VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INFORMAČNÍCH SYSTÉMŮ
DEPARTMENT OF INFORMATION SYSTEMS

VIZUÁLNÍ PROGRAMOVÁNÍ IOT ZAŘÍZENÍ

VISUAL PROGRAMMING OF IOT DEVICES

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — **LUKÁŠ PODVOJSKÝ**
AUTHOR

VEDOUCÍ PRÁCE — **Ing. JIŘÍ HYNEK, Ph.D.**
SUPERVISOR

BRNO 2024

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

Zadání bakalářské práce

154356

Ústav:	Ústav informačních systémů (UIFS)
Student:	**Podvojský Lukáš**
Program:	Informační technologie
Název:	**Vizuální programování IoT zařízení**
Kategorie:	Webové aplikace
Akademický rok:	2023/24

Zadání:

1. Prostudujte oblast internetu věcí (*Internet of Things*, IoT).
2. Prostudujte principy vizuálního programování a současné nástroje vizuálního programování.
3. Analyzujte požadavky na grafický editor, který umožní snadno vytvářet programy pro chytrá zařízení.
4. Dle konzultace s vedoucím navrhněte webový grafický editor pro tvorbu jednoduchých programů určených pro chytrá zařízení.
5. Navržený grafický editor implementujte.
6. Výsledný editor otestujte. Vyhodnoťte jeho použitelnost.

Literatura:

- Badii, C., Bellini, P., Difino, A., Nesi, P., Pantaleo, G., & Paolucci, M. (2019). Microservices suite for smart city applications. *Sensors*, *19*(21), 4798.
- Hynek, J. a spol. (2023). Služby pro systém řízení a monitoringu vody v retenčních nádržích. Výzkumná zpráva. Vysoké učení technické v Brně.
- Kuhail, M. A., Farooq, S., Hammad, R., & Bahja, M. (2021). Characterizing visual programming approaches for end-user developers: A systematic review. *IEEE Access*, *9*, 14181-14202.
- Ray, P. P. (2017). A survey on visual programming languages in internet of things. *Scientific Programming*, *2017*.

Při obhajobě semestrální části projektu je požadováno:
Body 1 - 4.

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

Vedoucí práce:	**Hynek Jiří, Ing., Ph.D.**
Vedoucí ústavu:	Kolář Dušan, doc. Dr. Ing.
Datum zadání:	1.11.2023
Termín pro odevzdání:	9.5.2024
Datum schválení:	30.10.2023

Abstrakt

Cílem této práce je poskytnout koncovým uživatelům nástroj vizuálního programování, který zjednoduší proces tvorby programů určených pro zařízení internetu věcí (IoT). Existuje mnoho typů zařízení IoT, která používají různé komunikační protokoly. Nedostatečná standardizace těchto zařízení nutí společnosti vytvářet řešení na míru. Uživatelé pak mají k dispozici předdefinovanou funkcionalitu, kterou lze jen mírně upravit. Jedním z řešení této nedostatečné schopnosti koncových uživatelů přizpůsobit chování zařízení je poskytnout jim větší volnost v definování logiky prostřednictvím konceptu vizuálního programování. Výsledkem této práce je vznik nové knihovny implementující vizuální programovací jazyk a vizuální editor, jehož výstupem je serializovaná podoba programu, která může být následně transformována do jazyka cílových zařízení. Výsledná knihovna je vytvořena pomocí webových technologií a lze ji integrovat do stávajících řešení.

Abstract

This thesis aims to provide end-users with a visual programming tool to simplify the process of creating programs for Internet of Things (IoT) devices. There are a lot of different types of IoT devices that use various communication protocols. The lack of standardization for these devices forces companies to create customized solutions. Users are then presented with a predefined functionality that can be only slightly altered. One of the solutions for this lack of ability for end users to customize device behaviour is to give them more freedom through the concept called visual programming. This thesis results in a new library implementing a visual programming language and a visual editor whose output is a serialized program form that can then be transformed into the language of the target devices. The resulting library is created using web technologies and can be integrated into existing solutions.

Klíčová slova

vizuální programování, internet věcí, IoT, vizuální editor, grafický editor, webová aplikace, JavaScript, TypeScript, Lit, webové komponenty

Keywords

visual programming, internet of things, IoT, visual editor, graphical editor, web application, JavaScript, TypeScript, Lit, web components

Citace

PODVOJSKÝ, Lukáš. *Vizuální programování IoT zařízení*. Brno, 2024. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Jiří Hynek, Ph.D.

Vizuální programování IoT zařízení

Prohlášení

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

.....
Lukáš Podvojský
7. května 2024

Poděkování

Chtěl bych poděkovat vedoucímu práce Ing. Jiřímu Hynkovi, Ph.D. za veškerou pomoc, trpělivost a v neposlední řadě jeho aktivní přístup a zpětnou vazbu při konzultacích.

Obsah

1 Úvod **2**

2 Internet věcí **3**
2.1 Architektura IoT 5
2.2 Přenos dat a komunikace zařízení IoT 6

3 Vizuální programování **9**
3.1 Vizuální programovací jazyky 10
3.2 Vývoj softwaru koncovým uživatelem 12
3.3 Současné nástroje vizuálního programování 14

4 Analýza **23**
4.1 Aktuální způsoby programování zařízení IoT 23
4.2 Požadavky na řešení 28

5 Návrh webového grafického editoru **30**
5.1 Navrhované řešení 30
5.2 Architektura knihovny 30
5.3 Návrh uživatelského rozhraní 31
5.4 Datový model knihovny 36

6 Implementace **40**
6.1 Integrace knihovny Lit 40
6.2 Vizuální programovací jazyk 43
6.3 Vizuální editor 44

7 Testování **52**
7.1 Testovací příklady 52
7.2 Výsledky testování 53

8 Závěr **55**

Literatura **56**

A Příklad definice zařízení **59**

B Příklad výsledného programu **60**

Kapitola 1

Úvod

V dnešní době se čím dál více elektronických zařízení stává chytrými díky integraci s internetem a schopnosti vzájemné komunikace. Tato zařízení označujeme pojmem IoT (*Internet of Things*) a může se jednat o běžné předměty jako jsou domácí spotřebiče, osvětlení, termostaty nebo automobily. Zařízení internetu věcí jsou vybavena senzory, které sbírají data, a mají schopnost připojit se k internetu k jejich sdílení a zpracování. Například chytrý termostat může monitorovat teplotu v místnosti a uživateli je pomocí aplikace umožněno regulovat teplotu na dálku. Existuje mnoho typů a modelů zařízení IoT, která využívají různé technologie pro komunikaci. Vznikají tak systémy, které zajišťují propojení, správu a konfiguraci těchto zařízení.

Programování zařízení IoT probíhá na mnoha vrstvách a za použití různých technologií. Od programování mikrokontroléru samotného zařízení, přes implementaci komunikačních protokolů v aplikačních rozhraních, až po sběr a analýzu dat v databázi. Vzhledem k různorodosti technologií používaných v IoT je nezbytné přizpůsobit programovací prostředky konkrétním potřebám a omezením daných zařízení. To má za následek vznik systémů IoT vytvořených na míru, kde je funkcionalita zařízení pro koncového uživatele většinou pevně definována. Záleží poté na vývojářích systémů IoT, zda poskytnou koncovým uživatelům vhodný nástroj pro upravení logiky zařízení a zpřístupní tak uživatelům větší kontrolu nad zařízeními.

Tato bakalářská práce se podrobněji zabývá průzkumem v oblasti vývoje softwaru koncovým uživatelem. Konkrétně přístupem zvaným vizuální programování, který uživatelům umožňuje definovat vlastní logiku za použití grafických prvků. Uživatelé mohou využít nástroje vizuálního programování pro snadnější a uživatelsky přívětivější tvorbu programů. Cílem této práce je zjednodušení procesu tvorby programů na zařízení IoT pro koncového uživatele vytvořením nového nástroje vizuálního programování. Výsledkem práce je nová knihovna v jazyce JavaScript implementující vizuální programovací jazyk a grafický editor, jehož výstupem je serializovaná podoba programu, která může být následně transformována do jazyka cílových zařízení IoT.

Teoretická část této práce je rozdělena do dvou kapitol. Kapitola 2 popisuje základní principy a technologie zařízení IoT. V kapitole 3 je popsán koncept vizuálního programování a proveden rozbor několika současných nástrojů vizuálního programování.

Kapitolou 4 začíná praktická část práce, kde jsou analyzována možná řešení programování zařízení IoT. V kapitole 5 je popsán návrh řešení v podobě knihovny, jejíž součástí je grafický editor. Kapitola 6 popisuje detaily řešení implementace knihovny. Uživatelské testování grafického editoru popisuje kapitola 7. Závěrečná kapitola 8 shrnuje dosažené výsledky a uvádí návrhy budoucích rozšíření knihovny.

Kapitola 2

Internet věcí

Pojem internet věcí (anglicky *Internet of Things*, zkráceně *IoT*) může být definován jako síť fyzických zařízení s integrovanými senzory, akčními členy a softwarem pro zpracování a sdílení dat, která jsou připojena k internetu. Nemusí se nutně jednat o speciálně navržená zařízení, nýbrž o věci, které používáme na denní bázi jako například chytrý telefon, pračka, varná konvice nebo auto. Tato zařízení mohou být propojena jak mezi sebou, tak i s *cloudem* pomocí různých síťových technologií, protokolů a standardů. Cloud je síť serverů, které jsou dostupné přes internet po celém světě. Tyto servery poskytují funkcionalitu ve formě různých aplikačních rozhraní a databází. Je potřeba zmínit, že zařízení IoT nutně nemusí cloud využívat a mohou tvořit pouze lokální nebo menší uzavřené sítě. Tento přístup se obecně označuje jako *IoT Edge*.

Smyslem IoT je usnadnit, zautomatizovat a zefektivnit naše každodenní problémy. IoT je využíváno v různých odvětvích. Některými z nich jsou například:

- **Zdravotnictví** – pacienti jsou vybaveni chytrými zařízeními (chytré náramky, kardiostimulátory), které monitorují jejich životní funkce a při zhoršení stavu mohou upozornit pacienta a ošetřujícího lékaře. V nemocnicích a lékárnách je také potřeba některé typy léků skladovat pod určitou teplotou nebo vlhkostí. K tomuto účelu mohou být použita chytrá čidla pro měření požadovaných hodnot. Včasná detekce nedovřených dvířek od skříně s léky nebo překročení teploty v místnosti pak může značně minimalizovat ztráty.

- **Zemědělství** – IoT může sloužit například jako monitorovací systém vlhkosti půdy, okolní teploty, počasí nebo jako automatizovaný systém zavlažování. Monitoring těchto hodnot poskytuje uživatelům důležitá data a tím může přispět ke zvýšení produkce.

- **Doprava** – v dopravě se zařízení IoT mohou využít k automatizaci řízení silničního provozu pomocí světelných semaforů, které na základě dostupných dat rozhodnou o nejlepším řešení situace. Jejich chování je například ovlivněno počtem aut na křižovatce nebo prioritou konkrétního vozidla (policie, záchranná služba). Dalším z praktických využití je sledování stavu parkoviště a poskytnutí informace o volných místech řidičům.

- **Chytrá domácnost** – v chytré domácnosti se klasická zařízení jako zvonek u dveří, termostat, osvětlení nebo žaluzie stávají chytrými. Většinou jsou připojena k centrálnímu *hubu* a uživateli je poskytnuta možnost jejich nastavení, automatizace a ovládání. Jako konkrétní příklad lze uvést automatické roztažení žaluzií a zapnutí

kávovaru při ranním vstávání nebo přijetí upozornění o pohybu od chytrého zvonku, společně s živým přenosem prostoru před vchodovými dveřmi. Chytrá domácnost je také vybavena senzory, které mimo jiné slouží k dlouhodobému sběru informací, na základě kterých se snaží optimalizovat spotřebu energií v domě a tím snížit jeho náklady.

Počet zařízení IoT se v průběhu let neustále zvyšuje. Již v roce 2008 přesáhl počet zařízení připojených k internetu celosvětovou populaci [11]. Podle platformy Statista[1], která se již 16 let zabývá sběrem a zpracováním dat z různých odvětví bude v roce 2030 dostupných odhadem 29 biliónů zařízení IoT. Jelikož vnikají rozsáhlé systémy s velkým počet propojených zařízení IoT vyvstává zde hned několik požadavků, které by IoT mělo splňovat [30].

Prvním z požadavků je *identifikace a rozšiřitelnost*. Každé zařízení IoT by mělo být snadno identifikovatelné pomocí unikátního ID nebo adresy. Toto platí i pro zařízení, která se v budoucnu mohou do systému přidat. Zde je možno narazit na problémy spojené s generováním unikátních identifikátorů nebo systémem adresování v počítačových sítích.

Dalším požadavkem je *autonomie*. Zařízení IoT vyžadují alespoň základní soběstačnost ve formě automatického přizpůsobení chování na základě získaných dat. Zařízení by měla být schopna inteligentně vyhodnotit situace bez zásahu uživatele.

Interoperabilita je u zařízení IoT nutná. Jedná se o schopnost spolupráce a sdílení dat mezi jednotlivými zařízeními. Tímto mohou zařízení využívat služeb a získaných dat dostupných u jiných typů zařízení. Pokud by například došlo k překročení vodní hladiny u retenční nádrže, tak zařízení se senzorem vyšle zprávu zařízení, které je vybaveno akčním členem pro ovládání stavu čerpadla a tím zastaví další přítok vody.

Správa dat je důležitým aspektem IoT. Čím větší je počet zařízení IoT v systému, tím větší je celkový objem získaných dat. Proto je nutné navrhnout vhodný systém pro jejich zpracování a uložení. Pokud se jedná o rozsáhlý systém, který má být v budoucnu rozšiřitelný, tak se jako nejlepší řešení jeví využití *cloudových služeb*, které mají téměř neomezenou možnost rozšiřitelnosti.

Bezpečnost a soukromí jsou klíčové pro adopci IoT. Pokud přenos citlivých dat a komunikace mezi zařízeními nebude dostatečně zabezpečená, tak je uživatelé nebudou chtít využívat. Zařízení IoT většinou fungují na bezdrátových technologiích a je tedy možné komunikaci mezi nimi zachytit. Implementace bezpečnostních vrstev do těchto technologií je u IoT náročná z důvodu omezení rychlosti přenosu dat nebo energetické náročnosti šifrovacích algoritmů.

Zařízení IoT jsou instalována v různých prostorách a podmínkách, kde nemusí být vždy zaručen přístup k elektrické energii, proto se při návrhu zařízení přihlíží na jejich *energetickou efektivitu a udržitelnost*. Zařízení mohou být napájena z elektrické sítě, akumulátorů, jednorázových baterií nebo také přírodními zdroji (solární panely, větrné turbíny). Existuje mnoho způsobů, jak u zařízení spotřebu energie zefektivnit (snížení výkonu procesoru, automatický režim spánku, omezení periody posílání dat), ale některé z nich přináší značné nevýhody a je proto mezi nimi potřeba najít jistou rovnováhu.

[1]Predikce růstu zařízení IoT, navštíveno 15.01.2024: https://www.statista.com/statistics/1183457/iot-connected-devices-worldwide/

2.1 Architektura IoT

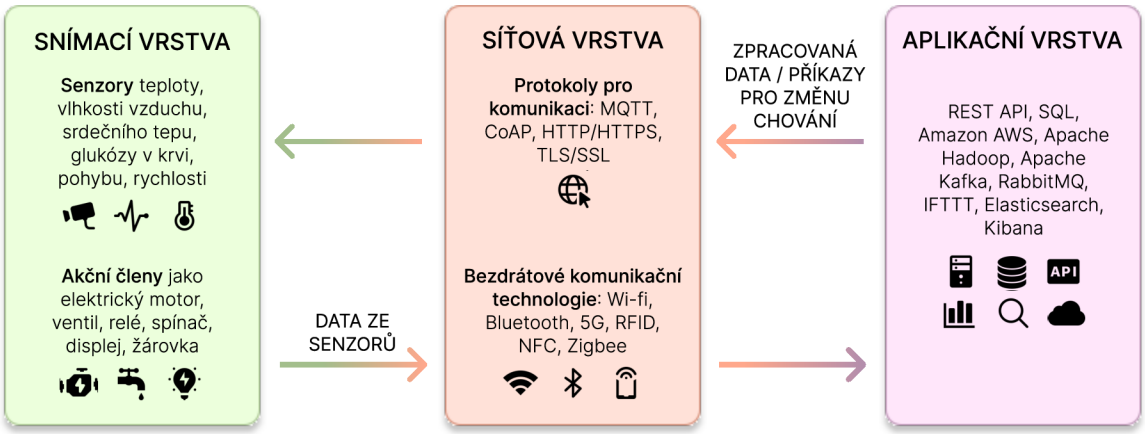
IoT trpí nedostatkem standardizace použitých technologií a protokolů. V průběhu let vzniklo několik pokusů o vytvoření a standardizaci architektury IoT [10, 11, 19]. Do dnešního dne však není jednoznačně definováno jakou architekturu a konkrétní technologie použít. Je otázkou, jestli se toto sjednocení vůbec někdy podaří vzhledem k různorodosti zařízení IoT a tvořených systémů. Architektura IoT by dle autorů [10] měla splňovat několik základních požadavků. Konkrétně se jedná o rozšiřitelnost, škálovatelnost, decentralizaci a zajištění komunikace mezi zařízeními. Architektury se obecně dělí na vrstvy, které zajišťují a zároveň rozdělují požadovanou funkcionalitu.

Za základní architekturu je považována třívrstvá architektura [1], která vznikla při počátcích výzkumu IoT. Z té následně vycházejí ostatní architektury, jako je například architektura navržená společností ITU (*International Telecommunication Union*) [11] nebo SoA (*Service-oriented Architecture*) [10], které přidávají zcela nové vrstvy nebo již existující rozdělují do více podvrstev. Na obrázku 2.1 lze vidět strukturu třívrstvé architektury společně s konkrétními příklady použitelných technologií pro každou vrstvu. Jak již název napovídá, třívrstvá architektura se skládá ze tří vrstev: *snímací*, *síťové* a *aplikační*.

Snímací vrstva obsahuje zařízení IoT, která jsou vybavena senzory a akčními členy různých typů. Senzory jsou fyzická zařízení, která snímají data (teplota, vlhkost, pohyb). Získaná data jsou následně předána do síťové vrstvy. Akční členy jsou prvky, které reagují na podněty ze senzorů nebo externí příkazy tím, že převedou elektrickou energii na mechanickou a provedou tak akci ovlivňující fyzický svět (motor, ventil, relé, displej).

Síťová vrstva slouží jako prostředník mezi snímací a aplikační vrstvou. Má za úkol tyto vrstvy propojit a umožnit bezpečný přenos dat. Využívá jak drátových, tak bezdrátových technologií a síťových protokolů.

Aplikační vrstva se stará o kolekci, zpracování (analýzu) a zobrazení dat. Do aplikační vrstvy spadá také business logika a řešení uchování dat.



Obrázek 2.1: Základní třívrstvá architektura IoT. Každá vrstva představuje jeden obdélník, v němž jsou uvedeny konkrétní příklady použitelných technologií. Šipky mezi vrstvami reprezenují typ zasílaných dat.

2.2 Přenos dat a komunikace zařízení IoT

Zařízení IoT využívají různé technologie a protokoly k přenosu dat a komunikaci. Níže je uveden přehled běžně používaných technologií pro komunikaci a sdílení dat rozdělen do kategorií s ohledem na jejich charakteristiky a využití.

Bezdrátové technologie krátkého dosahu

- **RFID** – *Radio Frequency Identification* je technologie sloužící k výměně dat na relativně krátkou vzdálenost. Skládá se ze čtecího zařízení a paměťového média (čtečky a tagu). Dosah komunikace RFID závisí na typu tagu (aktivní/pasivní) a na použitém frekvenčním pásmu (nízkofrekvenční – do 10 cm, vysokofrekvenční – do 1 metru, ultravysokofrekvenční – do několika desítek metrů). Čtečka je zařízení, které pomocí antény vysílá rádiové signály. Tyto signály jsou následně zachyceny tagem, který disponuje nízkokapacitním úložištěm kde jsou uložena potřebná data. Technologie RFID se nejčastěji používá k identifikaci zařízení IoT nebo obecně fyzických objektů. Technologie čárových kódů se využívá ke stejnému účelu a s největší pravděpodobností bude v budoucnu nahrazena právě technologií RFID [30]. Konkrétní použití RFID je například v maloobchodech ke sledování zboží, omezení krádeží a automatizaci pokladních systémů nebo u chytrých zámků, které využívají RFID tag místo klasického klíče.

- **NFC** – technologie NFC (*Near Field Communication*) vychází z RFID. Liší se v dosahu komunikace, ten je menší než u RFID (zhruba do 5 cm) a frekvencí přenosu (13.56 MHz). NFC také narozdíl od RFID umožňuje obousměrnou komunikaci dvou zařízení (*Peer-to-Peer*). Často se využívá v mobilních platebních systémech, kde může sloužit jako prostředek pro bezkontaktní platby.

- **Bluetooth** – umožňuje bezdrátovou komunikaci na krátkou vzdálenost mezi více zařízeními. Bluetooth rozhraní se většinou používá při komunikaci mezi chytrými zařízeními jako jsou chytrá světla, termostaty, reproduktory, hodinky a chytrými telefony nebo jinými ovládacími jednotkami. Existují různé verze tohoto standardu. V době tvorby této práce je nejnovější Bluetooth ve verzi 5.3. V IoT se taktéž využívá varianta BLE (*Bluetooth Low Energy*), protože umožňuje energeticky efektivnější komunikaci.

- **WiFi** – *Wireless Fidelity* je široce rozšířená skupina protokolů založených na standardech IEEE 802.11, která zajišťuje bezdrátové připojení k lokální síti a internetu. Pro přenos signálu se většinou využívá šířka rádiového pásma 2,4 GHz nebo 5 GHz a rychlost přenosu může teoreticky dosahovat až 9,6 Gbit/s [34].

- **ZigBee** – jedná se o bezdrátový protokol navržen primárně pro použití v IoT, který vychází ze standardu IEEE 802.15.4 a využívá různá frekvenční pásma (2,4 GHz, 915 – 921 Mhz, 868 Mhz). ZigBee pomocí síťové topologie typu *Mesh* propojuje jednotlivá zařízení IoT mezi sebou a tvoří tak PAN (*Personal Area Networks*), to má za následek lepší stabilitu sítě a možnost zotavit se z poruch[2]. Mezi jeho hlavní výhody patří nízká energetická náročnost, malá latence (vhodné pro systémy reálného času) a bezpečnost [15].

[2]Oficiální stránky Zigbee Alliance: https://csa-iot.org/all-solutions/zigbee/zigbee-faq/

Bezdrátové technologie dlouhého dosahu

- **LoRa a LoRaWAN** – LoRa (*Long Range*) je fyzická vrstva bezdrátové technologie vyvinuté pro energeticky efektivní přenos dat na velké vzdálenosti. Tato technologie byla dále standardizována a rozšířena přidáním síťového komunikačního protokolu LoRaWAN (*Long Range Wide Area Network*). Díky energetické efektivitě dosažené použitím hvězdicové topologie společně s nízkou rychlostí přenosu dat, dlouhému dosahu (jedna brána dokáže pokrýt oblast o rozloze až 100 km^2) a robustnímu zabezpečení (šifrování pomocí algoritmu AES) je ideální volbou pro využití v IoT [7].

- **SigFox** – bezdrátová komunikační technologie, která byla navržena pro přenos krátkých zpráv na velkou vzdálenost. Od roku 2022 je vlastněna společností UnaBiz[3], která zajišťuje provoz globální sítě (*0G network*) a poskytuje tak jednotné řešení pro připojení zařízení po celém světě. Hlavními výhodami použití tohoto řešení jsou nízké nároky na energii, jednoduchost konfigurace, nákladová efektivita a možnost použití s ostatními technologiemi jako WiFi, Bluetooth nebo LoraWAN[4].

Síťové protokoly pro komunikaci IoT

Existuje mnoho protokolů pro komunikaci, které lze v IoT využít, jako například protokol MQTT (Message Queuing Telemetry Transport), HTTP (Hypertext Transfer Protocol), CoAP (Constrained Application Protocol), AMQP (Advanced Message Queuing Protocol) nebo DDS (Data Distribution Service) [30]. Mezi ty nejpoužívanější se však řadí protokoly MQTT a HTTP(S).

HTTP (*Hypertext Transfer Protocol*) je aplikační protokol typu klient – server a využívá se pro přenos informací přes internet. Mezi jeho výhody patří široká adopce, standardizace a interoperabilita. Je bezstavový a podporuje webovou REST (*Representational State Transfer*) architekturu. Funguje na principu posílání požadavků a přijímání odpovědí. Protokol HTTP poskytuje soubor metod pro různé typy operací:

- **GET** – pro získání dat,
- **POST** – pro odeslání dat na server,
- **PUT** a **PATCH** – pro aktualizaci existujících dat,
- **DELETE** – pro smazání dat.

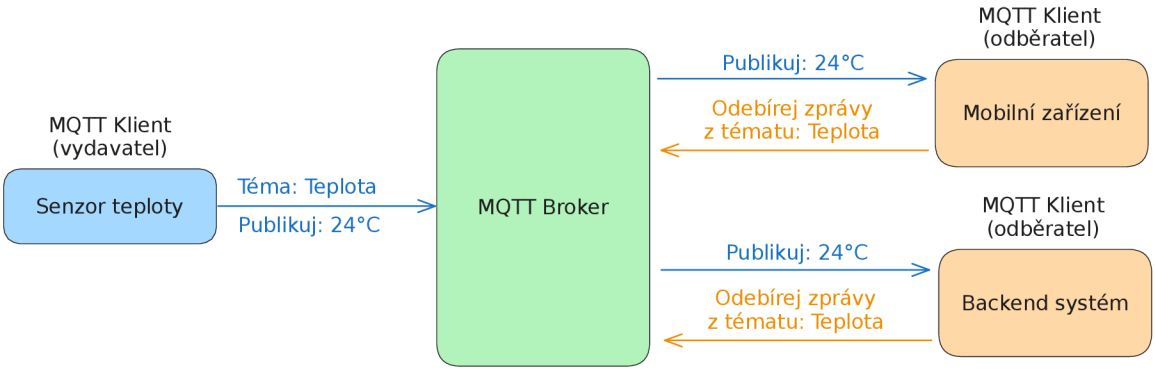
Formát posílaných dat není pevně stanoven, a tak HTTP v ohledu přenosu dat nabízí relativní volnost. Většinou se však používá nějaká serializovaná podoba dat ve formě HTML, JSON nebo XML. Pro zabezpečení komunikace se společně s HTTP využívá protokol TLS (*Transport Layer Security*).

Nevýhody využití protokolu HTTP v IoT jsou hlavně v jeho vysoké energetické náročnosti na koncová zařízení IoT, velikost zpráv nebo latenci při přenosu. Při srovnání několika protokolů se ukázalo, že HTTP byl z pohledu energetické náročnosti nejméně efektivní. HTTP se také na rozdíl od MQTT standardně nezabývá zajištěním QoS (*QoS – Quality of Service*) a spolehlivostí přenosu [22].

[3] Článek o akvizici firmy Sigfox společností UnaBiz: https://www.unabiz.com/unabiz-appointed-new-owner-sigfox-sa-sigfox-and-france-sas/

[4] Oficiální stránky technologie SigFox: https://www.sigfox.com/what-is-sigfox/

MQTT (*Message Queuing Telemetry Transport*) je komunikační protokol vytvořen v roce 1999 Dr. Andy Stanfordin-Clarkem z IBM a Arlenem Nipperem z Arcom (nyní Eurotech). Byl navržen pro odlehčenou M2M (*Machine-to-Machine*) komunikaci založenou na architektuře publikování (*publish*) a odebírání (*subscribe*), která je znázorněna na obrázku 2.2. Zabezpečení protokolu probíhá pomocí TLS/SSL. V protokolu jsou rozlišeny dva typy zařízení: klient a broker [22, 30].



Obrázek 2.2: MQTT architektura typu vydavatel/odběratel. Inspirováno [21].

Klient může být vydavatelem (*publisher*) a/nebo odběratelem (*subscriber*). Vydavatel publikuje (odesílá) zprávy na specifické téma (*topic*), které je dostupné na brokeru. Odběratel odebírá jednotlivá témata a tím je připraven přijímat zprávy, které na toto téma publikují vydavatelé. Broker zodpovídá za příjem, zpracování a distribuci příchozích zpráv všem odběratelům daného tématu.

MQTT na rozdíl od HTTP také zajišťuje tři úrovně kvality služeb[5]:

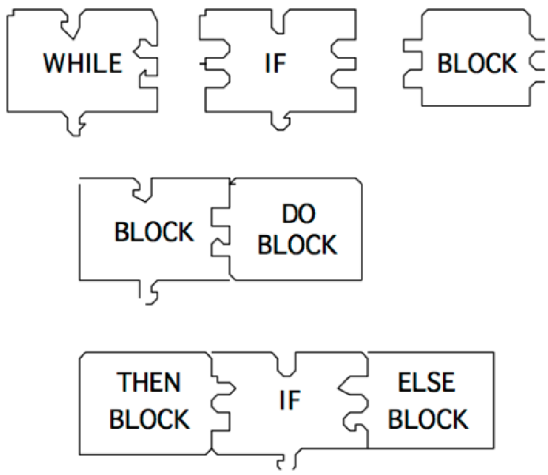
- **QoS 0** – v nejnižší úrovni klient při poslání zprávy neočekává od brokeru žádné potvrzení. Klient tak ani nemusí původní zprávu ukládat, protože se vždy pošle maximálně jednou a ne opakovaně.
- **QoS 1** – tato úroveň zajišťuje doručení zprávy k brokeru alespoň jednou. Když klient odešle zprávu, nechá si její kopii pro případné opakované odeslání do doby, než dostane od brokeru v odpovědi potvrzovací paket. Pokud potvrzovací paket klient nepřijme v rozumném čase, tak odešle původní zprávu znovu.
- **QoS 2** – nejlepší spolehlivost přenosu dat zajišťuje tato úroveň. Musí být zaručeno, že každá zpráva se pošle přesně jednou. Tohoto se dosáhne za pomocí techniky *four-part handshake*.

[5] https://www.hivemq.com/blog/mqtt-essentials-part-6-mqtt-quality-of-service-levels/

Kapitola 3

Vizuální programování

V 90. letech 20. století se v programování rozšířil nový trend v podobě grafické vizualizace programů. Hlavním důvodem tohoto rozšíření byla snaha vyřešit dlouhotrvající problém nalezení způsobu, jak efektivně zajistit komunikaci mezi uživatelem a počítačem [35]. Vizuální programování (VP) umožňuje uživateli vytvářet programy pomocí grafických elementů místo textové reprezentace [13]. Tento přístup uživateli značně ulehčuje pochopení konceptu programování, protože poskytuje vyšší vrstvu abstrakce. VP tak odbourává prvotní bariéru komplexity klasického programování. Uživatel se ze začátku tudíž nemusí zabývat přesnou syntaxí, sémantikou a složitějšími koncepty jako u klasických programovacích jazyků. Nejen z tohoto důvodu je VP vhodné pro začátečníky a je hojně využíváno ve vzdělávání [16, 26, 27, 28], ale i v dalších odvětvích jako je internet věcí (IoT), herní průmysl, multimédia nebo simulace. Postupně vznikají vizuální programovací jazyky (VPL) a nástroje, které je využívají. Jedním z nejranějších příkladů VPL je jazyk Blox Pascal (viz. obrázek 3.1), který využívá metodu skládanky pro reprezentaci syntaktických pravidel jazyka.



Obrázek 3.1: Vizuální programovací jazyk Blox Pascal z roku 1984. Dílky reprezentují konstrukce jazyka. Tvar dílků pevně určuje systém jak do sebe dílky mohou zapadat a kombinovat se mezi sebou (definuje syntax jazyka) [27].

VP má oproti klasickému programování velkou výhodu v uživatelské přívětivosti. Nástroje VP poskytují uživateli předdefinované konstrukce jazyka, které lze jen skládat do sebe

a tím vytvořit výsledný program, místo toho aby uživatel musel hledat co za konstrukce jsou dostupné a jak je použít. Ve většině případů je také uživateli poskytnuta interaktivní forma nápovědy, která popisuje funkcionalitu konstrukcí a jak je lze využít v konkrétních případech. Nástroj nápovědy se tedy stává zjednodušenou formou dokumentace jazyka, což je pro uživatele lepší nežli zdlouhavé pročítání běžné dokumentace. Díky tomu se i běžný uživatel může zapojit do procesu vývoje softwaru. Převážně se jedná o jednodušší programy, které jsou specifické a řeší konkrétní problém uživatele. VP ale také může sloužit jako rychlý prototypovací nástroj pro zkušenější programátory.

Problémy, které se VP snaží řešit můžeme rozdělit do 3 kategorií [27]:

- **Syntaktické** – syntaxe jazyka je korektní pořadí/uspořádání jazykových konstrukcí v programu. Nástroje VP poskytují předdefinované konstrukce a pravidla, jak do sebe mohou zapadat. Například jazyk Blockly[1] využívá metodu skládanky, kde je vizuálně odlišeno (pomocí výřezů v dílcích) jaké bloky (dílky skládanky) mohou být kam přiřazeny. Snižuje se tak chybovost syntaxe při vytváření programu, protože uživateli zkrátka není dovoleno konstrukce použít na místech, kde nepatří.

- **Sémantické** – sémantika jazyka definuje, jaký je význam konstrukcí jazyka. Ve VP je sémantika reprezentována jako odlišení konstruktů jazyka například podle barev, tvarů, ikon atd. Do sémantiky VP se také řadí způsob propojení a zanořování konstrukcí společně s hierarchií programu.

- **Pragmatické** – zaměřuje se na praktické pochopení vytvořených programů a jejich zasazení do kontextu konkrétních situací.

Vizuální programování sice poskytuje nástroje a způsoby zjednodušení konceptů programování, stále se ale jedná o programování. Logickým chybám či nedorozuměním se běžný uživatel nedokáže úplně vyvarovat. Šance, že dojde k chybě je ještě násobena komplexností tvořeného programu. Samotné nástroje a VPL se také potýkají s množinou problémů. Podle studie z roku 2017 [26] je podstatným problémem VPL rozšiřitelnost. VPL poskytují uživateli pouze limitovanou množinu operací, kterou může v programu použít. V praxi je v některých případech velmi těžké (někdy dokonce až nemožné) naprogramovat chtěnou funkcionalitu. Obzvlášť v oblasti IoT, kde je rozšíření funkcionality programu klíčové. Špatná optimalizace vygenerovaného kódu je další z problémů. Jelikož VPL využívají vysokou vrstvu abstrakce, je pro koncového uživatele těžké výsledný kód optimalizovat, jelikož nad ním nemá potřebnou kontrolu a samotné VPL optimalizaci často opomíjí.

Každý uživatel nad vytvářením programů uvažuje jiným způsobem. Záleží na jeho zkušenostech a specializaci v dané doméně. Dosud nebyl zdokumentován žádný společný „myšlenkový model“, kterým by se řídili všichni. Z hlediska vývoje VPL je proto složité vymyslet jedno rozhraní, které by mohl používat kdokoliv.

3.1 Vizuální programovací jazyky

Vizuální programovací jazyk (anglicky *visual programming language* – VPL) je typ programovacího jazyka, který využívá grafické elementy pro reprezentaci příkazů programu [26]. Každý VPL má svůj grafický systém, kterým zobrazuje konstrukce jazyka. Mezi populární systémy patří např. zobrazení příkazů jako bloků, které se do sebe mohou zanořovat nebo systém, který se podobá orientovaným grafům (vizuální bloky propojeny šipkami). Většina

[1] https://developers.google.com/blockly

VPL se snaží o vyšší vrstvu abstrakce z důvodu zrychlení a usnadnění vývoje programu. Při vývoji nového VPL je klíčové správně identifikovat koncového uživatele, jeho schopnosti a potřeby.

Pro lepší porozumění klasifikace VPL je nutno pochopit hlavní dělení programovacích jazyků. Programovací jazyky se dělí na dva hlavní typy:

- **Imperativní programovací jazyky** přesně definují jednotlivé příkazy, které se mají vykonat a v jakém pořadí. Pomocí nich se poté dojde ke kýženému výsledku. Zástupci imperativních jazyků jsou např. C, C# nebo Python.
- **Deklarativní programovací jazyky** definují, co se má udělat, a ne jak se to má udělat. Mezi nejběžnější zástupce patří jazyk SQL, kde je definováno, jaká data se z databáze mají získat, a ne jak se samotná data získají (o tento proces se již stará jazyk SQL a jeho konkrétní implementace). Za zmínku také stojí většina aplikačních rámců v jazyce JavaScript pro webový vývoj jako React, Vuejs nebo Angular.

Toto hlavní rozdělení platí i pro VPL, avšak tyto dva přístupy se v konkrétních VPL většinou kombinují a vzájemně se doplňují. Podrobněji mohou být dle Kuhail a spol. [16] VPL klasifikovány do čtyř hlavních kategorií:

- **Blokové programovací jazyky** (*Block-based*) – blokové programovací jazyky poskytují uživateli grafické elementy ve formě bloků. Tyto bloky se poté dají skládat a zanořovat do sebe podle pravidel definovaných daným nástrojem. Bloky představují pevně definovanou strukturu a zároveň formu abstrakce. Obě tyto vlastnosti snižují komplexitu pro koncového uživatele. Zástupcem je například jazyk Blockly (viz. sekce 3.3).
- **Ikonové programovací jazyky** (*Icon-based*) – jak je již z názvu patrné tyto jazyky ve velké míře využívají ikony a obrázky k reprezentaci objektů a akcí. Hodně z těchto jazyků také využívá konceptu *Trigger action programming* (TAP), který je podrobněji popsán v podsekci 3.2. MicroApp je jedním ze zástupců ikonových jazyků. Jak ikony MicroApp využívá je vidět na obrázku 3.2. Jedná se o mobilní aplikaci, jejíž myšlenka je podobná jako u nástrojů Puzzle nebo EUCalipTool a to zpřístupnit koncovému uživateli nástroj, který mu poskytne intuitivní rozhraní pro tvorbu jednoduchých generických programů [9].
- **Formulářové programovací jazyky** (*Form-based*) – u těchto jazyků uživatel doplňuje informace do formulářů, které mohou být přesouvány metodou *drag and drop*. Doplňování informací do formulářů může být pro uživatele více omezeno (výběr z předdefinované nabídky) nebo volnější (textový vstup). To již záleží na konkrétním VPL. Čistě formulářové VPL jsou v současnosti používány jen zřídka a spíše je jejich koncept využíván a kombinován s ostatními typy VPL. Mezi nástroje, které využívají konceptu formulářových jazyků by se dal zařadit například tabulkový editor MS Excel[2], i když se dle jeho definice vyloženě nejedná o nástroj vizuálního programování.
- **Diagramové programovací jazyky** (*Diagram-based/Flow-based*) – také známé jako *Data flow* jazyky propojují grafické objekty pomocí čar a šipek. Propojení objektů poté reprezentuje určitý vztah mezi nimi (vstup/výstup). Konečný program tvoří posloupnost propojených objektů, jejichž funkcionalita se postupně vykonává (flow).

[2] https://www.microsoft.com/cs-cz/microsoft-365/excel

Zástupci: Node-Red (IoT)[3], Unreal Engine - Blueprints (herní vývoj)[4], LabView (simulace)[5].

Obrázek 3.2: Příklad využití ikon v aplikaci MicroApp [9]

3.2 Vývoj softwaru koncovým uživatelem

Přístup, kdy jsou koncovému uživateli poskytnuty nástroje, metody a techniky pro tvorbu softwarových aplikací je označován jako *End-user development* (EUD). Jedná se o velmi rozsáhlé téma, které v sobě zahrnuje mnohem více než VP. Kromě EUD existují další dva vzájemně se prolínající koncepty [5].

- **End-User Programming** (*EUP*) – zaměřuje se na samotný proces programování, konkrétní nástroje a techniky.
- **End-User Software Engineering** (*EUSE*) – řeší celkový návrh softwaru, jeho udržovatelnost a kvalitu kódu vytvořeného koncovým uživatelem.

EUD se v nedávné době rozšířilo zejména kvůli vyšší poptávce než nabídce profesionálních programátorů. Kuhail a spol. [16] uvádí poměr běžných uživatelů a profesionálních programátorů 30:1. Koncový uživatel se v dané doméně orientuje nejlépe a uvědomuje si souvislosti, které se jen stěží vysvětlují a sepisují do formálních požadavků. Jelikož je koncový uživatel považován za člověka se základními technickými znalostmi, tak pro něj není možné vyvíjet software stejným způsobem jako profesionální programátor. Je tedy žádané mu poskytnout nástroje, které mu umožňují vývoj bez hlubší znalosti konceptu programování [6, 9, 16].

Další motivací pro použití EUD je pomalý cyklus vývoje profesionálního softwaru, kvůli neustále se měnícím požadavkům na výsledný systém. Koncový uživatel nejprve musí své požadavky formálně sepsat a následně předat programátorovi. Forma komunikace s programátorem také hraje velkou roli v rychlosti vývoje. Tento proces musí koncový uživatel opakovat pokaždé když se jeho požadavky změní. EUD poskytuje koncovému uživateli různé způsoby, kterými on sám může požadovanou funkcionalitu implementovat a následně upravovat. Přenáší tím na uživatele větší míru zodpovědnosti za cenu rychlejšího vývoje a přizpůsobitelnosti systému.

Mezi nejvíce používané způsoby EUD patří právě vizuální programovaní, které již bylo popsáno výše. Dalším zajímavým způsobem EUD je programování ukázkou (*programming-by-demonstration*). Uživatel postupně demonstruje jednotlivé kroky, které by výsledný program měl sám provádět a tím definuje jeho chování. Některé systémy tohoto typu dokáží

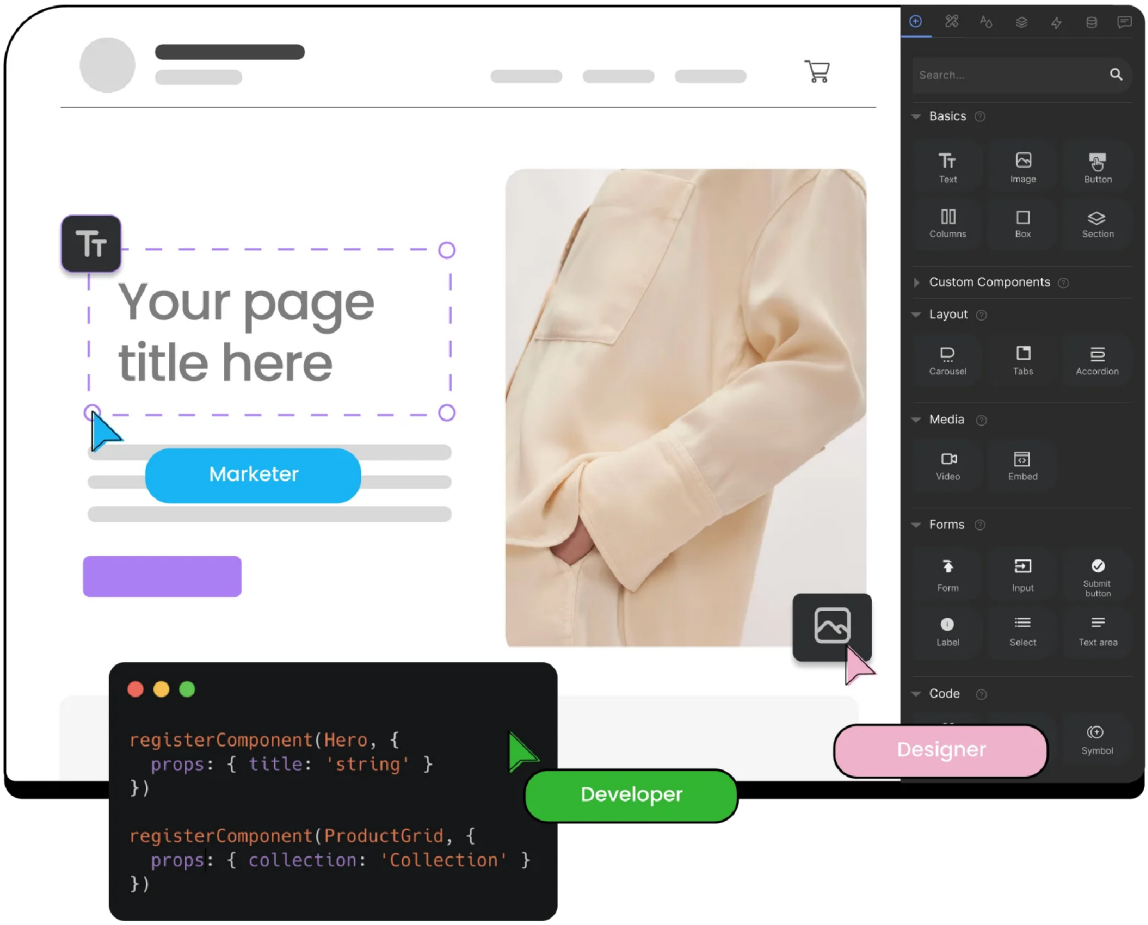
[3] https://nodered.org/

[4] https://docs.unrealengine.com/5.3/en-US/introduction-to-blueprints-visual-scripting-in-unreal-engine/

[5] https://www.ni.com/en/shop/labview.html

implicitně odvodit celkové chování programu, jiné pouze část a zbytek musí uživatel definovat ručně [23].

Některé nástroje EUD se zaměřují i na kolaboraci mezi profesionálním programátorem a koncovým uživatelem s doménovými znalostmi. Jedním z moderních nástrojů tohoto typu je vizuální redakční systém, vyvíjen společností Builder.io[6], který umožňuje spolupracovat různým uživatelům (např. grafický designér nebo marketingový manažer) s programátory. Pomocí grafického editoru, který se podobá nástrojům pro tvorbu UI designu a mockupů si uživatel tvoří grafické komponenty (viz. obrázek 3.3). Jednoduchou metodou *drag and drop* může komponenty přesouvat a seskupovat. Pro komponenty je poté automaticky vygenerován výsledný kód v jazyce HTML, CSS a případně i JavaScript. Dostupná je i kompatibilita s frontend rámci (React, Vue, Svelte). Nástroj také umožňuje propojení s vývojovým prostředím a dokáže automaticky synchronizovat vygenerovaný kód. Builder.io navíc ještě do svých produktů integruje různé formy umělé inteligence, která dále usnadňuje vývoj.

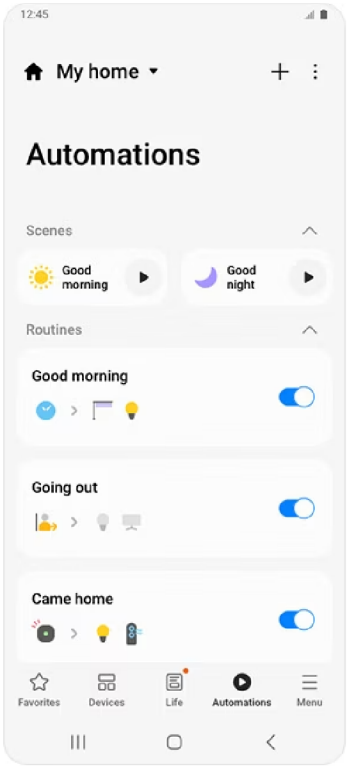


Obrázek 3.3: Ukázka vizuálního redakčního systému od firmy Builder.io[7].

[6] https://www.builder.io/m/visual-headless-cms
[7] https://builder.io/

Trigger action programming

Trigger action programming (TAP) je forma EUD nejvíce používaná při programování zařízení IoT v chytré domácnosti a automatizace. Ve zkratce se jedná o pravidla „pokud-toto-tak-něco“ (*if-this-then-that*), která definují podmínku/y a to, co se má stát, pokud je splněna. Příkladem reálného využití je aplikace SmartThings[8] od společnosti Samsung na obrázku 3.4, nebo platforma IFTTT[9], která propojuje různé distribuce zařízení IoT a umožňuje pravidla kombinovat mezi nimi. Uživatel si například pořídil chytrý zvonek s kamerou od společnosti Ring a chytrá světla HUE od společnosti Philips. Platforma IFTTT mu poté umožní definovat pravidlo, které zapne světla, když někdo zazvoní na zvonek. Toto by bez IFTTT platformy bylo jinak pro uživatele velmi složité kvůli uzavřenému ekosystému jednotlivých výrobců.



Obrázek 3.4: Automatizace chytrých zařízení v aplikaci SmartThings[10].

3.3 Současné nástroje vizuálního programování

Mezi nástroje VP se mohou řadit jak samotné vizuální programovací jazyky, tak grafické editory a prostředí pro jejich tvorbu. Nástroje pro VP se dělí do skupin podle různých kritérií. Kritéria samotná nejsou pevně definována a spíše vyplývají z průzkumů použití jazyků samotných. Do kritérií můžeme zařadit například: styl interakce s daným nástrojem, koncového uživatele (pro koho je nástroj primárně určen), cílovou doménu (automatizace,

[8] https://www.samsung.com/cz/apps/smartthings/
[9] https://ifttt.com/
[10] https://www.samsung.com/cz/apps/smartthings/

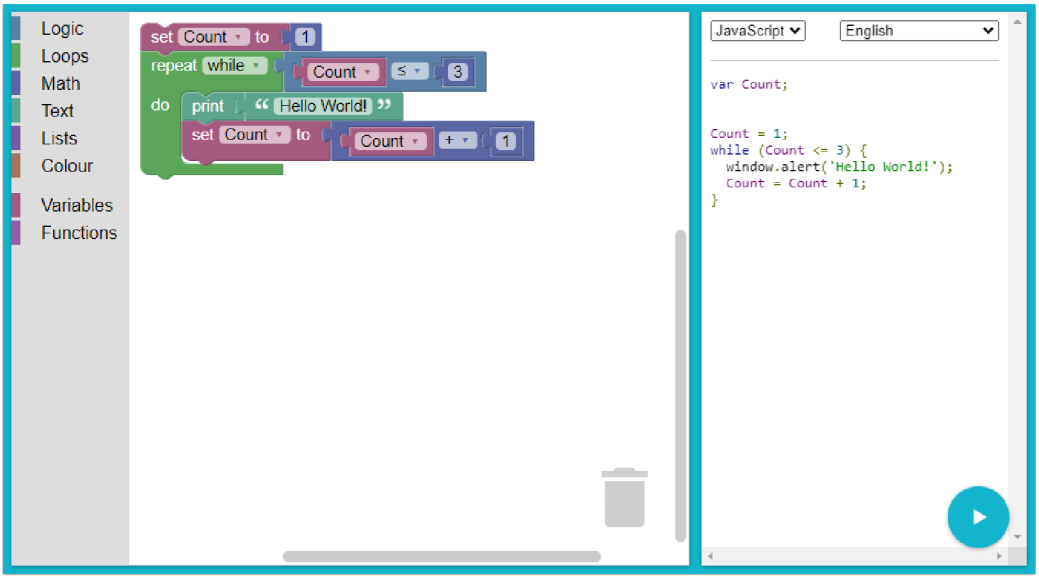
lékařství, vzdělávání, multimédia), platformu (webová aplikace, nativní aplikace) [16]. Použití VP silně závisí na typu určení a rozsahu výsledného systému. VP má svá úskalí a při návrhu systému je tedy nutné správně odhadnou jakou formou a zda jej vůbec použít. Níže jsou uvedeni zástupci VPL různých kategorií.

Blockly

Blockly je knihovna v jazyce JavaScript sloužící k tvorbě VPL. Byla vytvořena společností Google[11] a je distribuována pod svobodnou licencí Apache 2.0 [33]. Existují i verze knihovny pro operační systémy Android a iOS v podobě nativních aplikací[12]. V současnosti již ale nejsou aktivně vyvíjeny a je kladen důraz na použití webové verze.

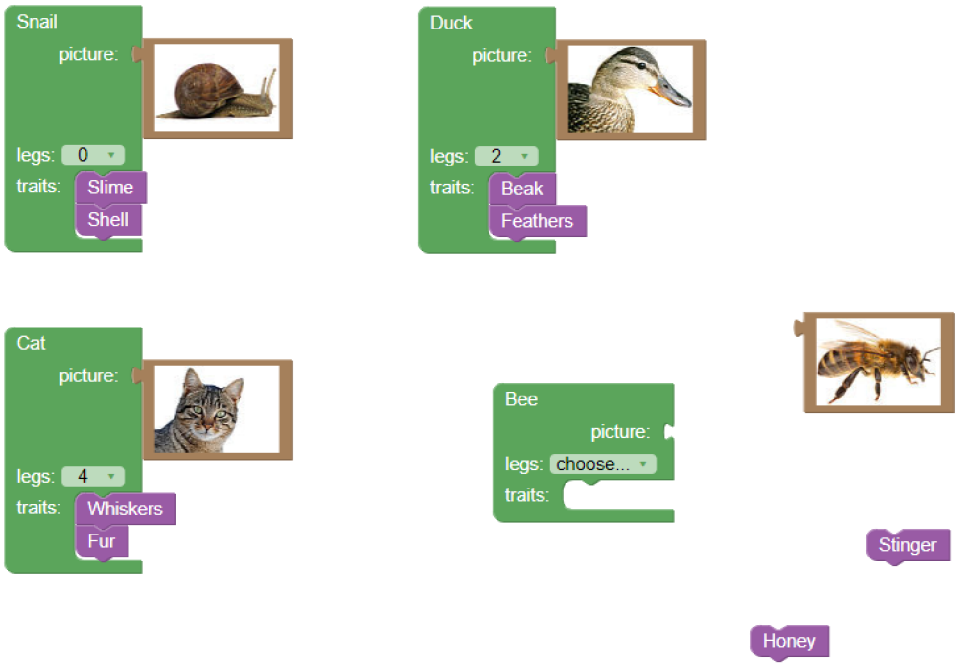
Jelikož se jedná o univerzální a rozšiřitelnou knihovnu, je využívána mnoha vývojáři pro tvorbu nových nástrojů a VPL. Většina z těchto nástrojů slouží pro edukativní účely (Scratch, Blockly Games, App Invertor), ale objevují se i implementace doménově specifických VPL jako například Smart Block [4] nebo Arduviz [25].

Součástí knihovny je i webový editor (viz. obrázek 3.5), který lze snadno integrovat do nové nebo existující webové aplikace. Na obrázku 3.6 lze vidět příklad integrace knihovny Blockly. Editor je rozdělen na pracovní plochu a vygenerovanou textovou podobu programu. Je podporováno generování kódu do jazyků JavaScript, Python, PHP, Lua a Dart. Vývojáři si také mohou vytvořit a integrovat své vlastní generátory do libovolného jazyka. Je ovšem potřeba zmínit, že Blockly pracuje primárně s dynamicky typovanými interpretovanými jazyky.



Obrázek 3.5: Webový editor nástroje Blockly. V levé části uživatel vytváří program na pracovní ploše (přidává a upravuje bloky). V pravé části je vygenerován výsledný kód vytvořeného programu v jazyce JavaScript[14].

[11] https://en.wikipedia.org/wiki/Google
[12] https://github.com/google/blockly-android, https://github.com/google/blockly-ios
[14] https://developers.google.com/blockly
[16] https://blockly.games/puzzle



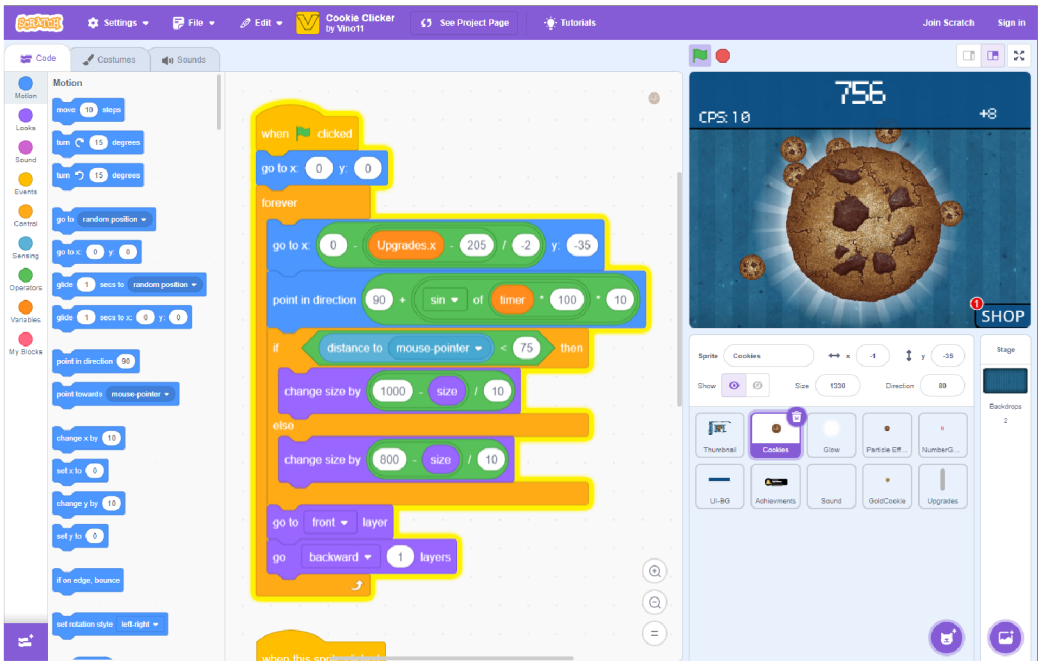
Obrázek 3.6: Jedna z interaktivních her nástroje Blockly Games. Hráč má za úkol korektně přiřadit obrázky a vlastnosti zvířat do odpovídajících bloků[16].

Pracovní plocha umožňuje uživateli vytvářet programy pomocí předdefinovaných bloků, které jsou rozděleny do kategorií a barevně odlišeny podle funkcionality (logické bloky, smyčky, matematické operace, atd.). Bloky se mohou vzájemně propojovat nebo zanořovat. Připomínají tak dílky skládačky. Tento koncept je využíván i v nástroji Puzzle [6] a značně redukuje vznik syntaktických chyb při vytváření programu. Uživatel totiž nemůže propojovat libovolné bloky, ale pouze ty, které jsou mezi sebou kompatibilní (ty které do sebe zapadají). Každý blok je unikátní podle funkcionality, kterou poskytuje. Například blok pro logický výraz má dvě volná místa pro vložení dalších bloků, mezi kterými se nachází nabídka s výběrem pro typ logického operátoru (a zároveň / nebo). V jiných blocích se bude výběrová nabídka lišit nebo jí například může nahradit textové pole pro zadání libovolného řetězce. Alexandr Repenning ve svém článku zmiňuje, že by uživateli měli být poskytnuty speciální nástroje na základě požadované funkcionality. Například pokud by uživatel měl zadat jako vstupní hodnotu barvu, měl by mu být poskytnut grafický nástroj pro výběr barvy, který mu umožní danou barvu zvolit a případně upravit. Uživatel by neměl být nucen zadávat řetězcovou reprezentaci kódu barvy [27].

Scratch

Scratch je postaven na knihovně Blockly a rozšiřuje její funkcionalitu. Cílí hlavně na děti (8 až 16 let) a programátorské začátečníky, kteří se chtějí naučit konceptům programování jednoduchou a zábavnou cestou [29]. Poskytuje webový grafický editor pro tvorbu jednoduchých her, příběhů a animací. Základní koncept bloků a manipulace s nimi je stejná jako u knihovny Blockly. Na rozdíl od ní není výstupem programu vygenerovaný kód, nýbrž grafická aplikace s různými elementy (postavami), jejichž chování bylo naprogramováno pomocí

editoru. Příklad výsledné aplikace je na obrázku 3.7. Uživatel má k dispozici také interaktivní nápovědu v podobě animovaných obrázků a videí, která ho krok po kroku provází při tvorbě programu.



Obrázek 3.7: Populární hra *Cookie Clicker* vytvořena za pomocí nástroje Scratch[17].

Jazyk Scratch se řídí motem: „škola hrou“ a díky jeho lokalizaci do více než 70 jazyků je hojně využíván jako učební nástroj ve školách po celém světě. Studenti si postupně osvojují základní principy programování a zdokonalují se v řešení problémů logickou cestou. Studie ukazují, že integrace vizuálního programování do výuky informatiky značně pomáhá k pochopení základních ale i složitějších konceptů programování. Když studenti pochopí základní principy, jako je dělení problémů na podproblémy a jejich abstrakce do funkcí, použití proměnných, podmínek, cyklů atd., je pro ně jednodušší přejít do klasického programovacího jazyka jako třeba C# nebo Java. Studenti nutně nemusí vědět přesnou syntax daného jazyka, ale při řešení problému vědí, jaké konstrukce a postup mají použít [2, 28]. Ukazuje se také, že vývojáři VPL by se místo řešení syntaxe jazyka měli detailněji zaměřit na problémy spojené s celkovým pochopením principů programování a chováním programů ve specifických scénářích [27].

Problém jazyka Scratch nastává při tvoření komplexnějších programů. R. Koitz a W. Slany [14] provedli studii, která se zaměřuje na řešení komplexity použití matematických výrazů v jazyce Scratch. Prezentují hybridní způsob tvorby a manipulace matematických formulí použitím kombinace textového a grafického přístupu. Při tvorbě složitějších matematických výrazů dochází k hlubokému zanořování bloků, což vede ke značné nepřehlednosti především na menších displejích mobilních zařízení. Pokud složené bloky výrazu přesáhnou velikost obrazovky, tak je uživatel nucen horizontálně rolovat přes kreslící plochu a vždy bude viditelná jenom část výrazu. Uživatelé, kteří byly součástí testování hybridního přístupu začali tvořit výrazy přepisem vzorců „zleva doprava“, aniž by se nejdříve podívali na

[17] https://scratch.mit.edu/projects/930655286/

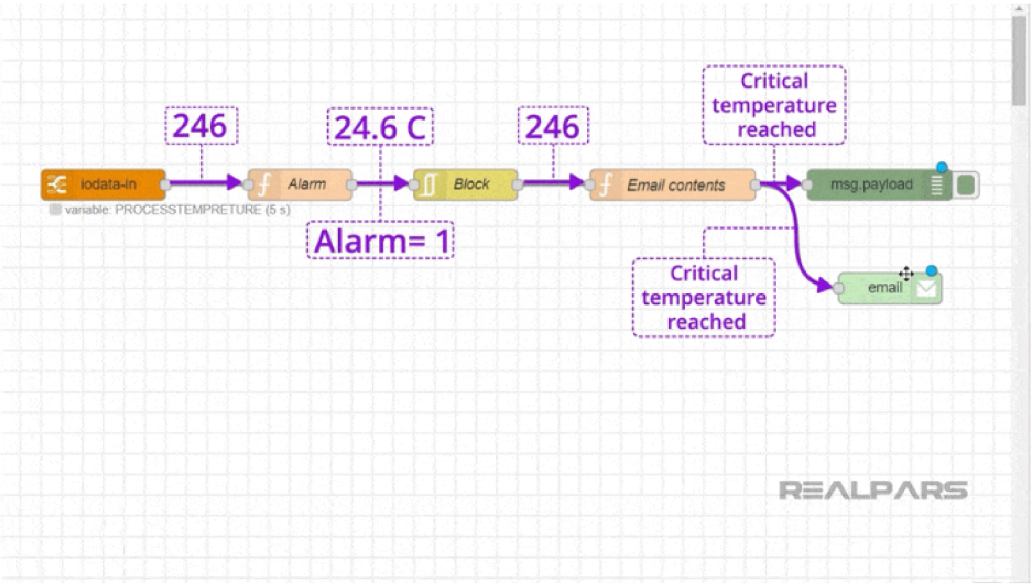
celou strukturu vzorce. V jazyce Scratch je tento přepis s použitím bloků složitější a uživatelé museli finální vzorec několikrát upravovat, aby se dostali ke správné podobě výrazu.

Textový přístup se snaží tyto problémy minimalizovat například zalamováním textu nebo obarvováním závorek částí výrazu. Dle výsledků experimentů této studie mělo zobrazení výrazů textovým přístupem pozitivní dopad na čitelnost a manipulaci s výrazy.

Node-RED

Node-RED spadá do kategorie Flow-based vizuálních programovacích jazyků (programování tokem dat). Je založen na webových technologiích (Node.js[18]) a jeho součástí je grafický editor určený pro webové prohlížeče. Byl vytvořen v roce 2013 vývojáři Nick O'Leary a Dave Conway-Jones. Později téhož roku byl projekt vydán pod Open source licencí Apache 2.0 [8].

Jedná se o univerzální nástroj sloužící pro propojení a komunikaci hardwarových zařízení, aplikačních programovacích rozhraní (API) a online služeb. Základním stavebním prvkem je uzel (*node*). Uzly jsou vzájemně propojeny cestami, které určují tok programu. Každý uzel má definovanou svou funkcionalitu a je reprezentován jako černá skříňka. Přijímá vstup ve formě dat (zpráv), ta následně modifikuje a výstup předá do dalšího uzlu. V praxi je nejvíce využíván k propojení a programování různých typů zařízení IoT. Typickým příkladem použití je monitoring hodnot senzoru a následné upozornění uživatele při změně nebo překročení hraniční hodnoty viz. obrázek 3.8.

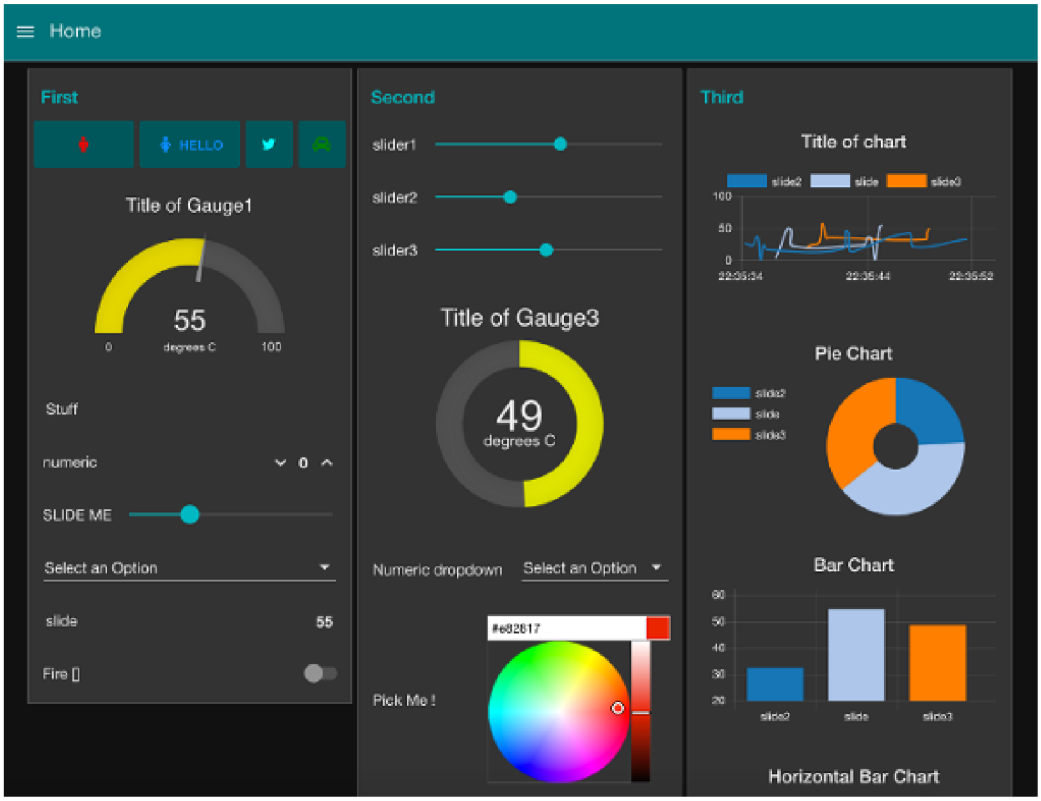


Obrázek 3.8: Program vytvořen v nástroji Node-RED, který zpracovává příchozí události z IoT zařízení pro monitorování teploty a při překročení kritické hranice zašle uživateli email s upozorněním[20].

Node-RED je atraktivní hlavně díky jeho flexibilitě a rozšiřitelnosti. Má velkou komunitu uživatelů a je stále aktivně vyvíjen. Podporuje vytváření doplňků třetích stran (uzly, toky a kolekce). Uživatelé tak mohou rozšiřovat a sdílet dodatečnou funkcionalitu. Modul „node-

[18] https://nodejs.org/
[20] https://www.realpars.com/blog-post/node-red

red-dashboard“ je momentálně nejvíce populární a přidává několik uzlů s možností vytvoření grafických pohledů pro sumarizaci dat v reálném čase (viz. obrázek 3.9).



Obrázek 3.9: Ukázka použití modulu node-red-dashboard[21].

Puzzle

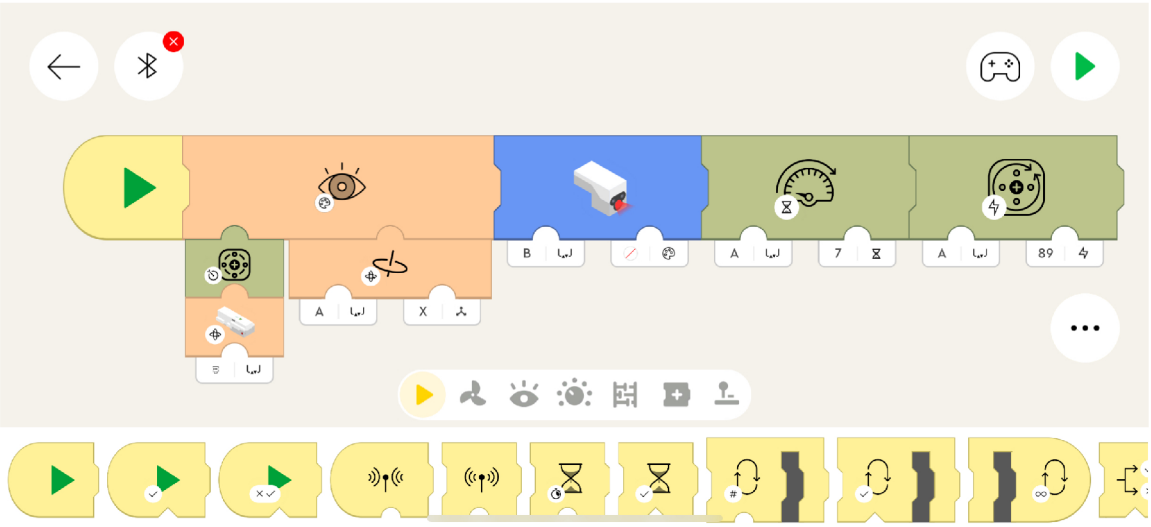
Nástroj Puzzle byl vytvořen za účelem umožnit koncovému uživateli vytvářet programy na mobilním zařízení, které interagují s chytrými zařízeními (IoT), funkcemi mobilního zařízení (fotoaparát, mikrofon) a webovými službami. Jak je již z názvu nástroje patrné, využívá metodu skládanky (*puzzle*). Konstrukce jazyka jsou reprezentovány jako dílky skládanky, které mají výřezy, podle kterých je možno dílky spojovat. Výřezy jsou taktéž označeny barevným indikátorem, který definuje, jaké dílky mohou být mezi sebou propojeny. Dílek s výřezem který má například modrý indikátor bude moct být spojen s jiným dílkem jehož výřez má indikátor stejné barvy. Použití stejných konceptů můžeme pozorovat i v aplikaci PoweredUp[22] od společnosti LEGO, která slouží k programování speciálních LEGO stavebnic s elektronickými součástkami jako jsou elektromotory, senzory, diody, reproduktory a další (viz. obrázek 3.10).

Grafické rozhraní nástroje Puzzle je rozděleno do tří hlavních sekcí: start, tvorba a prostředí pro provádění programu. Jednotlivé sekce a jejich použití lze vidět na obrázku 3.11. Sekce start zobrazuje seznam již existujících programů s možností vytvoření nového programu. Díky vyšší vrstvě abstrakce lze na programy nahlížet jako na samotné konstrukce

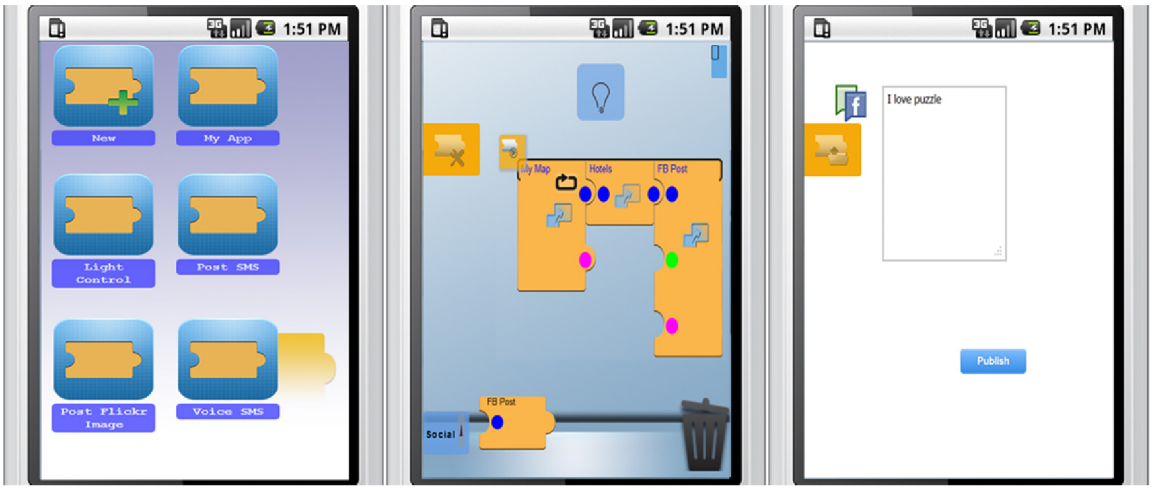
[21] https://flows.nodered.org/node/node-red-dashboard

[22] https://www.lego.com/en-us/service/help/power_functions/lego-powered-up-programming-blocks-kA06N000000g04eSAA

jazyka. Jeden program je tedy reprezentován jedním dílkem, ale uvnitř se může skládat z více dílků.



Obrázek 3.10: Tvorba programu v aplikaci LEGO PoweredUp.



Obrázek 3.11: Sekce nástroje Puzzle v chronologickém pořadí tvorby programu [6].

V sekci tvorba si uživatel může upravovat a vytvářet své programy. Uživateli je rovněž chytře poskytnuta nápověda v různých formách skrz celý proces tvorby programu. Když uživatel dlouze stiskne na dílek, zobrazí se mu nápověda co s dílkem může udělat. Taktéž je na pracovní ploše přítomna ikona žárovky, která zobrazí dialogové okno s detailnější nápovědou pro zvolený dílek.

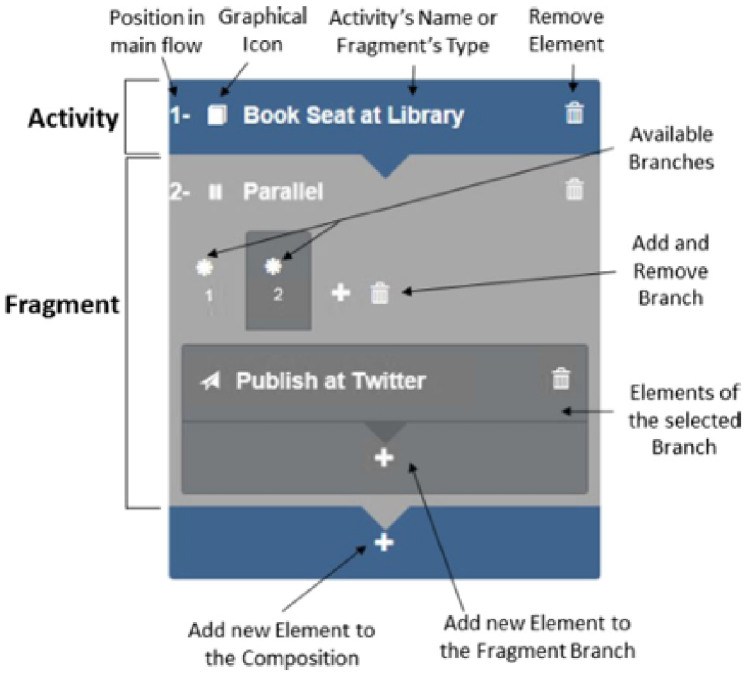
Puzzle poskytuje několik kategorií s předdefinovanými dílky, které mají různou funkcionalitu. Pomocí metody *drag and drop* lze dílky přesouvat a spojovat. V pravém dolním rohu je ikona odpadkového koše, do které lze dílky přesunout a tím je odstranit z pracovní plochy. Na rozdíl od nástroje EUCalipTool umožňuje Puzzle dílky volně přesouvat po celé pracovní ploše. Uživateli je tímto při tvorbě programu dána větší volnost a může si například postupně vytvářet různé části svého programu a na konci je spojit. S rostoucí

komplexitou ale mohou začít být programy značně nepřehledné. Uživatel musí k tvorbě programů přistupovat korektně (rozdělení jednoho programu na menší programy, které řeší specifický problém). Zde se opět setkáváme s problémem programátorského myšlení u koncových uživatelů [27].

Prostředí pro provádění programu umožňuje uživateli vytvořený program spustit a ovládat. Prezentuje grafické prvky, které mohou být implementovány z různých modulů. Například prvek tlačítka pro změnu stavu žárovky, nebo dokonce integrace různých webových modulů třetích stran, které poskytují funkcionalitu jako vytvoření a zveřejnění facebookového příspěvku [6].

EUCalipTool

EUCalipTool je mobilní aplikace, která v sobě integruje specifický VPL určený pro konkrétní doménu (*Domain Specific Visual Language – DSVL*). Tato aplikace uživatelům umožňuje tvořit programy, které propojují a využívají ostatní služby. Uživatel si může vytvořit kompozice, které automatizují jeho každodenní činnosti (viz. obrázek 3.12). Zástupci podobných aplikací jsou například Automate[23] nebo Tasker[24].



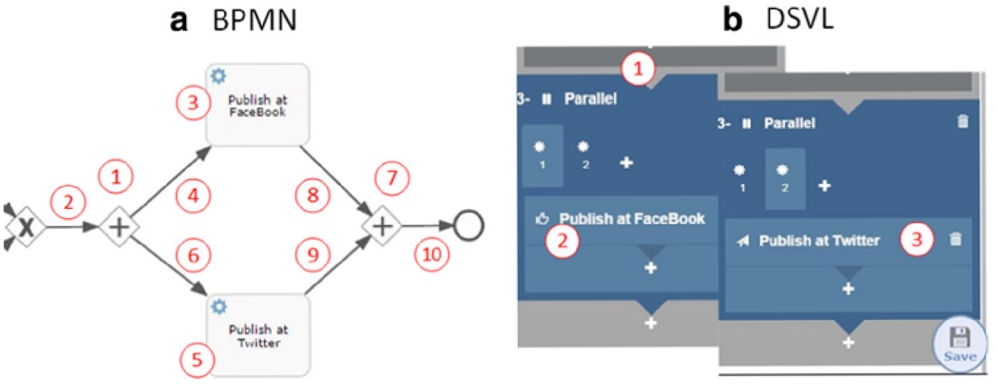
Obrázek 3.12: Ukázka grafické reprezentace kompozice [31]

Alternativní způsob tvorby kompozic je standard BPMN (*Business Process Model and Notation*)[25]. Jedná se o grafovou reprezentaci procesů, která by měla být pochopitelná širokou škálou uživatelů. Hlavním důvodem, proč EUCalipTool implementuje DSVL, je lepší uživatelská přívětivost a podpora mobilního zobrazení. Konkrétní příklad vytvoření

[23] https://play.google.com/store/apps/details?id=com.llamalab.automate&hl=en_US
[24] https://play.google.com/store/apps/details?id=net.dinglisch.android.taskerm
[25] https://cs.wikipedia.org/wiki/Business_Process_Model_and_Notation

kompozice a porovnání těchto dvou způsobů je na obrázku 3.13. Použití VPL v tomto případě značně zredukovalo počet kroků pro vytvoření kompozice.

Integrace VPL na mobilní zařízení není jednoduchá a naráží se na problémy spojené s velikostí displeje zařízení a zpracování uživatelského vstupu. Zároveň je ale nutná, jelikož většina běžných uživatelů používá nejvíce právě mobilní zařízení [6, 31].



Obrázek 3.13: Porovnání způsobů tvorby kompozic [31]

Kapitola 4

Analýza

Správa zařízení internetu věcí (IoT) je komplexní úkol. Proto vznikají dedikované platformy sloužící právě k tomuto účelu. Tyto platformy sjednocují jednotlivá zařízení IoT do komplexnějších systémů, které řeší konkrétní problém. Například systém pro automatizaci zavlažování se bude skládat z několika senzorů vlhkosti, senzoru slunečního svitu a relé pro ovládání čerpadla na vodu. Potřeba uživatele spočívá v monitorování hodnot zařízení (procentuální hodnota vlhkosti, svitu) a provádění akcí na zařízeních, která to umožňují (zapnutí nebo vypnutí relé). Uživatelé také potřebují systémy automatizovat (automatické spuštění zavlažování za jistých podmínek) a případně i definovat složitější logiku těchto systémů.

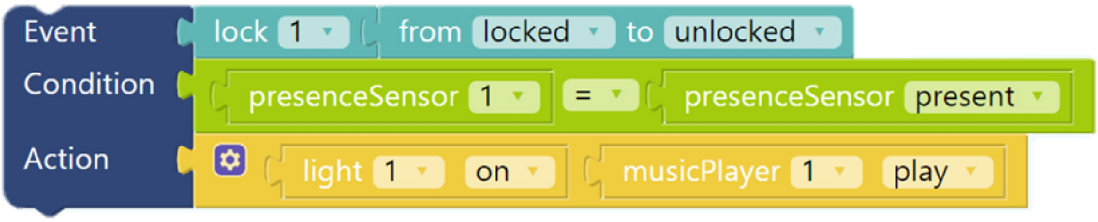
4.1 Aktuální způsoby programování zařízení IoT

S ohledem na široké spektrum technologií používaných v IoT je nezbytné přizpůsobit programovací přístupy konkrétním potřebám a specifikacím daných zařízení. V mnoha případech se pro programování zařízení IoT využívají běžné programovací jazyky, jako je C, C++, Python nebo Java. Tyto jazyky jsou často používány pro programování mikrokontrolérů nebo vývojových desek (např. Arduino nebo Raspberry Pi), které jsou jádrem mnoha zařízení IoT. Existují však nástroje, které se snaží programování zařízení IoT zjednodušit a poskytnout nejen profesionálním programátorům, ale i koncovým uživatelům vhodný nástroj pro definování vlastní logiky zařízení. Tyto nástroje přidávají vrstvy abstrakce a využívají různých technik pro usnadnění vytváření programů. Níže je popsáno několik aktuálních platforem a nástrojů, které lze k programování zařízení IoT využít.

Smart Block

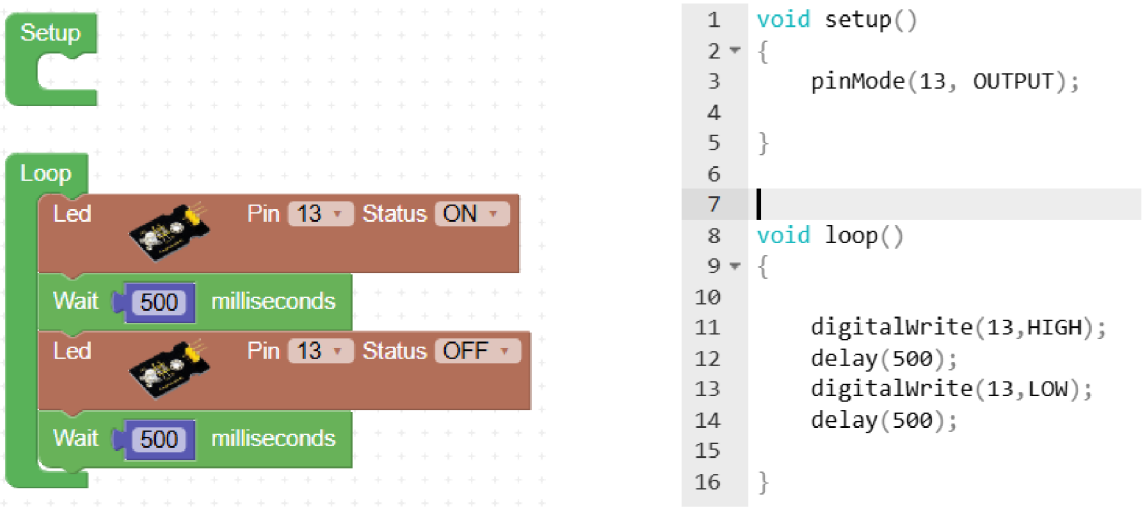
Smart Block je blokový vizuální programovací jazyk určený k vytváření programů na zařízení *SmartThings* od společnosti Samsung (viz. sekce 3.2) [4]. Byl vytvořen integrací nástroje Blockly, takže pro tvorbu programů využívá stejný grafický editor. Smart Block navíc přidává své vlastní *ECA* bloky (viz. obrázek 4.1), které by měli uživatelům usnadnit vytváření programů. Zkratka ECA znamená *event-condition-action* – česky událost-podmínka-akce a jedná se o definici pravidla. Jak je již z názvu patrné, pravidlo je rozděleno na tři části a sémantika pravidla je definována následovně: pokud byla detekována událost (odemknutí zámku, detekce senzorem, změna času), tak vyhodnoť podmínku, a pokud je podmínka splněna, proveď příslušnou akci [4]. Pravidla jsou poté součástí výsledného pro-

gramu na základě kterého je vygenerován kód pro zařízení SmartThings v jazyce Groovy[1]. Jazyk Groovy je běžně využíván pro programování zařízení SmartThings, ale nástroj Smart Blocks se tento proces programování snaží zjednodušit a zpřístupnit i koncovým uživatelům pomocí vizuálního editoru nástroje Blockly.



Obrázek 4.1: Příklad ECA bloku, který reprezentuje jedno ECA pravidlo. Logika pravidla definuje, že pokud došlo k události, kde zámek přejde ze stavu „zamknuto“ do stavu „odemknuto“ a osoba je přítomna u dveří, tak se provede akce rozsvícení světla a zapnutí hudby [4].

Nástroje *ArduinoBlocks* nebo *Arduviz* [25] taktéž využívají knihovnu Blockly, ale zaměřují se na programování vývojových desek od společnosti Arduino. Na obrázku 4.2 je ukázka výsledného programu vytvořeného v nástroji ArduinoBlocks.



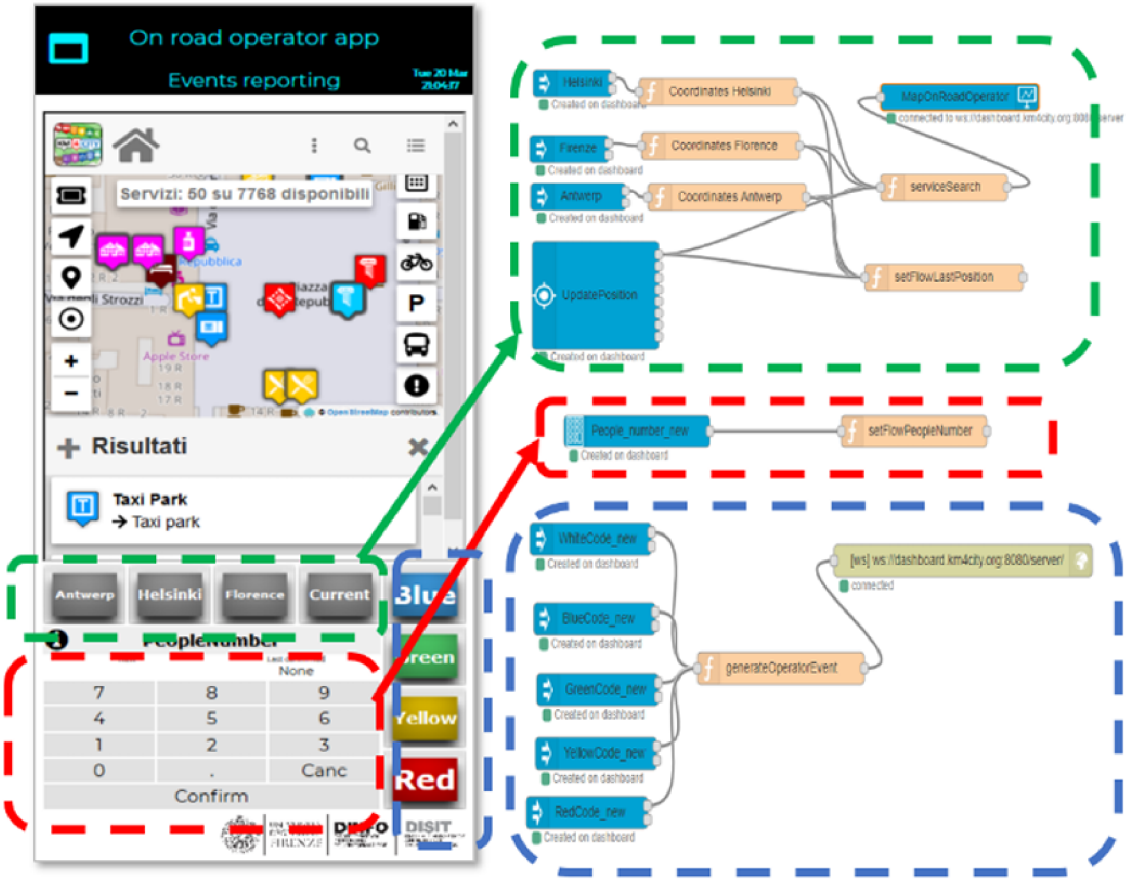
Obrázek 4.2: Na levé straně obrázku je pomocí webového editoru Blockly – do kterého byly integrovány speciální bloky pro manipulaci se zařízeními Arduino – definován výsledný program. Na pravé straně je vygenerovaný kód v jazyce C++ využívající funkce knihovny Arduino, který by byl nahrán na koncové zařízení IoT[3].

Snap4City

Snap4City je platforma usnadňující tvorbu systémů IoT chytrých měst [3]. Poskytuje kompletní řešení pomocí svých mikroslužeb. Tato platforma řeší programování koncových za-

[1]Oficiální stránky programovacího jazyka Groovy: https://groovy-lang.org/
[3]Oficiální stránky nástroje ArduinoBlocks: http://www.arduinoblocks.com/web/

řízení, ale hlavně se zaměřuje na agregaci a analýzu získaných dat. Většina nástrojů pro programování zařízení IoT je orientována na funkcionální programování a generování kódu pro tato zařízení. Platforma Snap4City na rozdíl od těchto nástrojů využívá přístup řízený daty (*data-driven approach*). Pomocí tohoto přístupu lze řešit komplexnější systémy určené pro chytrá města, která integrují velký počet zařízení. Platforma implementuje více než 150 generických mikroslužeb jako uzly do nástroje Node-RED (viz. sekce 3.3) a rozšiřuje tak jeho knihovnu [3]. Uzly jsou zpřístupněny uživatelům, kteří si pomocí nich ve webovém editoru nástroje Node-RED mohou vytvořit vlastní systémy IoT na míru, jak lze vidět na obrázku 4.3.

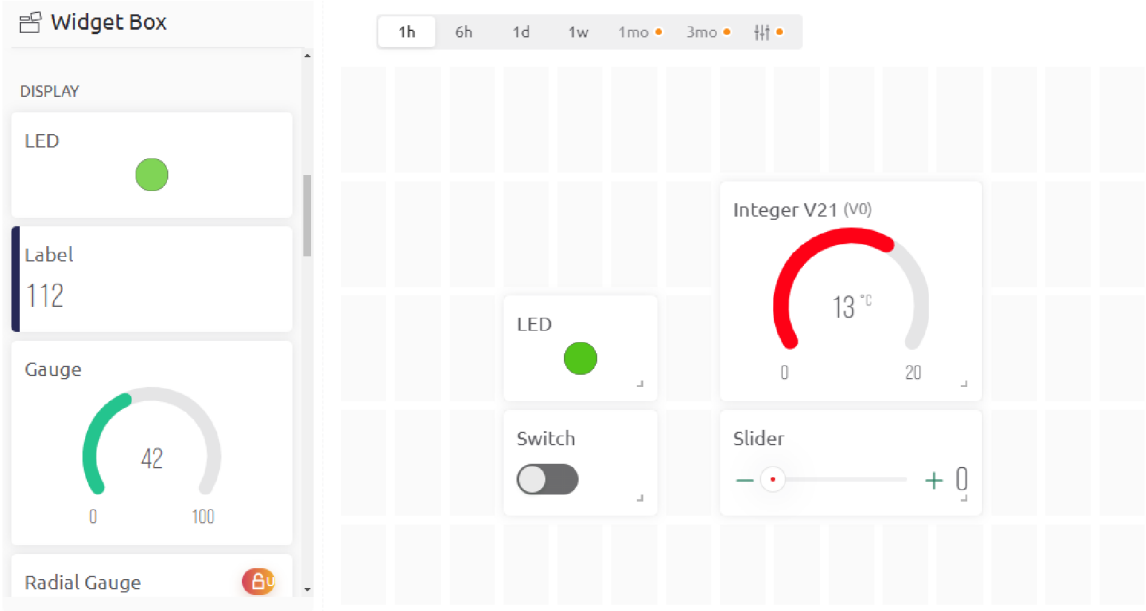


Obrázek 4.3: Příklad výsledné aplikace vytvořené za pomocí platformy Snap4City. Jedná se aplikaci určenou pro provozovatele pozemních komunikací, která umožňuje nahlašovat vzniklé události nebo nehody na silnicích. Na levé straně obrázku je výsledný vzhled aplikace s grafickými widgety. Na pravé straně je definována logika aplikace pomocí nástroje Node-RED. Barevné obdélníky vyznačují jaké widgety korespondují s jakými částmi logiky aplikace [3].

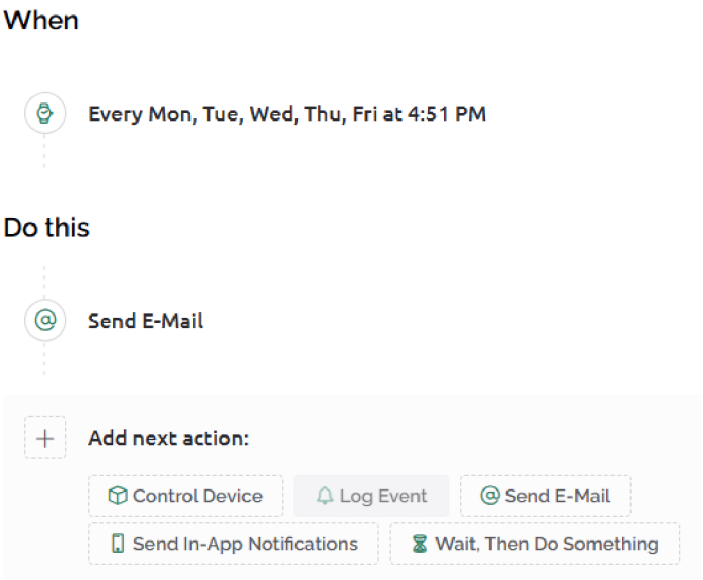
Blynk

Blynk je cloudová platforma sloužící pro integraci a správu zařízení IoT. Blynk při programování zařízení IoT poskytuje kompletní řešení v podobě webové aplikace. Nabízí přizpů-

sobitelný dashboard systém, kde si uživatel může vybírat z předpřipravených generických elementů (viz. obrázek 4.4). Jednotlivé elementy poté zapouzdřují funkcionalitu pro ovládání zařízení IoT. Tuto funkcionalitu uživatel nejprve musí sám definovat. Například v podobě nastavení příslušného hardwarového pinu zařízení, či zaslání správného příkazu přes síť. Uživatelé si také mohou vytvořit automatizace (viz. obrázek 4.5), ale pouze na základě jednoduchých podmínek, které jsou inspirovány technikou TAP (viz. sekce 3.2). Akce, které mohou být při automatizaci vykonány jsou taktéž limitovány.



Obrázek 4.4: Ukázka přizpůsobení dashboardu v platformě Blynk pomocí grafických widgetů[5].

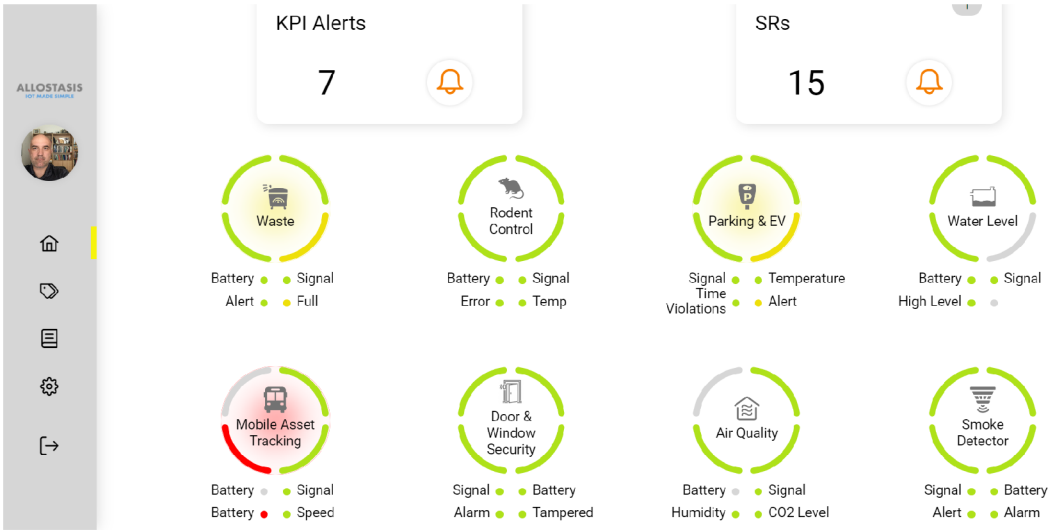


Obrázek 4.5: Příklad automatizace zařízení v platformě Blynk. Pokud je splněna podmínka, provedou se příslušné akce[5].

ACADA

ACADA (*Asset Control and Data Acquisition*)[6] je aplikační cloudová platforma vytvořena společností Logimic[7]. Slouží jako centrální systém zajišťující konfiguraci, ovládání, analýzu a prezentaci dat připojených zařízení IoT. Platforma ACADA je plně konfigurovatelná, umožňuje integraci a propojení již nainstalovaných zařízení IoT v existujících systémech.

Platforma ACADA poskytuje uživatelům řešení ve formě webové aplikace. Jednotlivá zařízení IoT jsou v aplikaci rozdělena do skupin. Každá skupina může obsahovat různé typy zařízení. Skupiny poskytují vrstvu abstrakce nad zařízeními, které řeší jeden konkrétní problém. Typickým příkladem je čtení hodnot ze senzorů zařízení IoT a jejich prezentace koncovému uživateli v podobě KPIs (*Key Performance Indicators*). Na obrázku 4.6 lze vidět reprezentace těchto indikátorů v dashboardu systému ACADA. Uživatelům je taktéž



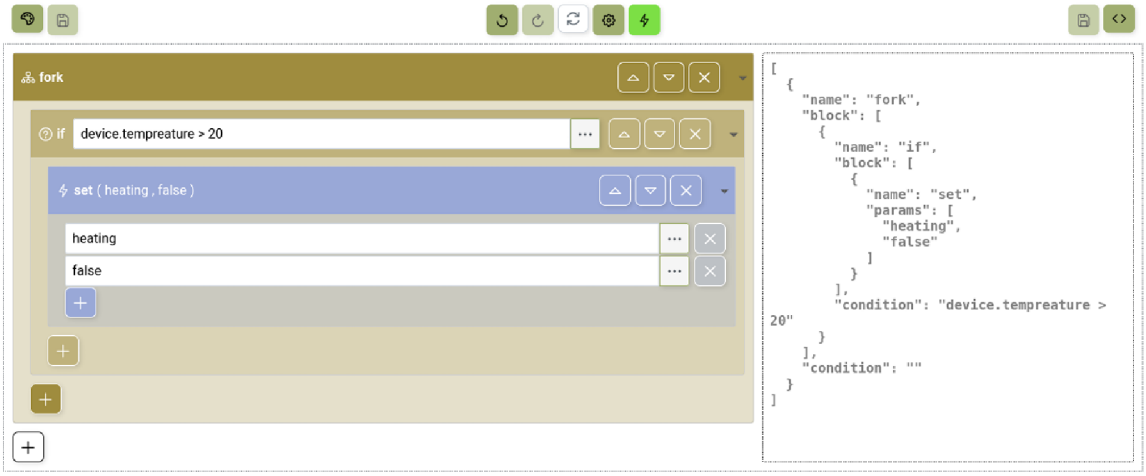
Obrázek 4.6: Úvodní obrazovka (dashboard) platformy ACADA, kde je zobrazen přehled všech skupin. Každá skupina má vlastní KPIs, které uživateli poskytují důležité informace a upozorňují ho na události (např. kritický stav baterie zařízení nebo přeplněnost odpadkových košů).

umožněno manuálně ovládat jednotlivá zařízení zasíláním příkazů. Chybí zde ale možnost automatizace a definování vlastní logiky zařízení uživatelem. Na základě tohoto problému byla vývojáři firmy Logimic vytvořena demo aplikace [12], integrující nový vizuální editor a vizuální programovací jazyk určený pro tvorbu programů na zařízení IoT.

Na obrázku 4.7 lze vidět vizuální editor zmiňované demo aplikace. Editor se zaměřuje na funkcionální styl programování a využívá známých programových konstrukcí jako jsou příkazy větvení nebo smyčky. Příkazy jsou reprezentovány jako grafické elementy, které uživatel může použít k tvorbě programu. Výsledný program je serializován do formátu JSON a zaslán na backend řešení, které jednotlivé příkazy programu interpretuje a zašle na koncová zařízení IoT.

Aktuální řešení má pár nedostatků, které je potřeba adresovat. Příkazy v programu jsou dvojího typu – jednoduché a složené. Složené příkazy mohou obsahovat další zanořené

[5]Oficiální stránka platformy Blynk: https://blynk.io/
[6]O platformě ACADA: https://www.logimic.com/cs/platforma/
[7]Oficiální stránky společnosti Logimic: https://logimic.com/



Obrázek 4.7: Ukázka demo aplikace vizuálního editoru. Na levé straně obrázku se nachází grafický editor tvořen jednotlivými příkazy. Příkaz s označením *if* je složeným příkazem s výrazem jako vstupním argumentem. Příkaz s označením *set* podporuje více vstupních argumentů textového typu. Na pravé straně obrázku je serializovaná podoba programu ve formátu JSON [12].

bloky příkazů. Příkazy také podporují rozšíření v podobě argumentů. Příkaz může mít jako vstupní argument podmínku nebo více vstupních argumentů textového typu. Podmínka je logický výraz, který uživatel tvoří v textové podobě. Pro uživatele by však mohlo být přívětivější využít specializovaný editor výrazů, který by jejich vytváření a zobrazení usnadnil. Taktéž by bylo vhodné rozšířit existující systém argumentů o další datové typy, aby uživateli mohla být automaticky nabízena korektní forma vstupu. Aktuální řešení například nepodporuje typ příkazu, který by byl složený a zároveň měl více vstupních argumentů různých datových typů. Kdyby se systém příkazů a argumentů více generalizoval, tak by se tento typ příkazu mohl v budoucnu použít například pro uživatelské funkce se vstupními parametry. Uživatelé by také mohli ocenit vytváření svých proměnných a funkcí pro lepší přehlednost a znovupoužitelnost v programu.

Momentálně je demo aplikace editoru součástí systému ACADA a implementována pomocí webového rámce Angular[8]. Editor by však mohl být koncipován jako nezávislá knihovna, která by se dala integrovat do existujících systémů IoT a tímto by došlo k rozšíření její působnosti. Knihovna by tak poskytovala část řešení pro programování zařízení IoT, která by umožnila koncovým uživatelům definici vlastního programu pomocí grafického editoru a následně ho převedla do serializované podoby pro další zpracování.

4.2 Požadavky na řešení

Při vytváření řešení jsem byl primárně inspirován demo aplikací z platformy ACADA. Toto řešení mně zaujalo hlavně díky možnosti vytvořit generické řešení pro různé typy zařízení IoT. Řešení také poskytuje uživateli dostatečnou volnost při tvoření programu. Ostatní nástroje se většinou snaží řešení zjednodušit na úkor funkcionality. Ve výzkumné zprávě, která popisuje fungování a implementaci vizuálního editoru a vizuálního programovacího

[8]Oficiální stránky rámce Angular: https://angular.io/

jazyka bylo také uvedeno několik návrhů na rozšíření a zlepšení stávajícího řešení. Tato zlepšení bych chtěl zakomponovat do nového řešení.

Na základě analýzy aktuálních nástrojů pro programování zařízení IoT a demo aplikace v systému ACADA jsem se rozhodl jako řešení vytvořit nový vizuální programovací jazyk, společně s grafickým editorem. Editor umožní koncovým uživatelům grafickým způsobem vytvořit serializovaný popis programu, který může být následně transformován do jazyka cílových zařízení nebo rovnou interpretován backend řešením. Z analýzy nedostatků existujících nástrojů jsem vyvodil požadavky, které by nový vizuální programovací jazyk a grafický editor měl splňovat:

- **Uživatelská přívětivost** – editor by měl být uživatelsky přívětivý a snadný na použití. Uživatel by se měl zaměřit na logiku svého problému a editor by mu měl pomoci tento problém vyřešit efektivním způsobem. Uživateli by také měla být přístupná nápověda.
- **Dostupnost** – editor by měl být dostupný na různé cílové platformy (mobilní telefony, tablety, počítače), které uživatel běžně používá. Z toho vyplývá, že editor bude muset mít responzivní design a fungovat na různých velikostech displejů. Editor by taktéž měl podporovat integraci do již existujících aplikací, které by ho chtěly využívat.
- **Obecnost řešení** – řešení by ale mělo být generické a fungovat v různých typech systémů (např. systém chytrého vytápění nebo systém automatizace zavlažování). Každý model zařízení IoT totiž podporuje jinou funkcionalitu a má rozdílné parametry.
- **Řešení výrazů** – výrazy se v definování logiky programu používají často a proto by měla jejich konstrukce a následná úprava být pro uživatele intuitivní.
- **Uživatelské proměnné a procedury** – uživatel by měl být schopen v programu vytvářet své vlastní proměnné s konkrétní hodnotou nebo výrazem. Uživatelsky definované procedury by definovaly znovupoužitelné části programu, které mohou být v programu použity na více místech. Proměnné a procedury by uživateli pomohly k lepší udržovatelnosti a přehlednosti programu.

Kapitola 5

Návrh webového grafického editoru

5.1 Navrhované řešení

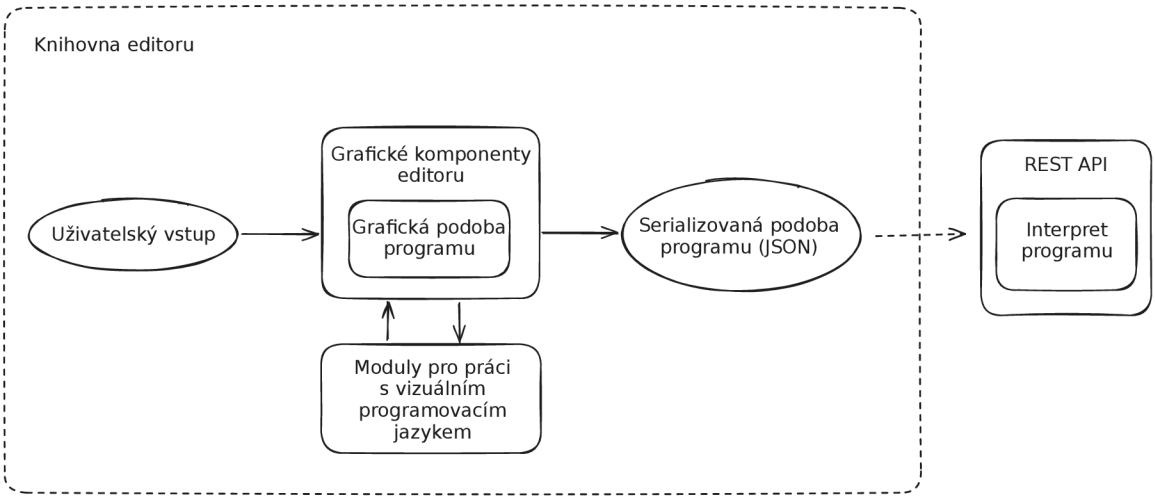
Z průzkumu vizuálního programování (VP) (kapitola 3) a analýzy problému (kapitola 4) jsem se jako řešení tohoto problému rozhodl vytvořit návrh modelu vizuálního programovacího jazyka, společně s webovým grafickým editorem, který tento jazyk bude využívat. Editor je navrhnut jako samostatná knihovna, která bude moci být implementována do existujících webových aplikací (jako je například systém ACADA společnosti Logimic). Návrh mého řešení se inspiruje již existujícím návrhem, který byl prezentován ve výzkumné zprávě vytvořené pro firmu Logimic [12]. Výzkumná zpráva se zabývá systémem řízení a monitoringu vody v retenčních nádržích, společně s návrhem a implementací prototypu editoru pro definování uživatelské logiky [12]. Existující návrh byl upraven a rozšířen o požadovanou funkcionalitu.

Jako alternativní řešení problému se nabízí implementace a upravení již existujících nástrojů VP. Tyto nástroje však v mnoha případech nepodporují rozsáhlé možnosti úprav samotného modelu jazyka a tudíž jsou značně limitující. Taktéž je u nich zřídka podporováno responzivní zobrazení. Konkrétním příkladem může být knihovna Blockly v jazyce JavaScript (podsekce 3.3). Jedná se o svobodný software s rozsáhlou dokumentací, která popisuje jak nástroj Blockly integrovat do existujících webových aplikací. Nástroj Blockly využívá systém bloků a kreslící plochy. Systém kreslící plochy nástroje Blockly není dle mého názoru pro můj návrh nejvhodnější. Obzvlášť při zobrazení na mobilních zařízeních. Uživatel se více zaměřuje na to, jakým způsobem bloky na kreslící ploše uspořádat než na problém, který se snaží vyřešit. Velikost bloků na mobilních zařízeních nebývá dostatečná a uživatel je poté nucen bloky přibližovat. Po přiblížení dochází k přetečení bloků a uživatel musí horizontálně rolovat. Při tvorbě komplexnějších programů se dá u mobilního zobrazení v programu lehce ztratit.

5.2 Architektura knihovny

Schéma architektury knihovny je na obrázku 5.1. Knihovna se skládá ze dvou hlavních částí – grafických komponent editoru a modulů pro práci s vizuálním programovacím jazykem (VPL). Grafické komponenty společně tvoří grafické uživatelské rozhraní editoru, kde bude uživatel tvořit výsledný program. Komponenty budou využívat funkcionalitu modulů pro práci s VPL (např. pro syntaktickou kontrolu nebo k manipulaci s objektovou reprezentací tvořeného programu). Pro práci s VPL jsou využívány dva hlavní moduly: modul jazyka

a modul programu. Oba moduly zapouzdřují datový model a příslušné funkce/metody pro práci s modely. Konečným výstupem editoru bude serializovaná podoba programu ve formátu JSON (*JavaScript Object Notation*). Tato podoba programu může být následně interpretována a jednotlivé instrukce zasílány na koncová zařízení IoT. Primární zaměření této práce je na frontend část editoru (grafické rozhraní společně s uživatelskou přívětivostí) a standardizaci modelu programu pro přenos na backend řešení.



Obrázek 5.1: Návrh architektury knihovny.

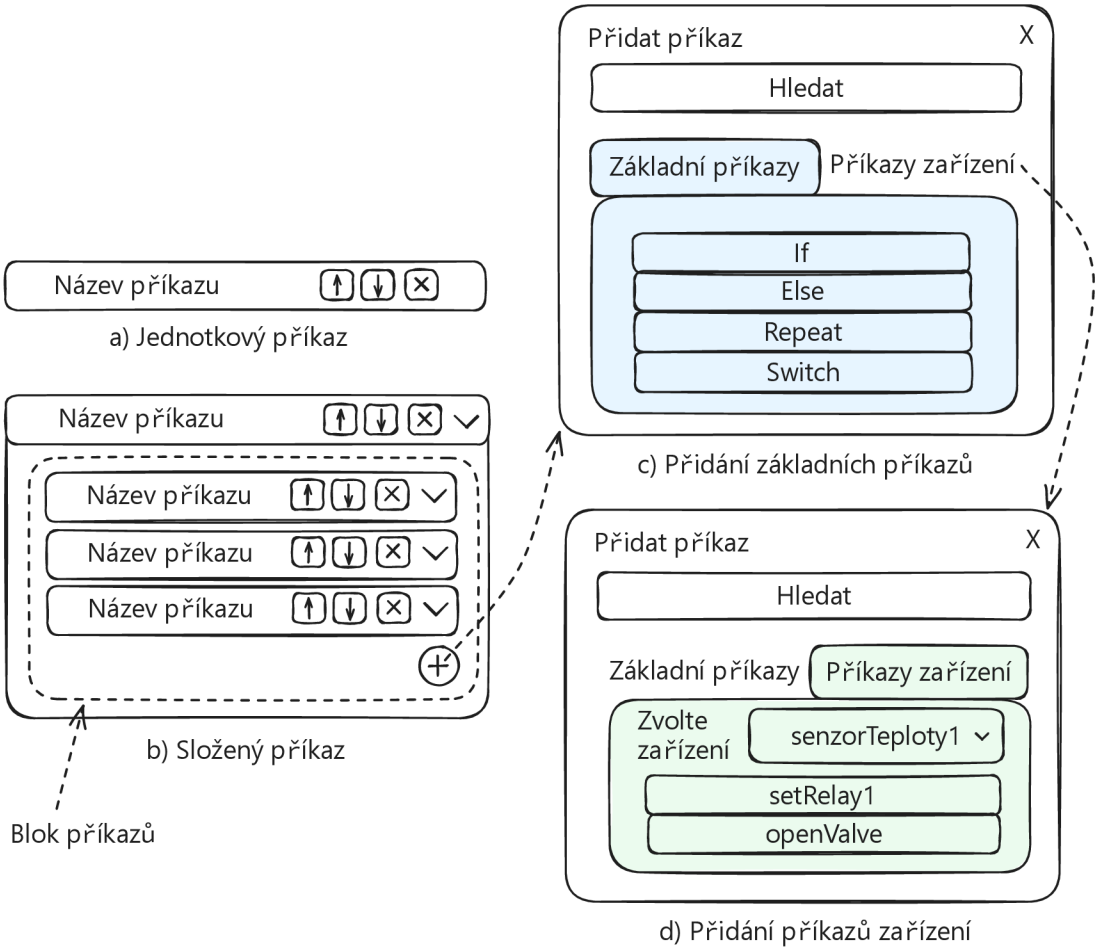
5.3 Návrh uživatelského rozhraní

Uživatelské rozhraní editoru je rozděleno do několika komponent. Rozložení jednotlivých komponent ve výsledné aplikaci záleží na jejich integraci vývojářem. Vývojář integrující knihovnu si sám může zvolit potřebné grafické komponenty.

Grafický editor

Grafický editor je tvořen posloupností příkazů. Příkaz je grafická komponenta, která má vizuální vlastnosti v podobě barvy, ikony a označení. Jsou podporovány dva hlavní typy příkazů – jednotkový a složený. Na obrázku 5.2 je možno vidět jednotkový i složený příkaz, společně s oknem pro přidání dalších příkazů. Jednotkový příkaz reprezentuje funkci. Ve většině případů se jedná o funkce zařízení, ale tento typ příkazu může reprezentovat například i uživatelsky definovanou proceduru. Složený příkaz podporuje zanoření dalších příkazů, které jsou seskupeny do bloku. Tělo složeného příkazu může být skryto nebo zobrazeno. Pořadí příkazů v bloku lze měnit pomocí tlačítek s ikonami šipek. Pro odstranění příkazu slouží tlačítko s ikonou křížku.

Oba typy příkazů podporují rozšíření o argumenty. Argumenty jsou vstupní pole různého typu. Mezi podporované typy argumentů patří: řetězec, číslo, Booleovská hodnota, výraz, proměnná, řetězcový a číselný výběr z předdefinovaných hodnot. Obrázek 5.3 znázorňuje příkaz s jedním i více vstupními argumenty.



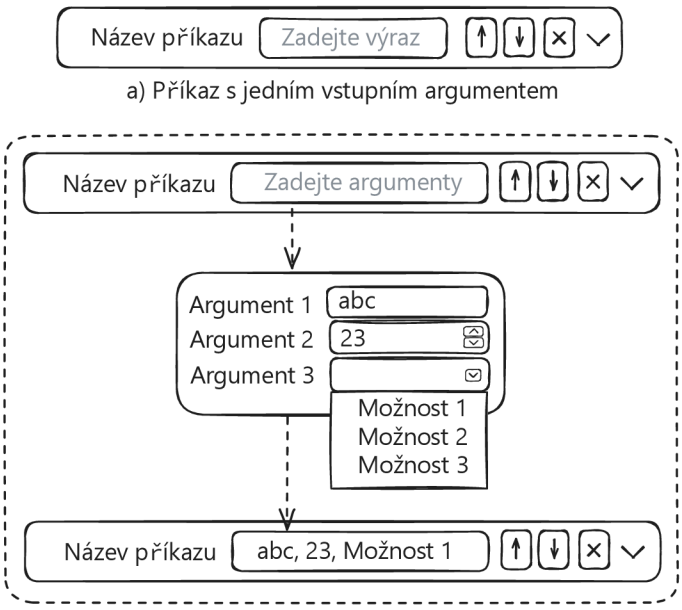
Obrázek 5.2: Na levé straně jsou dva hlavní typy příkazů. Na straně pravé je způsob přidání nových příkazů. Pro přehlednost je přidání příkazů rozděleno na základní příkazy jazyka a příkazy, které podporují jednotlivá zařízení. Pomocí vyhledávacího pole může uživatel snadno vyhledat příkaz pomocí jeho názvu.

Textový editor

Textový editor zobrazuje serializovanou podobu programu ve formátu JSON, vytvořeného v grafickém editoru. Grafický a textový editor jsou vzájemně provázány. Změny provedeny v textovém editoru jsou reflektovány do grafického editoru a obráceně. Textový editor je určen spíše pro zkušenější uživatele, kteří chtějí upravovat program v jeho serializované podobě. Zobrazení grafického a textového editoru lze přepínat pomocí ovládacích prvků. Uživatel si tedy může zvolit, jakou verzi editoru chce skrýt nebo naopak zobrazit.

Ovládací prvky editoru

Všechny ovládací prvky editoru jsou seskupeny do jedné grafické komponenty, která je vyobrazena na obrázku 5.4. Tlačítka reprezentující ovládací prvky zajišťují funkcionalitu jako import a export programu nebo přístup k uživatelským proměnným a procedurám.

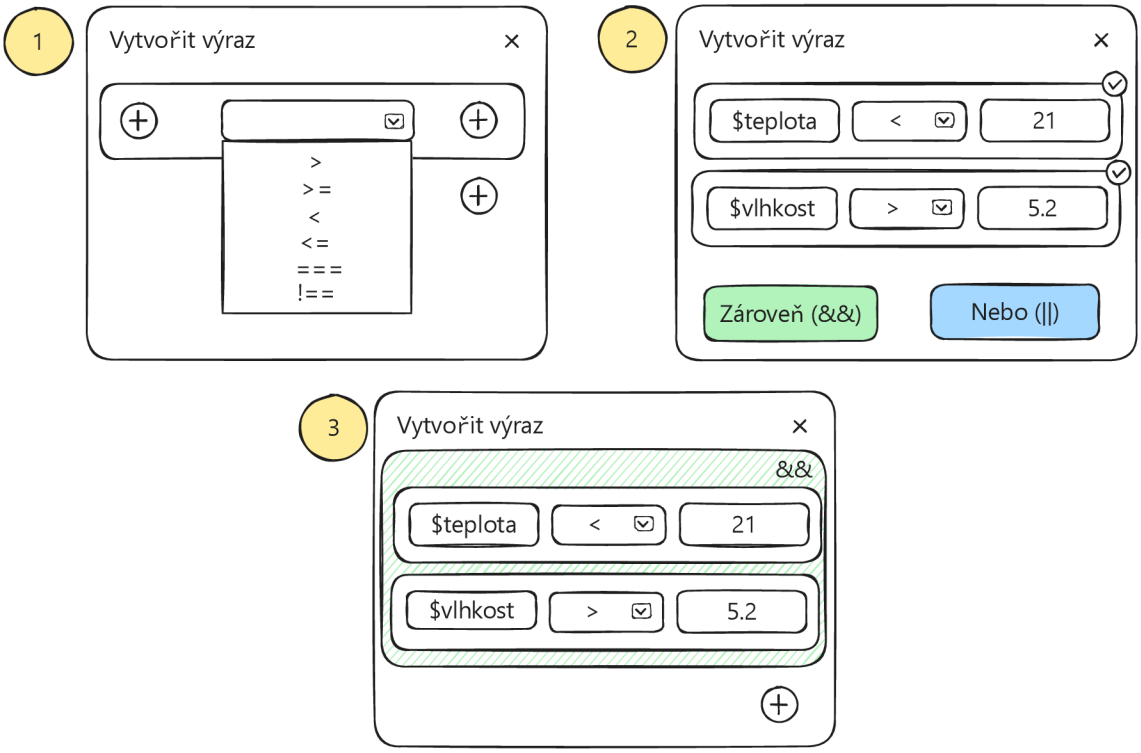


Obrázek 5.3: U příkazů s jedním vstupním argumentem je zobrazeno vstupní pole na základě typu argumentu (např. textové pole pro textový vstup nebo výběr z předdefinovaných hodnot). To umožňuje uživateli zadat hodnotu argumentu přímo, bez zobrazení dialogových oken. U příkazu s více vstupními argumenty je poté uživateli zobrazeno dialogové okno se všemi vstupními argumenty.

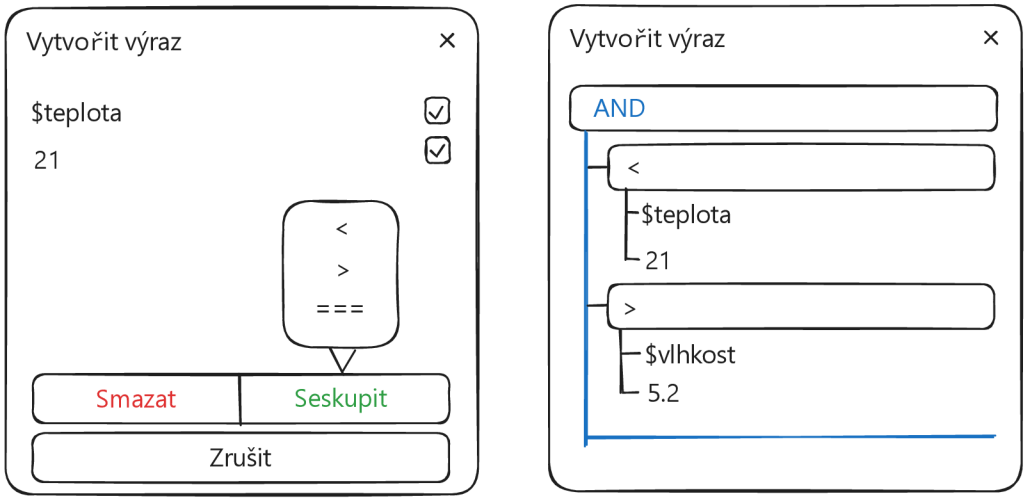
Obrázek 5.4: Každý ovládací prvek plní určitou funkcionalitu editoru. Na pravé straně je výběr zobrazení textového nebo grafického editoru.

Grafické řešení výrazů

Uživatel bude vytvářet výrazy grafickým způsobem. Výraz je tvořen jedním nebo více operandy a operátorem, který udává logický či matematický vztah mezi nimi. Operandem může být konstanta (číslo, řetězec, booleovská hodnota) nebo proměnná (uživatelská proměnná, proměnná zařízení). Při grafickém návrhu řešení výrazů jsem se rozhodoval mezi dvěma styly. První z nich je reprezentován na obrázku 5.5. Tento styl zobrazení je pro uživatele intuitivní, ale neřeší případ unárního nebo potenciálně n-árního operátoru. Totéž by platilo pro matematické operátory. Druhý styl, který vznikl postupnými iteracemi návrhu se snaží systém zobrazení výrazů více generalizovat. Výrazy jsou zobrazeny stromovou strukturou, jak je vidět na obrázku 5.6. Toto zobrazení může podporovat jakýkoliv typ operátoru a přehledně zobrazit i několik vrstev zanoření. Při implementaci jsem se tedy rozhodl použít druhý styl zobrazení.



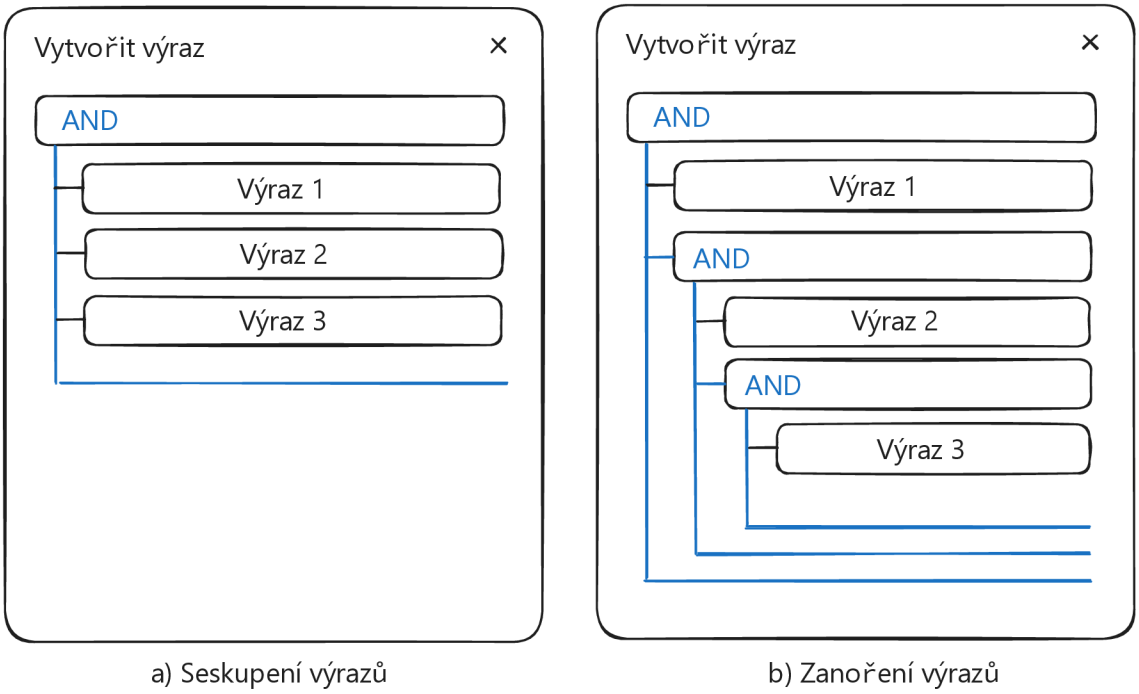
Obrázek 5.5: Prvotní návrh systému vytváření výrazů. Čísla u jednotlivých oken označují pořadí vytváření výrazů. Uživatel nejprve vytváří jednoduché výrazy a poté je seskupuje nebo zanořuje. Po vytvoření jednoduchých výrazů si uživatel zvolí, které výrazy chce seskupit a zvolí příslušný logický vztah, který bude aplikován mezi všemi zvolenými výrazy. Takto seskupené výrazy jsou poté barevně odlišeny na základě typu logického vztahu.



Obrázek 5.6: Zvolená verze návrhu systému vytváření výrazů. Systém seskupování výrazů zůstává stejný jako u prvotního návrhu.

Návrh zobrazení logických či matematických výrazů v grafické podobě tak, aby byl podporován responzivní design, je složitý. Naráží se zde na hlavní problém, kterým je za-

noření jednotlivých výrazů. Zanoření se většinou indikuje odsazením zanořeného prvku. Na mobilních zařízeních a zařízeních s menší úhlopříčkou displeje je však většinou omezená dostupná šířka displeje. Čím větší zanoření, tím menší je dostupný prostor pro vykreslení grafických elementů na displej. Z tohoto důvodu je lepší využívat vertikální délky displeje. Tento problém jsem se z části snažil řešit seskupováním výrazů do skupin podle logických vztahů mezi nimi. Z analýzy běžně vytvářených programů pro chytrá zařízení vyšlo najevo, že uživatelé většinou potřebují vytvářet konstrukce typu: „Když je splněno A a zároveň B a zároveň C, tak proveď nějakou akci.“ (např. Zapni topení, pokud je teplota v místnosti menší, než požadovaná hodnota, topení je vypnuto a je odpoledne.). Jelikož je logický člen mezi výrazy stejný, tak se výrazy nemusí zanořovat, ale pouze se seskupí. Na obrázku 5.7 je vidět rozdíl mezi reprezentací zanořením a seskupením. Seskupením výrazů se může ušetřit mnoho vrstev zanoření a tím se zobrazení výrazu zjednoduší.



a) Seskupení výrazů

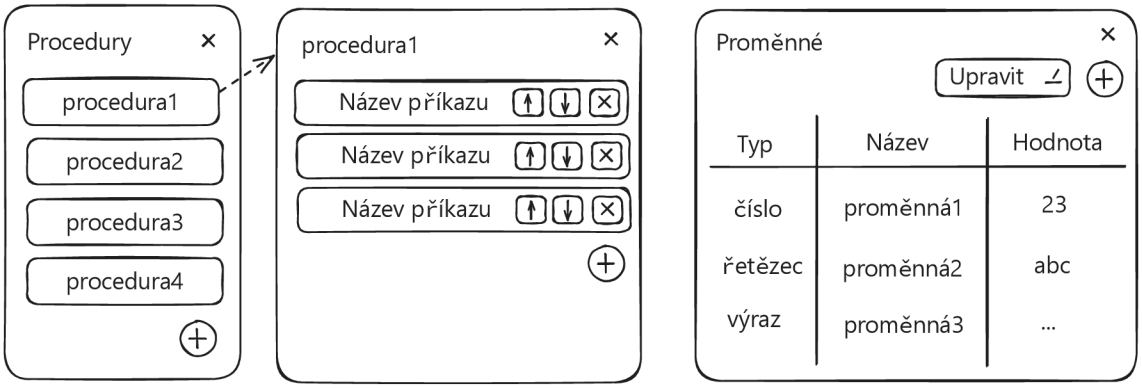
b) Zanoření výrazů

Obrázek 5.7: Porovnání systému seskupení a zanoření výrazů.

Správa uživatelských proměnných a procedur

Uživateli je umožněno vytvářet vlastní proměnné různých datových typů, které jsou poté součástí výstupního programu. Proměnné mohou být použity v argumentech příkazů se stejným datovým typem. Také mohou být součástí výrazů. Pro nastavení hodnoty do proměnné slouží konkrétní příkaz, který je v základní sadě příkazů jazyka.

Uživatel má také možnost definovat vlastní procedury. Procedura je blok příkazů, které chce uživatel opakovaně použít na více místech v programu. Procedury mohou zapouzdřovat složitější logiku a přispět k znovupoužitelnosti částí programu. Způsob spravování uživatelských proměnných a procedur je na obrázku 5.8.



a) Správa uživatelských procedur

b) Správa uživatelských proměnných

Obrázek 5.8: Na levé straně obrázku je způsob správy uživatelských procedur. Všechny procedury jsou zobrazeny v seznamu, ze kterého si uživatel vybere konkrétní proceduru a její definici následně může upravovat. Na pravé straně obrázku je okno s tabulkou uživatelských proměnných. Uživatel se pomocí tlačítka „Upravit“ může přepnout do editačního režimu a upravit název nebo počáteční hodnotu proměnné.

5.4 Datový model knihovny

Datový model knihovny je rozdělen do dvou hlavních částí:

- **datový model jazyka** definuje použitelné programové konstrukce, jejich syntaktické pravidla a vzhled,
- **datový model programu** popisuje serializovanou podobu programu vytvořeného grafickým editorem.

Datový model jazyka

Návrh definice datového modelu jazyka je popsána za pomocí jazyka TypeScript ve výpisu 5.1. Základními prvky jazyka jsou proměnné a příkazy.

```
type Language {
  variables: Variables;
  statements: Statements;
}

type Variables = {
  [id: string]: {
    type: VariableTypes;
    label: string;
  };
};


type Statements = {
  [id: string]: Statement;
};
```

```
type Statement =
  | UnitLanguageStatement
  | UnitLanguageStatementWithArgs
  | CompoundLanguageStatement
  | CompoundLanguageStatementWithArgs;

type UnitLanguageStatement = {
  type: LanguageStatementType;
  label: string;
  icon: string;
  foregroundColor?: string;
  backgroundColor?: string;
  predecessors?: string[];
  successors?: string[];
  parents?: string[];
};

type UnitLanguageStatementWithArgs = UnitLanguageStatement & {
  args: Argument[];
};

type CompoundLanguageStatement = UnitLanguageStatement & {
  nestedStatements?: string[];
};

type CompoundLanguageStatementWithArgs = UnitLanguageStatementWithArgs &
    CompoundLanguageStatement;

type Argument = {
  type: ArgumentType;
  options?: {
    id: string | number;
    label: string;
  }[];
};
```

Výpis 5.1: Zjednodušený datový model jazyka reprezentován v jazyce TypeScript.

Proměnné jazyka mohou být uživatelskými proměnnými, které si uživatel sám ve svém programu definoval nebo proměnnými jednotlivých zařízení. Proměnné zařízení reprezentují jejich atributy. Například senzor teploty bude mít atribut obsahující aktuální naměřenou teplotu v místnosti. Atribut zařízení bude převeden do proměnné a dynamicky vložen do seznamu proměnných jazyka. Tímto se zpřístupní uživateli, který ho může následně v programu použít. Uživatel si taktéž bude moci definovat své proměnné různých datových typů, což může pomoci k znovupoužitelnost těchto proměnných a zlepšení čitelnosti programu. Mezi aktuálně podporované typy proměnných patří: řetězec, číslo, Boolovská hodnota a výraz.

Množina příkazů jazyka definuje jednotlivé příkazy, které uživatel může při tvorbě programu použít. Příkazy obsahují atributy pro definici jejich vzhledu (barva, ikona, označení) a vztahů s ostatními příkazy. Ve výpisu 5.2 je definována základní množina příkazů, která může být dynamicky rozšířena o příkazy jednotlivých zařízení. Příkazy zařízení odpovídají funkcím, které zařízení podporuje. Například relé pro ovládaní vodního čerpadla má funkci na sepnutí ventilu, který pouští nebo zastavuje čerpání vody.

```
if: {
    type: 'compound_with_args',
    args: [{ type: 'bool_expr' }],
},
elseif: {
    type: 'compound_with_args',
    args: [{ type: 'bool_expr' }],
    predecessors: ['if'],
},
else: {
    type: 'compound',
    predecessors: ['if', 'elseif'],
}
```

Výpis 5.2: Ukázka definice základních příkazů větvení. U příkazů byly pro zkrácení výpisu odebrány jejich vizuální vlastnosti a definice nápovědy.

Původní návrh příkazů, který byl prezentován v technické zprávě [12] je v tomto návrhu upraven pro více generické řešení a rozšířen o systém argumentů. V původním návrhu byly příkazy rozděleny na dva hlavní typy – jednotkové a složené. Aktuální návrh příkazy rozděluje do čtyř typů a to na příkazy jednotkové, složené, jednotkové s argumenty a složené s argumenty. Každý příkaz tedy může být rozšířen o pevně daný počet vstupních argumentů různých datových typů. Původní návrh neumožňoval vytvořit složený příkaz s argumenty libovolného typu, který by mohl být použit například k vytvoření příkazu typu *switch* známého z ostatních programovacích jazyků. Přidáním systému vstupních argumentů s různými datovými typy k oběma typům příkazů vzniká více generické řešení, které může být v budoucnu dále rozšiřováno.

Pro zajištění korektní struktury výsledného programu musí mít každý příkaz definována omezení vztahů vzhledem k ostatním příkazům. Omezení jsou dvojího typu:

- **předchůdce – následník** definují, jaké konkrétní příkazy mohou být před a za příkazem,
- **rodič – potomek** definují, jaké konkrétní příkazy mohou být zanořeny v příkazu, případně v jakých rodičích příkaz může být zanořen.

Struktura programu je ověřována statickou analýzou na základě definice strukturálních omezení u příkazů. Pokud dojde k porušení těchto omezení, tak je uživatel vizuálně upozorněn na vzniklé chyby.

Datový model programu

Datový model programu popisuje grafickou verzi programu jako reprezentaci objektů, které jsou následně serializované do formátu JSON. Serializovaná podoba programu může být následně sdílena nebo zaslána na server, kde je program dále zpracován a vyhodnocen. Výpis 5.3 obsahuje typové definice programu a jeho částí.

Program se skládá z hlavního bloku, který je tvořen seznamem použitých příkazů. Typy příkazů programu korespondují s typy příkazů jazyka. Struktura programu byla rozšířena o systém argumentů, uživatelských proměnných a procedur. Uživatelské proměnné a procedury jsou součástí hlavičky programu. Každá uživatelská proměnná má svůj datový typ a počáteční hodnotu, kterou uživatel definuje při jejím vytvoření. Taktéž byla upravena

struktura výrazů z textové reprezentace celého výrazu na objektovou reprezentaci, jak lze vidět ve výpisu 5.4. Tato změna při přenosu serializované verze programu sice mírně zvýší objem přenášených dat, ale zato zjednoduší zpracování výrazů na straně serveru.

```
type Program {
  header: {
    userVariables: { [id: string]: UserVariable };
    userProcedures: { [id: string]: ProgramStatement[] };
  };
  block: ProgramStatement[];
}

export type UserVariable = {
  type: 'str' | 'num' | 'bool' | 'bool_expr';
  value: string | number | boolean | Expression;
};

type ProgramStatement = AbstractStatement | AbstractStatementWithArgs |
                        CompoundStatement | CompoundStatementWithArgs;

type AbstractStatement = { id: string };

type AbstractStatementWithArgs = AbstractStatement & {
  args: {
    type: ArgumentType;
    value: string | number | boolean | Expression;
  };
};

type CompoundStatement = AbstractStatement & { block: ProgramStatement[] };

type CompoundStatementWithArgs = AbstractStatement & CompoundStatement &
    AbstractStatementWithArgs;
```

Výpis 5.3: Zjednodušený datový model programu. Některé datové typy byly vynechány, protože se podobají struktuře datového modelu jazyka.

```
type Expression = {
  opds: (ExpressionOperand | Expression)[];
  opr?: ExpressionOperator;
};

export type ExpressionOperand = {
  type: ExpressionOperandType;
  value: string | number | boolean | Expression;
};
```

Výpis 5.4: Zjednodušený datový model výrazů. Výrazy se do sebe mohou rekurizvně zanořovat.

Kapitola 6

Implementace

Pro implementaci knihovny vizuálního programovacího jazyka (VPL) a grafického editoru jsem se rozhodl použít webové technologie. Díky tomuto rozhodnutí bude splněn požadavek ze sekce 4.2 na podporu integrace nástroje do již existujících webových aplikací. Jedná se o knihovnu vytvořenou v jazyce TypeScript. Programovací jazyk TypeScript rozšiřuje jazyk JavaScript o typovací systém. Při sestavení je knihovna transpilována do jazyka JavaScript. Knihovna je rozdělena na dvě hlavní části. První z nich implementuje sadu grafických komponent. Druhá část knihovny obsahuje moduly v jazyce TypeScript pro práci s VPL.

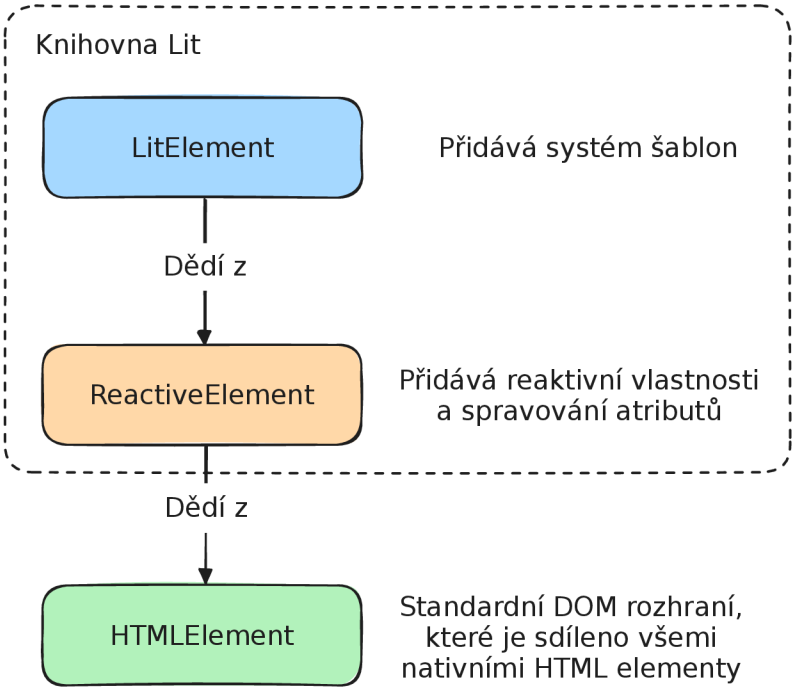
Pro implementaci grafických komponent byla zvolena knihovna *Lit* využívající technologii webových komponent. Webové komponenty jsou kolekcí různých technologií a webových aplikačních rozhraní (API), které umožňují vytvářet vlastní, znovupoužitelné a zapouzdřené HTML elementy [20, 32]. Tyto elementy mohou být poté využity ve webových stránkách a aplikacích. Webové komponenty fungují na podobném principu zapouzdřování logiky, stylování a HTML šablon do jednoho komponentu jako většina moderních frontend rámců (např. React, Vuejs nebo Angular). Hlavní výhodou webových komponent je ale standardizace procesu vytváření vlastních HTML elementů a využití již dostupných webových API. To zajišťuje podporu napříč všemi moderními prohlížeči a webové komponenty lze tedy použít v projektu, který využívá jakýkoliv frontend rámec nebo jen čistý JavaScript.

6.1 Integrace knihovny Lit

Lit je jednoduchá knihovna pro vytváření rychlých a odlehčených webových komponentů [17]. Rozšiřuje webové komponenty o dodatečnou funkcionalitu, jako je například reaktivita nebo deklarativní systém šablon. Tato knihovna byla zvolena hlavně pro snazší a rychlejší vývoj. Jedná se sice o přidanou závislost do projektu, je však dostatečně malá (minifikované zdrojové kódy knihovny zabírají okolo 5 KB [17]), na to aby byla přidána jako součást výsledného kódu knihovny. Lit také zachovává kompatibilitu webových komponent, protože každý Lit element je standardním HTML elementem, jak lze vidět na diagramu z obrázku 6.1.

Komponenty knihovny Lit

Grafické uživatelské rozhraní je složeno z několika Lit komponentů (dále označováno jen jako komponenty). Tyto komponenty mapují objektovou strukturu programu. Knihovna Lit k vytváření komponentů využívá systém tříd. Každý komponent je tedy třída, která dědí z třídy `LitElement`. Komponent se skládá z několika částí:



Obrázek 6.1: Diagram dědičnosti tříd Lit elementu. Inspirováno [18].

- lokálních stylů, které jsou (až na výjimky) aplikovány pouze na daný komponent,
- vlastností, které komponent obdrží ve formě vstupu a následně uchovává nebo mění jejich stav,
- referencí na elementy, pokud je potřeba manipulovat s konkrétním elementem,
- metod životního cyklu komponentu,
- metod pro zpracování uživatelských akcí,
- šablon sloužících pro rozdělení částí uživatelského rozhraní v rámci jednoho komponentu.

Komponenty se před jejich použitím v HTML dokumentu musí nejdříve registrovat. Registraci komponent zajišťuje knihovna Lit za pomocí dekorátoru `@customElement`. Tento dekorátor na pozadí využívá metodu `define` z rozhraní `CustomElementRegistry`, která je součástí základních webových API. Registrací komponentu se vytvoří nový záznam do registru vlastních komponentů a jeho jméno se namapuje na konstruktor třídy komponentu, který bude použit při jeho vytváření a vykreslování. Při následném použití komponentu v HTML dokumentu se vytvoří instance třídy, která komponent reprezentuje a za pomocí konstruktoru dané třídy se inicializuje.

Globální kontext

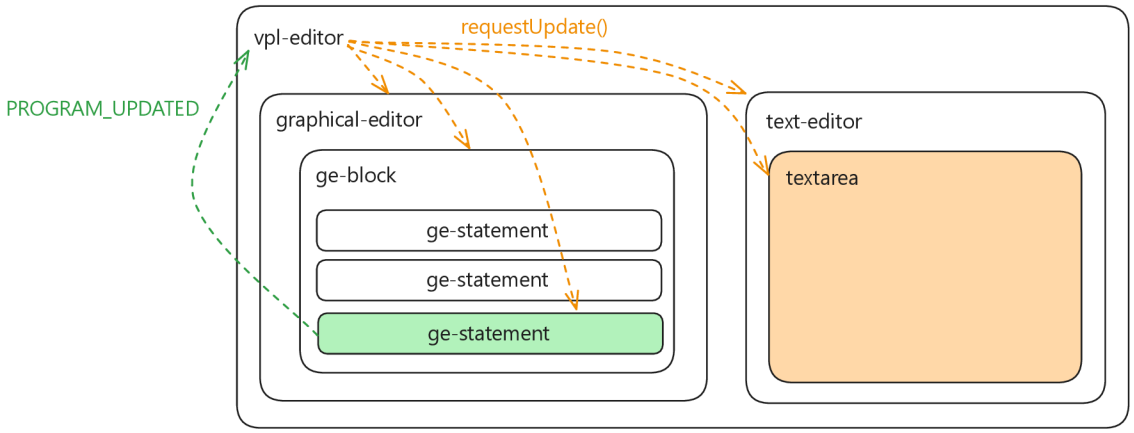
Kontext je způsob, jak poskytnout zanořeným komponentům data bez nutnosti jejich předávání přes všechny rodičovské komponenty. Data jsou tak globálně přístupná pro všechny

komponenty, které je potřebují. Komponent, který kontext (data) poskytuje dalším komponentům je označován jako *provider*. Komponent, který k datům přistupuje je označován pojmem *consumer*. Poskytnutí kontextu znamená vytvoření nové instance třídy nebo pouze vytvoření nového objektu. Do příslušného kontextu se poté uloží reference na danou třídu (objekt), která bude globálně dostupná. V tomto projektu je kontext využit při vytváření instance jazyka a programu. Instance jsou poté globálně dostupné, což znamená, že jakýkoliv komponent může v případě potřeby upravit strukturu programu nebo jazyka.

Synchronizace změn

Při změně programu (přidání příkazu, změna argumentu) se musí znovu vykreslit komponenty, které touto změnou byly ovlivněny. Příslušný komponent při změně objektové reprezentace programu vytvoří novou vlastní událost `PROGRAM_UPDATED`. Definice všech vlastních událostí jsou dostupné v souboru `editor-custom-events.ts`. Komponent `vpl-editor` má ve svém konstruktoru definováno, že má za pomocí metody `addEventListener` naslouchat právě tomuto typu události. Při přijetí události je volána metoda `handleGraphicalEditorProgramUpdated`, která se stará o znovu vykreslování změn, statickou analýzu a synchronizaci nové verze programu s textovým pohledem.

Jelikož systém reaktivity knihovny Lit automaticky nedokáže rozeznat změnu při modifikování objektu, je potřeba zažádat o opětovné vykreslení komponentu ručně. Lit totiž považuje pole a objekty za neměnné (*immutable*) a jediný způsob jak automaticky vyvolat aktualizaci vykreslení je tento objekt nahradit za nový. Při zažádání o opětovné vykreslení, knihovna Lit detekuje změny a upraví pouze části dokumentového objektového modelu (*DOM*), které byly změnami ovlivněny. Metoda `handleGraphicalEditorProgramUpdated` využívá pomocnou funkci `triggerUpdateAll` pro rekurzivní procházení všech zanořených elementů a vyvolání aktualizace o opětovné vykreslení pomocí metody `requestUpdate`, tak aby byla zajištěna aktualizace všech grafických komponent. Na obrázku 6.2 lze vidět schéma, které vizualizuje proces synchronizace a vykreslení změn.



Obrázek 6.2: Vizualizace procesu synchronizace a vykreslení změn. Obdélníky představují HTML elementy a jejich hierarchii v dokumentovém objektovém modelu. Zelenou barvou je označen element, jehož vlastnosti byly změněny. Oranžová barva označuje element, který bylo nutno aktualizovat. Šipky reprezentují zasílání událostí.

6.2 Vizuální programovací jazyk

Modul pro práci s vizuálním programovacím jazykem obsahuje typové definice částí jazyka společně s podpůrnými funkcemi a metodami. Samotný jazyk je reprezentován třídou `Language`. Při vytvoření instance jazyka probíhá jeho inicializace. Konstruktor třídy `Language` přijímá vstupní parametry `statements` a `devices`.

Parametr `statements` obsahuje definici základních příkazů. Základní příkazy jsou definovány v souboru `base.language.ts`. Tento soubor obsahuje objekt `baseLanguageStatements`, který je načten při poskytnutí kontextu (vytváření instance) jazyka v komponentu `vpl-editor`. Objekt `baseLanguageStatements` však může být nahrazen jakýmkoliv jiným objektem s definicí vlastního jazyka. Při reálném použití v praxi by knihovna poslala požadavek na backend řešení, které by poskytlo objekt s definicí příkazů jazyka.

V parametru `devices` se nachází objekt s definicí zařízení, které budou dostupné při vytváření programu. Příklad objektu definující použitelná zařízení je uveden v příloze A. Inicializací se převedou atributy a funkce jednotlivých zařízení do příkazů a proměnných, tak aby mohli být součástí výsledného jazyka. Tento dynamický přístup vytváření jazyka umožňuje vývojářům definovat vlastní jazyk pro různé typy systémů IoT. Tímto byl splněn požadavek na obecnost řešení, který byl definován v podsekci 4.2.

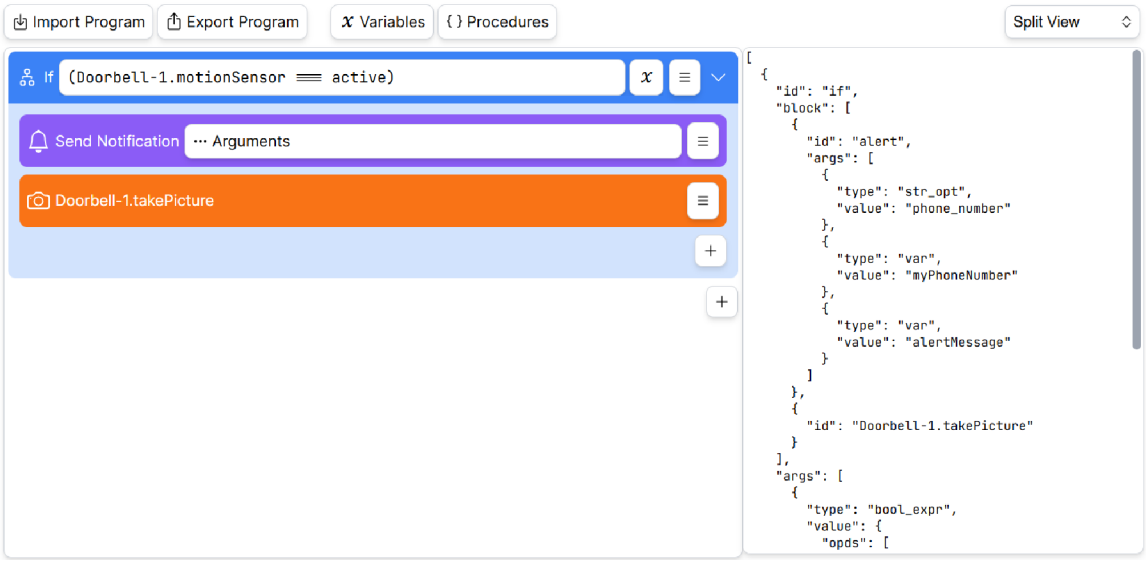
Modul pro práci s programem slouží k vytvoření a manipulaci s objektovou reprezentací výsledného programu. Program je reprezentován stejnojmennou třídou a skládá se z hlavičky a těla programu (bloku příkazů). Hlavička obsahuje definice uživatelských proměnných a procedur. Program může být importován a exportován. Příklad exportovaného programu ve formátu JSON je v příloze B. Metoda `loadProgram` zajišťuje korektní načtení programu ze souboru ve formátu JSON, včetně uživatelských proměnných a procedur (viz. výpis 6.1). Při načítání programu je potřeba přiřadit každému příkazu v bloku interní identifikátor, který je použit při vykreslování grafického rozhraní pro identifikaci elementů. Metoda `exportProgram` funguje na stejném principu jako metoda `loadProgram`, ale namísto přidávání, interní identifikátory odstraňuje.

```
loadProgram(programInJSON: string) {
    let program = JSON.parse(programInJSON);
    let procedureKeys = Object.keys(program.header.userProcedures)

    for (let procKey of procedureKeys) {
      assignUuidToBlock(program.header.userProcedures[procKey]);
    }
    assignUuidToBlock(program.block);

    this.header.userProcedures = program.header.userProcedures;
    this.header.userVariables = program.header.userVariables;
    this.block = program.block;
  }
```

Výpis 6.1: Definice metody `loadProgram`.

6.3 Vizuální editor

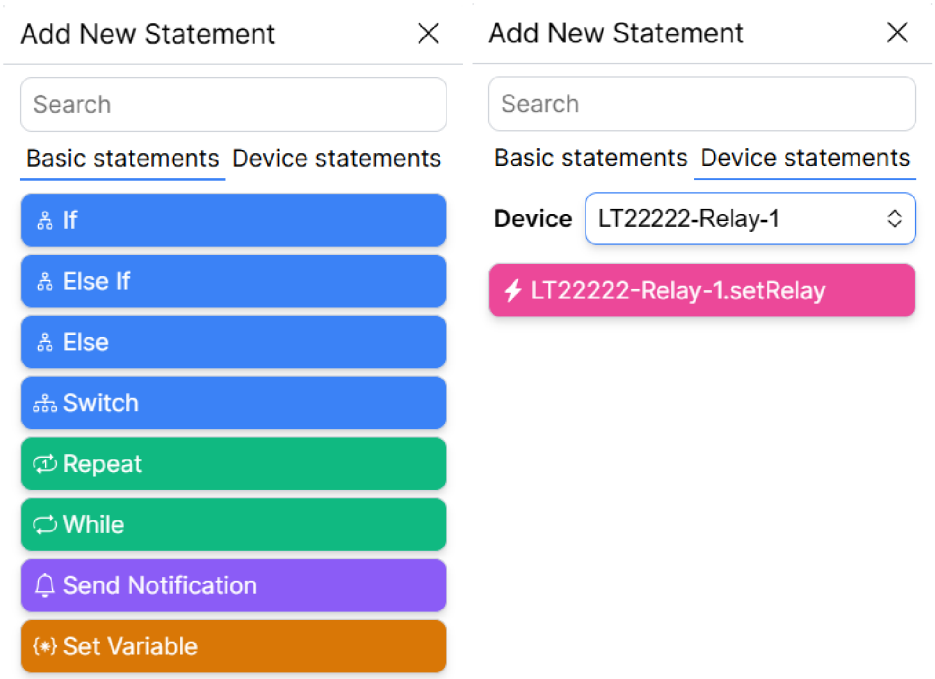
Hlavním komponentem je `vpl-editor`. Tento komponent zaobaluje všechny ostatní použité komponenty a slouží jako počáteční bod pro vykreslení celého editoru. Taktéž poskytuje kontext pro jazyk a program. Na obrázku 6.3 je vidět rozhraní editoru s příkladem vytvořeného programu. Editor se skládá ze dvou pohledů – grafického a textového.



Obrázek 6.3: Uživatelské rozhraní grafického editoru. V horní části obrázku se nachází ovládací prvky editoru. Na levé straně je grafický pohled, na straně pravé je textový pohled. Mezi pohledy lze libovolně přepínat pomocí výběru v ovládacích prvcích.

Grafický pohled (komponent `graphical-editor`) je tvořen blokem jednotlivých příkazů (komponent `ge-block`). Příkazy odpovídají komponentu `ge-statement`. Komponenty `ge-block` a `ge-statement` jsou rekurzivně vykreslovány, jelikož některé příkazy mohou obsahovat další vnořené bloky příkazů. Příkazy mají vizuální vlastnosti v podobě názvu, ikony, barvy pozadí a popředí. Tyto vlastnosti slouží k lepší identifikaci a kategorizaci příkazů. Kromě vizuálních vlastností jsou u každého příkazu definována syntaktická pravidla jako například datový typ a počet vstupních argumentů nebo strukturální omezení.

Uživatelé mohou přidávat nové příkazy do programu přes tlačítko se symbolem „+“. S příkazy lze v programu následně manipulovat (měnit jejich pozici) nebo je odstranit. Na obrázku 6.4 lze vidět modalové okno s dostupnými příkazy pro přidání. V seznamu příkazů lze také snadno vyhledávat pomocí vyhledávacího pole. Aktuální hodnota vyhledávacího pole je uložena ve vlastnosti `addStatementOptionsFilter` komponentu `ge-block` a je aktualizována při změně uživatelského vstupu. Metoda `filteredAddStatementOptions` vrací objekt s příkazy jazyka, jejichž identifikátor odpovídá hodnotě vyhledávacího pole. Tato metoda je automaticky volána při změně hodnoty vyhledávacího pole. Ve výpisu 6.2 lze vidět zjednodušená logika filtrování příkazů jazyka.



Obrázek 6.4: Modalové okno pro přidání příkazu. Příkazy jsou rozděleny na základní příkazy a příkazy zařízení. Příkazy jsou také barevně odlišeny podle jejich typu (podmínky, smyčky, manipulace s proměnnými, příkazy pro IoT).

```
statementKeysAndLabels = statementKeysAndLabels.filter((stmt) => {
  if (this.addStatementOptionsFilter) {
    return stmt.label.toLowerCase().includes(
      this.addStatementOptionsFilter.toLowerCase()
    );
  }
  return true;
});
```

Výpis 6.2: Implementace logiky filtrování při přidávání příkazu. Díky použití metody `toLowerCase` vyhledávání nerozlišuje velká a malá písmena.

Textový pohled odpovídá komponentu `text-editor`. V textové oblasti je zobrazena objektová reprezentace výsledného programu ve formátu JSON (viz. obrázek 6.5). Zkušenější uživatelé tak mohou za pomocí textového pohledu přímo upravovat objektovou reprezentaci programu. Prvotní implementace textového pohledu byla realizována integrací knihovny *monaco-editor*[1]. Monaco je webová verze editoru, který využívá populární textový editor *Visual Studio Code*. Integrací této knihovny textový editor podporoval zvýraznění syntaxe, automatické doplňování, čísla řádků a další funkce. Jednalo se však o velkou závislost, která by musela být zahrnuta při procesu sestavení knihovny. Z tohoto důvodu byl textový pohled

[1]Oficiální webové stránky knihovny monaco-editor: https://microsoft.github.io/monaco-editor/

nakonec implementován za pomocí HTML elementu `textarea`. Tím se výrazně zmenšila velikost výsledného balíčku knihovny. Pro textovou oblast bylo z důvodu lepší čitelnosti použito písmo *JetBrains Mono*[2].

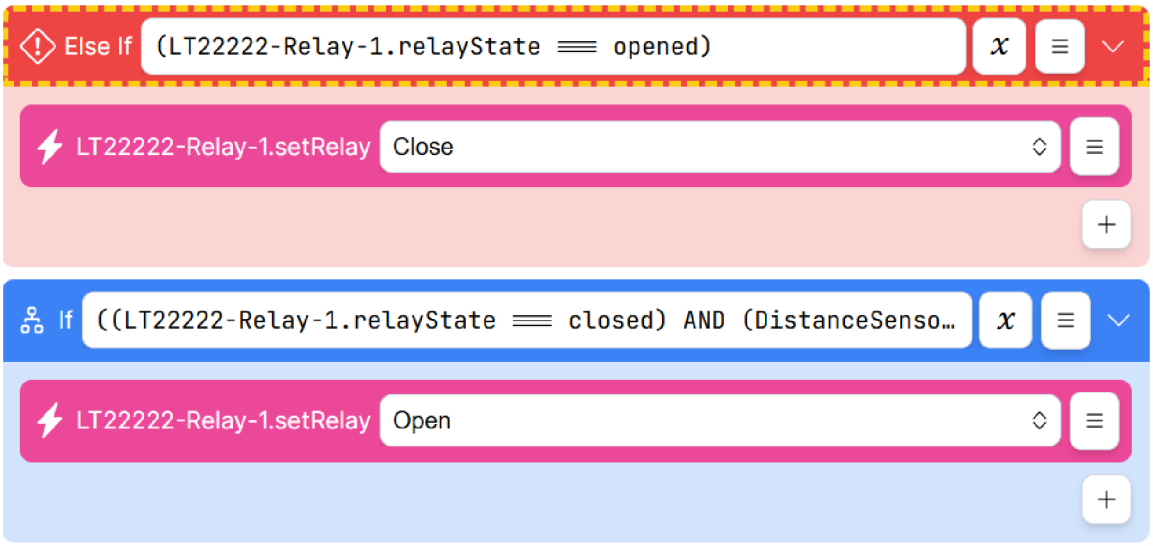
Obrázek 6.5: Ukázka výsledného programu v textovém pohledu. Na levé straně je korektní verze programu. Na pravé straně je verze programu se syntaktickou chybou. Na tuto chybu je uživatel vizuálně upozorněn změnou barvy textu.

Syntaktická omezení příkazů

Syntaktická omezení příkazů jsou kontrolována pomocí statické analýzy. Na obrázku 6.6 je ukázka chybně použitého příkazu. Statická analýza je prováděna při jakékoliv změně v objektové reprezentaci programu voláním funkce `analyzeBlock`, kterou poskytuje modul pro práci s programem (soubor `program.ts`). Funkce `analyzeBlock` rekurzivně provádí kontrolu jednotlivých syntaktických omezení, korespondujících s definicí příkazu jazyka. Ve výpisu 6.3 lze vidět příklad kontroly předchůdce příkazu.

Syntax je také zohledněna při vykreslování argumentů příkazu. Argumenty příkazů mohou nabývat různých typů. Na základě typu se poté vykreslí vhodný element. Například příkaz *If* má definován vstupní argument typu výraz, tudíž se vykreslí tlačítko, zobrazující textovou reprezentaci výrazu. Po kliknutí na tlačítko se otevře modalové okno zobrazující stromovou strukturu výrazu, kde uživatel může výraz vytvořit nebo upravit (viz. obrázek 6.7). Příkaz *repeat* má vstupní argument typu číslo, které určuje počet opakování zanořených příkazů. V tomto případě je uživateli ihned poskytnuta možnost změnit počet opakování, jak lze vidět na obrázku 6.8, protože vstupní argument se vykreslí jako číselný vstupní element.

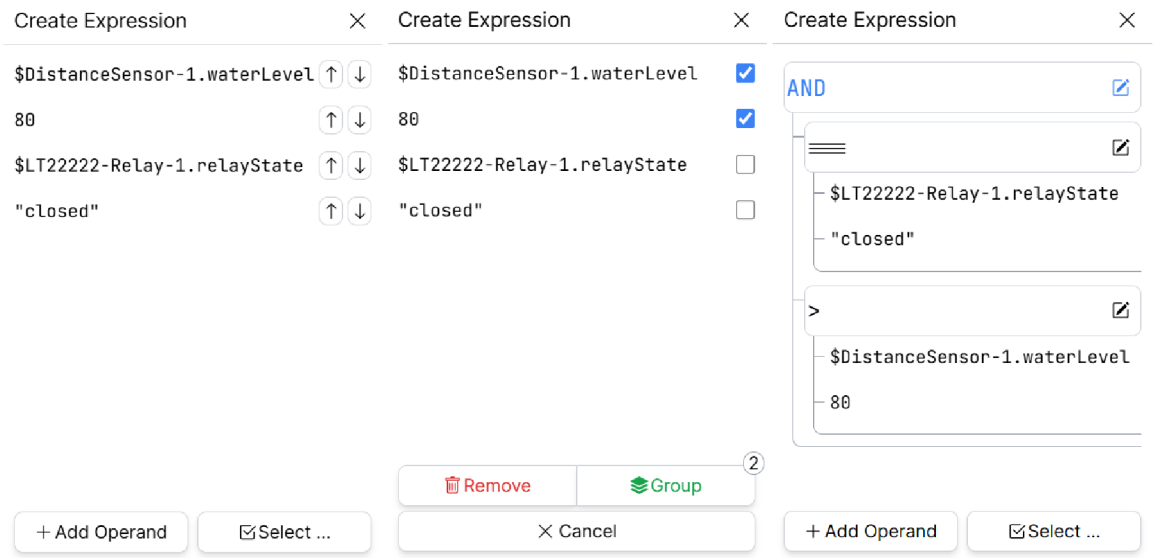
[2]Oficiální webové stránky písma JetBrains Mono: https://www.jetbrains.com/lp/mono/



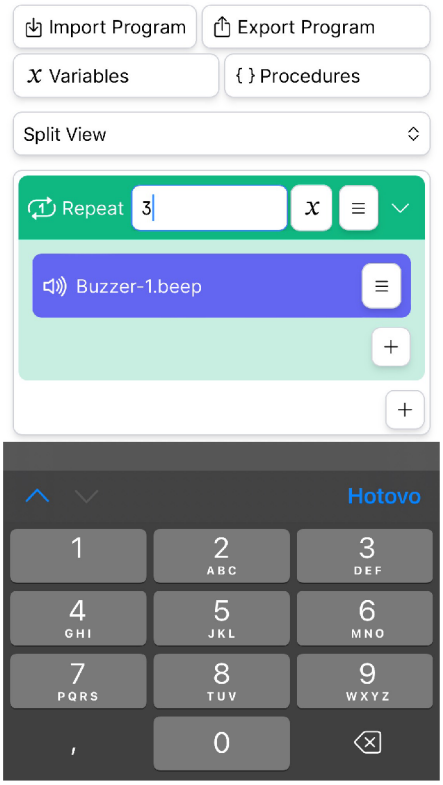
Obrázek 6.6: Program pro ovládání čerpadla retenční nádrže. Příkaz *Else If* je na nesprávné pozici. Uživatel je na tento fakt vizuálně upozorněn změnou barvy a ikony příkazu.

```
if (currentLangStmt.predecessors) {
  if (!previousPrgStmt) {
    currentPrgStmt.isInvalid = true;
    continue;
  }
  if (!currentLangStmt.predecessors.includes(previousPrgStmt.id)) {
    currentPrgStmt.isInvalid = true;
    continue;
  }
}
```

Výpis 6.3: Část funkce `analyzeBlock`, která kontroluje zda má příkaz správného předchůdce. V případě chyby se ke konkrétnímu příkazu nastaví příznak chyby, který je následně reflektován při vykreslování příkazu.



Obrázek 6.7: Příklad postupu vytváření výrazu. Na obrázku vlevo jsou zobrazeny uživatelem přidané operandy. Na obrázku uprostřed je aktivní režim seskupení, kde si uživatel označí potřebné operandy a vztah mezi nimi. Na obrázku vpravo je vidět výsledné seskupení v komplexnějším výrazu.

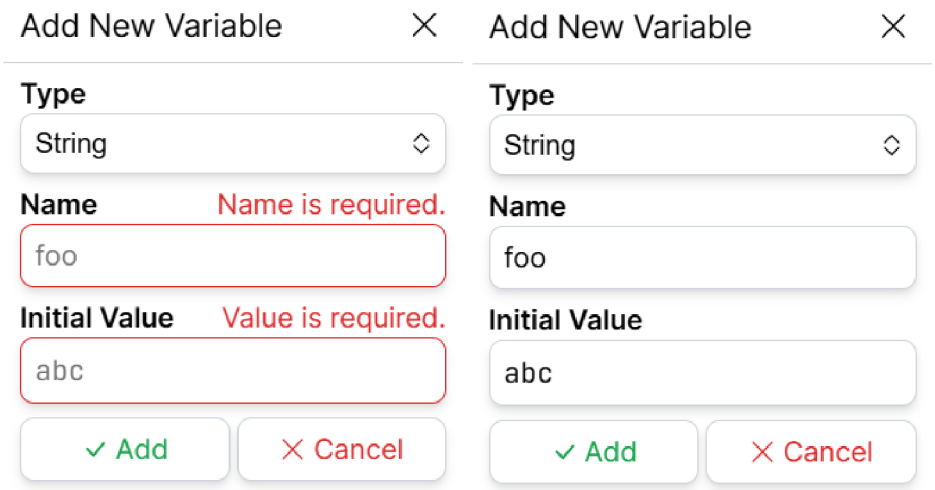


Obrázek 6.8: Příklad zobrazení editoru na mobilním zařízení. Uživateli je automaticky nabídnuta číselná forma vstupu. Výsledný program třikrát spustí funkci pro rozeznění bzučáku.

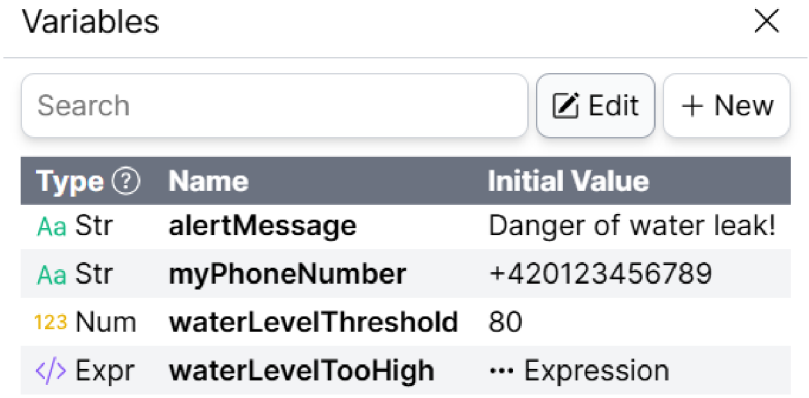
Uživatelské proměnné a procedury

Uživateli je umožněno vytvářet proměnné, které mohou být použity v argumentech příkazů či jako operandy ve výrazech. Při vytváření proměnné uživatel specifikuje její unikátní název a počáteční hodnotu (viz. obrázek 6.9). Definování počáteční hodnoty při vytváření proměnné je pro uživatele více přívětivé, než postupné nastavování hodnot proměnných na začátku programu pomocí příkazu *Set Variable*. Také se může stát, že uživatel by hodnotu proměnné zapomněl nastavit.

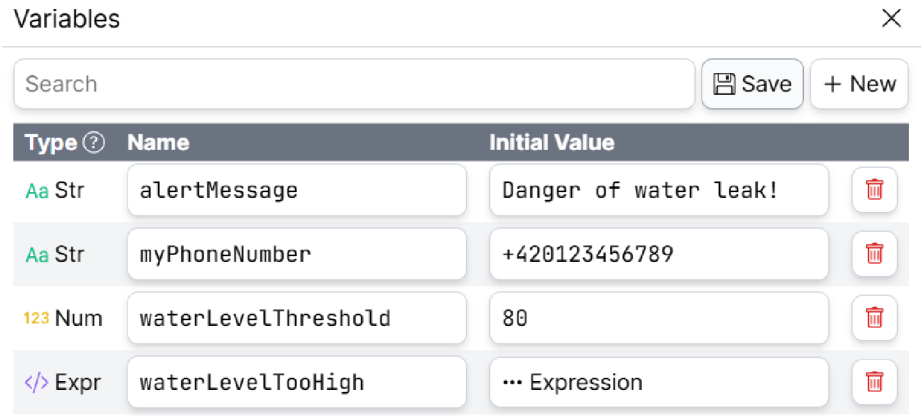
Uživatelské proměnné jsou zobrazeny formou tabulky. Tabulka proměnných poskytuje režim zobrazení a editace. Na obrázku 6.10 lze vidět režim zobrazení, který je kompaktnější, méně rušivý a umožňuje lepší zobrazení delších názvů proměnných a počátečních hodnot. V režimu editace může uživatel upravovat vlastnosti více proměnných najednou (viz. obrázek 6.11).



Obrázek 6.9: Modalové okno pro vytvoření proměnné. Na obrázku vlevo je ukázka pokusu o vytvoření proměnné s prázdným názvem a počáteční hodnotou. Uživatelský vstup je validován a uživatel upozorněn na chybné vstupní hodnoty. Na obrázku vpravo jsou všechny vstupy korektně vyplněny.

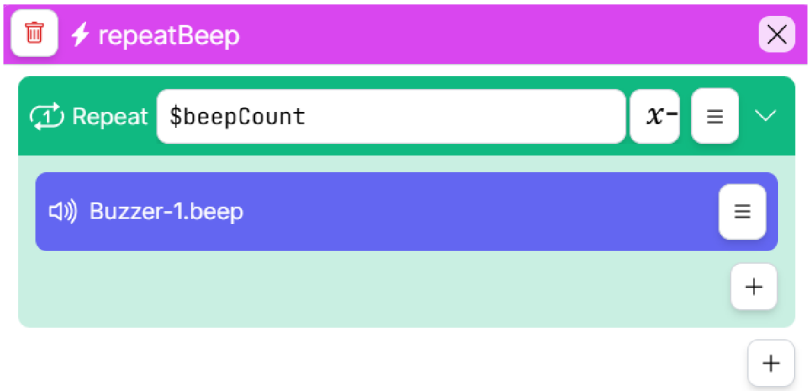


Obrázek 6.10: Režim zobrazení uživatelských proměnných.



Obrázek 6.11: Režim editace uživatelských proměnných.

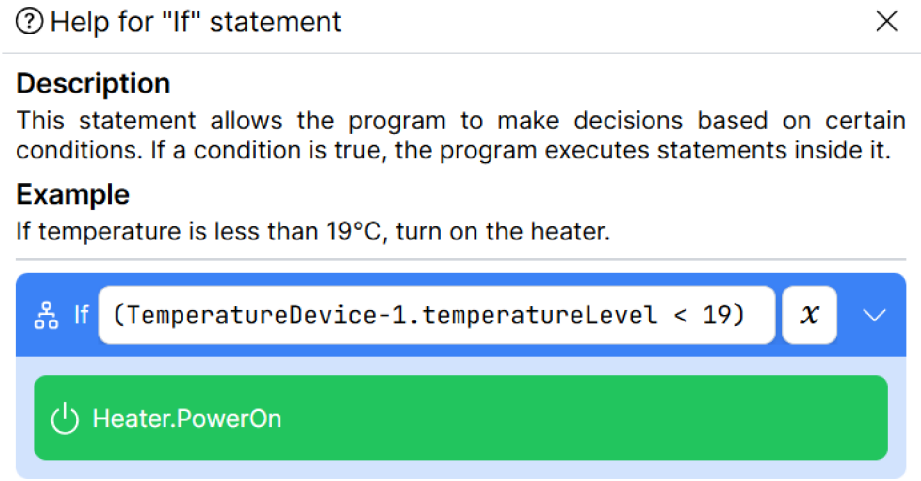
Uživatelské procedury slouží pro definování znovupoužitelných částí programu, nebo pouze pro zapouzdření a abstrakci složitější logiky. Uživatelé si při vytvoření procedury mohou zvolit její název a vizuální vlastnosti v podobě barvy pozadí, barvy textu a ikony. Po vytvoření si uživatel definuje funkcionalitu procedury stejným způsobem jako při vytváření hlavního programu (viz. obrázek 6.12). Procedury fungují na stejném principu jako ostatní příkazy a mohou být použity v těle hlavního programu. Při kliknutí na proceduru v hlavním programu se uživateli zobrazí modalové okno s definicí procedury, která může být následně upravena. Při exportování programu jsou jeho součástí i uživatelské proměnné a definice procedur. Mohou se tak sdílet i programy, které využívají proměnné a procedury.



Obrázek 6.12: Příklad definice uživatelské procedury, která opakuje rozeznění bzučáku. Počet opakování závisí na uživatelské proměnné `$beepCount`.

Uživatelská nápověda

Uživatelům je poskytnuta nápověda k jednotlivým příkazům. V grafickém pohledu se při kliknutí na ikonu příkazu otevře modalové okno s nápovědou. Nápověda obsahuje stručný textový popis příkazu a příklad jeho použití, jak lze vidět na obrázku 6.13. Nápověda příkazů je součástí definice jazyka (viz. výpis 6.4). Vývojářům je umožněno definovat svou vlastní nápovědu pro každý příkaz. Tento systém nápovědy přispívá k obecnosti celého řešení.



Obrázek 6.13: Nápověda pro příkaz *If*. Uživateli je prezentován konkrétní příklad použití příkazu, což napomáhá k lepšímu pochopení funkcionality.

```
description: {
     brief: 'This statement allows the program to make decisions based on
         certain conditions. If a condition is true, the program executes
         statements inside it.',
     example: {
       description: 'If temperature is less than 19C, turn on the heater.',
       block: [
         {
           id: 'if',
           block: [ { id: '_heaterOn', }, ],
           args: [
             {
               type: 'bool_expr',
               value: {
                 opds: [
                   {
                     opds: [ { type: 'var', value:
                               'TemperatureDevice-1.temperatureLevel' },
                             { type: 'num', value: 19 } ],
                     opr: '<',
                   },
                 ],
               },
             },
           ],
         },
       ],
     },
   },
```

Výpis 6.4: Objektová definice nápovědy pro příkaz *If*, která je součástí definice příkazu jazyka.

Kapitola 7

Testování

Uživatelské testování probíhalo buď formou individuální osobní schůzky nebo schůzky online přes videohovor, kde jsem byl přítomen po celou dobu testování. Celkem bylo osloveno 5 osob ve věku 22 až 30 let, z toho 4 muži a 1 žena. Účastníkům byl nejdříve představen koncept vizuálního editoru a následně položeny základní otázky, které pomohly k jejich kategorizaci na základě zkušeností s programováním. Dva z dotazovaných účastníků uvedli, že jsou programátoři. Zbytek uvedl, že mají pouze velmi základní zkušenosti s programováním ze základní nebo střední školy. Na otázku „Pracoval/a jste někdy s vizuálním programovacím jazykem (nástrojem), jako je například: Scratch, Blockly nebo Node-RED?“ pouze jeden účastník odpověděl negativně. Zbytek se alespoň jednou s nástroji vizuálního programování setkal.

Další z otázek byla: „Na jakém zařízení primárně pracujete?“. Záměrem této otázky bylo zjistit, zda účastníci využívají více mobilní či desktopová zařízení. Účastníci uvedli, že spíše používají stolní počítač s externím displejem nebo laptop. Při pokračování v konverzaci se však zmínili, že by očekávali podporu editoru na mobilních zařízeních pro rychlou úpravu programu v případě nedostupnosti jejich primárního zařízení.

Účastníkům byl následně představen vizuální editor a stručně popsány jeho základní prvky. Účastníci si poté samostatně zkusili navigaci v editoru a použití základní funkcionality. Byl taktéž vysvětlen systém příkazů, proměnných a zařízení, aby účastníci věděli, jakou funkcionalitu mohou při tvorbě programu využít. Z pozorování účastníků při práci a následného dotazování jsem usoudil, že navigace v prostředí editoru je až na pár výjimek intuitivní.

7.1 Testovací příklady

Po seznámení se základní funkcionalitou editoru jsem postupně účastníkům prezentoval sadu tří příkladů:

- **Příklad 1** – účastníci měli za úkol vytvořit program, který bude využívat zařízení v podobě chytrého zvonku u dveří. Zvonek v sobě integruje senzor pohybu a disponuje funkcionalitou pořízení fotografie. Výsledný program měl při detekci pohybu zaslat upozornění v podobě SMS zprávy a pořídit fotografii prostoru před dveřmi.
- **Příklad 2** – zadání tohoto příkladu vycházelo z jedné z *user stories* (popisu funkcionality z pohledu koncového uživatele), které mi byly zaslány zaměstnancem firmy Logimic. Cílem programu bylo automatizovat logiku přečerpávání vody v retenční nádrži. Při překročení horní hranice zaplnění nádrže se sepne relé, které ovládá čerpadlo.

Čerpadlo poté přečerpá vodu do jiné nádrže, aby nedošlo k jejímu přetečení. Účastníkům bylo zdůrazněno, že je potřeba ošetřit opakované zapínání a vypínání čerpadla kontrolou jeho stavu.

- **Příklad 3** – v posledním příkladu byl účastníkům popsán koncept chytrého podtácku, který pomocí barvy LED uživateli signalizuje teplotu nápoje a upozorní ho zvukovým signálem v případě poklesu teploty. Osoby měli za úkol pro každý z předdefinovaných teplotních rozsahů nastavit jinou barvu LED a při poklesu teploty na nejnižší úroveň třikrát aktivovat zvukový signál a nastavit teplotu zpět na ideální úroveň.

Příklady byly navrhnuty tak, aby reflektovali použití v reálném světě a zároveň se otestovala všechna funkcionalita editoru včetně použití uživatelských proměnných a procedur. Účastníci se následně pokusili vytvořit jednotlivé programy samostatně. Programy byly vytvářeny jak na desktopových, tak na mobilních zařízeních. Během vytváření programů jsem byl v případě dotazů k dispozici. Taktéž jsem účastníky požádal, aby při tvorbě programů přemýšleli nahlas a já tak mohl analyzovat jejich jednotlivé kroky. Po dokončení každého příkladu účastníci ohodnotili obtížnost jeho provedení a uvedli své poznámky ohledně změny či zlepšení editoru.

7.2 Výsledky testování

První příklad zvládli dokončit 3 účastníci samostatně a 2 s pomocí. Jeden účastník ohodnotil obtížnost úvodního příkladu jako standardní. Ostatní hodnotili příklad jako jednoduchý.

U druhého příkladu se všichni účastníci shodli na hodnocení obtížnosti jako standardní. I když bez pomoci ho dokázal dokončit pouze jeden z účastníků. V tomto příkladu bylo nutné použít více podmínek a vytvořit tak složitější výraz. Jakmile ale účastníci pochopili systém tvoření a seskupování výrazů neměli problém vytvořit i složitější zanořené výrazy. Při pozorování jsem zaregistroval, že účastníci se základními schopnostmi programování tvořili výrazy trochu jiným způsobem než programátoři. Nejdříve si postupně přidali všechny operandy, které byly ve výrazu/části výrazu potřeba a ty poté chtěli seskupit do vztahů. Programátoři chtěli vztahy definovat co nejdříve a postupně se v nich zanořovat.

Třetí příklad již po předchozích zkušenostech dokázali všichni účastníci dokončit bez problémů a samostatně. Všem účastníkům se příklady (bez nebo s pomocí) vždy podařilo korektně dokončit a nikdy se nestalo, že by účastník práci s editorem nezvládl.

Na závěr byli účastníci dotázáni, zda by využili knihovnu ve svém projektu, kde 3 účastníci odpověděli pozitivně a zda by knihovnu doporučili svým známým/kolegům pohybující se v oblasti IoT, kde 4 účastníci odpověděli „ano“ a jeden „možná“. Na základě zpětné vazby a poznámek z testování jsem opravil několik chyb:

- chybějící tlačítko pro zrušení při přidávání operandu ve výrazech
- chybějící potvrzovací tlačítko u modalového okna s více vstupními argumenty
- drobné vizuální chyby (odsazení, zarovnání)

a rozšířil funkcionalitu v podobě:

- přidání uživatelské nápovědy pro jednotlivé příkazy
- přidání rychlého náhledu procedury přímo v hlavním programu

Z uživatelského testování došlo k ověření splnění požadavků definovaných v sekci 4.2. Uživatelé jsou schopni vytvářet proměnné a procedury, byl implementován systém výrazů se stromovým zobrazením, editor je taktéž responzivní a funguje na různých typech zařízení.

Kapitola 8

Závěr

Cílem této práce bylo poskytnout koncovému uživateli využívající zařízení internetu věcí (IoT) nástroj, který by mu dovolil jednoduchým způsobem definovat vlastní logiku zařízení. Zároveň bylo také cílem tento nástroj koncipovat jako knihovnu, která může být využita vývojáři a integrována do existujících webových řešení. Oba cíle byly splněny a výsledkem této práce je nová knihovna implementující vizuální programovací jazyk, společně s grafickým editorem pro tvorbu programů na zařízení IoT. Výsledky práce byly prezentovány na konferenci Excel@FIT formou plakátu a krátkého článku [24].

Pro dosažení cíle práce bylo nejprve zapotřebí prostudování oblasti IoT. Dále jsem detailněji rozebral oblast vizuálního programování (VP). Analyzoval jsem existující nástroje VP a zaměřil se na jejich uživatelská rozhraní. Následně proběhla analýza existujícího řešení systému firmy Logimic a byly vyvozeny požadavky na nový grafický editor. Na základě požadavků jsem vytvořil návrh vizuálního programovacího jazyka a grafického uživatelského rozhraní editoru. Výsledný návrh jsem poté implementoval jako knihovnu v jazyce JavaScript. Kladl jsem důraz hlavně na použití co nejmenšího počtu závislostí. Z uživatelského testování a průběžných konzultací s vedoucím této práce vyplynulo pár nedostatků, které jsem adresoval a upravil výslednou verzi editoru.

Uživatelé mohou využít vizuální editor pro snadné definování programu na zařízení IoT bez znalosti klasického programování. Vizuální editor je responzivní a lze s ním pracovat na mobilních zařízeních. Mobilní verze editoru však není vhodná pro vytváření složitějších programů kvůli limitující velikosti obrazovky, ale dostačuje na menší úpravy programu. Vývojáři systémů IoT mohou knihovnu snadno integrovat do existujících webových řešení díky využití standardu webových komponent. Definice vizuálního programovacího jazyka může být taktéž upravena podle potřeb konkrétních typů systémů IoT.

Při dalším vývoji knihovny bych zkusil do vizuálního editoru implementovat *drag and drop* systém pro manipulaci s příkazy. Tento systém by uživatelům urychlil úpravu a refaktorizaci programu. Rovněž bych rozšířil uživatelskou nápovědu vizuálního editoru a udělal ji více interaktivní. Zajímavé by také bylo implementovat editor jako rozšíření do existujících vývojových prostředí jako je například *Visual Studio Code* a zaměřit se tak na více zkušené uživatele.

Výsledná knihovna řeší pouze generování serializované podoby programu. Při budoucím vývoji by bylo potřeba implementovat další část řešení, která by převedla serializovaný program do cílového kódu zařízení nebo rovnou program interpretovala a zasílala příkazy na koncová zařízení. Bylo by nutné formálně definovat přesnou syntax, sémantiku a gramatiku vizuálního programovacího jazyka, včetně výrazů.

Literatura

[1] Al Qaseemi, S. A., Almulhim, H. A., Almulhim, M. F. a Chaudhry, S. R. IoT architecture challenges and issues: Lack of standardization. In: IEEE. *2016 Future technologies conference (FTC)*. IEEE, 2016, s. 731–738 [cit. 2024-01-16]. DOI: 10.1109/FTC.2016.7821686. ISBN 978-1-5090-4171-8.

[2] Armoni, M., Meerbaum Salant, O. a Ben Ari, M. From scratch to „real“ programming. *ACM Transactions on Computing Education (TOCE)*. New York, NY, USA: Association for Computing Machinery. Únor 2015, sv. 14, č. 4, s. 1–15, [cit. 2023-12-15]. DOI: 10.1145/2677087.

[3] Badii, C., Bellini, P., Difino, A., Nesi, P., Pantaleo, G. et al. Microservices suite for smart city applications. *Sensors*. MDPI. Listopad 2019, sv. 19, č. 21, s. 4798, [cit. 2024-01-03]. DOI: 10.3390/s19214798. ISSN 1424-8220.

[4] Bak, N., Chang, B.-M. a Choi, K. Smart Block: A visual block language and its programming environment for IoT. *Journal of Computer Languages*. Elsevier. 2020, sv. 60, s. 100999, [cit. 2023-12-04]. DOI: 10.1016/j.cola.2020.100999. ISSN 2590-1184.

[5] Barricelli, B. R., Cassano, F., Fogli, D. a Piccinno, A. End-user development, end-user programming and end-user software engineering: A systematic mapping study. *Journal of Systems and Software*. Elsevier. 2019, sv. 149, s. 101–137, [cit. 2023-12-21]. DOI: https://doi.org/10.1016/j.jss.2018.11.041. ISSN 0164-1212.

[6] Danado, J. a Paternò, F. Puzzle: A mobile application development environment using a jigsaw metaphor. *Journal of Visual Languages & Computing*. Elsevier. 2014, sv. 25, č. 4, s. 297–315, [cit. 2023-11-06]. DOI: 10.1016/j.jvlc.2014.03.005. ISSN 1045-926X.

[7] Devalal, S. a Karthikeyan, A. LoRa technology-an overview. In: IEEE. *2018 second international conference on electronics, communication and aerospace technology (ICECA)*. IEEE, 2018, s. 284–290 [cit. 2024-01-22]. DOI: 10.1109/ICECA.2018.8474715. ISBN 978-1-5386-0965-1.

[8] Foundation, O. a Contributors. *Node-RED* [online]. 2023 [cit. 2023-11-20]. Dostupné z: https://nodered.org.

[9] Francese, R., Risi, M. a Tortora, G. Iconic languages: Towards end-user programming of mobile applications. *Journal of Visual Languages & Computing*. Elsevier. 2017, sv. 38, s. 1–8, [cit. 2023-12-21]. DOI: https://doi.org/10.1016/j.jvlc.2016.10.009. ISSN 1045-926X.

[10] Gokhale, P., Bhat, O. a Bhat, S. Introduction to IOT. *International Advanced Research Journal in Science, Engineering and Technology*. Leden 2018, sv. 5, č. 1, s. 41–44, [cit. 2024-01-03]. DOI: 10.17148/IARJSET.2018.517. ISSN 2393-8021.

[11] Gupta, B. B. a Quamara, M. An overview of Internet of Things (IoT): Architectural aspects, challenges, and protocols. *Concurrency and Computation: Practice and Experience*. Wiley Online Library. Září 2020, sv. 32, č. 21, [cit. 2024-01-14]. DOI: 10.1002/cpe.4946.

[12] Hynek, J., John, P., Formánková, K. a Valný, M. *Služby pro systém řízení a monitoringu vody v retenčních nádržích*. Výzkumná zpráva. Vysoké učení technické v Brně, 2023 [cit. 2023-12-21].

[13] Jost, B., Ketterl, M., Budde, R. a Leimbach, T. Graphical programming environments for educational robots: Open roberta-yet another one? In: IEEE. *2014 IEEE International Symposium on Multimedia*. IEEE, Prosinec 2014, s. 381–386 [cit. 2023-11-06]. DOI: 10.1109/ISM.2014.24. ISBN 978-1-4799-4311-1.

[14] Koitz, R. a Slany, W. Empirical comparison of visual to hybrid formula manipulation in educational programming languages for teenagers. In: *Proceedings of the 5th Workshop on Evaluation and Usability of Programming Languages and Tools*. New York, NY, USA: Association for Computing Machinery, říjen 2014, s. 21–30 [cit. 2023-12-15]. PLATEAU '14. DOI: 10.1145/2688204.2688209. ISBN 9781450322775.

[15] Komilov, D. R. Application of zigbee technology in IOT. *International Journal of Advance Scientific Research*. Září 2023, sv. 3, č. 09, s. 343–349, [cit. 2024-01-22]. DOI: 10.37547/ijasr-03-09-54.

[16] Kuhail, M. A., Farooq, S., Hammad, R. a Bahja, M. Characterizing visual programming approaches for end-user developers: A systematic review. *IEEE Access*. IEEE. 2021, sv. 9, s. 14181–14202, [cit. 2023-10-23]. DOI: 10.1109/ACCESS.2021.3051043. ISSN 2169-3536.

[17] *Lit – What is Lit?* [online]. [cit. 2024-03-17]. Dostupné z: https://lit.dev/docs/.

[18] *Lit – Defining a component* [online]. [cit. 2024-03-17]. Dostupné z: https://lit.dev/docs/components/defining/.

[19] Madakam, S., Lake, V., Lake, V., Lake, V. et al. Internet of Things (IoT): A literature review. *Journal of Computer and Communications*. Scientific Research Publishing. Duben 2015, sv. 3, č. 05, s. 164–173, [cit. 2024-01-16]. DOI: 10.4236/jcc.2015.35021.

[20] *MDN – Web Components* [online]. 9. února 2024 [cit. 2024-03-17]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/API/Web_components.

[21] *MQTT: The Standard for IoT Messaging* [online]. [cit. 2024-01-05]. Dostupné z: https://mqtt.org/.

[22] Naik, N. Choice of effective messaging protocols for IoT systems: MQTT, CoAP, AMQP and HTTP. In: IEEE. *2017 IEEE international systems engineering symposium (ISSE)*. IEEE, 2017, s. 1–7 [cit. 2024-01-25]. DOI: 10.1109/SysEng.2017.8088251. ISBN 978-1-5386-3403-5.

[23] Paternò, F. End user development: Survey of an emerging field for empowering people. *International Scholarly Research Notices*. Hindawi Publishing Corporation. Červen 2013, sv. 2013, s. 532659, [cit. 2023-12-21]. DOI: 10.1155/2013/532659. ISSN 2356-7872.

[24] Podvojský, L. Visual Programming of IoT Devices. In: *Sborník konference Excel@FIT* [online]. Brno: Fakulta informačních technologií VUT v Brně, Květen 2024 [cit. 2025-05-07]. Dostupné z: https://excel.fit.vutbr.cz/submissions/2024/041/41.pdf.

[25] Pratomo, A. B. a Perdana, R. S. Arduviz, a visual programming IDE for arduino. In: IEEE. *2017 International Conference on Data and Software Engineering (ICoDSE)*. IEEE, Listopad 2017, s. 1–6 [cit. 2023-12-04]. DOI: 10.1109/ICODSE.2017.8285871. ISBN 978-1-5386-1449-5.

[26] Ray, P. P. et al. A survey on visual programming languages in internet of things. *Scientific Programming*. Hindawi. Březen 2017, sv. 2017, s. 1–6, [cit. 2023-11-06]. DOI: 10.1155/2017/1231430. ISSN 1058-9244.

[27] Repenning, A. Moving Beyond Syntax: Lessons from 20 Years of Blocks Programing in AgentSheets. *Journal of Visual Languages and Sentient Systems*. Červenec 2017, sv. 3, č. 1, s. 68–91, [cit. 2023-11-06]. DOI: 10.18293/VLSS2017-010.

[28] Sáez López, J.-M., Román González, M. a Vázquez Cano, E. Visual programming languages integrated across the curriculum in elementary school: A two year case study using „Scratch“ in five schools. *Computers & Education*. Elsevier. 2016, sv. 97, s. 129–141, [cit. 2023-12-15]. DOI: 10.1016/j.compedu.2016.03.003. ISSN 0360-1315.

[29] *About Scratch* [online]. [cit. 2023-12-04]. Dostupné z: https://scratch.mit.edu/about.

[30] Sobin, C. C. A survey on architecture, protocols and challenges in IoT. *Wireless Personal Communications*. Springer. Červen 2020, sv. 112, č. 3, s. 1383–1429, [cit. 2024-01-16]. DOI: 10.1007/s11277-020-07108-5. ISSN 1572-834X.

[31] Valderas, P., Torres, V., Mansanet, I. a Pelechano, V. A mobile-based solution for supporting end-users in the composition of services. *Multimedia Tools and Applications*. Springer. Srpen 2017, sv. 76, č. 15, s. 16315–16345, [cit. 2023-11-06]. DOI: 10.1007/s11042-016-3910-4. ISSN 1573-7721.

[32] *What are web components?* [online]. [cit. 2024-03-17]. Dostupné z: https://www.webcomponents.org/introduction.

[33] Wikipedie. *Blockly* [online]. 1. dubna 2021 [cit. 2023-12-03]. Dostupné z: https://cs.wikipedia.org/w/index.php?title=Blockly&oldid=19670766.

[34] Wikipedie. *Wi-Fi* [online]. 14. ledna 2024 [cit. 2024-01-21]. Dostupné z: https://cs.wikipedia.org/w/index.php?title=Wi-Fi&oldid=23562095.

[35] Zhang, K. *Visual languages and applications*. 1. vyd. Springer Science & Business Media, březen 2007 [cit. 2023-11-06]. ISBN 978-0-387-68257-0.

Příloha A

Příklad definice zařízení

```
const exampleDevices: Device[] = [
  {
    deviceName: 'DistanceSensor-1',
    attributes: ['waterLevel'],
    functions: [],
  },
  {
    deviceName: 'LT22222-Relay-1',
    attributes: ['relayState'], // "opened", "closed"
    functions: [
      {
        type: 'unit_with_args',
        args: [
          {
            type: 'str_opt',
            options: [
              { id: 'open', label: 'Open' },
              { id: 'close', label: 'Close' },
            ],
          },
        ],
        backgroundColor: '#ec4899',
        foregroundColor: '#ffffff',
        label: 'setRelay',
        icon: 'lightningChargeFill',
        group: 'iot',
      },
    ],
  }
]
```

Příloha B

Příklad výsledného programu


```
{
  "header": {
    "userVariables": {
      "myPhoneNumber": {
        "type": "str",
        "value": "+420123456789"
      },
      "alertMessage": {
        "type": "str",
        "value": "Movement detected!"
      }
    },
    "userProcedures": {}
  },
  "block": [
    {
      "id": "if",
      "block": [
        {
          "id": "alert",
          "args": [
            {
              "type": "str_opt",
              "value": "phone_number"
            },
            {
              "type": "var",
              "value": "myPhoneNumber"
            },
            {
              "type": "var",
              "value": "alertMessage"
            }
          ]
        },
        {
          "id": "Doorbell-1.takePicture"
        }
      ],
```

```
      "args": [
        {
          "type": "bool_expr",
          "value": {
            "opds": [
              {
                "opds": [
                  {
                    "type": "var",
                    "value": "Doorbell-1.motionSensor"
                  },
                  {
                    "type": "str",
                    "value": "active"
                  }
                ],
                "opr": "==="
              }
            ]
          }
        }
      ]
    }
  ]
}
```