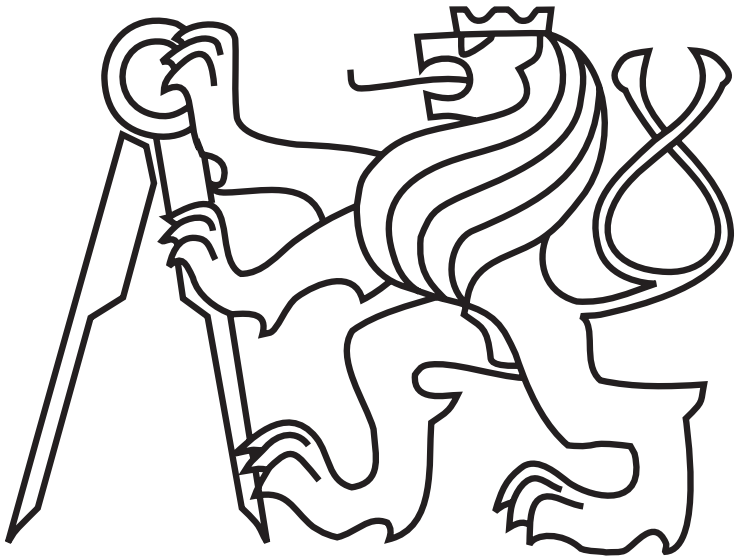
ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE

Fakulta elektrotechnická

Diplomová práce



Bc. Martin Ron

Energetické úspory v simulaci digitální továrny

Katedra řídicí techniky

Vedoucí práce: **Ing. Pavel Burget, Ph.D.**

České vysoké učení technické v Praze
Fakulta elektrotechnická

katedra řídicí techniky

ZADÁNÍ DIPLOMOVÉ PRÁCE

Student: **Bc. Martin Ron**

Studijní program: Kybernetika a robotika
Obor: Systémy a řízení

Název tématu: **Energetické úspory v simulaci digitální továrny**

Pokyny pro vypracování:

1. V prostředí Process Simulate rozšiřte funkcionalitu modelu robota o profil PROFIenergy. Vytvořte také energetický model robota na základě dat změřených v rámci jiné práce.
2. Vytvořte SW modul pro přenos externích dat jako například trajektorie robotů, parametrů spotřeby energie apod. do simulace v Process Simulate.
3. V rámci virtuálního zprovoznění spolupracujte na implementaci algoritmu uvádění linky do úsporných režimů. Algoritmus ověřte na simulované lince a na základě simulace mapujte spotřebu robotů na provozní stavy celé linky.

Seznam odborné literatury:

Profil PROFIenergy, v1.2. Profibus&Profinet International, 2011.
Process Simulate Reference Manual. Siemens Industry Software. 2013.
Process Designer Reference Manual. Siemens Industry Software. 2013.

Vedoucí: Ing. Pavel Burget, Ph.D.

Platnost zadání: do konce letního semestru 2014/2015

prof. Ing. Michael Šebek, DrSc.
vedoucí katedry

prof. Ing. Pavel Ripka, CSc.
děkan

V Praze dne 12. 2. 2014

Poděkování

Děkuji svým rodičům, kteří mě podporovali v průběhu celého studia a v době psaní této práce obzvlášť. Chci také poděkovat svému vedoucímu Ing. Pavlu Burgetovi Ph.D. za jeho vedení a cenné rady.

Abstrakt

Tato práce se zabývá návrhem pluginu pro digitální továrnu Tecnomatix - konkrétně pro její prostředí Process Simulate. Popisuji zde postup návrhu architektury pluginu pro modelování a simulaci profilu PROFIenergy, jeho programování a jeho použití při virtuálním zprovoznění modelového případu. V popisu návrhu architektury pluginu se zabývám návrhem stavového automatu pro simulaci profilu PROFIenergy, ideou jeho použití v Process Simulate a jeho napojení na již existující komponenty tohoto prostředí. V části o programování mimo jiné vysvětluji chování vybraných funkcí knihovny Tecnomatix.NET API, návrh simulačního stavového automatu a jeho začlenění do prostředí Process Simulate. Zabývám se dvěma funkčními strukturami prostředí Process Simulate CEE - logickým blokem a uživatelskou funkcí. V závěrečné části práce popisuji postup při virtuálním zprovozňování modelu výrobní buňky robotu, její simulaci a výstupní data ze simulace za použití vytvořeného pluginu.

Klíčová slova: Tecnomatix, Process Simulate, stavový automat, Tecnomatix.NET API, virtuální zprovoznění, PROFIenergy

Abstract

The scope of this thesis is about design of plugin for digital manufacturing tool Tecnomatix - with focus on its suite Process Simulate. I describe here procedure of designing architecture of plugin for modeling and simulation of PROFIenergy profile, its programming and its usage for virtual commissioning of demonstrative robotic cell. During description of design of the plugin architecture I develop system of state automata for simulation of PROFIenergy profile, I explain way of its usage in Process Simulate suite and integration with its existing components. In the part about programming beside other I describe behavior of functionalities I picked up from library Tecnomatix.NET API, I describe design of code implementing state automata and integration of its components with Process Simulate. I furthermore explain the two functioning structures of Process Simulate CEE used in my program - the logic block and the user functions. In the last part of thesis I describe procedure used for virtual commissioning of demonstrative model of production cell of robot, simulation of example and outputted data gained from simulation using developed plugin.

Keywords: Tecnomatix, Process Simulate, state automata, Tecnomatix.NET API, virtual commissioning, PROFIenergy

Prohlášení

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

V Praze dne 9. 5. 2014 ..... M. Roz .....

Obsah

Seznam obrázků

Seznam tabulek

Seznam výpisů kódu

1 Úvod

V současnosti se pozornost v naší společnosti hodně obrací na efektivitu využití zdrojů. Plýtvání je něco, co si při rostoucí spotřebě nemůžeme z dlouhodobého hlediska dovolit. Obzvlášť to platí o plýtvání energií. I když zatím je neefektivní nakládání s energiemi postiženo jen ekonomickými ztrátami, je to dostatečná motivace pro oblast automatizované výroby, kde spotřeba elektrické energie hraje nezanedbatelnou roli. Energetická krize možná nenastoupí, když budeme schopni dostatečně posílit výrobu energie, ale ekonomické hledisko zde zůstane v každém případě.

Energeticky úsporný režim elektrických spotřebičů není nic nového a zvláště v soukromém sektoru je u komplikovanějších zařízení vyžadovaným standardem. V automatizaci výroby se však dlouhodobě klade výkon na první místo a energetická efektivita je často odsouvána do pozadí. V poslední době však zájem o zefektivnění distribuce a nakládání s energií výrazně vzrostl. *PROFINET & PROFIBUS International* (PI) se rozhodl vytvořit profil, který by unifikoval principy v úsporných opatřeních. Mnoho výrobců zařízení se začíná postupně připojovat k tomuto novému vývojovému proudu. Nejspíš je PI jednou z prvních organizací, která se tímto směrem vydává. Technologie zabývající se touto problematikou je rozhodně perspektivní a dá se předpokládat, že kdo nasměruje své snažení tímto směrem teď, získá brzy náskok v rychle se rozvíjejícím odvětví automatizované výroby.

Projekt, na kterém jsem v rámci řešení své diplomové práce spolupracoval, je zaměřen právě na optimalizaci automatizované výroby z hlediska spotřeby energie. Zabývá se optimalizací robotických cest při zachování prováděných operací, plánováním pohybu materiálů ve výrobní lince a optimalizací sekvencí výrobních úkonů a také využitím profilu PROFIenergy k úspoře energie ve chvílích nečinnosti zařízení. K simulaci a modelování zkoumaných problémů využíváme software digitální továrny Tecnomatix. Odtud vzešlo zadání mé diplomové práce, protože Tecnomatix ještě nepodporuje modelování a simulaci zmíněného profilu PROFIenergy a pro testování navržených optimalizací je tato funkčnost zapotřebí.

Hlavním cílem tedy bylo naprogramovat pro prostředí Process Simulate (součást digitální továrny Tecnomatix) plugin, který by umožnil modelovat a simulovat PE profil, tento plugin otestovat a vytvořit pro něj základní energetický model robota, který je pro testování použit.

2 Metodologický rozbor

V této kapitole popíšu základní použitou terminologii a objasním volbu prostředků k řešení práce.

2.1 PROFIenergy profil

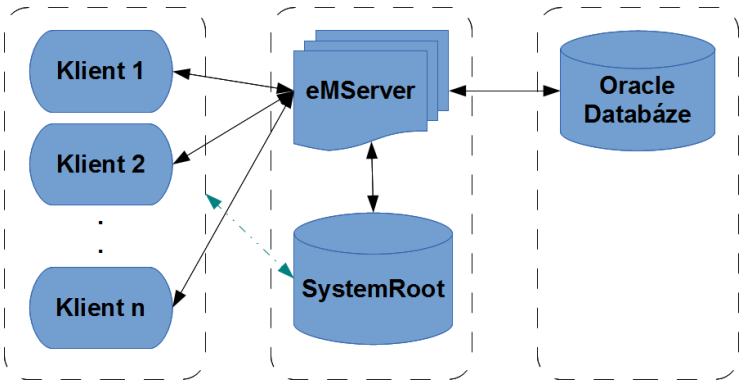
Řízení a monitorování spotřeby energie je pořád důležitější ve všech částech automatizované výroby. Z toho důvodu vznikl na základech standardu PROFINET profil PROFIenergy (dále v textu PE). Úkolem profilu PE je standardizovat rozhraní pro získávání dat o měřené energetické spotřebě a definovat sadu příkazů, které umožní převádět spotřebiče do energeticky úsporných režimů. V současnosti existuje mnoho zařízení, která nějakým způsobem měří svoji energetickou spotřebu a mohou být převedena do úsporných módů, ale tyto převody jsou často příliš časově náročné či nespolehlivé. PE profil by měl tyto nedostatky odstranit a dát do rukou uživatelů spolehlivý nástroj k měření a managementu energetických spotřeb. PE profil je zaměřen jak na autonomní rozhodování o převedení do úsporného režimu zařízení, tak na externí řízení spotřeby. [2]

2.2 Tecnomatix

Digitální továrna (Digital factory) je obecně soubor programových nástrojů, které umožňují modelovat a simulovat průmyslovou výrobu. Zabývá se částmi výroby nebo i celými továrnami. Umožňuje tak rychle vyvíjet nové pracovní postupy a nasazovat je do praxe rychleji, tím, že prototyping výroby může probíhat v simulovaném prostředí, stejně jako testování či validace. Odpadá tak nutnost odstavovat reálnou výrobnu pro dílčí testy a zkracuje výrazně dobu potřebnou na zprovozňování linky například při změně technologických postupů apod.[1][3]

Tecnomatix se snaží tomuto ideálu přiblížit. Jedná se o balík velkého množství nejrůznějších nástrojů rozčleněných do několik aplikací označovaných jako *prostředí*. Dále

v sobě integruje datovou základnu a tzv. eMServer, který zprostředkovává přístup k datové základně, spravuje přístupy a zpřístupňuje některé nástroje ke změnám na datech. Data jsou tak přístupná pro všechna prostředí, která v sobě Tecnomatix integruje a je tak usnadněna kooperace vývojářů na všech dotčených úrovních plánování a návrhu výroby. Základní architektura podle [1] je zobrazena na obrázku 1. Ve střední vrstvě je zobrazeno



Obrázek 1: Základní třívrstvá architektura Tecnomatixu podle [1]

ještě datové úložiště *SystemRoot*, které je přerušovanou čarou spojeno s vrstvou klientů. V *SystemRootu* jsou ukládána 3D modelovací data k objektům, strojová data pro simulace apod., obvykle bývá *SystemRoot* na sdíleném úložišti, ale jeho umístění pro klienty není pevné a často klientské aplikace využívají lokální kopii jen části této datové základny označované jako *SystemRoot*.

Zmíním zde dva hlavní stavební kameny Tecnomatixu, a totiž Process Simulate a Process Designer. Process Designer je nástroj určený především k plánovacím úkolům ve výrobě, umožňuje rychle zobrazovat a analyzovat data týkající se výroby a podporuje také 3D zobrazení oblastí zájmu. Zprostředkuje tedy především přístup k datům na eMServeru a umožňuje kombinovat informace získané z 3D zobrazení s dalšími plánovacími nástroji eMServeru. [4]

Process Simulate (dále v textu často jen PS) je pak určen pro designování, analyzování, simulaci a optimalizaci výrobních procesů, a to od té nejvyšší úrovně - hlediska celé továrny - až po tu nejnižší - úroveň výrobní buňky čítající třeba jediného robota. Obsahuje nástroje

pro 3D modelování výrobních zdrojů, simulaci v závislosti na čase a také v závislosti na událostech v modelu - jako například signál odeslaný z virtuálního senzoru.[3] Prostředí PS disponuje režimem nazývaným *CEE* (*Cyclic Event Evaluation*), kde řídící entitou není čas, ale simulované signály a události, například signál senzoru přiblížení apod. Naším cílem je simulovat nově se rozšiřující standard PROFIenergy, a to na úrovni jednotlivých zařízení, a proto je simulační prostředí PS v režimu CEE vhodná volba. Simulace profilu PE je v současné době do jisté míry implementována v programu Plant Simulation. V prostředí PS se objevují první náznaky zájmu o problematiku energetických spotřeb. Například KUKA RCS controller v8.3 pro PS umožňuje v rámci analýzy pohybu KUKA robotů zobrazovat simulovanou energetickou spotřebu. Odtud tedy vychází motivace pro programování rozšíření PROFIenergy funkčnosti pro Process Simulate.

2.3 Objektově orientované programování a C#

V textu zabývajícím se řešením práce často používám termíny z objektového programování. Je to výhodný postup i pro vysvětlení hierarchie datové struktury studie v Tecnomatixu, protože i ta je navržena ve smyslu objektového programování. Uvedu tedy jen velice povrchově použité pojmy.

Objektový přístup k programování se snaží o dosažení podobnosti struktury modelovaného problému s programem. Objekt v řešeném problému, například robot, obvykle dostane přiřazen objekt v programu, například třídu názvu *robot*. Robot se umí hýbat, tedy i třída bude obsahovat funkce názvu *pohyb*, která bude měnit vnitřní proměnnou *robotu*. Řekněme, že máme typ robota KUKA a máme od něj v továrně tři kusy, pak objektově modelovaná paralela bude mít třídu *KUKA* a budeme od této třídy vytvářet tři *instance* (lze chápat jako realizace). Vztah *třída - instance* lze tedy chápat jako vztah *typ - realizace*. Instance třídy musí být vytvořena tzv. *konstruktorem* - speciální metodou pro generaci instance. Pokud je konstruktor úmyslně znepřístupněn, obvykle je někde jinde vytvořena tzv. *tovární metoda*, která má ke konstruktoru zprostředkovaný přístup a zajišťuje provedení nějakých přídavných úkonů při vytváření instance. Další užitečnou vlastností

je tzv. *dědičnost*. Třída může *dědit* od jiné třídy, jako příklad mějme třídu *robot* a od ní *dědí* třída *KUKA*. Teprve až když máme ve výrobní buňce konkrétní kusy robotů KUKA, jedná se o instance. Třída také může být statická a mít statické metody. Taková třída dokáže vykonávat své funkce (poskytovat hodnoty statických proměnných, nechat na sobě volat statické metody apod.) aniž by měla vygenerovanou svoji instanci. Dalším nástrojem pro zpřehlednění kódu je tzv. implementování rozhraní. Rozhraní je zvláštní datový typ, který neobsahuje naprogramovanou funkčnost, ale pouze výčet metod a proměnných. Když některá třída *implementuje* rozhraní, znamená to, že je zaručeno, že tato třída má v sobě naprogramovanou funkčnost všech metod a obsahuje všechny proměnné, které jsou deklarovány v rozhraní. Kontrolu existence hlaviček metod a proměnných provádí překladač kódu.

Je třeba upozornit na termín *vlastnost* třídy. Je to zvláštní druh metody, která získává nebo nastavuje hodnoty proměnné v třídě. Navenek přitom vypadá jako proměnná a používá se jako proměnná. Jejím použitím jsou tak zapouzdřeny obě metody, které jsou pro získání a nastavení proměnné zapotřebí. *Vlastnost* třídy zpřehledňuje a zjednodušuje používání soukromých proměnných tím, že programátor s *vlastností* manipuluje stejně jako kdyby se jednalo o obyčejnou proměnnou.

Dále se ještě zmíním o událostech tzv. *events*. Jedná se o jakési vlajky, které říkají, že se v programu stalo něco, na co jsme chtěli dostat upozornění. Událostem se pak přiřazují posluchači - metody, které v okamžiku vzniku události nějak reagují. Například když uživatel stiskne tlačítko v uživatelském rozhraní, vznikne událost a obslužná nebo též naslouchající metoda tuto událost obslouží nějakou reakcí. Výhoda spočívá v nižším výpočetním zatížení, protože dokud nevznikne událost, žádné obslužné výpočty se neprovádí.

Nakonec k *dědičnosti*. Typicky všechny třídy mají svého předka krom jediné, od které zprostředkovaně *dědí* všechny třídy. Podobně je tomu u datové struktury Tecnomatixu, kde je nejvyšší úroveň projekt, a vše ostatní patří pod něj. Mohou zde být paralelní projekty, ale nemohou být načteny najednou v jednom prostředí.

Další otázkou byla volba programovacího jazyka. Tecnomatix.NET API nás omezuje na

jazyky pro platformu .NET, tu však podporují v základě tři programovací jazyky, které jsou zároveň dokumentovány v [5], jedná se o jazyky:

- Visual Basic .NET
- C++
- C#

Když jsem procházel dokumentaci [5], nejlépe zdokumentována byla syntaxe *C#*, proto jsem zvolil pro vývoj pluginu tento jazyk. Jako vývojové prostředí jsem použil MS Visual Studio.

Použil jsem zde pojem *API*, to znamená *Application Programming Interface*, v překladu rozhraní pro programování aplikací. Je to obvykle knihovna tříd a metod, které zprostředkují unifikovaný přístup do ucelené aplikace, aby mohla být obohacena o nějakou rozšiřující funkčnost.

3 Řešení

Nástroje pro řešení jsou dány zadáním práce. Nejprve jsem tedy prozkoumal možnosti, které nabízí knihovny softwaru Tecnomatix. Ukázalo se, že API pro vývoj aplikací pro Tecnomatix je velmi rozsáhlé, a přesto nepokrývá všechny prostředky, které bych pro programování pluginu rád využil. Tecnomatix je uzavřená aplikace bez možnosti zkoumání jejího zdrojového kódu, tato softwarová politika výrobce na mě kladla výrazná omezení, která bylo nutno obcházet či jinak redukovat. Pro vývoj pluginu jsem postupně navrhl celkem tři řešení, přičemž první dvě se dostala do slepé uličky teprve v průběhu jejich vývoje. Požadovaného výsledku jsem dosáhl až ve třetí generaci pluginu.

3.1 Formulace představy řešení

Při návrhu řešení jsem ale vždy použil základní ideovou strukturu, kterou nyní popíšu. Protože profil PROFIenergy (dále jen PE) v základu popisuje stavový automat, je páteří mého pluginu vždy stavový automat. Tento automat musí mít nastavitelné časové podmínky přechodu mezi stavy, přičemž čas se uvažuje jako podmínka jen mezi pevně danými stavy. Zároveň je zde přirozeně požadavek na to, aby použití stavového automatu hladce zapadalo do struktury simulace v prostředí Process Simulate (dále jen PS), protože plugin by měl být použit na již existující projekty a studie. Z toho také vyplývá nutnost zachovat konzistenci již existujících datových struktur studie.

Tento požadavek nelze s ohledem na ostatní požadavky stoprocentně uspokojit. Pokud použiji kvůli požadavku na kompatibilitu pouze datové struktury, které jsou v Tecnomatixu již vytvořeny, nelze s jistotou určit, zda vícenásobné použití v některé studii nepovede k narušení funkce studie. Příkladem může být třeba řešení pomocí operací. V prostředí PS operace mohou reprezentovat nesimulované časové úseky (tzv. non-sim operace), nabízí se tedy možnost využití těchto operací k simulování přechodu mezi jednotlivými PE stavy. Non-sim operace však v některých studiích už mohly být k simulování použity a mohlo by se stát, že by můj plugin pojmenoval operaci stejně a vznikla by tak duplicita, která může

vést k nestabilitě PS. Každý objekt ve studii má sice vlastní unikátní ID, jenže načítání tohoto ID z eMServeru je pomalé, obzvlášť pokud je prováděno na vzdálené klientské stanici. Takové řešení není nejvhodnější. Samotný PS nejspíše využívá nějaký způsob indexování objektů, ale není možné k tomuto indexování přistupovat pomocí API, a tak jediné spolehlivé řešení za použití Tecnomatix API je skutečně použít unikátní jméno operace. Lze samozřejmě použít dostatečně komplikované jméno, takže se duplicita stane spíše otázkou pravděpodobnosti. To je jen jedna ukázka toho, jaké problémy z posledního požadavku vyplývají.

Další výrazný požadavek je hromadné použití pluginu. Uživatel před sebou může mít desítky zařízení ve studii. U každého tohoto zařízení chce modelovat a simulovat pomocí pluginu profil PE a vzniklá režijní data musí zůstat v mezích přehlednosti typických pro zažitou praxi. Komplexita prostředí Tecnomatix přirozeně vede k ohromnému množství dat už jen v prostředí PS, a to pro každou studii. Tato data se třídí podle různých kritérií a uživatelé by měli být zvyklí pracovat s poměrně rozsáhlou množinou jednotlivých typů dat. Ovšem na nejvyšší úrovni třídění dat by neměl plugin vytvářet více než pár datových entit pro jedno zařízení, které by mělo disponovat PE funkcionalitou. Tento požadavek vyplynul z okolností až časem, kdy se ukázal v té době zvolený postup jako nevhodný právě kvůli velkému množství vytvářených datových objektů na jedno zařízení, se kterými se musel uživatel potýkat ve studii, aniž by je potřeboval přímo upravovat.

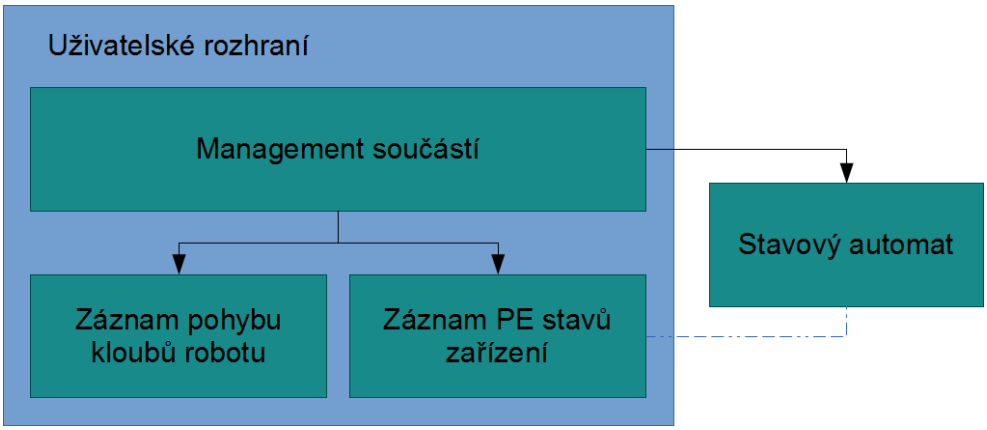
Plugin má nejen simulovat chování zařízení podle PE standardu, ale musí umožňovat i zaznamenávat průběh přechodů a setrvávání v jednotlivých PE stavech. Je tedy nutné umožnit vytvoření nějakého druhu databáze, v níž by každé dotčené zařízení mělo uchováno záznam svých stavů v závislosti na simulovaném čase. Tato databáze musí být na požadavek uživatele zpřístupněna prostřednictvím souboru, aby bylo možné v externím programu analyzovat výsledky simulace.

Všechny výše zmíněné požadavky na tomto místě shrnu.

- Integrovatelnost do prostředí Tecnomatix

- Stavový automat zahrnující časové podmínky přechodu
- Konzistence dat
- Hromadné použití

V současné verzi pluginu jsem tedy vzhledem k těmto požadavkům sledoval následující rozdělení. Zaměřil jsem se na prostředí Process Simulate (PS), protože v něm je studie simulována, je tedy logické, že na tomto místě by se měla funkčnost PE standardu zařízením přiřazovat. Základní částí pluginu je stavový automat. Tato část musí být co nejlépe integrována do PS a musí s ním co nejméně interferovat. Dalším stavebním kamenem je část pro záznam stavů v čase pro každé dotčené zařízení. Tato část bude napojena na stavový automat a na simulátor integrovaný v PS. Navíc bude podporovat export záznamů do souboru. Třetí součástí pluginu je správce předchozích dvou částí, který má za úkol přiřazovat zařízením ve studii PE funkčnost realizovanou stavovým automatem a dále má za úkol spravovat záznam stavů. Konfigurace stavového automatu pro konkrétní časování automatu načtené ze souboru také spadá do jeho pole působnosti. Všechny tři součásti musí být zastřešeny uživatelským rozhraním. Popsaná architektura pluginu je vidět na obrázku 2. Přerušovaná spojnice mezi blokem *Stavový automat* a *Záznam PE stavů zařízení* značí výměnu dat.



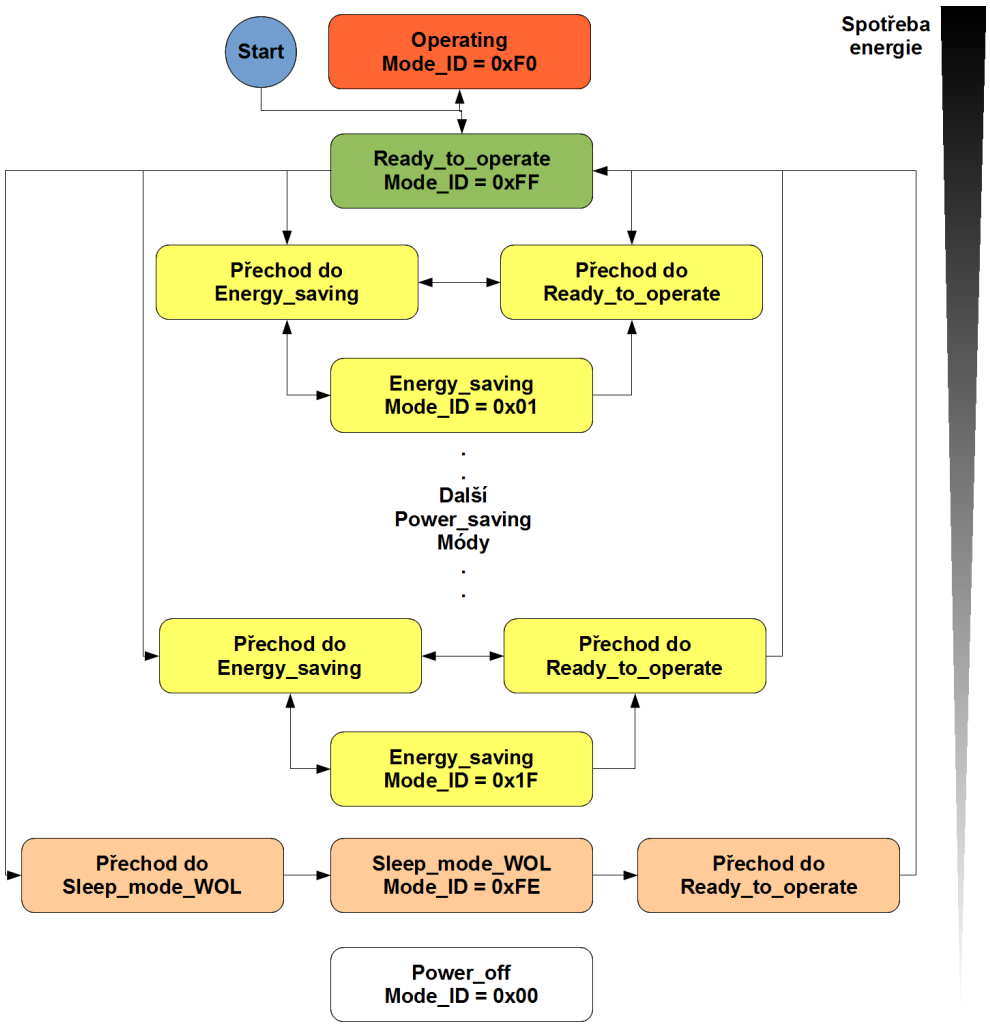
Obrázek 2: Základní architektura pluginu

Při vytváření představy o realizaci této architektury jsem se pro splnění požadavku na integrovatelnost držel myšlenky, že datová struktura musí být alespoň částečně vytvořitelná manuálním zadáváním příkazů v prostředí PS. Zkombinování již existujících stavebních kamenů by mělo vést k integrovatelnému řešení. Pokusil jsem se tedy vždy nejdříve sestavit funkční prototyp datové struktury manuálně v PS a pomocí něj simulovat PE funkcionalitu. Když jsem dospěl k funkčnímu prototypu, přešel jsem k programování automatického sestavení této struktury. Uživatel ve výsledku musí mít k dispozici nástroj, který bude s rozumnou náročností umožňovat vytváření a konfiguraci této struktury. Podobně jsem pak začal programovat také záznamovou část pluginu. Pak je na řadě refaktoring kódů a ladění.

3.2 Stavový automat PROFIenergy

Profil PROFIenergy je v [2] rozsáhle popisován do detailů. Jsou zde popsány protokoly komunikace, způsob ovládání a mnoho dalších důležitých aspektů, které jsou důležité pro výrobce, jenž chce ve svém zařízení implementovat funkci PE standardu. Pro potřeby simulace a modelování tohoto standardu v prostředí Process Simulate je pro mě však esenciální právě definice stavového automatu PE a částečně také protokol ovládání - konkrétně popis způsobu, jakým má být zařízení převáděno do jednotlivých energetických módů. Ostatní aspekty profilu PE tedy nebudu zmiňovat.

Standard PROFIenergy (PE) definuje podle [2] mód *Operate*, *Ready_to_operate*, 31 adresovatelných *Energy-saving* módů, *Sleep_mode* a *Power_off* mód. Diagram módů a povolených přechodů mezi nimi je znázorněn na obrázku 3. Ve středním sloupci se nachází PE módy, kterých zařízení může dosáhnout. Když zařízení přechází mezi módy, nevrací na vyžádání ID, které by popisovalo tento jeho stav. Z hlediska aplikace PE standardu to ani není bezpodmínečně nutné a benefity, které by z této informace vyplývaly, jsou zanedbatelné, případně nahraditelné. Aby PE standard byl samostatný modulární celek, jsou tyto postupy více než pochopitelné. Standard jako takový navíc modeluje ve svém stavovém automatu i přechodové stavy mezi jednotlivými PE módy. Jak je na diagramu 3



Obrázek 3: Diagram definice PROFIenergy standardu

vidět, standard předpokládá možnost přecházení i mezi přechodovými stavy. Také z jednotlivých *Energy-saving* módů je možno přejít zpět do stavů *Přechod do Energy-saving*. Velké množství přechodů je však podle standardu pouze volitelných. Je nutné splnit podmínku, že alepsoň jeden přechod vede do některého z *Energy-saving* módů a alespoň jeden přechod vede z *Energy-saving* módu do *Ready_to_operate* módu. Navíc jsou pak volitelné přechody mezi jednotlivými *Energy-saving* módy, to už záleží na výrobci zařízení.

Dalším volitelným módem je ještě *Sleep_mode_WOL*. Ten má oproti standardním *Energy-saving* módům zvláštní chování. Jedná se o úsporný mód vyvinutý původně pro průmyslové

počítače [2]. Zařízení do tohoto módu může přejít, ale v průběhu přechodu dojde k odpojení sběrnice a zařízení tak nemůže přijímat žádné příkazy profilu PE po sběrnici. Sběrnice se zapne až po přijetí speciálního probouzecího packetu, tzv. *Magic Packet*. Po probuzení je definováno v PE standardu, že zařízení automaticky přejde do stavu *Ready_to_operate*.

Profil PE počítá i se stavem *Power_off*, je to přirozený stav zařízení, kdy je zcela odpojeno od přívodu energie. Tento stav je terminální a podobně jako v případě stavu *Sleep_mode_WOL* není napájená sběrnice a není možné komunikací přes PROFINET zařízení zapnout. Tento stav tedy do stavového automatu není zahrnut, nepředpokládá se ani možnost vypnout zařízení pomocí nějakého příkazu PE profilu, protože by taková možnost postrádala smysl pro PE. Formálně je ale tento stav alespoň uveden a má i své Mode_ID.

Ovládání přechodů je realizováno pomocí kombinace několika signálů. Prvním je příkaz, ten může nabývat tří hodnot, a totiž *Start_Pause*, *End_Pause* a *Go_Sleep_Mode_WOL*. Při použití prvního příkazu *Start_Pause* je očekáván ještě signál s ID požadovaného *Energy-saving* módu. V případě prvních dvou řídicích příkazů signál požadovaného módu nehraje roli, při příkazu *Go_Sleep_Mode_WOL* se přejde do módu *Sleep_mode_WOL*, při příkazu *End_Pause* se začne přecházet přímo do módu *Ready_to_operate*. Pro příkaz *Start_Pause* je ještě důležitý parametr příkazu, v němž by měla být předána doba, na kterou je požadovaná pauza zařízení, než jej bude potřeba převést do stavu *Ready_to_operate*. V případě, že se zařízení dokáže autonomně rozhodovat, který *Energy-saving* mód je pro tuto dobu nejvhodnější, použije se předaný parametr jako rozhodovací kritérium.

Zařízení, které máme k dispozici, je kontroler KUKA RCS v8.2. Tento kontroler podporuje profil PROFIenergy ve velmi základní podobě. Podporuje jeden *Energy-saving* mód a podporuje i mód *Sleep_mode_WOL*. Krom toho přirozeně podporuje i *Ready_to_operate* a *Operating*. Mezi módy jsou implementovány jen základní nutné přechody, žádný z volitelných není implementován. Je sice možné vyslat požadavek na nepodporované přechody mezi stavy, ale taková akce vede k nepředvídatelnému chování kontroleru, proto jsem nepovolené přechody neuvažoval. Náš kontroler také nedisponuje autonomním rozhodováním o *Energy-saving* módech, volba úsporného režimu záleží pouze na řídicím PLC. V doku-

mentaci kontroleru jsou uvedena data k jednotlivým energetickým módům, jak je vidět v tabulce 1.

Jméno módu	Drive_Bus_OFF	Hibernate	Ready_to_operate
Jméno dle PE	Energy-saving	Sleep_mode_WOL	Ready_to_operate
Trvání přechodu do módu	5s	50s	0
Trvání přechodu do *Operate*	20s	50s	0
Minimální doba setrvání	0	10s	0
Energetická spotřeba	150W	30W	220W

Tabulka 1: PROFIenergy data z dokumentace použitého kontroleru KRC

Později se při měření spotřeby v jednotlivých stavech ukázalo, že reálná spotřeba se od té v dokumentaci mírně liší, jak je znát z grafu 24.

3.3 Vývoj architektury pluginu

Postup popisovaný v podkapitole 3.1 jsem stavěl na myšlence, že veškeré příkazy, které jsou umožněny zadávat uživateli manuálně, je možné zadávat i prostřednictvím API prostředí PS. Tato myšlenka se ukázalá být z části chybná.

V první generaci pluginu jsem modeloval stavový automat pomocí non-sim operací, které umožňovaly vynutit v simulaci časovou prodlevu. Podmínky přechodu mezi stavy pak zajišťovaly *Transition condition*, které se kontrolují na konci operace. Záznam o době strávené v jednotlivých PE stavech byl pořizován na základě informace, která operace je právě aktivní.

Toto řešení mělo samo o sobě už v prototypu slabá místa. Některé operace měly nulovou délku trvání a používaly se jako pomocné operace k rozhodování, do které z alternativních operací se přejde v dalším kroku simulace. V takovém případě operace byla při simulaci v aktivním stavu, ačkoliv už byla aktivní i jiná operace. Záleželo zřejmě na vnitřním pořadí

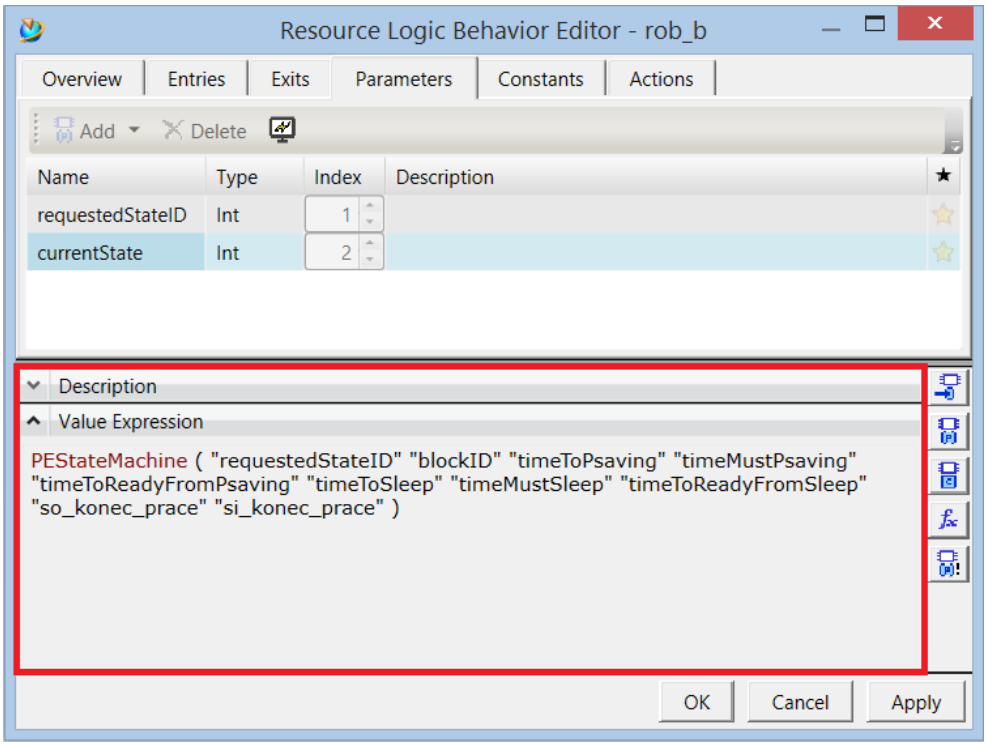
vyhodnocování operací v simulátoru. Toto chování by bylo ještě možné odfiltrovat analýzou vzniklých dat, vyvstal však jiný problém. Pomocí API nelze nastavovat *Transition condition* operací tak, jako to lze udělat manuálně v prostředí PS. Tato skutečnost nebyla zpočátku známá kvůli nedostatečné dokumentaci API, zjistil jsem ji až v průběhu řešení. V době, kdy jsem vyvíjel první generaci pluginu, byla vypuštěna verze Tecnomatix 10. V této verzi pokus o nastavení zřetězení operací přes API způsoboval pád prostředí PS. V následující verzi Tecnomatix 11.0 sice tento problém byl odstraněn, stále ovšem chybí možnost nastavovat *Transition condition* operací, takže jsem musel hledat jiné řešení.

Druhá generace pluginu byla postavena na tzv. *logických blocích*. To je struktura, která umožňuje v PS přiřazovat určitým objektům jisté logické chování. Dokonce umožňuje používat vnitřní proměnné pro jednotlivé bloky. Z mého hlediska pak nejpodstatnější vlastností logických bloků byla možnost pozdržet výstupní hodnotu o přesně definovaný časový úsek. Výstup byl pak vždy opožděn o tento čas. Vytvořil jsem tedy prototyp s řetězcem logických bloků, kde jeden logický blok reprezentoval jeden PE stav. Musel jsem se tak vzdát možnosti přiřazení stavového automatu přímo pod zařízení, jehož PE chování simuluje. Pro jeden stavový automat bylo zapotřebí celkem 9 logických bloků. Je jasné, že se vzrůstajícím počtem zařízení, na které by byl plugin použit, by neúnosně rostl i počet bloků, které by musel uživatel procházet při své práci se studií. Nevhodné smazání nebo odpojení kteréhokoli bloku by narušilo funkcionalitu stavového automatu. Toto řešení vedlo k funkčnímu prototypu a bylo i možné tento prototyp zreprodukovat za použití API, ale ze zjevných důvodů jsem od něj upustil.

Třetí generace pluginu už plně odpovídá požadované architektuře, jak je znázorněna v diagramu na obrázku 2. Stavový automat je realizován pomocí uživatelské funkce. Prostředí Process Simulate umožňuje používat takzvané logické bloky - objekty, které umožňují simulovat jistou omezenou logiku. V rámci těchto objektů mohou být nastaveny vnitřní parametry nebo výstupy na základě výsledku výpočtu této uživatelské funkce. Prostředí PS už od prvotní instalace obsahuje několik základních uživatelských funkcí, jsou jimi detekce náběžné hrany (*RE (X) - rising edge*), detekce sestupné hrany (*FE (X) - falling*

edge), dvě verze bistabilního klopného členu (*SR (X Y) - set-reset* a *RS (X Y) - reset-set*), generátor náhodného čísla (*RANDOM (minValue maxValue)*) a výpočet absolutní hodnoty (*ABS (Num)*). Při používání RS funkce jsem zjistil, že nefunguje správně - při nastavení prvního jejího parametru *X* na hodnotu *true* nespadne její výstup na *false*, jak by měl. Funkce SR naproti tomu funguje správně a liší se od RS jen pořadím vstupních parametrů. Stejně jako je tomu u RS a SR členů, i pro náš plugin je zapotřebí vnitřní paměti. Uživatelská funkce toto zjevně umožňuje a zároveň nezveřejňuje nikde v PS své vnitřní proměnné, a tak nevzniká problém s nepřehledností studie pro uživatele. Uživatelská funkce se tedy jeví jako ideální nástroj k realizaci stavového automatu.

Na tomto místě musím však upozornit na úskalí použití uživatelských funkcí v PS. Jak



Obrázek 4: Zadávání uživatelských funkcí do logických bloků

je vidět na obrázku 4, parametry uživatelských funkcí se zadávají ve volně textové podobě. To může svádět k připodobňování funkce k některým programovacím jazykům, kde jako vstupní parametr funkce může být použit výraz, či jiná vnořená funkce. Tuto funkčnost zde

ale PS nepodporuje a vstupním parametrem smí být jedině samotný čistý parametr. Kombinování funkcí a vytváření výrazů je jinak ale v poli *Value Expression* umožněno a funguje v pořádku. Jak je vidět na obrázku 4, lze toto omezení obejít vytvořením pomocného parametru, jehož hodnota je poté dosazena do uživatelské funkce. Na stejném obrázku je vidět, že jsem tuto metodu použil u parametru *requestedStateID*, který je prvním vstupním parametrem funkce *PEStateMachine* v červeně zvýrazněné oblasti. Při tomto postupu je nutné brát v úvahu, že vnitřní parametry logického bloku jsou vyhodnocovány postupně podle hodnoty v políčku *Index*. Na obrázku 4 je vidět, že nejdříve je vypočtena hodnota vnitřního parametru *requestedStateID* s indexem 1, a až poté je počítána hodnota parametru *currentState* s indexem 2 a při jejím výpočtu je použit výsledek nyní uložený v *requestedStateID*.

3.4 Uživatelská funkce - implementace stavového automatu

Když jsem vyvíjel uživatelskou funkci, aby plnila roli stavového automatu, prozkoumal jsem několik cest, jak se k tomuto cíli dostat, a ne každá cesta byla vhodná. Abych mohl navrhnout účinné algoritmy, často jsem musel testovat, jak funguje prostředí PS, aby bylo možné pochopit, proč některé postupy selhávají. Nejdůležitějšími body tohoto procesu se zabývá tato kapitola.

3.4.1 Obecné vlastnosti a životní cyklus uživatelské funkce

Uživatelská funkce má předepsanou strukturu, díky níž může být začleněna do funkčnosti prostředí PS. Implementuje rozhraní *Tecnomatix.Engineering.ITxPlcLogicBehaviorFunction* knihovny *Tecnomatix.Engineering.dll*. Toto rozhraní ale vynucuje pouze funkci *Evaluate()*, což samo o sobě nedostačuje. Funkce *Evaluate()* je volána každý krok simulace ve chvíli, kdy dojde k vyhodnocování *Expression Value*, kde je uživatelská funkce použita. Je-li použita na více místech, pochopitelně je *Evaluate()* volána vícekrát v jednom kroku simulace.

Tím se dostávám k otázce životního cyklu třídy uživatelské funkce. V průběhu vývoje pluginu se ukázalo, že při inicializaci při startu programu Process Simulate je vytvořena

jediná instance od každé třídy uživatelské funkce, a to se stane ještě před načtením jakékoli studie, tedy nezáleží, jestli je někde uživatelská funkce použita. Při startu je volán tedy konstruktor třídy (v kódu 2 je to *JmenoTridyUzivatelskeFunkce()*, řádek 12). Dále je při načtení nějaké studie, v níž je uživatelská funkce použita, zavolána metoda *ParametersTypes()*, a to právě jednou, nezávisle na počtu parametrů. Tatáž metoda *ParametersTypes()* je volána i v okamžiku, kdy uživatel otevře rozhraní pro prohlížení a editaci logických bloků (viz obrázek 4) a přejde na záložku, na níž je vidět použití této uživatelské funkce. Toto volání proběhne jen poprvé, při opakovaném zobrazení již volána není. Metoda *ReturnValueType()* je volána na začátku každého kroku simulace, a to před voláním metody *Evaluate()*.

Každá uživatelská funkce také musí mít pro svou třídu zavedený atribut (viz řádek 5 výpisu kódu 2) se jménem této funkce, které se pak zobrazí v prostředí PS. Zobrazení jména funkce je jen vedlejší role atributu, především umožňuje pomocí reflexe v běžícím programu (v mém případě běžícím PS) používat atributem indexovanou třídu. Tím jsou popsány všechny formální náležitosti uživatelské funkce.

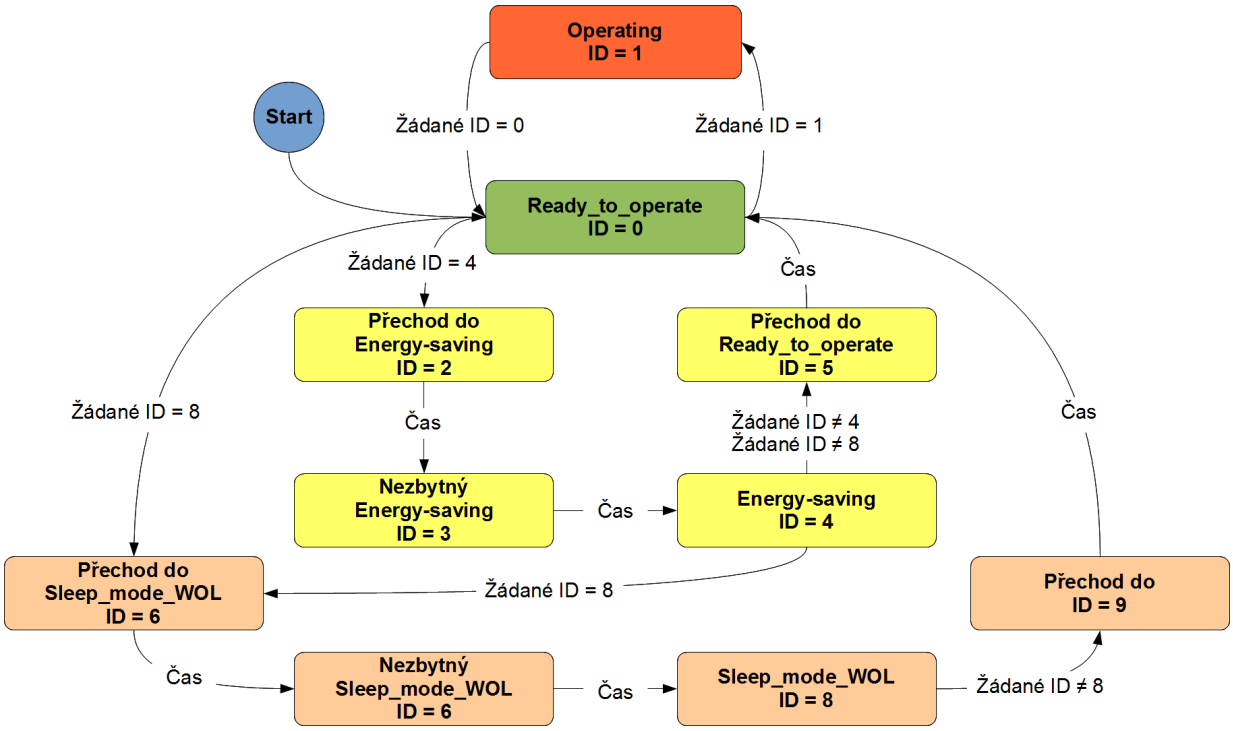
3.4.2 Použitý stavový automat

Stavový automat popsaný na obrázku 3 není pro programování simulace PROFIenergy vhodný. Na jednu stranu je příliš složitý a popisuje i možnosti, které výrobce našeho robotického kontroleru neimplementuje. Na druhou stranu je tento stavový automat o trochu jednodušší než aby bylo možné jej přímo implementovat v pluginu. Navíc ID stavů tak, jak jsou v profilu PE definována, nenechávají prostor pro zavedení ID přechodovým stavům a pro implementaci je potřeba označit každý simulovaný stav unikátním ID.

Náš kontroler podporuje jen jeden *Energy-saving* mód, mód *Sleep_mode_WOL* a mód *Ready_to_operate*, jak je uvedeno v tabulce 1. Vypustil jsem tedy ze stavového automatu všechny další hypotetické *Energy-saving* módy s tím, že jsem strukturu programu udržoval tak, aby bylo možné s rozumnou náročností později přidat další simulovatelné *Energy-saving* módy.

Dále model na obrázku 3 nemodeluje pomocí stavu fázi, kdy je zařízení v *Energy-saving* módu bezprostředně po dosažení tohoto stavu a nemůže ještě reagovat na příkazy k přechodu do jiného stavu. Pro potřeby vývoje pluginu jsem tento stav ve své uživatelské funkci zavedl a nazývám ho jako *Nezbytný Energy-saving* respektive *Nezbytný Sleep_mode_WOL*. Zjednodušuje se tak sada podmínek kontrolovaných při obdržení příkazu k přechodu mezi stavy a eventuální přidání dalších stavů se tím také stává procedurálnější činností.

Vznikl tak model stavového automatu, který je možné elegantně implementovat bez narušení požadavků na omezení či integrovatelnost PE stavového automatu. Jeho diagram je na obrázku 5. Jak je z diagramu patrné, každý stav má zavedené ID, které se neshoduje



Obrázek 5: Diagram upraveného stavového automatu pluginu

s ID definovaným v PE standardu. Jsou zde také vidět podmínky přechodů. Tam, kde je jako podmínka uveden *Čas*, tam dojde k přechodu právě tehdy, když uplyne definovaný časový úsek. Tyto časové úseky se liší zařízení od zařízení a je nutné umožnit uživateli

jejich nastavení na správnou hodnotu. Rozhodl jsem se, že seznam těchto časových konstant bude předáván uživatelské funkci jako parametr v každém kroku simulace. Tento postup byl vynucen životním cyklem uživatelských funkcí. Jedna a tatáž instance je použita pro simulaci mnoha zařízení a každé může mít jiné časové konstanty přechodů mezi stavy, takže není možné uložit časování nastálo v této instanci. Není ani možné s jistotou určit pořadí, v jakém budou jednotlivá zařízení simulována, takže ani nějaký vnitřní indexovaný seznam konstant není řešením.

Dostávám se tak k problému simulování více zařízení v jednom běhu simulace. Je už jasné, že stavový automat může být simulován jen pokud dokážeme uchovávat informaci o stavu pro každé dotčené zařízení zvlášť. Původně jsem intuitivně předpokládal, že pro každé použití uživatelské funkce je vytvořena její samostatná instance. Jak jsem popsal dříve, tento předpoklad byl mylný. Musel jsem hledat cestu, jak toto omezení obejít. Pokud by bylo možné v jediné databázi udržovat informaci o stavu každého zařízení a tuto databázi spravovat v instanci uživatelské funkce, bylo by to řešení. K tomu je potřeba nějakým způsobem indexovat jednotlivá zařízení. V prostředí Tecnomatix na eMServeru je přiděleno ke každému objektu tzv. *external ID* - unikátní řetězec znaků generovaný při vytvoření tzv. plánovacího objektu v databázi. Nastíním, co je plánovacím objektem myšleno. Objekty v PS potažmo v Tecnomatixu jsou buďto plánovací, nebo modelovací, nebo obojí. Plánovací objekt je reprezentace například robotu, která říká jen to, že robot je zde použit a má nějaké vlastnosti, ale nic neříká o jeho 3D modelu či kinematice. Tyto fyzické vlastnosti jsou zahrnuty v modelovacím objektu. Právě použitý příklad robotu je příklad objektu zároveň plánovacího i modelovacího. I modelovací objekt má unikátní popisovač - *object ID* - také řetězec znaků. Zde vzniká mírné zmatení. Prostředí PS je stavěno jako objektově orientované modelovací prostředí, existují třídy objektů a z nich se vytváří instance, příkladem je třeba třída *robot KUKA KRC5* a z něj je vytvořeno několik kusů robotů tohoto typu. Pro třídu objektu existuje také unikátní ID, a to je v atributu eMSType. Je jasné, že toto poslední ID je pro mé potřeby nepoužitelné.

Zformuloval jsem představu, čeho chci dosáhnout. V logickém bloku přiřazeném konkrét-

nímu zařízení v PS je zavolána uživatelská funkce a této funkci je předán unikátní popisovač zařízení, podle kterého instance uživatelské funkce vyhledá ve své vnitřní databázi stav, v němž se dané zařízení nachází. Bylo by logické použít jedno nebo druhé z vhodných ID výše popsaných jako identifikátor. Zde vyvstávají dvě překážky, které v takovém použití brání. Zaprvé v logickém bloku mohou být použity parametry, vstupy a výstupy jen a pouze číselných typů. Logickou hodnotu boolean považuji za číslo 1 nebo 0 vzhledem k tomu, že v logickém bloku je možné násobit booleanovskou hodnotou číselnou hodnotu ve smyslu $true = 1$ a $false = 0$. Toto chování logické hodnoty je překvapivé, ale často velmi užitečné. V předchozím odstavci jsem uvedl, že ID objektů jsou řetězce znaků, tedy neslučitelný datový typ. Není ani možné jednoduše před vstupem řetězce do logického bloku převést řetězec nějakým prostým zobrazením na číslo, protože i vstup logického bloku už musí být číslo. Odtud pak vychází druhá překážka, a totiž že nelze hodnotu ID dostat do signálu, aby tento signál pak mohl být předán na vstupu logického bloku, kam lze připojit jen signály. A opět zde vyvstává omezení, že signál smí být jedině číselná hodnota, prostředí PS nic jiného neumožňuje. Tím je vyloučeno použití již existujících ID objektů kvůli nekompatibilitě datových typů.

Je nutné tedy generovat vlastní ID zařízení pro potřeby hledání jejich stavů při simulaci. Toto ID musí být unikátní alespoň po dobu běhu simulace. Jak se později ukázalo, dočasná unikátnost ID je bohužel to nejlepší, čeho lze dosáhnout. Jsou dvě možnosti, jak zařízení ID přiřazovat. První možností je použít signál v prostředí PS. V tom případě by při simulaci musel být každý signál pevně nastavený na unikátní ID zařízení, tato hodnota se při každém spuštění simulace musí zkontrolovat a většinou znovu nastavit. Toto řešení se jeví být těžkopádné, kromě toho také přidělává uživateli starosti s novými signály, které musí spravovat. Druhou možností je uložit ID v každém logickém bloku v podobě konstanty. Na obrázku 18 v přehledu konstant v zeleném rámečku je vidět konstanta s názvem *blockID*, která plní přesně tuto úlohu. Ještě je nutné zajistit unikátnost této konstanty. Tím se zabývá PS command, který popisuji v následující kapitole.

Dále je na řadě vyřešit databázi stavů jednotlivých zařízení. Bude existovat jen jedna

databáze uložená v instanci třídy uživatelské funkce. Jako vhodný nástroj pro realizaci takové databáze se jeví slovník (generická třída jazyka C# - *Dictionary*), který umožňuje s klíčem mapovat libovolné datové typy. Vytvořil jsem výčtový typ *enStates* (energetic states) pro jednoznačný popis PE stavu. Databázi stavů zařízení tak tvoří slovník deklarace `Dictionary<int, enStates> myActualStatesDict;`. V každém volání funkce *Evaluate()*, tedy v každém kroku simulace, je ve slovníku vyhledán stav podle klíče, kterým je *blockID*. Při prvním hledání ještě není ve slovníku žádný záznam, a tak je přidán s předaným *blockID*. Díky tomu, že byl použit slovník, nemusí být ID řazena vzestupně, ale mohou být různě zpřeházena. Dále je potřeba zaznamenávat okamžik, kdy zařízení vstoupilo do některých stavů, aby tak mohlo být simulováno časování PE. K tomu jsem využil další slovník, který k jednotlivým *blockID* přiřazuje časovou známku (*timestamp*) vstupu do stavu. Použitím časových známek se předejde možným komplikacím při měření času, když je před uplynutím časového intervalu simulace pozastavena. Omezil jsem se na rozhodování o časově závislých událostech jen na základě času simulace. Později jsem rozšířil uživatelskou funkci ještě o jeden slovník, který udržoval informaci o tom, zda robot přešel v tomto kroku ze stavu *Operating* do stavu *Ready_to_operate*, aby bylo možné realizovat virtuální zprovoznění VKRC standardu, o tom podrobněji pojednávám v kapitole 4.

Stejně jako *blockID* vstupuje do uživatelské funkce i sada všech časových konstant náležejících konkrétnímu zařízení. Jejich hodnoty jsou uloženy v konstantách logického bloku a jsou nastaveny při sestavování logického bloku. Později je ovšem možno je upravit. V tabulce 2 uvádím seznam názvů časových konstant v uživatelské funkci, jejich ekvivalenty v diagramu 5 a hodnoty pro náš kontroler. Podle názvů v této tabulce je pak uložen externí seznam konstant v csv souboru. Po vystavění vstupních parametrů uživatelské funkce jsem získal hlavičku *PEStateMachine (requestedStateID blockID timeToPsaving timeMustPsaving timeToReadyFromPsaving timeToSleep timeMustSleep timeToReadyFromSleep)*, která je poměrně dost dlouhá, ale to je jen malá daň za přehlednost, kterou zapouzdřením všeho ostatního uživateli tato funkce přináší. Hned prvním parametrem je *requestedStateID*, o kterém jsem se ještě nezmínil. Jak název dává tušit, v tomto parame-

Název parametru	Popis či PE ekvivalent	Hodnota [s]
timeToPsaving	Přechod do *Energy-saving*	5
timeMustPsaving	*Nezbytný Energy-saving*	0
timeToReadyFromPsaving	Přechod do *Ready_to_operate*	20
timeToSleep	Přechod do *Sleep_mode_WOL*	50
timeMustSleep	*Nezbytný Sleep_mode_WOL*	10
timeToReadyFromSleep	Přechod do *Ready_to_operate*	50

Tabulka 2: PE časování a mapování názvů parametrů uživatelské funkce

tru se uživatelské funkci předává požadovaný PE stav, do něhož se má přejít. V tuto chvíli je na místě představit rozhodovací kritéria, podle kterých je stavový automat simulován. Definoval jsem celkem 10 stavů a 1 chybový stav pro účely ladění, jejich výčet je v tabulce 3. Základní úkony jsem zobrazil ve vývojovém diagramu 6. Předposlední blok je popsán jako *Podle aktuálního stavu a času simulace ulož do databáze nový stav*. Výkon této operace je nejnáročnější a nejdůležitější, proto se budu nyní zabývat jejím popisem.

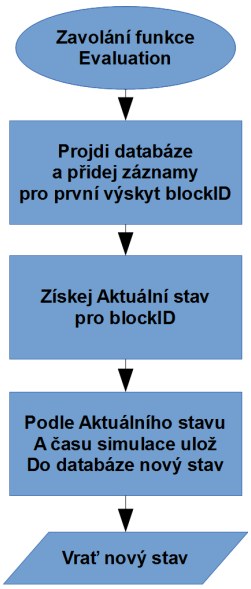
Předpokládejme, že v předchozím kroku jsem získal z databáze informaci o tom, v kterém PE stavu se tento logický blok potažmo zařízení nacházel v minulém simulačním kroku. Podle toho se pak liší reakce uživatelské funkce na požadovaný stav. Popíšu teď tyto reakce. V programu jsem použil názvy stavů, jak jsou uvedeny v tabulce 3 ve sloupci *Název stavu*.

Pro stav **Operating**:

- požadovaný stav $\neq$ *Operating* $\Rightarrow$ nastav aktuální stav na *Ready*

Pro stav **Ready** jsou 3 možné výstupy:

- nastav časovou známku na aktuální čas simulace
- požadovaný stav = *PSaving* $\Rightarrow$ nastav aktuální stav na *toPSaving*



Obrázek 6: Vývojový diagram simulace stavového automatu v uživatelské funkci

- požadovaný stav = *Sleep* ⇒ nastav aktuální stav na *toSleep*
- požadovaný stav = *Operating* ⇒ nastav aktuální stav na *Operating*

Pro stav **toPSaving**:

- (čas simulace - časová známka) ≥ *timeToPsaving* ⇒
  - nastav aktuální stav na *mustPSaving*
  - nastav časovou známku na aktuální čas simulace

Pro stav **mustPSaving**:

- (čas simulace - časová známka) ≥ *timeMustPsaving* ⇒ nastav aktuální stav na *PSaving*

Pro stav **PSaving**:

- požadovaný stav = *Ready* ⇒ nastav aktuální stav na *toReadyFromPSaving*

Název stavu	Název v PE diagramu	ID v programu
Operating	Operating	1
Ready	Ready_to_operate	0
toPSaving	Přechod do Energy-saving	2
mustPSaving	Nezbytný Energy-saving	3
PSaving	Energy-saving	4
toReadyFromPSaving	Přechod do Ready_to_operate (z ID=4)	5
toSleep	Přechod do Sleep_mode_WOL	6
mustSleep	Nezbytný Sleep_mode_WOL	7
Sleep	Sleep_mode_WOL	8
toReadyFromSleep	Přechod do Ready_to_operate (z ID=8)	9
errorState		10

Tabulka 3: Názvy stavů ve výčtu *enStates* a jejich ekvivalent v diagramu 5

- požadovaný stav = *Sleep* ⇒ nastav aktuální stav na *toSleep*
- nastav časovou známku na aktuální čas simulace

Pro stav **toReadyFromPSaving**:

- (čas simulace - časová známka) ≥ *timeToReadyFromPSaving* ⇒ nastav aktuální stav na *Ready*

Pro stav **toSleep**:

- (čas simulace - časová známka) ≥ *timeToSleep* ⇒
  - nastav aktuální stav na *mustSleep*
  - nastav časovou známku na aktuální čas simulace

Pro stav **mustSleep**:

- (čas simulace - časová známka) $\geq$ *timeMustSleep* $\Rightarrow$ nastav aktuální stav na *Sleep*

Pro stav **Sleep**:

- požadovaný stav = *Ready* $\Rightarrow$ nastav aktuální stav na *toReadyFromSleep*
- nastav časovou známku na aktuální čas simulace

Pro stav **toReadyFromSleep**:

- (čas simulace - časová známka) $\geq$ *timeToReadyFromPSaving* $\Rightarrow$ nastav aktuální stav na *Ready*

Pro případné přidávání stavů do stavového automatu je vhodné dodržovat pravidlo, že při přechodu do stavu, v němž se následně bude kontrolovat časová známka, je nutno časovou známku nastavit na aktuální čas. Tak je zajištěno, že časově závislá operace bude aktivní jen po definovanou dobu. Jak je vidět, všechny kontroly časové známky předpokládají, že čas simulace bude neklesající. V prostředí PS je možné v Sequence Editoru spustit simulaci pozpátku v režimu CEE (cyclic event evaluation), což může být nejspíš v některých situacích prospěšné, nicméně ve většině případů se tato funkce nevyužije a implementace stavového automatu tak, aby fungoval i takto zpětně, by se nevyrovnala vynaloženému úsilí. Navíc by kód přišel do velké míry o svoji přehlednost. Případy, kdy uživatel přejde v již jednou spuštěné simulaci zpět, musí uživatel opravit manuálně nastavením stavového automatu do správného stavu, a pak opětovným spuštěním simulace. Při takovém jednání je ovlivněn i záznam stavů, pokud byl při této události aktivován, a jeho výstup je třeba analyzovat. V záznamech bude více údajů ve stejném časovém intervalu, a to je znalost, kterou lze při korekci dat využít. Záznamem stavů se zabývám podrobně v kapitole 3.5.2.

Tím je uzavřen blok *Stavový automat* v obrázku 2. Vzhledem k tomu, že funkčnost stavového automatu přichází do prostředí PS až s uživatelskou funkcí, není možné dodržet

postup vývoje pluginu, jak byl popsán v úvodu kapitoly. Konkrétně se jedná o krok, kdy je nejprve prototyp funkční datové struktury manuálně sestaven v prostředí PS, aby tak byla dokázána integrovatelnost a kompatibilita s PS. Sestavování funkčního prototypu tedy přišlo na řadu až po naprogramování uživatelské funkce. Po vyladění uživatelské funkce jsem pak mohl už přistoupit k automatizaci sestavení funkční struktury - logického bloku, v němž je uživatelská funkce použita. Na tomto místě je vhodné upozornit, že tuto funkci lze k simulování PE standardu použít samostatně bez dalších nástaveb, pokud uživatel ví, jak tato funkce funguje. K praktickému hromadnému použití je ale zapotřebí spravující nadstavbová aplikace, kterou popisuji v kapitole 3.5.

3.5 Process Simulate Command

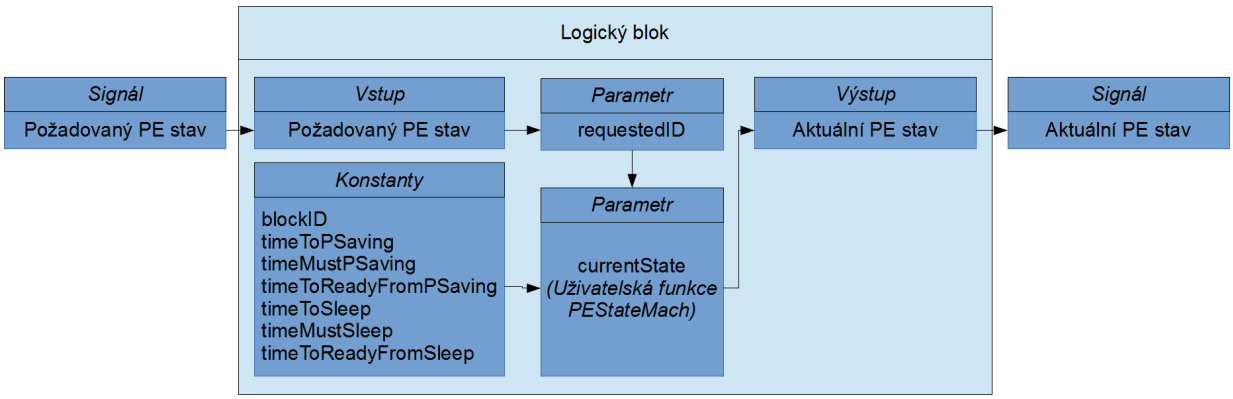
Představa, že by uživatel měl provést manuálně desítky malých kroků pro každé zařízení, kterému chce přiřadit chování podle PROFIenergy standardu, je přinejmenším odrazující. Pro pohodlné použití pluginu je také nutné uživateli dodat uživatelské rozhraní konzistentní s těmi, na které je v rámci nadřazené aplikace zvyklý. Na programu Process Simulate je znát, že je složen z velkého množství dílčích aplikací, které původně stály samostatně, a konzistence uživatelských rozhraní není ani v něm plně dodržena. Má však společné prvky a zdá se být logické, že jeho vývojáři mají tendenci směřovat k určitému sjednocení vzhledu a ovládání. Jedním z bodů zadání práce také bylo naprogramovat SW modul pro přenášení externích dat z a do prostředí Process Simulate. V průběhu řešení práce vznikl konkrétnější požadavek na umožnění exportovat časový záznam otočení kloubů vybraných robotů ze simulace do souboru.

Všechny tyto požadavky lze splnit v rámci možností Tecnomatix.NET API pomocí tzv. *command*. Je to funkce typická pro tlačítko - uživatel klikne na tlačítko, a to vyvolá sled nějakých akcí. Když je akce jednou dokončena, není command aktivní, dokud uživatel opět neklikne na tlačítko. Command také umožňuje vytvářet komplexnější uživatelské rozhraní než jen jedno tlačítko, umožňuje použít většinu GUI (graphical user interface) prvků, které v prostředí PS najdeme.

3.5.1 Přiřazení logického bloku

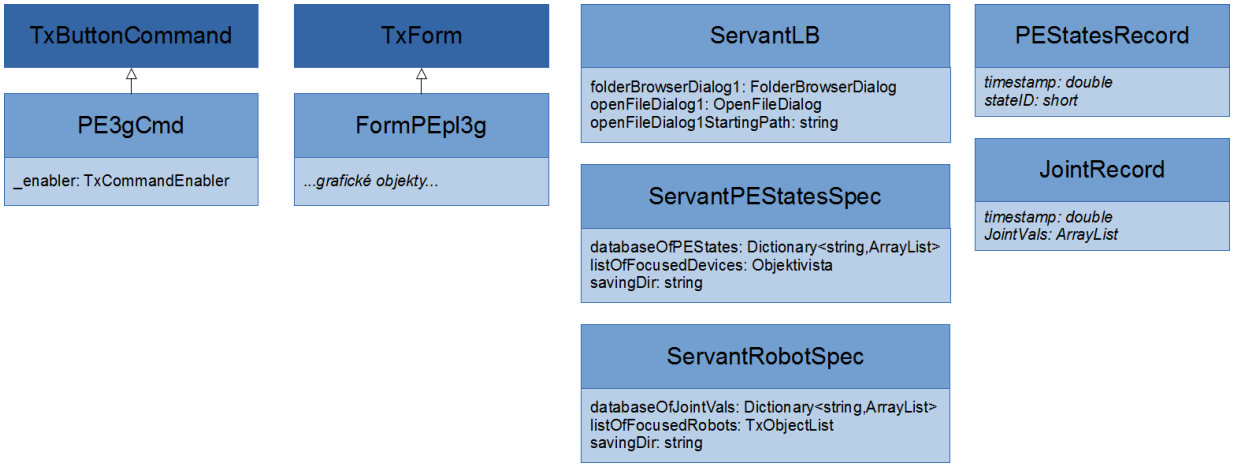
Pro simulování PE standardu je nutné použít logický blok, který umožní použít uživatelskou funkci na svůj vnitřní parametr či na svůj výstup. V PS lze logický blok vytvořit zcela samostatně bez návaznosti na kterýkoli objekt ve studii. Z objektového hlediska nelze mít samostatně stojící paralelní strom objektů s nezávislým kořenem, všechny objekty musí mít společného předka, či majitele. Logický blok je považován za modelovací objekt a při jeho samostatném vytvoření se zařadí jako potomek či majetek statické vlastnosti *LogicalRoot* náležející statické třídě *TxDocument*. Takto samozřejmě lze také v logickém bloku simulovat PE standard, a tím simulovat nějaký virtuální objekt, který není ve studii vidět. Praktičtější je však použít možnost přiřadit logický blok objektu, který tuto možnost podporuje. Obecně každý objekt implementující rozhraní *ITxPlcLogicResource* může dostat logický blok přiřazen. Prakticky toto rozhraní požaduje pouze dvě věci - zaprvé implementaci vlastnosti *LogicBehavior*, v níž se ukládá reference na objekt logického bloku, a zadruhé implementaci veřejné metody *CopySelfLogicToOtherLogicResource*. Logický blok je v API zastoupen třídou *TxPlcLogicBehavior*. Pro potřeby programování pluginu nás však nebude příliš zajímat, které objekty mohou logický blok pojmout, ani nebudeme množinu takových objektů rozšiřovat. V zadání práce je určeno, že standard PROFIenergy má být aplikován na model robota v PS, takže středem našeho zájmu je třída *TxRobot*, která rozhraní *ITxPlcLogicResource* implementuje, čímž je základní požadavek splněn a dále se o něj není třeba starat.

Pro logický blok platí ještě další zvláštnost, a totiž dokud není jeho vstup či výstup připojen na signál, není blok se zárukou simulována a je v jakési latentní fázi. Když uživatel chce simulovat PE standard, stejně musí k bloku připojit řídicí signály, čímž simulaci bloku aktivuje. Prototyp logického bloku je zobrazen v diagramu na obrázku 7. Zde zobrazené parametry, vstupy a výstupy jsou jen požadované minimum, uživatel může logický blok rozšířit o libovolnou další funkčnost (vstupy, parametry atd.), dokud se změny nedotknou zde uvedených prvků, nebude dotčena ani PE funkčnost. Máme tedy prototyp, command musí jen tento prototyp sestavit, správně propojit vnitřní proměnné a nastavit konstanty



Obrázek 7: Prototyp logického bloku

na požadované hodnoty. Tady uvedu zjednodušený UML diagram tříd pluginu 8. Neuvádím výčet metod tříd, pouze výčet vnitřních proměnných tříd.



Obrázek 8: UML diagram commandu

Přiřazení logického bloku objektu robotu probíhá následovně. Nejprve je zkontrolováno, zda robot už má přiřazený logický blok. Pokud ano, pracuje s tímto logickým blokem, pokud ne, je vytvořen nový logický blok (instance třídy *TxPlcLogicBehavior*). Vytvoření instance třídy *TxPlcLogicBehavior* nelze provést prostým volání kontruktoru třídy. Instanci lze vytvořit jen pomocí tovární metody *CreateLogicBehavior(TxPlcLogicBehaviorCreationData data)* zavolané na instanci objektu implementujícího rozhraní *ITxPlcLogicBehaviorCre-*

ation. Vstupním parametrem je instance třídy *TxPlcLogicBehaviorCreationData*, kterou už lze vytvořit obvyklým voláním konstruktoru třídy. V této instanci je nutné před jejím použitím nastavit všechny parametry, které chceme nastavit instanci *TxPlcLogicBehavior*, pozdější změny parametrů již existující instance třídy *TxPlcLogicBehavior* vedou k nestabilitě prostředí PS. Tento postup vytváření objektů, kdy se nejprve vytvoří datový objekt, který vnáší do tovární metody inicializační informace, je v Tecnomatix.NET API zavedeným standardem a principiálně je vždy stejný. Použití továrních metod zde má zajistit, že uživatel nebude vytvářet sirotčí objekty - tedy objekty, které by neměly svého majitele.

Poté se otevře .csv soubor, v němž jsou uloženy časy přechodů mezi PE stavy a načtou se hodnoty časových konstant s pevně danými jmény, jak jsou uvedena v tabulce 2 v prvním sloupci *Název parametru*. Soubor je ve formátu .csv, hodnoty jsou oddělené čárkou a pro neceločíselné hodnoty je použita desetinná tečka. V prvním sloupci je pevně daný název časové konstanty, v druhém sloupci pak její hodnota. Další sloupce na řádku jsou ignorovány. Pořadí řádků může být libovolné a v případě, že se na řádku vyskytuje neznámý název časové konstanty, je tento záznam ignorován. Všechny nenalezené časové konstanty jsou v logickém bloku nastaveny na hodnotu 0. Příklad obsahu souboru je vidět ve výpisu 1. V logickém bloku se pak hledají podle jména výskyty konstant (instance třídy *TxPlcLo-*

```
timeToPsaving, 5.3
timeMustPsaving, 0
timeMustSleep, 10.0
timeToReadyFromSleep, 50
```

Výpis 1: Příklad obsahu .csv souboru s PE časováním

gicBehaviorConstant), prochází se pole konstant, jehož adresa je uložena ve vlastnosti logického bloku *Constants*. Pokud je nalezen první výskyt daného jména konstanty, které odpovídá názvu konstanty z .csv souboru s časováním, přiřadí se její vlastnosti *Value* příslušná hodnota ze souboru. Pokud konstanta není v logickém bloku nastavena, je vytvořena její nová instance s názvem, který nebyl nalezen. Opět musí být použita tovární funkce instance třídy *TxPlcLogicBehavior* s názvem *CreateConstant* za použití *Creation-*

Data, ve stejném smyslu, jako při vytváření instance logického bloku. Stejným způsobem se pak vyhledává konstanta *blockID*, a pokud nebyla nalezena, vytvoří se s nulovou hodnotou.

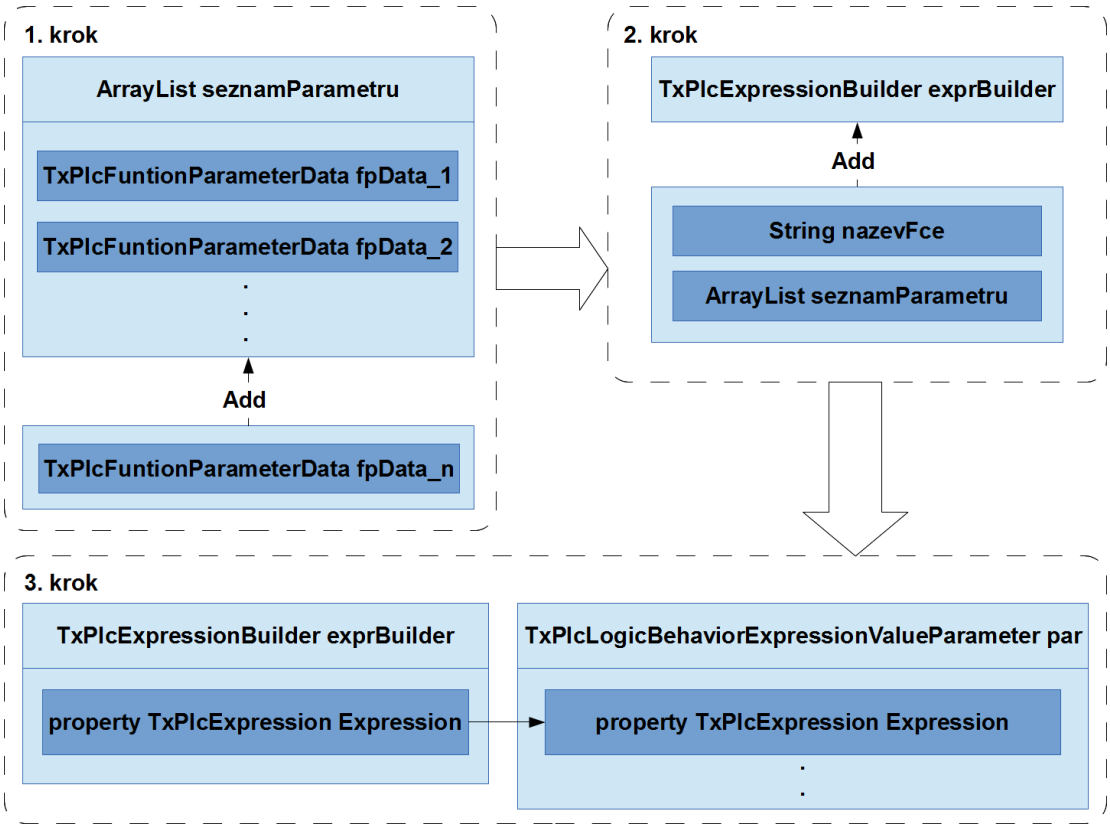
Nastavení konstanty *blockID* je tenké místo pluginu v současném stavu. Jak jsem vysvětlil dříve, musí to být unikátní hodnota alespoň v rámci jednoho každého běhu simulace. Připomenu také, že PS je objektově orientované modelovací prostředí. Aby bylo možné manuálně vytvořit logický blok a přiřadit ho zařízení, které to umožňuje, musí být toto zařízení přepnuto do tzv. modelovacího módu. V tomto módu jsou umožněny změny 3D modelu objektu a změny logického bloku, to jsou obojí data uložená v souboru .cojt na sdíleném úložišti digitální továrny (Tecnomatixu), a můžeme využít představu, že otevřením objektu pro modelování se upravuje lokální dočasná kopie těchto dat. Uzavřením modelování objektu tato data pak přepíší ta původní zdrojová. Protože by bylo datově náročné ukládat do samostatného .cojt souboru každou instanci modelovacího objektu v PS, načítají se pro všechny instance 3D data a přiřazený logický blok z jednoho rodičovského souboru. Když se teď vrátím zpět a zopakuji, že potřebuji pro každou instanci robota jednoho typu unikátní konstantu, která je uložena v .cojt zdrojovém souboru, je jasné, že při manuálním vytvoření a uložení logického bloku (ukončením modelovacího módu objektu) se přepíší všem ostatním výskytům instancí tohoto objektu hodnoty konstant *blockID* na tu poslední nastavenou hodnotu. U časových konstant je to vhodné chování, ale pro ID je to problém.

Přijal jsem tedy fakt, že po uložení jsou hodnoty *blockID* pro všechna zařízení s PE funkčností stejné. Musel jsem tedy vytvořit funkci, která projde dotčená zařízení a zajistí unikátní *blockID*. Zjistil jsem, že pomocí API lze měnit nastavení logických bloků, aniž by bylo zařízení otevřeno pro modelování, a po dokončení nastavení si každé zařízení drží ID, které mu bylo nastaveno. Je to tím, že sjednocení hodnot je vyvoláno až příkazem k ukončení modelování, a protože objekt vůbec nebyl pro modelování otevřen, ke sjednocování nedojde. Prostředí PS si pro období, kdy má načtenou studii, vytvoří v paměti dočasné kopie objektů a data v nich uchovává v každém tomto objektu zvlášť, proto je toto chování možné.

Napsal jsem tedy skript, který projde všechny instance třídy *TxPlcLogicBehavior* ve studii a pokud v nich nalezne konstantu s názvem *blockID*, přenastaví její hodnotu tak, že čísluje tato zařízení od nuly vzestupně. Dokud uživatel neotevře pro modelování a opět nezavře některý dotčený objekt, zůstávají ID unikátní. Tento skript proběhne před každým spuštěním simulace, ale pro případ špatného použití pluginu jsem umožnil uživateli toto přečíslování vyvolat i nezávisle na simulaci. Tím jsou hotovy konstanty logického bloku.

Další se vyhledá parametr s názvem *requestedStateID*, není-li nalezen, je vytvořen obdobným postupem, jako konstanta nebo logický blok. To, že *požadovaný PE stav* v diagramu na obrázku 7 je vstup, který nevede přímo do uživatelské funkce *PEStateMach*, má svůj důvod. Obecně může uživatel dojít k výslednému požadovanému PE stavu i za pomoci sestavení nějaké vnitřní logiky v logickém bloku. Kdyby vstup přímo vedl do *PEStateMach*, vedlo by to k jeho většímu omezení. Další je na řadě parametr *currentState*, pokud nebyl nalezen, vytvoří se a do něj se vloží uživatelská funkce. Postup pro to je komplikovanější, jeho schématické zobrazení je na obrázku 9.

Pro použití uživatelské funkce pomocí API je třeba připravit poměrně komplikovaný datový objekt, který slouží jako nosič parametrů pro tovární funkci vytvářející instanci uživatelské funkce. V popisu budu postupovat zvnějšku struktury dovnitř. Předpokládejme, že mám instanci parametru logického bloku. Tato instance má vlastnost názvu *Expression*, typu *TxPlcExpression*. Do této vlastnosti se ukládá objekt, který lze získat jedině z vlastnosti *Expression* objektu typu *TxPlcExpressionBuilder*. Tento objekt lze vytvořit přímo konstruktorem a jeho vlastnost *Expression* je naplněna pomocí volání jeho metody *Add*. Tím se sestaví celý matematickologický výraz. Potřebujeme do expressionBuilderu dosadit naši uživatelskou funkci. To se provede přetíženou verzí metody *Add("PEStateMach", seznamParametru)*. První parametr metody *Add* je *atribut*, který jsme přiřadili naší uživatelské funkci v kapitole 3.4.1, PS engine podle něj vyhledá příslušnou třídu a použije ji k výpočtu. Druhým parametrem metody *Add* je *seznamParametru* typu *ArrayList*. *ArrayList* je podobně jako *Dictionary* generická třída a nezáleží jí na typu objektu v ní uloženém, ale přesto v tomto případě v něm musí být uložený typ *TxPlcFunctionPara-*



Obrázek 9: Postup vytvoření uživatelské funkce pomocí API

meterData, jehož instance je možno vytvořit přímým voláním jeho konstruktoru. Instance této třídy disponují deseti metodami, které jí nastavují funkci. Může to být funkce vstupu, konstanty, parametru, či výstupu logického bloku a další. Všechny potřebné konstanty a parametry jsou zařazeny do *ArrayListu* označenému jako *seznamParametru*, a pak je *Expression* vyvolán v jediném volání vlastnosti expressionBuilderu. Je na místě upozornit na zvláštnosti chování vlastnosti *Expression*, a totiž že v okamžiku vyvolání její hodnoty je její obsah smazán, takže tuto hodnotu lze vyvolat pouze jednou, o její zachycení se musí programátor postarat. Druhé úskalí spočívá v použití instancí třídy *TxPlcFunction-ParameterData* - pro každý "symbol"musí být vytvořena nová instance této třídy, jinak by v *ArrayListu* byl uložený jen seznam ukazatelů na stejnou instanci a všechny vstupní parametry uživatelské funkce by byly stejné jako poslední přidaný parametr.

Na konci provádění celé sekvence se do stavového řádku prostředí PS vypíše zpráva o dokončení úlohy. Díky tomu, jak je tato sekvence napsána, se při jejím opakovaném spouštění neduplikují žádné vnitřní parametry logického bloku, a pouze se přenastavují hodnoty konstant a případně se opraví narušené pospojování uvnitř bloku.

3.5.2 Záznam PE stavů robotů

Původně jsem zamýšlel zaznamenávat PE stavy přímo v místě, kde se o nich rozhoduje v průběhu simulace, to je v třídě uživatelské funkce, abych příliš nezvyšoval výpočetní nároky na každý krok simulace. Takové řešení naráží na zásadní problém, a totiž není možné na okamžitý příkaz uživatele tento záznam uložit do souboru. Bylo by sice možné zavést do uživatelské funkce další vstupní parametr, který by spouštěl akci uložení do souboru, ale tato akce by mohla proběhnout jedině v průběhu běžící simulace, jindy funkce není vyhodnocována. Ukládání při běžící simulaci se rozchází s filozofií prostředí PS, takže tento postup jsem zavrhl. Při zkoumání chování událostí jsem na tomto místě také zjistil, že v API nefungují události vyvolané koncem či začátkem simulace.

Přistoupil jsem tedy k jinému řešení. Idea je taková, že databáze všech záznamů PE stavů bude uložena v objektu commandu, a uživatel tak bude moci se záznamy manipulovat obvyklým způsobem, nejen při běžící simulaci. Bude se ovšem muset každý krok simulace jednak simulovat PE stavový automat a jednak zachytávat na výstupu logického bloku informaci o aktuálním PE stavu v daném kroku simulace. To ale není příliš významné zvýšení výpočetní zátěže, při testování na třech robotech simulovaných současně nebyl rozpoznatelný rozdíl v zatížení použitého hardwaru. Protože v běžícím PS může existovat pouze jedna instance commandu, bude uchovávaná databáze také jediná a záznamy všech robotů v této databázi musí být indexovány pro jednotlivé roboty. Protože command přistupuje k celým objektům a nemusí se podřizovat žádným omezením datových typů na rozhraní, může použít jako popisovač robotů jejich atribut z eMServeru - jejich *external ID*. Díky tomu je indexování logičtější a také přehlednější. Databázi stavů v commandu realizuje opět instance generické třídy *Dictionary* s deklarací typové dvojice ⟨*klíč, prvek*⟩ jako

⟨*string, ArrayList*⟩. Přitom v políčku klíč je zmiňované externí ID a v *ArrayListu* je uložen seznam záznamů ve formě instancí mé třídy *PEStatesRecord*. V každé instanci záznamu je uložena časová známka a k ní příslušný PE stav, respektive jeho ID. Seznam jmen PE stavů a jejich ID je vidět v tabulce 3. Metoda, v níž se záznam v krocích simulace provádí, je zaregistrována jako posluchač události *SimulationPlayer.TimeIntervalReached*, přičemž vlastnost *SimulationPlayer* typu *TxSimulationPlayer* je dostupná ze statické třídy *TxApplication* - z vlastnosti *ActiveDocument*. Registrováním a odregistrováním metody jako posluchače této události je ovládáno zahájení a ukončení nahrávání PE stavů.

Naplněnou databázi je pak možné uložit do souboru, aby simulovaná data mohla být předána dále k externí analýze. Dohodli jsme se na formátu .csv, přičemž pro každý robot je vytvořen samostatný .csv soubor. Jednotlivé soubory jsou pojmenovány externím ID příslušných robotů, čímž je zachována filozofie souborů prostředí PS. První řádek souboru obsahuje čárkou oddělené popisy sloupců. Dále pak jsou v prvním sloupci časové známky a v druhém sloupci ID PE stavu podle tabulky 3. Pro neceločíselné hodnoty je použita desetinná tečka. Uživatel si může při vyvolání příkazu uložení vybrat umístění souboru, případně je uložen záznam automaticky do osobní složky uživatele momentálně přihlášeného do systému Windows. V osobní složce je vytvořen adresář s názvem *TecnomatixPEpluginData* a v něm je vytvořena složka s aktuálním datem, pokud už tam předtím tato adresářová struktura nebyla, uloží se do této složky všechny .csv soubory s názvy externích ID sledovaných robotů. Pokud už tam adresářový strom se složkou s aktuálním datem existoval, .csv soubory se stejným jménem se přepíší.

3.5.3 Záznam polohy kloubů robotů

Záznam poloh kloubů robotů jsem řešil stejným přístupem jako záznam PE stavů. Princip se liší v několika detailech. Zaznamenávaná data nejsou typu signálu vystupujícího z logického bloku, ale typu *TxPoseData*. Instance této třídy má jen jednu vlastnost - *JointValues* - a v této vlastnosti ukládá *ArrayList* se seznamem aktuálních hodnot poloh kloubů robota. Tyto hodnoty mají typ *double* a pro úhlové hodnoty mají rozměr v radiánech.

Vytváření záznamu jsem řešil opět v metodě, která je registrována jako posluchač události *SimulationPlayer.TimeIntervalReached*. Aktivuje se také registrováním a deaktivuje odregistrováním metody jakožto posluchače, stejně jako při záznamu PE stavů. V každém kroku simulace se ukládá do databáze tvořené pomocí *Dictionary⟨string, ArrayList⟩* do seznamu u klíče externího ID robota instance datového záznamu mé třídy *JointsRecord*, která v sobě uchovává dvě vlastnosti - časovou známku a *ArraList*, do něhož se ukládá výše zmíněný seznam *JointValues* s hodnotami natočení kloubů robotu.

Uložení dat z databáze je opět velmi podobné principu z předchozí kapitoly 3.5.2. Rozdíl je v tom, že robot může mít obecně různý počet kloubů, a tak .csv soubor, do něhož data ukládám, může mít různý počet sloupců. Soubor na prvním řádku obsahuje čárkami oddělené popisky sloupců a v dalších řádcích data. V prvním sloupci je vždy časová známka a v dalších sloupcích jsou pak hodnoty jednotlivých kloubů seřazené tak, jak jsou seřazené v seznamu *JointValues* v prostředí PS. Desetinný oddělovač je desetinná tečka.

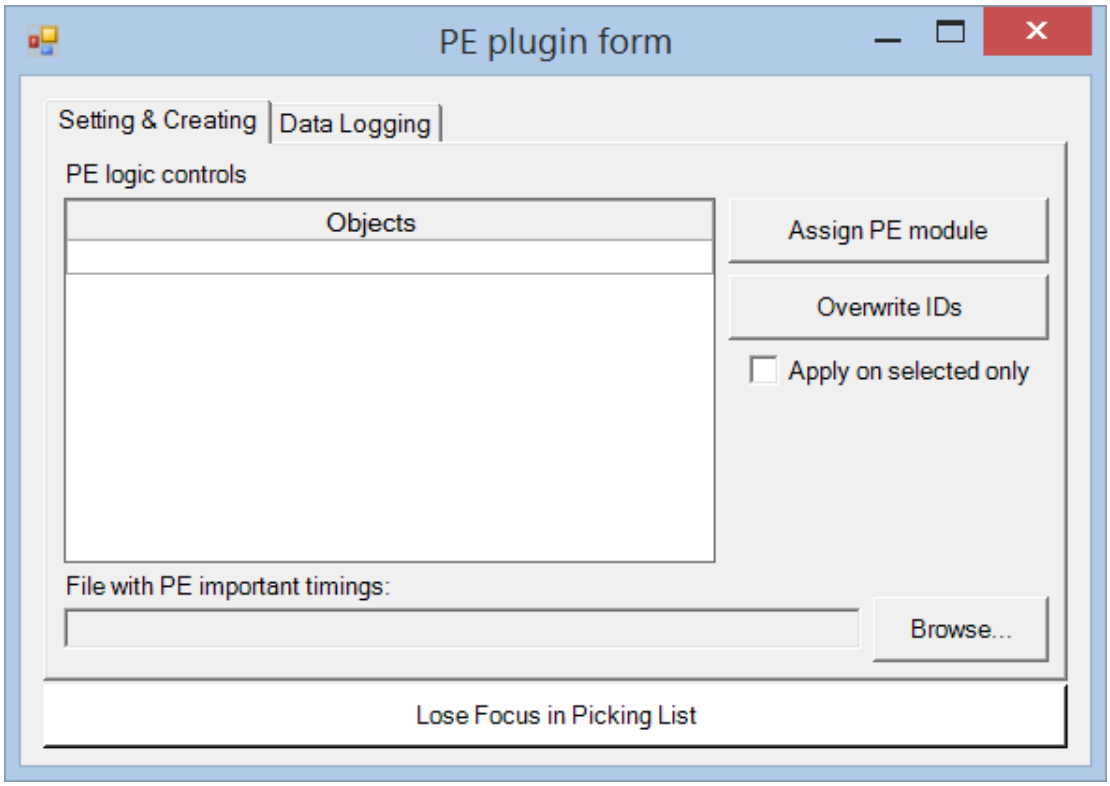
Pro upevnění představy o architektuře programu doporučuji nahlédnout do přiložené dokumentace programu pluginu, kde je funkce každé metody popsána. Zde uvedené vysvětlení doprovází UML diagram na obrázku 8.

3.5.4 Grafické uživatelské rozhraní

Když je funkčnost programu implementována a byla navržena i představa o postupu uživatele při použití programu, je dalším krokem vytvoření uživatelského rozhraní. Pro použití standardních grafických ovládacích prvků v prostředí PS je v Tecnomatix.NET API dostupná knihovna *Tecnomatix.Engineering.Ui.dll*. Tento soubor lze najít v instalačním adresáři *../Tecnomatix/eMPower* a po nastavení reference na něj v programovacím prostředí MS Visual Studio je pak zpřístupněn *GUI Designer*, s jehož pomocí lze efektivně vytvářet GUI commandu. Upozorním zde ale, že *GUI Designer* pro knihovnu komponent *Tecnomatix.UI* je dostupný jen pro 32bitovou verzi knihovny, tedy jen pro 32bitovou verzi Tecnomatixu. V rámci testování jsem vyzkoušel, jak můj plugin funguje na 64bitové verzi Tecnomatix, když byl zkompilován s 32bitovými knihovnami. Plugin fungoval normálně,

jen některé grafické prvky nebyly zcela správně napozicovány, ovšem rozdíl byl minimální. Co se týče vnitřní funkčnosti pluginu, funguje bez problémů, ale takovéto použití křížem 32bit na 64bit je samozřejmě funkční bez jakýchkoli záruk. Tuto komplikaci jsem obešel tak, že když jsem dokončil veškeré grafické úpravy rozhraní, připojil jsem ke kódu 64bitovou knihovnu Tecnomatixu a přeložil jsem plugin naslepo bez zobrazení GUI. Tím by měla být vnitřní funkčnost opět zaručena, daní za to je nutnost kompilovat dvě verze výsledného pluginu.

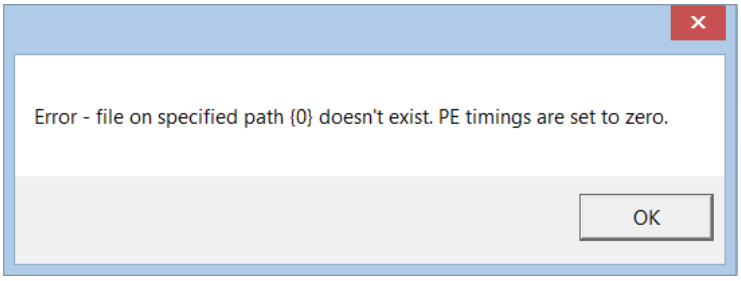
Uživatelské rozhraní pluginu do značné míry respektuje navrženou architekturu z obrázku 2. Na straně commandu má plugin dvě základní funkce - nastavení vlastností zařízení pro simulaci PE standardu a záznam simulace. Záznam simulace má pak dva pododdíly - záznam kloubů robotů a záznam PE stavů. Na obrázku 10 je vidět formulář pluginu. Záložka *Setting*



Obrázek 10: GUI - záložka nastavení či vytvoření logického bloku pro PE

& Creating obsahuje nástroje po přiřazení PE standardu vybraným robotům. Sekce *PE*

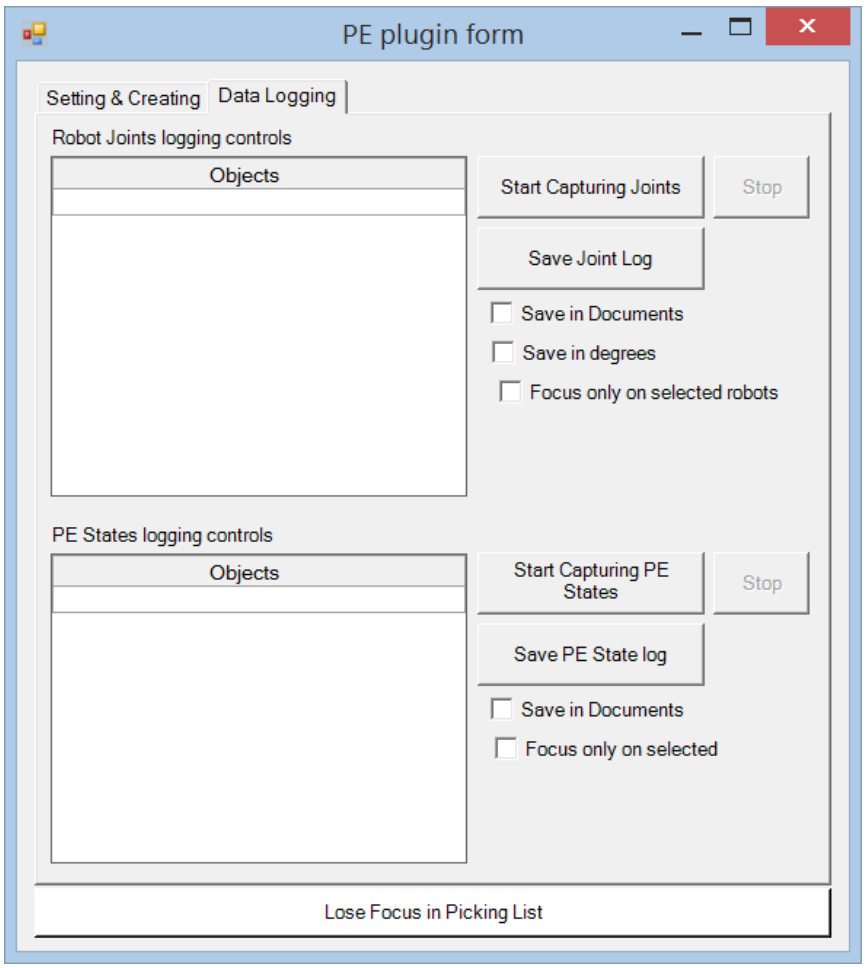
logic controls obsahuje pole pro hromadné vybírání objektů popsané titulkem *Objects*, v textu budu tento typ grafické komponenty označovat jako *Picking List*. Toto pole funguje tak, že uživatel do něj nejprve klikne kurzorem, tím se zamkne zaměření na toto pole, a pak začne klikat kdekoli v prostředí PS na roboty, kterým bude chtít přiřadit PE funkčnost. Všechny zakliknutné objekty se přidávají do tohoto pole. Při vybírání se aplikuje na označené objekty filtr, který umožňuje přidávat pouze roboty. Když jsou vybráni všichni roboti, uživatel pomocí tlačítka *Browse...* v dolní části okna vybere .csv soubor s PE časováním. Stiskem tlačítka *Assign PE module* pak spustí příkaz k vygenerování či přenastavení logického bloku, přičemž časové konstanty jsou nastaveny podle vybraného souboru s PE časováním. Pokud soubor s časováním nebyl vybrán nebo na dané cestě neexistuje, objeví se chybová hláška (obrázek 11) oznamující, že časové konstanty byly nastaveny na nulové hodnoty, PE funkčnost je ale přesto vytvořena. Ještě zde je zaškrtávací



Obrázek 11: GUI - chybová hláška - cesta k souboru neexistuje

políčko *Apply on selected only* - pokud je zaškrtnuté, aplikuje se přiřazení PE funkčnosti jen na roboty označené v *Picking Listu*, roboti, co se nachází v *Picking Listu*, ale nejsou tam označeni, jsou ignorováni. Poslední je tlačítko *Overwrite IDs*, stiskem tohoto tlačítka se provede přepsání konstant *blockID* ve všech logických blocích tak, aby byla zajištěna jejich unikátnost - tuto funkci jsem popisoval v kapitole 3.4.2. Není nutné mít označené žádné roboty, aby tato funkce v pořádku proběhla.

Druhá záložka obsahuje nástroje pro ovládání záznamu simulace. Jsou zde vidět dvě sekce - *Robot Joints logging controls*, v níž jsou nástroje k ovládání záznamu poloh kloubů robotů, a *PE States logging controls* pro záznam PE simulace. Obě sekce fungují velice po-



Obrázek 12: GUI - záložka k ovládání záznamu dat ze simulace

dobně. Obsahují *Picking List* pole pro výběr robotů, které se mají sledovat v záznamu, trojici tlačítek a několik zaškrtávacích políček. Tlačítka *Start Capturing...* zapne naslouchání datům, ale nespustí samo simulaci, tu musí uživatel běžným způsobem spustit sám. Když je toto tlačítko stisknuto, znepřístupní se a naopak vedlejší tlačítko *Stop* se povolí. Podle přístupnosti tlačítka uživatel pozná, zda je záznam aktivovaný, či nikoli. Stiskem *Stop* se záznam deaktivuje, ale simulace se nezastaví, pokud v době stisku běžela. Doporučuji simulaci nejprve zastavit a až potom stisknout tlačítko *Stop* - v tomto duchu se nese filozofie ovládání PS, je vhodné ji dodržovat. Nicméně není nezbytně nutné se tímto doporučením řídit, záznam je v pořádku i když se deaktivuje při běžící simulaci.

Další tlačítko *Save...* se chová podle zaškrtnutí políčka *Save in Documents* - pokud je políčko zaškrtnuto, pak stisk tlačítka uloží záznamy do automatické adresářové struktury v osobní složce přihlášeného uživatele Windows. Pokud políčko zaškrtnuté není, stisk tlačítka vyvolá dialogové okno pro výběr složky, v níž se uloží sada souborů se záznamy.

Zaškrtávací tlačítka *Focus only on selected* jsou brány v potaz jen v okamžiku aktivování záznamu dat. Rozhodují o tom, zda se záznam aktivuje jen pro vybranou podmnožinu robotů v příslušném *Picking Listu*, nebo jestli se aktivuje pro všechny roboty v příslušném *Picking Listu*. V sekci pro záznam natočení kloubů robotů je ještě zaškrtávací políčko *Save in degrees*, které v okamžiku ukládání dat o kloubech rozhoduje, zda se zaznamenané hodnoty převedou z implicitních radiánů na úhlové stupně, informace o použité jednotce je uložena v hlavičkovém řádku .csv souboru a navíc je zaznamenána na konci názvu souboru.

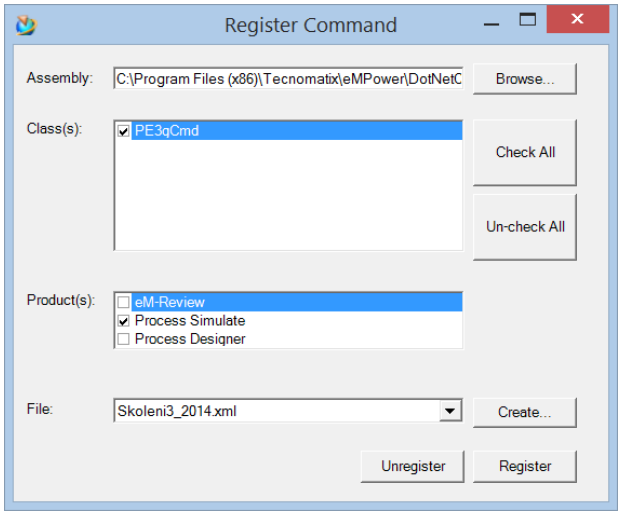
Obě záložky mají ještě společné tlačítko v zápatí okna - *Lose Focus in Picking List*. Toto tlačítko jsem přidal, abych tak vyřešil komplikaci s použitím *Picking Listů*, tím, že umožňují vybrat více objektů ve studii najednou a v libovolných zobrazovacích oknech - tzv. *Viewerech*, nerozpoznají, kdy uživatel už výběr objektů dokončil a chce nějaký jiný objekt jen upravit. Místo označení tak uživatel objekt znovu přidává a zbavit se zaměření *Picking Listu* je obtížné. Výhody použití těchto GUI prvků jsou však velké, takže jsem raději vytvořil tlačítko, které na stisk odebere zaměření všem třem *Picking Listům* použitým v mém pluginu. Usnadňuje se tím ovládání pluginu.

3.6 Postup při aplikaci pluginu

Plugin se skládá z několika souborů a aby mohl být v prostředí Process Simulate použit, je třeba jej nainstalovat a některé součásti zaregistrovat v Tecnomatixu. Když je pak uživatel obeznámen s GUI a funkcemi jednotlivých prvků pluginu, může dle svého uvážení plugin aplikovat se vší volností, která je mu v prostředí PS dopřána. Aby mohl ale ihned aplikovat plugin ve smyslu, pro jaký byl navržen, sestavil jsem několik vývojových diagramů, podle nichž uživatel rychle zavede plugin do své studie.

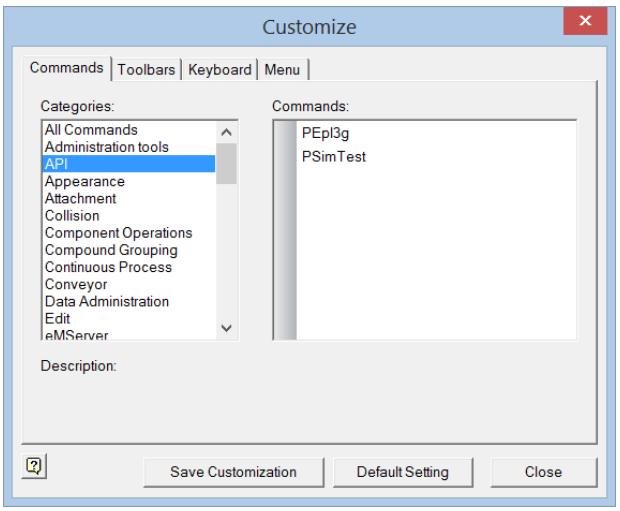
3.6.1 Instalace

Nejprve je třeba zkopírovat soubor *userFcn_stateMach.dll* do adresáře v instalační složce Tecnomatixu, typicky na adrese *../Program Files/Tecnomatix/eMPower/UserLBFunc*, kde jsou uloženy knihovny se všemi uživatelskými funkcemi v PS. Od příštího spuštění PS je uživatelská funkce přístupná. Dále je nutné zkopírovat soubor *PEpl3g.dll* do adresáře v instalační složce Tecnomatix na adrese *../Tecnomatix/eMPower/DotNetCommands*, v této složce jsou uloženy všechny commandy, které jsou do PS přidány, často už od implicitní instalace Tecnomatixu.



Obrázek 13: Registrování commandu pluginu

Pak je zapotřebí zaregistrovat command pro použití v PS. To se provede pomocí miniaplikace *Register Command* na adrese *../Tecnomatix/eMPower/CommandReg.exe*. Na obrázku 13 je vidět rozhraní této miniaplikace. V řádku *Assembly* uživatel vybere umístění souboru *PEpl3g.dll*, v poli označeném jako *Product(s):* vybere prostředí, v nichž chce plugin používat - plugin je vytvořen pro Process Simulate, zaškrtne tedy jen toto políčko. V posledním řádku *File:* buďto vybere .xml soubor již existující, nebo vytvoří nový. Tento .xml soubor je uložen ve složce *../Tecnomatix/eMPower/DotNetExternalApplications* a je možné jej distribuovat spolu se soubory pluginu namísto manuálního registrování pomocí

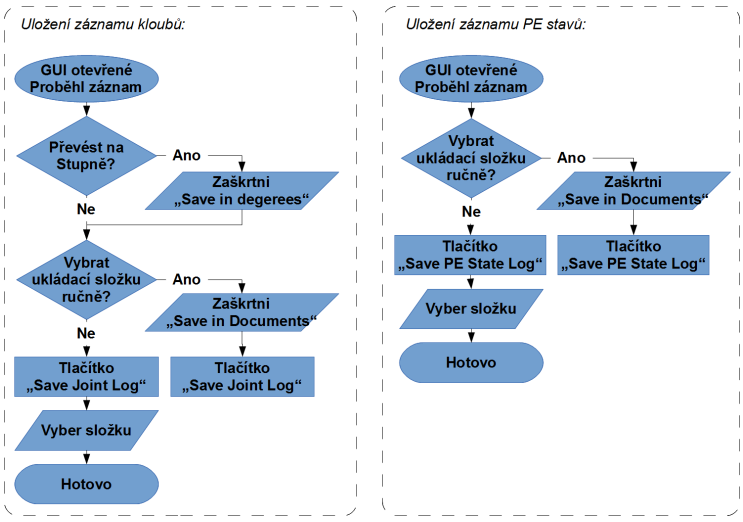


Obrázek 14: Zařazení tlačítka pluginu v PS

miniaplikace *Register Command*. Pokud chce uživatel této možnosti využít, musí zajistit, aby .xml soubor byl uložen v příslušném adresáři. Jakmile jednou zaregistrování proběhlo, je plugin nainstalován a uživatel jej může v prostředí PS přidat jako tlačítko na libovolnou nástrojovou lištu. Tlačítko nalezne v PS *Tools* → *Customize*, zobrazí se okno na obrázku 14. V záložce *Commands* v *Categories* vybere *API* a zde už je vidět jeho command v poli *Commands*. Přetažením na nástrojovou lištu si uživatel umístí tlačítko pluginu, kam potřebuje. Nastavení nástrojů si uživatel může tlačítkem *Save Customization* uložit pro příští relace. Stisknutím tlačítka se otevře GUI pluginu.

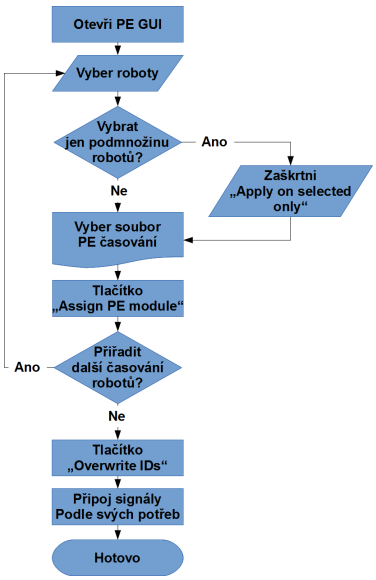
3.6.2 Použití

Pro použití pluginu jsem pro nové uživatele vytvořil vývojové diagramy, podle nichž se můžou řídit. Jejich sledováním uživatel zvládne provést všechny základní operace, které plugin poskytuje. Protože kroky pro aktivování záznamu otočení kloubů robotů a aktivování záznamu PE stavů jsou téměř stejné, vytvořil jsem pro ně společný vývojový diagram. Pro přiřazení či přenastavení PE funkčnosti jsem vytvořil diagram na obrázku 16. Pro aktivaci záznamu diagram na obrázku 17 a pro uložení záznamů diagram na

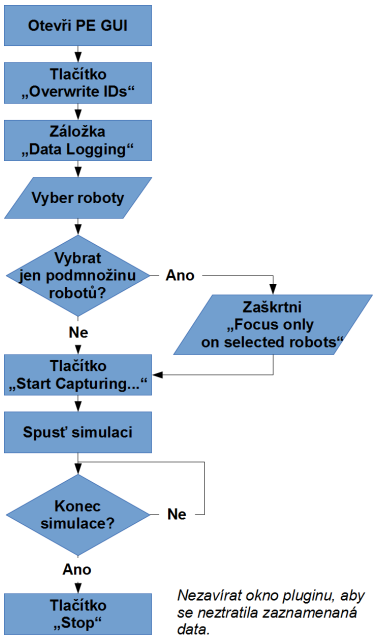


Obrázek 15: Workflow pro uložení záznamů

obrázku 15. Diagram popisující postup uložení záznamu předpokládá, že uživatel v průběhu záznamu nezavře okno pluginu. Tento předpoklad musí být splněn pro správnou funkčnost záznamu. Pokud uživatel v průběhu záznamu okno zavře, ztratí se aktuální instance pluginu a s ní i všechna doposud vygenerovaná data ze simulace - tedy ztratí se aktuální databáze záznamů. Znovuotevřením okna pluginu se vytvoří nová instance pluginu s prázdnými databázemi. Tuto funkčnost jsem v pluginu zachoval, protože ji lze i využít.



Obrázek 16: Workflow pro přiřazení či znovunastavení PE funkčnosti



Obrázek 17: Workflow pro aktivaci záznamu - stejný postup pro klouby i PE stavy robotu

```
using System;
using System.Collections;
using Tecnomatix.Engineering;
namespace userFcn_stateMach{
    [TxPlcLogicBehaviorFunctionAttribute("Nazev_funkce_v_prostredi_PS")]
    public class JmenoTridyUzivatelskeFunkce : ITxPlcLogicBehaviorFunction
    {
        ArrayList m_typesArray = new ArrayList();
        ArrayList m_namesArray = new ArrayList();
        TxPlcSignalDataType m_returnValueType;

        public JmenoTridyUzivatelskeFunkce()
        {
            m_typesArray.Add(TxPlcSignalDataType.Bool);
            m_namesArray.Add("VstupniParametr1");

            m_returnValueType = TxPlcSignalDataType.Int;
        }

        public TxPlcValue Evaluate(ArrayList parameters)
        {
            TxPlcValue navratHodnota = new TxPlcValue();
            return navratHodnota;
        }

        public ArrayList ParametersNames()
        {
            return m_namesArray;
        }

        public ArrayList ParametersTypes()
        {
            return m_typesArray;
        }

        public TxPlcSignalDataType ReturnValueType()
        {
            return m_returnValueType;
        }
    }
}
```

Výpis 2: Předloha prázdné uživatelské funkce

4 Virtuální zprovoznění

V průběhu vytváření části pluginu, která plní funkci stavového automatu, jsem co nejpřesněji sledoval definice dané PROFIenergy (PE) standardem. Naprogramoval jsem tak stavový automat ovladatelný přes rozhraní, do něhož vstupuje pouze ID požadovaného stavu a automat do tohoto stavu přejde tak rychle, jak podle definovaných časování přechodů může. Nicméně náš projekt je zaměřen na výrobní linku, kde jsou použity robotické kontrolery KUKA VKRC, a tyto kontrolery nepodporují standardní formu PE. Mají zpracovaný vlastní standard uspávání robotů do energeticky úsporných stavů. Můj plugin by měl umožňovat efektivně modelovat primárně tento VKRC standard. Pro potřeby virtuálního zprovoznění jsem musel tedy pozměnit chování automatu a implementovat konverzní rozhraní, které převádí zadané řídící signály na ID požadovaného stavu.

V dalším kroku jsme ve spolupráci s kolegou Vojtěchem Pavlíkem navrhli demonstrační program pro PLC, které v jednoduchém sledu příkazů provede robota dvěma demy výrobních robotických programů a převede robota do energeticky úsporného režimu a opět jej z něj probudí. Tento program jsme otestovali na reálném robotu a získali jsme data o spotřebách energie v průběhu demonstrace. Tato data jsem pak použil pro výpočet simulované energetické spotřeby.

Pro přípravu virtuálního zprovoznění jsem pak stáhl použité robotické programy z kontroleru robota a nahrál je do prostředí Process Simulate (PS) a vytvořil jsem tak operace, které lokacemi odpovídají lokacím fyzického robota v průběhu těchto programů. V prostředí PS jsem pak vygeneroval všechny signály, které byly použity v reálném případě k ovládání robotického kontroleru přes PLC. Vytvořil jsem logický strom operací reprezentující jednotlivé programy a pomocí svého modifikovaného pluginu jsem přiřadil robotu v simulaci VKRC PROFIenergy funkčnost. Nakonec jsem vše propojil pomocí dříve vytvořených signálů.

Signály jsem připojil k OPC serveru, který byl připojen k řídicímu PLC. Naneštěstí prostředí Process Simulate na stanici s připojením k PLC bylo z neznámých důvodů

poškozené - nejspíš instalací OPC serveru na tentýž počítač - takže jsem spínání signálů simuloval manuálním zadáváním signálů ve smyslu vytvořeného programu.

Ze simulace jsem získal jednak data o energetických stavech, v nichž se robot nacházel, a jednak videozáznam simulovaného pohybu robota. Z těchto dat jsem sestavil graf simulované spotřeby robota a demonstrační videosekvenci.

4.1 Konverze VKRC řídicích signálů na PE-ID

Standard VKRC má definované rozhraní v základě jako dva jednobitové vstupní signály.

- Drive_Bus_OFF
- Hibernate

Tyto dva signály jsou reálně přitomny v KRC kontrolerech. Krom nich jsem ještě pro potřeby simulace musel zavést třetí jednobitový signál:

- Operate

Tento signál zde musí být, protože oproti reálnému kontroleru, má simulace kontroler rozdělený v podstatě na dvě části - jedna simulující PROFIenergy činnost a druhá část je zcela externí a nezapouzdřená, jedná se o simulace robotických programů, které kvůli architektuře prostředí Process Simulate musí být reprezentovány jako operace, a jsou tedy nutně mimo logické bloky, ve kterých je simulováno PE. Když PLC vydává příkaz ke spuštění robotického programu, je tímto signálem *Operate* předána informace PE stavovému automatu informace o přechodu do módu *Operating*.

Pro takto popsané rozhraní jsem implementoval uživatelskou funkci pro PS, jejímiž vstupy jsou tři výše uvedené signály typu boolean, a jejím výstupem je ID výsledného stavu. Funkce má vstupy mapovány podle tabulky 4. Tuto funkci jsem zasadil do logického bloku a její výstup použil jako vstup pro funkci simulující stavový automat, takže stavový automat přijímá až zprostředkovaně přeložené ID stavu.

Vstupy			**Výstup**	
Drive_Bus_OFF	Hibernate	Operate	Jméno stavu	ID stavu
0	0	0	Ready-to-operate	0
0	0	1	Operating	1
1	0	x	Power-saving	4
0	1	x	Sleeping	8
1	1	x	Sleeping	8

Tabulka 4: Konverzní funkce VKRC vstupů na ID PE stavu

4.2 Vnitřní odlišnosti VKRC PE stavového stroje

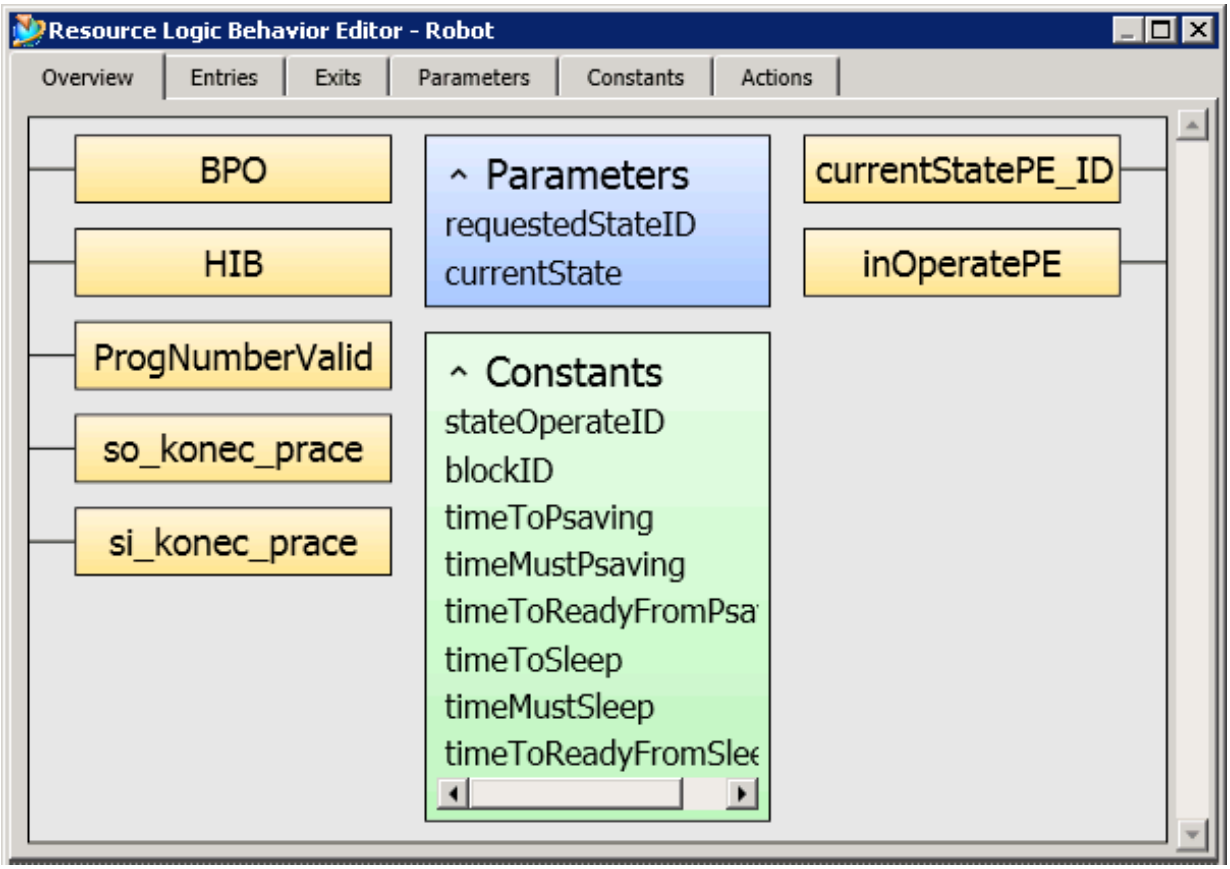
Kontrolery VKRC se od korektního PE standardu liší i ve vnitřní struktuře stavového automatu, konkrétně v přechodových podmínkách. Korektní PE standard jsem popsal v předchozích kapitolách, jeho redukovanou nicméně korektní verzi jsem implementoval jako první generaci části pluginu se stavovým automatem. Oproti zmiňovanému modelu nemá VKRC standard zautomatizovány některé přechody mezi stavy a veškerá zodpovědnost za správné vedení kontroleru vhodnou cestou k požadovanému stavu je zodpovědné čistě programování PLC. Například očekávaná funkčnost kontroleru robotu je, že když se nachází v *Power-saving* módu a přijde požadavek na přechod do módu *Operating*, kontroler se sám postará, aby nejprve přešel do módu *Ready-to-operate* a až potom do módu *Operating*. Standard VKRC nicméně v případě požadavku na přechod z módu *Power-saving* přímo do *Operating* jednoduše neprovede žádnou akci a zůstane v původním stavu. Je na zodpovědnosti PLC, aby byl zaslán vždy v daný čas požadavek na přechod do bezprostředně navazujícího módu.

Druhým výrazným omezením je zde, že pro přechod ze stavu *Operating* je nutné poslat informaci o dokončeném robotickém programu do PLC a počkat na příchod potvrzení z PLC. Tato komunikace je provedena pomocí signálů *so_konec_práce* a *si_konec_práce*, první je poslán z kontroleru do PLC, druhý je potvrzení z PLC do kontroleru. Tyto signály

jsem přidal jako vstupní parametry uživatelské funkce pro PE stavový automat.

Nakonec bylo nutné přidat výstupní proměnnou logickému bloku, která bude značit, kdy je PE stavový automat v módu *Operating*, což je signál, který jsem použil jako jednu podmínku ke spuštění robotické operace, čímž je simulované spuštění robotického programu ve fyzickém kontroleru.

Za zmínění stojí také to, že VKRC standard má umožněný přechod ze stavu *Power-saving* přímo do stavu *Hibernate*. Tento přechod je v PE standardu uveden jako volitelný. Na obrázku 18 je přehled vnitřní struktury výsledného logického bloku, který jsem přiřadil robotu v simulaci.



Obrázek 18: Přehled logického bloku VKRC PE standardu

4.3 Demonstrační PLC program

Pro demonstraci funkčnosti PE standardu a jeho začlenění do výrobního procesu jsme vytvořili snadno pochopitelnou jednoduchou sekvenci operací. Nejprve obsluha spustí robota a provede případně požadované kalibrace a testy (brzd, motorů apod.). Potom převede robotický kontroler do externího režimu, to znamená, že od té chvíle bude kontroler přijímat příkazy z připojeného PLC. Tyto první kroky jsou na reálné lince provedeny standardně. Od té chvíle robot cyklicky kontroluje, zda po něm není vyžadováno spuštění programu, a pokud ano, pak podle předaného čísla programu vybere ve své paměti příslušný program a vykoná jej. Po dokončení programu odešle kontroler signál do PLC a čeká na potvrzovací signál, jak je zmíněno výše.

Sekvenci jsme určili tak, že nejprve robot vykoná první robotický program (ID programu je 2), pak přejde do *Power-saving* módu, v něm setrvá po dobu několika sekund, a pak se vyšle povel k probuzení do módu *Ready-to-operate*, a pak se spustí druhý robotický program (ID druhého programu je 9). Tím sekvence končí a lze ji opakovat na základě povelu uživatele.

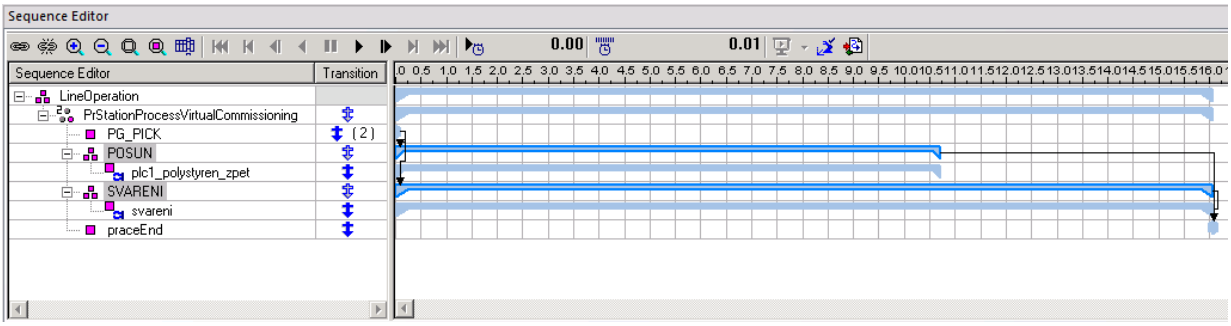
4.4 Modelování případu v Process Simulate

Jak jsem nastínil výše, v prostředí Process Simulate není možné modelovat robotický kontroler jako jeden zapouzdřený celek, kterým v reálu je. Není možné simulovat robotické programy jinak než pomocí robotických operací a zároveň nelze dostatečně elegantně simulovat energetické stavy PE standardu pomocí operací. Kontroler je tedy rozdělen na dvě základní části.

- PE stavový automat
- Robotické operace

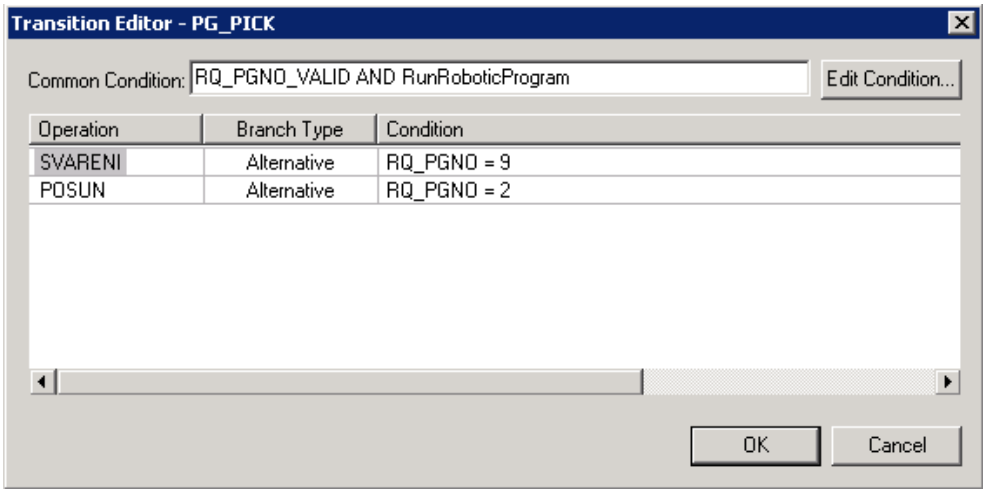
Přitom je ještě nutné zajistit, že se v simulaci spustí vybraný robotický program ze sady všech dostupných. To jsem realizoval pomocí větvení operací a pomocné operce s roz-

hodovacím kritériem, jak je vidět na 19. Operace *PG_PICK* se větví do dvou možností



Obrázek 19: Struktura operací v sekvenční editoru

podle signálu *RQ_PGNO*. Tento signál z PLC spouští v kontroleru robotický program se zadaným číslem. Nastavení operace a kritéria je vidět na obrázku 20. Na obrázku



Obrázek 20: Nastavení přechodových podmínek operace

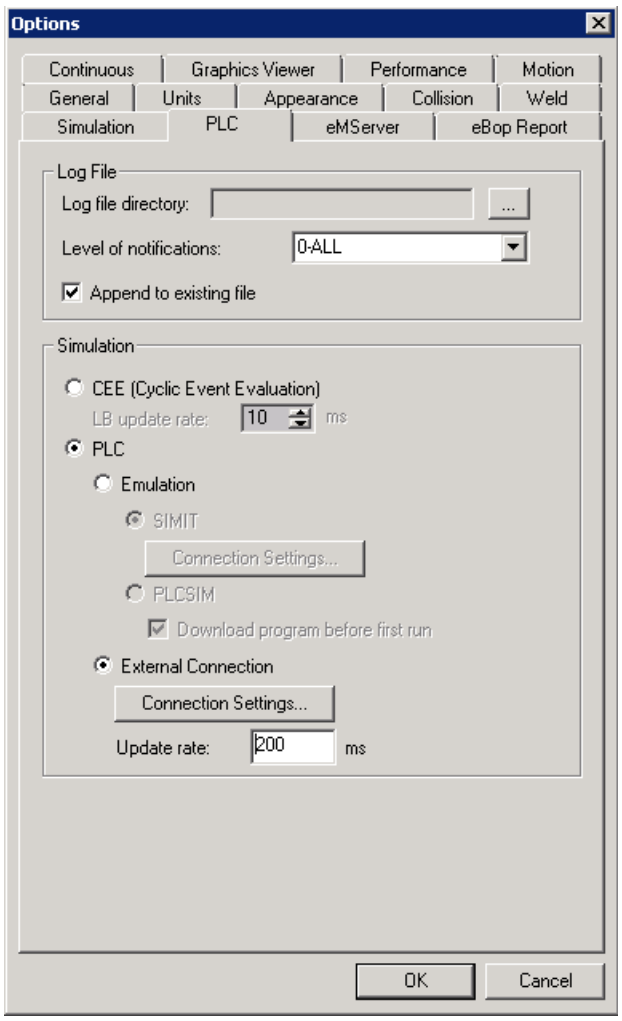
19 je také vidět, že všechny robotické operace ústí do non-sim operace *praceEnd*. To je pomocná operace, která při svém konci vygeneruje signál napojený na vstup logického bloku *si_konec_prace*. Případné rozšíření množiny robotických operací potažmo programů použitých ve virtuálním zprovoznění by se vyřešilo následovně. Rozšiřující robotické operace se přidají do studie a napojí se na pomocnou non-sim operaci *PG_PICK* a do větvicího

kritéria této pomocné operace se přidá jako alternativní následovník s příslušným *RQ_PGNO*. Pak se všechny alternativní operace připojí k ukončovací non-sim pomocné operaci *praceEnd*, tím je rozšíření hotové.

4.5 Připojení signálů

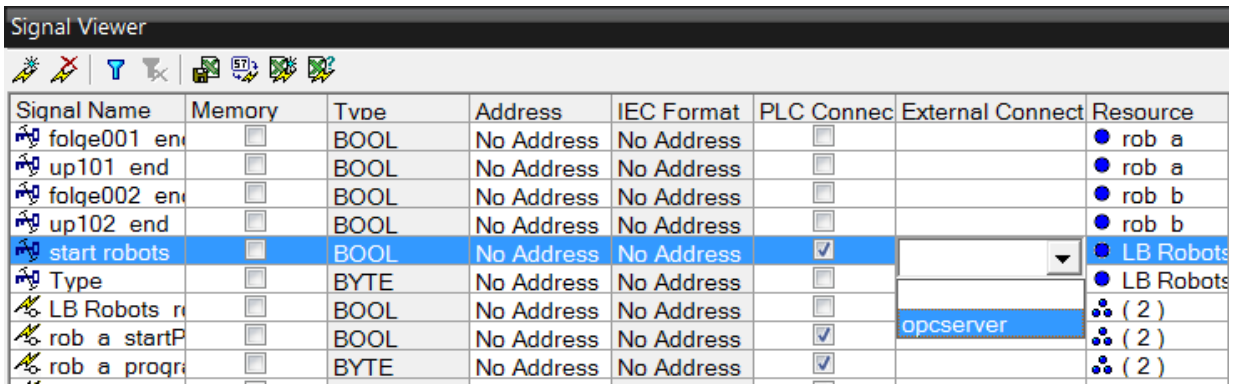
Řídicí signály použité ve studii v Process Simulate je možné připojit k OPC serveru nebo aplikaci SIMIT a řízení jejich hodnot přenechat PLC případně simulačnímu softwaru. My jsme zvolili možnost ovládat signály z PLC a tedy připojit signály k OPC serveru, který by byl připojen k PLC. K tomu jsme nainstalovali počítač nejblíže PLC OPC server SIATIC NET a namapovali všechny potřebné signály z PLC na OPC a pak jsem v Process Simulate simulované signály připojil k OPC. K tomu je třeba provést nastavení v *Options* v PS v záložce *PLC* v oddílu *Simulation*. Přepnout na *PLC*, a pak *External Connection*. Okno s nastavením je vidět na obrázku 21. Kliknutím na tlačítko *Connection Settings...* se otevře dialog, v němž přidáme *OPC Connection* a zadáme pro ně příslušná nastavení. Poté je ještě zapotřebí připojit příslušné signály v okně *Signal Viewer* k OPC. Jak je vidět na obrázku 22, zvolí se signál a ve sloupci *PLC Connection* se zaškrtne zaškrtávací políčko, a dále se ve sloupci *External Connection* vybere příslušný OPC server, v němž mají být signály připojeny.

Při spuštění simulace v PS všechny signály správně reagovaly na změny v PLC potažmo v OPC sereveru, ale vznikl zde jiný problém. V sequence editoru se nespustila žádná operace a nebylo možné spustit robotické operace. Prostředí Process Simulate také od té chvíle nebylo schopné uložit na eMServer změny ve studii. Domnívám se, že nejspíš při instalaci OPC serveru na stejný počítač některá komponenta PS musela být ovlivněna a funkčnost PS tím byla narušena. Možnost, že by chyba byla v nastavení studie či projektu, jsem vyloučil, protože stejná studie se stejným nastavením čerstvě načtená z eMServeru na jiném počítači fungovala, jak bylo plánováno, zatímco i při odpojených signálech na počítači s OPC serverem PS nefungoval, jak má.



Obrázek 21: Nastavení připojení k OPC serveru

Protože v závěru nezbýval čas na hledání řešení tohoto problému, upustili jsme od zprovoznění s připojením na OPC server a příkazy přicházející z PLC jsem simuloval manuálním zadáváním. Jedná se o jednoduchý PLC program, takže zastoupit funkci PLC nebyl problém.

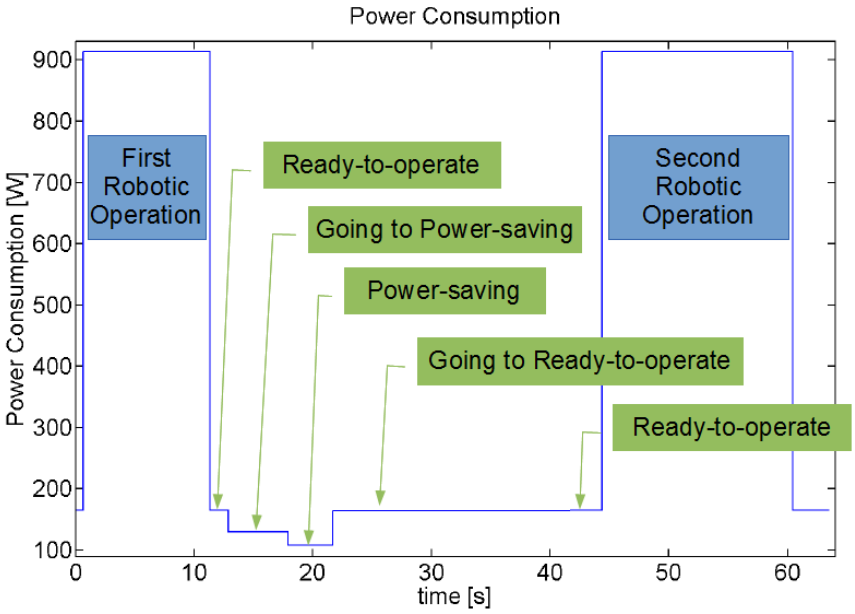


Obrázek 22: Připojení signálův Process Simulate k OPC

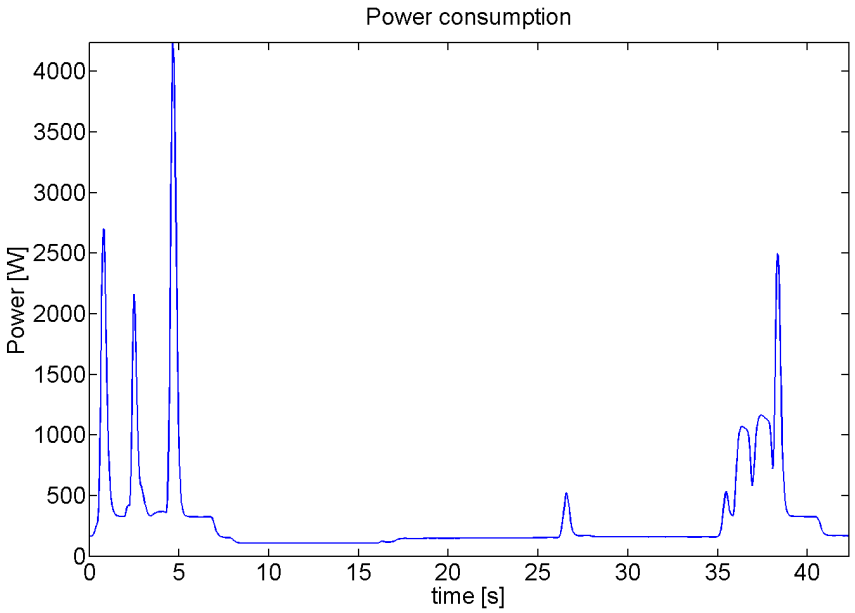
4.6 Simulovaná spotřeba

Při fyzickém testování demonstrační sekvence byla měřena hodnota okamžitého příkonu kontoleru robota a z těchto dat jsme vypočetli průměrné spotřeby v jednotlivých režimech. Výstup ze záznamu o PE stavech ze simulace se skládá z párů časové známky a PE stavu v příslušnou dobu. Tento řetěz dat je generován pro každého sledovaného robota v simulaci zvlášť a je uložen v csv souboru. Přiřazení simulovaných hodnot spotřeby jsem provedl externě v programu MATLAB. Hodnoty spotřeby pro jednotlivé PE módy jsou uvedeny v dokumentaci robota, stejně tak i časování těchto módů (jak dlouho trvá přechod mezi jednotlivými módy, jaká je minimální doba setrvání v daných módech apod.). Ještě ovšem zbývá odhadnout spotřebu robota při vykonávání jednotlivých robotických operací. Tato hodnota je silně závislá na spotřebě motorů robota a je tedy závislá na jeho pohybech. Energetická funkce, která by na základě údajů o pohybu kloubů robota určila předpokládanou spotřebu dané operace, je v projektu ve vývoji a není ještě použitelná. Nicméně pro moje potřeby pro demonstraci virtuálního zprovoznění lze použít průměrnou spotřebu v průběhu operace. Celková vynaložená energie na provedení operace bude ekvivalentní. Tuto průměrnou spotřebu operace jsem vypočetl z naměřených dat.

Podobně také přechod mezi jednotlivými stavy není konstantní funkce z hlediska spotřeby energie, ale opět souhrn spotřebované energie lze rovnoměrně rozprostřít na časový interval



Obrázek 23: Graf simulované spotřeby



Obrázek 24: Graf změřené spotřeby

trvání přechodu - tedy použít průměrnou spotřebu na tomto intervalu.

Po provedení měření jsme ale zjistili, že spotřeba uvedená v dokumentaci se mírně liší od skutečné spotřeby v jednotlivých PE stavech. To si vysvětluji tím, že údaje v dokumentaci jsou nejspíše získány měřením jen malého množství různých kontrolerů a také za různých podmínek, jako je teplota ve skříni kontroleru, uplynulá doba od zapnutí v okamžiku měření apod. Další nesrovnalosti tvoří časování PE přechodů. Podle naměřených průběhů energetické spotřeby je přechod proveden za kratší dobu, než jaká je uváděna v dokumentaci ke kontroleru. Tyto odlišnosti lze ovšem zahrnout do modelu zprůměrováním spotřeby přes celý časový interval uvedený v dokumentaci. Nevadí přitom, že se do přechodu započte i chvíle, kdy už kontroler přetrvává v cílovém stavu. Časování z dokumentace chápu jako korektní a bezpečné, a proto se ho držím v simulaci. Výsledkem jsem získal simulaci, která je vidět na obrázku 23. Pro orientační srovnání uvádím na obrázku 24 i graf z naměřených hodnot.

Časování se liší i proto, že graf z naměřených hodnot je sestaven z měření sekvence řízené pomocí PLC, zatímco simulace byla časována manuálně. Dále je vidět, že robotické operace se vyznačují špičkami ve spotřebě, což jsem už vysvětlil výše.

5 Závěr

Cílem práce bylo rozšířit prostředí Process Simulate o funkčnost profilu PROFIenergy, aby bylo možné tento profil modelovat a simulovat. Následujícím logickým krokem bylo otestování této funkčnosti na zařízení, které je v rámci projektu k testovacím účelům určené.

Podařilo se mi postupně vyvinout plugin, který tyto požadavky splňuje, a otestovat ho na modelové studii. Provedl jsem v rámci virtuálního zprovoznění také simulaci uvádění robotu do úsporného režimu a výsledky měření ze skutečného robota jsem srovnal se svou simulací se slibnými výsledky.

Moje práce byla jen jedním stavebním kamenem celého projektu a závisí na výsledcích práce dalších lidí. V současnosti je PE plugin také jen první funkční verzí s omezenými možnostmi, i když ani výrobci zařízení ještě nedosáhli úplné implementace všech součástí profilu PROFIenergy. PE plugin momentálně pokrývá schopnosti robotu, na který bude aplikován a jeho vývoj nejspíš ještě není u konce. I na to jsem při jeho programování myslel a snažil jsem se udržovat všechny vrstvy programů co nejjednodušší a nejpřehlednější, aby se výsledek mého snažení neuzavřel sám do sebe a byl snadno modifikovatelný.

PE plugin měl být původně aplikován na model výrobní linky v továrně ŠKODA AUTO, ale tento plán se nepodařilo splnit kvůli komplikacím s datovou kompatibilitou jimi dodaných PS studií výrobní linky. Převod studií je práce rozsahem srovnatelná s prací na implementaci PE pluginu a bude vyžadovat pro naplnění cílů projektu ještě pozornost.

Věřím však, že splnění hlavních cílů diplomové práce se mi podařilo dosáhnout.

Reference

[1] Siemens Industry Software. *Tecnomatix 11.1 Administration Guide*, 2013.

[2] PROFIBUS & PROFINET International. *Common Application Profile PROFIenergy, v1.1*, August 2012. Technical Specification for PROFINET.

[3] Siemens Industry Software. *Process Simulate version 11.1 Reference Manual*, 2013.

[4] Siemens Industry Software. *Process Designer 11.1 Reference Manual*, 2013.

[5] Siemens Industry Software. *Tecnomatix.NET version 11.1*, 2013.

Appendix

Obsah přiloženého CD

V tabulce 5 jsou uvedena jména všech kořenových adresářů přiloženého CD s popisem obsahu.

Jméno adresáře	**Popis**
diplomova_prace	diplomová práce ve formátu .pdf
installable	přeložené knihovny programu k instalaci
multimedia	videoprezentace

Tabulka 5: Obsah CD