VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INTELIGENTNÍCH SYSTÉMŮ
DEPARTMENT OF INTELLIGENT SYSTEMS

TRANSFORMACE MODELŮ ŘÍDICÍCH SYSTÉMŮ MEZI POWERDEVS, NODE-RED A 4DIAC

TRANSFORMATION OF CONTROL SYSTEM MODELS AMONG POWERDEVS, NODE-RED, AND 4DIAC

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE **Bc. TOMÁŠ SADÍLEK**
AUTHOR

VEDOUCÍ PRÁCE **doc. Ing. VLADIMÍR JANOUŠEK, Ph.D.**
SUPERVISOR

BRNO 2022

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav inteligentních systémů (UITS) Akademický rok 2021/2022

Zadání diplomové práce

Student:	**Sadílek Tomáš, Bc.**
Program:	Informační technologie a umělá inteligence
Specializace:	Kybernetická bezpečnost
Název:	**Transformace modelů řídicích systémů mezi PowerDEVS, Node-RED a 4diac** **Transformation of Control System Models among PowerDEVS, Node-RED, and 4diac**
Kategorie:	Umělá inteligence

Zadání:

1. Prostudujte problematiku modelem řízeného vývoje distribuovaných řídicích aplikací. Zaměřte se na nástroje Eclipse 4diac a PowerDEVS. Prostudujte odpovídající teorii modelování a simulace, a seznamte se s normou IEC 61499 pro modelování a programování řídicích systémů. Seznamte se také s nástrojem Node-RED pro koordinaci IoT.
2. Definujte způsob transformace modelů z PowerDEVS do modelů používaných prostředími Node-RED a 4diac.
3. Navrhněte prostředky pro transformaci modelů z PowerDEVS do vývojových a běhových prostředí Node-RED a 4diac. Implementujte potřebné atomické bloky, pokud v některém prostředí chybí.
4. Navržené prostředky realizujte a ověřte jejich použitelnost na distribuované řídicí aplikaci z oblasti Smart Home. Lze pracovat se simulovanými senzory a aktuátory v simulovaném prostředí. Vyhodnoťte dosažené výsledky. Vytvořte plakát shrnující podstatu této práce.

Literatura:

- Dle pokynů vedoucího.

Při obhajobě semestrální části projektu je požadováno:

- Bez požadavků.

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/

Vedoucí práce:	**Janoušek Vladimír, doc. Ing., Ph.D.**
Vedoucí ústavu:	Hanáček Petr, doc. Dr. Ing.
Datum zadání:	1. listopadu 2021
Datum odevzdání:	29. července 2022
Datum schválení:	29. června 2022

Abstrakt

Cílem této diplomové práce je návrh a realizace transformační aplikace z PowerDEVS do jazyků Node-RED a 4diac. Tento přístup vychází z principu model continuity. Překladač je napsán jako terminálová aplikace v jazyce Python 3. Tato aplikace je otestována na modelu řídícího systému v oblasti Smart Home.

Abstract

The aim of this diploma thesis is design and implementation of a transformation application from flow-based development tool PowerDEVS to Node-RED and 4diac languages. This approach is based on the model continuity principle. The compiler is written as a terminal application using Python 3 programming language. This application is tested on a model of the control system in the Smart Home area.

Klíčová slova

Transformace modelů řídících systémů, Vývoj řízený modely, Internet věcí, MQTT, PowerDEVS, Node-RED, Eclipse 4DIAC, HomeAssistant, Model Continuity

Keywords

Transformation of models of control systems, Model–driven engineering, Internet of Things, MQTT, PowerDEVS, Node-RED, Eclipse 4DIAC, HomeAssistant, Model Continuity

Citace

SADÍLEK, Tomáš. *Transformace modelů řídicích systémů mezi PowerDEVS, Node-RED a 4diac*. Brno, 2022. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce doc. Ing. Vladimír Janoušek, Ph.D.

Transformace modelů řídicích systémů mezi PowerDEVS, Node-RED a 4diac

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana doc. Ing. Vladimír Janoušek Ph.D. Uvedl jsem všechny literární prameny, publikace a další zdroje, ze kterých jsem čerpal.

.....

Tomáš Sadílek

29. července 2022

Obsah

Seznam obrázků

Kapitola 1

Úvod

V dnešní době se chytrá zařízení v domácnosti objevují stále častěji. Takovým příkladem může být chytrá žárovka nebo zásuvka. Jestliže nastane porucha chytré žárovky způsobí maximálně větší účet za elektřinu. Poslední dobou jsou chytré domácnosti přiřazovány náročnější úlohy. Chytrý zámek, automatické ovládaní oken nebo systém vytápění jsou dobrý sluha, ale můžou se stát zlým pánem. Chyba v softwaru už může být příčinou nemalé újmy na domácích aktivech. Proto bychom chtěli daný systém nejdříve simulovat a vyvarovat se chyb plynoucí z implementace softwaru.

Chytré domácnosti nejčastěji používají pro vytváření domácí automatizace program Node-RED s komunikací pomocí MQTT. 4diac s komunikací MQTT nejčastěji nalezneme v průmyslovém odvětví. Tyto technologie jsou popsané v kapitole 2 (Použité technologie). Při kombinaci těchto dvou technologií bychom chtěli nejdříve otestovat, zda jsou mezi sebou kompatibilní. K tomu využijeme simulátor v PowerDEVS, který nám umožní odsimulovat stávající systém s novým modulem ještě předtím, než ho vložíme do reálného systému.

Distribuovaný systém, jehož specifikace je ve standartu IEC 61499, má několik výhod oproti systému s centrální jednotkou. Hlavní výhodou distribuovaného systému je tolerance k chybám, kdy výpadku části systému nemá vliv na zbytek systému. Distribuovaný systém nám umožňuje větší škálovatelnost – přidáním nového modulu nemusíme zvyšovat výkon stávajících modulů. Oproti centralizovanému systému, kde musíme počítat s určitým výkonem centrální jednotky tak, aby stíhala odbavovat všechny senzory a aktuátory. Škálovatelnost nám také umožňuje doplňování chytré domácnosti postupně a rozložit tím nemalou investici v čase. Nevýhodou distribuovaného systému je, že čím je systém složitější, tím více obsahuje částí, které se mohou pokazit. Proto bychom chtěli vytvořený systém nejdříve odsimulovat, až poté ho nasadit na cílové zařízení.

Princip vývoje softwaru od simulačního modelu po finální aplikaci se nazývá model continuity a je stále častěji používanou metodou. Téma vývoje softwaru přes model aplikace je nastíněn v kapitole 3 (Modelovací technika). Simulace aplikace umožní nalezení architektonických chyb už v rané fázi vývoje aplikace. Nevýhodou této metody je, že většina simulačních jazyků není stavěná do reálného nasazení. Proto je cílem práce vytvořit překladač, který převede simulovaný model z PowerDEVS na model v Node-RED nebo 4diac. Tento přístup nám umožní celou aplikaci odsimulovat v simulovaném prostředí a následně pomocí překladače automaticky převést na cílovou platformu a porovnat, zda se oba systémy shodují.

Způsobů transformace modelů se zabývá kapitola 4 (Návrh řešení). Každá metoda má své kladné a záporné vlastnosti, které jsou popsány v první části kapitoly. Ve druhé části se práce zabývá transformací jednotlivých atomických bloků v závislosti na daném omezení

cílového jazyka. Tyto transformace mezi jazyky jsou demonstrovány na bloku přepínače. Ve třetí části jsem vytvořil návrh pro testovací aplikaci z oblasti Smart Home.

Implementace samotného překladače a pomocných nových atomických bloků je popsaná v kapitole 5 (Implementace).

Celý systém byl testován na modelu bytové jednotky z oblasti Smart Home. Toto testování odpovídá modelu continuity. Jednotlivé fáze testování a výsledky jsou ukázány v kapitole 6 (Testování).

Kapitola 2

Použité technologie

V této kapitole jsou popsány jednotlivé technologie pro modelování funkčních bloků, použitých v této práci. Jsou zde popsány také použité komunikační protokoly a vizualizační komponenty.

2.1 Internet věcí

Internet věcí, z anglického názvu Internet of Things (IoT), je v informatice označení pro globální síť fyzických zařízení, domácích spotřebičů, průmyslových strojů, vozidel a dalších zařízení. Zařízení obsahují vestavěné systémy: procesor, software, senzory a síťovou konektivitu, která umožňuje výměnu dat. Pro komunikaci se používá model „machine to machine (M2M)“, kdy komunikace probíhá bez aktivní účasti člověka. Komunikace nejčastěji probíhá pomocí bezdrátové komunikace *Wi-fi* nebo *Bluethooth*. Každé takové zařízení lze identifikovat podle unikátního názvu v daném implementovaném systému. [16]

Společnosti IDC, která se zabývá technickými analýzami, předpovídá, že do roku 2025 bude celkem 41,6 miliardy připojených zařízení do sítě IoT. Hlavním tahounem zvyšování počtu zařízení bude především průmyslový sektor, kde pokračuje zavádění chytrých měřičů. Dalším sektorem, zavádějícím IoT, je například zdravotnictví (zařízení pro monitorování chronických onemocnění), automobilový průmysl (chytrá auta) nebo automatizace budov (řízení světel, vytápění nebo střežení budov).

Průmyslový internet věcí (IIoT) neboli čtvrtá průmyslová revoluce ve zkratce Průmysl 4.0 jsou názvy pro použití technologie IoT v obchodním prostředí. Koncept vychází ze spotřebitelských zařízení v domácnosti, ale hlavním cílem je využít kombinace senzoru, bezdrátových sítí, velkých dat, umělé inteligence (AI) a analytiky k měření a optimalizaci průmyslových procesů. Mezi hlavní přínosy v průmyslu je monitoring výroby, vzdálená správa zařízení, identifikace položek, průběžné zlepšování díky analýze dat nebo úspora nákladů například za pomoci prediktivní údržby, kdy je stroj opraven ještě předtím, než na něm vznikne porucha.

Poslední dobou je také skloňována bezpečnost a spolehlivost, jelikož tyto senzory shromažďují v mnoha případech velmi citlivá data. Často se objevují také chyby v kódu, což může znamenat únik nejen dat ale také napadení cílového zařízení. Většina zařízení nemá možnost opravy kódu, a tak zůstávají s potencionálními zranitelnostmi až do konce životnosti zařízení. Proto je důležitý správný vývoj a testování nově vznikajících zařízení. [12]

2.2 Node-RED

Node-RED je grafický „flow-based“ vývojářský editor pro blokové programování. Vývoj začal v IBM a následně v roce 2016 přešel na open-source k organizaci OpenJS Foundation. Node-RED je napsaný v javascriptu a běží na platformě Node.js. Node.js je softwarový systém pro psaní vysoce škálovatelných internetových aplikací. „Flow-base“ programování staví na principu blackboxů propojených mezi sebou a navzájem komunikujících pomocí zpráv.

Vytvořené schéma můžeme exportovat do formátu JSON ve webovém prostředí. Ve formátu JSON má každý uzel unikátní identifikátor (id), type určující typ daného uzlu, pozice x a y určující umístění uzlu ve schématu, `z` je identifikátor rodičovského flow a další parametry určující vlastnosti daného uzlu. Načíst dané schéma ve formátu JSON můžeme pomocí webového prostředí (Import) na serveru. Pro nasazení na distribuovaná zařízení můžeme schéma předat jako parametr programu nebo ho nahrát v těle HTTP zprávy.

Node-RED můžeme nainstalovat jako samostatnou aplikaci nebo jako součást automatizovaného softwaru např. HomeAssistant. Pro instalaci samostatné aplikace stačí stáhnout zdrojové kódy z Githubu a ty následně pomocí balíčku npm nainstalovat. Tento systém je dostupný pro Linux i Windows. [6]

Základní Node-RED obsahuje pouze malé množství základních funkčních bloků. Pokud chceme další, můžeme je doinstalovat pomocí zabudované knihovny, nebo je získat z balíčků pomocí npm.

2.2.1 Koncept návrhu

Základním prvkem systému je uzel (Node). Každý uzel může mít pouze jeden vstupní port, ale neomezené množství výstupních portů. Uzel se spouští, buď příjmem zprávy od předchozího uzlu, čekáním na externí událost např. MQTT zpráva, HTTP požadavek, časovačem daného modulu nebo změnou HW např. GPIO, UART.

Jednotlivé uzly můžeme propojovat pomocí propojů (wire). Určují nám vstupní nebo výstupní port uzlu, kudy půjde daná zpráva. Do atributu wire se ve zdrojovém uzlu vkládá identifikátor následujícího uzlu. Pouze u výstupního portu se do výstupního portu ukládá identifikátor zdrojového uzlu.

Konfigurační uzel (Config Node) je speciální typ uzlu, který se nezobrazuje v přehledovém schématu a lze ho najít v pravé liště v záložce Global Config Node. Tento uzel umožňuje opakovatelné použití konfigurace různých protokolů např. MQTT, HTTP, sériový port. Konfigurace mezi sebou mohou sdílet běžné uzly ve schématu. Identifikátor uzlu se vkládá jako atribut uzlu, ve kterém je používán.

Kontext slouží k ukládaní informací. Ty lze použít mezi uzly, aniž by uzel vygeneroval nějaké zprávy. Máme tři druhy kontextu: *uzlový* – dané hodnoty vidí jen uzel, *flow* – hodnoty vidí všechny propojené uzly, a *globální* – danou hodnotu vidí všechny uzly.

Zpráva (message) umožňuje předávání informací mezi uzly ve flow. Jsou to objekty JavaScriptu, které mohou mít libovolnou hodnotu, nejčastěji se používá klíč/hodnota. Zpráva je v uzlu reprezentována objektem msg. Podle konvence by měl klíč, nejužitečnější informace ve zprávě, mít název payload.

Podschéma (Subflow) slouží k abstrakci určité části schématu a to podporuje jeho znovupoužitelnost. Má stejná pravidla jako obyčejný uzel a uvnitř tohoto uzlu jsou uzly či další Subflow. U každé Subflow lze nastavit počet vstupních a výstupních portů, které se zobrazí ve schématu jako input a output X, kde X je pořadí výstupního portu. Nové podschéma

se vytvoří jako samostatný prvek, který se následně do schématu vkládá jako nový blok s typem nazvaným subflow:id, kde id je identifikátor daného schématu. Bloky patřící do daného schématu jsou odlišené parametrem `z`, ve kterém je identifikátor daného schématu.

Všechny uzly mají jedno omezení. Počet vstupních portů může být pouze jeden nebo žádný. V případě, že aplikace potřebuje více vstupních portů, je nutné je odlišit v příchozí zprávě názvem proměnné.

2.3 Eclipse 4diac

Eclipse 4diac se skládá ze dvou částí sloužících pro vývoj a nasazení distribuovaných řídících systémů v souladu s normou IEC 61499. [5]

2.3.1 4diac IDE

4diac IDE je integrované vývojové prostředí založené na frameworku Eclipse běžícím v Javě. Umožňuje vytvářet funkční bloky, aplikace, konfigurace zařízení, testování v souladu s normou IEC 61499. Jednotlivé funkční bloky lze nasadit na zařízení 4diac FORTE a kontrolovat jednotlivé vstupní a výstupní hodnoty.

Základní jednotkou je funkční blok. Funkční blok obsahuje vstupy a výstupy pro data a události. Jednotlivé vstupní události horní části funkčního bloku spouští jednotlivé funkcionality funkčního bloku. Po ukončení funkce a vystavení dat na výstupní datový port je vyvolána patřičná výstupní událost. Tyto jednotlivé funkční bloky se mohou propojovat pomocí propojovacích vodičů. Tato datová spojení lze rozvětvovat. To znamená, že z jednoho výstupního portu můžeme napojit neomezené množství vstupních portů. Naopak to není povoleno. Napojení více datových připojení do jednoho vstupního portu je nutné řešit pomocí speciálního bloku, který tato datová spojení spojí.

Chování bloku Event Execution Control (ECC) je podobné konečnému automatu, který přijme vstupní události a na základě vnitřního stavu ECC vykoná danou část funkce daného funkčního bloku. Ke každému přechodu se dá přidat podmínka v hranatých závorkách k omezení vykonání přechodu. Ke každému stavu lze navázat funkce, která má být spuštěna při přechodu do daného stavu.

Funkční blok umožňuje specifikovat kód v jazyce TypeScript. Následně lze z tohoto kódu vygenerovat kód v jazyce C++ nebo Lua pro nasazení v aplikaci Forte.

System Configuration nám umožňuje nakonfigurovat, kde bude daný systém běžet. Základní schéma nám umožňuje zvolit komunikační linku mezi zařízeními. Na tuto linku následně navážeme jedním či více zařízeními. U těchto zařízení můžeme specifikovat jejich jméno a adresu, na které poslouchají (slouží pro nahrávací SW). Na výběr máme několik předpřipravených zařízení pro různé architektury x64, arm32, arm64 atd.

Pokud máme připravené zařízení, můžeme jej namapovat na jednotlivá zařízení. Ta se nám také objeví v System Configuration v sekci daného zařízení. Zde je funkční blok E_START a slouží pro inicializaci bloků na daném zařízení. Jsou zde tři události: COLD událost, že program začíná od znovu s počátečními hodnotami; WARM znamená, že program pokračuje v bodě, kde skončil před jeho ukončením, a STOP znamená, že se dané schéma má pozastavit.

Propoje na rozhraní dvou funkčních bloků, které jsme umístili na různá zařízení se zobrazí čárkovaně. To znamená, že musíme v System Configuration nastavit bloky pro přenos informací mezi porty. Proto máme speciální bloky PUBLISH_X a SUBSCRIBE_X, kde X udává množství přenesených datových hodnot při odesílání a příjmu hodnot. Tyto funkční

bloky jsou univerzální a lze jimi přenášet jakékoli datové typy. Jsou také nezávislé na použité přenosové technologii. Jednotlivé formáty přenosu specifikujeme portem ID. Základním protokolem je posílání dat v těle UDP paketu shodným se standardem IEC 61499. Dále lze použít některé z předpřipravených technologií HTTP, Modbus RTU, OPC, MQTT, Ethernet Powerlink atd. Případně je možné přidat knihovnu s rozhraním do zdrojových souborů s příponou .com a pomocí CMake vytvořit vlastní spustitelný soubor (FORTE).

Definování sady proměnných je umožněno pomocí virtuálního DNS. Tyto proměnné lze použít jako parametry funkčních bloků nebo zařízení. Před spuštěním daného funkčního bloku jsou všechny tyto proměnné nahrazeny danou hodnotou. Pro použití musí být název proměnné napsán mezi % např. %raspberry%. Pokud následně v tabulce máme dvojici: raspberry 192.168.1.1, je touto hodnotou daná proměnná nahrazena.

2.3.2 4diac FORTE

4diac FORTE je malá přenosná C++ implementace běhového prostředí IEC 61499. Obvykle běží nad operačním systémem v reálném čase, je vícevláknová a paměťově nenáročná. Umožňuje běh na vícero operačních systémech a architekturách.

Můžeme ji získat na stránkách Eclipse, buď ve formě předpřipravených balíčků pro Windows, Linux x64 nebo arm. Tyto balíčky, ale neobsahují všechny bloky. Pro ostatní bloky musíme stáhnout další zdrojové soubory a ty pomocí CMake přeložit. Jednotlivé moduly můžeme zapínat pomocí přepínačů v konzoli nebo v UI. Díky CMake můžeme zdrojové kódy přeložit i na jiné platformy než aktuálně používáme a následně můžeme tyto spustitelné soubory pouze rozdistribuovat.

Forte umožňuje spuštění s přepínačem -c <IP>:<port>. IP specifikuje IP adresu a port, na kterém bude daný program poslouchat. Pro automatické spouštění nebo pro vložení více informací je vhodné použít přepínač -f <file>. Zde specifikujeme soubor a ten se načte do dané aplikace. Soubor má formátování key=value. V daném souboru můžeme specifikovat nejen hodnoty pro běh programu, ale také proměnné pro jednotlivé komunikační moduly (heslo, jméno, certifikát).

Pro nahrávání schématu do zařízení se používá TCP spojení. Schéma je posíláno po jednotlivých elementech (bloky, propoje). Struktura daného paketu je ve formátu XML s kontrolním součtem. První element je Request s atributy ID, které určuje pořadí dotazů a atribut Action, ve kterém jsou specifikovány události (viz následující seznam).

CREATE	vytvoří funkční blok na cílovém zařízením
DELETE	odstraní funkční blok na cílovém zařízení
START/STOP	spustí nebo pozastaví vykonávání funkčních bloků
KILL	zastaví provádění všech funkčních bloků
RESET	nastaví počáteční hodnoty na blocích
READ	přečte jednotlivé vstupy/výstupy na daném funkčním bloku
WRITE	zapíše danou hodnotu na vstupní port funkčního bloku nebo aktivuje přerušení nějaké události na daném bloku
QUERY	slouží jako informace, že se bude nahrávat více funkčních bloků

Uvnitř Request je element FB s atributy. Ty slouží jako unikátní identifikátor daného funkčního bloku a Type určující daný typ. Dále máme element Connection, ve kterém jsou atributy *Source* a *Destination* ve tvaru NAME.PORT. NAME je unikátní název funkčního bloku. PORT je název daného rozhraní na funkčním bloku.

2.3.3 Výstupní formát

Výstupní formát je XML uložený v souboru s příponou .sys. Tento soubor je kódovaný ve formátu UTF-8 a má tři důležité části. První část je samotná aplikace, její bloky a propojení mezi bloky. Druhou částí jsou daná zařízení a funkční bloky namapované na zařízení. Třetí částí je mapování mezi bloky aplikace a bloky na jednotlivých zařízeních.

Zařízení a aplikace mají svoje parametry: pole funkčních bloků, pole datových propojení a pole propojení událostí. Funkční blok má atribut *name* jednoznačný identifikátor, *type* typ funkčního bloku, *comment* pro komentář bloku a pozici bloku ve schématu. Element connection má atributy *Source* a *Destination* ve formátu nazev_element.nazev_portu.

2.4 MQTT

MQ Telemetry Transport je lehký a nenáročný transportní protokol pro předávání zpráv mezi zařízeními (M2M) prostřednictvím centrálního bodu – brokeru. Díky této nenáročnosti a jednoduchosti je snadno implementovatelný do mikrokontroléru. Tento protokol byl navržen firmou IBM, dnes ho spravuje Eclipse Foundation a byl standardizován OASIS.

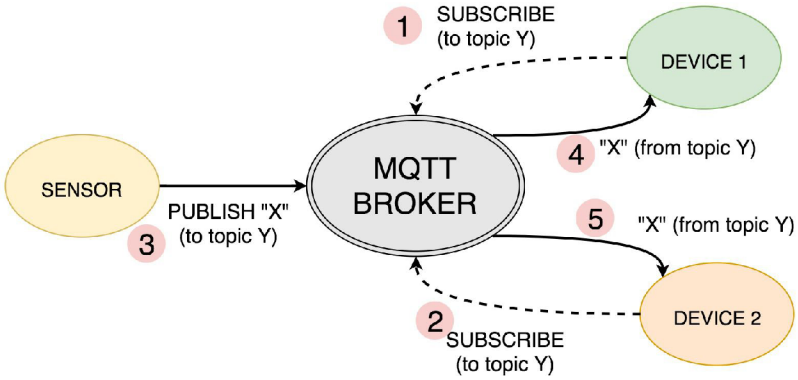
U protokolu MQTT probíhá přenos převážně přes TCP. Používá návrhový vzor Publish/Subscribe. Tento vzor popisuje dva objekty vydavatel (Publisher), co zprávy posílá, a odběratel (Subscriber), který zprávy přijímá. Zprávy jsou předávány pomocí centrálního bodu (MQTT broker). Ten se stará o příjem a výměnu dat. Zprávy jsou tříděny do témat (topic) a každá zpráva patří právě do jednoho tématu. Témata jsou hierarchická, oddělena lomítky a jsou kódovaná v UTF-8. Hierarchie není pevně daná a je na každém, jak si ji navrhne. Vydavatel nijak dané téma nezakládá, pouze pošle na centrální bod zprávu s daným tématem. Naopak odběratel se na začátku přihlásí o odběr zpráv určitého tématu a potom mu MQTT broker přeposílá zprávy s daným tématem. Zařízení může být zároveň vydavatel i odběratel.

Obsah zpráv není nijak specifikován a je na uživateli, jaký formát použije např. JSON, BSON, XML, binární data. Maximální velikost takové zprávy je 256 MB.

V této části se budu věnovat hlavně nejnovějšímu standardu MQTT a to ve verzi 5.0. Tento standard oproti verzi 3.1.1 vylepšuje vlastnosti, jak na straně brokera, tak na straně klienta. Nově umožňuje vlastní autentizaci, vypršení doby relace, odesílání zpráv se zpožděním, prostředky pro přeposílání zpráv mezi brokery. Žádanou funkcionalitou byla také tvorba aliasů k tématu. Alias umožní místo dlouhého tématu mít číselný alias. Příkladem takového aliasu může být, že místo dlouhého tématu /position/home/room1 používáme alias 123. Další novinkou je možnost odeslání zprávy před odpojením serveru, odesílání chybových zpráv, zachování zpráv na serveru nebo sdílené odebírání zpráv. [13] [17]

2.4.1 Ukázka MQTT komunikace

Komunikační protokol používá návrhový vzor Publisher-Subscriber, který je ukázán na obrázku 2.1. Senzory na levé straně publikují data ze senzorů X pod nějakým tématem pomocí zprávy PUBLISH. Tato zpráva přijde MQTT broker a ten zprávu zpracuje. Nové



Obrázek 2.1: Schéma komunikace v MQTT mezi Subscriber, Publisher a Broker. (zdroj: [7])

téma zaregistruje, ale pokud už je zaregistrované a jsou na něm navázaní nějací odběratelé, tak tuto zprávu přepošle. Pokud chce zařízení DEVICE 1 odebírat nějaké téma, musí se nejdříve připojit k BROKERU zprávou CONNECT a ten odpoví zprávou CONNACK. Následně pomocí zprávy SUBSCRIBE se odběratel přihlásí k odběru daného tématu. Při příchodu nové zprávy ze senzoru je dané téma přeposláno na toto zařízení.

2.4.2 Datagram

Velikost daného paketu je proměnlivá, záleží na velikosti užitečné informace a nejmenší možná zpráva má 2 bajty, což je zpráva o odpojení. Maximální velikost zprávy je 256 MB, což je dostatečná velikost pro různé senzory a většina takovou velikost ani nevyužije.

Zpráva MQTT se skládá z Control Header, Packet Length, Variable Lenght Header, Payload. Control Header se skládá z packet type, který určuje typ zprávy CONNECT (připojení k serveru), CONNACK (zpráva server-client), PUBLISH (zpráva client-server), Control flag ten se skládá z DUP, informace o duplicitě, QoS, RETAIN. QoS(kvalita doručení) zavádí tři úrovně 0 – 2. Nejnižší úroveň 0 je, že zpráva bude odeslaná bez potvrzení a není zaručeno doručení. Úroveň 1 – každá zpráva je doručena alespoň jednou. Na zprávu při příchodu je vygenerovaná potvrzovací zpráva PUBACK. Nejvyšší úroveň 2 – každá zpráva je doručena maximálně jednou. Je to nejbezpečnější ale nejpomalejší úroveň kvality služby. Záruku pro potvrzení zajišťuje pomocí čtyřdílného hanshaku. Klient nemusí podporovat všechny typy QoS. RETAIN znamená, zda se má hodnota na serveru uložit. [3]

Seznam možných typů MQTT zpráv:

CONNECT připojení klienta k serveru

CONNACK odpověď serveru klientovi o úspěšném připojení

PUBLISH klient odesílá data na server

SUBSCRIBE po navázaní spojení k serveru zažádá o odběr určitého tématu

SUBACK potvrzení serveru o úspěšné žádosti o odběr

UNSUBSCRIBE klient žádá o zrušení odběru daného tématu

UNSUBACK server potvrzuje zrušení odběru

AUTH	autentizační zpráva mezi zařízeními
PINGREQ	ping klienta k serveru, který může sloužit jako detekce, zda je zařízení stále přípojné k dané síti
PINGRESP	odpověď serveru na ping.

2.5 PowerDEVS

PowerDEVS je multiplatformní open-source softwarový nástroj pro modelování a simulaci diskrétních systémů (DEVS) nebo simulaci hybridních systémů. Umožňuje definovat atomické modely DEVS v jazyce C++. Tyto modely následně graficky propojit do hierarchických blokových diagramů a vytvářet tak složitější systémy. Prostředí PowerDEVS automaticky převádí graficky spojené modely do kódu v C++, který následně provádí simulaci.

Výhodou PowerDEVS je možnost provádět simulace pod operačním systémem v reálném čase (RTAI) synchronizovaným s hodinami reálného času. PowerDEVS má funkci pro propojení s numerickým balíčkem Scilab. Jednotlivé simulace mohou využívat funkce a proměnné z pracovního prostoru Scilabu a získané výsledky lze odeslat zpět do Scilabu k dalšímu zpracování a analýze.[9]

2.5.1 Formální definice DEVS

Discrite Event specification (DEVS) je formalismus, který zavedl v polovině sedmdesátých let Bernie Zaigler. DEVS model zpracovává trajektorii vstupních událostí a podle této trajektorie a počátečních podmínek vyvolá výstupní události. Jednotlivé DEVS modely lze hierarchicky propojovat, čímž lze získat přehlednější abstraktní model. Modely v DEVS jsou diskrétní, atomické a s možností propojení. Takové modely lze přesně a velice efektivně simulovat.

Atomický model DEVS lze nadefinovat jako uspořádanou sedmici.

$$M = (X, Y, S, \delta_{\text{int}}, \delta_{\text{ext}}, \lambda, ta)$$

kde:

- X je množina vstupních událostí (množina všech možných hodnot, které vstupní událost může přijmout),
- Y je množina všech výstupních událostí,
- S je množina stavových hodnot,
- $\delta_{\text{int}}, \delta_{\text{ext}}$ je vnitřní/externí přechodová funkce, která definuje, jak se změní stav systému,
- λ je výstupní funkce. Tato funkce definuje, jak se změní stav systému po uplynutí doby ta a generuje výstupní událost a
- ta je funkce posunu času definující dobu setrvání v daném stavu.

2.5.2 Spojování DEVS modelů

DEVS je velmi obecný formalismus. Pomocí něhož můžeme popisovat složité systémy. Reprezentace komplexních systémů za pomoci přechodových funkcí může být složitá, protože musíme popsat všechny možné situace, které mohou v systému nastat. Většinu složitých systémů lze však popsat složením jednodušších systémů. Teorie DEVS zaručuje, že spojením atomických DEVS modelů vznikají nové atomické systémy, které můžeme hierarchicky zanořovat. Toho docílíme tím, že výstupní porty jednoho subsystému přiřadíme na vstupní port druhého subsystému.

Spojení DEVS atomických komponentů formálně definujeme takto:

$$N = (X, Y, D, \{M_i\}, C_{xx}, C_{yx}, C_{yy}, Select)$$

kde:

- X je množina vstupních portů,
- Y je množina výstupních portů,
- D je množina unikátních jmen jednotlivých komponent, které systém obsahuje,
- $\{M_i\}$ je množina komponentů, kde $i \in D$; M_i, může být atomický nebo spojený model,
- $C_{xx} \subseteq X \times \bigcup_{i \in D} X_i$ párování vstupních portů na vstupy komponent
- $C_{yx} \subseteq \bigcup_{i \in D} Y_i \times \bigcup_{i \in D} X_i$ párování výstupních portů komponent na vstupy komponent.
- $C_{yy} : \bigcup_{i \in D} Y_i \to Y^{\phi}$ párování výstupních portů komponent na výstupní porty komponent
- $Select : 2^D \to D$ je funkce, která určuje prioritu zpráv při současném výskytu více událostí

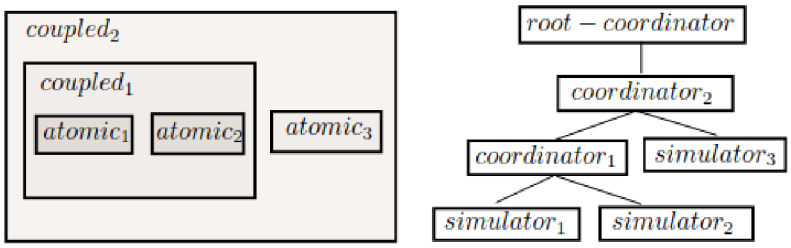
2.5.3 Simulace modelu

K atomickému modelu můžeme vytvořit simulátor, který složité modely simuluje jednoduchým a efektivním způsobem. Simulátor může být napsán v libovolném jazyce. Simulování složeného modelu (více atomických nebo dalších složených modelů) provádíme pomocí koordinátoru. Složením simulátorů a koordinátorů vzniká stromová struktura, viz obrázek 2.2. Stromová struktura má v listech daného stromu simulátory. Ve větvích stromu jsou koordinátory a v kořenovém uzlu stromu je hlavní koordinátor, jenž řídí globální čas simulace.

Koordinátoři a simulátory, kteří jsou navzájem propojení, spolu komunikují pomocí zpráv. Koordinátoři posílají zprávy potomkům, aby vykonali přechodové funkce. Vypočítáním přechodové funkce změní model svůj stav a pokud je přechod interní jeho výstup odešle nadřazenému koordinátorovi. Ve všech případech bude stav simulátoru shodný se stavem atomického modelu DEVS.

Koordinátor a simulátor navzájem komunikují pomocí čtyř druhů zpráv:

- Inicializační zpráva je poslaná rodičovským koordinátorem všem potomkům.
- Interní přechodová zpráva je odeslána přesně danému potomkovi.



Obrázek 2.2: Vlevo máme hierarchický model. Vpravo je model převeden na simulační strom. (zdroj: [9])

- Výstupní zpráva odeslaná od potomka k rodičovskému koordinátorovi slouží jako upozornění, že došlo k vygenerování zprávy.
- Vstupní zpráva slouží k delegaci vnitřní události od rodičovského koordinátora k potomkovi.

Každý blok v simulaci má vlastní lokální proměnnou tn. Tato proměnná určuje dobu, za níž dojde v dané komponentě k dalšímu vnitřnímu přechodu. Čas od poslední události je uložen v proměnné tl. Simulátory tuto hodnotu vypočítají pomocí funkce časového posunu odpovídající atomickému modelu $tn = tl + ta(s)$. Každý rodičovský koordinátor má následně hodnotu tn odpovídající nejmenší hodnotě tn svých potomků. V kořenovém koordinátoru hodnota tn značí dobu, kdy v dané simulaci dojde k další události a t odpovídající aktuálnímu globálnímu času dané simulace. Pomocí těchto hodnot můžeme vypočítat čas od poslední události $e = t - ta(s)$ nebo zbývající čas do nové události $o = tn - t = ta(s) - e$.

2.5.4 Prostředí PowerDEVS

Softwarový nástroj PowerDEVS se skládá ze čtyř nezávislých programů: Model Editor, Atomic Editor, Structure Generator a Preprocesor. Tyto aplikace jsou naprogramované v jazyku Visual Basic. Preprocesor ze zadaného modelu generuje kód v jazyce C++. Tento kód lze následně přeložit na libovolnou platformu a spuštěním odsimulovat model.

Model Editor

Model Editor je hlavním programem, poskytuje grafické rozhraní a propojuje zbylé části dohromady. Zde můžeme spravovat jednotlivé modely a knihovny, spustit simulaci daného modelu tím, že se spustí generátor kódu a Preprocesor. Případně umožní provádět editaci elementárních bloků či definovat nové atomické bloky pomocí Atomic Editoru.

V postranní liště můžeme nalézt atomické bloky rozdělené do jednotlivých knihoven. Bloky, jednoduchým přetažením, můžeme přesunout na pracovní plochu modelu. Jednotlivé bloky můžeme kopírovat, zvětšovat, přidat k nim komentář nebo editovat po kliknutím pravým tlačítkem myši. Zde můžeme nastavit jednotlivé parametry daného bloku. U atomických bloků můžeme editovat vstupní parametry, počet vstupů a výstupů. U abstraktních částí modelu můžeme vstoupit do editace vnitřního modelu.

Atomic Editor

Atomic Editor usnadňuje editaci nebo vytváření nového kódu v C++. Jednotlivé funkce z DEVS formalismu zde reprezentují jednotlivé stránky. V levé části programu můžeme nadefinovat proměnné, které tvoří vstupní a výstupní hodnoty a proměnné reprezentující parametry a stavy systému. Atomic Editor automaticky z daného kódu vytvoří odpovídající zdrojový .cpp a hlavičkový .h soubor. Ty potom používá Preprocessor k vygenerování simulace.

Structure generátor

Model PowerDEVS je kompletně definován po dokončení blokového diagramu. V systému je reprezentován jako soubor s koncovkou .pdm a atomické modely .cpp a .h.

Soubor .pdm má strukturu klíč-hodnota nebo klíč-pole. Jsou zde základní elementy Atomic pro specifikaci atomických bloků, Import – vstupní hodnoty, Outport – výstupní hodnoty, Coupled pro specifikaci části modelu a Line určující propojení jednotlivých bloků. Každý takový blok má atributy, jako je jméno, počet portů a grafickou reprezentaci. Do Couple bloku se hierarchicky zanořují další základní bloky. Atomic block má navíc atributy Parameters, kde lze specifikovat proměnné předávané do bloku a Path, do kterého se specifikuje umístění hlavičkového souboru. Propojení Line má atributy Source a Sink, což je vstup a výstup. Ty se skládají ze 4 hodnot: typ bloku, číslo bloku, číslo portu a směr. Číslování bloku určuje čas vytvoření nebo umístění v souboru. Směr je určen číslem: 0 je výstupní port a -1 je vstupní port. Systém umožňuje i rozvětvení daného propojení tím, že na místo spoje přidá bod, přes který je následně propojen s dalším bodem.

Soubor .pds je výtahem z daného souboru .pdm. Ze vstupního souboru jsou odstraněny nepotřebné atributy pro simulaci. Zůstanou pouze hlavičkové soubory propojení a parametry jednotlivých bloků.

Preprocesor

Vstupem Preprocesoru je blokový diagram v souboru vytvořený Structure generátorem .pdm. Ze vstupního souboru vygeneruje `model.h`, který váže jednotlivé bloky podle jejich propojení. Dále vytvoří Makefile `Makefile.include`.

Následně je zavolán kompilátor C++. Ten vygeneruje spustitelný soubor s názvem `model` (`model.exe` v OS Windows) nacházející se ve složce `powerdevs/output`.

Simulační prostředí

Po vygenerování spustitelného souboru Preprocesorem se spustí okno pro spuštění simulace. Model můžeme spustit jako simulaci nebo jako reálnou aplikaci. Simulační prostředí obsahuje následující parametry:

- Final Time určuje konečný čas simulace,
- Run N simulation umožňuje nastavit počet opakování simulací N pro počítání statistik z daného modelu,
- Perform umožňuje spustit aplikace po určitém množství kroků a
- Ilegimate check break, který automaticky ukončí simulaci po x krocích, pokud se model neposune ve vykonávání simulace.

2.6 Home Assistant

Home Assistant je open-source software pro automatizaci domácnosti napsaný v jazyce Python 3. Slouží jako centrální řídící jednotka v chytré domácnosti zaměřená především na soukromí uživatele. Lze se k ní připojit pomocí webové stránky nebo mobilní aplikace. Umožňuje běh na široké škále zařízení Windows, Linux, Raspberry Pi, NAS zařízení či jako virtualizované prostředí Virtualbox nebo Docker.

Základní frontend dashboard se jmenuje Lovelace a obsahuje různé moduly pro zobrazení informací a správu zařízení. Rozhraní je navrženo v Material Design, je napsané v YAML a je plně přizpůsobitelné.

Po nainstalování a spuštění je aplikace dostupná na adrese `homeassistant.local:8123`. Po nastavení základních údajů začne aplikace vyhledávat v lokální síti různé interakce a nabídne je k přidaní do aplikace. Pokud nějaký modul chybí lze ho doinstalovat ze stránky Home Asistent, kde jich je dostupných více než 1885 modulů např. Amazon Alexa, Google Assistant, ZigBee, HomeKit, MQTT broker, Node-RED, ESPHome, Philips Hue, TP-link, Pi-hole atd.

2.7 IEC 61499

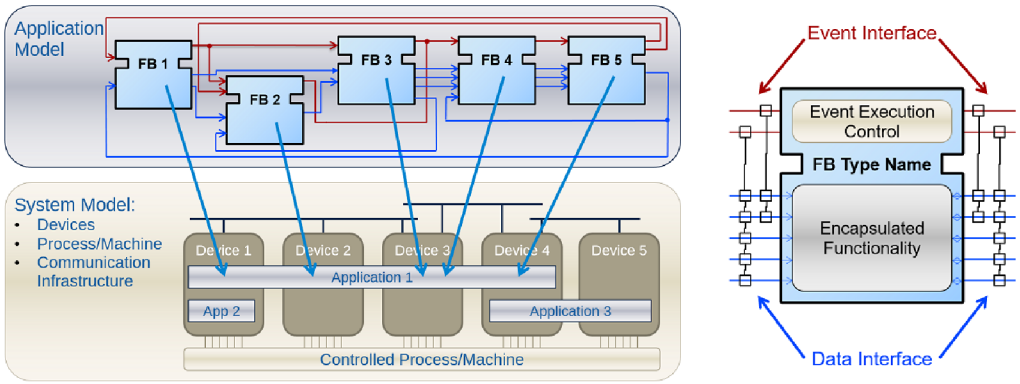
IEC 61499 je standard zabývající se tématem funkčních bloků pro měření a ovládaní průmyslových systémů. Průmysl se posouvá z centralizovaných systému k distribuovaným systémům. V distribuovaných systémech má každé zařízení svoji podúlohu a komunikací mezi nimi vzniká výsledná aplikace. Oproti předchozí verzi IEC 61131-3 se zlepšuje zapouzdření softwarových komponentů pro zvýšení znovupoužitelnosti kódu. Standard poskytuje formát nezávislý na prodejci a zlepšuje podporu komunikace mezi řídícími jednotkami. Podporou distribučního systému a podporou rekonfigurace systému za běhu poskytuje standard potřebnou podporu pro infrastrukturu Průmysl 4.0 a průmyslové IoT aplikace.[5]

IEC 61499 definuje modelovací jazyk, tento jazyk je orientován na distribuované systémy. Standard umožňuje definovat systém jako celek, i když každá jeho část bude na jiném zařízení. Standard definuje plně zapouzdřený funkční blok, jelikož používá pouze lokální proměnné. Norma navíc definuje model reprezentující řízení systému a jejich propojení. To nám umožní namapovat blok na specificky určené zařízení.[10]

IEC 61499 je deterministický, jelikož používá mechanismus událostí pro zahájení operací ve funkčním bloku. Specifikace IEC 61499 neřeší kvalitu implementaci funkčních bloků pro výkon aplikace. Model provádění událostí je vysoce nedeterministický z hlediska předvídatelnosti času provádění s více spouštěči událostí a funkcí. Pokud však použijeme vysoce distribuovaný systém, může se stát předvídatelný, jelikož každý systém bude mít minimum funkčních bloků. [15]

2.7.1 Rozhraní

Každý funkční blok obsahuje určitou funkčnost. Na levé straně má vstupy a na pravé straně výstupy. Vstupy a výstupy se ještě dělí na data (dolní vstupy) a události (horní vstupy). Události spouštějí dané funkcionality funkčního bloku, který využívá data na vstupních portech daného bloku, viz obrázek 2.3 vpravo. Datové porty a porty událostí nejsou stejné a nelze je tak kombinovat. Vstupy událostí nemají žádná omezení, ale datové vstupy mohou mít pouze jeden zdroj výstupu, který může být na více bloků zároveň.



Obrázek 2.3: Vlevo je zobrazena distribuce aplikace mezi koncová zařízení, kde jednotlivé funkční bloky z horní části jsou namapovány na spodní zařízení. Funkční blok na pravé straně znázorňuje rozhraní základního funkčního bloku. (zdroj: [5])

2.7.2 Vykonání bloku

Do bloku vstupuje událost. Pokud tato událost má přiřazené nějaké vstupní datové porty, tak obnoví jejich hodnotu. Následně je událost předána Event Execution Control. V závislosti na typu funkčního bloku se spustí vnitřní funkce bloku a v případě složitějších bloků se spustí i více funkcí. Po dokončení vnitřní funkce bloku, blok poskytne výstupní data. Výstupní data propojené s výstupními událostmi se obnoví a pošle se výstupní událost. Jeden blok může poslat i více událostí než jednu.

2.7.3 Distribuce aplikace

Jelikož jsou funkční bloky na sobě nezávislé, standard IEC 61499 nám umožňuje modelování distribuovaných systémů. Výsledná aplikace nemusí běžet pouze na jednom zařízení, ale může být rozdělena a nasazena na několika zařízeních zároveň, viz obrázek 2.3 vlevo. Na každé zařízení namapujeme jeden nebo více funkčních bloků. Můžeme mít i více aplikací, běžících na jednom zařízení. Nejmenší jednotka v aplikaci je funkční blok, který už nelze rozdělit na různá zařízení. Spojením různých bloků do sebe vniká cílová funkčnost celé aplikace.

2.7.4 Typy funkčních bloků

Basic Function Block (BFB) je základní funkční blok. V BFB definujeme stavový automat ECC, který rozhoduje o provedení algoritmu na základě svého stavu a stavu vstupních událostí.

Composite Function Block (CFB) je funkční blok pro zapouzdření další vnitřní sítě funkčních bloků.

Service Function Block (SFB) je funkční blok, který umožňuje přístup ke konkrétním rozhraním např. síť, HW (GPIO). Používá se, když potřebujeme přístup k rozhraní platformy, protože to BFB a CFB nemůžou. SFB jsou aktivované nejen příchozí událostí, ale také daným rozhraním.

2.8 Petriho sítě

Petriho síť je matematická reprezentace diskrétních distribuovaných systémů. Základy Petriho sítě uvedl v roce 1962 Adam Petrim v disertační práci s názvem Kommunikation mit Automaten (Komunikace skrze automaty). Petriho síť graficky znázorňuje strukturu distribuovaného systému jako orientovaný bipolární graf. Každá hrana má navíc ohodnocení, kolik tokenů je odebráno z místa.

Petriho sítě obsahují místa, přechody a orientované hrany. Hrany jsou vždy pouze mezi místy nebo přechody. Z přechodu vede hrana vždy do místa a z místa vždy do přechodu. Místa obsahují tokeny, které simulují procesy pohybující se ve směru orientovaných hran. Přechody mohou mít omezující podmínku, po jakou dobu v přechodu daný token počká. Přechody mohou mít podmínku, kolik daných tokenů potřebují pro přenos nebo prioritu. Priorita určuje, kde přechod proběhne dříve.

Místa v Petriho síti reprezentujeme graficky pomocí kruhů nebo oválu. Tečky uvnitř místa znamenají počet tokenů na začátku simulace. Přechod graficky reprezentujeme jako plný čtverec nebo obdélník. Omezující podmínky přechodu píšeme pod daný přechod. Ohodnocení hran znázorňujeme jako číslo u dané hrany.[8]

2.8.1 Formální definice

Petriho síť lze definovat jako uspořádanou pětici.

$$N = (S, T, F, M_0, W, K)$$

kde:

- S je konečná množina míst,
- T je konečná množina přechodů
- S a T jsou navzájem disjunktní množiny ($P \cap T = \emptyset$),
- F je konečná množina hran, kde žádná hrana nemůže spojovat dvě místa nebo dva přechody, neboli $F \subseteq (S \times T) \cup (T \times S)$,
- M_0 je počáteční rozložení tokenů v místech,
- W je množina ohodnocených hran, která každé hraně $f \in F$ přiřadí číslo $n \in N^+$, označující počet tokenů odebraných z místa daným přechodem.
- K je množina maximální kapacity. Ta ke každému místu $s \in S$ přiřadí číslo $k \in N^+$, označující maximální počet tokenů v daném místě.

Kapitola 3

Modelovací technika

V kapitole jsou popsány postupy vývoje softwaru pomocí modelu. Pro popis modelu používáme modelovací jazyky. Modelovacích jazyků je mnoho např. v dnešní době nejčastěji používané UML, ArchiMate nebo BPMN. Modelovací jazyk je jakýkoli umělý jazyk s definovanou konzistentní sadou pravidel, kterými lze strukturalizovaně vyjádřit informaci nebo znalost o nějakém systému.

3.1 Model Driven Engineering

Model Driven Engineering (MDE) je metodika vývoje softwaru. Má za cíl definovat modely s nejvyšší mírou abstrakce a zvýšit míru automatizace takových modelů. MDE umožňuje různou míru abstrakce modelu, který lze použít pro popis všech možných problémů. Metody MDE zvyšují produktivitu tím, že zapouzdří určitý problém do standardizovaných modelů a tím umožní její opětovné použití. Také se zjednoduší proces návrhu softwaru použitím návrhových vzorů.

MDE umožňuje jak zvyšování abstrakce, tak i transformaci modelu. Transformace modelu znamená, že model na vysoké úrovni abstrakce transformujeme na modely s nižší abstrakcí, dokud daný model není spustitelný. [14]

3.2 Model Driven Architecture

Model Driven Architecture (MDA) je přístup k návrhu, vývoji a implementaci softwaru od organizací Object Managment Groop (OMG). Hlavním cílem MDA je oddělit business a aplikační logiku od technologické platformy a tím snížit složitost, náklady a urychlit zavedení nových softwarových aplikací. Poskytuje množství pokynů pro strukturovanou specifikaci, které jsou vyjádřeny jako modely. Za základní myšlenku modelu považujeme řízení vývoje architektury spustitelného systému.

Samotné MDA není novou specifikaci OMG, ale spíše přístupem k vývoji softwaru pomocí stávajících specifikací UML, MOF, CWM.[4]

3.2.1 Model nezávislý na počítačovém zpracování

Model nezávislý na počítačovém zpracování (Computation Independent Model – CIM) je velmi obecný model, který se zaměřuje na popis prostředí systému a výsledkem je většinou doménový model. V CIM se reflektují převážně „business“ požadavky zákazníka. Proto musí

být nezávislý na technickém zpracovaní a popisovat systém čistě logicky a věcně. V CIM se nezaměřujeme pouze na daný problém, ale také se snažíme definovat nějaký sdílení zdroj pojmů (slovník). Slovník následně používáme i v ostatních modelech. CIM je důležitý pro překlenutí mezery, která obvykle vzniká mezi programátory a návrháři softwaru.

3.2.2 Model nezávislý na platformě

Model nezávislý na platformě (Platform Independent Model – PIM) vyjadřuje sémantiku činností softwaru nezávisle na použité platformě. Popisuje chování (algoritmy) a strukturu aplikace tak, aby byla přenositelná mezi platformami. PIM model nevychází přímo z CIM modelu, ale jsou z něho vyjmuty jen podstatné části, které považujeme za smysluplné pro potřeby počítačového zpracování konkrétní problémové oblasti. Oproti modelu CIM je doplněn informacemi o algoritmech, principech, pravidlech, omezeních atd. Ty jsou nezbytné pro řešení dané implementační oblasti.

Transformace z CIM do PIM neprobíhá automaticky a to hlavně z důvodu odlišnosti daných modelů. Pro transformaci nejčastěji používáme Use case scénáře, které transformují objektový model do procesního modelu. Velkou výhodou tohoto modelu je jeho znovupoužitelnost, proto často slouží jako výchozí bod pro různá další zadání.

3.2.3 Model závislý na konkretní platformě

Model závislý na konkretní platformě (Platform Specific Model – PSM) kombinuje PIM model s konkrétním technologickým řešením (např. .NET, JAVA EE, COBBRA). Tento model slouží jako vizualizace zdrojového kódu na úrovni abstrakce. Jsou zde přidány objekty související se zvolenou cílovou platformou (např. konstruktory a destruktory tříd, operace pro přístup k atributům apod.).

Z daného modelu je možné generovat zdrojový kód, popřípadě jiné soubory (např. dokumentace, návody), pokud máme plně vyjádřenou sémantiku všech operací v daném jazyce.

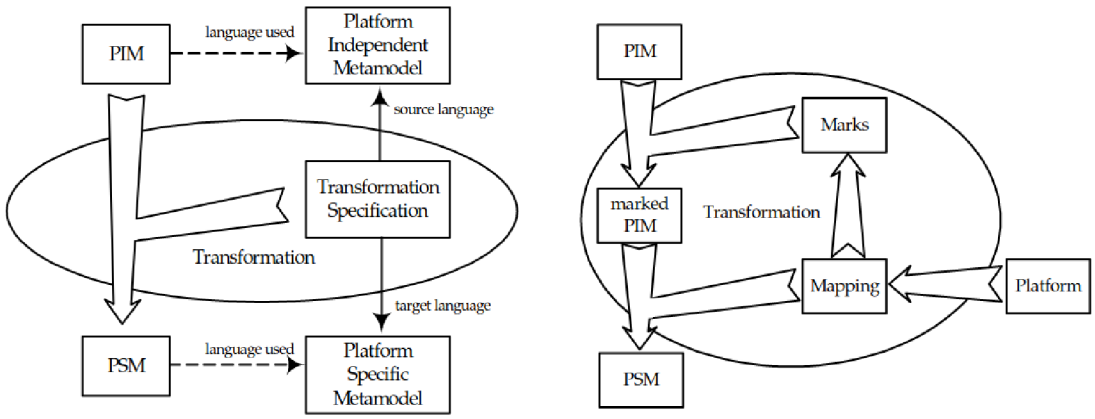
3.2.4 Transformace modelu PIM do PSM

Existuje více způsobů generovaní PSM modelů z PIM modelu. První z nich je generovaní PSM modelu týmem odborníků, který z PIM vytvoří PSM na základě cílové platformy. Ten je však neefektivní a proto MDA umožňuje několik způsobů, jak transformovat tyto vazby.

Prvním způsobem je mapování pomocí meta-modelů. Jednotlivé typy elementů musejí být specifikované ve dvou Meta Object Facility. Jeden pro PSM a druhý pro PIM. Transformace potom probíhá pomocí jasně určených pravidel, která určují protějšky meta-objektů z PIM do meta-modelu z PSM. Mapování pomocí meta-modelu má určitá omezení. Nedovoluje více instancím jednoho typu z PIM přiřadit různé protějšky z PSM.

Druhým způsobem je mapování pomocí značek. Jednotlivé elementy z PIM modelu jsou označeny značkami, identifikující dané mapovací pravidlo. Element může obsahovat více značek a to znamená, že daný element bude každým jednotlivým mapovacím pravidlem.

Ve skutečnosti se při transformaci modelu používá kombinace obou přístupů. Výsledkem transformace je PSM a transformační protokol. Ten umožňuje provádět transformaci v opačném směru. Tyto informace nám umožní udržet konzistenci a kontinuitu vývoje na různých úrovní abstrakce.



Obrázek 3.1: Transformace modelu na cílovou platformu. Vlevo je ukázka schéma transformace modelů pomocí meta-modelů. Vpravo schéma transformace modelu pomocí značek. (zdroj: [11])

3.3 Model Continuity

Model continuity je stále více uznávaná strategie při vývoji vestavěných systémů. Cílem této strategie je postupný vývoj konečného vestavěného systému ze simulačního modelu DEVS bez nutnosti mezikódů nebo opětovné implementace. Umožňuje části konečného systému nahradit simulovaným rozhraním tak, aby mohly být vyzkoušeny hraniční nebo těžce dosažitelné stavy, například nahrazení čidel vestavěného modelu simulovaným čidlem. Výsledky porovnáme s plně simulovaným modelem a tím získáme informace, jak se simulovaný model liší od reálného.

3.4 Model-base design

Metoda Model-base design (MBD) je matematická a vizuální metoda řešení problémů spojených s návrhem komplexních systémů. Cílem je tvorba modelů, které jsou různými cestami transformovány do podoby výsledné aplikace. Nejčastěji se metoda MBD používá při navrhování vestavěných softwarů v průmyslových zařízeních, letectví a automobilovém průmyslu. Pomocí virtuálního modelu mohou vývojáři snadno zjistit, zda celý systém (mechanický, elektrický a vestavěný software) bude fungovat tak, jak bylo zamýšleno, ještě předtím než bude hardware vyroben a připraven k testování.

Metoda MBD je obvykle založená na V-cycle modelu vývoje softwaru. Ten se skládá z těchto vývojových stupňů:

- modelování systému,
- simulace modelu,
- vytváření prototypů,
- generování kódu,
- testování a validace.

Modelování systému je činnost zahrnující vytvoření matematické a vizuální reprezentace cílového vestavěného systému. Modelování reprezentuje schéma bloků, které vzájemně

interagují. Každý model popisujeme standardizovaným jazykem např. UML, BPMN, EPC atd. Jednotlivé jazyky mají různou míru abstrakce definovanou syntaxí a sémantikou daného jazyka. Tuto míru abstrakce musíme promítnout do daného modelu, který omezuje možnost jeho použití. Modely nejčastěji popisujeme pomocí stavových a sekvenčních UML diagramů.

Simulace modelu umožní alternativu k vytváření prototypů hardwaru pro testovací účely. Spojité systémy jsou v simulaci řešené pomocí numerické integrace. Simulaci programu můžeme provádět spojitou simulací, kdy čas simulovaného systému se mění malým konstantním krokem a propočítávají se všechny děje, které v systému probíhají. Nebo pomocí diskrétní simulace, kdy čas simulovaného systému se mění nepravidelně vždy podle toho, kdy má dojít v systému k nové události.

Rychlé prototypování umožňuje ověřovat návrh už v rané fázi vývoje. Využíváme toho, že neimplementované části aplikace můžeme nahradit výstupy ze simulace.

Výsledný kód je následně generován z podrobného modelu řídící jednotky a nahrán do mikrokontroleru nebo vestavěného počítače (Electronic Control Unit – ECU).

Testování provádíme v cyklech, kdy můžeme kombinovat hardware s produkčním kódem v simulovaném modelu. Výsledné dynamické výstupy z produkčního kódu lze porovnávat s daty shromážděnými prostřednictvím simulace modelu. Následně lze na základě výsledků upravovat produkční kód nebo model.

Hlavní výhodou metody MBD je automatické generování kódu, které umožňuje eliminovat lidské chyby a jeho opětovné použití. Dále vytváříme demonstrační aplikace ze simulovaného modelu, iterační vývoj softwaru bez nutnosti použití cílového hardwaru, možnost nalezení chyb ještě před nasazením do cílového systému, souběžná implementace více částí modelu a opakovaně použitelné modely, mající za úkol snížit čas a náklady na vývoj. [2]

Pro simulaci vestavěných systémů můžeme použít Simulink. Simulink je grafické programovací prostředí založené na MATLABU a umožňuje modelování, simulaci a analýzu dynamických systémů. Simulink Coder generuje C nebo C++ kód z modelu v Simulink pro nasazení v široké škále aplikací.

3.5 Vývoj řízený testy

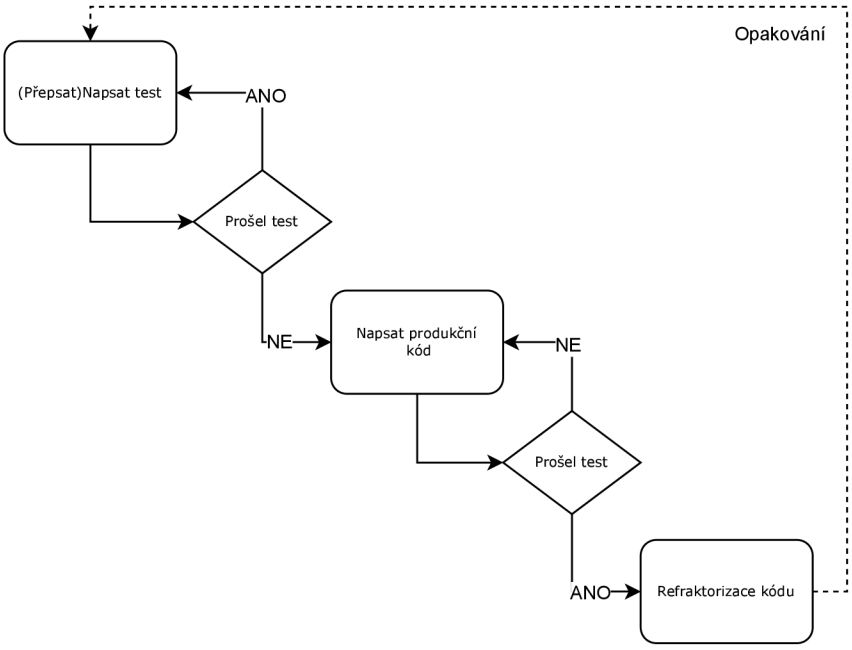
Vývoj řízený testy (Test-driven development – TDD) je přístup k vývoji softwaru. Objevení TTD připisujeme softwarovému inženýrovi Kentu Beckovi v roce 2003. Cílem TTD je vyvíjet aplikaci pomocí malých, stále se opakujících krocích, vedoucích k zefektivnění celého vývoje. Požadavky na software programátor převede na testovací případy, ještě než je software plně implementován. Vždy na konci cyklu pomocí všech testovacích případů otestovat funkčnost softwaru.

Výhodou vývoje kódu pomoci TDD je, že každá jednotlivá funkcionalita, bude mít testovací pokrytí. Neustálým ověřování dopadu změn pomocí testů dává metoda TDD vývojáři větší jistotu, kdykoli jsou provedené změny v kódu, nedochází k žádným negativním vedlejším účinkům na funkčnosti celé softwaru.[1]

Existuje spousta nástrojů, které umožňují automatizaci těchto činností. Většina z nich je integrována do vývojových prostředí.

3.5.1 Vývojový cyklus

Vývojový cyklus se skládá z pěti kroků, viz obrázek 3.2. Tyto kroky se cyklicky opakuji do konce vývoje softwaru.



Obrázek 3.2: Diagram aktivit vývoje softwaru pomocí testy řízený vývoj.

1. Přidání nového testu

   Prvním krokem je napsat testovací scénář na základě potřebné funkcionality cílového softwaru. Tímto získáme definici požadavků na danou funkcionalitu.

2. První spuštění testů

   Nové testy by neměly projít. Tímto zajistíme, že nový test není napsaný špatně a vždy projde.

3. Napsat kód

   V této části můžeme napsat i neefektivní a neelegantní kód, pokud projde testem. Kód následně je optimalizovaný v kroku 5. Kód přidáváme v rámci testovací funkčnosti a nepřidáváme žádný další kód.

4. Druhé spuštění testů

   Všechny testy by měly projít, pokud nějaký neprojde, je potřeba přezkoumat stávající kód a upravit tak, aby prošly všechny testy.

5. Refraktorizace kódu

   Kód je upraven tak, aby byl čitelný a udržovaný. Odstraňují se pomocná data, duplicity kódu. Kód se přesouvá tam, kam logicky patří.

Kapitola 4

Návrh řešení

Návrh řešení jsem rozdělil postupně do několika logických celků. Nejdříve popíšu návrh transformace vnitřní funkcionality atomických bloků. Dále je zde ukázka transformace rozhraní jednotlivých bloků v cílových prostředích, po kterých následuje návrh aplikace pro automatický překlad. Na konci kapitoly je uveden návrh systému, ověřující předcházející návrhy.

4.1 Transformace modelu

V této části bude popsána definice transformace modelu z PowerDEVS do cílového modelu. Cílovou transformaci si ukážeme na příkladu bloku přepínače (switch) viz obrázek 4.1 vlevo. Tento blok má 3 vstupní a jeden výstupní port a jeden parametr uvnitř bloku. Na výstupní port se vždy nastaví hodnota prvního nebo třetího portu podle toho zda je splněna podmínka, která vzniká z vnitřního parametru a druhého portu. V tomto případě je funkce bloku irelevantní a daná transformace bude platit pro jakýkoliv atomický blok.

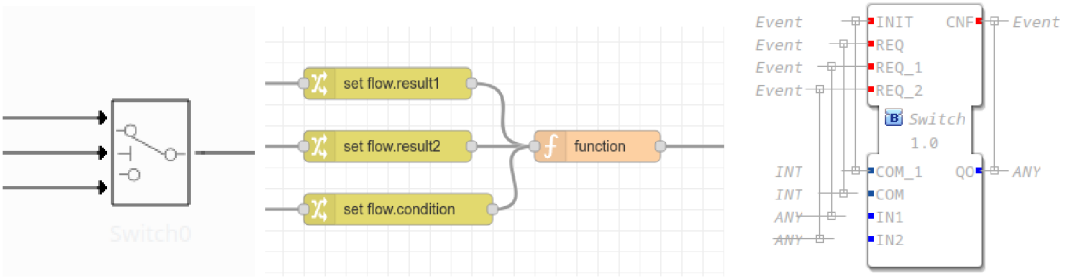
Transformace vnitřní funkce daného bloku může být provedena třemi způsoby. První způsob je vytvoření funkčnosti bloku pomocí již existujících bloků uvnitř modelovacích jazyků. Toto lze použít pouze u 4diac, jelikož obsahuje základní bloky. Node-RED obsahuje pouze univerzální blok, a proto by musely být implementovány nové základní bloky. Výhodou takového řešení je použití nativních běhových prostředí bez nutnosti jejich následné modifikace.

Druhou možností je využití interpretovaného jazyka. Toto lze jednoduše použít v Node-RED, kde by se kód z C++ transformoval do kódu v JS a ten by byl nasazen do funkčního bloku. V případě 4diac by musel být vytvořen speciální blok, který by zajišťoval interpretaci daného kódu. Kód by byl následně přiveden jako parametr vstupu bloku. Výhodou by bylo, že by stačilo pouze jednou vytvořit dané běhové prostředí ve 4diac, nebo čistou instalaci v Node-RED a kód vložit do daného specializovaného bloku.

Třetí možností je využití kódu v C++ uvnitř bloku v PowerDEVS, který by byl upraven pro chod v patřičném modelovém prostředí. V Node-RED je potřeba daný kód zkompilovat a daný balíček bloků vložit do běhového prostředí. V rámci 4diac stačí zdrojové soubory přidat při kompilaci běhového prostředí. Výhoda je minimální úprava zdrojového kódu.

Nezanedbatelná část transformace patří i k atributům a vývojovému prostředí. Jednotlivé bloky jsou v různých prostředí různě velké a tak převod by měl reflektovat novou pozici v prostoru. PowerDEVS používá pro číslování bloků a portů pouze číslo, čímž vzniká problém jak v Node-RED tak ve 4diac, kde daný blok nebo port musí mít unikátní název.

Pokud bychom chtěli reflektovat v PowerDEVS nasazení určitých bloků na dané zařízení. Museli bychom přidat do komentáře k bloku určitou proměnou, která by přesně určovala zařízení k namapování.



Obrázek 4.1: Příklad transformace bloku mezi modelovacími jazyky. Vlevo je vstupní blok switche v PowerDEVS. Uprostřed je výstupní blok pro switch v Node-RED a vpravo je výstupní blok ve 4diac.

4.1.1 Node-RED

Způsob transformace modelu z PowerDEVS do Node-RED vychází z omezení počtu portů pro daný blok. Pokud blok z PowerDEVS obsahuje maximálně jeden vstup, daný model se transformuje jako jeden blok v Node-RED. Pokud má však dva a více vstupních portů, je potřeba tyto vstupy namapovat na určité názvy (topic) tak, aby bylo možno je identifikovat v daném bloku viz obrázek 4.1 uprostřed. Jak je vidět na obrázku, každý vstup byl přepojen do samostatného bloku change. Všechny bloky change byly následně propojeny s daným funkčním blokem (function).

Funguje to následovně, zpráva z výstupu zdrojové zprávy vstupuje do bloku change, který ji přejmenuje na název proměnné. Po vstupu do funkčního bloku je potom přístupná z objektu msg.topic["název proměnné"]. Zde je dobré brát v úvahu, že zprávy z daných čidel nepřijdou ve stejný čas (zprávy se odbavují postupně), je tedy nutné ve funkčním bloku implementovat nějaké pomocné úložiště vstupních proměnných a poslat výstupní hodnotu až poté, co přijdou zprávy ze všech vstupů, je-li to potřeba.

Parametry, které jsou zapouzdřeny uvnitř funkčního bloku, budou uložené jako lokální proměnné nového funkčního bloku v Node-RED.

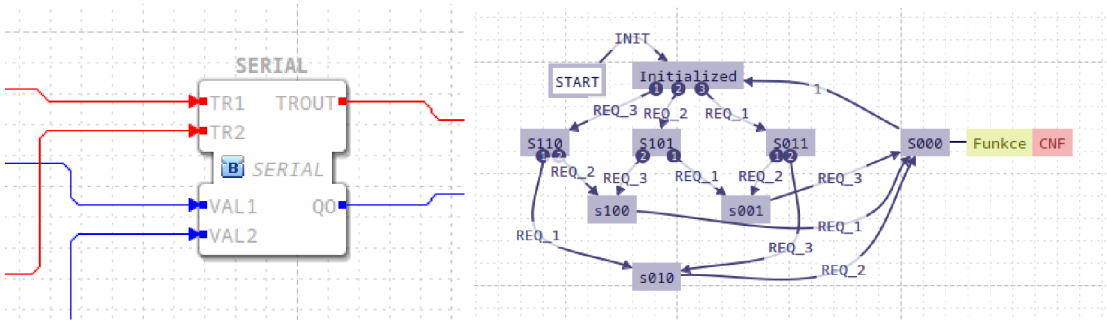
4.1.2 4diac

Problém s transformací z PowerDEVS do 4diac spočívá v rozdělení příchozí zprávy na datový přenos, sloužící k přenosu hodnoty, a přenos událostí, které řídí provádění jednotlivých bloků.

Pokud tedy máme X vstupních datových portů potřebujeme dalších X vstupních událostí, abychom si byli jisti, že na vstupu jsou všechny hodnoty aktuální viz obrázek 4.2 vlevo. Navíc pokud máme vstupní parametry, potřebujeme další inicializační událost, která při startu systému vloží parametry dovnitř bloku. Pokud daný blok nepotřebuje aktuální data od všech datových pinů, je potřeba datové piny inicializovat na nějaké počáteční hodnoty pomocí inicializační události. Každou inicializační událost bloků je potřeba propojit se startovním blokem. Následně je potřeba tyto události patřičně odbavit v ECC jako konečný automat viz obrázek 4.2 vpravo.

Jediné omezení, které 4diac má, je počet vstupních datových propojů do jednoho portu. Tento případ se dá řešit dvěma způsoby. První je přidání patřičného množství portů a následně uvnitř funkčního bloku reflektovat tuto vlastnost. Druhá možnost je použití speciálního bloku před, ten daný paralelní provoz serializuje. K tomu slouží blok SERIAL, viz obrázek 4.2 vlevo, který podle toho jaká přijde vstupní událost, tak takovou hodnotu vloží na výstup a vytvoří výstupní událost.

Jak jsme si ukázali v sekci 2.5, každý takový model je popsán šesticí metod a proměnnými. Funkci Init můžeme reprezentovat jako metodu, která se spustí pokud přijde inicializační událost do bloku nebo při příchodu první zpravy do bloku. Časový posun (Time advance) je nutné implementovat pouze, pokud vykonávání daného bloku trvá nějakou dobu. To lze vyřešit vestavěnou funkcí delay implementovanou ve zdroji událostí. Internal transition, External transition a Output jsou jednotlivé zpracování stejné jako v univerzální funkci. Funci Exit neboli destruktor reprezentujeme jako speciální metodu, která se zavolá při ukončení nebo při restartu aplikace.



Obrázek 4.2: Pomocné úpravy pro převod do 4diac. Vlevo je ukázka serializačního bloku. Vpravo je konečný automat, který zaručí spuštění funkčního bloku poté, co jsou k dispozici všechna data.

4.2 Návrh nástroje pro překlad

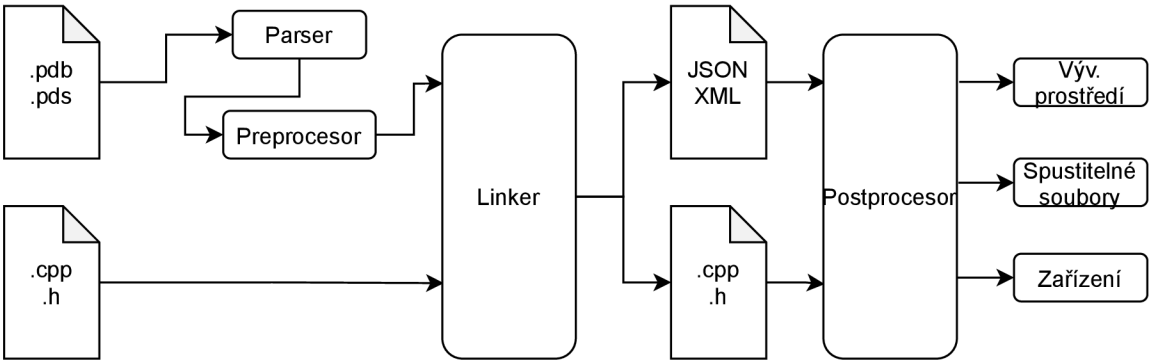
Vývoj cílové aplikace by měl vypadat tak, že se v simulátoru PowerDEVS vytvoří model celé aplikace, který se odsimuluje na testovacích datech. Následně uživatel přeloží určité části modelu a porovná funkčnost se simulovaným modelem. Proto je nejlepší zvolit terminálovou aplikaci, jelikož uživatel zdrojový model přeloží, následně ho v editoru cílového jazyka opraví a poté nahraje na cílové zařízení.

4.2.1 Části aplikace

Vstupem aplikace bude model PowerDEVS v souboru .pdm. Druhým vstupem budou prototypy jednotlivých bloků v cílových jazycích. V části Parser se daný model zkontroluje a vytvoří se z něho pole objektů. Toto pole vstoupí do preprocesoru, který tyto objekty doplní tak, aby byly kompatibilní s cílovým jazykem.

Pole objektů z preprocesoru vstupují do modulu Linker. Tento modul transformuje jednotlivé bloky z PowerDEVS do ekvivalentu bloku v cílovém jazyce. Následně tyto bloky propojí mezi sebou, aby odpovídaly funkčnosti aplikace ve zdrojovém modelu.

Na Linker naváže Postprocesor, tento modul podle zadaných parametrů aplikace vygeneruje buďto výstupní model na standardní výstup, nebo do souboru.



Obrázek 4.3: Schéma aplikace pro transformaci modelu. Diagram zobrazuje vstupní a výstupní soubory. Propojení bloků určuje tok výsledné aplikace.

Parser

Tato část programu zkontroluje vstupní model PowerDEVS. Z modelu extrahuje důležité elementy Atomic, Couple a Line. U Atomic elementu vytáhne název hlavičkového souboru, pokud má daný blok parametry, tak zjistí počty vstupních a výstupních portů a komentář k danému bloku.

Parser doplní názvy k jednotlivým portům a elementům tak, aby byly navzájem unikátní a souhlasily s daným standardem názvů. Tyto názvy následně rozdělí vstupní parametry tak, aby na ně bylo možné v daném programu odkazovat. Dále provede danou transformaci modelů do cílového jazyka. Přidání nových bloků docílí jejich propojení.

Linker

Linker transformuje zdrojový model na nový model, který bude reflektovat strukturu danou objektem vygenerovaným Parser. Výstupem budou zdrojové kódy modelu pro vybranou cílovou platformu.

Postprocesor

Postprocesor umožní nasadit výsledná schémata a funkční bloky. Výsledné zdrojové schéma nahraje do vývojového prostředí. Výsledné zdrojové kódy funkčních bloků přeloží do spustitelného souboru, aby byl spustitelný na cílové platformě.

4.3 Testovací model

Pro odzkoušení překladače, že vzniklá aplikace dokáže překládat i složitější modely, bude vytvořen model z oblasti Smart Home. Cílem bude vytvořit řídící systém bytového domu s třemi byty. Každý byt bude mít obytnou místnost a koupelnu. V místnostech se pomocí řídícího systému bude řídit teplota a větrání. Tyto úkony se budou řídit na základě teploty, vlhkosti, hladiny CO_2 a přítomnosti osob uvnitř místnosti. Jednotlivé hodnoty budou pocházet ze simulovaných senzorů. Simulované senzory budou do aplikace připojené přes síť a budou přenášeny pomocí protokolu MQTT.

Tento řídící kontrolér bude vymodelován v jazyce PowerDEVS a následně odsimulován na testovacích datech. Potom některé části budou převedeny do cílových jazyků, kde budou porovnány výsledné hodnoty.

Pro lepší představu daného modelu naznačím možné komponenty, které by šly u cílového modelu použít. Pro měření teploty použijeme čidlo DS18B20 a pro měření CO_2 senzor MH-Z19B, který umožňuje měřit v rozmezí 0–5000 ppm(0–0.5 %). Vlhkost v místnosti nebo venku budeme měřit senzorem DHT22. Pro regulaci teploty použijeme hlavici SEH01-NC, která má lineární pohon a dobu plného otevření ventilu 1–5 minut. Pro regulaci CO_2 a vlhkosti v místnosti budeme využívat pokojovou rekuperační jednotku Dalap ZEPHIR SIMPLE DOUBLE. Tato rekuperační jednotka nasává čerstvý vzduch z venku a následně ho ohřívá vnitřním vzduchem, který následně vypouští ven. Udávaná účinnost je až 92 % v závislosti na venkovní teplotě.

4.3.1 Řídící systém

Pro řídící jednotku každého bytu máme požadavky. Pro každý pokoj bude možné samostatně navolit požadovanou teplotní úroveň. Nastavená teplota bude vyžadována vždy, když se v dané místnosti bude nacházet osoba. Pokud osoba odejde z místnosti na krátkou dobu (odchod na záchod, vyzvednutí pošty atd.), chceme aby byla aktuální úroveň teploty zachována po dobu dalších 35 minut . V případě, že se v místnosti už osoba nenachází není potřeba ji vytápět na tak vysokou teplotu. Podle normy ČSN EN 12831 jsem zvolil teplotu 15 °C. Podle této normy jsou nastavené i výchozí teploty v místnostech; 20 °C v obývací místnosti, 24 °C v koupelně a 10°C na schodišti.

Řízení větrání bude probíhat automaticky. Podle normy ČSN 06 0210[1] by relativní vlhkost v obývací místnosti měla dosahovat rozmezí 35–60 %, v koupelně 40–90 % a na schodišti 30–60 %. V případě, že se v koupelně nachází osoba, bude po dobu 30 minut omezen chod recyklační jednotky aby se neochlazoval vzduch v místnosti. Koncentrace CO_2, jak je vidět v tabulce 4.1, by v místnosti neměla překročit 1500 ppm a měla by se pohybovat do 1000 ppm.

4.3.2 Human Machine Interface

Pro ovládaní a přehled nad chytrou domácností bude vytvořen monitorovací systém. Tento systém umožní nastavení teploty pro každou místnost zvlášť. Jednotlivé byty by měli mít oddělený monitorovací systém , aby bylo zajištěno soukromí jejich uživatelů. V rámci kotelny by mělo být rozhraní, které umožní vidět celkové statistiky domu pro případnou diagnostiku poruch.

V případě poruchy by takovou skutečnost měl systém propagovat od senzoru, kde vznikla do monitorovacího systému, kterého se daná porucha týká.

[1] https://vetrani.tzb-info.cz/vnitrni-prostredi/9595-hygienicke-pozadavky-na-vnitrni-prostredi-staveb

Koncentrace [ppm]	**Účinky**
cca 350	úroveň venkovního prostředí
do 1000	doporučená úroveň CO_2 ve vnitřních prostorách
1200–1500	doporučená maximální úroveň CO_2 ve vnitřních prostorách
1000–2000	nastávají příznaky únavy a snižování koncentrace
2000–5000	nastávají možné bolesti hlavy
5000	maximální bezpečná koncentrace bez zdravotních rizik
>5000	nevolnost a zvýšený tep
>15000	dýchací potíže
>40000	možná ztráta vědomí

Tabulka 4.1: Účinky CO_2 na lidský organismus. (zdroj: https://vetrani.tzb-info.cz/vnitrni-prostredi/9595-hygienicke-pozadavky-na-vnitrni-prostredi-staveb)

Kapitola 5

Implementace

V kapitole jsou popsány jednotlivé části implementace pro atomické bloky, překladač a testovací model obytného domu.

5.1 Atomické bloky

V této sekci si popíšeme, jak byla vyřešena implementace jednotlivých atomických bloků.

5.1.1 PowerDEVS

V programovacím jazyce PowerDEVS žádné komunikační bloky nejsou a tak bylo potřeba takové komunikační bloky implementovat. Jako knihovnu jsem použil Eclipse Paho C Client Library for the MQTT Protocol[1]. Tato knihovna je ve zdrojových souborech a je potřeba knihovnu nainstalovat. Jednotlivé chybové a informační zprávy z dané knihovny jsou schované do struktury `debug` a tudíž jsou vypisované pouze v ladícím režimu simulátoru.

Na operace nad daty se používají knihovny obsažené v běhovém prostředí PowerDEVS. Pro překlad do ostatních jazyků je nejlepší používat bloky z knihovny Continuous, Discrete, Hybrid a vybrané veličiny ze Sources. Nově implementované bloky byly přidány do knihovny `forte`, jak je vidět na obrázku A.2. Seznam nově vytvořených bloků:

Subscriber V PowerDEVS je implementován jako nový blok MQTT_Subscribe, který v *init* provede parsování parametrů (adresa, topic, QoS, jméno, heslo) a následné navázaní spojení podle zadaných parametrů. Implementace je asynchronní, aby nebránila v průběhu simulace. V případě nové příchozí zprávy se zavolá funkce *msgarrvd*, která danou zprávu zpracuje a zavolá funkci *externalinput*, která informuje simulaci, že bude nový čas následující události (event). Následně Simulace zavolá funkci *lambda* vracející hodnotu příchozí zprávy. Funkce *dext* slouží pouze pro nastavení sigma na hodnotu 0, aby odeslaní zprávy do systému proběhlo co nejdříve. Po ukončení simulace se zavolá funkce *exit*, která zaručí odpojení klienta od MQTT brokeru a dealokaci prostředků.

Publisher V PowerDEVS je implementován jako nový blok MQTT_Publish, který v *init* provede zpracování parametrů (adresa, topic, QoS, jméno, heslo) a následně naváže spojení s brokerem podle zadaných parametrů. Tento blok používá jenom funkce *init* a *dext*. V případě příchozí zprávy ze simulátoru zavolá funkci *dext*. Tato metoda zkontroluje zprávu a odešle se zadaným topikem na MQTT broker.

[1] https://github.com/eclipse/paho.mqtt.c

From CSV Tento blok načte ze zadaného souboru ve formátu `csv` a rozdělí data do pole. Název a cesta je zadány jako parametry funkčního bloku. Druhým parametrem bloku je `K`. Tato hodnota udává dobu mezi vystavením jednoho řádku pole na výstup. Tento blok nám umožňuje simulovat data i na reálných naměřených hodnotách.

Set reset delay Tento blok má dva parametry. První parametr `level` zajistí, že všechny příchozí hodnoty s menší hodnotou jsou na výstupu pozdrženy a všechny ostatní hodnoty jsou okamžitě předány na výstup. Druhý parametr `delay` určí dobu, po kterou je příchozí zpráva s podle hodnoty `level` pozdržena uvnitř bloku.

Petri2Continue Bloky Petriho sítě a bloky spojitého modelu nemají stejný datový typ na vstupu. Tím vzniká nestandardní chování bloků v síti. Tento blok převede výstupní událost Petriho sítě tak, aby mohla být použitá ve spojitém modelu. Tento blok mění pouze strukturu význam zprávy zůstává stejný.

Pro jednoduchou instalaci jsem připravil instalátor `install.sh` v jazyce Bash. Instalátor překopíruje složky s knihovnou do složky s PowerDEVS, která je umístěna v domovském adresáři. Pokud v domovském adresáři není stačí spustit program PowerDEVS. Ten tuto složky vytvoří a překopíruje tam základní knihovny. Nakonec instalátor doplní knihovnu do spustitelného rozhraní a přidá parametry potřebné k překladu s knihovnou MQTT.

5.1.2 4diac

4diac má podporu MQTT zabudovanou jen není zahrnuta ve výchozím modulu forte dostupném na webové stránce[2]. Proto je nutné stáhnout zdrojové kódy ze stránek Eclipse[3]. Pomocí CMake nastavíme, jaké moduly v cílovém modulu forte budeme potřebovat. Doporučené moduly jsou `FORTE_MODULE_CONVERT` pro převod typů, `FORTE_MODULE_IEC61131` pro základní funkční bloky, `FORTE_MODULE_UTILS` pro spojování více datových toků do jednoho nebo zápis dat do csv a `FORTE_COM_PAHOMQTT` pro komunikaci pomocí MQTT zpráv. Poté je ještě nutné zadat cestu ke zdrojovým kódům MQTT a cestu k dynamické knihovně MQTT. Poté už jde daný modul forte sestavit.

Jednotlivé funkční bloky jsou implementovány jako bloky skládající se z již existujících bloků. Tyto bloky mají funkcionalitu shodnou s bloky v PowerDEVS.

5.1.3 Node-RED

Node-RED je knihovna pro komunikaci s MQTT přímo zabudovaná do prostředí, a tak není nic potřeba instalovat. Většina funkčních bloků je implementována skrze funkční blok `function`. Tento blok umožňuje vykonávat kód nad příchozí zprávou v jazyce JavaScipt. Proto bylo potřeba implementovat patřičné funkce, aby jejich vstupy a výstupy odpovídaly funkčním blokům v PowerDEVS.

5.2 Návrh vylepšení PowerDEVS

V případě simulace v PowerDEVS hlavní simulační blok skáče podle následujících událostí a uvedených časů. V případe simulace v reálném čase se vždy hlavní blok uspí do následující události, aby simuloval, že běží v reálném čase. Zde vzniká problém, pokud jsou bloky

[2] https://www.eclipse.org/4diac/en_dow.php
[3] https://git.eclipse.org/c/4diac/org.eclipse.4diac.forte.git

simulace MQTT správně navrhnuty a probouzejí se až externím přerušením, tedy sigma je nastavena na dobu delší než maximální doba simulace. Hlavní problém je, že simulační blok ignoruje všechny následující události větší než maximální doba simulace. Pokud použijeme jako vstup MQTT, jenž čeká na externí událost, tak je simulace hned ukončena.

První příčinou je prázdný zásobník událostí, jelikož v simulovaném časovém úseku není žádná naplánovaná událost. To vyřešit přidáním bloku, který bude mít nastavenou událost před koncem simulace.

Za druhé musí daný blok počkat do následující události, než se znovu probudí hlavní blok a umožní přepočítání heapu (zásobník následujících událostí). Tento problém lze řešit několika způsoby, například pravidelným buzením v rámci bloku MQTT nebo vložením nic nedělajícího bloku, který akorát nutí pravidelně v určitý moment budit hlavní blok.

Řešení problému je několik, například zavedení proměnné následující události přímo do spací funkce. To má výhodu, že pokud by k vyhodnocení externí události docházelo nějaký čas po přijetí, tak by se hlavní blok nemusel probouzet a automaticky by počkal do další události. V práci jsem zvolil metodu externího přerušení, kdy jsem vytvořil proměnnou `wakeup`, která se přidá do spacího cyklu, jak je naznačeno v kódu 5.1 na řádku č. 6. Změna této proměnné při přerušení má za následek, že je znovu probuzen hlavní blok. Hlavní blok po probuzení vypočítá čas do dalšího probuzení a pokud je potřeba rovnou vykoná daný blok nebo se uspí. To má výhodu v případě příchodu více zpráv v krátké časovém okamžiku, protože se vyhodnotí všechny a ne pouze poslední přijatá. Tato úprava neřeší prázdný zásobník následujících událostí, proto je nutné vložit jeden blok, který se vykoná před koncem simulovaného času.

```
1  int waitFor(Time t, RealTimeMode m)
2  {
3      double ti=getRealSimulationTime();
4      if (t<=0)
5          return 0;
6      while (((getRealSimulationTime()-ti)<t) && wakeup) ;
7      wakeup = true;
8      return 0;
9  }
```

Zdrojový kód 5.1: Ukázka úpravy uspávací funkce hlavního řídícího bloku v PowerDEVS.

5.3 Překladač

Překladač je terminálová aplikace napsaná v jazyce Python 3. Je rozdělená do několika modulů podle funkčnosti. Modul `dictionary` slouží pro ukládaní otisku bloků ve formátu JSON pro jazyk Node-RED pro překlad mezi bloky PowerDEVS a Node-RED. Modul `noderedparser` slouží pro transformaci do Node-RED a modul `fourdiacparser` pro transformaci do 4diac.

Vstupní soubor, který je předaný přes první argument, překladač rozdělí do listu (list, měnitelný seznam) nebo slovníku (dict, asociativní pole) podle struktury příchozího souboru. Pokud je v dict více stejných klíčů, tak je z něho uděláno pole. Zpracování souboru probíhá po jednotlivých řádcích rekurzivní funkcí *parsefile*, což odpovídá struktuře programu PowerDEVS, kde je vždy jeden hlavní coupled blok, do něhož jsou zanořeny atomic bloky případně další zanoření pomocí bloku coupled. Jednotlivé řádky se kontrolují po-

mocí regulárních výrazů a následně se pomocí matching group rozdělí podle přiřazeného názvu. Tato funkce slouží také jako kontrola formátu daného vstupního souboru a měla by odpovídat výstupnímu souboru z programu PowerDEVS.

K jednotlivým blokům Atomic, Couple, Inport, Outport jsou ještě přidány klíče idblock, které identifikují pořadí daného bloku, pro propojování bloků mezi sebou.

PowerDEVS jako jediný podporuje point (bod). Point rozděluje jeden propoj na dva. Jelikož ani jeden z cílových jazyků tyto body nepodporuje, je zbytečné s nimi počítat. Proto po načtení všech propojů dojde k zavolání funkce `changepoint`, která všechny propoje mezi point a cílovým blokem nahradí propojením mezi zdrojovým a cílovým blokem. Pokud je propojení mezi dvěma pointy, je potřeba rekurzivně dohledat zdrojový blok a pomocí něho vytvořit nový spoj.

Výsledný datový tok odpovídá průniku všech blokových jazyků. Node-RED je nejbenevolentnější a jako zprávu posílá objekt. 4diac umožňuje přenést zprávu podle typu nadefinovaným v daném jazyce za předpokladu, že se typy shodují. PowerDEVS přenáší mezi bloky pouze ukazatele hodnoty double, které lze prohodit na jiný typ. V práci jsem zvolil pro komunikaci mezi bloky formát PowerDEVS pomocí proměnné double (real v programu 4diac).

Vývoj probíhal na základe metody TTD popsané v kapitole 3. Nejdříve byl vytvořen základní model v PowerDEVS. Ten byl následně cyklicky testován a kód překladače upravován tak, aby výstup z překladače odpovídal vstupnímu modelu. Tímto bylo vytvořeno řada testů pokrývající většinu případů zapojení jednotlivých modulů v PowerDEVS. Tyto testy jsou uloženy ve složce `TEST`.

5.3.1 Node-RED

Nejdříve jsem používal MD5 hash a jako seed jsem používal název daného bloku a výsledek zkrátil na prvních 17 znaků, tak aby odpovídal identifikátoru generovanému pomocí Node-RED. Toto se jeví jako problém, pokud z jednoho bloku musím vytvořit více bloků jako u atomického bloku MQTT, který se rozdělí na dva bloky (přijímající a odesílající) a broker, ten slouží pro připojení k serveru. Proto pro generovaní unikátního identifikátoru používám funkci *token_hex* z knihovny secret. Tato funkce generuje kryptograficky silné náhodné číslo, používané například pro tokeny pro přihlašování. Po nahrání do Node-RED dojde k přegenerování identifikátorů. Generování probíhá už při načítání bloku tak, aby identifikátor byl dostupný pro překlad zanořených bloků.

Jak bylo napsané v kapitole 4.1, je potřeba počítat s omezením, že každý blok má maximálně jeden vstup. Toto řeší funkce *generate_many_input*. Funkce generuje podle počtu vstupních portů bloky *change*, u kterých label topic změní na hodnotu *argx*, kde x je pořadové číslo vstupního portu. Proto u aplikací s více vstupy je potřeba počítat, že tyto vstupy přijdou s patřičným topic, který odpovídá pořadí daného vstupního portu.

Podobným způsobem je řešen i zanořený blok couple (subflow). Zde se pomocí funkce *generate_many_input* generují pomocné bloky change, které umožní spojení na jeden vstupní port. Následně je funkcí *generate_subflow* generován patřičný subflow, v němž jsou generovány pomocné bloky switch, které provádí opačný rozklad na jednotlivé porty.

Generování výsledného Node-RED flow probíhá rekurzivní funkcí, která nejdříve generuje atomic bloky. Funkce získá otisk bloku z modulu *dictionary* podle názvu hlavičkového souboru v prostředí PowerDEVS. Následně vyplní potřebné atributy pocházející z PowerDEVS bloku. Potom vyplní identifikátor, pozici nebo příslušnost k flow. Identifikátor bloku s pořadovým číslem bloku si uloží do struktury *translate*. Reference na strukturu wire s po-

řadovým číslem bloku je uložena do struktury *translateo*. Struktury *translate* slouží pro vyhledání identifikátoru bloku, který se vkládá do struktury wire. Struktura wire je vyhledána ve struktuře *translateo*. Spojením těchto dvou struktur následně vznikají propoje mezi bloky. Takto nově vzniklý blok se vloží do celkového flow. Následně se generují bloky coupled (subflow). Vygeneruje se, jak celý nový subflow podle počtu vstupních a výstupních portů, tak nový blok, který reprezentuje subflow v aktuálním flow. Nakonec si překladač všechny tyto struktury poznamená do pomocných struktur *translateo* a *translatea*.

Nakonec každého bloku coupled probíhá generovaní jednotlivých propojení mezi bloky na základě pole Lines a pomocných struktur *translateo* a *translatea*. Ve struktuře *translateo* se nalezne patřičný zdrojový blok a do něho se nahraje odkaz na následující blok, který získáme z pole *translate*. Zde jsou 4 možné způsoby propojení. Mezi dvěma atomickými bloky, kde se identifikátor následujícího bloku přidá do struktury wire bloku předcházejícího. Mezi vstupním a atomickým blokem, kde je pouze rozdíl v zápisu do struktury in ve zdrojovém bloku. Výstupní blok ale funguje obráceně, do cílového bloku se vkládají identifikátory zdrojových bloků. Pokud je propojení mezi atomickým a výstupním blokem, je nutné do výstupního bloku přidat identifikátor zdrojového bloku. V případě propojení vstupního a výstupního bloku je potřeba do výstupního bloku dát identifikátor daného subflow.

V případě, že daný blok je neznámý, nahradí se daný blok blokem *function*, což umožní generovat jakékoliv schéma v PowerDEVS. Následně uživatel doplní funkce k daným vygenerovaným blokům.

Takto vytvořené schéma se vypíše na standardní výstup nebo souboru, zadaného přes argument programu.

Pokud je přidán přepínač -s Server, kde je zadána url adresa serveru, je na daný server pomocí knihovny `pycurl` odeslán vygenerovaný model v JSON formátu. Pokud server vyžaduje heslo vloží se parametrem -u ve formátu jméno:heslo. Po odeslaní schématu na server lze schéma dále upravovat na daném serveru.

5.3.2 4diac

Pro generování modelu z funkčních bloků jsem musel zavést omezení na funkční bloky. Omezení se týká názvů vstupních a výstupních bloků. Hlavním cílem tohoto omezení je zjednodušit uživateli zadávání nových bloků do překladače. 4diac k propojení dvou funkčních bloků používá názvy portů a zároveň některé vstupní porty slouží jako parametry funkčních bloku. Uživatel by musel do pomocné struktury zanést, jaké vstupní a výstupní porty blok má a jak se následně mají mapovat na příchozí cesty. Proto jsem zvolil omezení na názvy vstupních a výstupních portů jak pro data tak pro signály. Každý funkční blok by měl mít vstupní signál init, který umožní vnitřním blokům inicializaci při startu systému. Všechny vstupní porty by měly mít označení DI (data input) pro data a EI (event input) pro signály s pořadovým číslem začínajícím od jedničky. Výstupní porty jsou pojmenovány totožným způsobem DO (data output) a EO (event output). Pokud funkční blok potřebuje parametr, je nastaven jako vstupní port s názvem PARA. Takto pojmenovaný funkční blok můžeme získat jako podaplikaci skládající se ze základních bloků nebo jako nový funkční blok.

Ze zpracovaného vstupního souboru z PowerDEVS vytvoříme cílový model ve 4diac pomocí funkce `createfourdiacxml`. Překlad probíhá rekurzivně tak, aby odpovídal zdrojovému souboru. Nejdříve se překládají atomické bloky na dané vrstvě modelu. Vyhledávání probíhá ve standardních typech překladače, nebo ve složce zadané jako argument překladače. Pokud se název hlavičkového souboru shoduje s názvem souboru v podadresáři, tak

se tento blok přeloží jako jeden z těchto typů. Následně jsou doplněny jednotlivé argumenty tohoto bloku. Není-li nalezen soubor se stejným názvem jako hlavičkový soubor, provede se vytvoření nové prázdné subaplikace. Toto umožní překlad i bloků, které nejsou ve standardní knihovně nebo ještě nebyly vytvořeny. Uživatel má možnost dodělat funkcionalitu bloku později. Uživatel může nový blok subaplikace uložit mezi typy a následně tento typ používat. Toto umožňuje generovat schéma z neznámých bloků a následně pouze přidat funkcionalitu k chybějícím blokům. Nakonec se vytvoří propoje mezi funkčními bloky. Propoje se rozlišují na základě typu bloku: a) při spojení mezi bloky se k názvu portu přidává i název bloku; b) při spojení se vstupně výstupním portem se uvádí pouze název portu.

Kapitola 6

Testování

V této kapitole si ukážeme jednotlivé scénáře propojení řídících systému, senzorů a aktuátorů v řídícím systému PowerDEVS, Node-RED a 4diac. Transformace dané části modelu by nemělo mít vliv na fungování celého systému. Jednotlivé testovací scénáře vychází principu model continuity.

6.1 Návrh testovacích scénářů

Testovací scénáře,viz obrázek 6.1, vycházejí z modelu continuity. První model je plně simulovaný v programu PowerDEVS a slouží k ověření architektury daného systému. Vyjmutím části komunikace mezi logickými celky přes síť vznikne druhý model. Ten nám lépe umožní distribuovat jednotlivé části na cílové platformy. Třetí model úplně nezapadá do modelu continuity, ale slouží k validaci překladu, kdy se řídicí aplikace nasadí na všechny platformy. Následným porovnáním výstupů z jednotlivých programů můžeme zhodnotit výsledek transformace.

Poslední scénář počítá s nasazením s reálnými senzory a aktuátory. Tento scénář ukazuje, jak by cílové nasazení v reálném prostředí mohlo vypadat.

6.2 Referenční příklad

Jako referenční příklad budeme uvažovat simulaci o délce trvání 3 dny (4320 minut). Pro simulaci venkovní teploty a vlhkosti byly použity vlastní naměřené hodnoty. Tyto hodnoty pocházejí z období 1.–3. března 2022. Toto období bylo vybráno proto, že v březnu byly teploty v noci pod bodem mrazu a přes den naopak vystoupaly až k $10\,°C$.

Pro simulace venkovního množství CO_2 jsem použil data z Global Monitoring Laboratory[1]. Hodnota venkovního množství CO_2 v krátkém časovém horizontu má zanedbatelný rozptyl, a proto jsem použil průměrnou hodnotu za měsíc březen. Pro simulaci delších časových úseků by bylo nejlepší použít vzorkování po měsících, jelikož průběh množství venkovního CO_2 má roční sezónnost.

Počáteční hodnota teploty v pokojích je nastavena na 15 °C. V prvním apartmá je uživatelem nastavená teplota na 20 °C. Uživatel chodí každý den do práce od 6 hodin a vrací se v 16 hodin. Pokud je uživatel doma tak každou hodinu na dobu 10 minut opustí daný pokoj, což simuluje pozdržení úrovně vytápění. Čas, po který má řídící systém udržovat nastavenou teplotní úroveň, je nastaven na 35 minut. Každý den pře a po práci použije

[1] https://gml.noaa.gov/ccgg/trends/weekly.html

uživatel sprchu. V koupelně má nastavenou teplotu 22 °C. Uživatel má problém se spaním pokud je vysoká koncentrace CO_2 proto má nastavenou koncentraci, aby se zapnulo větrání, na 1000 ppm. V rámci odvětrávaní vlhkosti mu stačí vnitřní vlhkost maximálně 90 %.

Ve druhém apartmá uživatel nastavil teplotu na 21 °C. Uživatel chodí do práce na celý den jednou za tři dny. Uživatel každé 2 hodiny odchází na 40 minut. Uživatel má nastavené pozdržení vytápění na 10 minut. Uživatel použije sprchu pouze po práci. V koupelně má nastavenou teplotu na 21 °C. Uživatel má problémy s plísní, a tak chce udržet vlhkost na co nejnižší úrovni kolem 70 %. Maximální úroveň koncentrace CO_2 si uživatel nastavil na 1500 ppm.

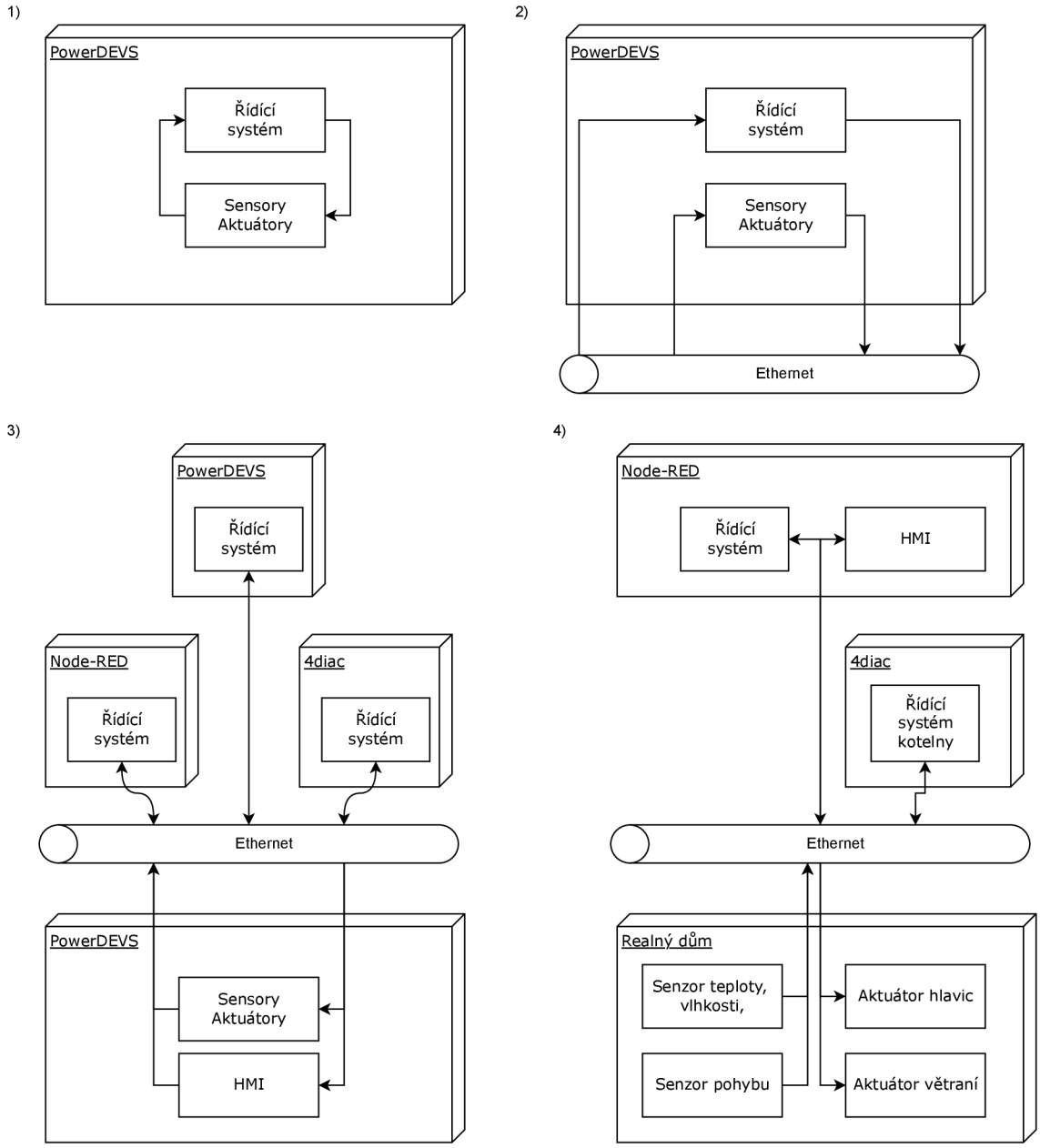
Ve třetím apartmá uživatel nastavil teplotu na 17 °C. Uživatel je stále doma, ale každé 3 hodiny chodí na 2 hodiny pryč. Pozdržení vytápění má nastaveno na 30 minut. Sprchu používá jednou během dne. Nastavenou teplotu v koupelně ma na 24 °C. Uživatel chce šetřit za vytápěni. Nastavil proto maximální koncentraci CO_2 na 3000 ppm. Maximální vlhkost vzduchu na 90 %.

6.3 Výsledky testování

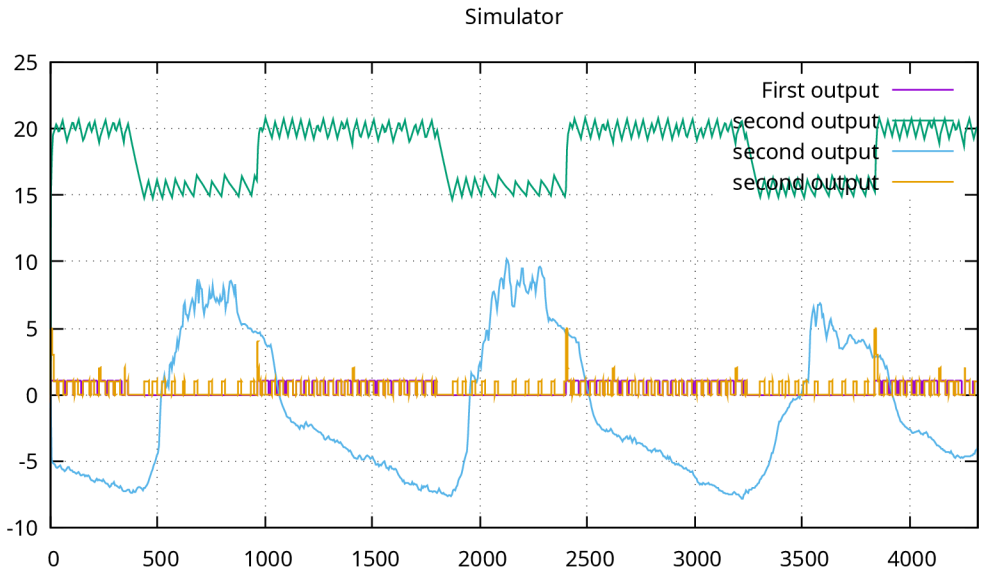
Tato simulace byla nejdříve spuštěna v plně simulovaném prostředím v programu PowerDEVS. Tato simulace trvala 226 ms. Celý rok by tedy trval přibližně 33 s. V následujícím kroku byly hlavní propoje mezi logickými celky nahrazeny komunikací přes MQTT broker. Tím už nemůžeme použít distribuovanou simulaci, ale musíme použít spojitou simulaci, protože posílání zpráv trvá nezanedbatelnou dobu. Tím by daný simulátor mohl být v budoucím časovým horizontu. Ve spojité simulaci musíme vhodně zvolit časový skok tak, aby se stihly poslat všechny zprávy do všech modulů. Tato simulace má stejný výsledek jako simulace s diskrétním časem a tedy vyčleněním komunikace vně simulátoru jsme nezměnili výsledek simulace.

Ve třetím scénáři byly použity řídící jednotky ve všech jazycích. Daný scénář měl všechny apartmány nastaveny na stejné hodnoty tak, aby šly mezi sebou výsledky porovnat. Všechny výstupy byly podobné ale ne stejné. To je působeno delším intervalem vyhodnocením poslání zprávy a také delším zpracování bloků v některých jazycích. Tyto odchylky nicméně nemění funkčnost programu a v delším časovém horizontu jsou zanedbatelné.

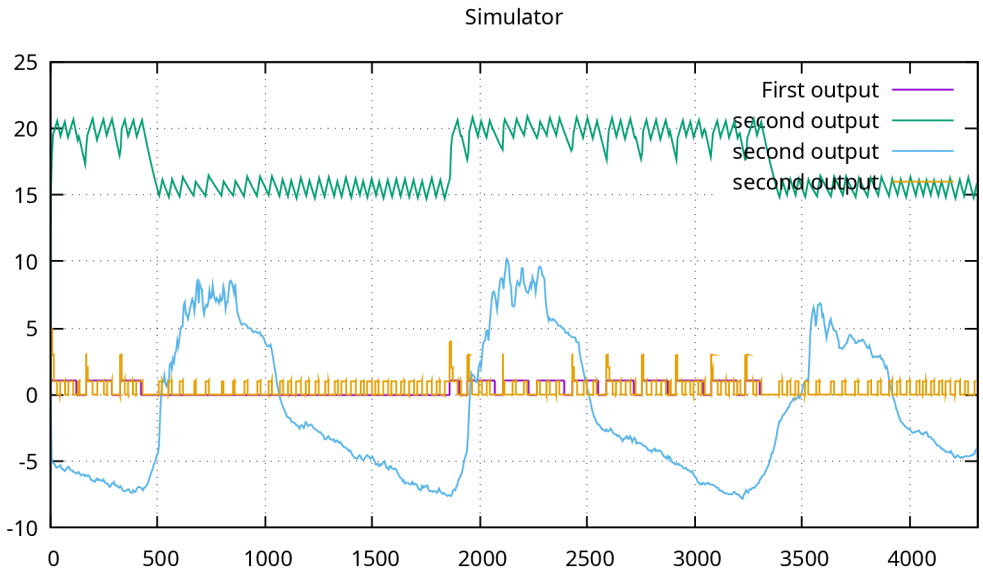
Na grafu 6.2 je v čase 360 minut vidět, že osoba opustila místnost, ale topení ještě topí po dobu dalších 35 minut. V grafu druhého apartmánu 6.3 je vidět v čase od 500–1000 minut nižší úbytek teploty, a tak je nutné méně často otevírat ventil, ale v čase 1400–1900 minut je vidět nutnost častějšího otevírání ventilu.



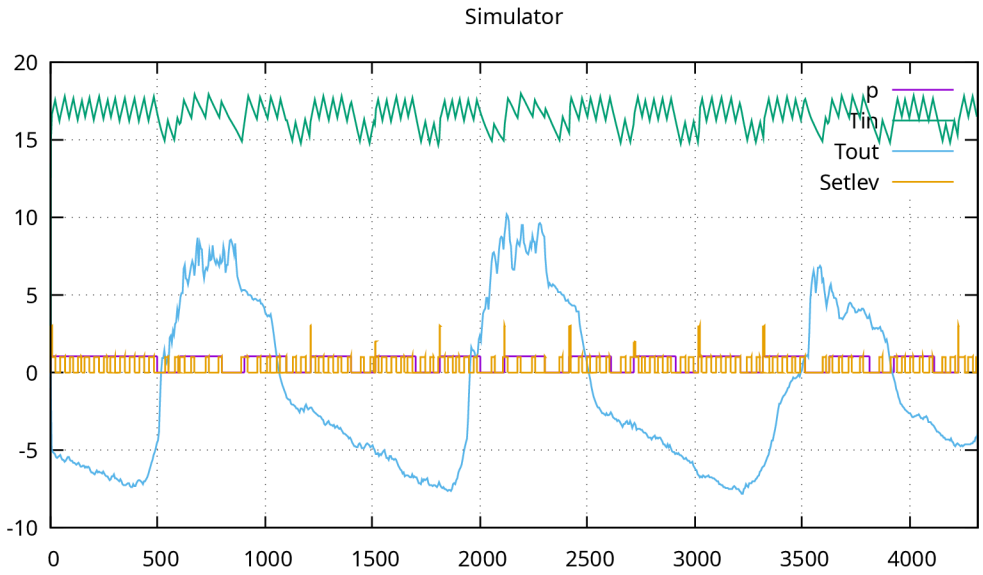
Obrázek 6.1: Schéma jednotlivých testovacích scénářů.



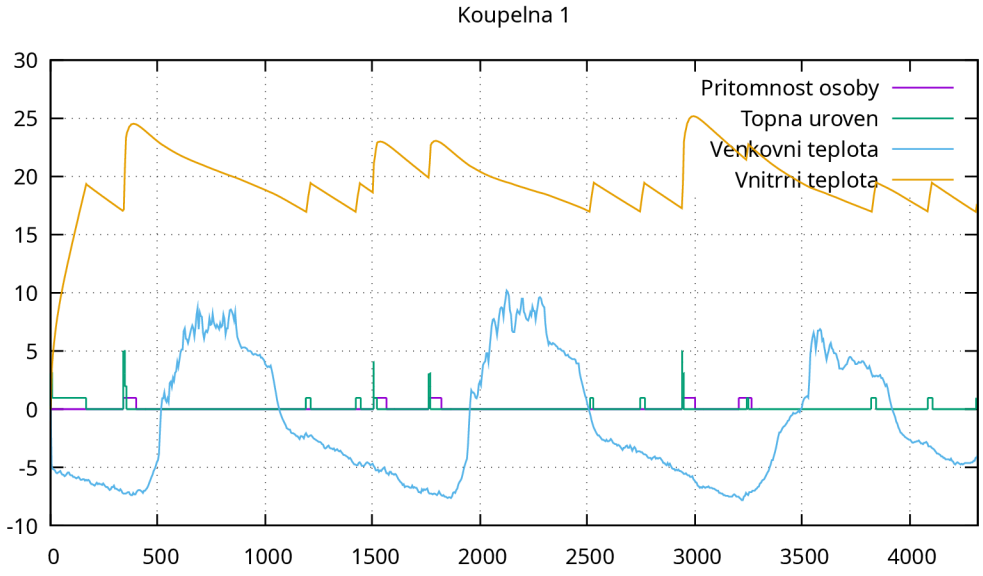
Obrázek 6.2: Průběh teploty v apartmánu 1. Venkovní teplota (modrá), teplota v místnosti (zelená), aktuální topná úroveň (oranžová), přítomnost osoby (fialová).



Obrázek 6.3: Průběh teploty v apartmánu 2. Venkovní teplota (modrá), teplota v místnosti (zelená), aktuální topná úroveň (oranžová), přítomnost osoby (fialová).



Obrázek 6.4: Průběh teploty v apartmánu 3. Venkovní teplota (modrá), teplota v místnosti (zelená), aktuální topná úroveň (oranžová), přítomnost osoby (fialová).



Obrázek 6.5: Průběh teploty v koupelně apartmánu 1. Venkovní teplota (modrá), teplota v místnosti (oranžová), aktuální topná úroveň (zelená), přítomnost osoby (fialová).

Kapitola 7

Závěr

Předložená práce se zabývá návrhem a realizací převodu modelů z PowerDEVS do modelů Node-RED a 4diac. Výsledná aplikace usnadňuje převod z PIM do PSM za využití principu model continuity. Tento překladač ušetří programátorovi až polovinu času, jelikož po naprogramování simulace v PowerDEVS pouze převede aplikaci na cílovou platformu.

PowerDEVS je komplexní simulační nástroj. Jeho nevýhodou je občasné dělání chyb. Převážná porce chyb vzniká v grafickém rozhraní (výsledek neodpovídá uživatelskému zadání). Některé chyby vznikají i v simulačním prostředí. Tyto chyby jsem zaznamenal a navrhl řešení v sekci 5.2. Další chybou je například nepropagování zprávy ze vstupního do výstupního portu. K lepšímu pochopení by přispěla lepší dokumentace k danému prostředí.

V kapitole 4 byly nastíněny možné způsoby transformace, z nichž jsem vybral první variantu, která vyžaduje minimální úpravy původních běhových prostředí. V této kapitole jsou také popsány jednotlivá omezení modelů v Node-RED a 4diac a ukázka jejich řešení demonstrovaná na bloku přepínače (switch).

V kapitole 5 je nastíněna implementace nových atomických prvků pro PowerDEVS. Nové bloky slouží pro komunikaci skrze MQTT nebo pro simulování senzorů. Také je zde nastíněna implementace překladače. Překladač se skládá ze tří modulů. První modul načte model vytvořený v PowerDEVS do vnitřní struktury, kterou poté používají zbylé dva moduly. Druhý modul slouží pro překlad do modelu Node-RED do formátu JSON. Třetí modul provádí překlad do modelu 4diac do formátu XML.

Výsledná terminálová aplikace je napsána v programovacím jazyce Python 3. Tato aplikace umožňuje překlad podle zadaných parametrů do jednoho ze dvou prostředí Node-RED nebo 4diac. V Node-RED překládá na cílové bloky, které jsou v modulu `dictionary` a neznámé bloky nahradí blokem `function`. Ve 4diac překládá cílové bloky podle bloků uložených v typové knihovně a ostatní bloky překládá jako nové subaplikace. Při překladu do Node-RED aplikace umožňuje distribuci na vestavěné zařízení. V případě 4diac vygeneruje složku s projektem, kterou uživatel pomocí importu načte do vývojového prostředí. V prostředí 4diac není podpora přímého nahrávání do vestavěného zařízení.

Nefunkčnost prvního modelu byla způsobena několika faktory. Jedním z nich byl, že mezi Petriho sítí a Spojitý systémem nebyl stejný typ proměnné, a tak vznikaly náhodné hodnoty na blocích `Sum`. Tento problém lze opravit pomocí nového bloku `Petri2Continue`. Druhým problémem bylo nesprávné použití bloku Vector_to_scalar, který nefungoval správně, pokud nebyl připojený jako poslední.

Možnosti dalšího rozšiřování navrženého překladače

- obousměrný překlad, zde bylo potřeba vytvořit nové atomické bloky v PowerDEVS tak, aby odpovídaly blokům v Node-RED nebo 4diac;
- vytvoření transformace bloku podle nastíněných možností v kapitole 4;
- automatického přeskládání bloků ve vývojovém prostředí Node-RED, jako je například ve 4diac;
- provázání Java modulů z 4diac pro generování sekvence bloků pro běhové prostředí forte nebo nahrání skrze síť;
- vytvoření typové kontroly propojů v prostředí PowerDEVS;
- přidání parametru do PowerDEVS, aby šlo odlišit bloky na distribuovaných zařízení.

Literatura

[1] Beck. *Test Driven Development: By Example*. USA: Addison-Wesley Longman Publishing Co., Inc., 2002. ISBN 0321146530.

[2] Chauhan, K. *Why is Model-Based Design Important in Embedded Systems?* 2020. [Online; navštíveno 21.12.2021]. Dostupné z: https://www.einfochips.com/blog/why-is-model-based-design-important-in-embedded-systems/.

[3] Coppen, R., Banks, A., Briggs, E., Borgendale, K. a Gupta, R. *OASIS MQTT Version 5.0*. [Online; navštíveno 29.12.2021]. Dostupné z: https://docs.oasis-open.org/mqtt/mqtt/v5.0/os/mqtt-v5.0-os.html.

[4] Corp., I. *Concept: Model-Driven Development (MDD) and Model Driven Architecture (MDA)*. [Online; navštíveno 05.01.2021]. Dostupné z: https://swi.cs.vsb.cz/RUPLarge/tech.rsa/guidances/concepts/mdd_and_mda_8F9B3685.html.

[5] foundation, E. *Documentation*. [Online; navštíveno 18.12.2021]. Dostupné z: https://www.eclipse.org/4diac/en_help.php?helppage=html%2Fbefore4DIAC%2Fiec61499.html.

[6] foundation, O. *Documentation*. [Online; navštíveno 15.12.2021]. Dostupné z: https://nodered.org/docs/.

[7] Homb, R. R. *MQTT – What Is It? And How Can You Use It?* [Online; navštíveno 03.01.2022]. Dostupné z: https://www.norwegiancreations.com/2017/07/mqtt-what-is-it-and-how-can-you-use-it/.

[8] Jörg Dese, G. J. *What Is a Petri Net?* 2001. [Online; navštíveno 30.05.2022]. Dostupné z: https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.135.1477&rep=rep1&type=pdf.

[9] Kofman, E., Lapadula, M. a Pagliero, E. PowerDEVS: A DEVS-Based Environment for Hybrid System Modeling and Simulation. 2003.

[10] Lydon, B. *IEC 61499 Programming Standard: Will it Become Mainstream?* [Online; navštíveno 01.01.2022]. Dostupné z: https://www.automation.com/en-us/articles/april-2021/iec-61499-programming-standard-become-mainstream#authorInfo.

[11] Miller, J. a Mukerji, J. *MDA Guide Version 1.0.1*. 2003. [Online; navštíveno 08.01.2021]. Dostupné z: https://www.omg.org/news/meetings/workshops/UML_2003_Manual/00-2_MDA_Guide_v1.0.1.pdf.

[12] Ranger, S. *What is the IoT? Everything you need to know about the Internet of Things right now*. [Online; navštíveno 20.04.2022]. Dostupné z:

https://www-zdnet-com.translate.goog/article/what-is-the-internet-of-things-everything-you-need-to-know-about-the-iot-right-now/?_x_tr_sl=en&_x_tr_tl=cs&_x_tr_hl=cs&_x_tr_pto=sc.

[13] Steve. *Understanding the MQTT Protocol Packet Structure*. [Online; navštíveno 21.12.2021]. Dostupné z: http://www.steves-internet-guide.com/mqtt-protocol-messages-overview/.

[14] Truyen, F. *The Fast Guide to Model Driven Architecture*. 2006. [Online; navštíveno 05.01.2021]. Dostupné z: https://www.omg.org/mda/mda_files/Cephas_MDA_Fast_Guide.pdf.

[15] Wikipedia. *IEC 61499*. [Online; navštíveno 29.12.2021]. Dostupné z: https://en.wikipedia.org/wiki/IEC_61499.

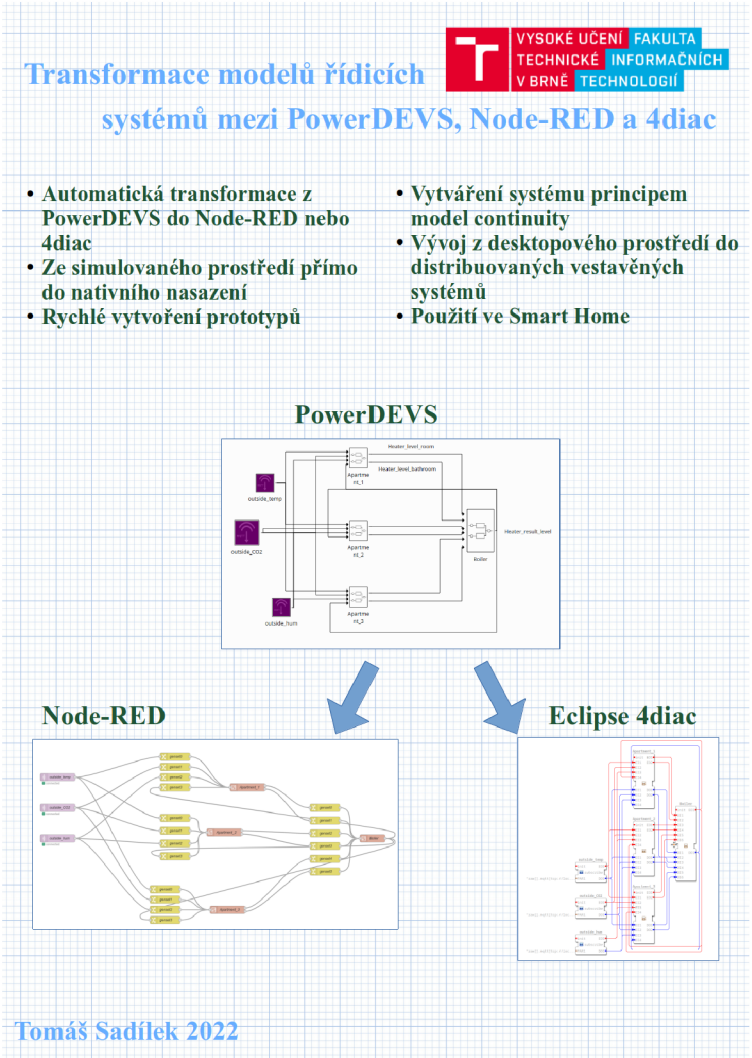
[16] Wikipedia. *Internet věcí*. [Online; navštíveno 28.12.2021]. Dostupné z: https://cs.wikipedia.org/wiki/Internet_v%C4%9Bc%C3%AD.

[17] Yuan, M. *Getting to know MQTT*. [Online; navštíveno 10.01.2021]. Dostupné z: https://developer.ibm.com/articles/iot-mqtt-why-good-for-iot/.

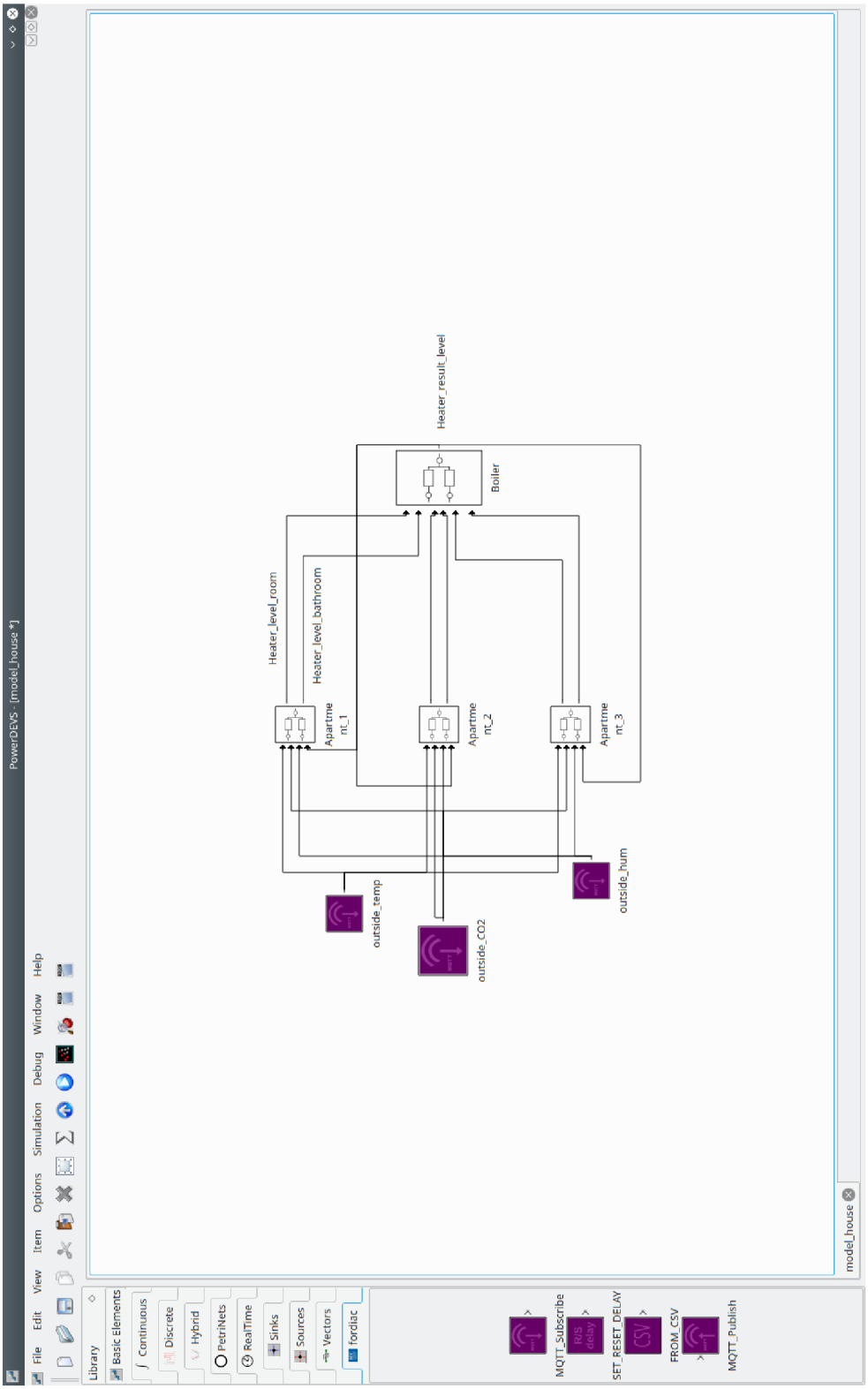
Příloha A

Obrazové přílohy

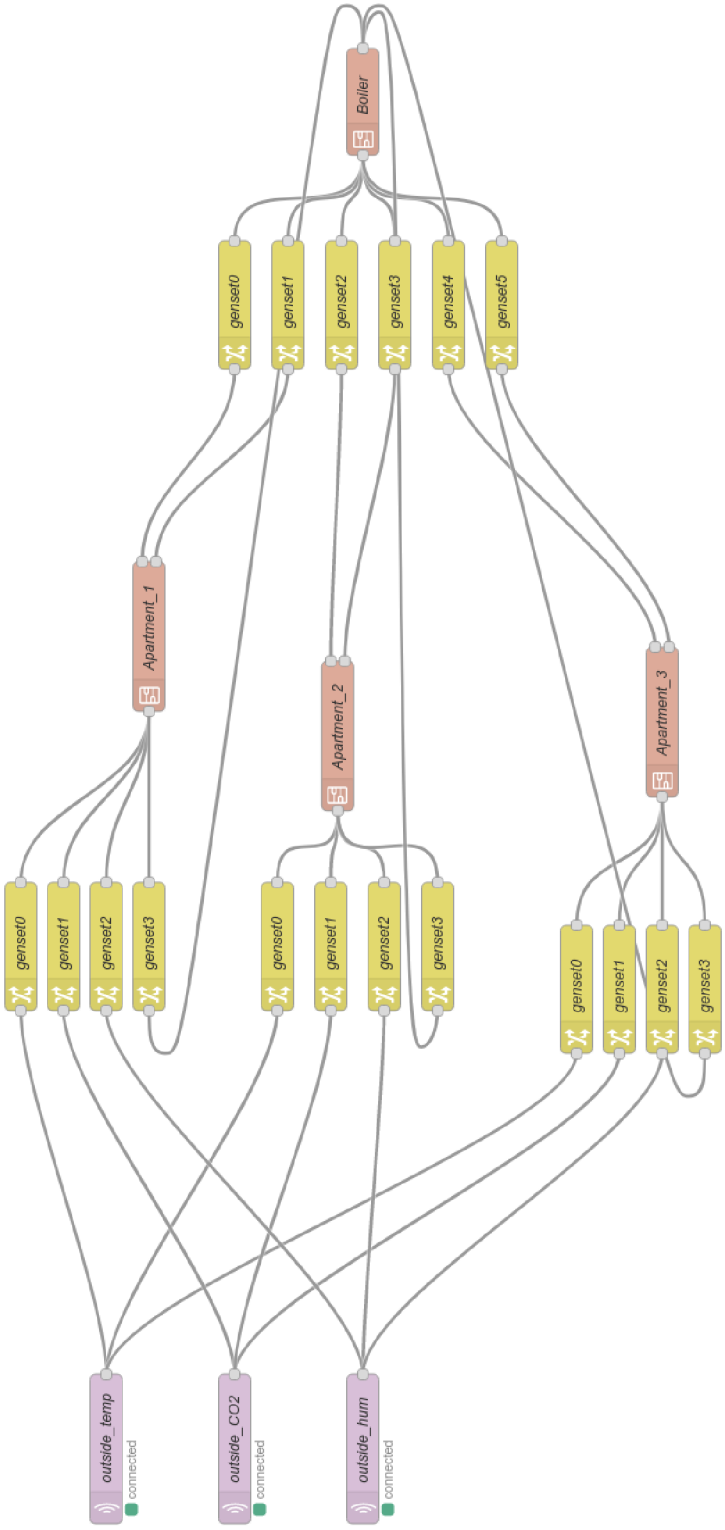
A.1 Plakát



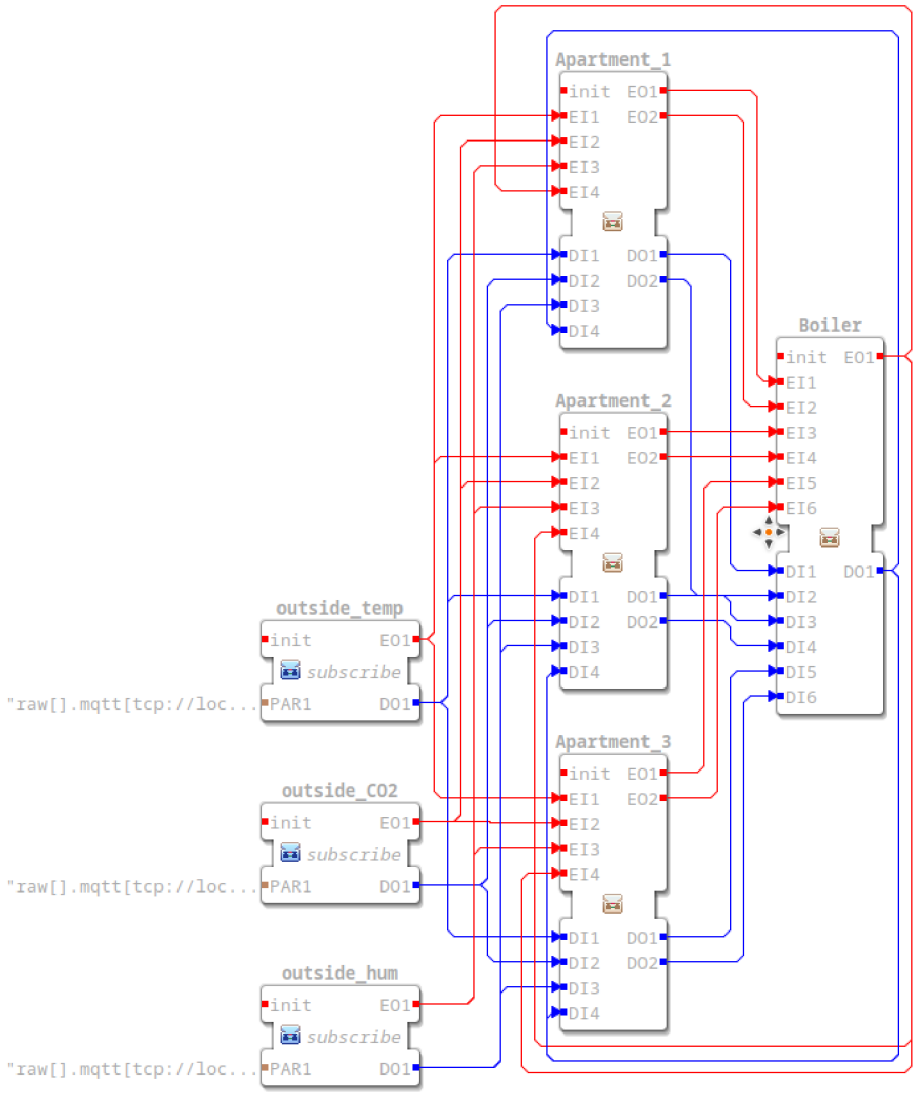
Obrázek A.1: Plakát na prezentaci výsledků (v plné kvalitě je na přiloženém CD).



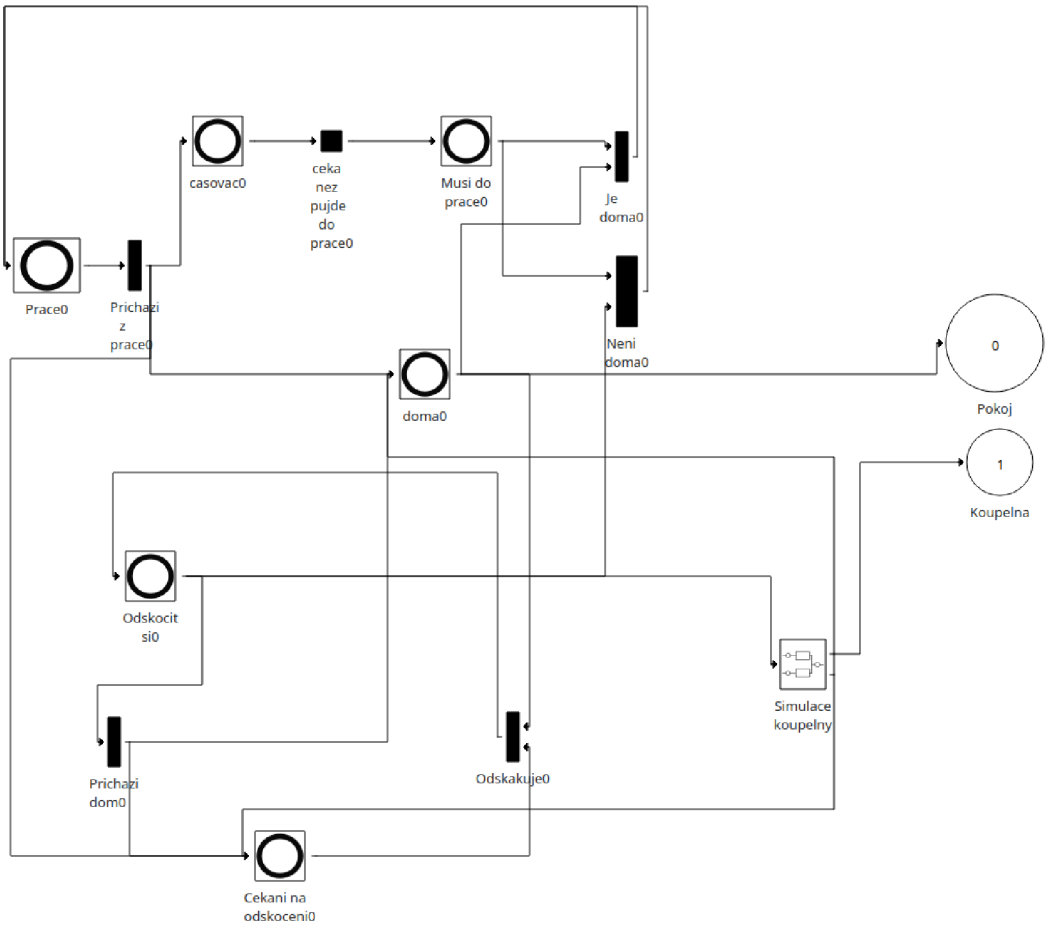
Obrázek A.2: Ukázka prostředí PowerDEVS. V levé části okna vidíme nově vytvořenou knihovnu a nové atomické bloky. Uprostřed vidíme model domu.



Obrázek A.3: Model domu v Node-RED vygenerovaný z modelu v PowerDEVS A.2 pomocí realizovaného překladače.



Obrázek A.4: Model domu v 4diac vygenerovaný z modelu v PowerDEVS A.2 pomocí realizovaného překladače.



Obrázek A.5: Model Petriho sítě simulující přítomnost osoby v obývací místnosti a v koupelně.

Příloha B

Obsah přiloženého CD

- odevzdání
  - Překladač
    - parser.py ..... Aplikace překladače
    - requirements.txt
    - TEST ..... Poloautomatické testy použité při vývoji.
      - test8.pdm
      - test14.pdm
      - test6.pdm
      - test5.pdm
      - test2.pdm
      - test4.pdm
      - test11.pdm
      - test13.pdm
      - test15.pdm
      - test7.pdm
      - test1.pdm
      - test12.pdm
      - test10.pdm
      - test9.pdm
      - test3.pdm
      - Makefile ..... Automaticky překlad všech testů
      - README.md ..... Popis jednotlivých testů
    - library ..... Jednotlivé moduly k překladači
      - fourdiacparser.py
      - noderedparser.py
      - dictionary.py
      - testdemo
        - testdemo.xml
        - testdemo.sys
        - testdemo2.xml
        - .project
        - Type Library
      - xmlibrary
        - fordiac
          - publish.xml

```
        │
        │
        └── sources
            └── triangular_sci.xml
├── model
│   ├── 1modelsimhouse.pdm
│   ├── 2modelcomouthouse.pdm
│   ├── ridicijednotka.json
│   └── ridicijednotka.xml
├── powerDEVS ..... Zdrojové kódy k atomickým blokům
│   ├── atomic
│   │   └── fordiac
│   │       ├── set_reset_delay.cpp
│   │       ├── publish.cpp
│   │       ├── MQTTAsync.h
│   │       ├── publish.h
│   │       ├── set_reset_delay.h
│   │       ├── subscribe.h
│   │       ├── subscribe.cpp
│   │       ├── from_csv.cpp
│   │       ├── from_csv.h
│   │       └── MQTTClient.h
│   ├── library
│   │   └── fordiac
│   │       ├── fordiac.pdl
│   │       ├── rsdelay.svg
│   │       ├── up_tmp.svg
│   │       ├── index.png
│   │       ├── csv.svg
│   │       ├── up.svg
│   │       ├── down_tmp.svg
│   │       ├── index.bmp
│   │       └── down.svg
│   ├── Makefile.include ..... Přidání přepínačů k překladu
│   ├── powerdevs.ini ..... Import knihovny do rozhraní
│   └── install.sh ..... Automatická instalace nových bloku do složky s PowerDEVS
├── Diplomová práce ..... Zdrojový kód k Latex
└── xsadil06.pdf ..... Vysázená diplomová práce
```