VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INTELIGENTNÍCH SYSTÉMŮ
DEPARTMENT OF INTELLIGENT SYSTEMS

NA MODELECH ZALOŽENÝ NÁVRH ŘÍZENÍ SMART HOME

MODEL-BASED DESIGN OF SMART HOME CONTROL

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE **ROMAN ČADA**
AUTHOR

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

BRNO 2021

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav inteligentních systémů (UITS) Akademický rok 2020/2021

Zadání bakalářské práce

Student: **Čada Roman**
Program: Informační technologie
Název: **Na modelech založený návrh řízení Smart Home**
Model-Based Design of Smart Home Control
Kategorie: Softwarové inženýrství

Zadání:

1. Prostudujte problematiku modelem řízeného návrhu se zaměřením na řídicí systémy. Seznamte se s vhodnými nástroji pro modelování řídicích systémů a průmyslovou automatizaci.
2. Seznamte se se soudobými technologiemi pro pro realizaci Smart Home a s dostupným opensource softwarem, jako je Home Assistant, Domoticz a Node-RED.
3. Navrhněte a realizujte řídicí systém pro vybraný subsystém Smart Home s využitím modelů za pomoci vybraných nástrojů z bodu 1. Pro potřeby průběžného ověřování návrhu vytvořte i vhodně abstrahovaný model řízeného prostředí včetně senzorů a aktuátorů a propojení s dohledovým systémem, jako je např. Home Assistant nebo libovolný SCADA systém.
4. Pro porovnání realizujte alternativní systém řízení nástrojem Node-RED.
5. Ověřte funkčnost vytvořeného systému řízení v simulovaném prostředí, proveďte vyhodnocení zvoleného postupu porovnání řešení z bodů 3 a 4.

Literatura:

- 4diac/FORTE. URL: https://www.eclipse.org/4diac/
- Modelio. URL: https://www.modelio.org/
- OpenModelica. URL: https://www.openmodelica.org/
- Node-RED. URL: https://nodered.org/

Pro udělení zápočtu za první semestr je požadováno:

- První 2 body a návrh řešení bodu 3.

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 2020
Datum odevzdání: 30. července 2021
Datum schválení: 11. listopadu 2020

Abstrakt

Tato práce se zabývá návrhem, který je založený na modelu, řídicího systému Smart Home a následnou realizací tohoto systému. Cílem je vytvořit model, který slouží jako návrh, a podle tohoto modelu i realizovat systém. K tomu dobře slouží programovací nástroje, které fungují na bázi modelu. V práci je nejprve popsán na modelech založený návrh, modelovací jazyky, které slouží k tvorbě takového návrhu, a technologie, které se ve Smart Home používají. Na základě těchto poznatků je vytvořen model jako návrh pomocí modelovacího jazyku SysML a následně je tento návrh realizován pomocí programů PowerDEVS a Node-RED.

Abstract

This work deals with model-based design of Smart Home control and implementation of this system. The goal is to create a model that serves as a design and according to this model implement the system. Model-based programming tools do this well. The work first describes the model-based design, the modeling languages that are used to create such a design and the technologies that are used in Smart Home. Based on this knowledge, the model is created as a design using the SysML modeling language and then this design is implemented using PowerDEVS and Node-RED.

Klíčová slova

smart home, domácí automatizace, modelem řízený návrh, princip model continuity, simulačně založený návrh, DEVS, PowerDEVS, Node-RED

Keywords

smart home, home automation, model-based design, model continuity, simulation based development, DEVS, PowerDEVS, Node-RED

Citace

ČADA, Roman. *Na modelech založený návrh řízení Smart Home*. Brno, 2021. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce doc. Ing. Vladimír Janoušek, Ph.D.

Na modelech založený návrh řízení Smart Home

Prohlášení

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

.....
Roman Čada
26. července 2021

Poděkování

Mé poděkování patří za veškerou pomoc a informace od pana docenta Vladimíra Janouška. Dále chci poděkovat za psychickou podporu celé mé rodině a přítelkyni.

Obsah

Seznam zkratek

CASE Počítačově podporované softwarové inženýrství (Computer-Aided software engineering). 3

DEVS Specifikace systému diskrétních událostí (Discrete EVent System Specification). 18–22, 24, 39, 40

FB Funkční blok (v rámci 4diac). 15, 16

IoT Internet věcí (Internet of Things). 10–12, 16

JSON JavaScript Object Notation. 13, 31, 32, 36

MDA Modelem řízená architektura (Model-Driven Architecture). 4–6

MDE Modelem řízené inženýrství (Model-Driven Engineering). 4

MQTT Message Queue Telemetry Transport protokol. 11–13, 17, 24–26, 28, 29, 31, 32, 34–36, 38–40

OMG Object Management Group. 4, 5, 7

QSS Quantized State System. 18, 23

SysML Systems Modeling Language. 6–9, 26

UML Unified Modeling Language. 6–8, 26

Kapitola 1

Úvod

V dnešní době si život bez chytrých zařízení snad ani nedovedeme představit. Vezměme si například chytrý telefon (smartphone). Díky jednomu malému zařízení už nepotřebujeme například tištěný kalendář, budík, fotoaparát či počítač k přístupu na internet. To vše se nám vleze do kapsy jako jediné zařízení a velice nám to zjednodušuje život.

Chytrá domácnost, anglicky smart home, se snaží o totéž. Ať už jsou to světla, jejichž barvu a intenzitu si nastavíme mobilem, brána, která se automaticky otevře a zavře při příjezdu autem domů, kamerový systém, který nám pošle notifikaci do mobilu, že někdo stojí u vchodových dveří, nebo topení, které ví, kdy a kdo obvykle přijde domů, a podle toho předem nastaví teplotu. Díky tomu můžeme i výrazně snížit spotřebu energie, která v dnešní době hraje velkou roli, a tak i ušetřit. Na druhou stranu zavedení chytré domácnosti, která by opravdu dobře a rychle fungovala, bývá většinou drahé.

Existuje mnoho způsobů, jakými se můžeme při návrhu takového systému vydat. Já jsem se zaměřil na návrh, který je založený na modelech. K tomu, abych se při implementaci tohoto návrhu držel, mi pomohly programovací nástroje založené na bázi modelu.

Cílem této práce je poukázat na výhody modelem řízeného návrhu a také na skutečnost, že chytrou domácnost lze vytvořit i levně pomocí programů, které se dají zdarma stáhnout z internetu, a čidel nebo senzorů, které lze zakoupit v řádu maximálně stovek korun. Chytrá domácnost má nekonečně mnoho možností automatizace díky širokému spektru užití. Pokud člověk porozumí principům domácí automatizace, tak si může svůj byt nebo dům nakonfigurovat jak jen bude chtít.

Kapitola 2

Problematika modelem řízeného návrhu

Pokud není uvedeno jinak, v celé této sekci se vychází ze zdroje [2].

V druhé polovině 20. století se softwarový výzkumníci a vývojáři snažili vytvořit abstrakce, které by jim pomohly programovat na základě jejich návrhu a schovaly tak složitosti výpočetního prostředí, jako je například CPU[1], paměť nebo síťové zařízení. Tyto abstrakce zahrnovaly jak jazykové, tak i platformní technologie. Například nízkoúrovňové jazyky (assembly language, Fortran) zaobalovaly pro vývojáře složitosti programování se strojovým kódem. Stejně tak první platformy operačních systémů (OS/360, Unix) zaobalovaly pro vývojáře složitosti programování přímo na hardware. Ačkoliv tyto programovací jazyky a platformy zvedly úroveň abstrakce, stále měly výrazné „výpočetně orientované" zaměření. [21]

Softwarový výzkumníci a vývojáři se tak dále snažili vytvořit technologie, které by ještě více zvýšily úroveň abstrakce a zjednodušily tak vývoj softwaru. Jedním z významných kroků pro toto zjednodušení bylo počítačově podporované softwarové inženýrství, anglicky computer-aided software engineering (CASE), které se zaměřilo na vývoj softwarových metod a nástrojů, tzv. CASE nástroje [28]. Tyto nástroje vývojářům umožnily vyjádřit své návrhy pomocí stavových automatů, strukturních diagramů (člověk lépe chápe obrázek než složitě psané slovo), synchronizaci modelu a zdrojového kódu nebo i vytvořit dokumentaci z modelu. Původní myšlenka CASE nástrojů tedy byla ta, že vygenerují již samotný kód, který nebude potřeba upravovat a bude se moci tak, jak byl vygenerován, použít. Bohužel, CASE nebyl v praxi přijat širokou veřejností, protože se potýkal s řadou problémů. Množství a složitost vygenerovaného kódu byla nad rámec tehdy dostupných překladových technologií, což znesnadňovalo vývoj, ladění a zlepšování CASE nástrojů a jimi vytvořených aplikací. Dalším problémem bylo, že synchronizace mezi návrhem a kódem nebyla úplná. To znamenalo, že když se změnil návrh v pozdějších fázích projektu, musel se změnit i kód. Ještě větší problém představovalo, když se musel aktualizovat návrh, protože se změnil kód. [21]

Později se úroveň softwarových abstrakcí ještě více zvýšila díky zlepšujícím se technologiím a stálému pokroku programovacích jazyků. Rozšířily se tak jazyky 3. generace, které jsou mnohem méně závislé na strojích a jsou tak programovatelnější. Vylepšují podporu

[1] Centrální procesorová jednotka (anglicky Central Processing Unit – CPU) je v informatice označení základní elektronické součásti v počítači.

agregovaných typů a vyjadřují koncepty způsobem, který je pro programátora pohodlnější. Počítač se pak postará o zbytek. [26]

Velký technologický pokrok však přichází i u platforem[2] a navíc přibylo jejich výrobců. Kvůli tomu nastávají problémy, kdy existující programovací jazyky nejsou kompatibilní s novými technologiemi platforem. Z tohoto důvodu vývojáři vynakládají značné úsilí na manuální portování kódu aplikace na různé platformy nebo na novější verze stejné platformy. Například technologie webových služeb se staly natolik složité, že vývojáři stráví roky učením a osvojováním platforem API[3], navíc často pouze s těmi, které pravidelně používají. Jazyky třetí generace navíc vyžadují, aby vývojáři věnovali velkou pozornost imperativním detailům programování. Často se tak nemohou soustředit na to, aby systém pracoval výkonně a správně jako celek. [21]

Slibný přístup jak vyřešit tento problém je modelem řízené inženýrství.

2.1 Modelem řízené inženýrství

Modelem řízené inženýrství, anglicky Model-driven engineering (MDE), je metodologie vývoje software a odvětví softwarového inženýrství. Snaží se o co nejefektivnější návrh softwarových systémů zaměřením na vytváření a využívání modelů. [27]

MDE je přístup k vývoji komplexních, spolehlivých systémů, které se často opakovaně používají. Toho je dosaženo pomocí sady nástrojů, které umožňují vytváření a následnou transformaci modelů do finálního systému. MDE se proto soustředí hlavně na definici modelů a transformačních zařízení namísto ručního psaní a iterativního ladění nízkoúrovňového kódu. Modelem řízené inženýrství se již úspěšně používá v široké škále aplikačních domén a stále více ve vývoji softwaru. [25]

2.2 Modelem řízená architektura

Modelem řízená architektura, anglicky model-driven architecture (MDA), je rozvíjející se standard konsorciem Object Management Group (OMG). MDA podporuje efektivní využívání systémových modelů při procesu vývoje softwaru a opětovné použití osvědčených postupů. OMG propaguje MDA jako způsob vývoje systémů, které přesněji uspokojují potřeby zákazníků a které nabízejí větší flexibilitu při vývoji systému.

2.2.1 Principy MDA

Pohled OMG na modelem řízenou architekturu tvoří čtyři principy:

1. Modely vyjádřené v dobře definované notaci jsou základem pro pochopení systémů pro zákazníka.
2. Zavedením řady transformací mezi modely uspořádaných do architektonického rámce vrstev lze organizovat budování systému.
3. Formální základ pro popis modelů v sadě metamodelů umožňuje smysluplnou integraci a transformaci mezi modely a je základem pro automatizaci prostřednictvím nástrojů.

[2] Počítačová platforma je pracovní prostředí, jak po stránce hardware (stavba počítače – procesor, paměti, připojené periférie), tak i software (operační systém, knihovny). Toto prostředí umožňuje činnost programů. [32]

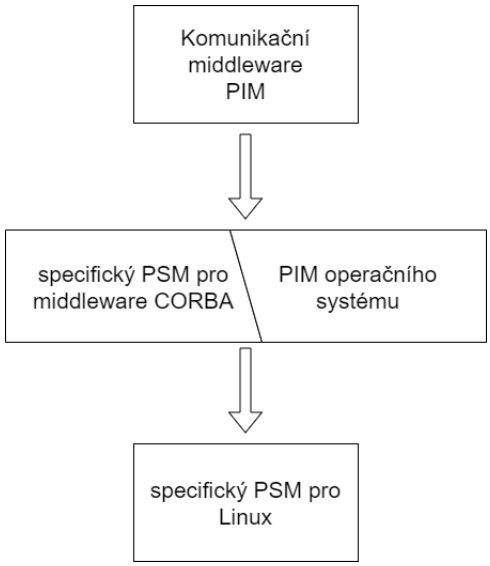
[3] API (zkratka pro Application Programming Interface) označuje v informatice rozhraní pro programování aplikací.

4. Přijetí tohoto modelového přístupu širokou veřejností vyžaduje, aby průmyslové standardy poskytovaly zákazníkům otevřenost a podporovaly konkurenci mezi prodejci.

Pro tyto principy OMG definovala řadu vrstev a transformací, které poskytují koncepční rámec a slovník pro MDA. Modelem řízená architektura se tak člení na tyto typy modelů:

- Model nezávislý na počítačovém zpracování (Computation Independent Model – CIM)
- Model nezávislý na platformě (Platform Independent Model – PIM)
- Model specifický na konkrétní platformě (Platform Specific Model – PSM)
- Zdrojový kód aplikace, neboli výsledná implementace

Pro MDA má „platforma“ smysl pouze ve vztahu ke konkrétnímu úhlu pohledu – jinými slovy, PIM jedné osoby je PSM jiné osoby. Například v modelu PIM modelujeme komunikační middleware[4] bez toho, aniž bychom museli vybrat konkrétní technologii. Když se však rozhodne použít konkrétní technologie middleware, např. CORBA, model se transformuje na PSM specifický pro middleware CORBA. Avšak nový model může být stále PIM s ohledem na výběr konkrétního middlewaru, cílového operačního systému a hardwaru. Tento příklad je znázorněn na obrázku 2.1.



Obrázek 2.1: Příklad transformací z PIM na PSM, převzato z [2].

Ve výsledku můžou nástroje MDA podporovat transformaci modelu v několika krocích. To znamená od modelu počáteční analýzy až po spustitelný kód.

Rozdělení do různých typů modelů je kritickou součástí metodiky vývoje softwaru pro situace, které zahrnují zdokonalení různých aspektů systému, přidání dalších podrobností k modelu nebo převod mezi různými druhy modelů. Zde jsou popsány tři důležité myšlenky s ohledem na abstraktní povahu modelu a podrobnou implementaci, kterou představuje.

2.2.2 Klasifikace modelu

Modely můžeme snadno klasifikovat podle toho, jak explicitně reprezentují aspekty cílených platforem. V celém vývoji softwaru a systémů existují důležitá omezení vyplývající z výběru

[4]Middleware je software, který je mezi operačním systémem a aplikacemi, které jsou v něm spuštěné.

programovacích jazyků, hardwaru, topologie sítě, komunikačních protokolů, infrastruktury atd. Všechny tyto věci lze považovat za prvky výsledné platformy. Přístup MDA nám pomáhá se soustředit na požadavky zákazníka odděleně od detailů této platformy.

2.2.3 Nezávislost platformy

Pojem „platforma“ je poměrně složitý a vysoce závislý na kontextu. V některých případech může být platformou operační systém a nástroje s ním spojené, v jiném to může být technologická infrastruktura reprezentována dobře definovanými knihovnami (například J2EE, .NET), také to může být konkrétní instance nějaké hardwarové topologie. V každém případě je důležité se zamýšlet nad tím, jaké modely se používají v různých úrovních abstrakce ke konkrétním účelům, než na to, jak přesně definovat tuto platformu.

2.2.4 Transformace a upřesnění modelu

Uvažujeme-li o vývoji softwaru jako o řadě upřesnění modelů, stávají se transformace mezi modely nejdůležitějšími prvky vývojového procesu. Definování těchto transformací vyžaduje specializované znalosti požadavků zákazníka, technologií potřebných pro implementaci nebo obojího. Můžeme zlepšit efektivitu a kvalitu systémů explicitním zachycením těchto transformací a jejich opětovným použitím napříč řešeními. Například mezi návrhovými modely vyjádřenými v Unified Modeling Language (UML) a implementacemi v J2EE můžeme v mnoha případech použít dobře pochopené transformační vzory UML-to-J2EE.

2.2.5 Metamodely

Podkladem modelových reprezentací a podporou pro transformace je sada metamodelů. Analyzovat, automatizovat a transformovat modely vyžaduje jednoznačný způsob, jak popsat sémantiku modelů. Metamodely jsou tedy modely, které popisují tyto modely. Například standardní sémantika a zápis UML jsou popsány v metamodelech, které byly použity pro implementaci jazyka UML. Metamodel UML popisuje podrobně význam třídy, atributu a vztahů těmito dvěma koncepty. Dalšími jazyky, které jsou popsány v metamodelech, jsou například SysML nebo Modelica.

Kapitola 3

Modelovací jazyky

3.1 Modelica

Tato sekce vychází ze zdroje [18].

Modelica je volně dostupný objektově orientovaný jazyk pro modelování velkých, komplexních a heterogenních fyzických systémů. Je vhodný pro vícedoménové modelování, jako jsou například mechatronické modely v robotice, automobilovém a leteckém průmyslu zahrnující mechanické, elektrické, hydraulické a řídicí systémy, dále pak modely procesně orientovaných aplikací a modely pro generování a výrobu elektrické energie.

Modelica je navržena tak, aby mohla být použita podobným způsobem, jako když inženýr staví reálný systém. Nejprve se snažíme najít standardní komponenty jako jsou motory, čerpadla nebo ventily v katalozích výrobců s vhodnými specifikacemi a rozhraními. Pouze v případě, že neexistuje patřičný subsystém, by se nově sestavil komponentový model na základě standardizovaných rozhraní.

Modely jsou v Modelice matematicky popsány diferenciálními, algebraickými nebo diskrétními rovnicemi.

3.2 Jazyk UML

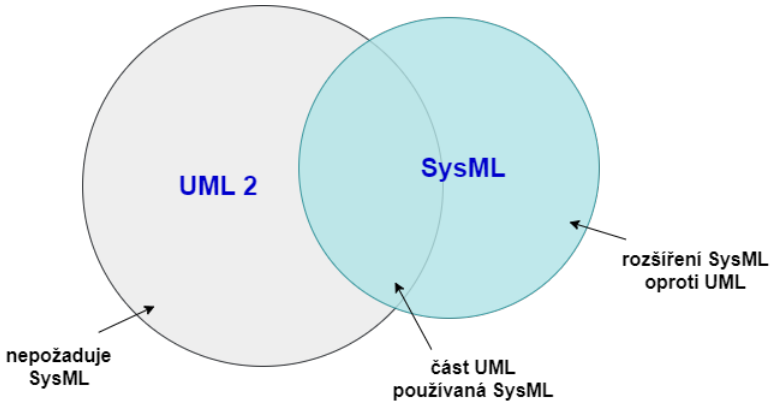
Unified Modeling Language je grafický jazyk, jehož cílem je poskytnout systémových architektům, softwarovým inženýrům a vývojářům softwaru nástroje pro analýzu, návrh, implementaci a dokumentaci softwarových systémů. Umožňuje standardním způsobem zapisovat návrh systému včetně konceptuálních prvků, jako jsou business procesy a systémové funkce, a konkrétních prvků, jako jsou příkazy programovacího jazyka, databázová schémata a znovupoužitelné programové komponenty. Tento jazyk podporuje objektově orientovaný přístup k analýze, návrhu a popisu počítačových systémů. Standard UML definuje standardizační skupina Object Management Group. [14]

3.3 Jazyk SysML

Tato sekce vychází ze zdroje [15].

Systems Modeling Language (SysML) je obecný jazyk grafického modelování pro specifikaci, analýzu, návrh a verifikaci komplexních systémů, které mohou zahrnovat hardware, software, informace, personál, procedury a zařízení. Vyvinulo ho OMG společně s International Council on Systems Engineering (INCOSE). Tento jazyk poskytuje zejména grafická

znázornění se sémantickým základem pro modelování systémových požadavků, chování, struktury a parametrů, které se používají k integraci s jinými modely technické analýzy. SysML představuje podmnožinu jazyka UML 2 s rozšířeními potřebnými pro splnění požadavků UML pro systémové inženýrství. Tento vztah je uveden na obrázku 3.1.



Obrázek 3.1: Vztah mezi UML 2 a SysML, převzato z [15].

Pro modelování v tomto jazyku můžeme použít například internetovou aplikaci Diagrams.net[1] nebo program Modelio, což je modelovací prostředí, které podporuje řadu modelovacích jazyků včetně UML a SysML. [12]

3.3.1 Shrnutí diagramů SysML

Blok je základní jednotkou struktury v SysML a lze jej použít k reprezentaci hardwaru, softwaru, vybavení, personálu nebo jakéhokoli jiného prvku systému.

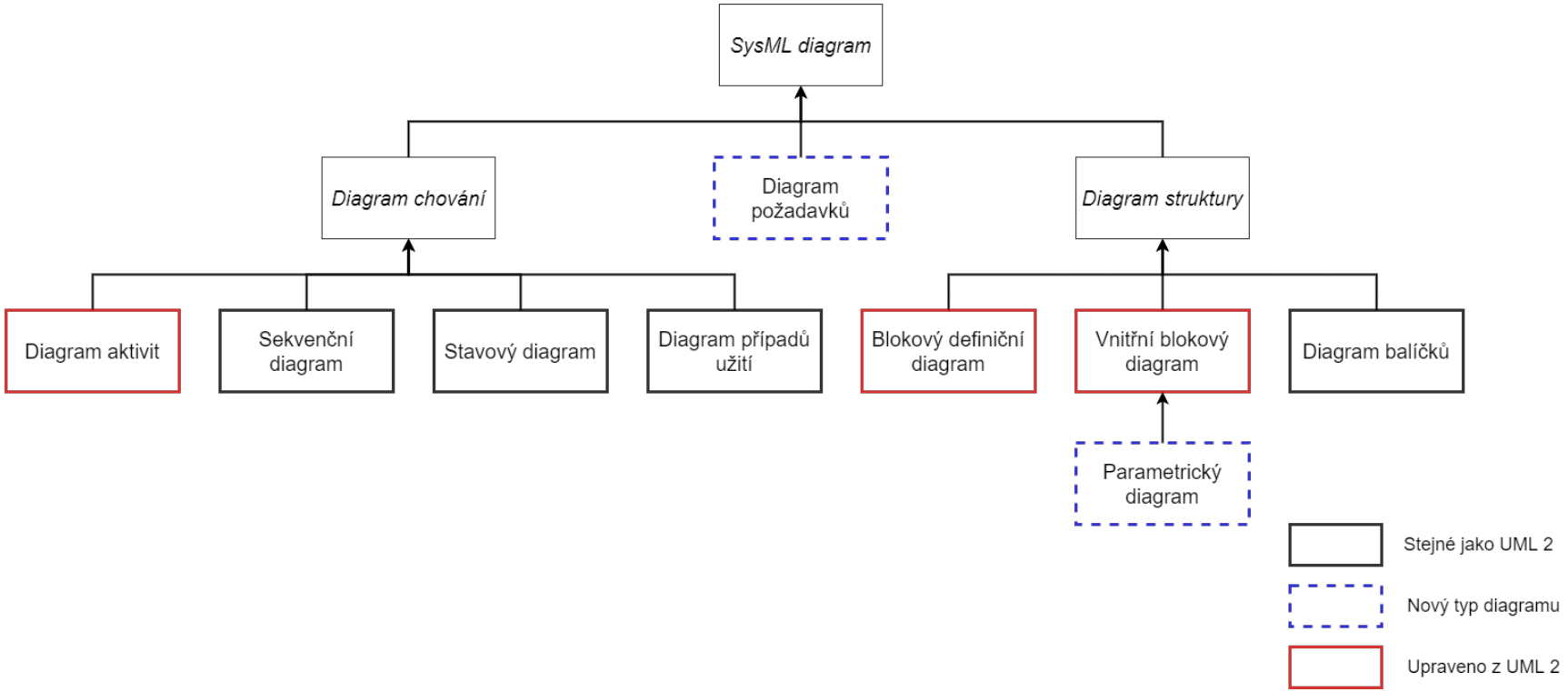
Struktura systému je reprezentována blokovým definičním diagramem a vnitřním blokovým diagramem. Blokový definiční diagram popisuje hierarchii a klasifikaci systému nebo jeho částí. Vnitřní blokový diagram popisuje vnitřní strukturu systému z hlediska jeho částí, portů a konektorů. Diagram balíčků je používán k popisu uspořádání tohoto modelu. Všechny tyto diagramy patří pod diagramy struktury.

Parametrický diagram reprezentuje omezení hodnot vlastností systému, jako je například výkon, spolehlivost. Navíc slouží jako prostředek ke spojení specifikačních a návrhových modelů s modely technické analýzy.

Diagram případů užití poskytuje popis funkcionality na vysoké úrovni, kterého je dosaženo interakcí mezi systémy nebo jeho částmi. Diagram aktivit slouží k popisu toku a řízení dat mezi aktivitami. Sekvenční diagram představuje interakci mezi spolupracujícími částmi systému. Stavový diagram popisuje stavové přechody a akce, které systém nebo jeho části provádí v reakci na nějaké události. Tyto diagramy spadají pod diagramy chování.

Součástí SysML je i grafický konstrukt, který představuje systémové požadavky a spojuje je s dalšími prvky modelu, v podobě diagramu požadavků. Tento diagram zachycuje hierarchie požadavků a spojuje typické nástroje pro správu požadavků s modely systému.

[1] app.diagrams.net



Obrázek 3.2: Přehled všech diagramů SysML, převzato z [24].

Kapitola 4

Přehled použitých technologií

V této kapitole jsou popsány technologie a termíny, které jsou potřebné k porozumění této práce.

4.1 Internet věcí (IoT)

Internet věcí, anglicky Internet of Things (IoT), je označení pro síť fyzických zařízeních, telefonů, domácích spotřebičů, vozidel a dalších zařízení, která jsou vybavena elektronikou, softwarem, senzory, pohyblivými částmi a síťovou konektivitou, která umožňuje těmto zařízením se propojit a vyměňovat si data. Každé z těchto zařízení musí být jasně identifikovatelné díky implementovanému výpočetnímu systému a navíc je schopno pracovat samostatně v existující internetové infrastruktuře. [29]

Ekosystém IoT se skládá z chytrých zařízení s připojením na internet. Tato zařízení používají vestavěné systémy, jako jsou procesory, senzory a komunikační hardware, ke shromažďování, odesílání a práci s údaji, které získávají od svého okolí. Zařízení v IoT sdílejí data ze senzorů, která sbírají tím, že jsou připojena na výchozí bránu IoT nebo na jiná okrajová zařízení, kde jsou data vyhodnocována. Data mohou být vyhodnocována i lokálně na těchto zařízeních. IoT zařízení mohou komunikovat i na základě dalších souvisejících zařízeních a jednat podle toho, jaké informace od sebe získávají. Většinu práce dělají samy bez lidského zásahu, ale pokud chceme, můžeme je například nastavit, dát jim nějaké instrukce nebo se podívat na jejich data. [20]

4.2 Vestavěný systém

Vestavěný systém je jednoúčelový počítač, ve kterém je řídicí systém zcela zabudován do zařízení, které ovládá. Na rozdíl od osobních počítačů, jsou vestavěné systémy navrženy pro konkrétní, předem definované činnosti. Vzhledem k tomu, že operační systém tohoto počítače je určen pro konkrétní účel, se může zjednodušit a optimalizovat hlavní aplikace a díky tomu pak bývají tyto systémy levnější. [34]

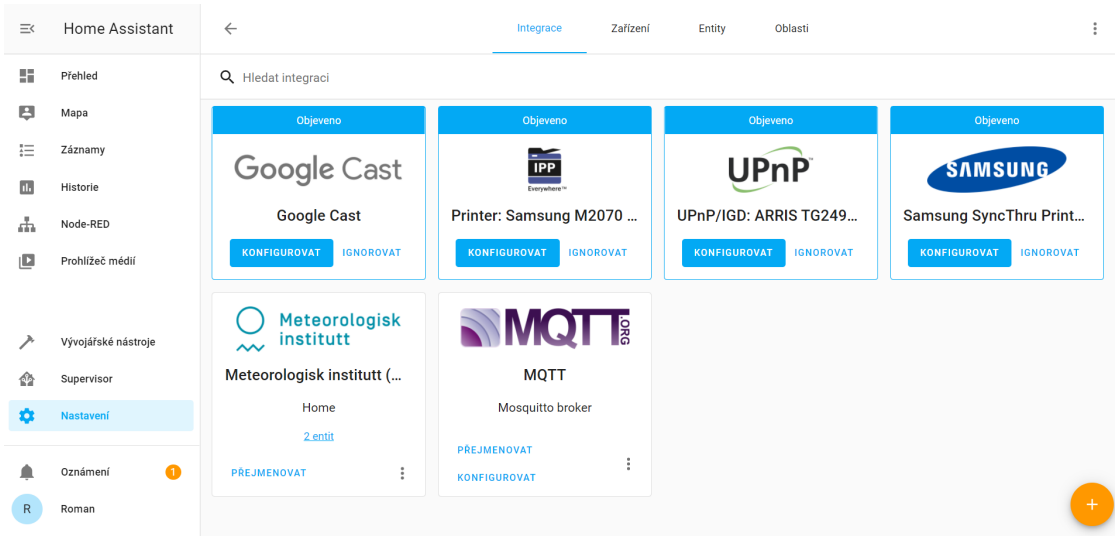
Vestavěný systém je například bankomat, kalkulačka, herní konzole, routery, modemy atd.

4.3 Home Assistant

Home Assistant je bezplatný open-source[1] systém domácí automatizace, který je zaměřen na soukromí uživatele. Tento systém je napsán v jazyce Python a lze ho nainstalovat na širokou škálu zařízení, mezi které patří například systémy Linux, datové uložiště na síti (Network Attached Storage – NAS) nebo dokonce i Raspberry Pi. [3] [11]

Podporuje integraci s více než 1700 různými komponentami od platforem jako jsou Philips Hue, ESPHome, Google Assistant, Google Cast, IKEA Smart Home, Nest, Sonos, MQTT a další zařízení, služby a ekosystémy IoT. Tyto patří pouze do základních, mnoho dalších je součástí Home Assistant Community Store. [7]

Potom, co nainstalujeme a zapneme systém Home Assistant, se můžeme přes internetový prohlížeč dostat na jeho uživatelské rozhraní, které je zobrazeno na obrázku 4.1. Najdeme ho na internetové adrese http://homeassistant.local:8123. Home Assistant sám ihned vyhledá dostupné integrace, které jsou připojeny na stejnou lokální síť, jak můžeme vidět na obrázku 4.1. Tyto integrace následně můžeme konfigurovat a automatizovat pomocí dostupných nástrojů, jako je například Node-RED, viz sekce 5.3.



Obrázek 4.1: Uživatelské rozhraní Home Assistant.

4.4 MQTT protokol

Pokud není uvedeno jinak, tato sekce vychází ze zdroje [35]. Message Queue Telemetry Transport protokol (MQTT) je klient-server komunikační protokol, který slouží pro zasílání zpráv. Je jednoduchý a navržený tak, aby byl snadno implementovatelný. Díky tomu je ideální pro komunikaci IoT zařízení, kde je vyžadována malá „stopa“ kódu (kód odeslaný společně s daty) a/nebo omezená šířka datového pásma. Protokol MQTT běží přes TCP/IP, což je základní síťový protokol internetu. Používá vzor „publish/subscribe“, který poskytuje rámec pro výměnu zpráv mezi *subscribers* (odběrateli) a *publishers* (vydavateli). Zprávy se rozlišují na základě témat neboli topiků. Subscribers se příhlásí k odběru zpráv

[1] Open-source je označení programů, jejichž zdrojový kód byl poskytnut dalším vývojářům, kteří jej mohou studovat a většinou i upravovat a dále vylepšovat.

od konkrétního publishera a tento publisher následně posílá zprávy pouze jeho subscribers. MQTT podporuje také SSL/TLS, což je protokol pro zabezpečení nad TCP/IP, tak aby byla zajištěna šifrovaná a zabezpečená veškerá datová komunikace. [17] [13]

4.4.1 MQTT broker

Protokol MQTT definuje dva typy entit v síti: message broker (zprostředkovatel zpráv) a počet klientů. Broker je server, který přijímá všechny zprávy od klientů a poté tyto zprávy směruje k příslušným cílovým klientům. Klient je cokoliv, co může komunikovat s MQTT brokerem za účelem odesílání a přijímání zpráv, může to být například IoT senzor nebo aplikace v datovém centru, která zpracovává IoT data.

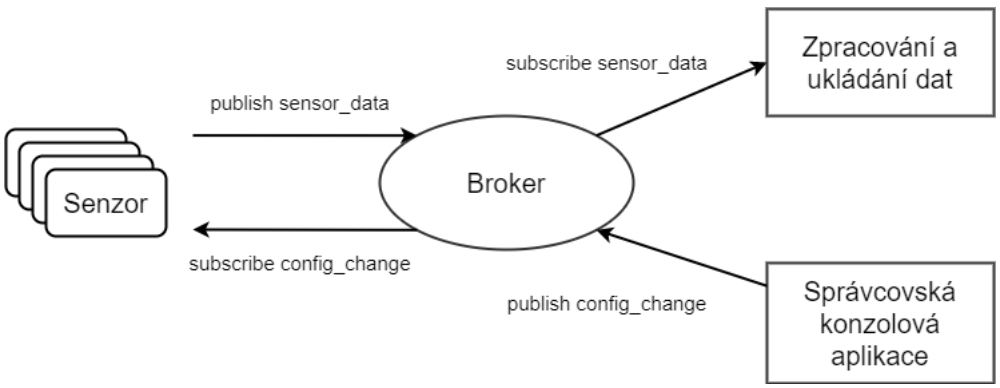
Funkce MQTT brokeru na základě vzoru publish/subscribe:

1. Klient se připojí na broker. Může se přihlásit k odběru zpráv jakéhokoliv topiku.
2. Klient publikuje zprávy konkrétního topiku tak, že posílá tyto zprávy na broker.
3. Broker následně předá tyto zprávy všem klientům, kteří se k danému topiku zprávy přihlásili.

Jak je již bylo řečeno, zprávy MQTT jsou organizovány podle témat (topiků). Vzhledem k tomu má vývojář aplikace možnost určit, s jakými konkrétními zprávami můžou klienti komunikovat.

4.4.2 Příklad MQTT komunikace

Na obrázku 4.2 můžeme vidět příklad MQTT komunikace podle vzoru publish/subscribe. Senzory zveřejňují své naměřené hodnoty v topiku zprávy `sensor_data` a jsou přihlášeni k odběru zpráv topiku `config_change`. Aplikace pro zpracování dat, které ukládají data senzorů do databáze, se přihlásí k odběru topiku `sensor_data`. Správcovská konzolová aplikace může publikovat systémové příkazy správce pro úpravu konfigurace senzorů (citlivost, frekvence odesílání vzorků atd.) v topiku `config_change`.

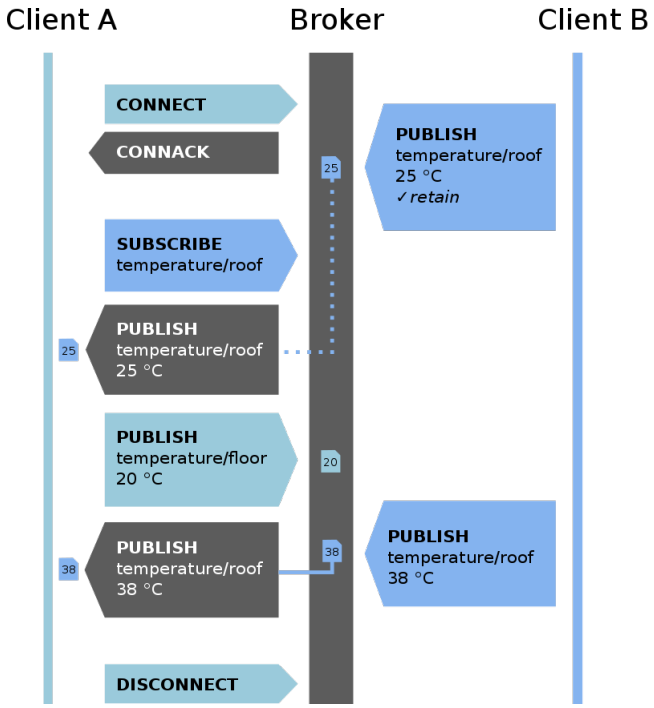


Obrázek 4.2: Příklad komunikace MQTT pro IoT senzory, převzato z [35].

4.4.3 Typy zpráv MQTT

MQTT má jednoduchou hlavičku k určení typu a topiku zprávy a konkrétní data zprávy. Typy zpráv jsou následující:

- Klient se nejprve musí připojit k brokeru odesláním zprávy **CONNECT**. Tato zpráva požádá o navázání spojení od klienta k brokeru.
- Klient následně obdrží od brokera zprávu **CONNACK**, která reprezentuje potvrzení navázání spojení.
- Po navázání spojení může klient poslat brokerovi jednu nebo více zpráv **SUBSCRIBE**, aby označil jaká témata zpráv chce přijímat.
- Poté, co se klient úspěšně přihlásil k odběru potřebných témat, broker vrátí zprávu **SUBACK** s potvrzením všech témat, která chtěl klient odebírat.
- Zprávou **UNSUBSCRIBE** se klient může odhlásit z odběru jednoho nebo více témat.
- Pokud klient chce publikovat zprávy určitého tématu, pošle brokerovi zprávu **PUBLISH**. Ten poté předá zprávu všem klientům, kteří se k danému tématu přihlásili.



Obrázek 4.3: Příklad komunikace MQTT s konkrétními typy zpráv, převzato z [30]

4.4.4 Formát dat zpráv MQTT

Je dobré, aby konkrétní data zprávy měly nějaký formát, který specifikuje klient, který zprávu s daty odesílá. Tento formát může být například JSON[2] nebo XML[3]. Důležité je, aby cíloví klienti byli schopní tento formát přečíst.

[2]JavaScript Object Notation (JSON) je způsob zápisu dat nezávislý na počítačové platformě. Je určený pro přenos dat.

[3]Extensible Markup Language (XML) je obecný značkovací jazyk. Je to standardní formát pro výměnu informací.

4.5 Petriho sítě

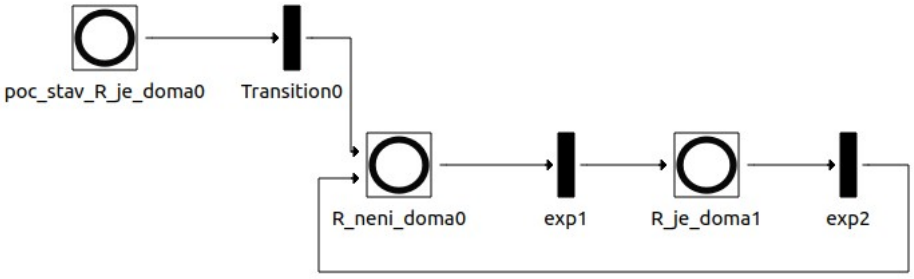
Tato sekce vychází ze zdroje [31].

Petriho síť je forma grafového popisu diskrétního systému. Popisuje stavy a přechody mezi stavy, kterým se říká události. Událost popisuje změnu stavu diskrétního systému a z hlediska doby trvání je to atomická operace. To znamená, že proběhne celá v jednom okamžiku.

Petriho síť má dva druhy uzlů, které jsou označovány jako místa a přechody, a orientované hrany, které spojují tyto uzly (nesmí to být uzly stejného typu).

Na obrázku 4.4 můžeme vidět jednoduchou Petriho síť, která modeluje přítomnost osoby v domě. Uzly, které mají kulatý tvar, jsou místa a uzly, které mají tvar obdélníkový tvar, jsou přechody.

Místa mohou obsahovat tzv. tokeny, které reprezentují procesy, a ty se přemísťují do míst jiných podle definovaných pravidel hran.



Obrázek 4.4: Jednoduchý příklad petriho sítě, vytvořeno v PowerDEVS, viz 5.4.

Kapitola 5

Programovací nástroje založené na bázi modelu

V této kapitole si představíme programovací nástroje na bázi modelu. Node-RED a PowerDEVS jsou popsány trochu více, protože je v nich vytvořena implementační část této práce.

5.1 OpenModelica

Tato sekce vychází ze zdroje [6].

OpenModelica je open-source prostředí pro modelování a simulaci. Podporuje objektově orientované modelování a simulace založené na rovnicích popsaných pomocí modelovacího jazyka Modelica, viz 3.1. Tento program se snaží o poskytnutí úplné implementace jazyka Modelica, včetně modelování a simulace modelů založených na rovnicích, optimalizace systému a dalších možností programovacího/modelovacího prostředí. Uživatelům nabízí interaktivní[1] výpočetní prostředí.

Ukazuje se, že s podporou vhodných nástrojů a knihoven se Modelica velmi dobře hodí jako výpočetní jazyk pro vývoj a provádění numerických algoritmů, například pro návrh řídicího systému a pro řešení systémů nelineárních rovnic.

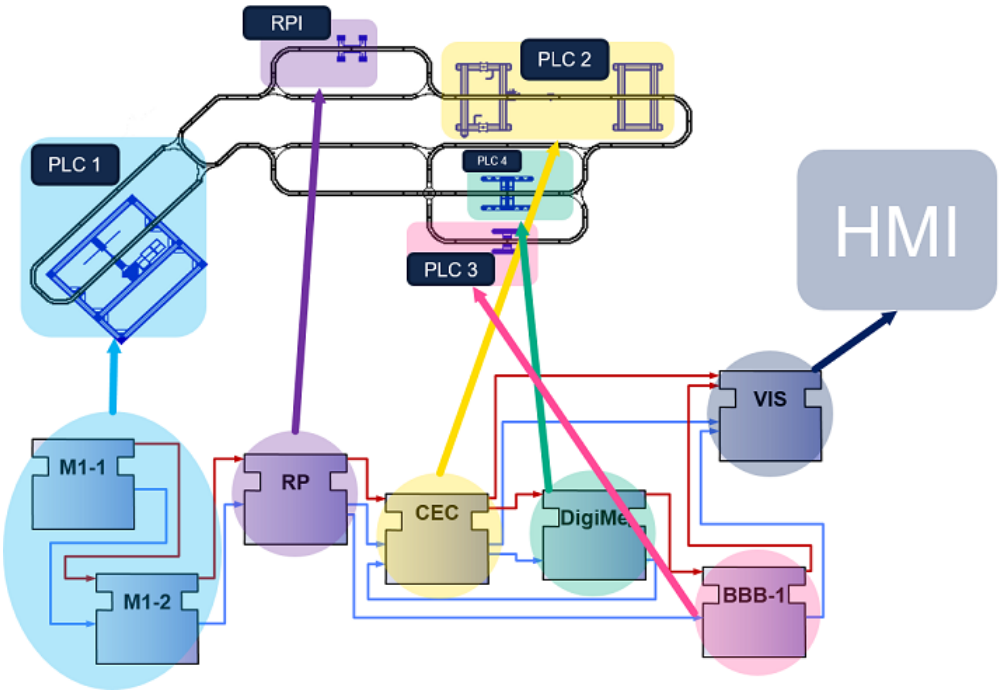
5.2 4diac

4diac je open-source vývojové prostředí určené pro distribuované průmyslové systémy. Je také použitelný v jiných oblastech, jako je domácí automatizace, elektrické sítě nebo kdekoliv, kde je potřeba nějaký automatizovaný systém. Ve 4diac se aplikace programují pomocí funkčních bloků (dále FB), které jsou nějak pospojovány. [4]

Příklad programu ve 4diac je na obrázku 5.1. Každý FB reprezentuje nějakou funkci, která se provede, když přijde událost (červené šipky směřující do FB). Příchozí modré šipky jsou datové vstupy (parametry funkce) a odchozí modré šipky jsou parametry odesílané jiným FB. Aplikace navíc nemusí být určena pouze pro jedno zařízení, ale pro celý systém, který se skládá z několika zařízení. V takovém případě některé FB běží na jednom zařízení

[1] Prostředí je interaktivní a „inkrementální“ pokud poskytuje rychlou zpětnou vazbu, například bez přepočítávání všeho od nuly, a udržuje si dialog s uživatelem, včetně zachování stavu předchozích interakcí s uživatelem. Interaktivní prostředí je obvykle produktivnější a zábavnější než neinteraktivní.

a některé FB běží na druhém – vytvoří se tak distribuovaný systém. Zařízení mohou být například Raspberry Pi (RPI), stolní počítač nebo PLC[2], jak lze vidět na obrázku 5.1. [4]



Obrázek 5.1: Příklad programu distribuovaného systému ve 4diac, převzato z [4].

5.3 Node-RED

Tato sekce vychází z webových stránek Node-RED [16], pokud není uvedeno jinak.

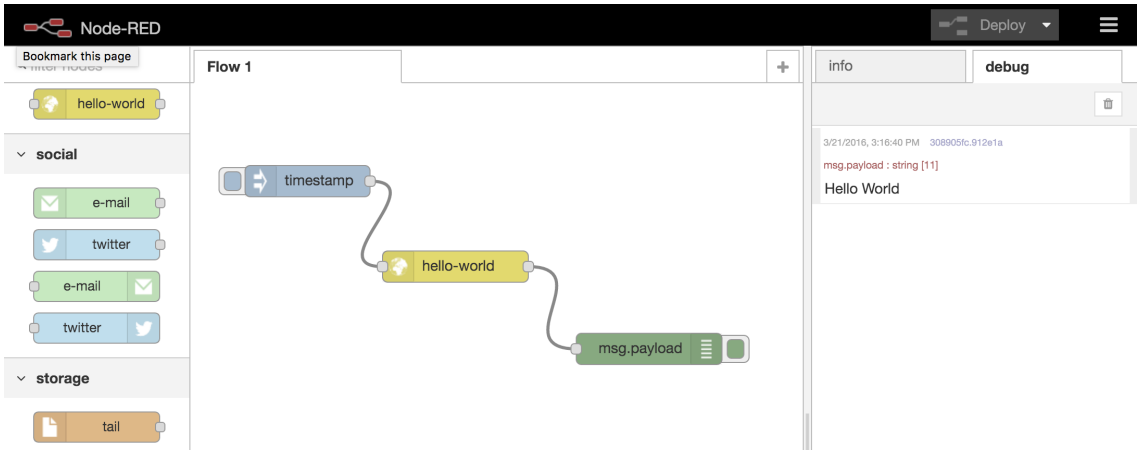
Node-RED je „flow-based“ open-source programovací nástroj, který původně vyvinul tým Emerging Technology Services společnosti IBM a nyní je součástí JS Foundation. „Flow-based“ programování je způsob, jak chování aplikace popsat jako síť blackboxů neboli „uzlů“ (anglicky node), jak se jim říká v Node-RED. Každý uzel má přesně definovaný účel: uzlu jsou dána nějaká data, ten s nimi něco udělá a potom je někomu předá. Jedná se o model, který je snadno graficky zobrazitelný a je tak přístupný širšímu okruhu uživatelů. Pokud někdo dokáže rozdělit problém na jednotlivé kroky, může se na *flow* podívat a získat představu o tom, co dělá, aniž by musel rozumět jednotlivým řádkům kódu každého uzlu.

U Node-RED je zajímavé nejen to, že poskytuje hodně použitelných zdrojů a všechny možné typy uzlů, ale také to, že uživatelským prostředím je internetový prohlížeč. Ten poskytuje nástroje pro spojování uzlů i pro ladění celého procesu. Doménou Node-RED je čtení a ovládání IoT zařízení. Charakter uzlů definuje prostředí pro skládání jednoduchých algoritmů pro ovládání domácí automatizace. [19]

Tento program lze nainstalovat například na jakýkoliv počítač s operačním systémem Linux, Raspberry Pi nebo jako doplněk systému HomeAssistant. Po nainstalování je Node-RED k dispozici na konkrétní internetové adrese.

[2]Programovací logický automat (anglicky programmable logic controller) je malý průmyslový počítač používaný pro automatizaci procesů v reálním čase – řízení strojů nebo výrobních linek v továrně. [33]

Na obrázku 5.2 je jednoduchý program v Node-RED, který vytiskne řetězec „Hello World“.



Obrázek 5.2: Jednoduchý program v Node-RED, převzato z [16].

5.3.1 Koncepty Node-RED

Pro vysvětlení, jak je program koncipován a jak funguje, si popíšeme některé pojmy, které se v Node-RED používají.

Node (uzel)

Uzel je základní blok pro vytváření flow. Uzly se spouštějí buď přijetím zprávy z předchozího uzlu v toku nebo čekáním na nějakou externí událost (například příchozí zpráva z MQTT brokeru, časovač). Tuto zprávu nebo událost zpracují a následně ji odešlou dalším uzlům ve flow.

Uzel může mít maximálně jeden vstupní port. Počet výstupních portů naopak není omezen.

Configuration node (konfigurační uzel)

Konfigurační uzel je speciální typ uzlu, který obsahuje opakovaně použitelnou konfiguraci. Tuto konfiguraci mohou sdílet běžné uzly v toku.

Například uzly *MQTT In* a *MQTT Out* používají společný konfigurační uzel k reprezentaci sdíleného připojení na MQTT broker.

Flow (tok)

Flow je reprezentováno jako karta v pracovním prostoru Node-RED editoru a je to hlavní způsob, jak organizovat uzly.

Termín „flow“ se také používá k neformálnímu popisu jedné sady propojených uzlů. Flow (karta) může tedy obsahovat více flows (sad propojených uzlů).

Message (zpráva)

Zprávy jsou to, co prochází mezi uzly v toku. Jsou to obyčejné objekty JavaScriptu[3], které mohou mít libovolnou sadu vlastností. V editoru se často označují jako `msg`.

Zprávy mají jednu nebo více vlastností. Podle konvence vlastnost `payload` obsahuje nejužitečnější informace. Další vlastnost, která stojí za zmínku je vlastnost `topic`. Jak jsme si řekli, uzly mohou mít pouze jeden vstupní port a díky této vlastnosti můžeme například rozlišit zprávy na vstupu uzlů.

Subflow

Subflow je kolekce uzlů, které jsou zaobaleny do jednoho uzlu.

Lze je použít ke zjednodušení určité vizuální složitosti toku nebo k zabalení skupiny uzlů, které jsou opakovaně používány jako komponenty.

Wires (dráty)

Dráty spojují uzly a představují to, jakým způsobem zprávy procházejí flow.

5.4 PowerDEVS

Tato sekce vychází ze zdrojů [10, 1].

PowerDevs je multiplatformní open-source nástroj pro modelování a simulaci založený na formalismu Discrete Event System Specification, česky Specifikace systému diskrétních událostí (dále DEVS). Přestože se jedná o univerzální simulační nástroj DEVS, PowerDEVS je vhodný i pro modelování a simulaci spojitých a hybridních systémů pomocí algoritmů Quantized State System (QSS). Tento nástroj i jeho modely jsou naprogramované v jazyce C++.

5.4.1 DEVS

DEVS je formalismus, který poprvé představil Bernard P. Zeigler. Model DEVS zpracovává trajektorii vstupní události a podle této trajektorie a vlastních počátečních podmínek určuje trajektorii výstupní události.

Jedná se o nejobecnější formalismus pro modelování systému diskrétních událostí. Umožňuje reprezentaci libovolného systému za předpokladu, že provádí konečný počet změn systému v konečných časových intervalech. Za příklady DEVS lze tedy nejen považovat například Petriho sítě nebo stavové diagramy, ale také všechny diskrétní časové systémy.

Vzhledem k tomu, že běžné diferenciální rovnice lze aproximovat diskrétními časovými systémy za použití numerických integračních metod, DEVS dokáže modelovat i spojité systémy. Kromě toho existují numerické metody, např QSS, které vytváří simulační modely, které nemohou být reprezentovány v diskrétním čase, ale pouze jako modely DEVS.

Definice

Atomický model DEVS je definován jako sedmice tvaru: $M = (X, Y, S, \delta_{int}, \delta_{ext}, \lambda, ta)$, kde:

- X je množina všech možných hodnot vstupní události, neboli množina všech možných hodnot, kterých může vstupní událost nabýt,

[3] JavaScript je multiplatformní, objektově orientovaný, událostmi řízený skriptovací jazyk.

- Y je množina všech možných hodnot výstupní události,
- S je množina všech stavových proměnných a
- δ_{int}, δ_{ext}, λ a ta jsou funkce, které definují dynamiku systému.

Každý možný stav s ($s \in S$) má přidělený *časový posun*, neboli *Time Advance*, vypočtený *funkcí časového posunu* $ta(s)(ta(s) : S \to \Re_0^+)$. *Časový posun* je nezáporné reálné číslo udávající, jak dlouho systém přetrvává v daném stavu při absenci vstupních proměnných.

Pokud tedy stav přijme hodnotu s_1 v čase t_1, po $ta(s_1)$ jednotkách času (tj. v čase $ta(s_1) + t_1$) systém provede *interní přechod*, neboli *Internal Transition*, do nového stavu s_2. Nový stav je vypočítán jako $s_2 = \delta_{int}(s_1)$. Funkce $\delta_{int}(\delta_{int} : S \to S)$ se nazývá *funkce interního přechodu*.

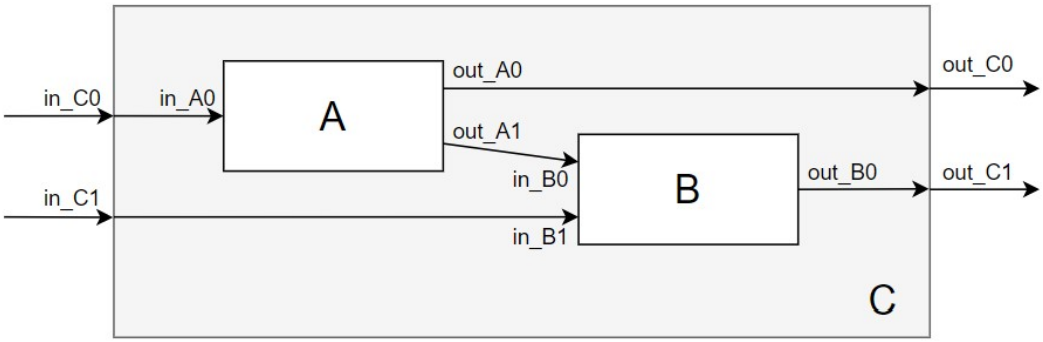
Když systém přejde ze stavu s_1 do stavu s_2 je vytvořena *výstupní událost*, neboli *Output Event*, s hodnotou $y_1 = \lambda(s_1)$. Funkce $\lambda(\lambda : S \to Y)$ se nazývá *výstupní funkce*. Tím způsobem funkce ta, δ_{int} a λ definují autonomní chování modelu DEVS.

Když přijde vstupní událost, stav se okamžitě změní. Nová stavová proměnná nezáleží pouze na vstupní proměnné, ale také na předchozí stavové proměnné a uplynulém čase od posledního přechodu. Pokud systém dorazí do stavu s_2 v čase t_2 a vstupní událost dorazí v čase $t_2 + e$ s hodnotou x_1, nový stav je vypočten jako $s_3 = \delta_{ext}(s_2, e, x_1)$ (všimněte si, že $ta(s_2) > e$). V tomto případě se jedná o *externí přechod*, neboli *External Eransition*. Funkce $\delta_{ext}(\delta_{ext} : S \times \Re_0^+ \times X \to S)$ se nazývá *funkce externího přechodu*. Během externího přechodu není vytvořena žádná výstupní událost.

Spojování modelů DEVS

Atomické modely DEVS mohou být spojovány. Teorie DEVS zaručuje, že spojování atomických modelů DEVS definuje nové modely DEVS. Tím pádem mohou být složité komplexní systémy reprezentovány hierarchicky.

Spojení v DEVS je obvykle reprezentováno použitím vstupních a výstupních portů. Díky těmto portům se propojení modelů DEVS stává jednoduchým prvkem blokového schéma. Obrázek 5.3 ukazuje model C, který vznikl spojením modelů A a B. Porty jsou na tomto obrázku označeny jako *in* (vstupní) a *out* (výstupní). Model C může být sám „uzavřen“ a díky tomu pak může být používán jako samostatný atomický DEVS.



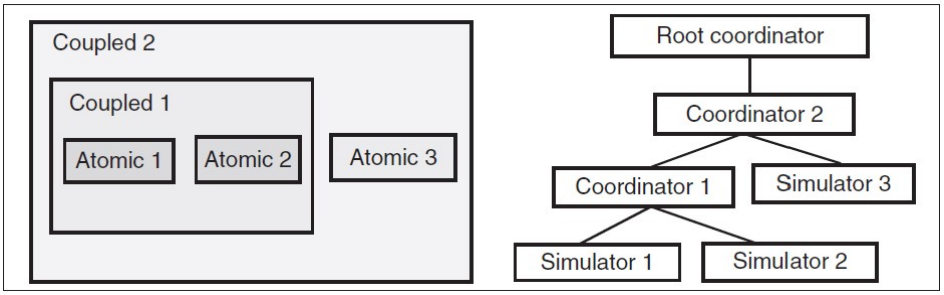
Obrázek 5.3: Příklad spojeného modelu DEVS. Převzato z [1] a upraveno.

5.4.2 Simulace modelu DEVS

Jedna z nejdůležitějších vlastností DEVS je schopnost snadno a efektivně simulovat velmi složité modely. Základní myšlenku simulace spojeného modelu DEVS lze popsat pomocí následujících kroků:

1. Nechť je atomický model, který je podle jeho časového posunu a uplynulého času dalším, který provede interní přechod, označen d^* a čas zmíněného přechodu tn.
2. Posuňte simulační čas t na $t = tn$ a proveďte interní přechodovou funkci d^*.
3. Předejte výstupní událost vytvořenou d^* na všechny atomické modely, které jsou k němu připojeny a které provedou odpovídající externí přechodové funkce. Poté se vraťte ke kroku 1.

Jeden z nejjednodušších způsobů implementace těchto kroků je napsat program, který má hierarchickou strukturu ekvivalentní hierarchické struktuře simulovaného modelu. Toto je metoda, kterou vymyslel B. P. Zeigler. Ke každému *atomickému* modelu DEVS je přidělena procedura zvaná *DEVS-simulator* a ke každému *spojenému* modelu DEVS se vztahuje jiná procedura zvaná *DEVS-coordinator* (koordinátor). Na vrcholu je procedura s názvem *DEVS-root-coordinator*, neboli kořenový koordinátor, která řídí globální čas simulace. Tuto myšlenku můžeme vidět zobrazenou na obrázku 5.4.



Obrázek 5.4: Příklad hierarchického modelu a k němu odpovídající simulační schéma. Převzato z [1].

Simulátory a koordinátoři sousedních vrstev spolu komunikují pomocí zpráv. Koordinátoři posílají zprávy svým potomkům, aby prováděli přechodové funkce. Když simulátor provede přechod, vypočítá si svůj další stav, a když je přechod interní, odešle výstupní hodnotu svému otcovskému koordinátorovi. V každém případě se stav simulátoru bude shodovat s přidruženým stavem atomického modelu DEVS. Když koordinátor provede přechod, odešle zprávy některým svým potomkům, aby provedly své odpovídající přechodové funkce. Pokud se výstupní událost vytvořená jedním z jeho potomků musí dostat ven ze spojeného modelu, koordinátor odešle zprávu svému vlastnímu otcovskému koordinátorovi, který nese výstupní hodnotu.

Každý simulátor a koordinátor má lokální proměnnou tn, která označuje čas, kdy dojde k jeho dalšímu internímu přechodu. U simulátorů se tato proměnná počítá pomocí funkce časového posunu odpovídajícího atomického modelu. U koordinátorů se určí podle minimální tn všech jejich potomků. Z toho plyne, že tn koordinátoru na vrcholu simulačního schéma je čas, ve kterém dojde k další události celého systému. Kořenový koordinátor se tedy vždy podívá na tento čas tn, posune globální čas t na tuto hodnotu a poté pošle zprávu svému potomku, aby provedl další přechod. Tento cyklus opakuje, dokud simulace neskončí.

5.4.3 Komponenty

V této sekci jsou popsány základní komponenty a funkce PowerDEVS.

Modely se zde tvoří vzájemným spojováním atomických bloků a mohou být vytvořeny jako hierarchické blokové diagramy pomocí bloku `Coupled`, který atomické bloky shromažďuje a vytvoří tak spojený model. Lze použít předdefinované atomické bloky z existujících knihoven nebo mohou být vytvořeny vlastní. Knihovny PowerDEVS obsahují bloky pro modelování spojitých, diskrétních i hybridních časových systémů. Pro modelování klasických diskrétních systémů je k dispozici například i knihovna pro Petriho sítě, viz sekce 4.5. [10]

PowerDEVS navíc běží pod operačním systémem reálného času (Real Time Application Interface, RTAI14), který poskytuje možnost synchronizovat události průběžně (v reálném čase) se schopností zachytit přerušení na atomické úrovni.

Tento software byl koncipován pro použití odbornými programátory DEVS a stejně tak koncovými uživateli, kteří chtějí pouze propojit předem definované bloky a vytvořit tak model pro simulaci.

Skládá se z více na sobě nezávislých programů, které jsou popsány níže. Jak tyto programy vypadají naleznete v příloze A.

Model editor

Model editor je z uživatelského hlediska hlavním programem PowerDEVS. Obsahuje grafické rozhraní umožňující hierarchickou konstrukci blokových diagramů, správu knihoven, výběr parametrů a další definice na vysoké úrovni. Poskytuje také propojení s ostatními programy, takže například atomic editor nebo preprocesor jsou spouštěny odsud.

Hlavní okno model editoru umožňuje uživateli vytvářet a otevírat modely a knihovny. Uživatel je tedy schopen z knihoven „vytáhnout“ požadované bloky do modelů a podle potřeby propojit. Okna modelů (Model Windows) poskytují všechny klasické možnosti grafické editace, například kopírování bloků, změna jejich velikosti, otáčení atd. Propojení mezi bloky je možno udělat přímo mezi různými porty.

Každý blok (po stisknutí pravého tlačítka myši) lze upravit, nastavit jeho parametry nebo upravit jeho kód. Na obrázku A.2 je okno *Edit*, které umožňuje upravit název, popis, počet vstupních a výstupních portů a grafické vlastnosti bloku (záložka *Properties*). V záložce *Parameters* si uživatel může upravit parametry a záložka *Code* umožňuje přiřadit k bloku soubor, který obsahuje kód s definicemi modelu DEVS. Pokud uživatel chce kód upravit (tlačítko *Edit Code*) nebo vytvořit nový soubor (tlačítko *New File*), otevře se mu atomic editor.

Okno *Parameters*, slouží pro nastavení parametrů bloku, což je zobrazeno na obrázku A.3. Toto okno se otevře i při „poklepání“ na určitý blok. U předdefinovaných bloků samozřejmě můžeme změnit hodnoty parametrů bez jejich úpravy.

Atomic editor

Atomic editor umožňuje úpravy atomických modelů elementárních bloků. Definuje chování bloků pomocí přechodových funkcí, výstupní funkce, časového posunu atd., viz sekce 5.4.1, v jazyce C++. Hlavní okno atomic editoru je znázorněno na obrázku A.4.

Při vytvoření nového bloku si uživatel musí pomocí atomic editoru definovat proměnné, které tvoří stav a výstup modelu DEVS, a proměnné, které představují parametry systému. Poté musí napsat kód funkce časového posunu, přechodových funkcí a výstupní funkce do příslušných záložek. Jsou tu ještě dvě další záložky (*Init* a *Exit*), kde uživatel může přidat

část kódu, která se provede před spuštěním simulace (například pro nastavení počátečních stavů a parametrů) a část kódu, která se provede na konci simulace (například zavření některých otevřených souborů).

Uživatel může k tomuto souboru zahrnout třeba i některé další hlavičkové soubory či externí knihovny, které nejsou součástí PowerDEVS, v nabídce *Tools*, *Configure*. Po uložení modelu se kód automaticky dokončí a uloží se do příslušných souborů .cpp a .h, které jsou následně použity preprocesorem pro generování simulace celého systému.

Preprocesor

Preprocesor si vezme soubor .pdm (nebo .pds), který vytvořil model editor, a vytvoří simulační program. V zásadě překládá soubor .pdm do hlavičkového souboru .h (nazvaného `model.h`), který spojuje simulátory a koordinátory podle vazebné struktury. Dále také předává parametry bloků.

Následně vytváří soubor makefile (`Makefile.include`), který je použit při kompulaci a generování programu, který implementuje simulaci (tento program se nazývá *model*).

Preprocesor lze spustit z model editoru pomocí tlačítka *Simulate* a také z příkazového řádku, protože je to samostatná aplikace.

Simulační rozhraní

Když je spuštěn vygenerovaný model programu, který vygeneroval preprocesor, spustí se simulace daného modelu DEVS. PowerDEVS poskytuje grafické rozhraní pro spuštění simulace, viz obrázek A.5. Toto rozhraní umožňuje uživateli změnit některé parametry:

- Koncový čas (Final Time): udává, jak dlouho by měl být model simulován.
- Počet simulačních běhů, které mají najednou proběhnout (Run N simulations). Toto může být užitečné při výpočtu statistik z výsledků simulace.
- Neoprávněné kontrolní zastavení (Illegitimate check break): PowerDEVS zastaví simulaci, pokud se čas neposune po zvoleném počtu kroků. Tímto se může zabránit tzv. „hang-ups“ v nelegitimních modelech.
- Simulace krok za krokem (step-by-step): simulaci lze provádět postupně provedením jednoho kroku (nebo více) najednou.
- Synchronizace času: simulaci lze spustit synchronizovanou s reálným časem (tlačítko `Run Timed`).

Program Scilab

Běžící instance programu Scilab funguje jako pracovní prostor pro definování parametrů modelu, provádění analýzy výsledků atd.

5.4.4 Knihovna

Součástí programu PowerDEVS je i sada naprogramovaných atomických a spojených modelů, které lze využít k vytváření nových modelů. Tyto základní modely tvoří knihovnu PowerDEVS a kdokoliv může tuto knihovnu rozšířit o svoje vlastní modely.

Knihovna je rozdělena do následujících kategorií:

Basic Elements

Toto je jediné jádro knihovny PowerDEVS. Na základě těchto modelů byly vyvinuty všechny ostatní knihovny. Obsahuje čtyři základní modely: atomický (kostra pro všechny atomické modely), spojený (neboli `Coupled`) a dva speciální objekty `inport` a `outport`. Poslední dva zmíněné objekty představují externí vstupní a výstupní rozhraní pro spojené modely.

Continous

Zde jsou obsaženy modely, které zpracovávají spojité signály založené na metodách QSS pro simulaci spojitých systémů. Patří sem integrátory, různé nelineární statické funkce, sčítačka, násobička atd.

Discrete

Tato knihovna obsahuje modely pro simulaci diskrétních systémů.

Hybrid

Zde je zahrnuta sada modelů kombinujících spojité a diskrétní funkce pro simulaci hybridních systémů. Jsou zde tedy například přepínače, komparátory, vzorkovače atd.

Real Time

Tato knihovna nabízí konkrétní bloky pro použití různých funkcí PowerDEVS v reálném čase.

Source

V této kategorii můžeme najít několik zdrojů spojitých signálů aproximovaných metodami QSS, jako jsou sinusové vlny, pulsy, signály obdélníkového průběhu atd. Jsou zde i generátory náhodných čísel nebo blok, který čte vstupní signál z programu Scilab.

Sink

Zde můžeme najít různé modely, kde jsou výsledky simulace odesílány pro vizualizaci nebo budoucí zpracování. Patří sem například `GnuPlot`, který vykresluje jeho vstupní signály, `ToDisk`, který zapisuje své vstupní signály do souboru CSV atd. Je zde i blok, který zapisuje přijatý signál do programu Scilab.

Vectors

Tato knihovna poskytuje práci s vektory.

Kapitola 6

Návrh řešení

Vytvoření řídicího systému, který bude řídit teplotu v domu. Teplota bude regulována na základě přítomnosti osoby a teploty v pokoji. Zda je přítomna osoba v pokoji, zjistí motion senzor HC-SR501[1]. Teplotu v pokoji bude snímat senzor s firmwarem ESPEasy. Tyto senzory budou komunikovat s řídicím systémem pomocí protokolu MQTT přes lokální síť.

Tento řídicí systém (kontrolér) i s modelem domu vytvořím pomocí programu PowerDEVS. Díky tomu, že senzory komunikují přes lokální síť, bude možné řídicí systém snadno zapojit do reálného domu s reálnými, opravdovými senzory. Tento přístup je zaměřen na end-to-end metodiku pro návrh hybridních kontrolérů, která je založena na kontinuitě modelu (anglicky *model continuity* [8]) pro DEVS. Kromě toho se jedná o takzvaný simulačně založený návrh (anglicky *simulation-based development* [22]), protože vyvíjíme systém pomocí simulace.

Alternativně realizuji řídicí systém i s modelem budovy v programu Node-RED. Tyto dvě realizace systému by spolu měly být schopny komunikovat a stejně tak by měly být schopny komunikovat i s reálným domem a reálnými senzory.

Hlavním cílem této bakalářské práce je poukázat na návrh, který je založený na modelech, a na princip model continuity. Nikoliv na vytvoření modelu, který přesně odráží reálné chování budovy, a tak se ani při řešení této práce nebude klást na tuto skutečnost důraz.

6.1 Princip model continuity

Hlavní myšlenka tohoto přístupu je, že simulační model DEVS pro kontrolér se může postupně „vyvinout“ z desktopového prostředí až do konečného vestavěného systému bez nutnosti mezikódování nebo opětovné implementace. Většina jiných stávajících metod jsou stále dost složité, nákladné nebo těžké na zavedení do reality, avšak techniky založené na modelech a simulacích nabízejí výhodnější funkce, které umožňují rychlé, robustní prototypování a rychlejší konečné dodání vestavěných řídicích systémů. Model continuity se stává stále více uznávanou strategií, kde je DEVS považováno za technologii vhodnou i pro kyberfyzikální systémy. [5]

Klíčové předpoklady jsou:

(a) Cílový systém je nějaká platforma, která sama může být ve vývoji a tím pádem je potencionálním zdrojem chyb.

[1] www.mpja.com/download/31227sc.pdf

(b) Teplota je řízena pomocí jednodeskového počítače (anglicky single-board computer), který je schopen běhu standardního operačního systému Linux (např. Raspberry Pi).

(c) Platforma pro vývoj modelu je běžný stolní počítač se standardním operačním systémem Linux.

(d) Je k dispozici lokální síť pro komunikaci počítače s jednodeskovým počítačem pomocí standardní technologie (Ethernet, WiFi).

Pro to, aby princip model continuity dával smysl, musíme určit, kde dojde ke spojení simulačního modelu a řídicího systému. Konkrétně v našem případě se nám nabízí senzory, které komunikují přes lokální síť. Toto spojení představuje klíčovou vrstvu rozhraní mezi softwarovou abstrakcí (model domu) a fyzickou platformou (opravdový dům).

6.2 Model domu

Před tím, než realizujeme model domu, je dobré si určit, jak bude vypadat. Dům bude mít čtyři místnosti a tyto místnosti budou využívat čtyři lidé - osoby R, F, L a T. Osoby R a F mají každý svůj vlastní pokoj a osoby L a T se dělí o jednu společnou ložnici. Poslední místnost jsou tzv. společné prostory, kde se pohybují a pobývají všechny vyjmenované osoby.

V každé této místnosti je jednoduchý senzor pohybu a teplotní senzor. Oba dva tyto senzory jsou schopny posílat v určitých časových intervalech zprávy přes protokol MQTT řídicímu systému. Součástí pokoje je také aktuátor ventilu topení. Ten dokáže přijímat MQTT zprávy, které řídicí systém posílá, a bude mít 5 úrovní intenzity topení (0 – netopí se, 5 – maximální úroveň topení). Dům má tedy funkční bezdrátovou lokální síť (WLAN), přes kterou tato komunikace bude probíhat.

Aby se mohlo topit v pokojích, musí být zapnutá kotelna, která je také součástí domu. Kotelna má 10 úrovní intenzity a měla by mít nějaký přijímač, který bude připojený na síť a bude schopný přijímat MQTT zprávy.

6.3 Řídicí systém

Na řídicí systém máme také nějaké požadavky. Jak již bylo řečeno, chceme, aby řídil teplotu v každém pokoji zvlášť, na základě toho, zda se někdo v pokoji pohybuje. Určitě nechceme aby se topení okamžitě vypínalo pokaždé, co někdo z pokoje odejde, byť jen třeba na hodinu. Proto by měl řídicí systém „počkat“ nějakou dobu a až poté topení vypnout.

Pro každý aktuátor ventilu topení bude naimplementován termostat, který určí, jak má aktuátor topit. Je tedy dobré si určit, jaká úroveň se zapne při konkrétní teplotě. Bylo by vhodné, aby se tato teplota odvíjela od nastavené teploty, kterou uživatel v pokoji chce. Na tabulce 6.1 můžeme vidět vztah intenzity topení ke konkrétním teplotám.

řídicí systém navíc reguluje i sílu topení kotelny podle toho, kde a jak moc se v domě topí.

6.4 Dohledový systém

Pro přehled nad celým systémem vytvořím i dohledový systém, který bude sloužit pro monitorování teplot v pokojích, nastavení požadované teploty atd. Kvůli tomu by měl tedy být přístupný z jakéhokoliv zařízení, které je připojeno na lokální síť, aby do tohoto systému měla přístup každá osoba, která v domě bydlí. Dále v něm bude možnost monitorování

Úroveň topení	Teplota [°C]
0	$t \geq t_{set}$
1	$t_{set} - 0.5 \leq t < t_{set}$
2	$t_{set} - 1 \leq t < t_{set} - 0.5$
3	$t_{set} - 1.5 \leq t < t_{set} - 1$
4	$t_{set} - 2 \leq t < t_{set} - 1.5$
5	$t < t_{set} - 2$

Tabulka 6.1: Závislost topné úrovně termostatu na teplotě, kde t značí aktuální teplotu v pokoji a t_{set} nastavenou teplotu.

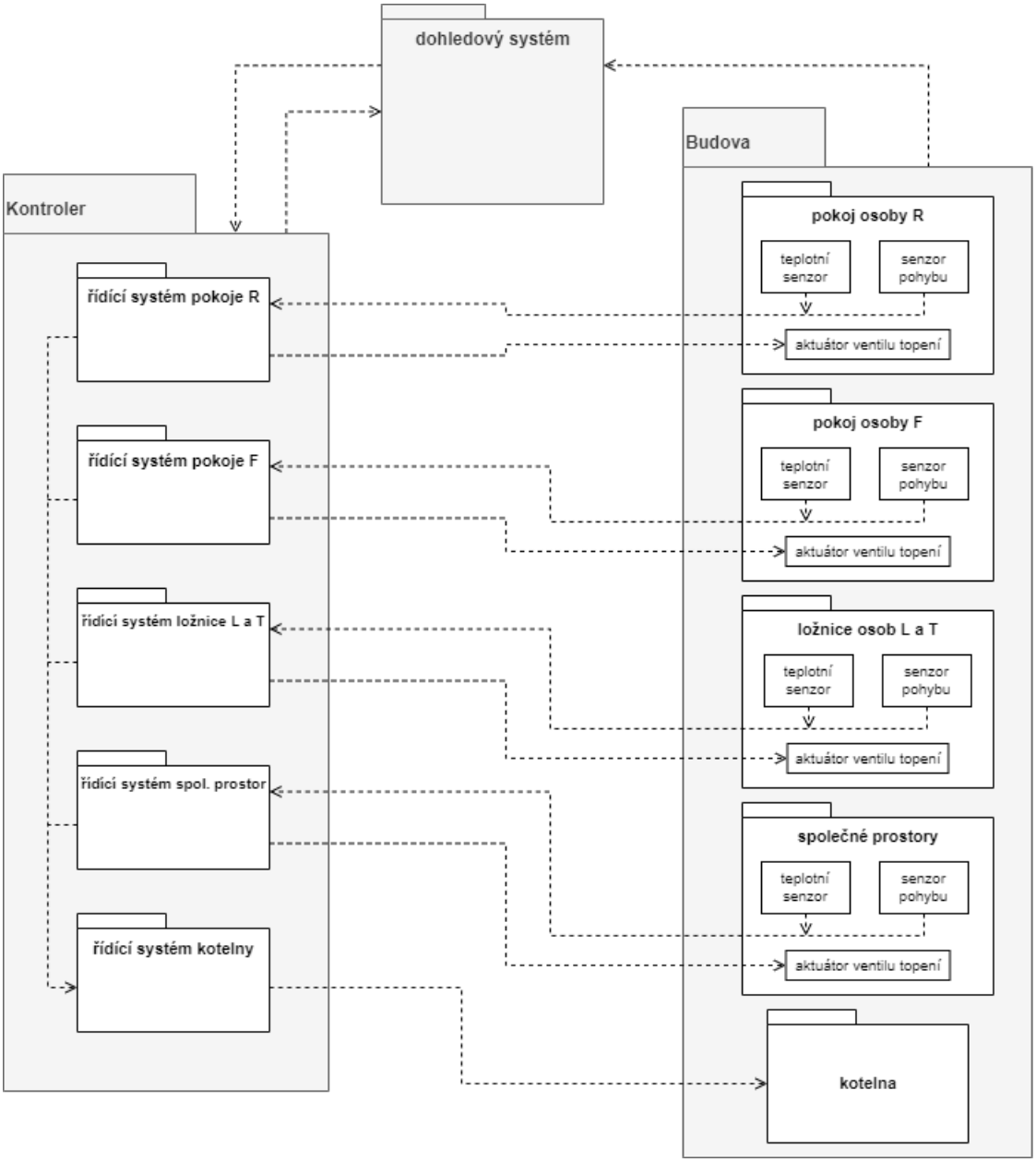
konkrétních MQTT zpráv a jejich struktur. Díky tomu můžeme například zjistit, kde se nachází chyba.

6.5 Diagram balíčků

Pro lepší pochopení, jak bude vypadat a fungovat řídicí systém a jeho spojení s domem, nám pomůže diagram balíčků.

Diagram balíčků je druh diagramu struktury, který je součástí UML i SysML. Graficky zobrazuje uspořádání a organizaci prvků modelu. Může zobrazit jak strukturu tak i závislosti mezi podsystémy nebo moduly a díky tomu můžeme vidět systém jako vícevrstvý model. Diagram balíčků se používá ke strukturování prvků systému na vysoké úrovni. Balíček je kolekce logicky souvisejících prvků UML/SysML a je zobrazen jako složka souborů. [23]

Na obrázku 6.1 můžeme vidět rozdělení celého systému na tři podsystémy – kontrolér (řídicí systém), dohledový systém a budova. Šipky reprezentují komunikaci mezi moduly. Pokud se jedná o šipky mezi mezi dvěma podsystémy (např. budovou a kontrolérem), jedná se o komunikaci pomocí MQTT zpráv.



Obrázek 6.1: Diagram balíčků celého systému.

Kapitola 7

Dokumentace implementační části

Tato část slouží jako dokumentace pro implementační část této práce (programová příloha). Jedna část řešení je naimplementována v programu PowerDEVS a druhá část je naprogramována v Node-RED. Tyto modely jsou naimplementovány tak, aby mohly vzájemně komunikovat a co nejvíce spolu korespondovaly. Jak Node-RED flows korespondují s PowerDEVS modely můžeme vidět v příloze B.

7.1 PowerDEVS

Pro implementaci řídicího systému a modelu domu byl použit PowerDEVS ve verzi 3.0 pro Linux, které je volně ke stažení[1]. Pomocí něj byly naimplementovány čtyři modely, které byly vytvořeny postupně podle principu model continuity, viz sekce 6.1. První dva modely se liší pouze v komunikaci mezi řídicím systémem a modelem domu, další dva modely představují buď jen řídicí systém nebo jen model domu.

1. `ridici_system_s_modelem_domu.pdm` – komunikace mezi moduly probíhá jen v rámci PowerDEVS
2. `ridici_system_s_modelem_domu_MQTT.pdm` – komunikace probíhá po síti pomocí MQTT zpráv
3. `ridici_system_MQTT.pdm` – zde je pouze řídicí systém, který komunikuje s domem nebo modelem domu po síti pomocí MQTT zpráv
4. `model_domu_MQTT.pdm` – zde je pouze model domu, který komunikuje s řídicím systémem po síti pomocí MQTT zpráv

Nejprve si popíšeme model domu a následně řídicí systém. Tyto moduly jsou stejné ve všech vytvořených modelech (pokud jsou přítomny). Následně je popsána komunikace mezi těmito moduly.

7.1.1 Model domu

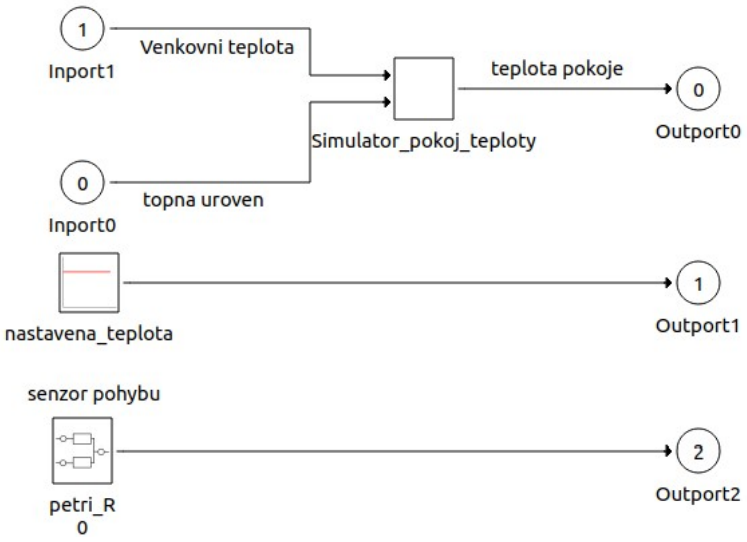
Model domu byl realizován pomocí spojeného bloku `coupled`, který má 5 vstupních a 12 výstupních portů. První 4 vstupní porty jsou výstupy řídicího systému, tedy topné úrovně všech čtyř místností. Poslední vstupní port je pro venkovní teplotu, protože je to jeden z parametrů, který ovlivňuje teplotu v místnostech domu. Model domu se skládá z dalších

[1] www.github.com/CIFASIS/power-devs

čtyř bloků `coupled`, kde každý jeden blok reprezentuje jednu místnost, jak můžeme vidět na obrázku B.1. Blok `Pokoj_R` představuje pokoj osoby R, `Pokoj_F` představuje pokoj osoby F, `spolecne_prostory` představuje společné prostory a `Pokoj_TL` představuje ložnici osob L a T.

V každé místnosti je teplotní a pohybový senzor, avšak v tomto bloku, který reprezentuje konkrétní místnost, není namodelované veškeré chování těchto senzorů. Senzory kromě toho, že snímají teplotu nebo pohyb v pokoji, také odesílají získané data na MQTT broker. Tento blok modeluje pouze snímání teploty nebo pohybu v pokoji, nikoliv odesílání dat. Odesílání dat a přijímání dat a tedy komunikace mezi řídicím systémem a modelem domu je namodelována v jiné části modelu, která je popsána v sekci 7.1.3.

Každý blok, co reprezentuje konkrétní místnost, má dvě vstupní hodnoty – topnou úroveň a venkovní teplotu. Tento blok je zobrazen na obrázku 7.1.



Obrázek 7.1: Blok modelu pokoje osoby R. Je to tedy blok `Pokoj_R` z obrázku B.1.

Můžeme zde vidět blok `nastavena_teplota`, což je konstanta, která reprezentuje teplotu, kterou si uživatel nastavil a kterou chce v pokoji mít.

Atomický blok `Simulator_pokoj_teploty` je blok, který má nadefinované vlastní chování. Simuluje (snižuje nebo zvyšuje) teplotu pokoje na základně venkovní teploty a topné úrovně, což jsou hodnoty na vstupu. Na výstupu je teplota v pokoji. Tento blok zároveň modeluje chování aktuátoru ventilu topení a senzoru teploty kromě komunikace s MQTT brokerem. Parametry tohoto bloku jsou počáteční pokojová teplota (v °C) a perioda vzorkování. Perioda vzorkování je důležitá z hlediska toho, jak rychle se bude teplota v pokoji měnit (po jak velkých časových úsecích). Jeho zdrojový kód je soubor `room_temperature_sim.cpp`[2].

Součástí modelu domu musí být také osoby, které v tomto domě pobývají. Jejich pohyb (jestli jsou nebo nejsou doma) byl naimplementován pomocí Petriho sítí, jejichž knihovnu PowerDEVS nabízí. Na obrázku 7.1 je to blok `petri_R`. Výstupem tohoto bloku je buď hodnota 1 (osoba je přítomna v místnosti) nebo hodnota 0 (osoba není přítomna). Dá se tedy říct, že se jedná o reprezentaci pohybového senzoru, ale opět bez komunikace s MQTT brokerem.

[2]Zdrojové kódy všech atomických bloků, které mají definované vlastní chování, jsou součástí programové přílohy.

Blok reprezentující konkrétní místnost má tedy 3 výstupní porty (teplota v pokoji, nastavená teplota uživatelem a hodnotu udávající zda je osoba v místnosti).

Kotelna zde není, protože by neměla na model žádný vliv. Samozřejmě v realitě má vliv na topení a na fungování celého systému, nicméně svou funkcí nijak nezasahuje do řídicího systému. Pouze od něj přijímá informace o tom, jak má či nemá topit.

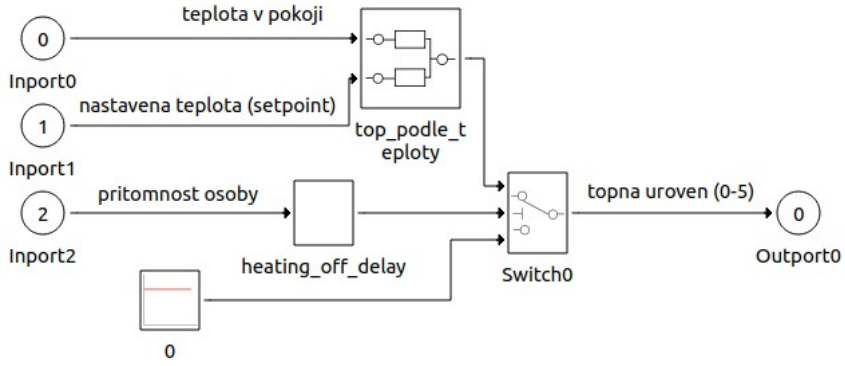
Generátor venkovní teploty

Součástí modelu prostředí je i blok `generator_venkovni_teploty` který generuje venkovní teplotu. Tento blok není úplně součástí modelu domu, jak můžeme vidět na obrázku B.1, nicméně jeho výstup (venkovní teplota) je jeden ze vstupů modelu domu. Skládá se z bloků pro generování náhodných čísel a sinusových vln tak, aby co nejlépe vystihoval venkovní teploty. Generátor lze i upravit tak, aby generoval více zimní/letní teploty.

7.1.2 Řídicí systém

řídicí systém, který je zobrazen na obrázku B.3, se skládá z pěti bloků `coupled`. Čtyři bloky zde představují takové menší řídicí systémy (regulátory teploty), které regulují teplotu v každém pokoji zvlášť, a jeden blok se stará o správný chod kotelny.

Každý regulátor teploty má tři vstupy a jeden výstup. Aktuální teplota v pokoji (výstup z teplotního čidla), nastavená teplota v pokoji a hodnota udávající přítomnost osoby v pokoji (výstup z pohybového senzoru) jsou vstupní hodnoty a topná úroveň (pro aktuátor ventilu topení) je výstupní hodnota.



Obrázek 7.2: Blok regulátoru teploty pokoje osoby R.

Jak můžeme vidět na obrázku 7.2, blok regulátoru teploty se skládá ze čtyř různých bloků. Blok `top_podle_teploty` na základě aktuální teploty v pokoji a nastavené teploty uživatelem určí topnou úroveň. Skládá se z několika atomických bloků konstant, sčítaček, porovnání a přepínačů. Atomický blok `heating_off_delay` má nadefinované vlastní chování a jeho zdrojový kód je ve stejnojmenném souboru `.cpp`. Umožňuje to, aby řídicí systém při nepřítomnosti osoby nějakou dobu „počkal“ před tím, než topení úplně vypne. Tuto dobu si můžeme nastavit jako parametr bloku. Atomický blok `switch` určuje, zda na výstup pošle hodnotu topné úrovně (z bloku `top_podle_teploty`) nebo hodnotu 0 (z atomického bloku konstany), na základě toho, jestli osoba je či byla před určitou dobou přítomna v pokoji.

Řízení kotelny je realizováno tak, že se sečtou topné úrovně všech pokojů a na základě výsledku se určí topná intenzita kotelny. Výstupem tohoto bloku je tedy hodnota 0 až 10.

7.1.3 Komunikace mezi moduly

Komunikace mezi řídicím systémem a modelem domu nebo skutečným domem se nachází v samostatném bloku. Zde se nachází zbylé funkce, které vykonávají senzory, a spolu s namodelovaným chováním v modelu domu tvoří kompletní model teplotního či pohybového senzoru. Jak přesně vypadá struktura MQTT zpráv ze senzorů a řídicího systému je popsána v sekci 7.3.2.

Model `ridici_system_s_modelem_domu.pdm` jako jediný nevyužívá komunikaci přes síť pomocí protokolu MQTT. Přenos dat po síti je tu pouze naznačen atomickými bloky `Delay` pro zpoždění. V ostatních modelech je komunikace realizována pomocí protokolu MQTT.

Atomické bloky pro formátování

Když odesíláme data na MQTT broker, je dobré je vložit do nějakého formátu (v tomto případě JSON), aby zpráva měla nějakou strukturu a aby se vědělo jaká data jsou odesílána. MQTT zpráva je tedy řetězec znaků a čísel v nějakém formátu. Na druhou stranu, všechny původní atomické bloky v PowerDEVS, které jsou k dispozici, pracují pouze s číselným datovým typem (double). Proto byly vytvořeny tyto atomické bloky, které slouží nejen pro formátování, ale také pro převedení datového typu double na datový typ string[3] a naopak.

Nejen pro teplotní senzor byl vytvořen atomický blok (zdrojový kód `set_template.cpp`), který umí vstupní hodnotu „zaobalit“ do určitého formátu. Tento formát si uživatel zvolí sám pomocí parametrů tohoto bloku, kde zadá prefix (část zprávy před hodnotou) a postfix (část zprávy po hodnotě). Další atomický blok, který je vytvořený pro pohybový senzor, (`double_to_bool.cpp`) umí převést hodnotu ze vstupu na pravdivostní (pokud dostane na vstupu kladné číslo, tak je výstup „TRUE“, jinak „FALSE“). Oba zmíněné bloky na vstupu očekávají datový typ double a jejich výstup je v datovém typu string. Opak bloku `double_to_bool` je blok `bool_to_double`, který na vstupu očekává řetězec „TRUE“ nebo „FALSE“ a podle toho pošle na výstup hodnotu 1 nebo 0.

Další blok, který pracuje s formátováním, je používán pro nalezení konkrétních dat ve formátu JSON a převedení těchto dat na datový typ double. Tento blok je používán pro zpracování dat z přijatých MQTT zpráv. Zdrojový kód se nachází v souboru `json_parser.cpp` a využívá knihovnu JsonCpp[4], která slouží pro vyhledávání dat ve formátu JSON.

Atomické bloky pro komunikaci s MQTT brokerem

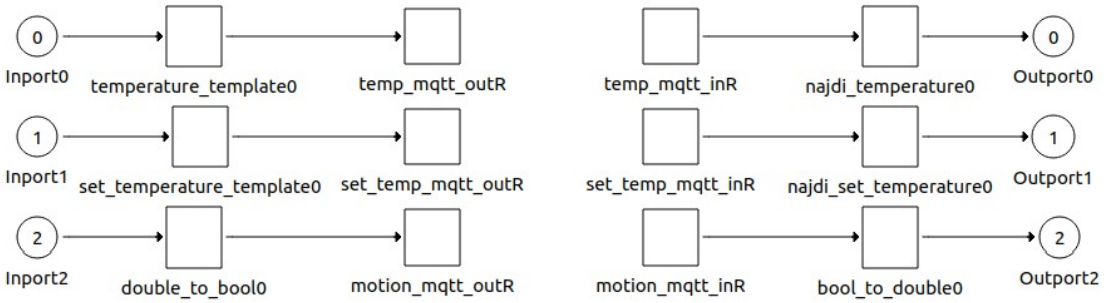
Atomické bloky, které slouží pro odesílání a přijímání dat, vyžadují nainstalovanou knihovnu Eclipse Paho MQTT C++ Client Library[5], která je zahrnuta v jejich zdrojovém kódu. Tyto bloky slouží jako klienti, kteří při inicializaci (funkce `Init`) navážou spojení s MQTT brokerem a před úplným ukončením simulace zavolají funkci `Exit`, která zaručí to, aby se klienti odpojili od MQTT brokeru.

Atomický blok, který slouží jako klient pro odeslání dat na MQTT broker (*publisher*), očekává na vstupu již kompletní strukturu MQTT zprávy v datovém typu string. V parametrech tohoto bloku si uživatel nastaví IP adresu a port MQTT brokeru, uživatelské jméno a heslo, kterým se přihlásí na broker, ID klienta a topik publikované zprávy. Interní přechod ani výstupní událost se v tomto bloku nevykonávají. Když přijde nějaká vstupní událost,

[3] Datový typ string slouží v některých programovacích jazycích pro uchovávání nějaké posloupnosti znaků.
[4] https://github.com/open-source-parsers/jsoncpp
[5] https://github.com/eclipse/paho.mqtt.cpp

tak blok publikuje vstupní hodnotu (textový řetězec) jako MQTT zprávu na určený topik, vykonává se tedy pouze externí přechod. Zdrojový kód tohoto atomického bloku je uložen v souboru `mqtt_pub.cpp`.

Pro přijímání MQTT zpráv byl vytvořen blok, který slouží jako *subscriber*. V parametrech si uživatel opět nastaví potřebnou IP adresu a port MQTT brokeru, uživatelské jméno a heslo, kterým se na broker přihlásí, ID klienta a hlavně topik, na kterém bude klient odposlouchávat. Blok by neměl zasahovat do běhu simulace tím, že bude čekat než dojde zpráva. Z tohoto důvodu je zde použita asynchronní callback funkce, jejíž zdrojový kód je v souboru `mqtt_sub_callback.cpp`. Tato funkce běží na pozadí a pokud přijde nějaká zpráva s topikem, na kterém se odposlouchává, callback zavolá funkci `externalinput()`, která způsobí, že se vykoná externí přechod. Výstupní událost následně předá na výstup obdrženou zprávu. Interní přechod se nevykonává. Zdrojový kód tohoto bloku je `mqtt_sub.cpp`.



Obrázek 7.3: Příklad síťové komunikace za použití atomických bloků, které využívají protokol MQTT.

7.2 Node-RED

V programu Node-RED byl vytvořen alternativní řešení řídicího systému i modelu domu pro porovnání s tím, který je vytvořený v PowerDEVS. Jde tedy o implementaci řídicího systému i modelu domu, které jsou komunikačně propojené stejně jako v PowerDEVS (pomocí protokolu MQTT). Díky této komunikaci lze řídicí systém a model domu propojit bez ohledu na to, jakým způsobem jsou tyto moduly realizovány. Toto propojení můžeme vytvořit v simulaci, která je synchronizovaná s reálným časem, nebo můžeme i případně připojit reálný dům s jeho komunikační infrastrukturou.

Node-RED byl nainstalován jako doplněk pro systém HomeAssistant, který běží jako nezávislé zařízení v lokální síti.

Je zde vytvořen model domu, řídicí systém a dohledový systém, jako tři na sobě nezávislé flows. Všechny tyto tři moduly mají v sobě uzly pro komunikaci s MQTT brokerem. Pro zpracování dat z přijatých MQTT zpráv jsou použity uzly `change node` s výrazy JSONata[6] a pro vytvoření správného formátu dat před odesláním zpráv slouží uzly `template`.

7.2.1 Model domu

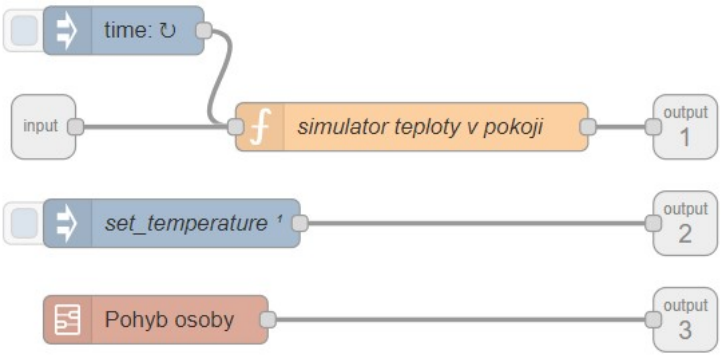
Model domu se skládá ze čtyř identických subflows, které představují pokoje, a jednoho samostatného subflow *generátor venkovní teploty*. Subflow pokoje se skládá z funkce, která

[6] JSONata je lehký dotazovací a transformační jazyk pro JSON data. [9]

simuluje teplotu pokoje na základě venkovní teploty a úrovně topení. Model domu můžeme vidět na obrázku B.2.

Každý pokoj očekává na vstupu zprávu od řídicího systému o intenzitě topení. Spolu s venkovní teplotou, kterou generuje generátor venkovní teploty, tvoří vstupy do subflow pokoje, který má navíc ještě vlastnosti, ve kterých lze nastavit počáteční hodnoty (teplota pokoje a nastavená teplota uživatelem). Jak můžeme vidět na obrázku 7.4, subflow pokoje má pouze jeden vstupní port. Více vstupních portů totiž Node-RED neumožňuje. Z tohoto důvodu jsou různé typy zpráv rozlišeny vlastností `topic`.

V subflow pokoje je uzel `function` (*simulator teploty v pokoji*), který zpracovává venkovní teplotu a topnou úroveň ze vstupního portu a na základě jejich hodnot mění teplotu v pokoji, což je zároveň výstupní hodnota této funkce. Další uzel, který vstupuje do této funkce je uzel `inject`, který nám zajišťuje to, aby se zpráva o teplotě v pokoji pravidelně odesílala každých x sekund. Kromě toho má subflow pokoje na výstupu také nastavenou teplotu uživatelem (*set_temperature*) a přítomnost osoby. Simulace pohybu osob je „zaobalena“ do subflow *Pohyb osoby* a to je vytvořeno kombinací uzlů `input`, `delay` a `change node`. Výstupy každé místnosti tedy jsou pokojová teplota, nastavená teplota uživatelem a hodnota o přítomnosti osoby.



Obrázek 7.4: Subflow pokoje.

Subflow pokoje je, co se týče funkcionality, identický jako blok pokoje vytvořený v PowerDEVS (obrázek 7.1).

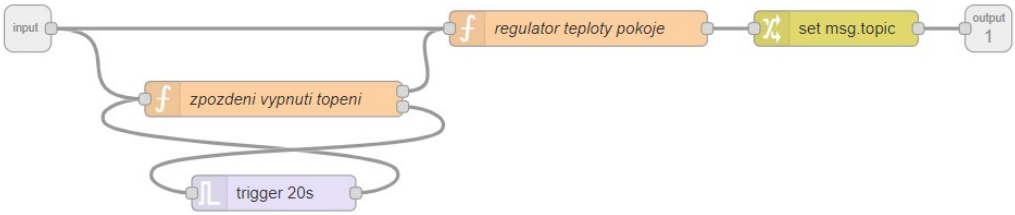
7.2.2 Řídicí systém

řídicí systém v Node-RED je zobrazen na obrázku B.4. Skládá se, podobně jako v PowerDEVS, ze čtyř regulátorů teploty. Tyto regulátory teploty jsou zde vytvořeny pro každou místnost jako čtyři identické subflows. Zpracovávají 3 různé typy zpráv (nastavená teplota uživatelem, hodnota o přítomnosti osoby a pokojová teplota), které jsou rozlišeny vlastností `topic`. Součástí tohoto subflow, které je zobrazeno na obrázku 7.5, jsou dva uzly `function` a jeden uzel `trigger`.

Uzel funkce *regulator teploty pokoje* reguluje teplotu v místnosti na základě přijatých zpráv a uzel funkce *zpozdeni vypnuti topeni* spolu s uzlem `trigger` slouží pro to, aby se přestalo topit až po nějaké době po tom, co osoba z místnosti odejde.

Co se týče funkčnosti, subflow tohoto regulátoru je identický s blokem regulátoru, který je vytvořený v PowerDEVS (obrázek 7.2).

Součástí řídicího systému je i uzel funkce, který se stará o správný chod kotelny.

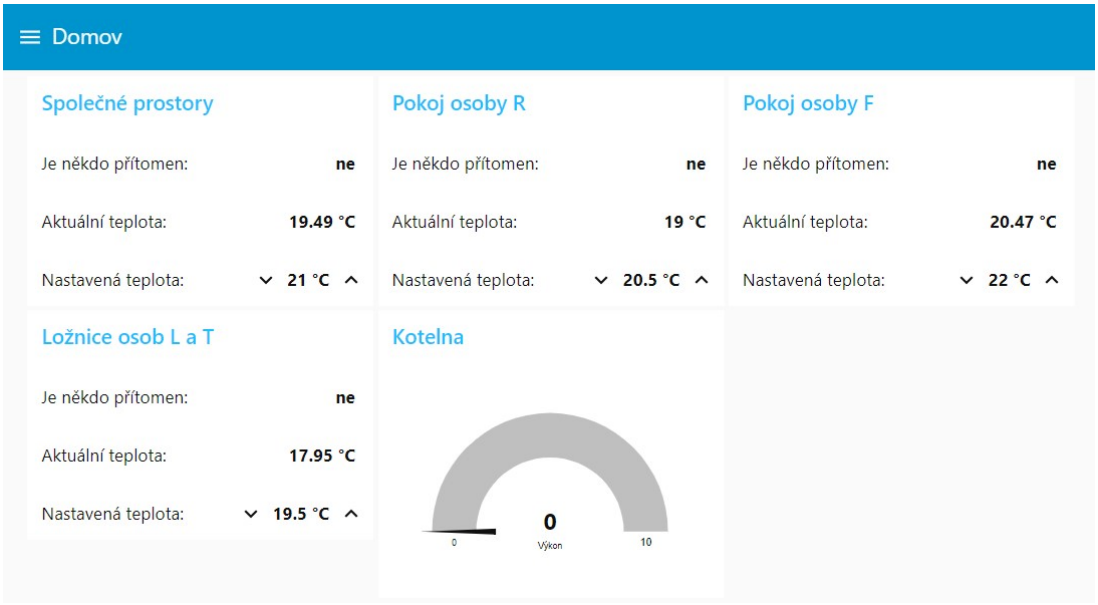


Obrázek 7.5: Subflow regulátoru teploty v pokoji.

7.2.3 Dohledový systém

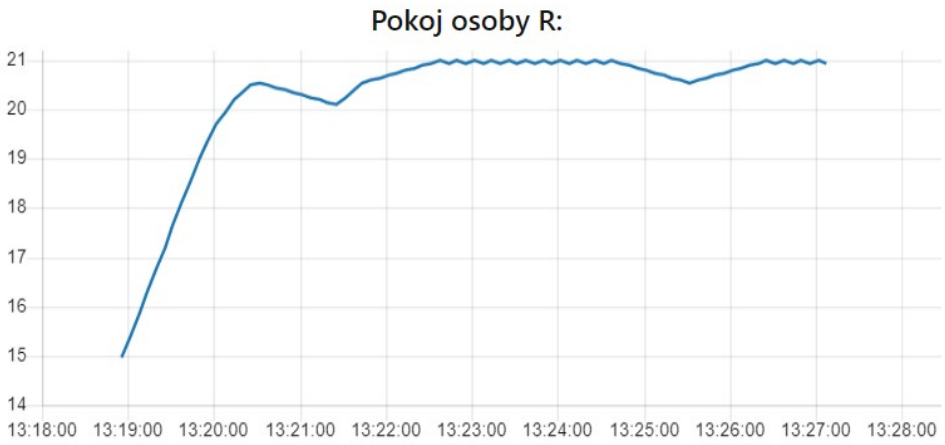
Pro dohledový systém byl využit Node-RED Dashboard. Tento modul poskytuje sadu uzlů pro vytvoření řídicího panelu dat „v živém přenosu“. Uživatel si ho může snadno otevřít v internetovém prohlížeči na jakémkoliv zařízení, které je připojeno na stejnou lokální síť jako Node-RED.

Dohledový systém monitoruje všechna data, která přijdou na MQTT broker. K řídicímu systému a senzorům je připojen pomocí uzlů pro MQTT komunikaci, které fungují jako *subscriber*. V listu „Domov“ uživatel může vidět teplotu ve všech pokojích, zda je osoba přítomna a na jakou intenzitu kotelna topí. Může si zde nastavit i jakou teplotu chce v konkrétním pokoji. Tento systém odesílá zprávu řídicímu systému opět pomocí uzlů pro komunikaci skrz protokol MQTT, které fungují jako *publisher*.



Obrázek 7.6: Uživatelské rozhraní dohledového systému.

Další list „Monitor“ slouží pro monitorování konkrétních MQTT zpráv, které zrovna přijdou na MQTT broker. Jsou zde i grafy (obrázek 7.7), ve kterých můžeme vidět, jak se teplota v pokojích během dne změnila. Z grafu lze vyčíst, že počáteční teplota byla 15 °C a následně rychle stoupala. Podle toho, kdy teplota klesá, je možné zhruba určit, kdy se v místnosti nikdo nenacházel.

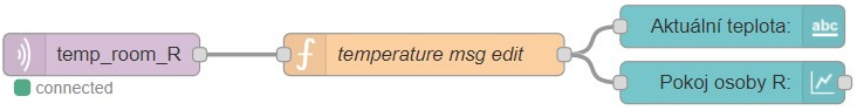


Obrázek 7.7: Graf průběhu teploty v pokoji osoby R.

Dokumentace

Flow celého dohledového systému je zobrazeno na obrázku B.5 v příloze B.

Na obrázku 7.8 můžeme vidět flow, které zpracovává zprávy o aktuální teplotě. Po tom, co je zpráva odchycena z MQTT brokeru, je upravena uzlem funkce *temperature msg edit* pro její správné zobrazení do dashboardu. Zobrazení zařizují poslední dva uzly flow. Jeden je pro zobrazení teploty jako takové a druhý vytváří z přijatých zpráv graf o průběhu teploty během dne.



Obrázek 7.8: Flow pro zobrazení aktuální teploty pokoje R do dashboardu.

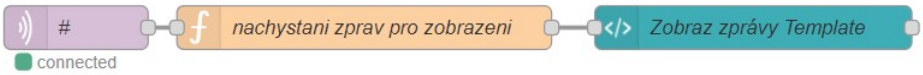
Podobně fungují i flowy pro zpracování a zobrazení výkonu kotelny, informace o tom, zda je osoba přítomna v pokoji, a část flowu o nastavené teplotě[7]. Rozdíl je pouze při upravení zprávy, kde zprávu upravuje uzel `change node` místo uzlu funkce. Pro zobrazení výkonu kotelny dobře slouží uzel `gauge`, který se zobrazuje jako měřič.

Jak je možno vidět na obrázku 7.6, uživatel si nastavuje potřebnou teplotu pomocí „šipek“. Při každé změně této hodnoty se odešle zpráva. Aby se řídicímu systému neposílala zpráva pokaždé, kdy uživatel tuto hodnotu mění, je zde vložen uzel `stoptimer`, který tomuto zabraňuje. Tento uzel donutí zprávu „počkat“ (v tomto konkrétním případě 5 sekund) a pokud při čekání přijde zpráva nová (nová hodnota nastavené teploty), původní zprávu uzel nahradí novou zprávou a čeká nových 5 sekund. Pokud žádná nová zpráva během čekání nepřijde, uzel původní zprávu pustí dál a následně je odeslána na řídicí systém.

Na zobrazení zpráv (obrázek 7.9), které přijdou na MQTT broker, dobře slouží dashboard uzel `template`, do kterého lze napsat HTML[8] kód, pomocí něhož je vytvořena tabulka. Tabulka se skládá z 10 posledních zpráv a zobrazuje čas, kdy přesně konkrétní zpráva přišla, její topik a obsah. Pro každou tuto vlastnost zprávy je vytvořen seznam, o který se stará

[7] Model domu simuluje i chování osob, které si také mohou nastavit potřebnou teplotu.
[8] Hypertext Markup Language (HTML) je značkovací jazyk používaný pro tvorbu webových stránek.

uzel funkce *nachystani zprav pro zobrazeni*. Tato funkce navíc zajišťuje to, aby tabulka zpráv byla stále aktuální.



Obrázek 7.9: Flow pro zpracování a zobrazení do tabulky všech zpráv, co přijdou na MQTT broker.

7.3 Komunikace mezi řídicím systémem a senzory/aktuátory

Jak již bylo řečeno, pro komunikaci řídicího systém s modelem domu (reálným domem) jsem se rozhodl pro protokol MQTT. Tento protokol využívá plno IoT zařízeních a tím pádem napojení řídicího systému na reálný dům by nemělo být moc komplikované.

7.3.1 MQTT broker

Pro MQTT broker jsem vybral doplněk Mosquitto broker[9] pro HomeAssistant. Tento doplněk nám spustí plně funkční MQTT broker, který si můžeme nakonfigurovat podle sebe. Jako konfiguraci jsem si zvolil přihlášení klientů pod jménem a heslem, což nám poskytuje alespoň nějaké zabezpečení MQTT zpráv.

7.3.2 Struktura MQTT zpráv

Je důležité si určit přesnou strukturu MQTT zpráv, kterou budeme používat. Namodelované senzory by měli odesílat stejné nebo velice podobné zprávy jako reálné senzory. Různé teplotní a pohybové senzory používají různé struktury zpráv. Kromě pohybových senzorů je použit JSON pro ukládání dat.

Většina pohybových čidel umí odesílat zprávy s pouze pravdivostní hodnotou (řetězec „TRUE“ nebo „FALSE“) stejně jako například senzor HC-SR501.

Teplotní čidla jsem si vybral ta, která běží na firmwaru ESPEasy. Podle dokumentace[10] by struktura zprávy, která nese informaci o tom, že je v pokoji 19.5 °C, mohla vypadat následovně:

```
{
  "TaskValues": [{
    "ValuesNumber": 1,
    "Name": "temperature",
    "Value": 19.5
  }]
}
```

Tyto teplotní čidla umí většinou odesílat více informací (např. vlhkost vzduchu), nicméně my pracujeme pouze s teplotou.

Zprávy, které odesílá řídicí nebo dohledový systém mají strukturu velice jednoduchou. Většina zařízeních by měla být schopná přijmout a zpracovat data tak, jak si to uživatel

[9] https://mosquitto.org
[10] https://espeasy.readthedocs.io/en/latest/Reference/Command.html

nakonfiguruje. Zde můžeme vidět například zprávu, kterou posílá řídicí systém aktuátoru ventilu topení, o tom, jakou intenzitou má topit:

```
{
   "data": {
      "heat_level": 3
   }
}
```

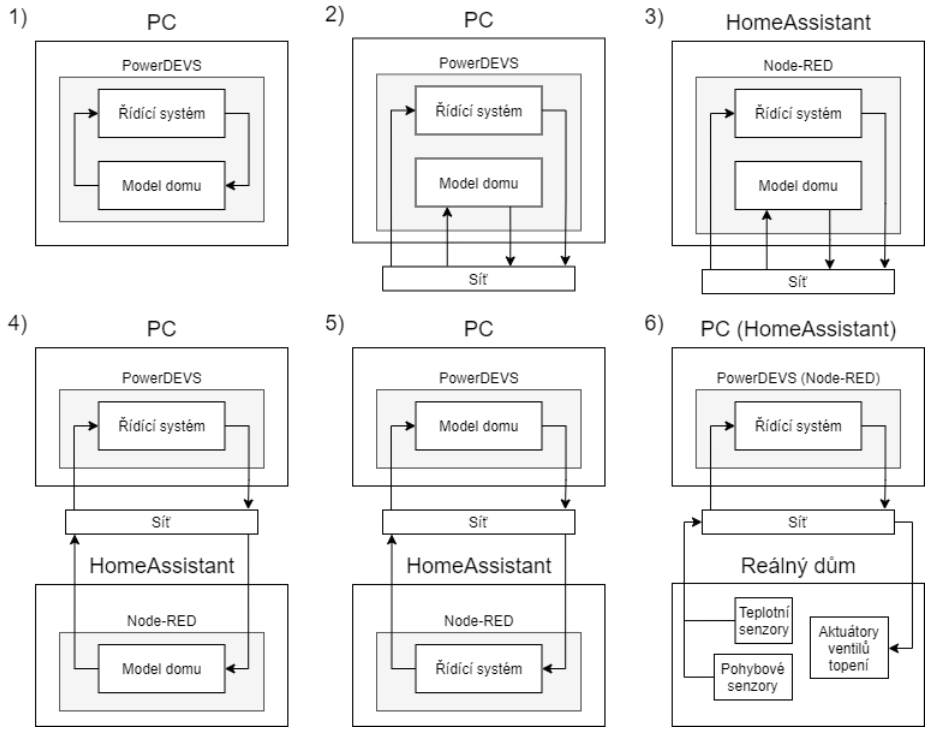
Podobně jako tato vypadají i další zprávy.

Kapitola 8

Ověření funkčnosti a vyhodnocení

V této kapitole si ukážeme, že řídicí systém v PowerDEVS a Node-RED lze vzájemně zaměnit beze změny fungování celého systému.

Nejprve byl systém i s komunikací vytvořen pouze v prostředí PowerDEVS. Poté byla přidána komunikace přes síť pomocí protokolu MQTT. Celý tento systém byl následně vytvořen v programu Node-RED. Toto nám umožnilo spustit část systému v PowerDEVS a část v Node-RED a částečně tak nasimulovat spojení s reálným domem. Řídicí systém by nakonec mohl pracovat s reálným domem a jeho senzory. Vytváření celého systému je děláno principem model continuity.



Obrázek 8.1: Fáze vytváření systému podle principu model continuity.

Jako referenční příklad si vezměme pouze pokoj osoby R, kdy celá simulace bude trvat 200 jednotek času. Počáteční teplota v tomto pokoji je 15 °C a nastavená teplota uživatelem je 21 °C. Řídicí systém by měl tedy vytopit pokoj na 21 °C, pokud je osoba přítomna

v pokoji. Osoba je v pokoji přítomna od času 0 do času 100, poté senzor pohybu nesnímá žádný pohyb po dobu 50 časových jednotek a následně se osoba opět do pokoje vrátí v čase 150 a bude v něm až do konce simulace. Čas, po kterém řídicí systém vypne topení při nepřítomnosti osoby v pokoji, je nastaven na 30.

Tato simulace byla spuštěna ve všech fázích z obrázku 8.1. Výstupní hodnoty těchto simulací byly ve všech případech velmi podobné (lišily se pouze v rámci konfiguračních problémů – je těžké plně synchronizovat PowerDEVS s Node-RED). Nicméně z hlediska funkčnosti jsou tyto systémy naprosto stejné.

Co se týká přehlednosti systému a modelování, je rozhodně přehlednější systém, co je vytvořený v PowerDEVS. Díky blokům `coupled` se mi podařilo vytvořit prakticky stejný diagram jako diagram balíčků ze sekce 6.5. V Node-RED můžeme něco podobného vytvořit pomocí uzlů `Subflow`, nicméně není to zdaleka tak praktické jako blok `coupled`. Při vytváření většího systému pak začíná být Node-RED nepřehledný.

Další věc je vytvoření modelu domu. Díky tomu, že PowerDEVS je udělaný pro vytváření simulačních modelů formalismem DEVS, bylo vytvoření modelu domu o dost jednodušší a lepší i díky možnosti použití Petriho sítí. Na druhou stranu Node-RED vyniká při automatizaci. Vytvořit řídicí systém je zde tedy o dost pohodlnější.

Absence více vstupních portů uzlů v Node-RED způsobovala zbytečné komplikace, nicméně díky vlastnostem zpráv se tyto komplikace dají lehce obejít.

Velká výhoda, kterou Node-RED má, je velké množství knihoven, které jsou k dispozici pro práci s IoT zařízeními. Součástí těchto knihoven jsou i uzly pro práci s protokolem MQTT. V PowerDEVS žádné bloky pro komunikaci pomocí tohoto protokolu nejsou a tak jsem si je musel vytvořit.

Z textu jde vidět, že PowerDEVS a Node-RED jsou dva velice rozdílné programy a každý je zaměřený na něco jiného. I přes to se mi podařilo v PowerDEVS vytvořit plně funkční řídicí systém a model domu, které jsou naprosto stejné jako v Node-RED.

Kapitola 9

Závěr

PowerDEVS je velice silný nástroj, který nám dobře slouží nejen pro modelování, ale také pro simulaci. Naimplementované atomické bloky, které nám umožňují komunikaci po síti pomocí protokolu MQTT, tvoří komunikační rozhraní, díky němuž jsme schopni propojit simulovaný systém s reálným systémem.

Tímto způsobem jsem tedy schopen dělat simulační studie. Řídicí systém, který i se simulací namodelujeme v PowerDEVS, nebudeme muset při zasazení do reálného systému nijak měnit.

Na druhou stranu PowerDEVS není moc dobře optimalizovaný a dle mého názoru ani otestovaný nástroj. O tom i dost dobře vypovídá dokumentace, která není úplná. Při implementaci systému jsem narazil na řadu chyb, bugů a problémů, které mi dost komplikovaly tuto práci. Tyto chyby se týkají hlavně editorů modelu a atomických bloků, simulační jádro DEVS se zdá být v pořádku. PowerDEVS je zkrátka silný nástroj s velkým potenciálem, který zatím bohužel není dotažený do konce.

V tomto programu se mi nicméně podařilo vytvořit systém, který je schopen běhu na lokálním počítači, cloudu či na Raspberry Pi. Celý systém je vytvořen tak, aby další programátor mohl lehce navázat na započatou práci, avšak já silně doporučuji počkat na optimalizovanou a stabilnější verzi PowerDEVS.

Jako další možnosti vylepšení systému bych přidal větší bezpečnost síťové komunikace a možná i mobilní notifikace. Co se týká principu model continuity, jako návrh na zlepšení by mohlo být vytvoření programu, který bude schopen transformovat modely mezi Node-RED a PowerDEVS.

Ke Smart Home bych chtěl říct jen to, že chytré topení se dá jednoduše vytvořit finančně nenáročným způsobem, pokud se člověk naučí Node-RED. Tento nástroj je velice jednoduchý a poměrně rychle si zde vytvoříte celý řídicí systém. Celý systém, i s aktuátory ventilu topení, teplotními a pohybovými senzory lze vytvořit v rámci stovek korun.

Literatura

[1] Bergero, F. a Kofman, E. PowerDEVS: A tool for hybrid system modeling and real-time simulation. *Simulation*. Leden 2011, sv. 87, s. 113–132, [cit. 2021-05-25]. DOI: 10.1177/0037549710368029.

[2] Brown, A. *An introduction to Model Driven Architecture* [online]. 2004 [cit. 2021-01-15]. Dostupné z: https://www.ibm.com/developerworks/rational/library/3100.html.

[3] Dague, S. *Why can't we have the Internet of Nice Things? A home automation primer* [online]. 2017 [cit. 2021-01-19]. Dostupné z: https://opensource.com/article/17/7/home-automation-primer.

[4] eclipse. *4diac Documentation* [online]. 2021 [cit. 2021-01-21]. Dostupné z: https://www.eclipse.org/4diac/en_help.php.

[5] Ezequiel Pecker Marcosig, J. I. G. a Castro, R. *DEVS-OVER-ROS (DOVER): A FRAMEWORK FOR SIMULATION-DRIVEN EMBEDDED CONTROL OF ROBOTIC SYSTEMS BASED ON MODEL CONTINUITY*. Buenos Aires, Argentina: IEEE, 2018 [cit. 2021-06-07]. ISBN 978-1-5386-6572-5. Dostupné z: https://www.informs-sim.org/wsc18papers/includes/files/105.pdf.

[6] Fritzson, P., Pop, A., Abdelhak, K., Ashgar, A., Bachmann, B. et al. The OpenModelica Integrated Environment for Modeling, Simulation, and Model-Based Development. *Modeling, Identification and Control*. 1. Norwegian Society of Automatic Control. 2020, sv. 41, č. 4, s. 241–295, [cit. 2021-01-18]. DOI: 10.4173/mic.2020.4.1.

[7] Home Assistant. *Integrations* [online]. 2020 [cit. 2021-01-19]. Dostupné z: https://www.home-assistant.io/integrations/.

[8] Hu, X. a Zeigler, B. Model continuity in the design of dynamic distributed real-time systems. *IEEE Transactions on Systems, Man, and Cybernetics - Part A: Systems and Humans*. 2005, sv. 35, č. 6, s. 867–878, [cit. 2021-07-23]. DOI: 10.1109/TSMCA.2005.851283.

[9] JSONata.org. *JSONata Documentation* [online]. 2021 [cit. 2021-06-20]. Dostupné z: https://docs.jsonata.org/overview.html.

[10] Kofman, E. a Bergero, F. *PowerDEVS* [online]. CIFASIS – CONICET, FCEIA, UNR, Argentina, 2015 [cit. 2021-05-25]. User's Guide. Dostupné z: https://twiki.cern.ch/twiki/pub/Main/PowerDEVSProyect/PD_UserGuide.pdf.

[11] Lakshmanan, S. *House Automation using Home Assistant* [online]. 2020 [cit. 2021-01-19]. Dostupné z: https://towardsdatascience.com/house-automation-using-home-assistant-191ee017027d.

[12] Modeliosoft. *Modelio* [online]. 2021 [cit. 2021-01-21]. Dostupné z: https://www.modelio.org.

[13] Oasis Open. *MQTT Version 5.0* [online]. 2014 [cit. 2021-01-19]. Dostupné z: https://docs.oasis-open.org/mqtt/mqtt/v5.0/os/mqtt-v5.0-os.html#_Toc3901000.

[14] Object Management Group. *Unified Modeling Language* [online]. 2017 [cit. 2021-05-24]. Dostupné z: https://www.omg.org/spec/UML/.

[15] Object Management Group. *What is SysML?* [online]. 2021 [cit. 2021-01-17]. Dostupné z: https://www.omgsysml.org/what-is-sysml.htm.

[16] OpenJS Foundation. *Node-RED* [online]. 2021 [cit. 2021-01-20]. Dostupné z: https://nodered.org.

[17] O'Riordan, M. *Everything You Need To Know About Publish/Subscribe* [online]. ? [cit. 2021-01-19]. Dostupné z: https://www.ably.io/topic/pub-sub.

[18] Otter, M. a Elmqvist, H. *Modelica Language, Libraries, Tools, Workshop and EU-Project RealSim* [online]. 1. vyd. German Aerospace Center, Oberpfaffenhofen, Germanz and Dznasim AB, Lund, Sweden, 2001 [cit. 2021-01-18]. Modelica overview. Dostupné z: https://www.modelica.org/documents/ModelicaOverview14.pdf.

[19] Peška, R. *Node Red slibuje snadné programování IoT aplikací* [online]. 2019 [cit. 2021-01-20]. Dostupné z: https://vyvoj.hw.cz/node-red-slibuje-snadne-programovani-iot-aplikaci.html.

[20] Rouse, M. *Internet of Things (IoT)* [online]. 2020 [cit. 2021-01-16]. Dostupné z: https://internetofthingsagenda.techtarget.com/definition/Internet-of-Things-IoT.

[21] Schmidt, D. C. Model-Driven Engineering. *Computer*. 1. vyd. IEEE Computer Society. 2006, sv. 39, č. 2, s. 25–31, [cit. 2021-01-14]. ISSN 0018-9162.

[22] Schwatinski, T., Pawletta, T., Pawletta, S. a Kaiser, C. SIMULATION-BASED DEVELOPMENT AND OPERATION OF CONTROLS ON THE BASIS OF THE DEVS FORMALISM. In:. Leden 2010 [cit. 2021-07-23].

[23] Visual Paradigm. *What is Package Diagram?* [online]. 2020 [cit. 2021-06-10]. Dostupné z: https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-package-diagram/.

[24] Visual Paradigm Community Circle. *SysML Diagram Types* [online]. 2021 [cit. 2021-07-23]. Dostupné z: https://circle.visual-paradigm.com/docs/uml-and-sysml/sysml-diagram-types/.

[25] Wikibooks. *Embedded Control Systems Design/Model driven engineering* [online]. 2017 [cit. 2021-01-14]. Dostupné z: https://en.wikibooks.org/wiki/Embedded_Control_Systems_Design/Model_driven_engineering.

[26] Wikipedia Contributors. *Third-generation programming language* [online]. 2008 [cit. 2021-01-13]. Dostupné z: https://en.wikipedia.org/wiki/Third-generation_programming_language.

[27] Wikipedia Contributors. *Model-driven engineering* [online"]. 2014 [cit. 2021-01-14]. Dostupné z: https://en.wikipedia.org/wiki/Model-driven_engineering.

[28] Wikipedia Contributors. *CASE nástroje* [online]. 2020 [cit. 2021-01-09]. Dostupné z: https://cs.wikipedia.org/wiki/CASE_nástroje.

[29] Wikipedia Contributors. *Internet věcí* [online]. 2020 [cit. 2021-01-16]. Dostupné z: https://cs.wikipedia.org/wiki/Internet_v%C4%9Bc%C3%AD.

[30] Wikipedia Contributors. *MQTT* [online]. 2021 [cit. 2021-01-19]. Dostupné z: https://en.wikipedia.org/wiki/MQTT.

[31] Wikipedia Contributors. *Petri nets* [online]. 2021 [cit. 2021-07-24]. Dostupné z: https://en.wikipedia.org/wiki/Petri_net.

[32] Wikipedia Contributors. *Počítačová platforma* [online]. 2021 [cit. 2021-01-24]. Dostupné z: https://cs.wikipedia.org/wiki/Po%C4%8D%C3%ADta%C4%8Dov%C3%A1_platforma.

[33] Wikipedia Contributors. *Programovatelný logický automat* [online]. 2021 [cit. 2021-01-21]. Dostupné z: https://cs.wikipedia.org/wiki/Programovateln%C3%BD_logick%C3%BD_automat.

[34] Wikipedia Contributors. *Vestavěné systémy* [online]. 2021 [cit. 2021-01-24]. Dostupné z: https://cs.wikipedia.org/wiki/Vestav%C4%9Bn%C3%BD_syst%C3%A9m.

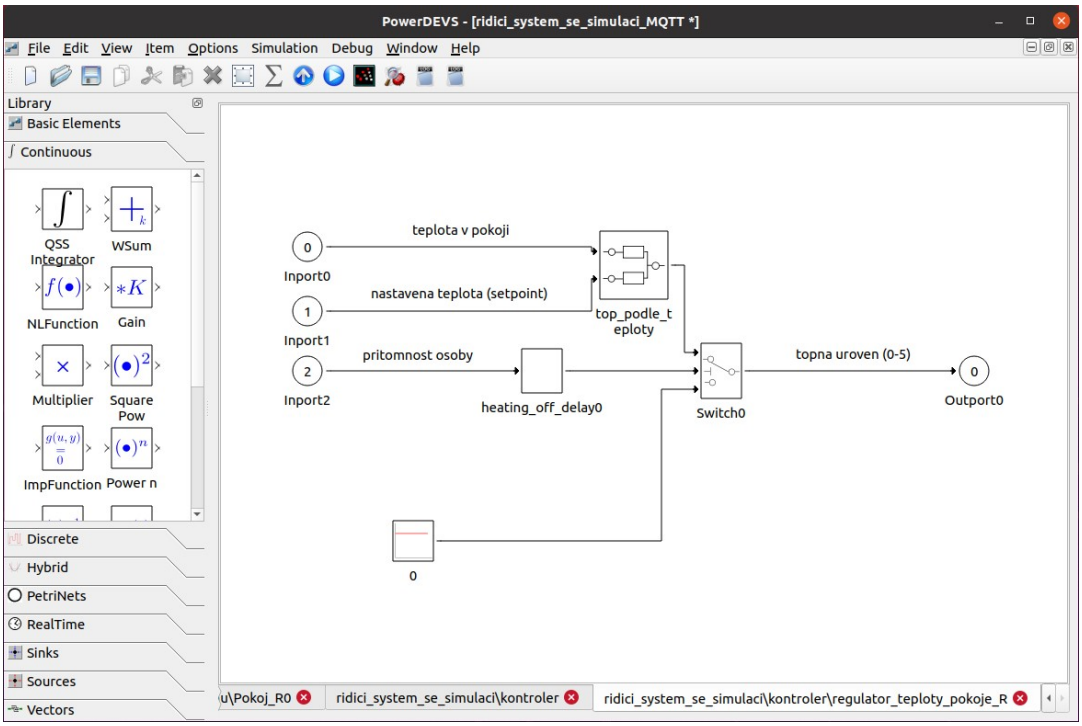
[35] Yuan, M. *Getting to know MQTT* [online]. 2020 [cit. 2021-01-19]. Dostupné z: https://developer.ibm.com/articles/iot-mqtt-why-good-for-iot.

Přílohy / Appendices

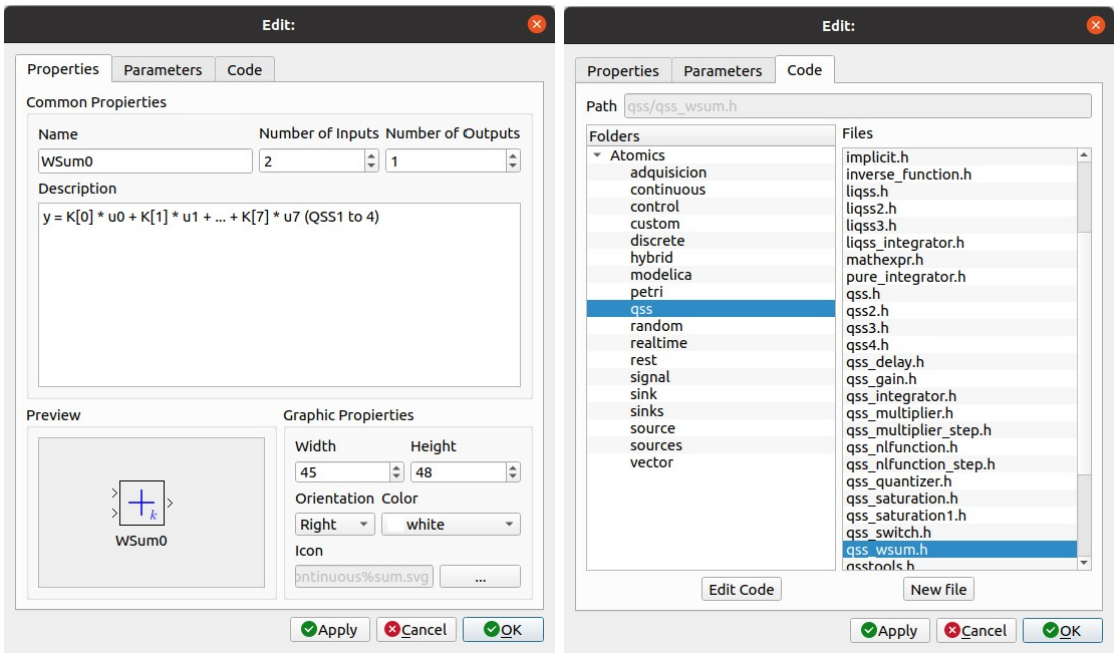
Příloha A

Snímky programu PowerDEVS

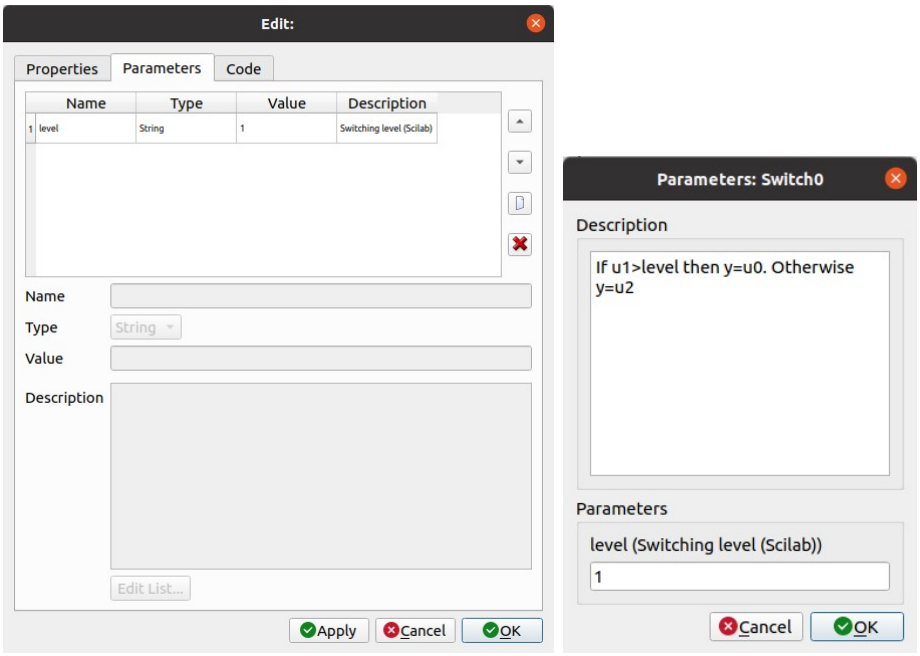
V této příloze jsou snímky programu PowerDEVS.



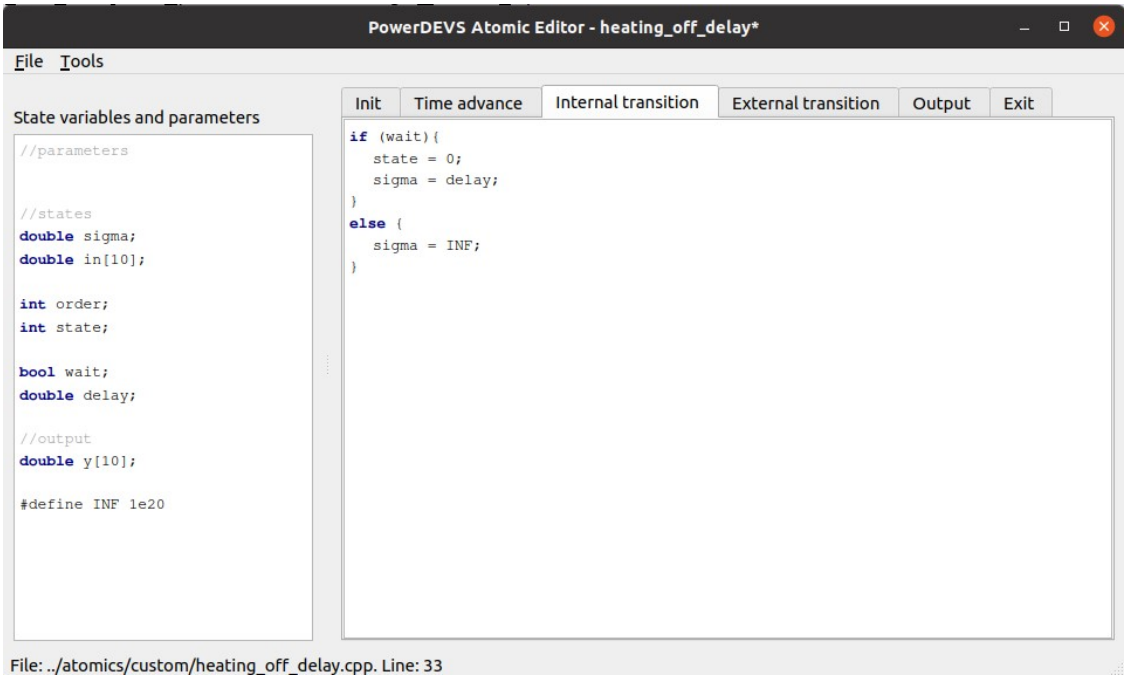
Obrázek A.1: Model Editor.



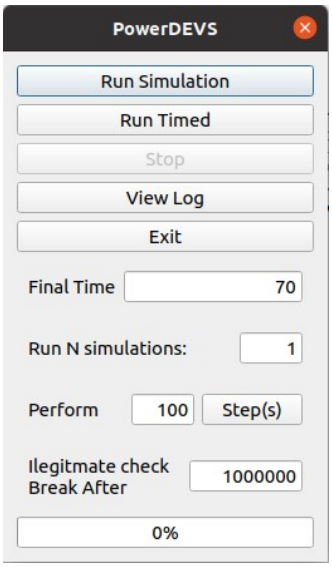
Obrázek A.2: Okno *Edit* pro upravení bloku `WSum`.



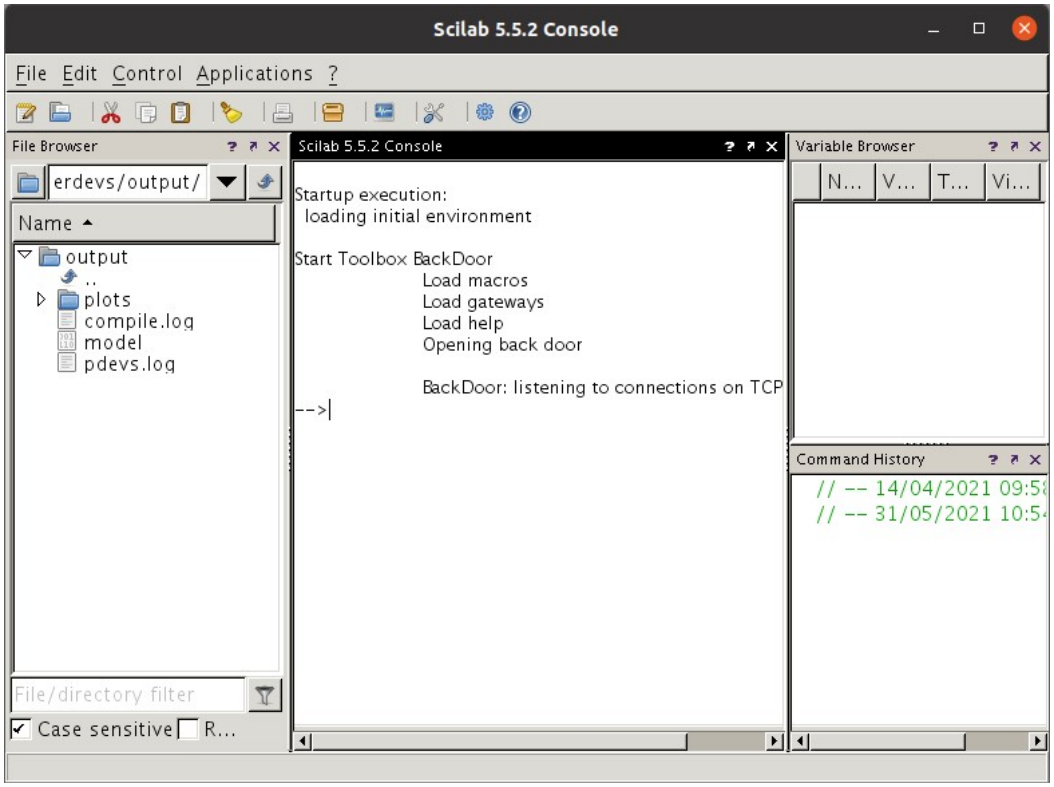
Obrázek A.3: Záložka *Parameters* okna *Edit* pro úpravu parametrů a okno *Parameters* pro nastavení parametrů bloku `Switch`.



Obrázek A.4: Atomic Editor.



Obrázek A.5: Simulační rozhraní.

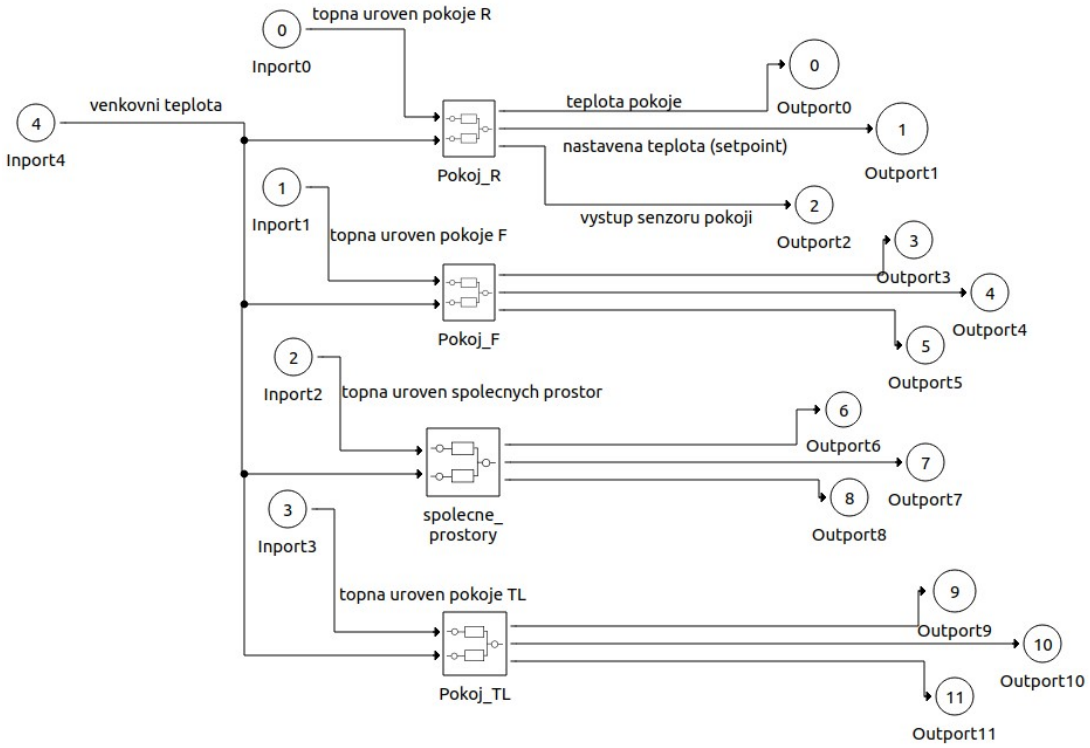


Obrázek A.6: Instance programu Scilab.

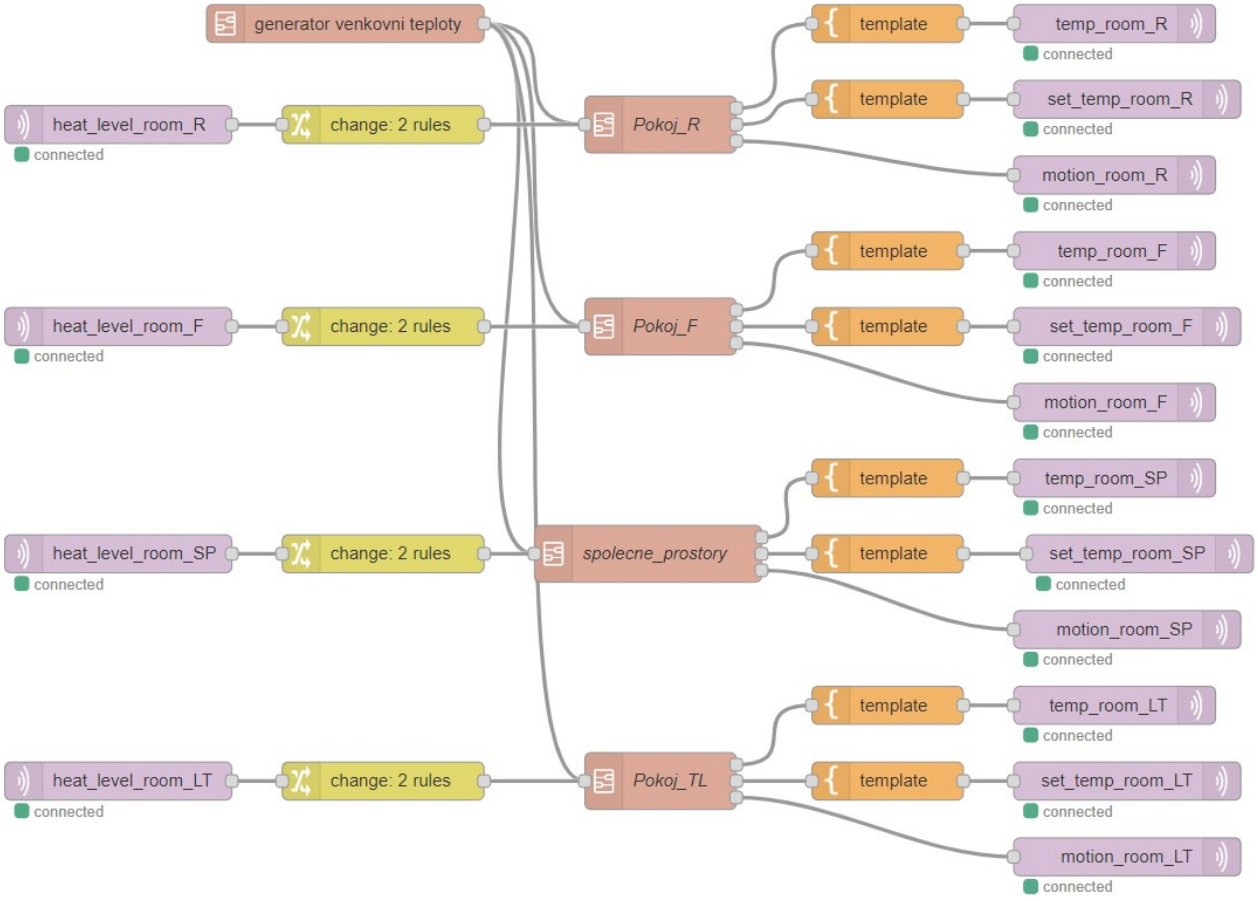
Příloha B

PowerDEVS a Node-RED modely

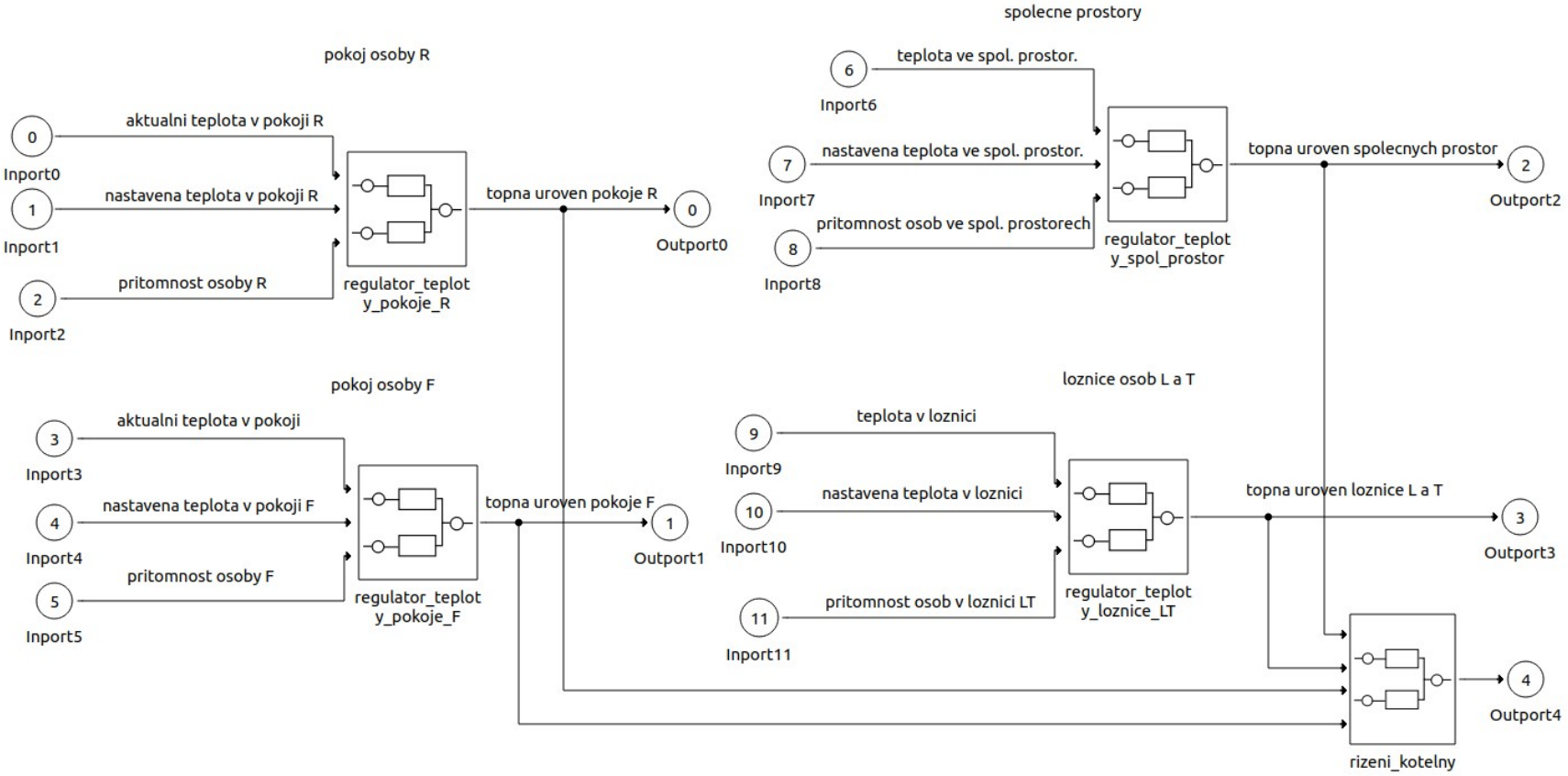
Tato příloha zobrazuje diagramy modelu domu, řídícího systému a dohledového systému. Všechny tyto modely PowerDEVS a flowy Node-RED jsou součástí programové přílohy.



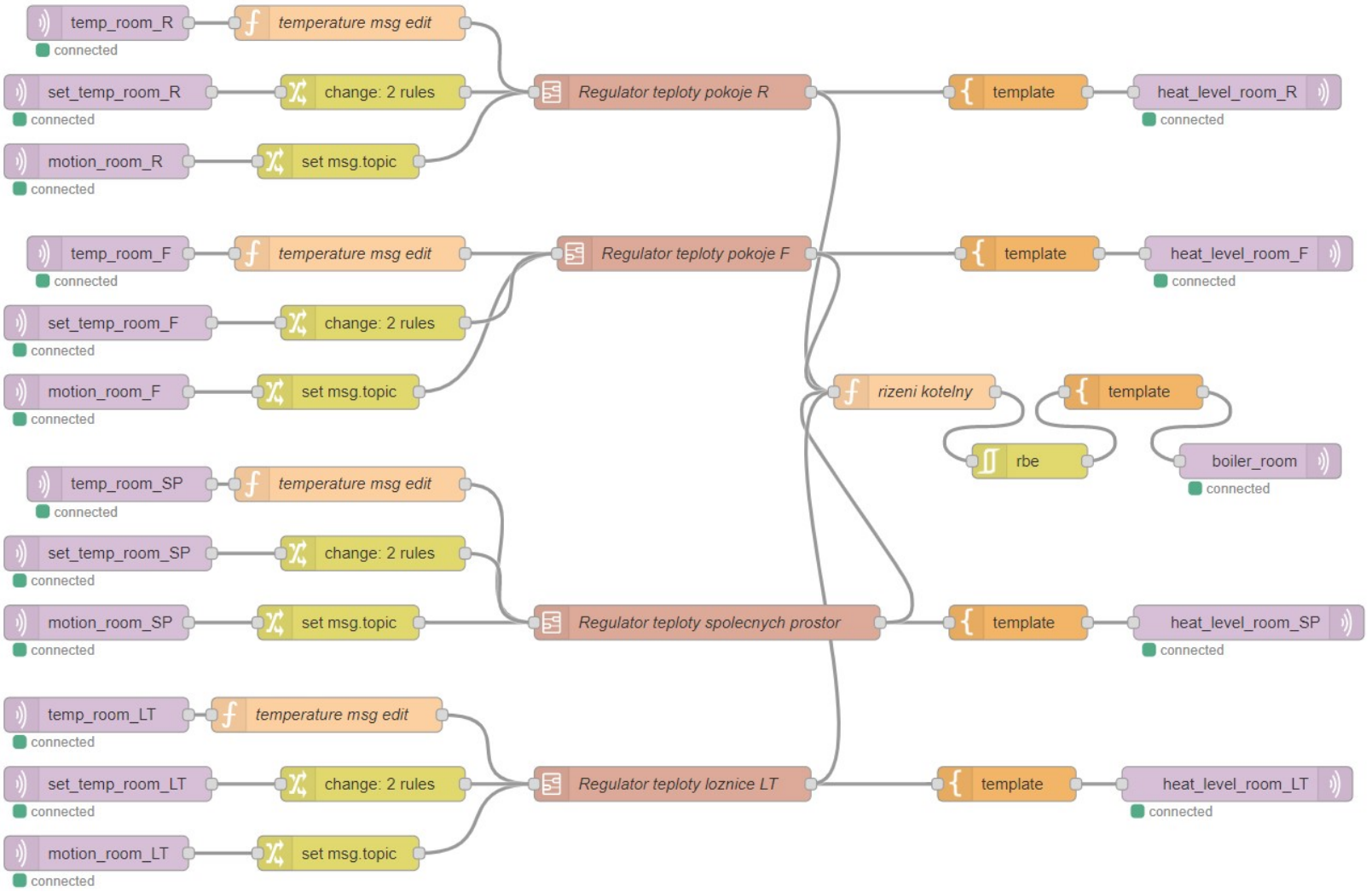
Obrázek B.1: Blok modelu domu v PowerDEVS.



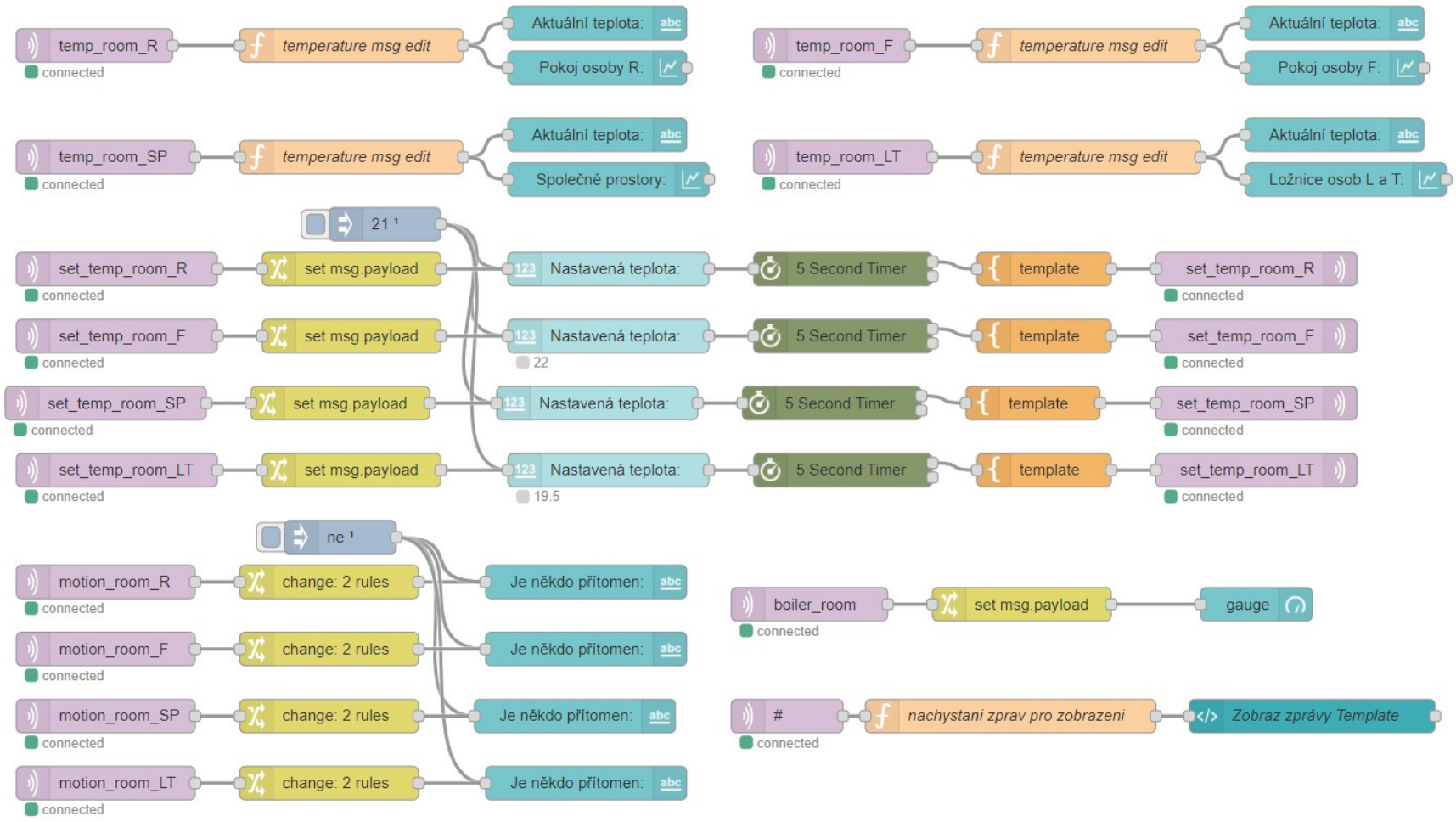
Obrázek B.2: Flow modelu domu v Node-RED.



Obrázek B.3: Blok řídícího systému v PowerDEVS.



Obrázek B.4: Flow řídícího systému v Node-RED.



Obrázek B.5: Flow dohledového systému v Node-RED.

Příloha C

Obsah přiloženého paměťového média

V paměťovém médiu se nachází programová příloha, text písemné zprávy a zdrojové soubory.

- `xcadar00.pdf` – PDF soubor písemné zprávy
- `xcadar00` – složka se zdrojovými soubory písemné zprávy
- `src` – složka se zdrojovými soubory programů
  - `README.md` – spouštěcí manuál programů
  - `MQTT_broker_config` – konfigurace MQTT brokeru
  - `powerdevs_models` – složka s modely PowerDEVS
  - `custom` – složka se zdrojovými kódy atomických bloků PowerDEVS, které mají definované vlastní chování
  - `node-red_flows.json` – JSON soubor obsahující Node-RED flows