České vysoké učení technické
Fakulta strojní
Ústav přístrojové a řídící techniky

Využití softwarového jazyka UML pro programování řídicích systémů PLC

Bakalářská práce

David Jakubec

Bakalářský program: Strojírenství
Studijní obor: Informační a automatizační technika
Vedoucí: Ing. Mgr. Jakub Jura, Ph.D.

Praha, červen 2018

Vedoucí práce:
Ing. Mgr. Jakub Jura, Ph.D.
Odbor Automatického řízení
Fakulta strojní
České vysoké učení technické v Praze
Technická 2
160 00 Praha 6
Česká republika
Jakub.Jura@fs.cvut.cz

Prohlášení

Prohlašuji, že jsem bakalářskou práci zpracoval samostatně a že jsem uvedl všechny použité prameny a literaturu, ze kterých jsem čerpal.

v Praze, dne 15.6. 2018

.....
David Jakubec

Abstrakt

Cílem této práce je navrhnout způsob generování kódu ze softwarového jazyka UML nebo SysML pro řídicí systémy PLC. V práci je nejdříve rešerše na toto téma. Popisuje co UML je a jaké diagramy je dobré na co používat. Dále jsou sestaveny modely pro některé školní úlohy pro předmět Řízení programovatelnými automaty. Tyto modely jsou poté podle návrhu napsány v jazyce Strukturovaného Textu podle toho, jak by mohly být generované.

Klíčová slova: PLC, UML, Strukturovaný Text, generování kódu

Abstract

The aim of this work is to design a method of generating code using software languages UML and/or SysML for PLC automation. This research primarily focuses on the following topics. Describing what UML is and which diagrams work best with it. Furthermore, designed models for a few school tasks on the subject of Programmable Controller Applications. Afterwards, these models are programmed in Structured Text based on how they could be generated.

Keywords: PLC, UML, Structured Text, code generating

Poděkování

Děkuji všem lidem kolem mě, kteří mě podporovali při psaní této bakalářské práce. Osobně bych pak chtěl poděkovat mému vedoucímu Mgr. Ing. Jakubovi Jurovi Ph.D., za jeho čas a chuť mi radit a za jeho připomínky k mé práci.

Seznam obrázků

Seznam použitých zkratek

AMD-V X86 Virtualization.

BIOS Basic Input-Output System.

CASE Computer Aided Software Engineering.

FB Function Block.

FBD Function Block Diagram.

GVL Global Variable List.

HMI Human-Machine Interface.

I/O Input/Outpu.

IDE Integrated Development Environment.

IL Instuction List.

LD Ladder Diagramm.

MBSE Model-Based Systems Engineering.

MMI Machine-Machine Interface.

OOP Object Oriented Programming.

PAC Program Authorisation Code.

PLC Programmable Logic Controler.

POU Program Organisation Unit.

SFC Sequential Function Chart.

SM State Machine.

ST Structured Text.

SysML System Modeling Language.

TON Timer On-delay.

UML Unified Modeling Language.

Obsah

Kapitola 1

Úvod

Jednoduché aplikace mohou být napsány jedním člověkem za krátkou dobu. Samotný člověk má plán aplikace v hlavě, má vizi jak by to mělo být a v klidu aplikaci vytvoří za pár dní. V případě velkých projektů to tak být nemůže, jelikož je nutností spolupráce v týmu. Člověk může mít v hlavě celý projekt, je-li dostatečně zdatný, ale musí ho umět ostatním lidem v týmu přednést. Proto se používá UML, což je modelovací jazyk pro návrh systémů. UML je grafický nástroj a člověk díky němu vidí celý modelovaný systém v nadhledu, pomáhá k orientaci a k pochopení jednotlivých částí krok po kroku. UML nabízí i způsob tvorby dokumentace, jelikož bývá často zanedbávána a je důležitá nejen pro ostatní spolupracovníky v týmu, kteří díky ní rychle pochopí, co daná část znamená, nebo k čemu slouží. Každému netriviálnímu modelovanému systému většinou nebude stačit jenom jeden model. Pro lepší představu zmíním příklad. Při návrhu domu architekt vypracuje plán domu a rozmístí objekty po místnostech. Pokud ale někoho zajímá, kudy vedou elektrické obvody ve zdech, v tomto plánu to neuvidí a musí si najít plán elektro, který tento návrh vyobrazuje.

Další součástí UML je možnost generování kódu. Generátory kódu jsou dnes normální věc, ale existuje jen několik diagramů z UML, které umí být vygenerovány. Generování je další věcí pro ulehčení práce, kde např. z diagramu tříd se vygeneruje statická struktura systému společně s proměnnými v jedné třídě. Ulehčení práce spočívá v tom, že programátor nemusí přepisovat jednotlivé řádky zvlášť a už se zabývá pouze programováním algoritmů. Tyto generátory ale neexistují pro generování jazyků podporované standardem IEC 61131-3, který tyto jazyky definuje.

Pokusy o zavedení UML do průmyslu zde již byly před dekádou[1] a nevím, zda firmy UML používají jako pomůcku při řešení průmyslových aplikací. V dnešní době je svět rychlejší a jsou více kladeny nároky na rychlost vypracování. UML by tomu mělo dopomoci a řešení se tak stanou realitou rychleji. Do budoucna je dobré počítat i s většími nároky na algoritmizaci vzhledem k éře průmyslu 4.0. Myšlenkou této práce tedy je pokusit se využít modelovací UML jako nástroj pro ulehčení práce a navrhnout metodiku modelování pro řízení průmyslovými programovatelnými automaty.

1.1 Cíle práce

Cílem této práce je vypracovat rešerši na počítačem podporované softwarové inženýrství (CASE) a sjednocený modelovací jazyk (UML) a jeho odvozený standard SysML. V rámci rešerše uvedu téměř všechny diagramy, ale pro vypracování úloh použiji jen několik vybraných z nich, kde důvodem je nepoužitelnost některých pro programování PLC, nebo programování vůbec. Proberu a aplikuji nejvhodnější diagramy pro modelování projektování průmyslových řešení a z nich následně implementuji kód do vybraného vývojového prostředí pro programování PLC. Pokusím se o dodržení nového vydání standardu IEC 61131-3, ve které se nachází pasáž o objektově orientovaném programování Structured Textu. Implementace by měla být důsledkem vygenerovaného kódu z použitých diagramů a zjistit, zda-li je to možné a viabilní.

Kapitola 2

CASE

CASE je zkratka pro Computer-Aided System (Software) Engineering, tedy vývoj systému (software) s využitím počítačové podpory. Jelikož člověk lépe chápe obrázek, tak se používají většinou nástroje grafické, které nám i umožní vidět v nadhledu celý navrhovaný systém. Je určený pro návrháře, programátory, manažery a další lidi spolupracující na projektu. Používají se k ulehčení práce, lepšímu strukturování, lepší spolupráci s větším počtem lidí v týmu účastnicích se na projektu, dále pro nižší chybovost, lepší dokumentaci, jelikož často bývá zanedbávána, a CASE nástroje ji dokáží i generovat. [2]

2.1 Historie

začaSoftwarové inženýrství se začalo objevovat koncem 60. let minulého století, jelikož nastala softwarová krize. Vývoj hardware byl daleko rozsáhlejší, jak vývoj software. Do té doby počítače sloužily pouze jako nástroje pro vědecko-technické výpočty, které požadovaly spíše preciznost, než propracovanost software. V roce 1968 konference NATO začala s popularizací softwarového inženýrství a pár let na to byly vyvíjeny strukturované a objektové metodiky pro analýzu. S tímto příchodem se začaly vyvíjet první generace nástrojů CASE, které se nazývaly Lower Case, které jsou určeny například ke generování kódu. Myšlenka CASE byla taková, že programátoři nebudou tolik zapotřebí a pomocí nástroje CASE se vygeneruje plný funkční kód, bez potřeb dalších úprav. Byla to myšlenka hezká, avšak utopistická. Tato technologie udržuje dodnes spíše na úrovni návrhu, než na úrovni generování plnohodnotného zdrojového kódu. Hlavním problémem bylo, že

při změně návrhu, musel být změněn i kód a při změně kódu musel být změněn návrh, což bylo problémové a stálo to hodně úsilí. Tento problém se časem v souladu s vývojem strukturovaných a objektově orientovaných metod začaly odstraňovat. Dnes se stále nedokáže vygenerovat funkční kód z vytvořeného modelu, ale dokáže se vygenerovat šablona, do které poté programátor doplní vše potřebné k plné funkčnosti kódu. [2] [3]

2.2 Komponenty CASE nástrojů

Centrální úložiště

Uchovává informace o všech objektech navrhovaného systému, zaručuje znovu použití informace v libovolném dalším kroku projektování. Slouží také jako datový slovník a poskytuje mechanismus pro sdílení informací ve skupině vývojářů. [2] [4]

Integrované CASE nástroje

Tyto nástroje jsou užitečné ve všech fázích návrhu vývoje životního cyklu systému, od souhrnu požadavků k testování a dokumentaci. Dělí se na Lower a Upper CASE nástroje. [4]

Upper CASE nástroje

Používají se ve fázi plánování, návrhu a analýze vývoje životního cyklu systémů. [4] [2]

Lower CASE nástroje

Používají se při implementaci, testování a údržbě. [4] [2]

Nástroje pro tvorbu diagramů

Umožňuje návrh a kontrolu datového toku a struktury v grafické podobě. [4]

Nástroje pro dokumentace

Umožňují automatickou tvorbu, modifikaci a aktualizaci návrhové dokumentace a technických zpráv, podle požadované nebo navržené šablony, aby vyhovovala standardům, nebo požadavkům uživatele. [4]

Nástroje pro analýzu

Tyto nástroje pomáhají shromažďovat požadavky, automaticky kontrolují případné

nesrovnalosti, nepřesnosti ve schématech, propouštění dat nebo chybné vynechání. [4]

Generátor kódu

Metoda, která umožňuje generování kódu z diagramů, založené na ontologickém modelu do programovacího jazyka obsahující jazykové koncepty a jejich vztahy. [3]

Kapitola 3

UML

UML je standardní jazyk pro specifikaci, vizualizaci, konstrukci a dokumentaci artefaktů softwarových systémů. Patři mezi grafické jazyky, které se nejčastěji používají při návrhu velkých projektů. Často bývá využíván při komunikaci programátor-zákazník, kde zákazník lépe porozumí kódu, který je reprezentován obrázkově. Po vytvoření modelu se obě strany díky tomuto prostředku lépe domluví a to vede k úspěšnosti projektu a ke spokojenosti zákazníka. Pro jednoduché a jednoúčelové aplikace tento přístup vůbec nemusí být efektivní, ale pokud budeme uvažovat velký projekt, tak právě UML přináší velkou výhodu. V grafické reprezentovatelnosti a lepší přehlednosti, jednoduchost nastavování globálních proměnných pro systém a další. Díky tomu, pak přináší UML další výhodu a tím je generování šablony podle určených proměnných v UML společně s vazbami, do které programátor vyplní potřebnou sekvenci. [5] [6]

3.1 Historie UML

UML začali vyvíjet Grady Booch a Jim Rumbaugh ve své firmě Rational Software, která se později stala součástí firmy IBM. Spojovali různé metodiky, OMT (Object Modeling Technique), což byla práce Booche a po vstupu do firmy Ivara Jacobsona v roce 1996 s jeho metodologií OMSE (Object-Managed Software Engineering), dali dohromady již dnes známý UML. O rok později tento návrh přijala standardizační organizace OMG (Object Management Group) jako standard UML verze 1.1. UML 1.0 je označení verze, kterou firma Rational Software poslala komisi a s každým rokem přišla novější verze,

která upřesňovala specifikace. Od roku 2001 organizace OMG připravuje verzi 2.0, která je rozšiřující verzí 1.0 a přináší i SysML. [7]

3.2 Obsah UML

Celé UML má 2 základní pilíře a 2 sub-pilíře, na kterém je celý tento modelový jazyk postaven. Diagramy struktury a diagram behaviorální, tyto dva diagramy se dále dělí na statický a dynamický. Pokud je diagram behaviorální, zobrazuje aktivitu, nebo sekvenci, nebo jiný náznak chování. Dynamický je tehdy, když se v diagramu uvažuje časová složka. [8]

3.3 Základní terminologie

V této sekci jsou vysvětleny základní pojmy, podrobněji bude vysvětleno dále v práci.

- Artefakt

  Artefakt jsou veškeré prvky, které tvoří statickou část modelu, jako jsou třídy, rozhraní, use case a komponenty, nebo tvoří dynamickou část modelu jako interakce, stavy a aktivity, nebo jsou v organizační části systému jako balíčky (package) a takové, které vysvětlují účel, mezi které patří popis, anotace a poznámka. [10] [11]

- Vztahy

  Vztahy jsou prvky, které spojují dva nebo více artefaktů a přidávají sémantickou informaci do modelu. Může se použít více jak jeden vztah v diagramu popisující strukturu. Příkladem vztahů je asociace, agregace, generalizace, přechod a další, které jsou vysvětleny dále. [10] [11]

- Diagramy

  Jsou kombinací vztahů a artefaktů. Diagramů je v celém UML mnoho a vyobrazují grafický celek návrhu. Obvykle se používá více jak jeden diagram k popisu systému. Každý diagram dokáže vystihnout systém z jiného pohledu. Pohledy lze rozlišit na statický, dynamický a uživatelský. [10] [11]

Diagram	Stručný popis	Typ
1. Diagram případu užití	Poskytuje přehled systému z uživatelské perspektivy	Staticky behaviorální
2. Diagram aktivit	Modeluje procesy a workflow	Staticky behaviorální
3. Diagram tříd	Reprezentuje třídy, jejich definice a vztahy	Staticky strukturovaný
4. Sekvenční diagram	Modeluje interakci mezi objekty založené na časové posloupnosti	Dynamicky behaviorální
5. Diagram přehledu interakcí	Prezentuje přehled interakcí v systému. Dále pomáhá pochopit, jak na sobě UML diagramy závisí.	Staticky behaviorální
6. Diagram komunikací	Zobrazuje jednotlivé objekty, které mezi sebou interagují. Je podobný sekvenčnímu diagramu.	Staticky behaviorální
7. Objektový diagram	Zobrazuje objekty a jejich propojení.	Dynamicky behaviorální
8. Diagram složených struktur	Modeluje chování komponentu nebo objektu, zobrazuje uspořádání, vztahy a instance při spouštění.	Dynamicky behaviorální
9. Stavový diagram	Zobrazuje životní cyklus objektu, stav ve kterém se nachází a podmínky pro změnu stavu.	Dynamicky behaviorální
10. Diagram komponent	Zobrazuje strukturovaně vymodelované komponenty a jejich vztahy.	Staticky strukturovaný
11. Diagram rozestavení	Modeluje architekturu hardwarových komunikačních uzlů.	Staticky strukturovaný
12. Diagram balíčků	Reprezentuje subsystém a pole organizace systému. Dokáže oddělit jednotlivé entity od sebe	Staticky strukturovaný
13. Diagram časování	Modeluje způsob, jakým se mění stavy objektu v čase.	Dynamicky behaviorální

[9]

Obrázek 3.1: Tabulka přehledu diagramů v UML

3.4 Diagramy struktury

Diagramy struktury, nebo také statické diagramy jsou grafickou reprezentací navrhovaného systému v UML, kde spojením jednotlivých artefaktů systému vazbami se vytvoří struktura, nebo funkce. Zobrazují statickou strukturu systému nebo runtimovou architekturu. [12]

3.4.1 Diagram tříd

Diagram tříd, je nejběžnější diagram používaný v UML, který dává statický nadhled nad celým programovaným systémem. Tento diagram se používá naostro a slouží již jako šablona, ale i jako prostředek pro vytvoření dokumentace. Na obrázku 3.2 je znázorněné, jak vypadá třída v UML. První kolonka je pro název třídy, bez diakritiky. Druhá kolonka náleží atributům a pro třetí jsou vyhrazeny funkce (metody). V některých CASE nástrojích je i čtvrtá kolonka, která zahrnuje signály, nebo také zprávy, které se mezi třídami posílají. Atributům a funkcím je dán i datový typ, to je znázorněné znaménkem před jejich názvy. Mínus (-) je pro privátní, Plus (+) je pro veřejný, křížek (#) pro chráněný a vlnovka je (˜) viditelná pouze v rámci balíku.

Jednotlivé třídy mají mezi sebou vztahy, pokud existuje asociace mezi třídami, je tím popsán vztah mezi instancemi těchto tříd a popisuje lépe celý systém. V diagramu tříd máme několik druhů vztahů, které jsou popsány níže. Tyto níže zmíněné vztahy nejsou ale unikátním pro diagram tříd a jsou používány i jinými diagramy, jako diagram komponent, balíčků aj. [13] [14] [15]

- Asociace

  Asociace je vztah, který je mezi klasifikátory a ukazuje, že jednotlivé instance mohou být buď provázány společně, nebo zkombinovány logicky, nebo fyzicky do agregace. Nejčastějším typem asociace je binární, kde má dva konce a jedna třída je provázaná s druhou a je reprezentována jako čára. Relace může být i n-ární a je značena rozvětvením. Asociace může být v UML pojmenována, mít přiřazena jména rolí, mohutnost, viditelnost a další jiné vlastnosti. Existují čtyři různé druhy sdružení: jednosměrný, obousměrný, agregace a reflexivní. [16] [15] [14] [17]

- Agregace

Agregace je slabším vztahem mezi objektem a jeho částmi. Objekty, které se reprezentují jako celek jsou nazývány agregačními objekty. Jeho části jsou konstituční objekty. Seskupený objekt může existovat bez jakýchkoliv konstitučních objektů. Pro představu relace počítač - tiskárna. Tiskárna je agregační objekt, a může existovat bez existence počítače, k tiskárně může být připojen jiný počítač. Konstituent může být součástí více seskupení. Značení je prázdným diamantem. [18] [17]

- Kompozice

  Kompozice vyjadřuje silnější vztah mezi objekty a jeho částmi, na rozdíl od agregace. Je jí hodně podobná, ale reprezentují celek - složení - kompozitní objekt, kde jsou jeho části nazývány komponentními objekty. Jeho vlastností je, že složený objekt nemůže existovat bez svých komponent a komponentní objekt může být součástí pouze jedné kompozice. Pro představu, hotel může existovat bez pokojů, ale pokoj bez hotelu nikoliv. Značení je začerněným diamantem. [18] [17]

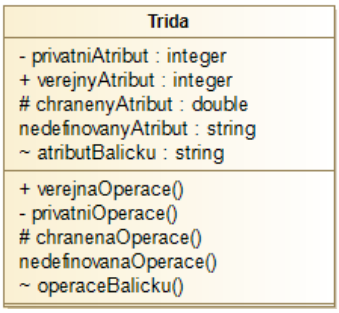
- Generalizace

  Dědičnost je hierarchický vztah mezi třídami. Jeden ze dvou příbuzných tříd (podtříd), je považován za specializovaný druh druhého, který je mu nadřazený. Nebo každý potomek je instancí rodičovské třídy. V UML se značí jako trojúhelník. [16] [15]

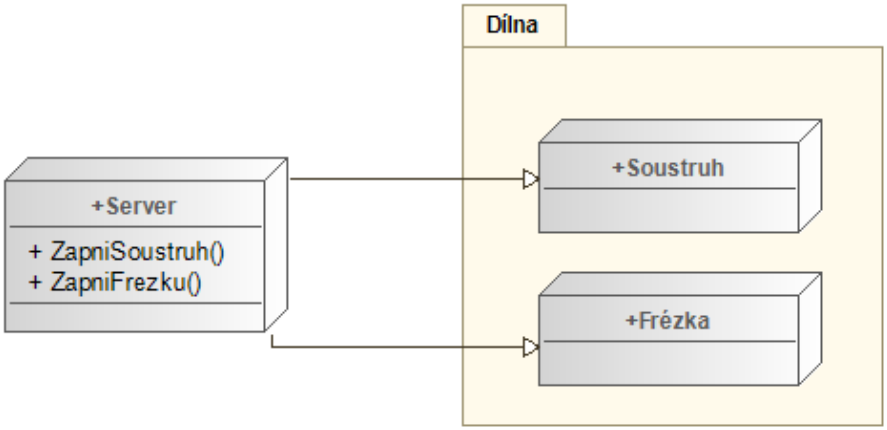
  Vztah mezi mateřskou a dceřinou třídou je realizace - nebo také vztah mezi třídou a rozhraním. Jsou-li dvě třídy spojené vztahem realizace, označujeme fakt, že třída implementuje všechny operace z daného rozhraní. Realizace může být také vztah mezi třídami, rozhraní, komponenty a balíčky, které spojuje klientský prvek s dodavatelským prvkem. [18] [19] [17]

- Závislost

  Závislost je slabší forma vztahu, která naznačuje, že jedna třída je závislá na té druhé. [17] [13] [19]



Obrázek 3.2: Diagram tříd



Obrázek 3.3: Diagram nasazení

3.4.2 Diagram komponent

Je rozdílný od ostatních diagramů, protože nepopisuje funkčnost systému, ale komponenty použité pro jeho funkčnost. Z tohoto pohledu jsou diagramy komponent používány pro vizualizaci fyzických komponent. Pod pojmem fyzická komponenta si můžeme představit ve vyšším programovacím jazyce jako knihovnu, balíček, soubor nebo spustitelný soubor aj. [9] [20]

3.4.3 Diagram nasazení

Diagram nasazení (deployment), vyobrazuje architekturu hardwarových uzlů, procesorů systému a umožní nahlédnout do uzlů na kterých komponenty software budou pracovat. Je užitečný při zobrazování více zařízení, které spolu budou komunikovat. Příklad na obrázku 3.3, je znázorněná komunikace mezi dílnou a serverem, kde dílna obsahuje soustruh a frézku. Ze serveru se posílá informace o zapnutí soustruhu nebo frézky. [9]

3.4.4 Diagram složených struktur

Diagram složených struktur (composite structure) reprezentuje runtimovou architekturu skupiny objektů a komponentů, zahrnující jejich rozhraní a uskutečnění. Jeho současné využití je omezené pouze na strukturu runtimových prvků v systému. [9]

3.4.5 Diagram balíčků

Diagram balíčků (package), se typicky používá pro sdružení souvisejících tříd. Umožňuje sdružit elementy modelu do strukturovaných skupin, které se nazývají balíčky. Například jeden balíček může být složený z více balíčků a ten zase z více balíčků. Jednotlivé balíčky se spojují závislostmi mezi nimi, které nejsou povinné a nejsou ani tak důležité, jako samotné balíčky, jelikož se s nimi nedá asociovat žádnou funkčnost systému. [9] [21]

3.5 Behaviorální diagramy

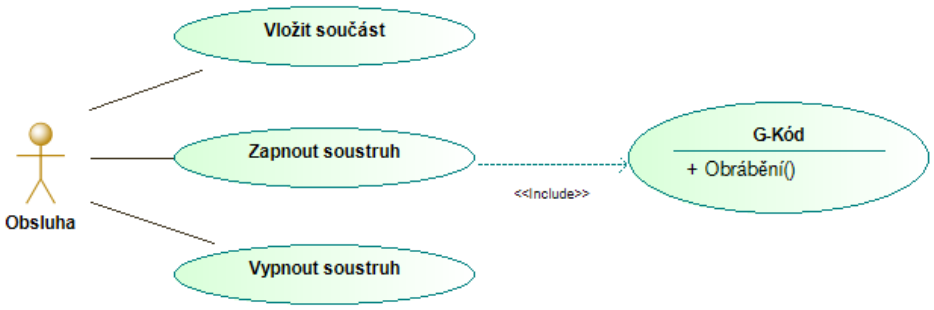
Diagramy chování popisují dynamickou strukturu systému.

3.5.1 Diagram Aktivit

Diagram aktivit se používá pro modelování procedurální logiky, procesů a zachycení pracovního postupu. Je tedy podobný jako vývojový diagram. Je reprezentován sekvencí kroků, které jsou zakreslovány jako akce nebo vnořené aktivity. Akce jsou dále nedělitelné kroky. Vnořené aktivity volají jiné procesy (aktivity) a mohou být reprezentovány dalším diagramem aktivit. Aktivita přebírá na vstupu data jako parametr a předává data na výstup. Velkou výhodou diagramu aktivit je schopnost vyobrazit závislost mezi aktivitami a pomáhá zmapovat aktivity korespondující k jejich akčním členům. Umí vyobrazovat i co se děje paralelně v systému, jelikož mají schopnost spojovat/rozdělovat proces do více větví. Jelikož ukazují aktivitu a pořadí, ve kterém se tyto aktivity spustí, tak je považován za diagram chování, ale nedokážou ukázat, v jakém čase aktivita nastane. Dá se říci, že diagramy aktivit jsou genericky se chovající vývojové diagramy a nejsou považovány za diagramy ukazující dynamiku, jako je například sekvenční. Povaha je tedy staticky-behaviorální. [9]

3.5.2 Diagram případu užití

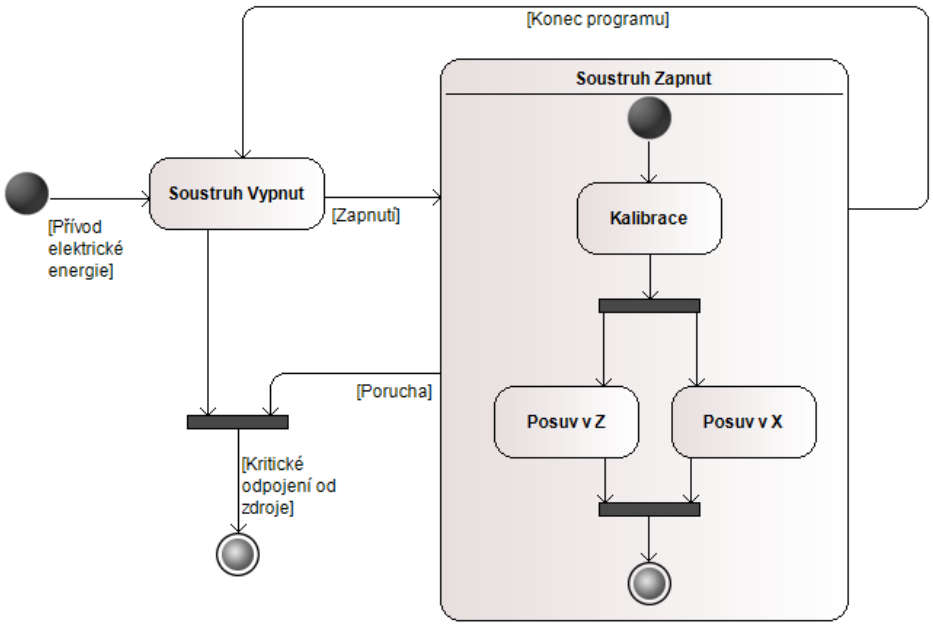
Diagram případu užití (use case), je diagram, který zobrazuje chování systému tak, jak ho vidí uživatel. Tento diagram se prvotně používá k vyobrazení případů užití, korespondujících sektorů a jejich interakcí. Vypovídá o tom, co má systém umět, ale neříká jak to bude dělat. Při návrhu systému se tento diagram nakreslí jako první, jelikož je jednoduchý a druhý člověk díky němu dostane povědomí o tom, co systém má umět. Povaha Use case diagramu je staticky-behaviorální, jelikož pomáhá zorganizovat a vyhodnotit požadavky na systém. Behaviorální aspekt požadavků není v diagramech případu užití viditelný, jelikož vztahy mezi dvěma případy užití, nebo mezi aktérem a případem užití není reprezentován čas a je tedy proto kategorizován jako statický diagram. Diagram případu užití má dvě části, aktéra a případ užití. Případ užití je reprezentování jedné akce, nebo sady akcí, které dosáhnou určitého cíle. Aktér je role, která komunikuje s jednotlivými případy užití, můžeme si ho představit jako člověka používající HMI, jako server nebo i jako čas a vždy inicializuje případ užití. Příklad je na obrázku 3.4, kde je obsluha stroje a ta má na výběr mezi několika akcemi, které může vykonat. Vložit součást, zapnout/vypnout soustruh. Zapnutí soustruhu má v sobě G-Kód - program pro obrábění, které je znázorněno funkcí veřejnou funkcí Obrábění(). [9] [22]



Obrázek 3.4: Diagram případu užití

3.5.3 Stavový diagram

[h] Stavový diagram (state machine) popisuje chování objektů jako stavy, ve kterých se objekt může nacházet. Chování je vyobrazeno přechody mezi jednotlivými stavy a jak objekt může reagovat na akce. Stavový diagram ukazuje všechny stavy, ve kterých se



Obrázek 3.5: Stavový diagram

může objekt nacházet a v UML umí reprezentovat čas precizně a v reálném čase. Pokud si položíme otázku "co se stane v konkrétním čase?", tak se můžeme podívat na diagram, který nám to řekne. Jelikož je uvažován čas, tak je dynamicky-behaviorální a je ideální pro modelování systému v reálném čase. Ukazuje také chování jednoho celého objektu a vyobrazuje cyklus jednoho objektu, jak v čase mění stavy jako odpověď na příchozí zprávy. Demonstrace využití je na obrázku 3.5, kde je použitý zjednodušený model soustruhu. [9]

3.5.4 Diagram přehledu interakcí

Tento druh diagramu je nadřazený diagramu sekvenčnímu a jak již z názvu vyplývá, popisuje přehled interakcí v diagramech na vyšší úrovni. Ukazují závislosti a toky mezi případy užití. Sekvenční diagramy též zobrazují interakce, jsou ale používané pro detailnější pohled. Diagram přehledu interakcí zahrnují reference ke korespondujícím sekvenčním diagramům, tedy jeden diagram může zahrnovat více sekvenčních diagramů a také ukázat závislosti mezi těmito sekvenčními diagramy. Může také odkazovat na ostatní diagramy jako je diagram aktivit, nebo případ užití a poskytnout přehled, jak jsou tyto diagramy na sebe vázané. Jiné diagramy, které jsou obsaženy v diagramu přehledu interakcí se nazývají vnořené diagramy interakcí a mají klíčová slova podle typu diagramu, kde klíčové slovo

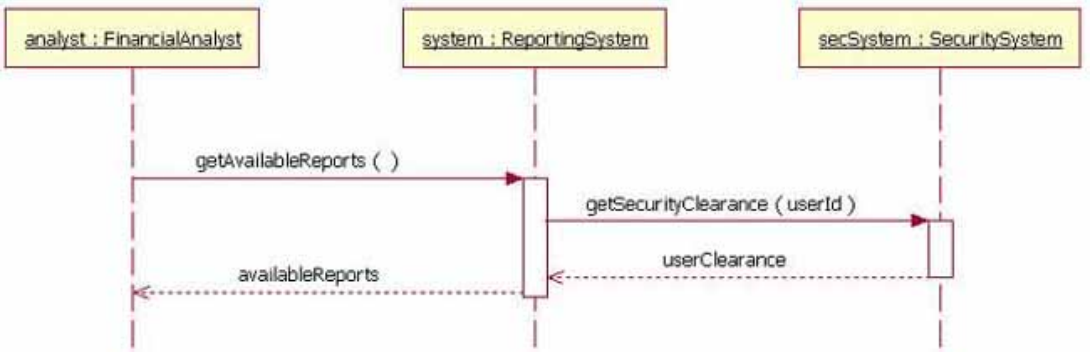
je anglickou zkratkou použitého diagramu. Nebo klíčové slovo s výskytem interakcí, které je „ref“. Prvky diagramu jsou uzly a přechodové hrany. Uzly jsou počáteční a koncové, které indikují stav. kde začnou a kde skončí. Vždy pro jeden diagram je jeden počáteční a jeden koncový. Další uzly jsou rozhodovací, které mají vždy jeden vstup a více výstupů. Většinou do rozhodovacího uzlu přijde otázka, na kterou se odpoví binárně „ano/ne“ a podle toho je cesta diagramu vyhodnocena. Další uzel je rozvětvení a sloučení, což znamená, že pokud chceme paralelně vyhodnocovat cestu, musíme jí nejdříve rozdělit. Jinými slovy, pokud je splněna podmínka a chceme udělat akci A, a zároveň akci B, pak je toto jediný způsob, jak toho docílit. Sjednocení funguje tak, že pokud chceme udělat akci C, potřebuje být splněna podmínka A, nebo podmínka B. [9] [23]

3.5.5 Sekvenční diagram

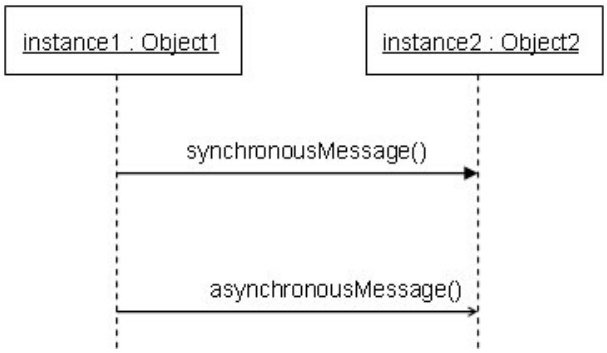
Sekvenční diagram, vyjadřuje interakci mezi třídami a jako dokumentace pro chování v rámci případů užití. Jelikož dokáží vyjádřit co se děje "uvnitř"případu užití, tak je široce používán jak systémovými inženýry, tak i obchodními analytiky. Sekvenční diagramy reprezentují podrobnější interakci mezi aktérem a systémem, nebo mezi kolaborujícími objekty v rámci daného časového bloku. Nejsou zobrazovány informace o tom, co se stalo před započetím interakce a co se stane po uplynutí určitého časového bloku. Zprávy v sekvenčním diagramu můžou mít předpoklady a následné podmínky, tak tyto podmínky nejsou v diagramu přímo viditelné. I přes tuto limitaci je možné čas zobrazit v sekvenčním diagramu daleko lépe než v diagramu aktivit. Díky této vlastnosti je sekvenční diagram považován za dynamicky-behaviorální.

Interakce mezi třídami je pomocí zpráv, které se posílají na tak zvané čáry života (z anglického "lifelines"). Není tak důležité, co tyto zprávy obsahují, tak jako v jakém pořadí se posílají, tedy sekvenčně. Z diagramu vidíme, že horizontálně máme třídy, mezi kterými posíláme zprávy a vertikálně máme tyto čáry života, které reprezentují časovou sekvenci, jak se zprávy posílají.

Správná notace zpráv, které se posílají je, že šipka mezi čárou života 2 a 3 musí být o něco níže, než šipka mezi čárou života 1 a 2, pokud je respondentem na zprávu. To je z toho důvodu, že zpráva mezi 2 a 3 se nemůže poslat dříve, než zpráva mezi 1 a 2, jelikož je na ní závislá, a jak bylo zmíněno výše, tak čáry života reprezentují "čas". Zprávy se dělí



Obrázek 3.6: Sekvenční diagram
[24]



Obrázek 3.7: Synchronní a asynchronní zpráva
[24]

na synchronní a asynchronní. Asynchronní zprávy nečekají na odpověď, vytvoří se další vlákno a třída, která poslala zprávu pokračuje v tom co dělala předtím, načež třídy, které vyslaly synchronní zprávu čekají na navrácení hodnoty a poté pokračují dál. Grafický rozdíl mezi synchronní a asynchronní zprávou je na obrázku 3.7. [24] [9]

Kapitola 4

SysML

SysML je o něco složitější, jelikož je obohacený o další dva pilíře, na kterých je postaven, a tím jsou diagramy požadavků, kde se uvedou podmínky pro správnou funkcionalitu a diagram parametrů, které vyjadřují matematické vztahy mezi jednotlivými kousky navrhovaného systému. [11] [25]

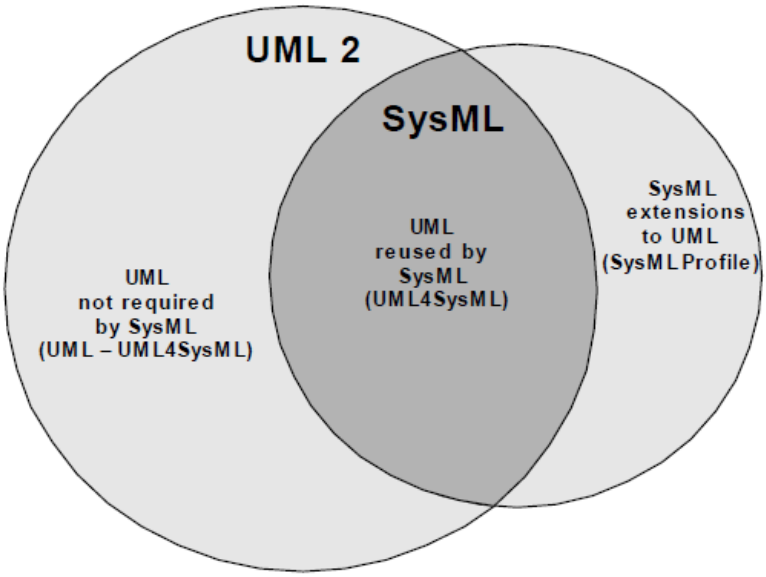
4.1 SysML

SysML používá určitou část UML a poskytuje další rozšíření potřebné k adresování požadavků v UML pro systémové inženýrství. Vizualizace relace mezi UML a SysML je na obrázku 4.1. Podporuje specifikace, návrh, analýzy a další ze širokého spektra. Tyto systémy mohou zahrnovat hardware, software, informace, procesy a různé doplňky. SysML je tedy technologie umožňující Modelově Založeného Systémového Inženýrství (MBSE z anglického názvu Model-Based Systems Engineering). [8]

4.1.1 Diagram struktury

Diagram definice bloku (BDD) - Tento diagram obsahuje bloky, respektive komponenty, které se v systému nachází a přináší do systému nějakou hodnotu. Můžou být zde i bloky, které jsou externí, čili z knihovny, a mají vnitřní chování definované pomocí IBD. Jednotlivé bloky se mohou spojovat vazbami z klasického UML.

Vnitřní blokové schéma (IBD) - Reprezentuje vnitřek bloku z BDD, který ukazuje vztah jednoho bloku s vazbami, kterými jsou spojeny. Tím dostaneme podrobnější pohled



Obrázek 4.1: Vennův diagram relace UML a SysML

na chování a vztah mezi vazbami v bloku. [26]

4.1.2 Diagram chování

Sekvenční diagram (SD) - Neměnné od UML2.0

State machine - Taktéž převzato z klasického UML. Zobrazuje stavy a jejich podmínky přechodu ve kterých může komponenta být.

Diagram aktivity - Derivát ze známého vývojového diagramu. Zobrazuje tok dat, což je tok mezi dvěma akcemi a zobrazuje i tok řízení. Používá se především v časném návrhu systémů, kde není jisté, se kterými prvky struktury se má počítat. Také je dobrý při vylepšování use case diagramů a funkčních požadavků.[26]

4.1.3 Diagram požadavků

Zde se píšou veškeré specifikace a požadavky pro celý systém. Vlastní dva fundamentální atributy. Jeden z nich je ID, který může být datový typ jako integer, string nebo jiné. Druhý je textové pole, do kterého se vyplní požadavky pro chování systému, kde pak další inženýr tyto požadavky vyplní parametry.[26]

4.1.4 Diagram parametrů

Do parametrického bloku se píší rovnice systému, které se spolu provážou s dalšími vlastnostmi modelu. Může se tímto popsat dynamické chování celého systému.[26]

Kapitola 5

Standard IEC 61131 pro průmyslové automaty

IEC je anglickou zkratkou pro mezinárodní elektrotechnickou komisi. Což je nezisková, zdánlivě připomínající vládní společnost, založená v roce 1906. Členové IEC jsou národní výbory, které jmenují odborníky a delegáty z průmyslu, vládních orgánů a akademické obce, aby se účastnili na technické normalizaci pro elektrické a elektronické výrobky, systémy a služby. [27]

5.1 IEC 61131

IEC 61131 je standard pro průmyslové programovatelné automaty. Obsahuje celkem 9 vydaných částí, které kladou nároky průmyslové programovatelné automaty.

1. IEC 61131-1: Obecné informace

   Obsahuje obecné informace pro programovatelné automaty a jejich asociované periferie, jako jsou nástroje pro odladění, rozhraní člověk-stroj aj., které jsou používány v řídící technice a jiné průmyslové procesy. [28]

2. IEC 61131-2: Požadavky na zařízení a testování

   Specifikuje funkční a elektromagnetickou kompatibilitu požadavků a s tím spojené ověřovací testování pro jakékoliv produkt, kde je primárním cílem zajišťovat funkční průmyslové zařízení, zahrnující PLC, PAC a/nebo jejich asociované periferie,

které mají být použity jako zařízení pro řízení strojů, automatické výroby nebo průmyslových procesů. [29]

3. IEC 61131-3: Programovací jazyky
   Upřesňuje syntaxi a sémantiku jako jednotný soubor programovacích jazyků pro programovatelné automaty. Tento soubor zahrnuje dva textové jazyky - Instruction List (IL) a Structured Text (ST), a dva grafické jazyky, Ladder Diagram (LD) a Function Block Diagram (FBD). [30] [31]

4. IEC 61131-4: Podpora uživatelů
   Pomáhá uvést koncového uživatele PLC do IEC 61131 a asistuje mu pro výběr a specifikaci jejich vybavení podle IEC 61131. [32]

5. IEC 61131-5: Komunikace
   Specifikuje komunikační aspekty programovatelných automatů. Specifikuje z pohledu jak jakékoliv zařízení může komunikovat s počítačem jako se serverem a jak počítač dokáže komunikovat s jakýmkoliv zařízením. [33]

6. IEC 61131-6: Funkční bezpečnost
   Specifikuje požadavky podle IEC 61131-1, které ale jsou určeny pro logický subsystém elektronického systému z hlediska bezpečí. [34]

7. IEC 61131-7: Programování fuzzy řízení
   Definuje jazyk pro programování řízení fuzzy aplikací používané programovatelnými automaty. [35]

8. IEC TR 61131-8: Pokyny pro aplikaci a implementaci programovacích jazyků
   Platí pro programovací jazyky pro programovatelné automaty definované v IEC 61131-3. Obsahuje různé pokyny. [36]

9. IEC 61131-9: Drobné digitální komunikační rozhraní pro maalé snímače a ovládací členy
   Rozšiřuje tradiční digitální rozhraní pro vstupy a výstupy podle IEC 61131-2 pro komunikaci mezi dvěma koncovými body. [37]

Kapitola 6

Řešení pro generování dle standardu IEC 61131-3

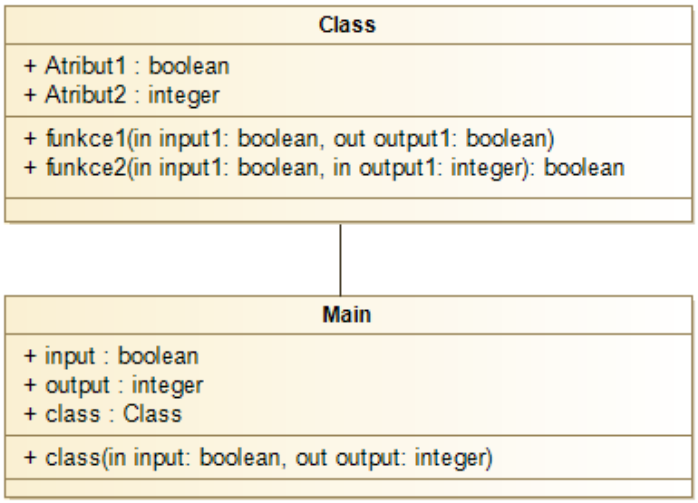
6.1 Strukturovaný text

Strukturovaný text spadá pod vyšší programovací jazyky a je jedním z textových jazyků standardu IEC 61131-3. Jazyk seznamu instrukcí (Instruction List), by šel použít taktéž, ale není objektově orientovaný, což je podmínkou pro řešení těchto úloh. Navíc je méně přehledný a o něco těžší na použití. Ve třetím vydání standardu 61131-3 byla přidána kapitola o objektově orientovaném programování Structured Textu. [30] [31]

6.2 Návrh způsobu generování z diagramu tříd

Pro názornost je použita jednoduchá třída Class z obrázku 6.1 obsahující 2 atributy a 2 funkce. Funkce s názve funkce1, je funkční blok, funkce s názvem funkce2 je metoda, která má návratovou hodnotu boolean. Hlavním rozdílem v metodě a funkčním bloku je to, že metoda vrací právě jednu nastavenou návratovou hodnotu (bool, int, string, aj...), kdežto funkční blok nemá návratovou hodnotu, ale má hodnoty výstupní. Výstupních hodnot může být více jakéhokoliv datového typu. V UML se vstupy/výstupy mohou specifikovat právě tím způsobem, jak je nakreslené na obrázku 6.1. funkce, která je specifikovaná funkce1(in input1: boolean, out output1 : integer).

Každý program obsahuje vždy metodu Main (v ST program main) a slouží jako vstupní brána do programu. Tímto způsobem by se tedy mělo při návrhu vždy začínat a poté definovat ostatní třídy - funkční bloky. V mainu pak vytvořenou třídu inicializovat a zadat vstupní/výstupní hodnoty ve funkčním bloku v oblasti pro funkce.



Obrázek 6.1: Názorný diagram tříd pro návrh řešení

Následné potencionální generování je v tomto bodě problematické a diagramy tříd nemají potřebné parametry pro funkční bloky. Jelikož funkční blok obsahuje vstupní, výstupní a interní proměnné, není jednoduchý možný způsob z diagramu tříd podle klasického UML/SysML generovat deklaraci proměnných podle požadavků jazyka ST.

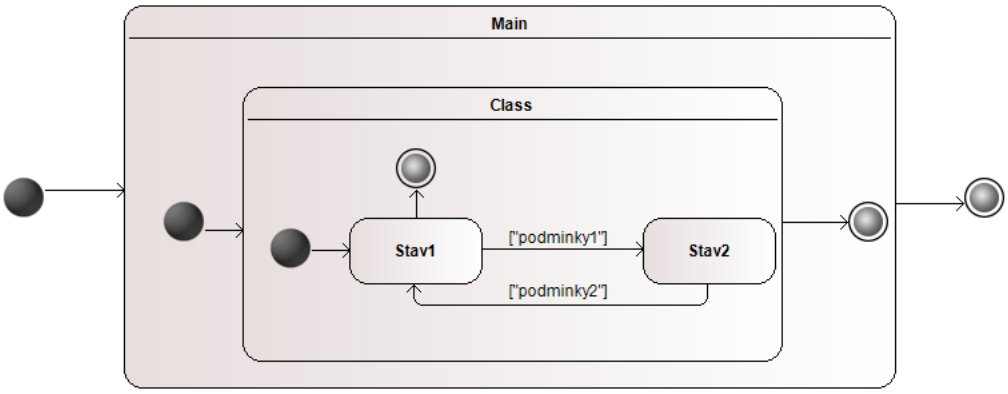
```
1  FUNCTION_BLOCK Class
2  VAR_INPUT
3    //vstupni hodnoty do FB
4  END_VAR
5  VAR_OUTPUT
6    //vystupni hodnoty z FB
7  END_VAR
8  VAR
9    //promenne potrebovane pouze pro funkcni blok
10 END_VAR
```

Možný workaround by bylo napsání skriptu, který by hledal klíčové slovo in / out v metodě třídy a následnou shodu v názvu by zaznamenal do šablony jako vstup nebo výstup. Vylučovací metodou by vše ostatní bylo jako interní proměnná funkčního bloku.

Pro ukázku návrhu budu uvažovat dvoustavový model na obrázku 6.2 se stavy "Stav 1"a "Stav 2", který bude mít souhrnné přechodové podmínky pod jedním slovem "pod-

minky1"a "podminky2". Jelikož stavový diagram má zobrazovat možné stavy jedné třídy, pak lze jednoduše říci, že každá třída bude mít svůj stavový diagram.

Z toho tedy vyplývá, že Main bude mít svůj stavový diagram a druhý stavový diagram je vnořený do stavového diagramu Main. Main již nemá další stavy, tak skáče hned do stavového diagramu třídy class. V SM class jde do stav1 a čeká na trigger souhrnných přechodových podmínek "podminky1". Po splnění přechodových podmínek se dostane do stav2, kde opět čeká na "podminky2".



Obrázek 6.2: Názorný stavový diagram pro návrh řešení

Potencionální generování je u stavového diagramu naopak od diagramu tříd jednoduché a znám dva možné způsoby zapsání kódu stavového diagramu.

6.2.1 První způsob generování ze stavového diagramu

Prvním způsobem je CASE statement, tedy přepínač mezi jednotlivými stavy po splnění přechodových podmínek. Do deklarační části se vypíše state : INT, což je proměnná pro přepínač. Dále se ihned musí inicializovat, aby mohl hned skočit do stavu1, tedy state : INT := 10 - je způsob inicializace a musí se vždy udělat v deklarační části, jinak přes to program bude pořád přejíždět a vždy se bude nastavovat na hodnotu, v tomto případě, 10. Inicializace na hodnotu 10 je pouze podle konvence, kde v případě potřeby se můžou ručně dopsat další mezistavy. Kód níže je kód pro stavový diagram třídy Class.

```
1 VAR
2   state : INT := 10;
3 END_VAR
```

```
4
5  CASE state OF
6    //Stav1
7    10: // vykonani FB/metody
8        IF ("podminky1") THEN // prechodove podminky do dalsiho stavu
9              state := 20;  // prepnuti do nasledujiciho stavu
10         END_IF
11     //Stav2
12     20: // vykonani FB/metody
13       IF ("podminky2") THEN
14             state := 10;
15         END_IF
16 END_CASE
```

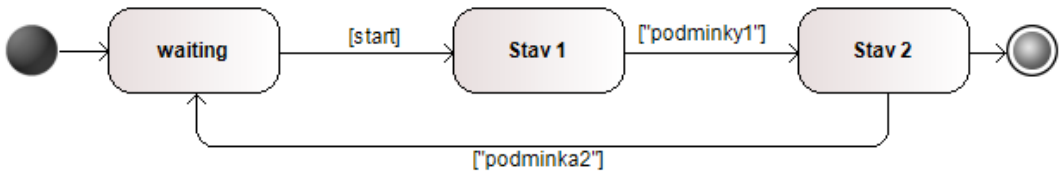
6.2.2 Druhý způsob generování ze stavového diagramu

Jako druhý způsob je kroková posloupnost za pomocí pole, nebo krokování podle indexů. Pro odlišnost jsem vytvořil nový diagram pouze třídy class, který je na obrázku 6.2. Je přidán pouze stav čekání na akci, tedy kdy se nic neděje.

```
1  VAR
2  krok  := 0; // inicializace prvniho kroku
3  krok : INT := 0;
4  aktKrok : ARRAY [0 .. 2] OF BOOL; //pole pro maximalni pocet kroku
5  krokHotov : ARRAY [0 .. 2] OF BOOL; //pole hotovych kroku
6
7  start : BOOL;
8  trigger : R_TRIG;
9  END_VAR
10 IF start AND krok = 0 THEN
11   krok := 1;
12 END_IF
13
14 IF krok > 0 THEN // pricitani kroku
15   IF krokHotov[krok] AND trigger.Q THEN
16       krok := krok + 1;
17     END_IF
18 END_IF
19
20 IF aktKrok[1] THEN
21 // vykonani FB/metod v kroku 1
22     IF "podminky1" THEN
23       krokHotov[krok] := TRUE;
24     END_IF
25 END_IF
```

```
26
27 IF aktKrok[2] THEN
28 // vykonani FB/metod v kroku 2
29     IF "podminky2" THEN
30       krokHotov[krok] := TRUE;
31     END_IF
32 END_IF
33
34 IF krokHotov[2] THEN
35   krok := 0;
36 END_IF
37
38 trigger(CLK:=krokHotov[krok]);
```

Program funguje tak, že se inicializuje krok a pole kroků datového typu BOOL. Po sepnutí tlačítka start a podmínky, že jsem v nulovém kroku, se změní krok na krok = 1 a přejde do stavu 1, kde se vykoná funkce/funkční blok a po splněných podmínkách se nastaví tak, že krok je dokončen. Pak se přičte 1 ke kroku a vykoná se to, co je v kroku 2. Aktivní krok je označen aktKrok[krok], kde krok je číslo kroku. Poté, co přišel do posledního kroku se nastaví krok na 0 a čeká. Jednoduše řečeno, je to část programu, která krokuje a po splnění podmínek přejde do kroku následovného. Další část programu je to, co se má v tom kroku udělat. Na příklad nastavení výstupů z PLC, nebo vykonání FB/metody aj.



Obrázek 6.3: Názorný stavový diagram pro druhý způsob řešení

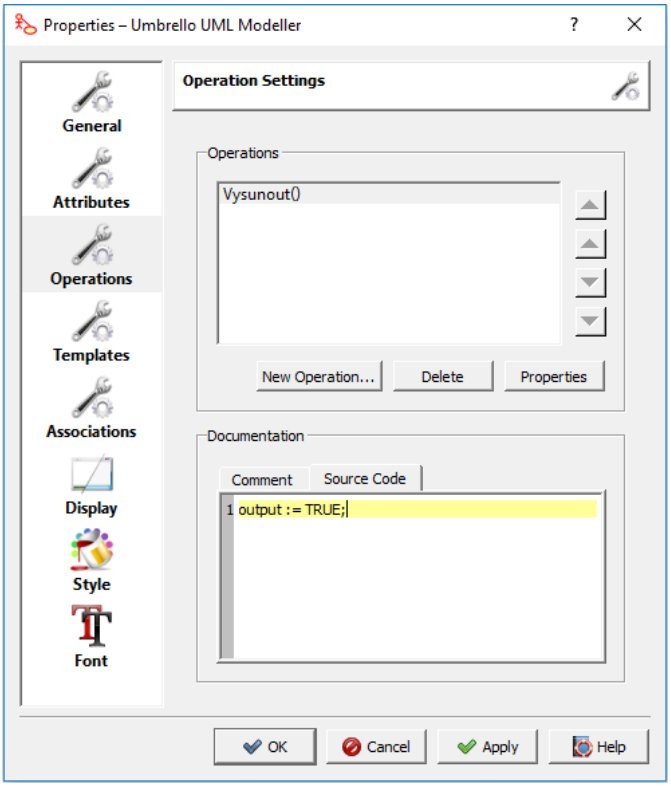
6.2.3 Porovnání

Oba způsoby jsou viabilní řešení, a oba způsoby fungují. Každému může být sympatičtější něco jiného. Práce s poli v ST je pouze základní a neobsahuje pokročilejší funkce - např. linked listy jako mají C++/C#, pak by v případě potřeby práce s nimi byla větší radost. Použití CASE statementu je jednodušší, ale zaplatí se daní trošku menší přehledností. V

porovnání s délkou kódu je druhá metoda delší. Způsob generování je otázkou, ale myslím si, že obtížnost návrhu algoritmu pro generování bude podobná.

6.2.4 Implementace funkcí

V kódu jsou funkce, které se mají vykonat a tyto funkce jsou variabilní na základě řešeného problému, tudíž je úlohou programátora napsat je ručně. Tedy neexistuje způsob generování těchto funkcí, ale můžou se specifikovat přímo v nástroji pro UML. Jedním z takových nástrojů je na příklad Umbrello, který je na obrázku 6.4. Pro jednoduchost názornosti jsem dal do zdrojového kódu v kolonce pouze nastavení výstupní hodnoty.



Obrázek 6.4: Specifikace zdrojového kódu v programu Umbrello

Další možností je nepoužívat tyto specifikátory a ručně vepsat potřebný kód.

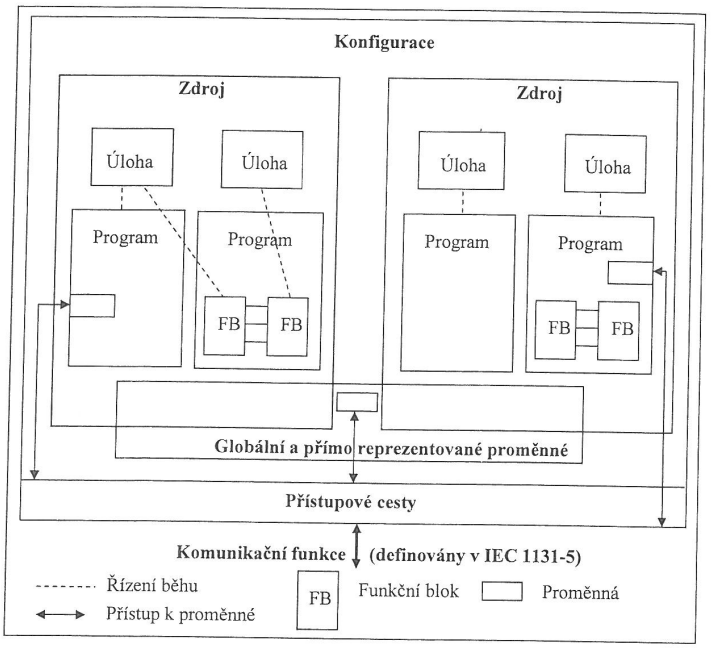
6.2.5 Globální proměnné a mapování I/O

Globální proměnné jsou proměnné, které jsou přístupné v rámci jedné konfigurace PLC. Konfigurace je prvek jazyka, který je přiřazen k celému systému PLC. Obsahem konfigu-

race jsou zdroje, které zpracovávají signály přicházející z komunikačního rozhraní (MMI, HMI). Za pomoci zdrojů se mohou namapovat vstupy a výstupy programu.

Je tedy vhodné při začátku si zmapovat vstupy ze kterých se budou přímo brát informace a výstupy, do kterých se budou posílat. Díky globálním proměnným může jakýkoliv kus kódu, kdekoliv v programu, ihned změnit výstupní hodnotu. Jak je z obrázku patrno 6.5, globální proměnné mají přímý přístup pro přístupové cesty.

Většina vyšších programovacích jazyků globální proměnné jako takové nezná (resp. neznají klíčové slovo global). Mají ale workaround. Proměnné deklarované mimo funkci mohou být přístupné v rámci souboru.[38]



Obrázek 6.5: Softwarový model
[31]

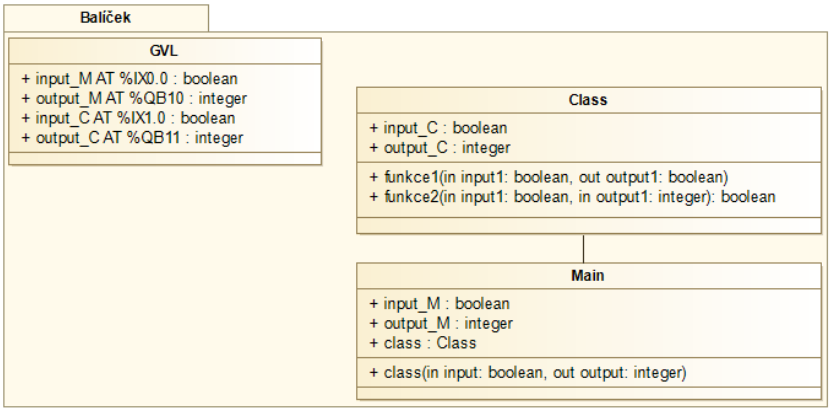
Z toho důvodu ani UML nevlastní žádný způsob pro nastavování a mapování globálních proměnných. Pro programování PLC je to ale velmi důležitá věc. Proto je vhodné si zavést konvenci jejich definování. Globální proměnné by vlastnily speciální třídu, která by speciálním znakem určila, že se jedná o Globální proměnné a při generování by sevytvořil právě soubor pro globální proměnné. Globální proměnné by byly přístupné pouze v balíčku, kde balíček je samostatná jednotka PLC, ve kterém jsou veškeré třídy pro řešení v programovém prostoru.

Další věcí je jejich použití v kódu. Při volání funkčního bloku se musí vyplnit vstupní

proměnné podle požadavků funkčního bloku. Funkční blok Class má jeden vstupní parametr, tím je input_C. Jako globální proměnnou máme též input_C a její použití je klíčové slovo .gvl.

```
1  class(input_C:=.gvl.input_C)
```

Musí být jasné co je globální proměnná, co je vstup a výstup a musí se to správně deklarovat v jednotlivých částech programu.



Obrázek 6.6: Balíček s globálními proměnnými

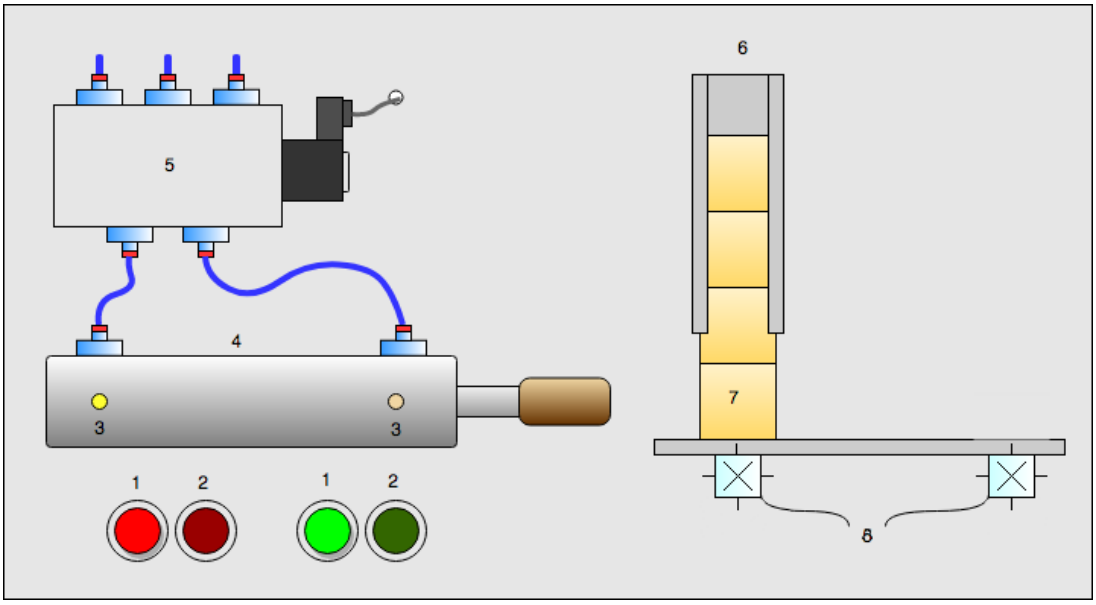
Kapitola 7

Zpracování úloh

V této kapitole jsem vybral několik zadání ze školních laboratorních úloh, ve kterých budu pomocí UML modelovat systém podle navrženého způsobu v kapitole **??**. Cílem je jednoduchost modelu a následný funkční kód podle postulátů objektově orientovaného programování. Jako vývojářské prostředí jsem vybral TWinCAT 3 od firmy Beckhoff a všechny úlohy jsou zpracované jako softwarová simulace.

TwinCAT 3

TwinCAT 3 je vývojářské prostředí pro PLC s možností simulace, které vyvinula firma Beckhoff. Prostředí je navrženo jako nadstavba Visual Studia od firmy Microsoft. Licence je zdarma volně dostupná pro vzdělávací a výcvikové účely. Obsahuje všechny jazyky normy IEC 1131-3, včetně nejnovějších norem ohledně OOP, a lze je mezi sebou různě kombinovat prostřednictvím POU, které mezi sebou pomocí deklarovaných proměnných navzájem komunikují. Funkci jednotlivých prvků, jakými jsou např. funkční bloky, lze též celou ručně naprogramovat. Nedílnou součástí je nástroj pro vizualizaci, který se dá sestavit z předdefinovaných objektů a přes dialogová okna definovat vlastnosti a funkce. Pro spuštění simulovaného PLC je nutné, aby zařízení na kterém TwinCAT běží, mělo procesor podporující x86 virtualizaci a musí být povolena v BIOSu. Tyto procesory vyrábí firma Intel s názvem virtualizace VT-x i firma AMD s názvem virtualizace AMD-V a dnes je to zcela běžná věc. [39]



Obrázek 7.1: Popis pracoviště
(1) - Kontrolní světla, (2) - Tlačítka, (3) - Koncové senzory polohy, (4) - Bistabilní pneumatický motor, (5) - Monostabilní elektricky ovládaný rozvaděč 5 na 2, (6) - Zásobník na krychle, (7) - Krychle, (8) - Hallovy sondy

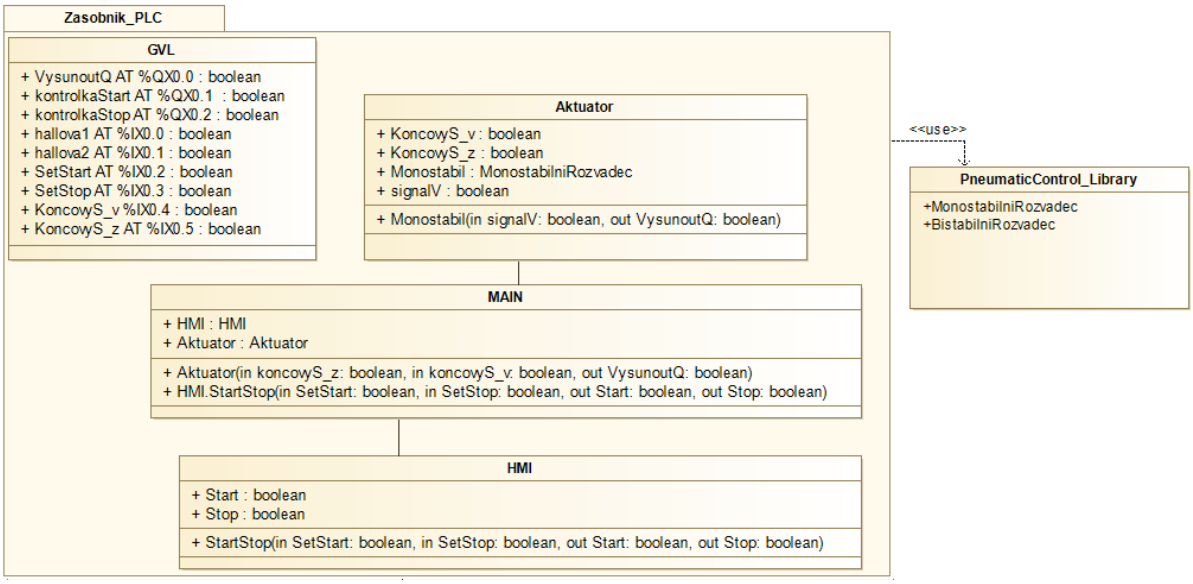
7.1 Simulace skladového zásobníku

7.1.1 Zadání úlohy

Jak vypadá pracoviště je na obrázku 7.1. Pracoviště má posouvat objekty (7), když je alespoň jedna přítomna. Detekce přítomnosti krychle je Hallovou sondou (8) v místě zásobníku a druhou Hallovou sondou (8) po odstrčení kostky. Dále jsou zde dva koncové senzory (3) bistabilního pneumatického motoru (4), který je ovládán monostabilním rozvaděčem 5 na 2 (5). Tlačítky (2) se spouští/zastavuje proces posouvání krychlí a jestli je proces spuštěn nebo zastaven je indikováno světly (2). Aktuátor nevyjede dříve, než bude v zajeté poloze, bude zmáčknuto tlačítko START a bude přítomná alespoň jedna krychle (7) v zásobníku (6). Proces bude spuštěn do doby, než bude zmáčknuté tlačítko STOP, nebo nedojdou krychle v zásobníku, tedy hallova sonda vlevo nedostane signál.

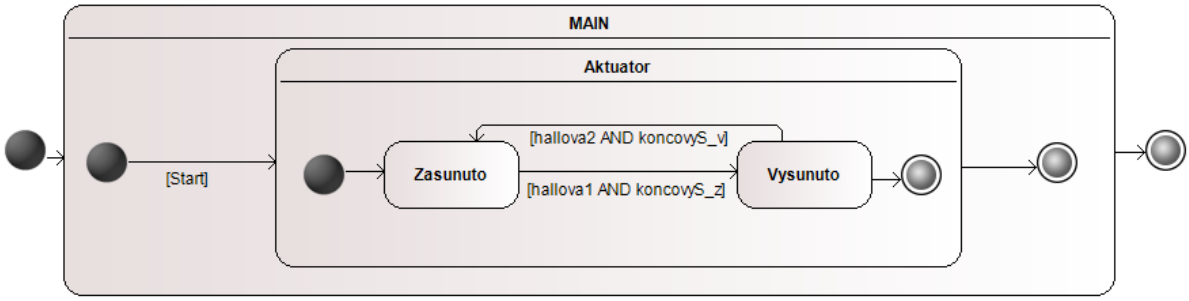
7.1.2 Řešení v UML

Při řešení této úlohy v UML jsem si prvně nakreslil statickou strukturu, jak bude systém vypadat, použitý je diagram tříd na obrázku 7.2 Každá třída reprezentuje část programu,



Obrázek 7.2: Diagram tříd pro simulaci skladového zásobníku

která má v reálném světě nějakou funkčnost. Například HMI je panel, který má tlačítko pro start a tlačítko pro stop a dvě kontrolní světla. Aktuátor je zase bistabilní pneumatický motor, který je ovládán monostabilním rozvaděčem. Ovládání monostabilního rozvaděče je posílání obyčejného signálu a je to neměnná logika, závislá pouze na vstupních proměnných. Všechny tyto neměnné ovládací logiky mohou být v knihovně připravené pro použití. Třída MAIN je povinná třída, bez toho program nefunguje.



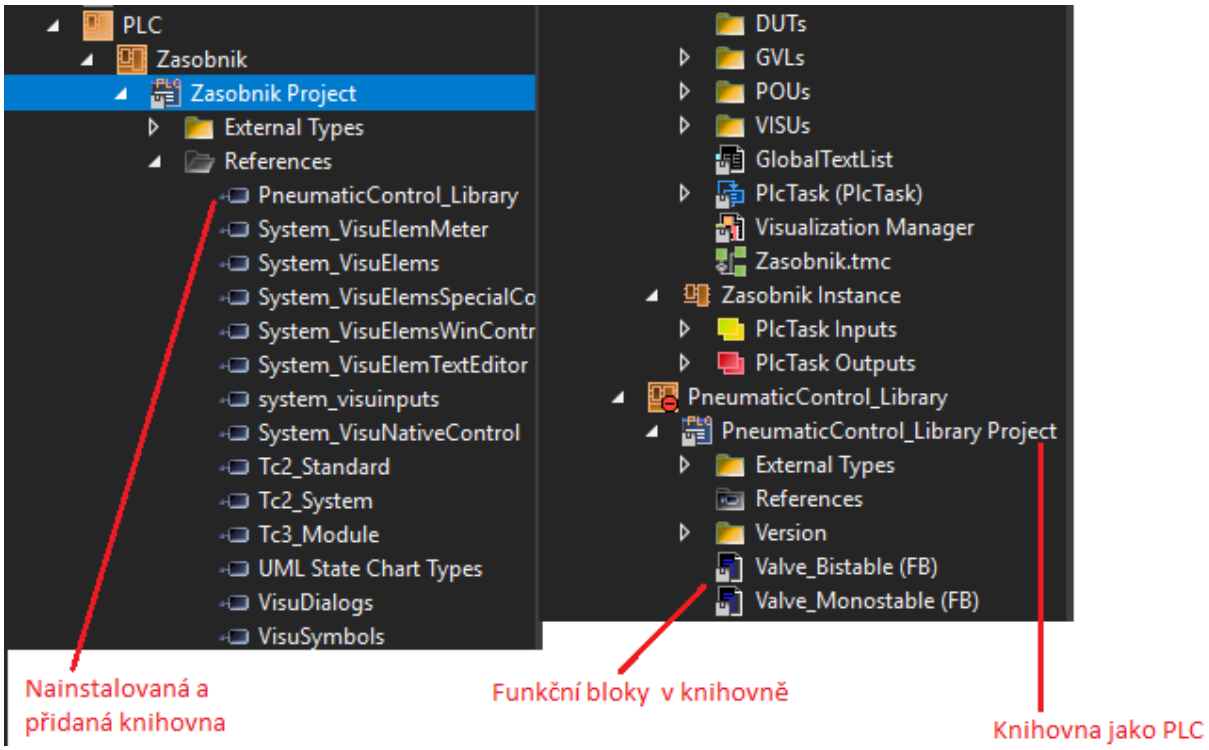
Obrázek 7.3: Stavový diagram pro simulaci skladového zásobníku

Po statické struktuře musí být popsána dynamika systému, pro kterou je stavový diagram, který je možný vidět na obrázku 7.3. Aktuátor může mít jen dva stavy, ve kterých se může nacházet. Těmito stavy je pouze zasunutý nebo vysunutý. Přechodové hrany reprezentují podmínky uvedené výše v sekci Zadání úlohy 7.1.1. Po té co se spustí program a je v MAINU, tak čeká na zmáčknutí tlačítka Start. Jakmile je tlačítko zmáčknuté, dostane

se do stavu Aktuátor: Zasunuto, tam vyčkává na splnění podmínek hallova1 a koncovyS_z, což jsou názvy proměnných v programu podle diagramu tříd na obrázku 7.2. jsou opět globální proměnné a pro generování bude opět hledat shodu jmen a před jejich názvy hodí .gvl. Pro zasunutí je to stejné ale s hallova2 a koncovyS_v. Hallova1 a Hallova2 jsou hallovy sondy a koncovyS_z a koncovyS_v jsou koncové senzory pneumatického motoru.

7.1.3 Zavedení knihovny

Jako zavedení knihovny v TwinCATu je zavedení nového PLC, které hned zakáže a vyplní se funkčními bloky, které chceme aby byly v knihovně použitelné. Pak pravým tlačítkem na projekt zvolíme "Save as library and install..."uložíme jako knihovnu a nainstalujeme do libovolné složky. Jelikož je hned vzápětí knihovna nainstalovaná, můžeme jí použít. To se udělá tak, že do PLC projektu, ve kterém je složka References. Na ní stačí kliknout pravým tlačítkem a zobrazí se možnost "Add library...". Nainstalovaná knihovna je ve stromovém seznamu pod kořenem (Miscellaneous). [40]



Obrázek 7.4: Vytvoření knihovny

7.1.4 Implementace pro skladový zásobník dle návrhu v kapitole 6

V této kapitole je ukázka kódu podle návrhu v kapitole 6. Sestává se z hlavní části programu MAIN, ze třídy Actuator, který vlastní stavový diagram a ze třídy HMI, ve které je použita metoda StartStop, které uvede tlačítko po zmáčknutí v programové části do samodržného stavu. Všechen kód je doplněn o chybějící části pro plnou funkčnost a je popsáno co je vygenerované.

```
1  PROGRAM MAIN
2  VAR
3    HMI : HMI;
4    Actuator : Actuator;
5  END_VAR
6
7  HMI.Start_Stop(Start, Stop);
8
9  HMI.StartStop(.GVL.SetStart, .GVL.SetStop);
10
11 IF HMI.Start THEN
12   Actuator(koncovyS_z := .gvl.koncovyS_z, koncovyS_v := .gvl.koncovyS_v);
13 END_IF
```

Funkční blok HMI s metodou StartStop. Deklarace proměnných je dle stavového diagramu tříd na obrázku 7.2. VAR_INPUT a VAR_OUTPUT vzaté z funkčního bloku deklarovaným v metodě MAIN podle klíčových slov IN a OUT. Start a Stop ve třídě HMI je pro ovládání kontrolních světel. Logika pro jejich ovládání je na řádcích 15 - 23 v kódu níže.

```
1  FUNCTION_BLOCK HMI
2  VAR_INPUT
3    SetStart: BOOL;
4    SetStop: BOOL;
5  END_VAR
6  VAR_OUTPUT
7    Start: BOOL;
8    Stop: BOOL;
9  END_VAR
10 VAR
11 END_VAR
12
13 StartStop(SetStart := .gvl.SetStart, SetStop := .gvl.SetStop);
14
15 IF Start THEN
16   .GVL.kontrolkaStart := TRUE;
```

```
17   .GVL.kontrolkaStop := FALSE;
18 END_IF
19
20 IF Stop THEN
21   .GVL.kontrolkaStart := FALSE;
22   .GVL.kontrolkaStop := TRUE;
23 END_IF
```

Funkční blok Aktuátoru. Deklarace funkčního bloku je z diagramu tříd. Na řádcích 15-26 v kódu níže je přepínač mezi jednotlivými stavy. První stav je dán jako 10, komentář před ním aby programátor věděl, o jaký stav se jedná. podmínky přechodu s globálními proměnnými jsou vygenerované ze shody názvů ze speciálního GVL souboru v diagramu tříd.

```
1  FUNCTION_BLOCK Actuator
2  VAR_INPUT
3    koncovyS_v : BOOL;
4    koncovyS_z : BOOL;
5  END_VAR
6  VAR_OUTPUT
7    VysunoutQ : BOOL;
8  END_VAR
9  VAR
10   state: INT := 10;
11   signalV : BOOL; // signal pro ovladani monostabilniho rozvadece
12   Monostabil : Valve_Monostable; //deklarace funkcniho bloku pro ovladani monostabilniho
       rozvadece
13 END_VAR
14
15 CASE state OF
16   //Zasunuto
17   10: signalV:=FALSE; // rucne doplnena logika pro ovladani monostabilniho rozvadece.
       FALSE = zasunuto
18     IF .GVL.hallova1 AND .GVL.koncovyS_z THEN
19       state := 20;
20     END_IF
21   //Vysunuto
22   20: signalV:=TRUE; // rucne doplnena logika. TRUE = Vysunuto
23     IF .GVL.hallova2 AND .GVL.koncovyS_v THEN
24       state := 10;
25     END_IF
26 END_CASE
27 monostabil(signal:=signalV); // pouziti funkcniho bloku, generovane z diagramu trid
28 .gvl.VysunoutQ := Monostabil.Q; // rucne doplnena vystupni hodnota do globalni promenne
```

Funkční blok Monostabilního rozvaděče. Funguje jenom pro přeposílání na výstup. Sta-

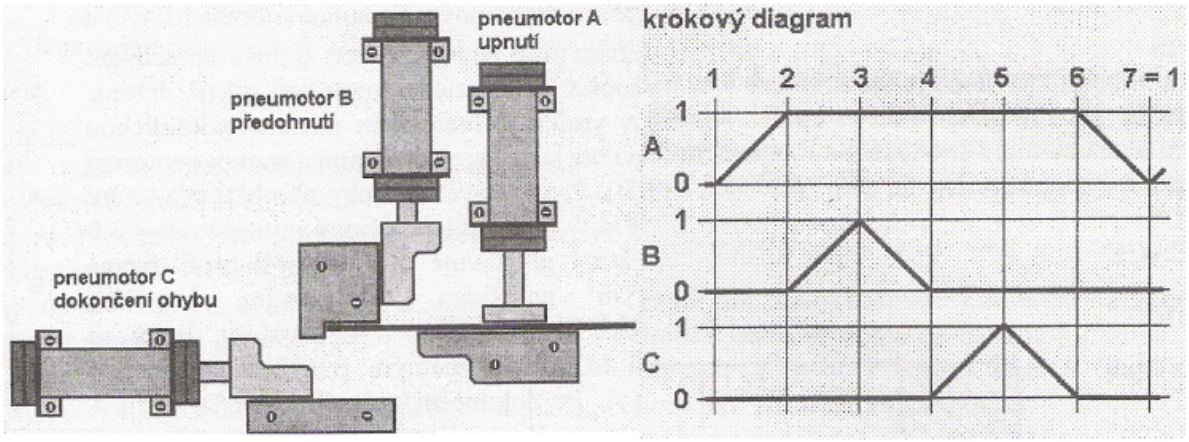
čilo by na řádek 17 v kódu výše místo ”10: signalV := FALSE”dát ”.gvl.VysunoutQ :=FALSE”a na řádek 22 ”.gvl.VysunoutQ:=TRUE”. Ale z hlediska OOP a teoretického použití knihoven pro ovládací prvky jsem tento funkční blok napsal a využil.

```
1  FUNCTION_BLOCK Valve_Monostable
2  VAR_INPUT
3    signal : BOOL;
4  END_VAR
5  VAR_OUTPUT
6    VysunoutQ : BOOL;
7  END_VAR
8  VAR
9  END_VAR
10
11 IF signal THEN
12   VysunoutQ := TRUE;
13 END_IF
14
15 IF NOT signal THEN
16   VysunoutQ := FALSE;
17 END_IF
```

7.2 Přípravek na ohýbání

7.2.1 Zadání úlohy

Na pneumaticky ovládané ohýbačce mají být lemovány plechy. Po stisknutí startovacího tlačítka má být ohnuto Nc plechů v jedné dávce. Počet kusů Nc v dávce může být změněn kdykoliv z operátorského pracoviště. Jeden pracovní cyklus probíhá následovně: Plech je podán ze zásobníku, jeho přítomnost v pracovní poloze je indikována čidlem. Po zjištění přítomnosti plechu je tento upnut jednočinným upínacím pneumotorem A a po upnutí je dvojčinným pneumotorem B předohnut. Poté je dalším pneumotorem C do ohnut do žádaného tvaru. V této poloze je třeba pneumotor po jistou technologickou dobu Tc ponechat. Tuto dobu je možno zadávat z pracoviště operátora a po změně bude platná již pro následující součást. Poté je plech uvolněn pneumotorem A a mechanickým vyhazovačem odstraněn. Situace je znázorněna s krokovým diagramem na obrázku 7.5 [41]



Obrázek 7.5: Náčrt situace s krokový diagram
[41]

7.2.2 Řešení v UML

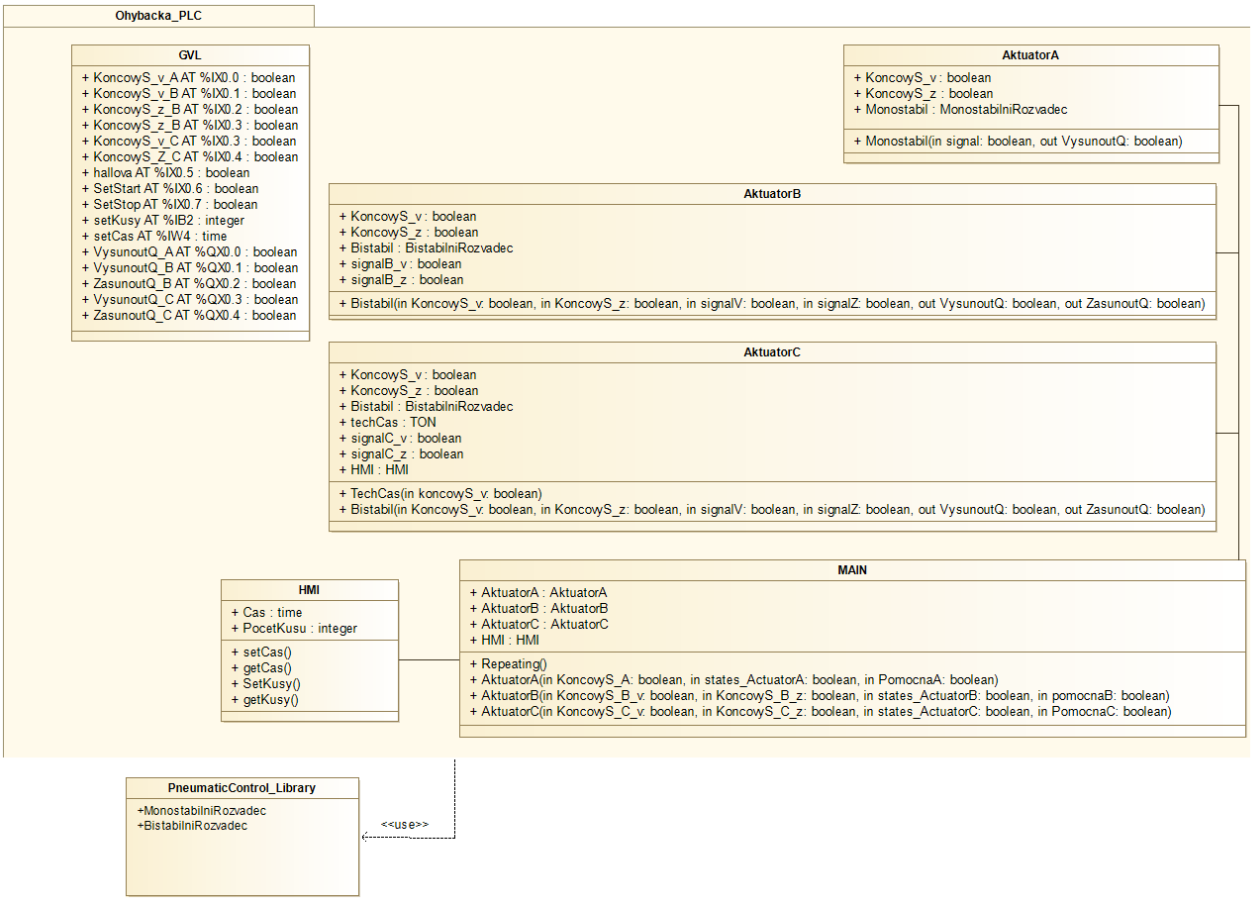
Řešení této úlohy se tolik neliší od řešení předchozí úlohy. Vytvoření diagramu tříd je na obrázku 7.6 na stránce 38 a stavových diagramů pro aktuatory 7.8, 7.9 a 7.10 na stránce 38. Diagramy jsou rozděleny do čtyř částí. První diagram na obrázku 7.7, je pro program main, která vlastní tři stavy, pro každý aktuátor zvlášť. A: A je přiřazen aktuátoru A na obrázku 7.8, B: B aktuátoru B na obrázku 7.9 a C: C aktuátoru C na obrázku 7.10.

Na stavovém diagramu pro metodu main na obrázku 7.7, je nejprve inicializace do počátečního stavu stroje a následná posloupnost. Přechodové podmínky jsou zároveň přechodovými podmínkami z tohoto konkrétního stavu. Na příklad ze stavu A:A, do stavu B:B se dostanu vystoupením ze stavu A, kde je přechodová podmínka PomocnaA. Tato pomocná značí, že v jednom cyklu aktuátor A již byl vysunut.

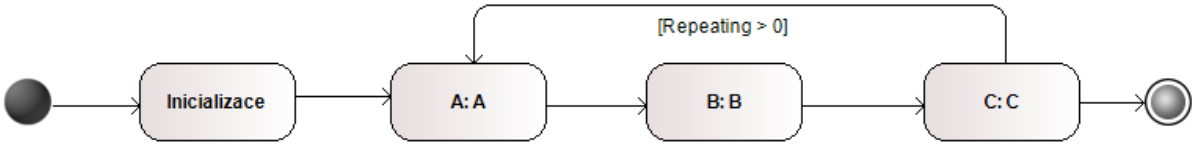
Na stavovém diagramu na obrázku 7.8 je po zmáčknutí tlačítka start a po detekčním členu (čidlu) vysunutí a následné uložení do pomocné proměnné, která vyjde ze stavového diagramu.

Diagram se poté vrátí do stavového diagramu samotného PLC a jde na další, který je na 7.9. Tento pohon se má pouze vysunout a zase zasunout. Opět se uloží do pomocné proměnné, která je i podmínkou pro zasunutí a výstupu ze stavového diagramu pro pohon B do stavového diagramu pro PLC.

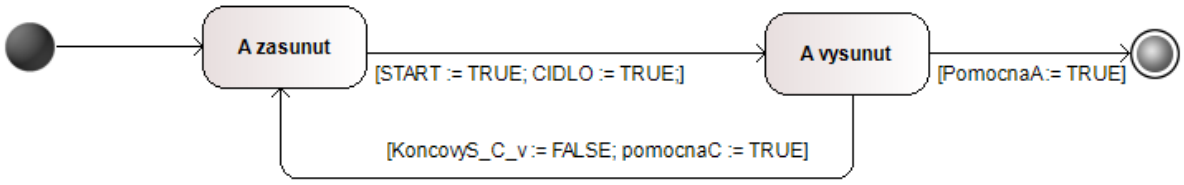
Pro pohon C, který se vysune a zůstane ve stavu vysunutí požadovanou technologickou



Obrázek 7.6: Diagram tříd pro řešení úlohy

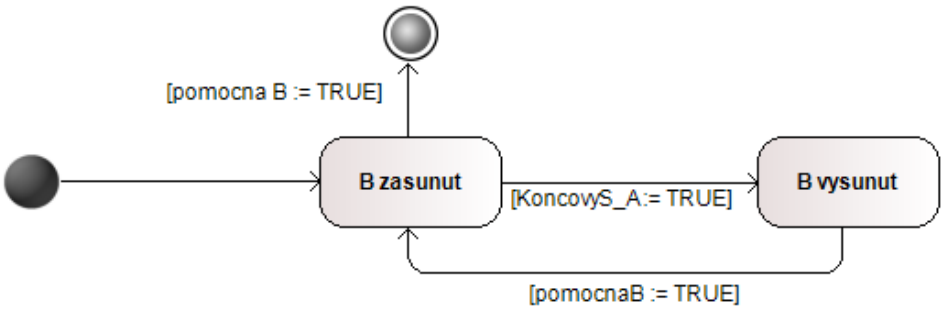


Obrázek 7.7: Stavový diagram pro MAIN



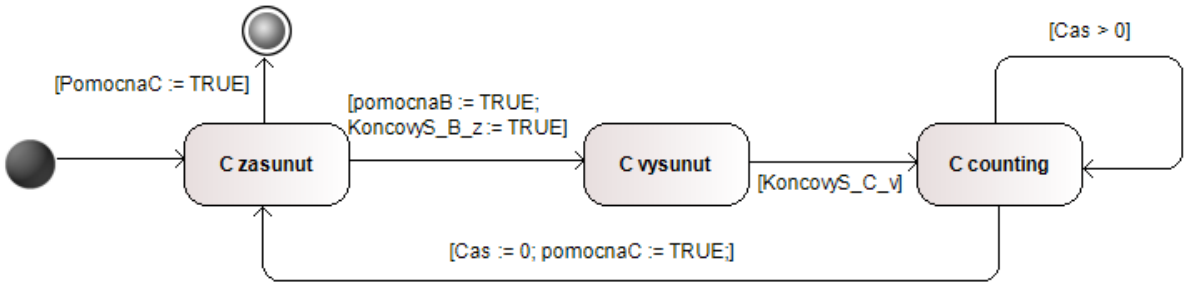
Obrázek 7.8: Stavový diagram pro aktuátor A

dobu, je znázorněn stavový diagram na obrázku 7.10. Pro zachování posloupnosti kroků nemůže tento motor být vysunut dříve než motor B, což je také podmínkou společně s tím, že motor B je již zasunutý. Po vysunutí motoru C, který má podmínku vysunutého



Obrázek 7.9: Stavový diagram pro aktuátor B

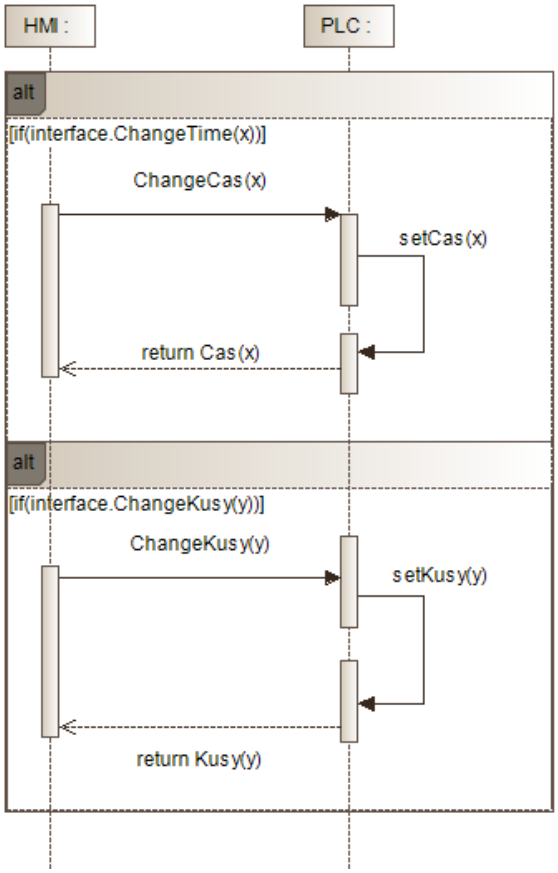
koncového senzoru C, se dostane do stavu počítání času "C counting". Ve stavu počítání zůstane patřičnou dobu, pro splnění podmínky technologického času. Pro stav počítání je potřeba v kódu použít "on delay timer - TON", časovač, který se spustí s náběžnou hranou. Časovač je již vymyšlen jako funkční blok a je v normě IEC 61131-3 tudíž i definován. Jeho použití při následném generování kódu by mohlo být vyznačeno speciálním symbolem v diagramu, speciálním atributem, nebo označením převzaté z diagramu tříd «TON». Nebo se použije odkaz na použitou knihovnu jak jsem již zmínil v řešení skladového zásobníku v kapitole 7.1.



Obrázek 7.10: Stavový diagram pro aktuátor C

Na obrázku 7.11 je sekvenční diagram pro nastavování žádaných parametrů. Tyto parametry jsou nastavení technologického času pro ohnutí a nastavení počtu kusů pro výrobu. Ty jsou v sekvenčním diagramu označeny jako ChangeCas(x) pro změnu času a ChangeKusy(y) pro změnu počtu vyrobených kusů v jedné dávce. Mezi změnou těchto parametrů si vybírám, podle obdélníku s označením alt, ve kterém se sekvence nachází. Výběr sekvence je kontrolován hlídačem, tak zvaným guardem. Guard má za úkol hlídat, jestli je podmínka splněna pro odstartování sekvence, kterou lze vidět v levém horním

rohu hned pod "altem". Podmínka je smyšlená z hlediska HMI - když je vybraná třída interface s metodou ChangeCas(x) s proměnným parametrem x, je tato informace poslána do PLC, který nastaví čas na x a po nastavení vrátí jeho hodnotu zpátky do HMI. Tím se v HMI opět zobrazí hodnota po nastavení. To samé se děje s druhou alternativou, kde se nastavuje počet kusů.

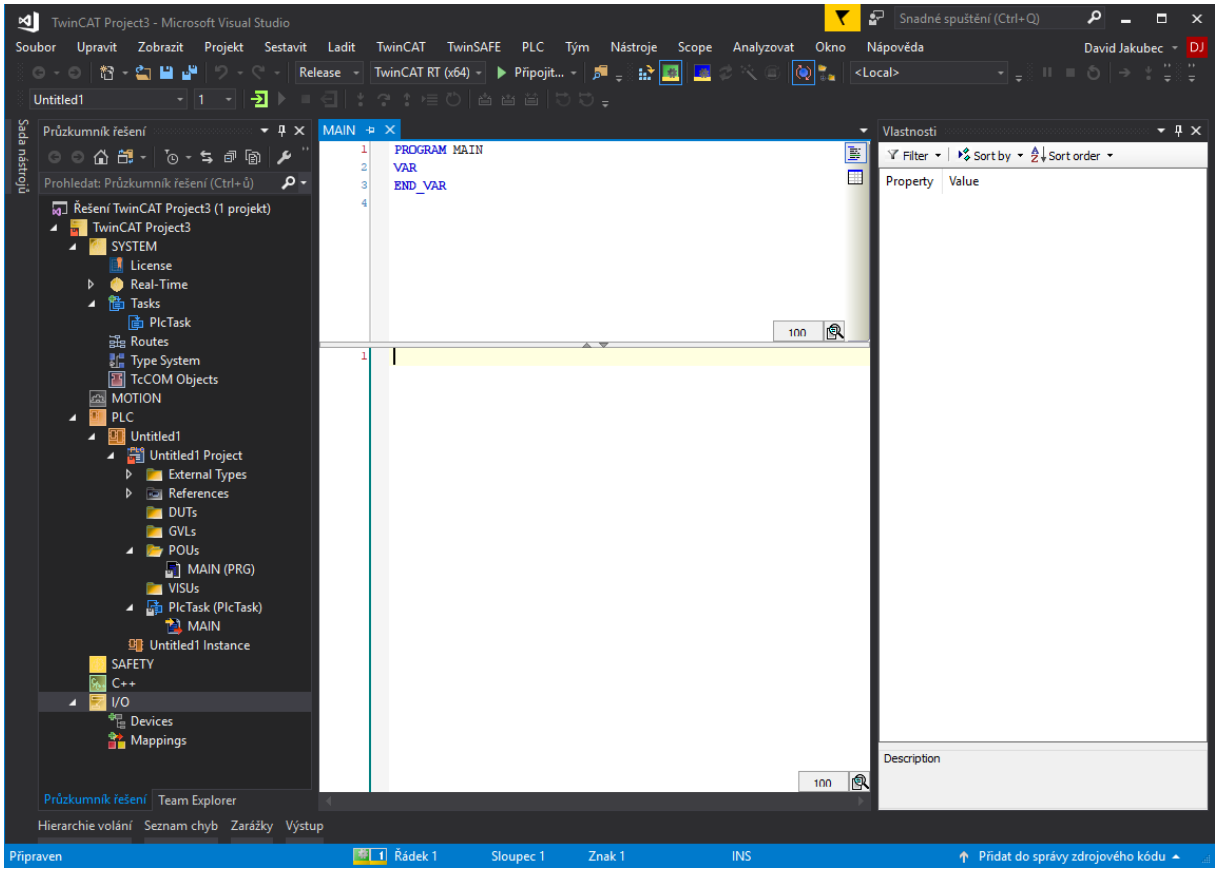


Obrázek 7.11: Sekvenční diagram pro komunikaci mezi HMI a PLC

7.2.3 Řešení v TwinCAT 3

Následující kódy jsou rozdělené podle obrázku 7.7, na kterém je stavový diagram stavových diagramů a vždy je uvedené podle kterého diagramu je kód udělán. V této sekci již nebudu tolik rozepisovat jak je co jednotlivě generované, nebo co k čemu slouží.

Program MAIN, je možná na první pohled vidět jedna monstrozita s názvem VectorStates, ke které je něco napsané níže. Jinak je vše podle již zmíněného diagramu tříd.



Obrázek 7.12: Rozhraní TwinCAT

```
PROGRAM MAIN
VAR
  HMI : HMI;
  ActuatorA : ActuatorA;
  ActuatorB : ActuatorB;
  ActuatorC : ActuatorC;
END_VAR
VAR_INPUT
  VectorStates : VectorStates;
END_VAR
//vektor stavu, zapisuji se do nej stavy a pomocne promenne, posila je pote na vystup
VectorStates(
inStates_ActuatorA := ActuatorA.VectorStates.inStates_ActuatorA,
inStates_ActuatorB := ActuatorB.VectorStates.inStates_ActuatorB,
inStates_ActuatorC := ActuatorC.VectorStates.inStates_ActuatorC,
inPomocnaA := ActuatorA.VectorStates.inPomocnaA,
inPomocnaB := ActuatorB.VectorStates.inPomocnaB,
inPomocnaC := ActuatorC.VectorStates.inPomocnaC,
inEndOfcycle := VectorStates.inEndOfCycle,
inStates_main := VectorStates.inStates_main);

```

```
22 HMI(SetStart := .GVL.SetStart, SetStop := .GVL.SetStop);
23
24 IF VectorStates.outEndOfCycle OR HMI.reset THEN
25   endOfCycle();
26 END_IF
27
28 IF HMI.Start THEN
29   CASE VectorStates.inStates_main OF
30     100:  ActuatorA(koncovyS_v := .GVL.KoncovyS_v_A);
31         VectorStates.inStates_main := ActuatorA.VectorStates.inStates_main;
32
33     200:  ActuatorB(KoncovyS_V := .GVL.KoncovyS_v_B, KoncovyS_Z := .GVL.KoncovyS_z_B);
34         VectorStates.inStates_main := ActuatorB.VectorStates.inStates_main;
35         pomocnaB_main := ActuatorB.VectorStates.inPomocnaB;
36
37     300:  ActuatorC(KoncovyS_V := .GVL.KoncovyS_v_C, KoncovyS_Z := .GVL.KoncovyS_z_C);
38         VectorStates.inStates_main := ActuatorC.VectorStates.inStates_main;
39   END_CASE
40 END_IF
```

Vektor stavů, není uvedený v diagramu tříd, z důvodu popsané v 7.2.3 na stránce 47.

```
1  FUNCTION_BLOCK VectorStates
2  VAR_INPUT
3    inStates_ActuatorA : INT := 10;
4    inStates_ActuatorB : INT := 10;
5    inStates_ActuatorC : INT := 10;
6    inStates_main : INT := 100;
7    inPomocnaA : BOOL := FALSE;
8    inPomocnaB : BOOL := FALSE;
9    inPomocnaC : BOOL := FALSE;
10   inEndOfCycle : BOOL;
11 END_VAR
12 VAR_OUTPUT
13   outStates_ActuatorA : INT := 10;
14   outStates_ActuatorB : INT;
15   outStates_ActuatorC : INT;
16   outStates_main : INT;
17   outPomocnaA : BOOL;
18   outPomocnaB : BOOL;
19   outPomocnaC : BOOL;
20   outEndOfCycle : BOOL;
21 END_VAR
22 outStates_ActuatorA := inStates_ActuatorA;
23 outStates_ActuatorB := inStates_ActuatorB;
24 outStates_ActuatorC := inStates_ActuatorC;
25 outStates_main := inStates_main;
26 outPomocnaA := inPomocnaA;
```

```
outPomocnaB := inPomocnaB;
outPomocnaC := inPomocnaC;
```

Kód pro aktuátor A.

```
FUNCTION_BLOCK ActuatorA
VAR_INPUT
  koncovyS_v : BOOL;
END_VAR
VAR_OUTPUT
  VysunoutQ : BOOL;
END_VAR
VAR
  VectorStates : VectorStates;
  monostabilA : Valve_Monostable;
  signalA_V : BOOL;
END_VAR
CASE VectorStates.inStates_ActuatorA OF
          10: signalA_V := FALSE; //stav vysunuto - bude drzet nez s

            IF .gvl.hallova AND NOT VectorStates.inPomocnaC AND .gvl.KoncovyS_z_C THEN
              VectorStates.inStates_ActuatorA := 20;
            END_IF

            20: signalA_V := TRUE;
              VectorStates.inPomocnaA := TRUE;
            IF VectorStates.outPomocnaC AND .GVL.KoncovyS_z_C THEN
              VectorStates.inStates_ActuatorA := 10;
            END_IF
            VectorStates.inStates_main := 200;
END_CASE
monostabilA(signal:=signalA_V);
.GVL.VysunoutQ_A := monostabilA.Q;
```

Kód pro Aktuátor B.

```
FUNCTION_BLOCK ActuatorB
VAR_INPUT
  KoncovyS_v : BOOL;
  KoncovyS_z : BOOL;
END_VAR
VAR_OUTPUT
END_VAR
VAR
  VectorStates : VectorStates;
  signalB_V : BOOL;
  signalB_Z : BOOL;
  bistabil : Valve_Bistable ;
END_VAR
```

```
CASE vectorStates.inStates_ActuatorB OF
          10: signalB_z := TRUE;
            IF .GVL.KoncovyS_v_A AND NOT VectorStates.inPomocnaB THEN
              VectorStates.inStates_ActuatorB := 20;
            END_IF
            VectorStates.inStates_main := 200; // zabrana k prechodu do predchoziho stavu
            IF .GVL.KoncovyS_z_B AND VectorStates.inPomocnaB THEN
              VectorStates.inStates_main := 300;
            END_IF
          20: signalB_v := TRUE;
            IF .GVL.KoncovyS_z_B AND VectorStates.inPomocnaB THEN
              VectorStates.inStates_ActuatorB := 10;
            END_IF
END_CASE

bistabil(koncovySenzorZ := .GVL.KoncovyS_z_B, KoncovySenzorV := .gvl.KoncovyS_v_B,
    signalV := signalB_v, signalZ := signalB_z);
.GVL.VysunoutQ_B := bistabil.Vysunout; // nastaveni vystupu na globalni promennou
.GVL.ZasunoutQ_B := bistabil.Zasunout;

IF .gvl.KoncovyS_v_B THEN // nastaveni pomocne promenne
  VectorStates.inPomocnaB := TRUE;
END_IF
```

Kód pro Aktuátor C.

```
FUNCTION_BLOCK ActuatorC
VAR_INPUT
  KoncovyS_v : BOOL;
  KoncovyS_z : BOOL;
END_VAR
VAR_OUTPUT
END_VAR
VAR
  VectorStates : VectorStates;
  signalC_v : BOOL;
  signalC_z : BOOL;
  techCas : TON;
  bistabil : Valve_Bistable ;
  HMI : HMI;
  PomocnaC : BOOL;
END_VAR
CASE VectorStates.inStates_ActuatorC OF
          10: signalC_z := TRUE;
            IF main.pomocnaB_main AND .gvl.KoncovyS_z_B AND NOT main.VectorStates.
    outPomocnaC THEN
              VectorStates.inStates_ActuatorC := 20;
            END_IF
```

```
22              IF main.VectorStates.outPomocnaC THEN
23                VectorStates.inStates_main := 100;
24                VectorStates.inEndOfCycle := TRUE;
25              END_IF
26              VectorStates.inStates_main := 300; // zabrana k prechodu do predchoziho stavu
27            20: signalC_v := TRUE;
28              IF .GVL.KoncovyS_v_C THEN
29                VectorStates.inStates_ActuatorC := 30;
30                .gvl.PomocnaC := TRUE;
31              END_IF
32            30: TechCas(IN:=.GVL.VysunoutQ_C,PT:=HMI.getCas);
33              IF TechCas.Q THEN
34                VectorStates.inStates_ActuatorC := 10;
35                VectorStates.inStates_ActuatorA := 10;
36              END_IF
37  END_CASE
38  bistabil(koncovySenzorZ := .GVL.KoncovyS_z_B, KoncovySenzorV := .gvl.KoncovyS_v_B,
        signalV := signalC_v, signalZ := signalC_z);
39  .GVL.VysunoutQ_C := bistabil.Vysunout; // nastaveni vystupu
40  .GVL.ZasunoutQ_C := bistabil.Zasunout;
```

Funkční blok HMI, kde se berou z globálních proměnných čas a počet kusů výrobků. Do globálních proměnných se zapíšou hodnoty přes komunikační protokol z HMI.

```
1  FUNCTION_BLOCK HMI
2  VAR_INPUT
3    SetStart : BOOL;
4    SetStop : BOOL;
5  END_VAR
6  VAR_OUTPUT
7    Start : BOOL;
8    Stop : BOOL;
9    setCas : TIME;
10   setKusy : INT;
11   reset : BOOL;
12 END_VAR
13 VAR
14   endOfCycle: BOOL;
15   getKusy: INT;
16   getCas : TIME;
17 END_VAR
18 setKusy := .GVL.setKusy;
19 getKusy := setKusy;
20
21 setCas := .Gvl.setCas;
22 getCas := setCas;
23
```

```
IF .gvl.SetStop THEN
  Stop := TRUE;
  Start := FALSE;
  reset:=TRUE;
END_IF

IF .gvl.SetStart THEN
  Start := TRUE;
  Stop := FALSE;
END_IF
```

Metoda endOfCycle, kde se po skončení cyklu, nebo po zmáčknutí tlačítka stop resetují proměnné na původní iniciální stav.

```
  VectorStates.inPomocnaA := FALSE;
  VectorStates.inPomocnaB := FALSE;
  VectorStates.inPomocnaC := FALSE;
  VectorStates.instates_main := 100;
  VectorStates.instates_ActuatorA := 10;
  VectorStates.instates_ActuatorB := 10;
  VectorStates.instates_ActuatorC := 10;
  VectorStates.inendOfCycle := FALSE;
```

Metoda Repeating pro opakování počtu cyklů zadané z HMI.

```
METHOD Repeating : BOOL
VAR_INPUT
END_VAR
VAR
  VS : VectorStates;
  CTD : CTD;
  trigg : R_TRIG;
END_VAR
trigg(CLK:=HMI.Start);

CTD(CD:=VS.outEndOfCycle, LOAD := trigg.Q, PV:=HMI.getKusy);

IF CTD.Q = 0 THEN
  HMI.SetStop := TRUE;
END_IF
```

Bistabilní rozvaděč potřebuje pro funkci dva signály, kde každý ze signálu rozvaděč přestaví a stačí mu impuls. Monostabilní naopak potřebuje dodávat signál neustále k tomu, aby byl přestavený. Kód níže je kód pro ovládání bistabilního rozvaděče, který jsem použil v knihovně.

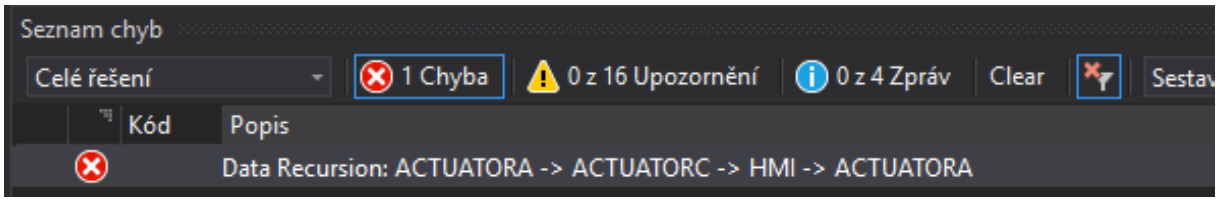
```
FUNCTION_BLOCK Valve_Bistable
```

```
2  VAR_INPUT
3    KoncovySenzorZ : BOOL; // koncove senzory
4    KoncovySenzorV : BOOL;
5      signalV: BOOL; // signaly pro vyjeti / zajeti
6      signalZ: BOOL;
7  END_VAR
8  VAR_OUTPUT
9    Vysunout : BOOL; // vystupni signaly
10   Zasunout : BOOL;
11 END_VAR
12
13 IF signalV AND NOT KoncovySenzorV THEN
14   Vysunout := TRUE;
15     ELSE Vysunout := FALSE; // resetovani signalu
16 END_IF
17
18 IF signalZ AND NOT KoncovySenzorZ THEN
19   Zasunout := TRUE;
20     ELSE Zasunout := FALSE; // resetovani signalu
21 END_IF
```

Vektor Stavů

Při pokusu o naprogramování podle uvedené metodiky se vyskytl problém. Pro uchování a čtení proměnných v jiných třídách, např. mezi jednotlivými aktuátory, se musí udělat vazba. Křížová reference, jako deklarování funkčního bloku Aktuátoru C ve třídě Aktuátoru A se udělat nemůže, jelikož to vytvoří datovou rekurzi a IDE vyhodí chybovou hlášku, kvůli které se program ani nestáhne, jak lze vidět na obrázku 7.13. Pokusil jsem se tedy zachovat co největší kompatibilitu s modelem a vytvořit funkční blok, vektor stavů, do kterého by se všechny pomocné proměnné pro zachování posloupnosti kroků ukládaly, včetně stavů stavového diagramu. Při tomto řešení jsem se hlavně potýkal s problémem, kde se do tohoto vektoru na výstup nechtěly promítat vstupní hodnoty. Kód je funkční, ale při krokování a ladění trochu nechtěl opět některé hodnoty dávat na výstup, i když byly zapsány stejným způsobem. Jednodušším řešením by bylo ukládat proměnnou z funkčního bloku do proměnný v mainu a poté ji volat ve funkčním bloku, kde je potřeba, nebo použít globální proměnné. Mojí snahou však bylo plně využít objektově orientovaného programování ST a pokusit se s tímto problémem vypořádat přes funkční blok, o kterém jsem si myslel, že by mohl být lehce použitelný.

Ale nejjednodušším řešením vůbec bylo to nedělat za pomocí OOP.



Obrázek 7.13: Datová rekurze

7.2.4 Neobjektově orientované řešení

Tento kód níže je mým prvním prototypem řešení této úlohy pro stavový diagram a určen hlavně pro vizualizaci. Nejsou zde žádné globální proměnné, ani posílání na reálný výstup z PLC.

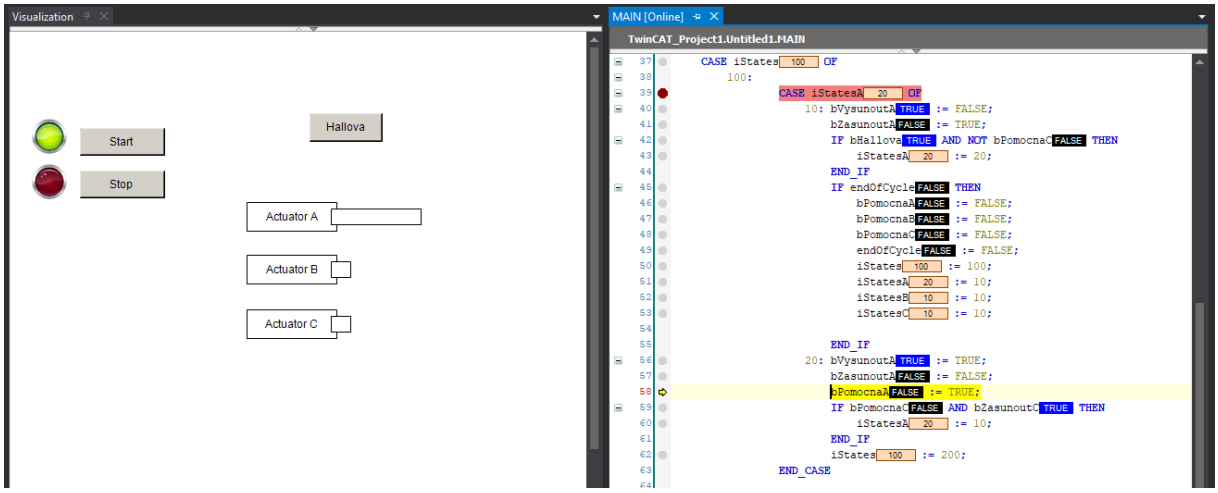
```
IF bStart THEN
  CASE iStates OF
    100:
        CASE iStatesA OF
          10:
                      bVysunoutA := FALSE;
            bZasunoutA := TRUE;
            IF bHallova AND NOT bPomocnaC THEN
              iStatesA := 20;
            END_IF
            IF endOfCycle THEN
              bPomocnaA := FALSE;
              bPomocnaB := FALSE;
              bPomocnaC := FALSE;
              endOfCycle := FALSE;
              iStates := 100;
              iStatesA := 10;
              iStatesB := 10;
              iStatesC := 10;
            END_IF
          20:
                      bVysunoutA := TRUE;
            bZasunoutA := FALSE;
            bPomocnaA := TRUE;
            IF bPomocnaC AND bZasunoutC THEN
              iStatesA := 10;
            END_IF
            iStates := 200;
        END_CASE
```

```
30
31      200:
32          CASE iStatesB OF
33            10: bZasunoutB := TRUE;
34              bVysunoutB := FALSE;
35              IF bVysunoutA AND NOT bPomocnaB THEN
36                iStatesB := 20;
37              END_IF
38              IF bZasunoutB AND bPomocnaB THEN
39                iStates := 300;
40              END_IF
41            20: bZasunoutB := FALSE;
42              bVysunoutB := TRUE;
43              bPomocnaB := TRUE;
44              iStatesB := 10;
45          END_CASE
46      300:
47          CASE iStatesC OF
48            10: bZasunoutC := TRUE;
49              bVysunoutC := FALSE;
50              IF bPomocnaB AND bZasunoutB AND NOT bPomocnaC THEN
51                iStatesC := 20;
52              END_IF
53              IF bPomocnaC THEN
54                iStates := 100;
55                endOfCycle := TRUE;
56              END_IF
57            20: bZasunoutC := FALSE;
58              bVysunoutC := TRUE;
59              bPomocnaC := TRUE;
60              IF bVysunoutC THEN
61                iStatesC := 30;
62              END_IF
63            30: tonTechCas(IN:=bVysunoutC,PT:=tTechCas);
64              IF tonTechCas.Q THEN
65                iStatesC := 10;
66                iStatesA := 10;
67              END_IF
68          END_CASE
69    END_CASE
70  END_IF
```

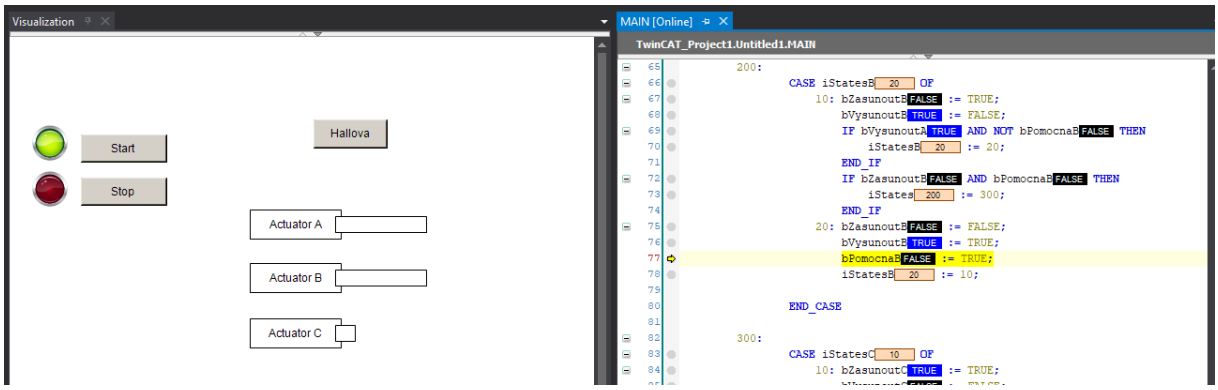
Kód reprezentuje vnořené stavové diagramy. Kód je tvořen z proměnné iStates, která má hodnoty 100, 200 a 300, tedy vyšší vrstvy a nižší vrstvy mají proměnné iStatesA, iStatesB, iStatesC. Vrstvy jsou rozdělené kvůli uchování informace o stavu, ve kterém se jednotlivý aktuátor nachází. iStatesA náleží objektu aktuátoru A, který aktivuje pneumatický motor

A. S iStatesB a iStatesC je to samé, ale pro B a C. Pozn.: Kvůli vizualizaci byl použito nastavování stavu "bVysunout"a "bZasunout"jako true nebo false. Je to z důvodu, že vizualizace používá geometrické útvary, které se zneviditelní za uvedené podmínky. To dodává pocit, že se pneumatické motory ve vizualizaci hýbou.

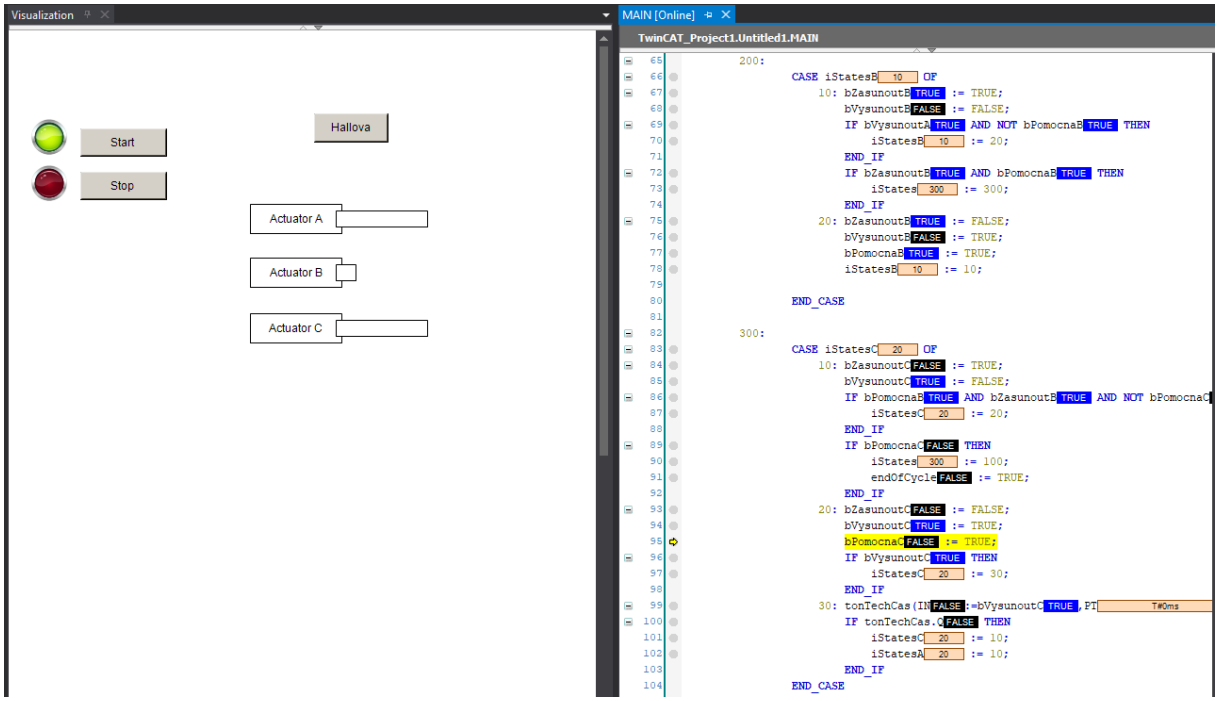
Procházení kódu s vizualizací



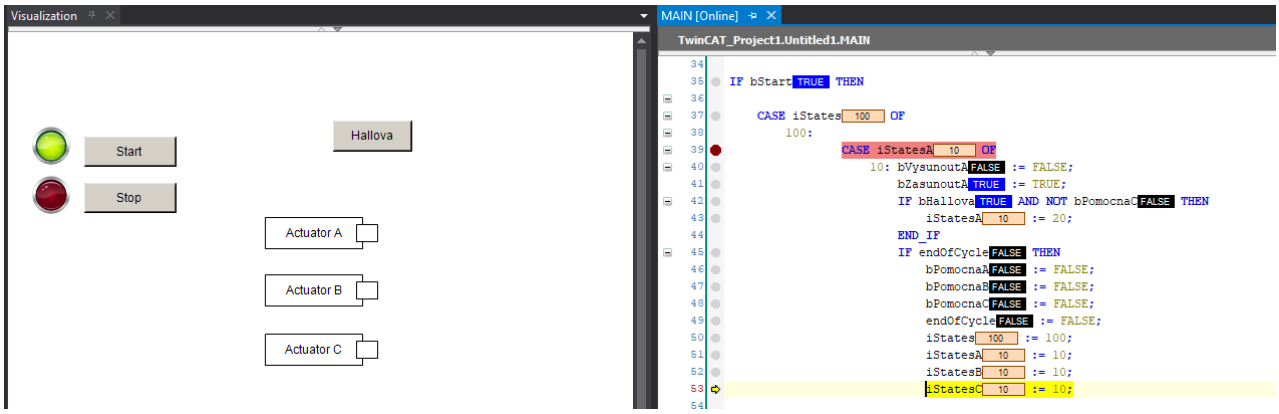
Obrázek 7.14: Krokování kódu v rozhraní TwinCAT - setování A



Obrázek 7.15: Krokování kódu v rozhraní TwinCAT - setování B



Obrázek 7.16: Krokování kódu v rozhraní TwinCAT - setování C



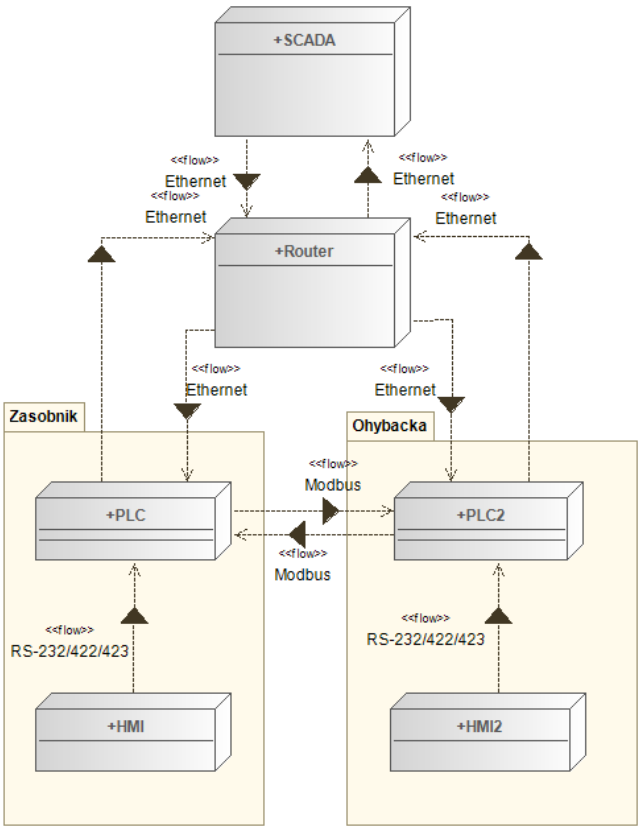
Obrázek 7.17: Krokování kódu v rozhraní TwinCAT - resetování proměnných po skončení cyklu

7.3 Jiné využitelné diagramy pro návrh systému

V této části uvedu další diagramy, které by mohly být užitečné spíše z hlediska organizace projektu, nežli generování kódu pro PLC.

7.3.1 Diagram nasazení

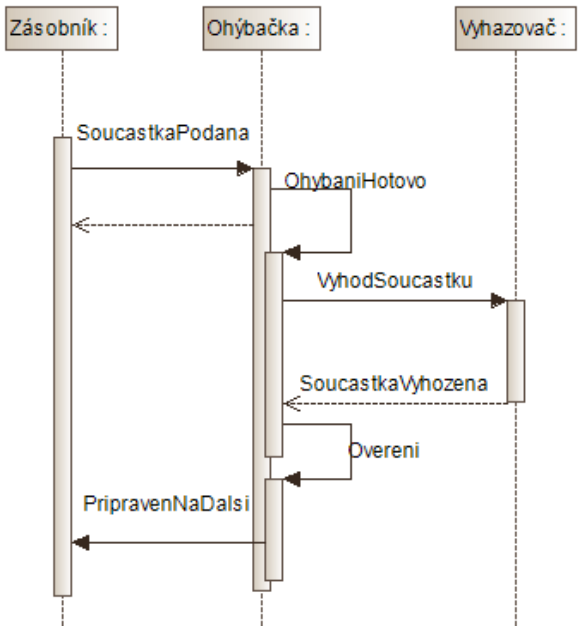
Při použití více zařízení je dobré si rozvrhnout jejich komunikaci, kterou jsem udělal pomocí diagramu nasazení, který je možné vidět na obrázku 7.18. Tím se ustanoví použití prostředků a potřeba balíčků nebo knihoven v programech. Všechny komponenty spolu mající přímý vztah jsou uvedené do jednoho balíčku a pro tento konkrétní diagram jsem se nechal inspirovat syntézou řešených úloh. Zásobník má své PLC a HMI a Ohýbačka také. Obě PLC spolu komunikují a veškeré informace se můžou pomocí síťového routeru, nebo jiné sběrnice posílat do SCADY.



Obrázek 7.18: Diagram nasazení pro syntézu úloh

7.3.2 Sekvenční diagram

Pro posílání zpráv mezi jednotlivými PLC v sekvenci, se může tento diagram dobře využít. Z něho je dobře vidět, která část programu je závislá na jakém zařízení. Na obrázku 7.11 je sekvenční diagram opět inspirován syntézou úloh. Na první čáře života je zásobník, který podává předmět ohýbačce, která je na druhé čáře. Jakmile je součástka podána, nastaví se proměnná a pošle zprávu, že ohýbačka může začít ohýbat. Jakmile je dokonáno ohýbání, pošle zprávu samo sobě ať vyšle signál, že má vyhazovač ohnutý objekt vyhodit. Po vyhození objektu ohýbačka ověří, že má volné pracovní místo a pošle zprávu zásobníku, že je připravený na další součástku.



Obrázek 7.19: Sekvenční diagram pro posílání zpráv mezi PLC

Kapitola 8

Závěr

Ve své práci jsem nejprve shrnul rešerší na UML a SysML především s ohledem na budoucí použití v oblasti řídící techniky. SysML je z části odvozený od UML2 a z části je to nový jazyk s novými prostředky pro modelování systémů. Ve své práci používám prvky UML i prvky SysML, kterých je však po skromnu, jelikož jsou to drobné úpravy z klasického UML. Většina výrazových prvků SysML jsou orientovány pro použití v systémech spojitého řízení, kde se do systému pomocí parametrů dají vložit diferenciální rovnice k popisu řízeného systému. Nejužitečnějším pro generování kódu jsem shledal diagram tříd, jelikož se z něj již dlouhá léta generuje struktura programového celku, která je doplněna v procesu implementace. Pro generování chování systému jsem zvolil stavový diagram. Ten je možné algoritmizovat jako konečný automat a zobrazuje stavy, ve kterých se může nacházet. Nedílnou součástí mé práce bylo naprogramovat úlohy v jazyce Structured Textu definovaném standardem IEC 61131-3. Ve třetím vydání tohoto standardu je pasáž o jeho objektově orientovaném programování, kterého jsem se chtěl hlavně držet a TwinCAT tento standard podporuje. Mým prvním dojmem bylo, že objektově orientované programování pro logické řízení PLC působilo předimenzovaně. Jednoduché výroky se píší složitě a musel jsem o poznatelně delší dobu vymýšlet a testovat, jak OOP ve strukturovaném textu funguje. Rozmýšlel jsem se, proč komplikovat věci, když jdou napsat jednodušeji v již ověřených způsobech jako je SFC. Porovnáním kódu v sekci 7.2.3 a 7.2.4 je rozdíl ve složitosti a přehlednosti znatelný. Ačkoliv má být objektové programování záležitostí strukturovanosti a lepší přehlednosti, ne vždy tomu tak musí být, jak je vidět v sekci 7.2.3. Tyto kódy byly ozkoušeny softwarovou simulací ve vývojářském prostředí TwinCAT a

ocenil bych i hardwarovou emulaci pro úplné odladění, ale nebyli k dispozici prostředky. Výhodou OOP je možnost lepší modifikace kódu, kde jednotlivé funkce mohou být snáze pozměněny, nebo úplně vyměněny. Generování kódu z těchto dvou diagramů by o poznání urychlilo a usnadnilo práci a vidím to jako možný potenciál a prostor pro realizaci. UML ani SysML nedisponuje výrazovými prostředky pro řízení PLC, ale v práci jsem navrhl možné alternativní metodiky pro generování ze současných diagramů. Modifikace UML pro generování PLC kódu by ulehčilo možnou realizaci programu pro generování. Jako další výhodu použití UML vidím v ostatních diagramech, které nemusí sloužit jako nástroj pro generování kódu, ale jako nástroj pro analýzu projektu. V minulosti již byly pokusy o použití UML jako nástroje pro modelování, například v publikaci [42] je navržen podobný způsob a na portálu www.ieeexplore.ieee.org je spousta článků na téma této práce, které jsou již alespoň dekádu staré, například na odkazu citace [1].

Příloha

Obsah přiloženého CD je rozdělen do následujících adresářů:

- Adresář Řešení obsahuje projekty do IDE TwinCAT, které se otevírají příponou .sln. Každé řešení je ve svém vlastním adresáři rozdělené podle úlohy.
- Adresář text obsahuje text této práce se zadáním bakalářské práce a další adresář se zdrojovým kódem textu v LaTeX. Pro vypracování jsem použil šablonu dostupnou z [43]

Pozn.: IDE TwinCAT je možné stáhnout z [44]

Bibliografie

[1] B. Vogel-Heuser, D. Witsch a U. Katzke, "Automatic code generation from a UML model to IEC 61131-3 and system configuration tools", in *2005 International Conference on Control and Automation*, sv. 2, červ. 2005, 1034–1039 Vol. 2. DOI: 10.1109/ICCA.2005.1528274.

[2] Wikipedie, *CASE nástroje*, lis. 2017. WWW: https://cs.wikipedia.org/w/index.php?title=CASE_n%C3%A1stroje&oldid=15597717 (cit. 01. 04. 2018).

[3] Select Business Solutions, Inc., *What is Computer Aided Software Engineering? (CASE) \textbar Analysis and Design \textbar FAQ*, 1988. WWW: http://www.selectbs.com/analysis-and-design/what-is-computer-aided-software-engineering (cit. 01. 04. 2018).

[4] tutorialspoint.com, *Software Case Tools Overview*, 2018. WWW: https://www.tutorialspoint.com/software_engineering/case_tools_overview.htm (cit. 01. 04. 2018).

[5] D. Čápka, *1. díl - úvod do UML*, 2018. WWW: https://www.itnetwork.cz/uml-uvod-historie-vyznam-a-diagramy (cit. 01. 04. 2018).

[6] Wikipedie, *Unified Modeling Language*, dub. 2018. WWW: https://cs.wikipedia.org/w/index.php?title=Unified_Modeling_Language&oldid=16031471 (cit. 23. 04. 2018).

[7] *Unified modeling language*, in *Wikipedie*, Page Version ID: 16031471, 21. dub. 2018. WWW: https://cs.wikipedia.org/w/index.php?title=Unified_Modeling_Language&oldid=16031471 (cit. 23. 04. 2018).

[8] S. Friedenthal, A. Moore a R. Steiner, "OMG systems modeling language (OMG SysML™) tutorial", in *INCOSE international symposium*, sv. 18, Wiley Online Library, 2008, s. 1731–1862.

[9] B. Unhelkar, *Software engineering with UML*. Boca Raton: Taylor & Francis, CRC Press, 2017, ISBN: 978-1-138-29743-2.

[10] (2018), WWW: https://is.mendelu.cz/eknihovna/opory/zobraz_cast.pl?cast=18435 (cit. 23. 04. 2018).

[11] (2018). IBM knowledge center - using SysML parametric diagrams, WWW: https://www.ibm.com/support/knowledgecenter/en/SSB2MU_7.5.3/com.ibm.rhapsody.designing.doc/topics/rhp_c_dm_parametric_dgrms.html (cit. 01. 04. 2018).

[12] (2011). UML structural models [enterprise architect user guide], WWW: https://www.sparxsystems.com/enterprise_architect_user_guide/9.3/standard_uml_models/structuraldiagrams.html (cit. 01. 04. 2018).

[13] tutorialspoint.com. (2018). UML - class diagram, www.tutorialspoint.com, WWW: https://www.tutorialspoint.com/uml/uml_class_diagram.htm (cit. 01. 04. 2018).

[14] D. Čápka, *Lekce 4 - UML - Doménový model*, cs. WWW: https://www.itnetwork.cz/uml-domenovy-model-diagram.

[15] Wikipedie, *Diagram tříd*, cs, Page Version ID: 16101882, květ. 2018. WWW: https://cs.wikipedia.org/w/index.php?title=Diagram_t%C5%99%C3%ADd&oldid=16101882.

[16] *UML: Diagramy tříd | Archiv*. WWW: http://www.milosnemec.cz/clanek.php?id=199.

[17] Miloš Němec, *UML: Diagramy tříd | Archiv*, 2010. WWW: http://www.milosnemec.cz/clanek.php?id=199 (cit. 10. 06. 2018).

[18] (2018). Dudka.cz: Softwarové inženýrství (IUS), WWW: http://dudka.cz/studyIUS (cit. 23. 04. 2018).

[19] D. Čápka. (). 5. díl - UML - class diagram, WWW: https://www.itnetwork.cz/uml-class-diagram-tridni-model (cit. 01. 04. 2018).

[20] tutorialspoint.com. (2018). UML - component diagrams, www.tutorialspoint.com, WWW: https://www.tutorialspoint.com/uml/uml_component_diagram.htm (cit. 01. 04. 2018).

[21] (2018). What is package diagram?, WWW: https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-package-diagram/ (cit. 01. 04. 2018).

[22] D. Čápka. (2018). 2. díl - UML - use case diagram, WWW: https://www.itnetwork.cz/uml-use-case-diagram (cit. 01. 04. 2018).

[23] (2018). What is interaction overview diagram?, WWW: https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-interaction-overview-diagram/ (cit. 23. 04. 2018).

[24] (2018). UML basics: The sequence diagram, WWW: https://www.ibm.com/developerworks/rational/library/3101.html (cit. 23. 04. 2018).

[25] (2018). Sysml-architect diagram-requirement - SysML user manual (english) - modelio community forge, WWW: https://forge.modelio.org/projects/sysml-user-manual-english/wiki/Sysml-architect_diagram-requirement (cit. 01. 04. 2018).

[26] S. Consulting. (4. ún. 2015). The four pillars of SysML (in 30 minutes) - YouTube, WWW: https://www.youtube.com/watch?v=998UznK9ogY (cit. 24. 02. 2018).

[27] IEC, *IEC - About the IEC*, 2018. WWW: http://www.iec.ch/about/?ref=menu.

[28] ——, *IEC 61131-1:2003 | IEC Webstore | water automation, water management, smart city*, 2018. WWW: https://webstore.iec.ch/publication/4550.

[29] ——, *IEC 61131-2:2017 | IEC Webstore*, 2018. WWW: https://webstore.iec.ch/publication/31007.

[30] ——, *IEC 61131-3:2013 | IEC Webstore | water automation, water management, smart city*, 2018. WWW: https://webstore.iec.ch/publication/4552.

[31] M. Martinásková, L. Šmejkal, České vysoké učení technické v Praze a Strojní fakulta, *Řízení programovatelnými automaty III: Softwarové vybavení*, Czech. Praha: Vydavatelství ČVUT, 2003, OCLC: 85078362, ISBN: 978-80-01-02804-9.

[32] IEC, *IEC TR 61131-4:2004 | IEC Webstore | water automation, water management, smart city*, 2018. WWW: https://webstore.iec.ch/publication/4553.

[33] ——, *IEC 61131-5:2000 | IEC Webstore | water automation, water management, smart city*, 2018. WWW: https://webstore.iec.ch/publication/4554.

[34] ——, *IEC 61131-6:2012 | IEC Webstore | cyber security, smart city, water automation, water management*, 2018. WWW: https://webstore.iec.ch/publication/4555.

[35] ——, *IEC 61131-7:2000 | IEC Webstore | water automation, water management, smart city*, 2018. WWW: https://webstore.iec.ch/publication/4556.

[36] ——, *IEC TR 61131-8:2017 | IEC Webstore | water automation, water management, smart city*, 2018. WWW: https://webstore.iec.ch/publication/33021.

[37] ——, *IEC 61131-9:2013 | IEC Webstore | water automation, water management, smart city*, 2018. WWW: https://webstore.iec.ch/publication/4558.

[38] Wikipedie, *Global variable*. WWW: https://en.wikipedia.org/w/index.php?title=Global_variable&oldid=820743105.

[39] *Beckhoff Information System - English*, eng, 2018. WWW: https://infosys.beckhoff.com/english.php?content=../content/1033/tc3_licensing/63050394963075467.html&id= (cit. 04. 06. 2018).

[40] Scott Whitlock, *TwinCAT 3 Tutorial: Multiple Virtual PLCs · Contact and Coil*, en-US. WWW: http://www.contactandcoil.com/twincat-3-tutorial/multiple-virtual-plcs/ (cit. 13. 06. 2018).

[41] *Course: Řízení programovatelnými automaty 2371527, resp. 2375007*. WWW: https://moodle.fs.cvut.cz/course/view.php?id=271 (cit. 04. 06. 2018).

[42] Lorenzo Racchetti, Cesare Fantuzzi, Marcello Bonfé a Lorenzo Tacconi, "The PLC UML State-chart design pattern", Barcelona: Lorenzo Racchetti, čvc 2015. DOI: 10.1109.

[43] *Šablona pro psaní disertační prace na čVUT FEL - LaTeX Template on Overleaf*, en. WWW: https://www.overleaf.com/latex/templates/sablona-pro-psani-disertacni-prace-na-cvut-fel/ptpvbxhsjdmg (cit. 14. 06. 2018).

[44] Beckhoff Automation GmbH & Co. KG, *TwinCAT 3 Download*, červ. 2018. WWW: https://www.beckhoff.com/english.asp?download/tc3-downloads.htm.