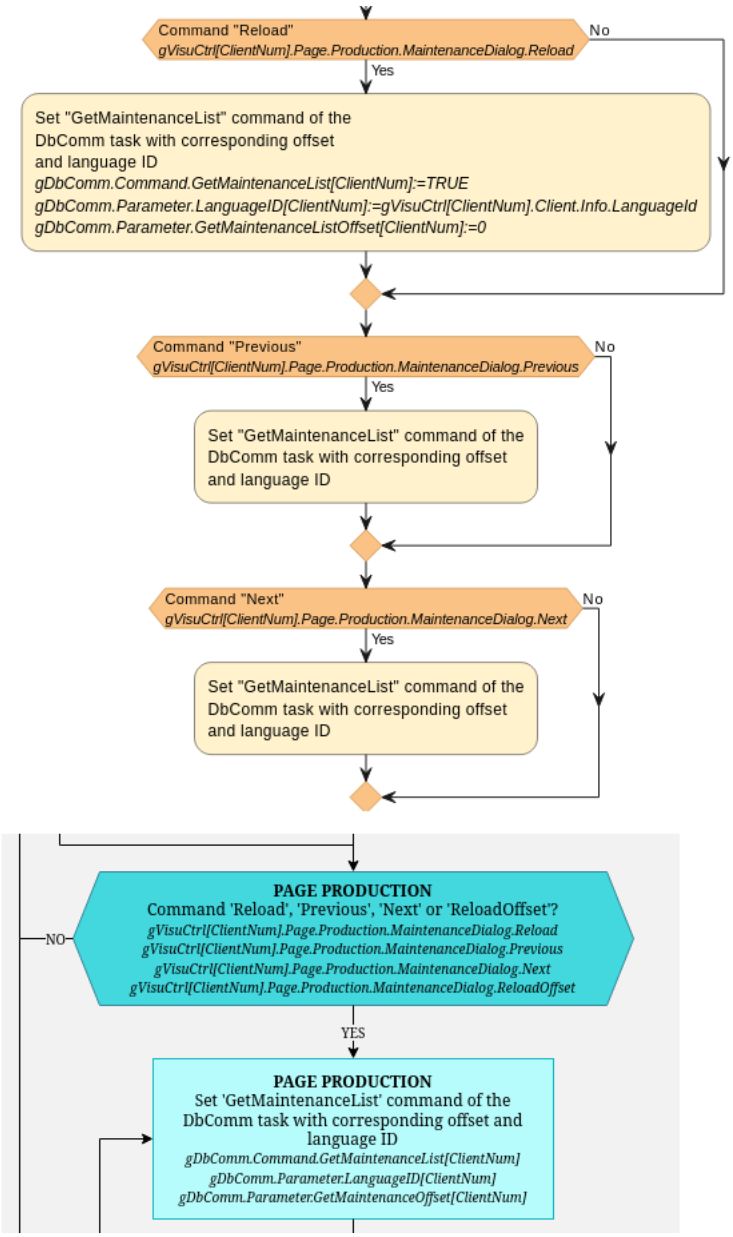
Obr. 7.3: Výstřižek strany s vývojovým diagramem úlohy `Downtime` ve vzorové dokumentaci. Obsahuje také popisek a legendu, která barevně odlišuje příslušnost uzlu k dané kategorii projektu. Obrázek je oříznut z důvodu úspory místa.



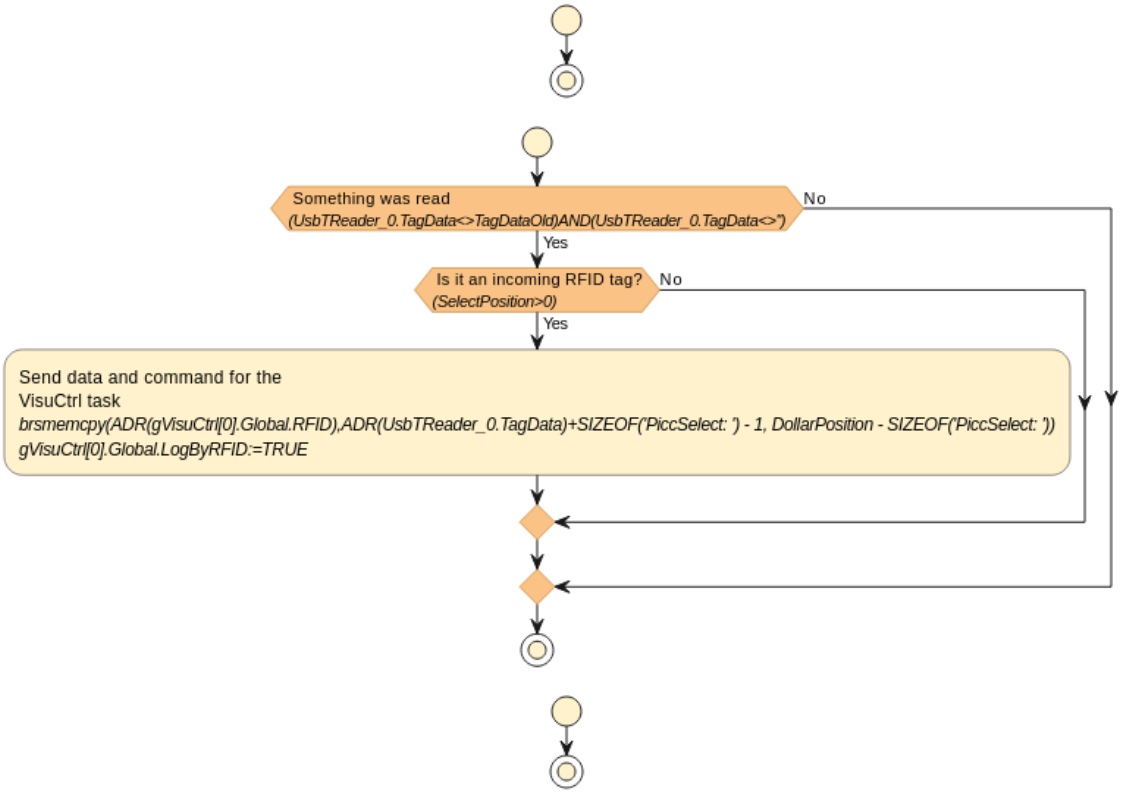
Obr. 7.4: Výstřižek strany s vývojovým diagramem úlohy `RFID` ve vzorové dokumentaci. Její barevné schéma bylo přejato pro celou automaticky generovanou dokumentaci.



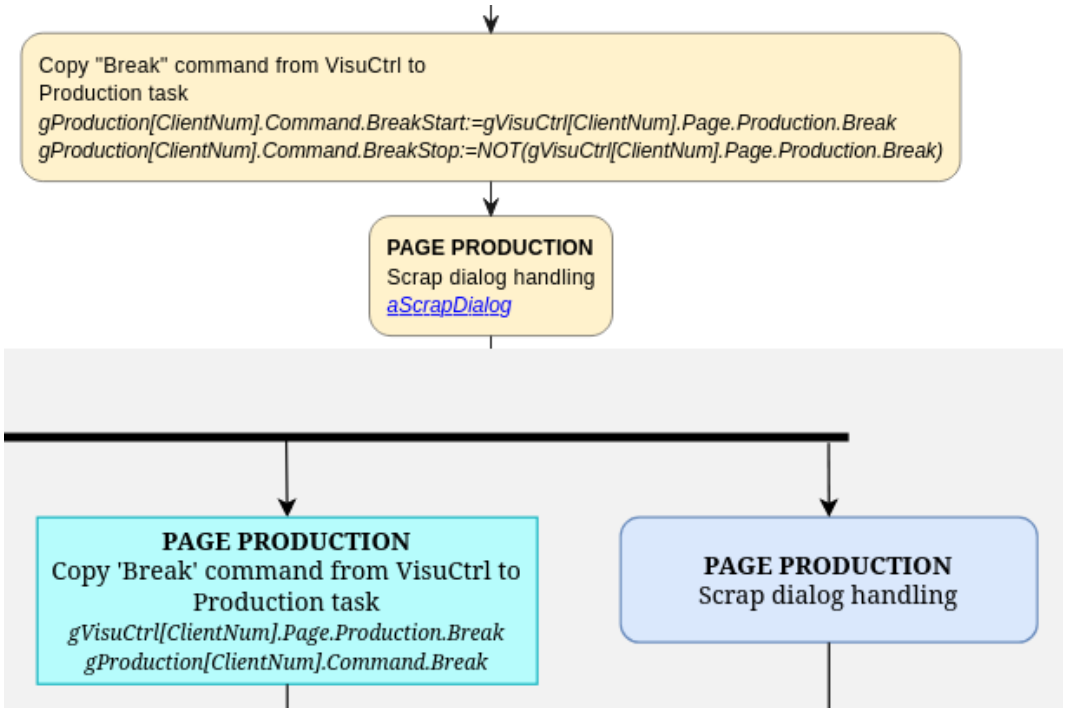
Obr. 7.5: Hlavní strana vygenerované dokumentace. Rozvržení prvků diagramu je určeno programem `PlantUML`. Z důvodu úspory místa obsahuje dokumentace pouze úlohy `RFID` a `VisuCtrl`.



Obr. 7.6: Demonstrace rozdílů v diagramech v dokumentacích na výstřižku vývojových diagramů akce `aMaintenanceDialog`. Nahoře je automaticky vygenerovaný diagram. Dole je ručně vytvořený diagram. Ne všehny prvky diagramů ve vzorové dokumentaci šlo napodobit v dokumentaci vygenerované jako zde, kdy ve vygenerované dokumentaci je vícero podobných podmínek spojených do jednoho uzlu a odpovídající příkazy také. To je dané tím, že automaticky generovaná dokumentace je stále vázaná na strukturu zdrojového kódu a není schopna takové abstrakce.



Obr. 7.7: Strana s vývojovým diagramem úlohy `RFID` ve vygenerované dokumentaci, kterou lze srovnat se vzorovou dokumentací na obrázku 7.4. Ve vygenerované dokumentaci je automaticky vykreslen i nezdokumentovaný inicializační a ukončovací program, který se ve zdrojovém kódu nachází, i když neobsahuje žádné zdokumentované konstrukce. Dále také jsou také zkopírovány celé výrazy a nikoliv pouze názvy proměnných, kterých se daná konstrukce týká, což u náročnějších příkladů může činit diagram méně přehledným. Na druhou stranu je přesně zdokumentována hodnota nebo operace, která má proměnné příslušet. Pokud není žádoucí, aby se výraz v dokumentaci vyskytoval, ale text komentáře ano, lze vždy použít prázdný příkaz s komentářem.



Obr. 7.8: Demonstrace rozdúlů v diagramech v dokumentacích na výstřižcích vývojových diagramů akce `aPageProduction`. Nahoře na obrázku je automaticky vygenerovaný diagram, dole je ručně vytvořený diagram. Všimněme si možnosti tučného formátování textu komentáře ve vygenerovaném diagramu. Nástroj `PlantUML` podporuje stylizování textu pomocí značkovacího jazyka `Creole`. [43]. Lze tedy vložit značky tohoto jazyka do komentáře ve zdrojovém kódu a `PlantUML` zajistí jeho formátované vykreslení.

8 Návrh dalších možných rozšíření

V této kapitole jsou uvedeny návrhy na další možná rozšíření implementovaného řešení, která by ho mohla zlepšit a zjednodušit jeho použitelnost.

Jedním z vylepšení je rozšíření možností úpravy vzhledu vygenerovaných diagramů. V současnosti je pro každý vygenerovaný diagram použit stejný styl definující barvy a tvary jednotlivých elementů. Ten je ovšem možné v rámci možností nástroje `PlantUML` upravit. Vytvoření vlastního stylu diagramu by bylo možné například specifikováním pomocného souboru s definicí stylu, případně jeho definováním přímo ve zdrojovém textu programu. Stylování by mohlo zlepšit přehlednost diagramů v dokumentaci a zjednodušit jejich interpretaci.

V současnosti je možné před zpracováním nástrojem Doxygen každý diagram ručně upravit editací vygenerovaného `Markdown` dokumentu a využít všech možností, které nástroj `PlantUML` nabízí. Při nové generaci diagramů implementovaným programem `stdoc` jsou však všechny tyto změny přepsány. Bylo by tedy možné program `stdoc` upravit tak, aby při generaci diagramů zohledňoval předchozí ruční úpravy a tyto změny do nově vygenerovaných diagramů zahrnoval. V demonstračních úlohách však nebylo potřeba žádné ruční úpravy `Markdown` dokumentů provádět.

Co se týče vývojářského prostředí Automation Studio, bylo by možné implementovaný program `stdoc` integrovat přímo do jeho nástrojů. Zde se vyskytuje také správce úryvků kódů (anglicky Code Snippet Manager), který usnadňuje psaní zdrojových souborů vkládáním často používaných konstrukcí jazyka. Jedním z těchto úryvků je také hlavička souboru (anglicky File Header), jejíž současná podoba by mohla být zaměněna za užitečný komentář pro program `stdoc`. Nabízí se tedy vzor této hlavičky upravit tak, aby rovnou obsahoval užitečný obsah pro nástroj Doxygen a vygenerovaná dokumentace by tak byla přehlednější (Viz vygenerovanou dokumentaci v příloze, kde původní texty hlaviček u každého souboru musely být ve zdrojovém textu ručně upraveny.) Naštěstí správce úryvků kódu v prostředí Automation Studio umožňuje snadno implicitní hlavičku upravit tak, aby obsahovala požadovaný text ve správném formátu.

Jedním z možných rozšíření je zlepšení propojení s nástrojem Doxygen. I když je možné implementovaný nástroj použít jako vstupní filtr pro nástroj Doxygen (viz kapitolu 5.1), tak výsledkem zpracování je pouze vykreslený vývojový diagram s textovým popisem. Nástroj Doxygen však umožňuje i generování přehledné dokumentace tříd (datových struktur v případě jazyka `C`), funkcí, proměnných a závislostí. V současné implementaci jsou tyto informace ve zdrojových textech v jazyce Structured Text zcela ignorovány, jelikož jsou texty přeloženy do `Markdown` dokumentů a nástroj Doxygen je jako zdrojový text nezpracuje. Bylo by tedy možné implementovat nástroj, který by překládal jazyk Structured Text do jazyka, který je

nástrojem Doxygen nativně podporován, a tím by bylo možné využít všech možností tohoto dokumentačního nástroje. Je ovšem potřeba zvážit, že struktura softwarového projektu pro PLC je odlišná od způsobu, jakým je strukturován software pro PC, a že některé informace, které nástroj Doxygen očekává, nemusí být v jazyce Structured Text dostupné. Navíc takový typ dokumentace nebyl ze strany společnosti B&R požadován a není tedy předmětem této práce.

Závěr

Cílem této diplomové práce bylo vytvořit dokumentační nástroj pro programy napsané v jazyce Structured Text pro PLC společnosti B&R Industrial Automation. Tento nástroj je schopen na základě komentářů ve zdrojovém kódu generovat UML diagramy popisující chování v projektu obsažených programů a podprogramů a umožňuje ve vzniklé dokumentaci tyto funkční celky propojit vazbami.

V první kapitole práce byla provedena rešerše nástrojů pro generování diagramů. Byly zkoumány možnosti, zdali umožňují automatizované kreslení diagramů, případně jakými způsoby by tato činnost šla zautomatizovat. Původně byl zvolen nástroj diagrams.net, známý také jako draw.io, jelikož umožňoval generovat UML diagramy pomocí tří různých textových reprezentací.

Tyto reprezentace jsou detailněji rozebrány v druhé kapitole. Jmenovitě se jedná o jazyky `Mermaid`, `PlantUML` a tabulková data s anotací ve formátu podporovaném v nástroji diagrams.net. Na příkladech jsem demonstroval jejich syntaxi a prezentoval z nich vygenerované diagramy a srovnal možnosti jejich přizpůsobení. Srovnáním jsem dospěl k zjištění, že nejvhodnějším nástrojem pro generování diagramů je `PlantUML`, jelikož automaticky vygenerované diagramy nevyžadují ruční úpravy a jsou přehledné (viz obrázek 2.5 na straně 27). Nevýhodou tohoto nástroje je, že při použití s nástrojem diagrams.net není možné v diagramech vytvářet hypertextové odkazy. Toto je možné při generování diagramů z tabulkových dat, ale takto vytvořené diagramy nejsou po vizuální stránce dostatečné (viz obrázek 2.3 na straně 23). Jelikož možnost propojení diagramů pomocí hypertextových odkazů byla pro implementovaný dokumentační nástroj klíčová, byl zvolen pro další práci nástroj Doxygen, který nástroj `PlantUML` podporuje a je oproti nástroji diagrams.net lépe konfigurovatelný a zdokumentovaný.

V třetí kapitole jsem představil vývojové prostředí Automation Studio a syntaxi jazyka Structured Text, který je oproti normě IEC 61131-3 rozšířen o možnost psaní jednořádkových komentářů a dalších v kapitole uvedených funkcí.

Ve čtvrté kapitole jsem představil základní strukturu překladačů. V jejím rámci jsem představil nástroje zvané lexikální a syntaktický analyzátor. Dále zde uvádím nástroje `Flex` a `Bison`, pomocí kterých lze tyto analyzátory vygenerovat. Nakonec jsem v této kapitole představil původní koncept syntaxe, kterou implementovaný nástroj měl používat pro extrakci informací a komentářů ze zdrojového kódu v jazyce Structured Text.

V páté kapitole jsem se zabýval propojením těchto analyzátorů s nástrojem Doxygen. Výsledná varianta implementovaného programu rekurzivně prochází adresář `PLC` projektu a pro každý nalezený soubor s příponou `.st` vytvoří stejnojmenný `Markdown` dokument obsahující kód přeložený do jazyka `PlantUML`. Tyto dokumenty

jsou následně zpracovány nástrojem Doxygen. Pro implementaci odkazů mezi diagramy bylo potřeba nástroj rozšířit o další lexikální analyzátor a jeho obslužné funkce. Následně pro snadnou navigaci ve vytvořené dokumentaci byl vytvořen rovněž diagram komponent, kde každý program v projektu je reprezentován jednou komponentou, přes jejíž odkaz je možné se dostat na vývojový diagram daného programu. Tato kapitola rovněž obsahuje návod, jak správně nastavit konfigurační soubor `Doxyfile` nástroje Doxygen, jak používat implementovaný program a jak upravovat zdrojový kód, aby byl vygenerován požadovaný diagram.

V šesté kapitole jsem představil demonstrační úlohu, která byla zvolena pro ověření funkčnosti implementovaného programu.

V sedmé kapitole jsem provedl srovnání vzorové dokumentace s dokumentací vygenerovanou implementovaným programem a nástrojem Doxygen. Je zde představena struktura obou dokumentací a jsou zde ukázány možnosti generovaných diagramů na příkladech.

V osmé kapitole jsem uvedl návrhy na další možná rozšíření implementovaného řešení, která by mohla zlepšit a zjednodušit jeho použitelnost.

Výstupem této práce je tedy implementovaný program s názvem `stdoc`, který je schopen na základě komentářů ve zdrojovém kódu v jazyce Structured Text generovat s pomocí nástroje Doxygen UML diagramy popisující chování programů a přehledně vizualizovat softwarový projekt ve formátu HTML. Zdrojové soubory samotného programu jsou rovněž zdokumentovány pomocí nástroje Doxygen, a jsou uloženy v příloze. Rešeršní částí práce jsem strávil přibližně 350 hodin. Celkově jsem psaním práce a implementací programu, včetně jeho testování a propojení s nástrojem Doxygen, strávil přibližně 800 hodin.

Hlavním přínosem implementovaného programu je ušetření doby potřebné na ruční tvorbu dokumentace, která byla v případě demonstrační úlohy odhadnuta na 20 hodin. Taková dokumentace proto nebývá ani běžně vytvořena, jelikož na její vyhotovení nezbývá dostatek financí a času. Automaticky vygenerovaná dokumentace umožňuje být rozšířena o ručně psaný obsah. Již vygenerované diagramy nelze ručně upravovat, zato jsou přehledné, a to i pro osoby, které nejsou autory zdrojového kódu. To může zjednodušit údržbu a rozšiřování projektu, obzvláště při předávání projektu jinému vývojáři, kterému by bez dokumentace trvalo mnohem déle se v projektu zorientovat. I když samotné psaní dokumentujících komentářů ve zdrojovém kódu demonstračních úloh mi trvalo přibližně 4 hodiny, tak samotné vytvoření diagramů implementovaným programem a nástrojem Doxygen trvá pouze několik sekund. Následné změny ve zdrojovém kódu se tak mohou snadno promítnout do dokumentace, kterou není nutno ručně upravovat a je tak ušetřen čas vývojáře na udržení jejího aktuálního stavu.

Literatura

[1] FOWLER, M. Destilované UML. Praha: Grada, 2009, 173 s. ISBN 978-80-247-2062-3.

[2] Structured Text: TM246. PDF Online. V2.0.0.0 EN. B&R Straße 1 5142 Eggelsberg, Austria, 2023. Dostupné také z: https://www.br-automation.com/cs/ke-stazeni/automation-academy/training-modules/control-technology/tm246-structured-text-st/.

[3] History of Visio. Online. 2011. Dostupné z: https://web.archive.org/web/20110418052835/http://visio.mvps.org/History.htm. [cit. 2023-11-24].

[4] Introduction to the Visio file format (.vsdx). Online. MICROSOFT. Microsoft Learn. 2015. Dostupné z: smallhttps://learn.microsoft.com/en-us/office/client-developer/visio/introduction-to-the-visio-file-formatvsdx. [cit. 2023-11-25].

[5] MICROSOFT. Office Add-ins platform overview. Online. 2023. Dostupné z: smallhttps://learn.microsoft.com/en-us/office/dev/add-ins/overview/office-add-ins. [cit. 2023-12-21].

[6] THE INTELIGENT SOLUTIONS COMPANY. Flow charts generator from Code : create code flowcharts, documenting source code, flowcharting program, reverse engineering, FLOWCHART SOFTWARE, visio flowcharts. Online. 2022. Dostupné z: smallhttps://www.fatesoft.com/s2f/upgrade_flowchart.htm. [cit. 2023-11-25].

[7] AIVOSTO OY. Aivosto. Analyze, Document and Flowchart Source Code. Online. 2023. Dostupné z: smallhttps://www.aivosto.com/. [cit. 2023-11-26].

[8] Apps/Dia/Python - GNOME Wiki!. Online. 2020. Dostupné z: smallhttps://wiki.gnome.org/Apps/Dia/Python. [cit. 2023-12-02].

[9] Apps/Dia/Python - GNOME Wiki!. Online. 2020. Dostupné z: smallhttps://wiki.gnome.org/Apps/Dia/Python. [cit. 2023-12-02].

[10] Code2flow - Pricing. Online. 2022. Dostupné z: smallhttps://code2flow.com/pricing. [cit. 2023-12-02].

[11] JGRAPH LTD. Draw.io Integrations. Online. 2023. Dostupné z: smallhttps://www.drawio.com/integrations. [cit. 2023-12-17].

[12] Draw.io - Diagramming in Confluence - Features. Online. 2023. Dostupné z: small`https://info.seibert-media.net/display/DRAWIO/draw.io+-+Diagramming+in+Confluence+-+Features`. [cit. 2023-12-17].

[13] THE GRAPHVIZ AUTHORS. Graphviz. Online. THE GRAPHVIZ AUTHORS. External Resources. 2024, 2024-04-03. Dostupné z: small`https://graphviz.org/`. [cit. 2024-04-25].

[14] Doxygen. Online. Doxygen: Getting started. 2024, 2024-04-08. Dostupné z: small`https://www.doxygen.nl`. [cit. 2024-04-25].

[15] JGRAPH LTD. Blog - Generate diagrams from code. Online. 2023. Dostupné z: small`https://www.drawio.com/blog/diagrams-from-code`. [cit. 2023-12-17].

[16] JGRAPH LTD. Export a diagram to various file formats. Online. 2023. Dostupné z: small`https://www.drawio.com/doc/faq/export-diagram`. [cit. 2023-12-17].

[17] JGRAPH LTD. Blog - Insert from text to create tree and entity diagrams. Online. 2023. Dostupné z: small`https://www.drawio.com/blog/insert-from-text`. [cit. 2023-12-17].

[18] JGRAPH LTD. Blog - Insert a diagram from specially formatted CSV data. Online. 2023. Dostupné z: small`https://www.drawio.com/blog/insert-from-csv`. [cit. 2023-12-17].

[19] About Mermaid | Mermaid. Online. 2023. Dostupné z: small`https://mermaid.js.org/intro/`. [cit. 2023-12-28].

[20] Pricing | Mermaid Chart. Online. 2023. Dostupné z: small`https://www.mermaidchart.com/pricing`. [cit. 2023-12-28].

[21] FLOWCHARTS SYNTAX | MERMAID. Pricing | Mermaid Chart. Online. 2023. Dostupné z: small`https://mermaid.js.org/syntax/flowchart.html`. [cit. 2023-12-28].

[22] Open-source tool that uses simple textual descriptions to draw beautiful UML diagrams. Online. 2023. Dostupné z: small`https://plantuml.com/`. [cit. 2023-12-30].

[23] JGRAPH LTD. Blog - Generate diagrams from code. Online. 2022. Dostupné z: small`https://www.drawio.com/blog/diagrams-from-code`. [cit. 2023-12-17].

[24] OMG® Unified Modeling Language® (OMG UML®). 2.5.1. 2017. Dostupné také z: small`https://www.omg.org/spec/UML/2.5.1/PDF`.

[25] B&R INDUSTRIAL AUTOMATION GMBH. B&R: Perfection in Automation. Online. 2024. Dostupné z:
small`https://www.br-automation.com/cs/`. [cit. 2024-01-01].

[26] B&R INDUSTRIAL AUTOMATION GMBH. B&R: Perfection in Automation. Online. 2024. Dostupné z: `https://www.br-automation.com/cs/produkty/software/automation-software/automation-studio/`. [cit. 2024-01-01].

[27] B&R INDUSTRIAL AUTOMATION GMBH. Programming. Online. 2024. Dostupné z: `https://www.br-automation.com/cs/produkty/software/additional-information/programming/`. [cit. 2024-01-01].

[28] V. AHO, Alfred; S. LAM, Monica; SETHI, Ravi a ULLMAN, Jeffrey. *Compilers, Principles, Techniques, and Tools*. 2. Addison Wesley, 2006. ISBN 9780321486813.

[29] LESK, M. E. a SCHMIDT, E. *Lex - Alexial Analyzer Generator*. Online. 7. edice, svazek 2B. Bell-labs.com, 1975. Dostupné z: `https://epaperpress.com/lexandyacc/download/lex.pdf`. [cit. 2024-01-03].

[30] LEVINE, John R. *Flex & bison: unix text processing tools*. Sebastopol: O´Reilly, c2009. ISBN 978-0-596-15597-1.

[31] LEVINE, John R.; MASON, Tony a BROWN, Doug. *Lex & yacc*. 2nd ed. Sebastopol: O'Reilly, 1992. ISBN 1-56592-000-7.

[32] Antlr/antlr4 at master. Online. Github.com. 2023, 2023-09-04. Dostupné z: `https://github.com/antlr/antlr4/tree/master`. [cit. 2024-01-03].

[33] HAVRANEK, Daniel. *Pokročilé generování syntaktických analyzátorů*. Brno, 2022. Bakalář- ská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Zbyněk Křivka, Ph.D.

[34] Lexical Analysis With Flex, for Flex 2.6.2:. Online. Lexical Analysis With Flex, for Flex 2.6.2. 2016, 2016-10-22. Dostupné z: `https://westes.github.io/flex/manual/index.html`. [cit. 2024-04-09].

[35] FREE SOFTWARE FOUNDATION, INC. Bison 3.8.1. Online. 2021. Dostupné z: `https://www.gnu.org/software/bison/manual/html_node/index.html`. [cit. 2024-04-11].

[36] Lexical Analysis With Flex, for Flex 2.6.2: Actions. Online. 2016, 2016-10-22. Dostupné z: `https://westes.github.io/flex/manual/Actions.html#Actions`. [cit. 2024-04-26].

[37] Lexical Analysis With Flex, for Flex 2.6.2: Start Conditions. Online. 2016, 2016-10-22. Dostupné z: `https://westes.github.io/flex/manual/Start-Conditions.html#Start-Conditions`. [cit. 2024-04-26].

[38] Parser Function (Bison 3.8.1). Online. 2021, 2021-12-10. Dostupné z: https://www.gnu.org/software/bison/manual/html_node/Parser-Function.html. [cit. 2024-04-27].

[39] Shift/Reduce (Bison 3.8.1). Online. 2021, 2021-12-10. Dostupné z: https://www.gnu.org/software/bison/manual/html_node/index.html. [cit. 2024-04-27].

[40] Reduce/Reduce (Bison 3.8.1). Online. 2021, 2021-12-10. Dostupné z: https://www.gnu.org/software/bison/manual/html_node/Reduce_002fReduce.html. [cit. 2024-04-27].

[41] Doxygen: Configuration. Online. Doxygen. 2024, 2024-04-08. Dostupné z: smallhttps://www.doxygen.nl/manual/config.html. [cit. 2024-05-05].

[42] Valgrind. Online. 2024. Dostupné z: https://valgrind.org/docs/manual/manual.html. [cit. 2024-05-05].

[43] PlantUML.com. Online. Use creole syntax to style your texts. 2024. Dostupné z: https://plantuml.com/creole. [cit. 2024-05-14].

Seznam symbolů a zkratek

PLC anglicky programmable logic controller, česky programovatelný logický automat

UML anglicky unified modelling language, česky unifikovaný modelovací jazyk,

XML anglicky extensible markup language, česky rozšířitelný značkovací jazyk

HTML anglicky hypertext markup language, značkovací jazyk pro tvorbu webových stránek

GNOME anglicky GNU Network Object Model Enviroment

PDF anglicky Portable Document Format, přenosný formát dokumentů

SQL anglicky Structured Query Language, strukturovaný dotazovací jazyk

CSV anglicky Comma-separated values, česky hodnoty oddělené čárkami

C4 Technika pro modelování architektury software. 4 písmena C značí 4 druhy diagramů: Context, Container, Component, Code. Česky kontextový, kontejnerový, komponent a kódu

CSS anglicky cascading style sheets, česky kaskádové styly

PNG anglicky portable network graphics, česky přenosná síťová grafika

ASCII anglicky American Standard Code for Information Interchange, česky americký standardní kód pro výměnu informací

SVG anglicky scalable vector graphics, česky škálovatelná vektorová grafika

CNC anglicky computer numeric control, česky číslicové řízení

HMI anglicky human-machine interface, česky rozraní člověk-stroj

BNF Backusova Naurova forma

UAMT Ústav automatizace řídicí techniky na Fakultě elektrotechniky a komunikačních technologií v Brně

GCC GNU Compiler Collection

DOT anglicky DAG of tommorow, česky DAG zítřka, kde DAG znamená orientovaný acyklický grag

EPL anglicky Eclipse Public Licence

GNU GPL anglicky GNU General Public License, česky obecná veřejná licence GNU

OEE anglicky Overall Equipment Effectiveness, česky Celková efektivnost zařízení

A Obsah elektronické přílohy

V tomto obsahu jsou vypsány jen nejrelevantnější adresáře a soubory. Ve složce `stdoc` se rovněž nachází zkompilovaný implementovaný program včetně zdrojových kódů. Příloha obsahuje také již automaticky vygenerovanou dokumentaci, kterou je však možné podle postupu v práci zmíněného znovu vygenerovat.

- `/`
  - `AutoDoc` ..... Adresář s automaticky vygenerovanou dokumentací
    - `html`
      - `index.html` ..... Hlavní stránka automaticky vygenerované dokumentace
  - `Documenation` ..... Adresář se vzorovou ručně vytvořenou dokumentací
    - `Project_Documentation.drawio` .. Ručně vytvořená dokumentace pro nástroj `diagrams.net`
    - `Project_Documentation.html` .... Ručně vytvořená dokumentace ve formátu HTML dokumentu
  - `eM` ..... Adresář s demonstračním projektem
    - `Logical` ..... Adresář se softwarovou částí projektu v Automation studio
      - `Doxyfile` ..... Předpřipravený konfigurační soubor pro nástroj Doxygen
      - `Package.pkg` . Soubor obsahující logickou strukturu projektu v Automation Studiu
      - `Documentation` ..... Adresář obsahující soubory pro dokumentaci projektu
        - `mainpage` ..... Přídavný obsah hlavní stránky dokumentace
        - `index.md` ..... Vygenerovaný `Markdow` dokument pro hlavní stránku dokumentace
      - `Libraries` ..... Adresář s knihovními funkcemi použitými v projektu
      - `VisuCtrl` ..... Adresář programu `VisuCtrl`
        - `VisuCtrl.st` ..... Zdrojový kód programu `VisuCtrl`
        - `VisuCtrl.md` ..... Vygenerovaný diagram programu `VisuCtrl` v textové podobě
        - `Actions` ..... Adresář s akcemi použitými v programu
          - `PageOee.st` ..... Zdrojový kód akce `aPageOee`
          - `VisuCtrl.md` ..... Vygenerovaný diagram akce `aPageOee` v textové podobě
  - `stdoc` ..... Repozitář s implementovaným programem a zdrojovými kódy
    - `stdoc` ..... Spustitelný implementovaný program
    - `Makefile` ..... Soubor pro kompilaci programu pomocí nástroje `make`
    - `PROGRAM_SYNTHETIC.st` . Syntetický zdrojový kód, který lze použít pro ověření správné činnosti analyzátoru
    - `PROGRAM_SYNTHETIC.md` ... Vygenerovaný `Markdown` soubor ze stejnojmenného zdrojového kódu
    - `grammar.png` ..... Grafická reprezentace implementovaného syntaktického analyzátoru
    - `doc` ..... Adresář s dokumentací programu `stdoc`
    - `tests` ..... Adresář s testovacími programy pro implementované struktury