BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INFORMATION SYSTEMS
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

VISUAL PROGRAMMING LANGUAGE EVALUATION SIMULATOR

SIMULÁTOR VYHODNOCENÍ VISUÁLNÍHO PROGRAMOVACÍHO JAZYKA

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR **Bc. FILIP ŠTOLFA**
AUTOR PRÁCE

SUPERVISOR **Ing. PETR JOHN**
VEDOUCÍ PRÁCE

BRNO 2025

BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Master's Thesis Assignment

161062

Institut:	Department of Information Systems (DIFS)
Student:	**Štolfa Filip, Bc.**
Programme:	Information Technology and Artificial Intelligence
Specialization:	Machine Learning
Title:	**Visual Programming Language Evaluation Simulator**
Category:	Web applications
Academic year:	2024/25

Assignment:

1. Study the Internet of Things (IoT) and Smart City.
2. Study the principles of visual programming language (VPL) representation, focus on visual languages applicable to IoT.
3. Analyze the requirements for simulators and debuggers for visual programming languages and the current way of evaluating VPLs.
4. Propose an appropriate way to simulate the execution of programs, including evaluation and visualizations.
5. After consultation with the supervisor, implement the proposed solution.
6. Test the resulting solution for usability and evaluate it. Suggest possible extensions.

Literature:

- Greengard, S. (2015). The Internet of Things. MIT Press, ISBN 978-026-2527-736.
- 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. Research report. Brno University of Technology.
- Kuhail, M. A., Farooq, S., Hammad, R., & Bahja, M. (2021). Characterizing visual programming approaches for end-user developers: A systematic review. IEEE Access, 9, (pp. 14181-14202).
- Ray, P. P. (2017). A survey on visual programming languages in internet of things. *Scientific Programming*, *2017*.
- John, P. (2024). Optimising processes in IoT. Brno. PhD thesis proposal University of Technology, Faculty of Information Technology. Supervisor prof. Ing. Tomáš Hruška, CSc.
- Bureš, M. (2024). Systém pro zpracování dat z chytrých zařízení, Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Jiří Hynek, Ph.D.

Requirements for the semestral defence:

- Points 1 - 4.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor:	**John Petr, Ing.**
Head of Department:	Kolář Dušan, doc. Dr. Ing.
Beginning of work:	1.11.2024
Submission deadline:	21.5.2025
Approval date:	22.10.2024

Abstract

As the popularity and number of smart devices grows, so does the need for robust tooling aimed at novice and experienced users alike, allowing them to utilize their devices to their full potential. Users of IoT systems create user defined programs to connect their devices into a useful smart system. This master's thesis deals with creating a user-friendly debugger and simulator for a visual programming language, which enables end users to test and better understand their programs using visual cues. It focuses especially on how different programs interact with each other and how they affect devices in the system. The goal of this thesis is to design and implement a library for parsing, evaluating and debugging user defined programs inside a simulated environment, to prevent potential harm of running faulty programs on real devices. The library also contains visual components, which can be used to build a responsive user interface for the underlying debugger and simulator. An example application is built to showcase the capabilities of the implemented solution.

Abstrakt

S rostoucí popularitou a počtem chytrých zařízení roste i potřeba kvalitních nástrojů určených pro začínající i zkušené uživatele, které jim umožní využít plný potenciál jejich zařízení. Uživatelé systémů IoT vytváří uživatelsky definované programy, které propojují jejich zařízení do užitečného chytrého systému. Tato diplomová práce se zabývá vytvořením uživatelsky přívětivého debuggeru a simulátoru pro vizuální programovací jazyk, který koncovým uživatelům umožní testovat a lépe pochopit jejich programy pomocí vizuálních nápověd. Zaměřuje se zejména na objasnění interakce programů a na jejich vliv na zařízení v systému. Cílem práce bylo navrhnout a implementovat knihovnu pro zpracování, vyhodnocování a ladění uživatelsky definovaných programů v simulovaném prostředí, čímž lze předejít škodám, která mohou nastat spouštěním chybných programů na reálných zařízeních. Knihovna obsahuje také vizuální komponenty, ze kterých je možné vytvořit responzivní uživatelské rozhraní pro implementovaný debugger a simulátor. Pro demonstraci implementovaného řešení byla sestavena ukázková aplikace.

Keywords

debugger, simulator, IoT, visual programming, smart devices, TypeScript, Lit, responsive design

Klíčová slova

debugger, simulátor, IoT, vizuální programování, chytrá zařízení, TypeScript, Lit, responzivní design

Reference

ŠTOLFA, Filip. *Visual Programming Language Evaluation Simulator*. Brno, 2025. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Petr John

Rozšířený abstrakt

Počet chytrých zařízení v posledních letech stále roste. Nachází využití ve velké řadě různorodých oblastí, jako jsou například domácnosti, chytrá města, nebo zemědělství. Pojmem chytré zařízení se myslí zařízení schopné vnímat, nebo ovlivňovat své okolí a komunikovat s jinými zařízeními, nebo počítači. Takováto zařízení jsou součástí takzvaného internetu věcí (angl. Internet of Things, IoT), mimo jiné právě kvůli jejich schopnosti komunikovat. Příkladem chytrého zařízení je chytrá stmívatelná žárovka, bezdrátový teploměr, spínatelná zásuvka, nebo senzor otevření okna či dveří.

Způsob, jak zařízení komunikují, se liší podle daného odvětví a specifických potřeb každého nasazení. Chytrá zařízení velmi často komunikují nějakou bezdrátovou technologií, jelikož ta v porovnání s drátovým připojením přináší značnou flexibilitu při instalaci zařízení. Nevýhodou ovšem je menší spolehlivost připojení. Podobně je to se způsobem napájení zařízení. Bateriové napájení přináší značnou flexibilitu v umístění zařízení, za cenu nutnosti nahrazovat, či nabíjet baterie a přepínání zařízení mezi aktivním a úsporným stavem.

Aby se chytrá zařízení stala opravdu užitečná, je potřeba je navzájem propojit, aby bylo možné využít informací z jednoho zařízení pro ovlivnění stavu jiného zařízení. Touto spoluprací se z jednoduchých zařízení stává chytrý systém.

Jelikož propojení zařízení musí být provedeno na míru danému uživateli a jeho systému, je potřeba, aby samotný uživatel byl schopen určit, jak spolu mají jednotlivá zařízení spolupracovat. Za tímto účelem se v IoT systémech často mimo jiné využívají visuální programovací jazyky. Oproti běžným textovým programovacím jazykům nabízí přívětivější prostředí pro vytváření uživatelských programů. Jedním z důvodů je možnost zabránění vzniku syntaktických chyb, právě díky použití vizuální metafory, která umožňuje editoru takového jazyka nabízet uživateli pouze syntakticky korektní prvky.

I přes jejich výhody, visuální programovací jazyky nezabrání logickým chybám v uživatelských programech. Pro jejich odhalení je užitečné mít možnost si program projít po jednotlivých krocích. Tento postup je užitečný nejen pro zkušené uživatele, ale zejména potom pro začátečníky, kteří nemají s programováním zkušenosti a nemají tak zažité všechny potřebné koncepty. V programování je tento koncept běžný a pro jeho realizaci se využívá specializovaný nástroj nazývaný debugger. Ten umožňuje pozastavit vykonávání programu a následně tento program provádět po jednotlivých krocích. V každém momentě také zobrazuje stav systému ve kterém program běží, aby si jej programátor mohl prohlédnout a ověřit si, zda je vše tak, jak by očekával.

Využití debuggerů v IoT systémech komplikuje závislost programů na stavu jednotlivých zařízení v daném systému. Hlavním účelem uživatelských programů v IoT je změna stavu jednoho zařízení na základě stavu jiného. Pro sledování chování programů za různých podmínek by tedy bylo nutné pokaždé změnit stav reálných zařízení, což není praktické. Často to ani není možné. Pokud například chceme otestovat program, který má upustit nádrž tak, aby nedošlo k jejímu přetečení, není žádoucí, aby každé spuštění programu v debuggeru vyžadovalo napuštění nádrže nad limitní hranici a její následné vypuštění. Proto je nutné mít možnost testovat uživatelské programy v simulovaném prostředí, kde je možné vyzkoušet všechny situace, bez obav ze způsobení škody. Dnešním IoT systémům tato funkcionalita schází a pro koncové uživatele je potom velice obtížné ověřit, zda jejich programy mají zamýšlený efekt.

Cílem této diplomové práce bylo navrhnout a implementovat debugger a simulátor pro uživatelské programy v IoT systémech. Díky poskytnutému simulátoru zařízení v daném IoT systému je možné bez obav testovat programy, které by mohly při chybném spuštění na

reálných zařízeních způsobit škody. Simulátor dále také zjednodušuje testování programů s takovým stavem zařízení, kterého by bylo komplikované dosáhnout za použití reálných zařízení. Tento systém umožní zejména méně zkušeným uživatelům využít jejich zařízení naplno a nebát se experimentovat s novými způsoby propojení jejich zařízení.

Pro tuto práci byl zvolen již existující vizuální programovací jazyk a jeho editor, který je zaměřen na použití právě v IoT systémech. Systém byl navržen jako knihovna pro webovou platformu, kterou je možné zakomponovat do libovolného webového uživatelského rozhraní. Knihovna je rozdělena na tři části, kde první implementuje logiku pro zpracování zdrojového kódu zvoleného vizuálního programovacího jazyka do interní reprezentace a jeho následné vyhodnocení, které je možné pozastavit implementovaným debuggerem. Druhá část knihovny implementuje simulátor pro chytrá zařízení. Simulátor umožňuje číst stav všech simulovaných zařízení a také zařízením posílat zprávy, na které mohou reagovat právě změnou svého stavu. Jak zařízení reaguje na danou zprávu, je definováno v rozšířené specifikaci zařízení, kterou musí poskytnout IoT systém, ve kterém je tento simulátor využíván. Třetí část knihovny obsahuje vizuální komponenty, ze kterých je možné sestavit bohaté uživatelské rozhraní pro debugger a simulátor zařízení. Při návrhu vizuálních komponent byl brán velký ohled na jejich responzivnost, aby je bylo možné snadno využít pro tvorbu uživatelského rozhraní na mobilní zařízení.

Součástí práce bylo i provedení změn ve zvoleném editoru vizuálního programovacího jazyka, aby byl lépe použitelný na mobilních zařízeních. Další změny se týkaly zpřehlednění zobrazování výrazů, aby bylo pro uživatele snazší na první pohled poznat, co daný výraz znamená, zda je v něm použita nějaká proměnná či zda se v něm čte stav nějakého zařízení. Byla také přidána možnost zobrazovat chybové a varovací hlášky, které jsou produkovány při zpracování programu výše zmíněnou první částí knihovny. Aby bylo možné použít tento stejný editor pro zobrazení pozastaveného programu v debuggeru, byla přidána možnost zvýraznění libovolného bloku programu.

Uživatelé mohou v systému vytvářet testovací scénáře, pomocí kterých mohou nasimulovat libovolné změny v jejich zařízeních tak, aby bylo možné otestovat, jak na tyto změny zareagují jimi vytvořené programy. Testovací scénáře nikdy neovlivňují reálná zařízení. Stejně tak programy vyvolané změnami popsanými v testovacím scénáři nemohou ovlivnit reálná zařízení. Jejich výstup je vždy využit pouze ke změně stavu simulovaných zařízení. Do testovacích scénářů je také možné přidat příkazy, které v daný okamžik otestují, zda stav simulovaných zařízení odpovídá očekávání uživatele. Pokud tedy uživatel vytvoří testovací scénář, na jehož začátku provede simulované zmáčknutí vypínače světla, může do scénáře přidat příkaz, který otestuje, že nejpozději za jednu sekundu bude světlo ovládané tímto vypínačem zapnuto.

Ukázková aplikace, která byla sestavena pro demonstraci využití navržené knihovny, byla testována uživateli se širokou škálou technických znalostí. Cílem testování bylo ověřit, jak rychle rozdílní uživatelé pochopí základní principy navrženého systému a zda jej jsou schopni samostatně využít k ověření funkčnosti programu a nalezení chyb. Na základě tohoto testování bylo navrženo několik možných rozšíření a vylepšení, které by mohly vést ke zpřehlednění a vylepšení aplikace.

Visual Programming Language Evaluation Simulator

Declaration

I hereby declare that this master's thesis was prepared as an original work by the author under the supervision of Ing. Petr John. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

.....
Filip Štolfa
May 21, 2025

Acknowledgements

I would like to sincerely thank Ing. Petr John for his guidance, supervision and patience during the development of this work. I would also like to thank Ing. Michal Valný, Ph.D. and Ing. Ondřej Šulc for their valuable feedback. Last but not least I would like to thank my friends and family for supporting me during the entire process.

Contents

List of Figures

Chapter 1

Introduction

Smart devices are capable of sensing or influencing their environment and communicating with other such devices, or computers. As their number grows, so does the need for robust and easy to use tools, which empower both novice and experienced users to utilize their smart devices to their full potential. When these devices are connected together, in order to exchange information and influence each other, they form networks which are referred to as the Intern of Things (IoT).

Smart devices, such as controllable light bulbs, environmental sensors, or autonomous vacuum cleaners find applications in a large number of areas. Such as smaller home networks, larger industrial applications, or entire citywide installations. The true usefulness of these devices only starts when they are connected together according to the specific needs of the system's user. This means, that to make their usage practical, users with different technical skill levels need to be able to create user defined programs, which specify how their devices should interact.

Today's systems often utilize visual programming languages, to make creation of user defined programs approachable by novice users, while not hindering more experienced users. These languages can help prevent syntactical errors, but they do not prevent logic errors. Users need to be able to test their programs, by running them and observing the results. IoT systems often don't provide any specific tooling to aid with this task, requiring users to test their programs with their real devices. This makes testing certain scenarios difficult, or impossible. To make it easy for users to test whether their programs do what they intended, specialized tools with simulated device support are required.

The goal of this master's thesis is to design and implement a flexible library, which would allow users to test and debug their programs in a simulated environment, that represents their IoT system. To achieve this, first a visual programming language and editor had to be chosen. Changes to the editor were required, to support debugger specific features, as well as to work better on mobile devices. A parser for this language was developed, in order to transform programs into an internal representation, suitable for evaluating the program inside a debugger. The debugger was designed with novice users and IoT systems in mind. A smart device simulator was designed, which makes it possible to run and observe user defined programs without modifying any real devices, as well as testing programs in situations which would be difficult to set up if real device state was used. The library also includes visual components, which can be used to construct a rich user interface, designed to work well on both small and large screen devices.

An example application was built to demonstrate the usage of the implemented library. It provides editor views, where users can modify their programs, manage devices in their

system and construct test scenarios, which can then be used to test how their programs work and interact. Launching a given test scenario opens a view where the user can step through their scenario and see, what program is triggered when and how programs affect the state of their devices. This application was also used in user testing of the implementation.

Chapter 2 introduces the field of IoT in more detail. It talks about its use and applications, describes different commonly used system architectures, communication technologies and protocols. Chapter 3 presents different approaches to visual programming, as well as its advantages and uses. Debuggers and simulators are described in detail in Chapter 4. It begins with a general introduction to the concept, then delves into its use specifically for visual programming. It also highlights and describes existing solutions of IoT specific simulators and debuggers. Chapter 5 presents the proposed solution. It describes the solution's architecture and goes into detail about the design of each part in the system, including the user interface. The implementation of the proposed solution is documented in Chapter 6. Both automated and user testing of the implementation is described in Chapter 7.

Chapter 2

Internet of Things

Microcontrollers embedded in seemingly ordinary things are becoming ubiquitous. More and more devices which people use day to day are becoming smart and are being given the capabilities to communicate with other devices or computer systems, either through a wire or wirelessly. These so-called smart devices can report information about their environment gathered from embedded sensors, or they can act on their environment, either through their own decisions, or based on a command from outside themselves.

This phenomenon is called the Internet of things (IoT). The term is widely attributed to a presentation given by Kevin Ashton in 1999 [4], in which he discusses using RFID tags to help computers identify physical objects. Even tough today the term describes more complex systems, the general idea remains – allowing computers and computer systems to sense and effect the physical world around them. These systems are composed of things, which are able to communicate with each other. A device can range from traditional computers, smartphones and tablets, to more simple wearable devices like smart-watches and fitness trackers or even more specific IoT devices, such as every-day objects equipped with embedded computers. These can include things like smart light bulbs, doorbells, fridges and various standalone sensors.

The number of IoT devices is reported to be around 16.6 billion according to a report by IoT Analytics [41] and 15.9 billion as reported by Statista [42]. They predict the number of IoT devices to grow to 41.1 billion and 32.2 billion by 2030 respectively.

Devices in IoT can be broadly grouped into two categories: sensors and actuators. Sensors are devices, which observe the world around them and provide information about the current state of their environment to the IoT system. Common sensors include temperature and humidity sensors, light sensor, presence detectors for monitoring occupancy status of different rooms, water leakage detectors and many more. Actuators offer a way for the IoT system to interact with the physical world. This interaction can be operations such as turning on a light (thus increasing the level of light in an area, a change which can be independently measured and verified by a light sensor, allowing the system to confirm the actual state of the environment), opening or closing window blinds, or moving a robotic arm designed to manipulate items.

Since actuators have the capability to effect the physical world, they typically have stronger requirements when it comes to reliability. Often these types of devices will be powered from a persistent power source (like an outlet), since their operation often requires larger amounts of power, over a prolonged period of time (e.g. lights, room fan, motorized window blinds). Meanwhile sensors are often designed in such a way, which allows them to spend most of the time in an efficient sleep mode, thus reducing power consumption.

Devices like that can wake up for only a few seconds every X minutes or hours, or only when a significant change in the measured variable is detected. Meaning they can be battery powered, while lasting a long enough time, where their maintenance is not a nuisance to the users of the system. Not needing any wiring for their power allows them to be placed in hard to reach places, or simply in places, where running a wire would be impractical or impossible. This difference in operating modes effects what roles these types of device can play in the system. Devices which are not limited by their power supply can always be listening for commands from a central server, therefore are more suited for critical tasks, where communication latency is an issue. They can also be used as routers in mesh–style wireless networks, while battery powered devices can only act as end nodes, not being able to extend the reach of the network.

If a device contains more than just sensors or actuators, it can be labeled as a Smart Object [14]. These devices can be equipped with different sensors and actuators, and their behavior can be modified accordingly. Such devices can be classified based on their level of intelligence. The authors of [14] classify Smart Objects based on three dimensions: level of intelligence, location of the intelligence and aggregation level of the intelligence. According to their classification, Smart Objects can, for example, include smartphones, Arduino[1], or RaspberryPi[2] boards.

Each *thing* in IoT has a unique identification number, which makes the device addressable by all other devices in the system [15]. Depending on the system configuration, it might also have an IP address, making it addressable by any other device on the internet. Some configurations use IoT specific communications technologies and protocols, which do not support the internet protocol. Such systems use a gateway in order to connect the devices to the broader internet. Although in some systems, it may be desirable to not expose internal devices to the internet, allowing the system to control which data to share with external systems, or making the system available through local networks only [12]. By their very nature, IoT systems capture sensitive information, be it in the household, or in factories. Thus, privacy and security is a very important aspect to consider in the context of IoT.

2.1 IoT Use Cases and the Smart City Concept

Common use cases of IoT systems are to automate and streamline every day tasks, industrial processes and other similar tasks. As such, IoT is used in many different industries.

One possible usage of IoT systems is in healthcare. Bandyopadhyay [5] describes implantable and wearable medical devices that can be used to closely monitor patients with chronic illnesses, serving as an early warning system, if signs of their condition worsening are detected. Another use case is monitoring drug delivery through RFID tags, or providing implants for stimulating muscles for individuals with limited mobility. Qadri [32] provides a review of recent advances in IoT usage in healthcare.

In industrial applications, complex process monitoring can help in finding potential bottlenecks, identifying defects in manufactured products or alerting personnel to unexpected and anomalous situations, before they become noticeable [5]. In the chemical sector (and many others), they can help in ensuring that proper storage requirements for products are met all the way from production to delivery.

[1] https://www.arduino.cc/

[2] https://www.raspberrypi.com/

Agriculture requires monitoring of live stock as well as harvested crops. Having trackable information about each animal can help in disease prevention and isolation. Soil moisture sensors can be used to implement automatic irrigation systems, as well as to monitor all plants are actually getting enough water [5]. This data can be used to improve production, save money in cases of over-watering and serve as an indicator in environmental monitoring. Machine vision can also be used for crop monitoring [33]. Such systems can be used to detect plant diseases or pests. Similarly other deep–learning techniques can be used for soil quality and state estimation. Given sensor readings of soil moisture, humidity, temperature, rainfall and others, digital soil maps can be constructed, which then serve in identifying areas that need attention.

Outside of industrial applications, IoT can be utilized in cities and other municipalities. Such systems fall under the term Smart City. Possible use cases include smart outdoor lighting, environment monitoring [36], using sensors which measure variables such as air pollution and quality, temperatures, humidity or noise pollution. Urban areas can also benefit from smart parking systems for personal vehicles [36]. Such systems can monitor parking space occupancy and make that information available to both city officials for analysis as well as drivers. Smart irrigation techniques can also be helpful in managing parks and other greenery in the city. Freeing up resources required for manual upkeep.

On even smaller scale, smart devices can be used to implement Smart Home systems. Common components of these systems include smart lighting, paired with room-level presence detection for automatic lighting management. Air quality sensors, which can alert the occupants of high CO_2 content, cleaning automation in the form of robot vacuums and more. Smart Home Energy Management Systems provide a way to monitor and manage energy usage in the household [23], allowing for optimizing how energy is utilized. This can be particularly interesting if the house is equipped with solar panels, or other means of producing electricity. The system can then automatically decide when to use the generated energy and when to sell it to grid, based on the energy usage patterns of the household, current price (based on demand), weather forecast and other metrics.

IoT systems can also be used to provide support and enable the aging population to live more independently [5]. Sensor can monitor the conditions of the occupants as well as aid in daily tasks by automating activities, making sure required activities are done and reminding the occupant if they forget. Wearable and implantable device can provide health monitoring.

Though most of the functionality of such IoT systems is limited to only a single house or apartment, they can communicate with other Smart Homes and with the broader Smart City systems. For example, information about energy usage can be used to make more accurate predictions about energy grid load, since the systems have data on a more granular level than in regular energy grid monitoring systems.

2.2 IoT Architectures

A prevailing issue in IoT is lack of standardization [3]. Many approaches to creating IoT systems exist, some are more suited for specific applications, while others are more general. When conceptualizing a system, an overall architecture must be chosen. There is no unified system architecture, even for systems in the same field. Architectures are often described in terms of layers.

The simplest and among the most common one of these is the three–layer architecture [27, 3, 16]. It is made up of the following layers: perception layer, transmission layer,

application layer. The perception layer (sometimes also referred to as the device layer, sensory layer or recognition layer) contains the actual end devices, be it sensors, actuators or other supporting devices used for establishing communication between other end devices. This layer collects information about the physical world and sends it to the rest of the system for processing, analysis and storage. It also contains devices which are able to interact with the environment, based on both their internal intelligence and on commands from the upper layers of the system. The transmission layer (also referred to as the network layer) transports data from the perception layer to the higher layers. It can act as a bridge between IoT specific communication technologies and protocols used in the perception layer (e.g. ZigBee, LoRaWAN) and some standardized protocol, sent over a more conventional network (e.g. HTTP, MQTT). This allows for use of multiple different technologies in the lowest layer, thus allowing for greater choice when designing the system. The application layer provides the applications which can make use of the collected data, presents this data to the users and enables them to cause changes in the environment, either by directly controlling actuators, or by creating automations.

The four–layer architecture [21] extends the three–layer architecture with an additional layer between the network layer and the application layer. Thus this architecture is made up of: perception layer, transport (network) layer, service layer and application (interface) layer. The service layer manages services, which are then used by users or applications through the interface layer. Services manage data processing, storage and provide an interface to different parts of the underlying network. Because of the focus on services, this architecture is also known as a service–oriented architecture (SoA).

The three–layer architecture is sometimes also extended to include five layers, as is the case in [44]. The additional layers are the processing (middleware) and business layers. The layers are ordered as follows: perception, transport, processing, application and business. Data storage, analysis and processing is performed in the processing layer. The business layer is responsible for managing applications in the application layer. A comparison of all three mentioned models can be seen in Figure 2.1.

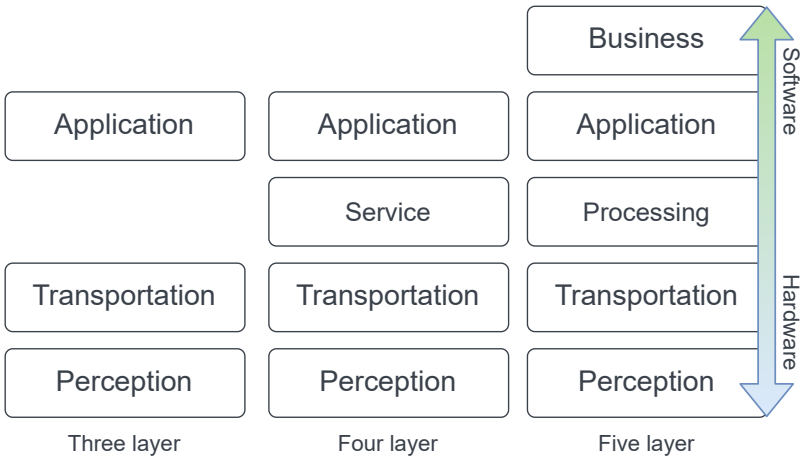


Figure 2.1: Comparison of three, four and five layer IoT architectures.

2.3 Communication Technologies and Protocols

Depending on context and requirements for a particular IoT system, different communication technologies and protocols are utilized. Choosing the correct combination is a crucial decision for any project, as it can greatly effect the projects feasibility, cost and usability. This section contains an overview of common technologies and protocols used in the IoT space.

According to a report by IoT Analytics [41], 80% of IoT devices utilized a wireless communications technology. With WiFi connected devices making up 31%, Bluetooth LE 25% and Cellular being about 21% of all wirelessly connected devices. A common way of grouping different wireless technologies is by their range.

Communications Technologies

The first wireless technology to be associated with IoT was **Radio Frequency Identification** (RFID). It uses tags attached to objects, which can be used to identify them by the rest of the system. RFID tags can be active or passive. Active tags transmit information on their own, allowing for greater range. Passive tags only operate when a reader tries to read information from them. They get powered by the radio waves sent by the reader. Active tags offer range up to 100 meters, while passive have a more limited maximum range of about 10 meters. But some tags are designed for very close operation, where their working range is sub one meter.

Zigbee is a wireless technology and protocol. It is based on the IEEE 802.15.4 specification. It is designed for use in small networks with low power devices. A common usage is for IoT systems, like Home Automation. The reported transmission distance is between 10 and 100 meters, depending on the environment. There are three types of devices in a Zigbee network: end devices, routers and the coordinator. There is always only one coordinator device, which serves the purpose of bridging the Zigbee network with the rest of the system. To compensate for the relatively low range, Zigbee is capable of creating a mesh network, where some devices act as routers and extend the reach of the network. Only devices, which are not battery powered can serve as routers in a practical way. Routers can be connected to the coordinator, or to other routers. End devices can be connected directly to the coordinator, or to routers. No device can connect to any end device. End devices are typically battery powered and spend most of the time in a low power sleep mode, thus not being capable of extending the Zigbee network.

Bluetooth Low Energy[3] (BLE) is used to create personal wireless networks. It is often used to connect wearable devices to the users smartphone, or other computer. Its maximum range is about 100 m, depending on the environment. BLE devices can be however connected in a mesh configuration, extending the networks reach. But such network configuration and cooperation has to be supported by all target devices. An example use might be for controlling smart light bulbs, where each bulb can communicate with the others, relaying commands from the users smartphone, depending on where in the house the user is currently located.

WiFi is an attractive wireless technology for use in IoT systems, primarily because of its popularity and widespread availability in both households, industrial installations and public spaces. IoT devices using WiFi can thus be installed without adding any specialized networking technology, utilizing the existing network. A potential downside of this

[3] https://www.bluetooth.com/learn-about-bluetooth/tech-overview/

approach could be the fact, that IoT devices connected to the main network of a household present an additional attack surface, while when specialized communication technology is used, devices can be limited in the types of messages they can transmit and the network manager (e.g. gateway, coordinator) can perform additional scanning and filtering, to prevent malicious devices from causing issues.

Microcontrollers from the ESP family offer a proprietary transmission technology based on WiFi called **ESP-WIFI-MESH**. Unlike WiFi, it creates a mesh network, utilizing the capability of the Microcontrollers to simultaneously connect to a WiFi network and broadcast their own. Unlike the Zigbee mesh network, it does not separate devices into end-devices and routers. But this functionality could be implemented using the provided development framework. Another difference is that there is no specific node in the network, which connects it to the outside network, in this case the standard WiFi router. There is always one such node, but it can be chosen dynamically, based on which device is closest or has the best signal. Diagrams of both a Zigbee and ESP-WIFI-MESH networks are shown in Figure 2.2.

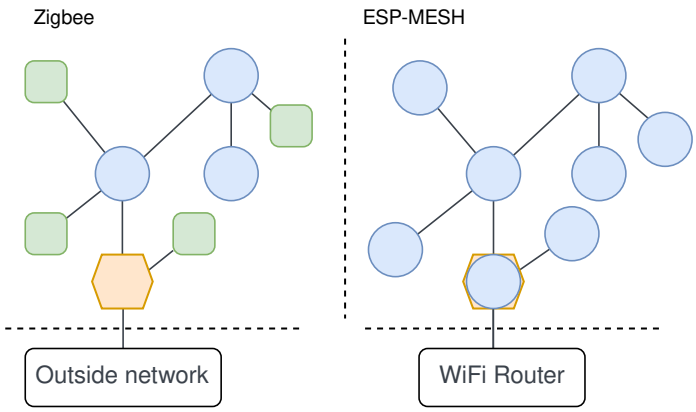


Figure 2.2: Diagrams of a Zigbee and an ESP-WIFI-MESH mesh networks.

LoRa is a wireless communication technology meant for wide area networks. It operates 868 MHz in Europe. The physical layer of the network utilizes a proprietary spread spectrum modulation. The communication protocol used in the network is called LoRaWAN. It defines how LoRa gateways communicate with end devices. Gateways then transmit data from the devices to a LoRa Network Server (LNS). The protocol specifies the shape of the data, but between the gateways and the server it can be transmitted over any internet protocol. The LNS manages the network, including the gateways and devices. It can also decode messages from the devices into a different format, which can then be forwarded to the rest of the system. Example uses of LoRa and LoRaWAN include smart street lighting in Smart City projects, where greater range than provided by short range technologies is required.

Communication Protocols

Some communications protocols are designed to only be used with a specific technology (e.g. ZigBee, LoRa), but others can be used together with many different underlying methods of data transfer. Some of the most prominent examples of such protocols which are readily used in IoT are discussed in this subsection.

A common protocol for IoT systems both on the perception layer and on higher layers is **MQTT**[4]. It is a lightweight protocol based on the publish–subscribe architecture. It was originally developed in 1999 by IBM, but it later became an open protocol. It runs over the TCP/IP stack, but a variant of this protocol meant for networks which do not support TCP/IP called MQTT-SN (MQTT for sensor networks) also exists. Under MQTT, the network is made up of end devices and a central broker. The broker acts as a central server, through which all communication flows. Each end device can subscribe to a number of topics. A topic is a string which consists of parts separated by `/` (forward slash). These parts can be used to create a logical hierarchy of topics. A subscription can then be made to either one concrete topic (e.g. `/home/bathroom/lights/1`) or to multiple topics, using wild-card patterns. The symbol `#` (hash) can take the place of 0 or more parts, but it can only be at the end of the topic specification. If a device subscribes to the topic `/home/bathroom/#`, it will receive messages published on `/home/bathroom/lights/1`, but also on topics like `/home/bathroom/temperature/1` and similar. The symbol `+` (plus) can take place of exactly one topic part. The protocol offers three levels of reliability guarantees, called Quality of Service (QoS)

- QoS 0 – no guarantee regarding the message transmission is given. The publisher does not wait for any confirmation from the broker about the message delivery. This means that the message might get lost, but it will be delivered at most once.
- QoS 1 – the publisher waits for a confirmation from the broker. If no confirmation comes in a given time window, it will send the message again. It is possible, that the original message got delayed on the way to the broker, but it was ultimately delivered. If this delay was long enough to cause the confirmation message to come after the publishers timeout window, a second message will be sent. For that reason, a message sent with this QoS might get delivered multiple times. This level of QoS is called at least once.
- QoS 2 – this is the strongest level of QoS, which guarantees that the message will be delivered exactly once. A handshake between the publisher and the broker is performed, which is used to establish a connection.

Hyper Text Transfer Protocol (HTTP) is a multipurpose protocol, used extensively in standard internet operations. Although not designed for use in IoT or similar environments, it is still a common protocol in these systems. It uses a request-response architecture, where the communication is always initiated by the client, and the server can only respond to requests. Requests are made for a particular endpoint, identified by a Universal Resource Locator (URL). Every request consists of the target URL and a *verb* (*method*). HTTP has several standard verbs, which signify the intent of the request. Among the most commonly used are: `GET`, `POST`, `DELETE` and `PUT`. The `GET` method is used when the client wishes to extract some data from the server, the `POST` method serves the opposite purpose – it signifies that the client is sending some data to the server. To request the deletion of some content, the `DELETE` method is used, and `PUT` is used to overwrite data for some existing entity. The exact effect of each method is dependent on the specific server and its interface specification.

Constrained Application Protocol[5] (CoAP) is a REST-based protocol, meant for use in constrained environments, such as low-power microcontrollers. It utilizes HTTP

[4] https://mqtt.org/
[5] https://datatracker.ietf.org/doc/html/rfc7252

verbs, but introduces a new verb `OBSERVE`. Which serves a similar purpose as subscribing in MQTT. Unlike in MQTT, CoAP uses Unique Resource Identifiers to identify resources, on which actions can be performed using HTTP verbs. It uses `UDP` to send messages. Therefore it needs to implement its own reliability mechanisms.

Chapter 3

Visual Programming Languages

End-User Development (EUD) describes a process, where users of a particular tool, product or system can alter the systems behavior to make it perfectly suited for their needs [22]. Users have a deep understanding of their particular domain, as well as the requirements for the given application, but they often lack experience and skill in software development. Thus EUD systems have to be tailored for inexperienced users, and offer an intuitive way for them to achieve their tasks, while also being flexible enough as to not limit the users ability to construct complex programs as their experience with the tool and programming in general grows.

Systems and applications which support EUD empower users to modify the application to their exact needs, making them more productive and making the tool more useful for them. This property also benefits the developers of the original system, since they do not need to spend time and resources on adding features, which only a small subset of users will use, instead, they can focus on improving the core of the system and on offering a robust way of modifying the system from the outside by end-users.

A simple example of a way for users to modify a tool for their needs could be recording a macro, for example in a text, photo or video editor. Macros are recorded sequences of actions performed by the user. After being created, macros can be replayed multiple times, always performing the recorded actions. This can greatly save time, if the user does repetitive tasks, which involve multiple steps. A macro allows them to record it once and then apply it the next time they need to perform the operations. A more capable example of EUD is the ability to create automations, functions and manipulate the worksheet via the Visual Basic programming language in Microsoft Excel. It allows for far greater customizability than simple macros, but at a cost of complexity and approachability for inexperienced users. That is because most end-user, even if they are professionals in their field, do not have software development experience, so customizing their tools using textual programming languages comes with not only the added logical complexity, but also with the need for them to learn the syntax and semantics of the given language. One way to make EUD more approachable is by offering a more intuitive way to create programs, while not sacrificing the capabilities of the method. A popular way of offering this capability is by utilizing a visual programming language [6].

Visual programming languages (VPLs) allow users to construct programs by way of manipulating visual elements, as opposed to manipulating text, which is how most programming languages are used. The graphical approach provides a more intuitive way to compose programs [34], by representing the program in 2D space, connecting arrows to represent the flow of data, or nesting elements in predefined ways. VPLs allow users to focus

on implementing the logic of the program, without the need to learn or focus on syntax, by abstracting it behind the visual metaphor [18]. VPLs have a wide array of use cases, for example in education, game development, 3D modeling and video editing, automation, or data analysis [34]. They are also commonly used in IoT systems as a way for users to setup the system to their specific requirements, by connecting different devices together and creating automations.

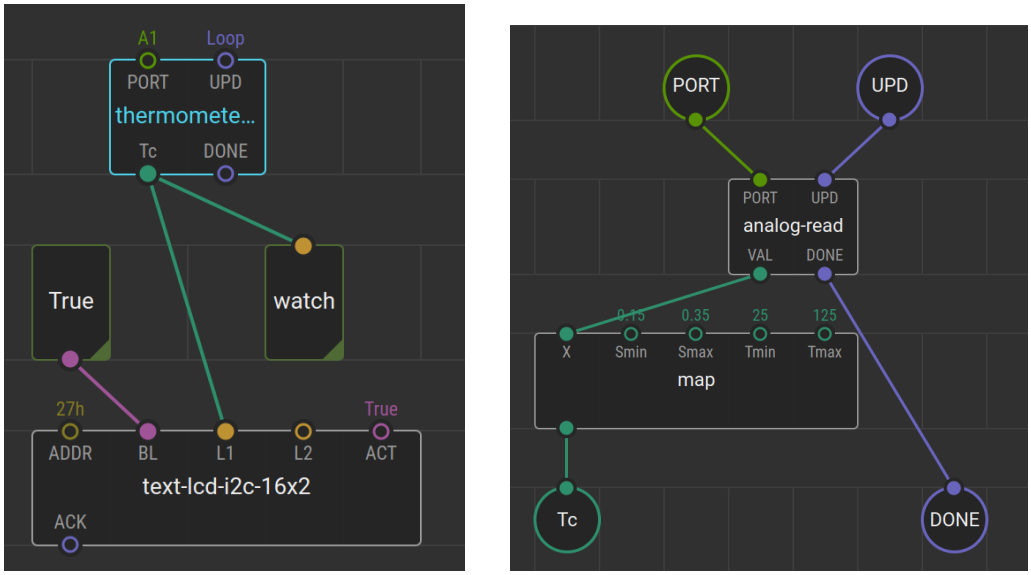
A review of VPL usage in pre-higher education and very early higher education [28] shows that VPLs can provide a good platform for introductory courses to computer science. But they might not be well suited as a teaching tool for more complex topics. When used as a first programming language, VPLs have the benefits of not requiring precise knowledge of the syntax of the particular language, instead allowing learners to focus on the more basic principles of creating programs and developing better problem solving skills. Several of the reviewed studies showed, that learners performed better in regular textual programming languages after an introductory course using a simple VPL. Although the choice of a first programming language can have an effect on learners performance, it is still more important that the course is well structured, organized and presented.

In IoT, VPLs can be used for both creating automations on the application level, and for creating firmware for sensors or actuators on the perception layer [34]. An example of the former use case is the Node-RED platform. It allows users to process messages from devices and construct responses by building a dataflow diagram from available blocks (nodes). Messages then pass through the nodes, where they can be modified, discarded, or they can trigger creation of more messages. For firmware creation, one such VPL is Xod[1]. Users can create firmware for Arduino[2] or ESP[3] based microcontroller. Programs are created by connecting blocks in a dataflow style. Blocks can for example represent variables, boolean operations, or an internal or external peripheral like the systems clock, counter, or input/output pins. New blocks can be created by the user, encapsulating multiple other blocks into a single one, with a specified interface. An example program can be seen in Figure 3.1. The program (in Subfigure 3.1a) receives a value from the thermometer block and displays it both in the debug console (via the `watch` block) as well as on an LCD display, by sending the value from the thermometer block to the correct port on the LCD block. On the right (in Subfigure 3.1b) we can see the implementation of the thermometer block. It reads an analog value from the given pin (port) and maps the value from one range (given by the `Smin` and `Smax` values) to another (between `Tmin` and `Tmax`). The program runs in a continuous loop.

Myers [26] classifies VPLs based on the form in which the program is represented. They present multiple categories, which all represent the program as a graph of connected nodes. These categories are: flowcharts, flowchart derivatives, data flow graphs, Petri nets, directed graphs, graph derivatives. Other presented categories are: matrices, jigsaw puzzle pieces, forms, iconic sentences. They also provide categories for non-visual programming languages, which are not interesting to the discussed topic.

Shu [40] categorises VPLs into three groups: diagrammatic, iconic and forms-oriented. They describe diagrammatic languages as a useful extension of already familiar tools (diagrams), which are used in software development in the planning and documentation stages. But instead of simply describing an existing system, diagrammatic languages allow for the visual description of the program to be the program itself. Iconic languages use simplified

[1] https://xod.io/
[2] https://www.arduino.cc/
[3] https://www.espressif.com/en/products/socs/esp32



(a) The main firmware body.

(b) Thermometer block implementation.

Figure 3.1: A firmware created in the VPL Xod, which reads the current temperature from a temperature sensor and displays it on an LCD display.

icons to represent actions, operations or objects. These are then arranged in some way to construct a series of instructions. The exact method of arranging the icons varies. An example language described by Shu uses icons to represent basic programming primitives. These operations are selected from a predefined set and placed on a two dimensional grid. Icons are then connected by colored pathways, which denote flow control. According to Shu, form-based languages take advantage of users familiarity with form interfaces. They allow for a great degree of flexibility in terms of the language's target domain.

According to Burnett [8] VPLs have fewer syntactic restrictions, which makes them suitable for exploring different mechanisms for program specification. They also state that the goal of VPLs is not to eliminate text from program specification, as most VPLs use text, but it is not displayed in a simple linear fashion. The purpose of VPLs is to make programming more accessible to a given audience, improve correctness and speed with which users perform the programming tasks. VPLs achieve this by utilizing the following strategies [8]:

- Concreteness – expressing an aspect of a program using existing instances. For example, modifying some attribute of an object alters its shape, making it no longer fit, where it was previously positioned.
- Directness – allowing users to directly modify aspects of the program, like setting some attribute of a given object directly on its representation and seeing not only the change of the given value, but also any derived values.
- Explicitness – displaying some semantic information explicitly, instead of implicitly and forcing the programmer to infer it. A good example is showing how the data flows in a program as edges or arrows in a diagram based language.
- Immediate Visual Feedback – how fast and when are the effects of changes to the program visible.

Burnett [8] categorises VPLs into the following categories: imperative visual programming by demonstration, forms/spreadsheet based, dataflow and rule-based. Kuhail et al. [20] present a taxonomy based on two previous taxonomies, one by Myers [26] and the other by Burnett and Baker [9]. They present the following categories:

- Block – programs are constructed from blocks, which the user selects from a list of the available blocks. Blocks are connected together, similar to connecting puzzle pieces. By changing the shape of the connection point of some blocks, languages of this type can help prevent syntax errors, since it can limit which blocks can connect to or nest inside other blocks. For example, they can limit which blocks can be used as a predicate in if statements, so that only blocks which produce a boolean value are accepted.
- Diagram – the user places objects (blocks, sometimes also called nodes) in the development area and connect them together with lines or arrows. These connections signify the flow of data between the various objects. For this reason, they are sometimes called data flow languages.
- Form – users create programs by configuring a form – by selecting operations and attributes from a drop down, or by dragging them into the form from a predefined list. They sometimes allow for textual input in various areas of the form.
- Icon – icons (graphical elements) are used to construct visual sentences, spatial sequences of icons, which can represent actions or objects. Such languages can be used, for example, to create trigger–action programs, meant for home automation.

3.1 Examples of VPLs

This section describes examples of existing VPLs grouped by their type as categorized by Kuhail et al. [20].

Block VPLs

Scratch[4] is an example of a block–based VPL aimed at children and young students, for the purpose of learning basics of programming. Its main usage is for creating interactive visual games and experiences. Learners create scenes by using sprites, which can be manipulated through the VPL. A simple program can be seen in Figure 3.2. It moves a sprite forward if the up arrow is pressed or rotates it by a variable number of degrees if the right or left arrows are pressed. When a right arrow press is detected, the `rotation` variable is increased by ten. Scratch offers a web–based development environment, which includes:

- list of all the available blocks, sorted into categories by their function,
- canvas for placing blocks
- and a scene view, where the users can display and manipulate sprites.

Blocks can be connected together in a sequence, or nested in other blocks. Each block denotes the possible ways it can be connected to other blocks by its shape. If a block can be

[4] https://scratch.mit.edu/

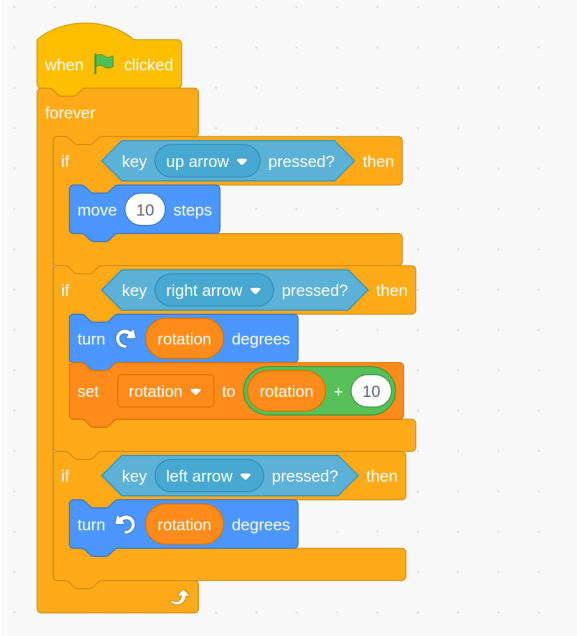


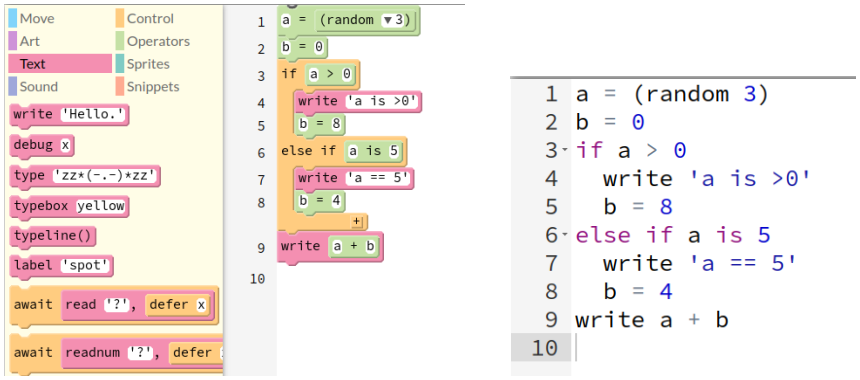
Figure 3.2: Scratch program for movement of a sprite.

placed only as the last statement in a sequence, it will not have a notch on the bottom, which is required to connect more blocks to it from that side. In a similar manner, conditions for use in if statements or loops have a specific shape, which does not fit anywhere else. This provides an intuitive way of understanding the syntax of a language, without it needing to be extensively taught, allowing the students to focus on the semantics of their programs. Scratch allows users to define their own blocks as encapsulation of other existing blocks. In this way, the feature functions similarly to creating functions in a conventional language.

Scratch is built on the Blockly[5] project, which is a versatile library, that can be used to create customized VPL applications. It provides total control over the blocks which are available in the development environment, as well as how those blocks get executed. This makes it very useful as a base for building educational tools and tools for EUD, including for IoT systems. The library offers both the core block definitions and functionality as well as the visual editor. When additional block functionality is needed, the Blockly library can be extended with plugins. They can be installed as `npm` packages, making them easy to share and to create custom plugins if none of the existing ones satisfy the project needs. Plugins can be made for both the visual interface as well as the Blockly core, offering additional block customizability.

A similar block–based VPL to Blockly is the Droplet[6] project, which is used as the editor in the PencilCode[7] platform. The main purpose is to provide educational experiences for young learners. It directly supports the bi-directional translation between code blocks and text. The user can switch between the block and text view of the program at any point while developing. Changes to either representation are immediately translated into the other. This way, learners can transition from purely visual style programming, while staying in a familiar language. They can also see how the textual code they write translates to the

[5] https://developers.google.com/blockly
[6] https://github.com/PencilCode/droplet
[7] https://pencilcode.net/



(a) The block representation. (b) The text representation.

Figure 3.3: A program in the PencilCode VPL. The program is represented both by a visual block–based code as well as a textual representation.

block form, which they already know. An example program can be seen in Figure 3.3, with the block representation visible in Subfigure 3.3a and the equivalent textual representation shown in Subfigure 3.3b. The program sets the variable `a` to a random value between zero and two and sets another variable `b` to zero. Then it checks if `a` is bigger than zero and if it is, it write the string `a is >0` to the output and sets `b` to eight. Otherwise it checks if `a` is equal to five, which will never happen, so the nested code blocks will never be executed. At the end it writes the sum of `a` and `b` to the output.

Diagram VPLs

Node-RED[8] is a diagram–based VPL and runtime, used mainly for home and industrial automation. The program is constructed by placing and connecting nodes together with wires, or virtual connections on a 2D surface. The execution model of Node-RED is based on message passing, where each node can have zero or one input on which it can receive messages, and zero or more outputs. Messages get into the system from nodes, which are connected to some outside source, like an MQTT subscriber node, a timer or an HTTP endpoint. Messages then flow through the program, based on the connections between nodes. The entire runtime is built on the Node.js[9] platform and it is written in JavaScript. Custom nodes can also be written in JavaScript and installed as plugins. Nodes can be grouped into flows and subflows for better code organization and reuse.

Node-RED contains a selection of built-in nodes, which are grouped by their functionality into the following groups: common, function, network, sequence, parser and storage. Common nodes include a very useful node called `debug`, which simply prints the message it receives into the debug console. Other nodes in this category include the `inject` node, which can be used to manually inject messages into the system for testing, virtual link nodes and more. Function nodes include the `function` node, which can be used to write arbitrary JavaScript code, adding great customizability level to the system, at the expense of forcing users to learn a different programming language other than the main flow–based VPL. This group also includes built-in nodes, which can be used to alter contents of a message, route messages to different connections, delay message and more, without the need

[8] https://nodered.org/
[9] https://nodejs.org/en

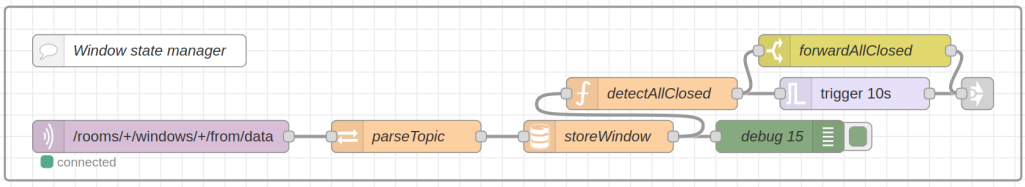


Figure 3.4: Example of a Node-RED program, which detects if one or more windows in a room is open for more than a specified time, so that heating of that room can be paused until they are closed again.

to use a different programming language. The Network group includes nodes for receiving and sending messages over the following protocols: MQTT, HTTP, Websocket, TCP and UDP. Nodes in the sequence group operate on multiple consecutive messages, providing the ability to sort, join, split or batch them. Parser nodes can parse various common textual formats such as JSON and HTML and storage nodes offer an API for working with system files.

Figure 3.4 displays a program which receives messages from a group of MQTT topics, on which the state of multiple window sensors is sent. The message first passes through a function node, which extracts information from the topic string and attaches it as an attribute to the message, the window state then gets stored in a flow level cache, before being processed by the rest of the program. This could be used in a larger system to detect if one or more windows in a room are open for longer then some specified amount of time, so that the room heating could be paused.

VPLs are commonly used in multimedia applications, such as video editing programs or game engines. A prominent example is the Unreal Engine (EU) Blueprints[10] VPL, which can be used to implement functionality for objects in UE games. UE Blueprints are offered alongside a C++ API, with the intention that artists and other non-programmer project members can use UE Blueprints to implement functionality, which is not computationally expensive, while more intensive features can be implemented in C++ by the team's programmers. UE Blueprints offer a more user-friendly API, when compared to the standard C++ API, but at the expense of some performance loss. It is however possible to implement custom nodes in C++. So expensive operations can be moved to C++, while still leveraging the flexibility of the UE Blueprints VPL for the rest of the system.

Nodes are positioned on a 2D surface and are connected together with wires. Each node can have multiple inputs and outputs, with data flowing between nodes and being modified by the individual nodes. Nodes include modifiable attributes directly as part of their visual representation, which is different to the previously discussed Node-RED nodes, where attributes are only visible and modifiable in a pop-up window, which can be opened by clicking on a node.

UE Blueprints are not executed directly, they are compiled into bytecode specific to the UE runtime. This bytecode is the executed by a virtual machine inside the UE runtime. It is possible to have the Blueprints compiled into C++-like code, but this is meant for debugging purposes only.

[10] https://dev.epicgames.com/documentation/en-us/unreal-engine/blueprints-visual-scripting-in-unreal-engine

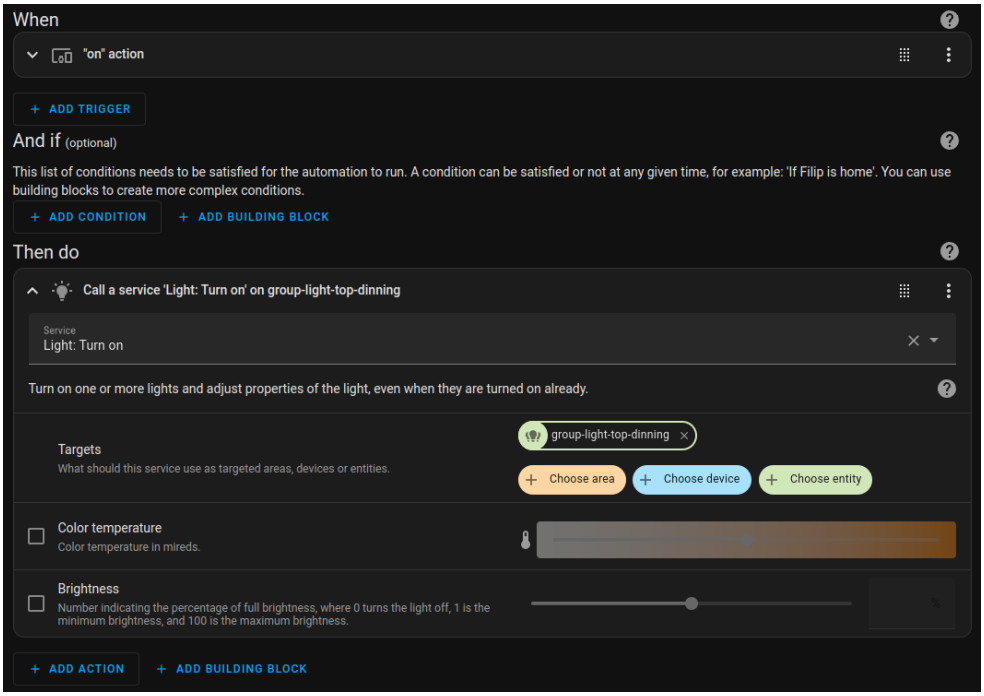


Figure 3.5: HomeAssistant automation editor, showing an automation which upon receiving a message from a remote control turns on a group of smart light bulbs.

Form VPLs

The popular home automation system HomeAssitant[11] offers a built-in GUI automation editor. The automations are specified using a form–based VPL, in the form of trigger–action programs. Where each program definition begins with some trigger, which will start the execution of the given program. Each program can have multiple triggers, which can be as simple as receiving a message from some device, or more complex, such as a periodic trigger, or a location based condition. After the trigger definition, each program can have zero or more conditions. This way, multiple programs can have the same trigger, but only run at certain times of day or when the whole system is in some specific state. The last part of each program is the body – actions. A program can have one or more actions, which can either be a simple modification of some entities state (e.g. by sending a message to a device), or they can be more complex operations, such as loops, if-else blocks, delays and more. All programs follow this same limited formula, making them easily accessible for less experienced users. They can, however, be quite limiting for more experienced users, as creating more complex automations can be cumbersome. An automation which reacts to a message from a remote control and turns on a group of smart light bulbs can be seen in Figure 3.5.

HomeAssitant automations don't offer any form of interactive debugging, but it is possible to view a log of each automation invocation and which steps were performed and in what order. These logs are called a trace. This is useful in debugging more complex automations, which are not simply a sequential list of actions, but include conditionals or loops. Each invocation's log is stored, so it is possible to go back in the history of invoca-

[11] https://www.home-assistant.io/

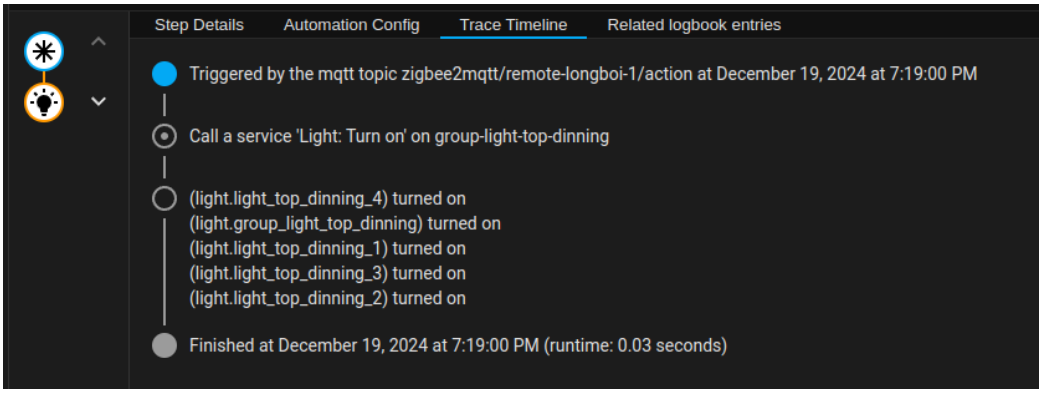


Figure 3.6: Trace log of an invocation of a HomeAssistant automation, which reacts to a message from a remote control and turns on a group of smart light bulbs.

tions and diagnose the issue even if the last invocation didn't have any issues. Example of a trace log can be seen in Figure 3.6.

Another representative of a form–based VPL is iOS Shortcuts. It is a built-in application, which allows users to create shortcuts and automations. Programs are made from actions, which represent both basic programming primitives such as if statements and loops, but also more complicated actions which modify the state of device, either in software, or by manipulating the phone's hardware itself (e.g. turning on the LED torch, turning on or off Bluetooth), or shortcuts offered by other applications which are installed on the device. Each application can register actions, which will be available in the list of possible actions usable in shortcuts[12]. This way, internal functionality of the application can be exposed for use in EUD and user customization. Actions can have customizable properties, which can either be selected from a predefined list or entered manually, depending on the type of property. For example, the action which send a text message to someone has two properties which need to be set: the message body and the list of recipients. The former is a text field, which can be set to any text value, while the latter opens the contacts list and allows the user to select from it. Actions can be reordered and nested by holding and dragging them. Users can also create variables, which can then later be used in properties. Shortcuts are then triggered by clicking on their icon, which can be either placed on the home screen, or found in the Shortcuts application. Automations run automatically, based on some specified trigger event. The body of an automation is the same as a shortcut. An example program can be seen in Figure 3.7.

3.2 VPL Execution

The way VPL programs are executed differs based on their purpose and the system they are a part of. Some VPLs have their own specific runtime, whose sole purpose is to execute the program represented by the visual model (e.g. Node-RED). Others are integrated into an existing runtime, extending it in some way (e.g. HomeAssitant automations), while others still are capable of being translated into textual languages, which can then be executed in any way available for that language (e.g. Blockly, Unreal Engine Blueprints).

[12] https://developer.apple.com/documentation/appintents/app-shortcuts

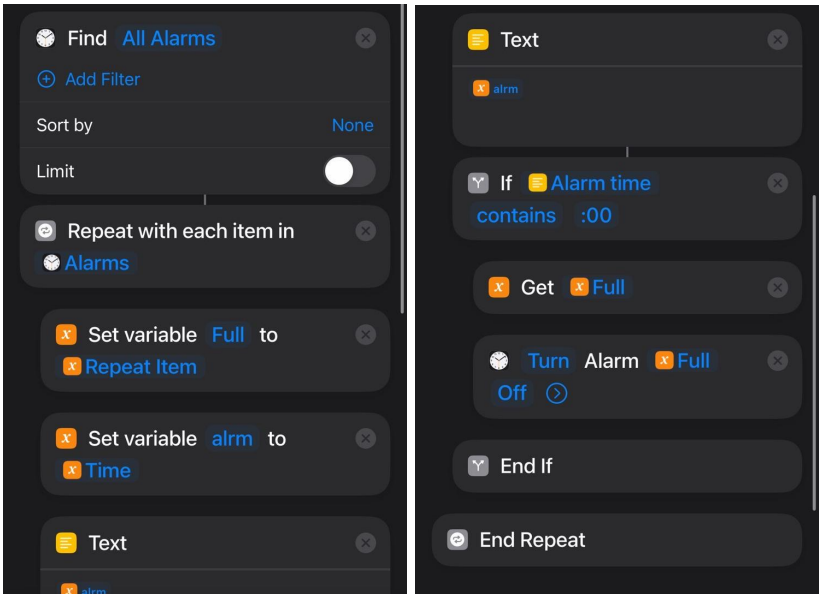


Figure 3.7: A program in the iOS Shortucts VPL. It first creates a list of all alarms which exist on the device, then for each alarm it checks if the time is set to an exact hour (minutes are equal to 0). If so, it disables the alaram. A second program could then be used to enable these alarms. Together the programs could be used to toggle a group of alarms based on user input or based on some change in the device's state, like the current location.

In Node-RED, the program is is split into groups of *nodes* called *flows*. Flows can be used to keep related functionality together and separated from other parts of the program. Virtual connections between flows can be created using a special pair of nodes, a virtual input and output, which when paired together act as if a regular wire connection was made between them.

The Node-RED runtime is built using the Node.js JavaScript runtime. The execution of the program is entirely driven by the passage of messages between nodes. Messages enter the program as a result of an outside event, like an incoming message on an MQTT topic, a request made to an exposed HTTP endpoint, or as a result of a timer or a similar event. Regardless of the reason for a new message creation, all new messages are introduced to the system as output from some node. This way, the system doesn't see any difference between a new message triggered by an outside source and any other message, which was processed and output by some other node in the graph.

When a node outputs a message, the runtime adds a destination node to the message object, based on the connections between the node's output and any other node's input port. If there are multiple nodes connected to the output, the message gets multiplied and each node is delivered its own copy of the message. If the payload of the message is an object, it also gets copied, otherwise all the messages would point to the same underlying payload object and could modify each others messages, leading to unwanted behavior.

Nodes are represented as a function, which gets run each time a new node of that type is added to the program. Inside the function the node can setup its local variables and it can also register callbacks to certain events, which can be triggered by the runtime. This makes it easy for developers to add new nodes and create plugins for the Node-RED ecosystem.

The runtime doesn't need to know anything specific about the nodes internal working, as long as it can respond to events. Nodes can use any library available in the Node.js runtime, without interfering with the Node-RED runtime in any way.

The most important event is the `input` event. This is called every time a message is delivered to the node. Inside this callback the message can be processed in some way, like adding additional attributes, replacing the payload or it can be rewritten by a completely new message object. If the node wishes to output a message, it can either return the message object from the `input` callback, or it can call the `send()` function, which is passed to the callback as a parameter. This will place the message in the nodes output list and start the routing procedure. If it does neither, no message is output.

The Blockly VPL differs from Node-RED in way in which the program specified by the visual elements gets executed. Instead of a runtime which would execute the blocks directly, Blockly generates textual code from the visual program specification, which can then be executed. The Blockly library offers five built-in code generators, but it is possible to define custom code generators for any language. The built-in generators are: JavaScript ES5, Python 3, Lua 5.1, Dart 2 and PHP 7. Each generator can convert a given workspace to the textual representation in the given language. To do so, the generator includes *block-code* generators for each block available in the Blockly toolbox. This generator is then responsible for converting the given block to the target language. The result of block-code generation is a string, so workspace code generation is simply a concatenation of the individual strings. Code execution differs based on the target language and it is completely unrelated to the Blockly library. Custom blocks can either be defined using a JSON object, or by using the javascript builder API. These two approaches can even be combined, which makes it possible to define most of the block using JSON and add some dynamic elements using javascript code. Because Blocks are not executed in any specialized runtime, but simply translated into a textual programming language, some projects using the Blockly library also offer the inverse functionality – converting the textual language into blocks and showing the result immediately in the graphical editor. One such example is the PXT[13] project, which is aimed at creating programming experiences mainly focused on programming education. It also implements an ARM Thumb machine code generator.

HomeAssitant automations are translated from the form–based visual editor into a YAML representation. It is possible to choose to edit this representation directly, instead of using the visual editor. The triggers for each automation are evaluated on system start (or automation creation) and they are registered to the associated event. When this event is received by HomeAssitant, all registered automations will be executed.

[13] https://github.com/Microsoft/pxt

Chapter 4

Debuggers and Simulators

With the advent of higher level languages came the expectation, that bugs in code would become rare, easier to find and fix [17]. This was, however, not the case. The increasing quantity of code being written thus required better tools for finding defects, understanding them and fixing them.

Debugging is the act of eliminating errors or malfunctions of computer programs [24]. It does not require any specialized tools and can be performed in a wide variety of ways. In large and complicated systems, just localizing a fault can be very challenging, especially if not done in a systematic manner [25, 7]. A 2017 survey [30] of 303 professional software developers found that 51% of the participants have not received any formal debugging education. Of those who have, most report receiving it only once, most often as part of their higher education studies. Several studies [25, 2, 13] suggest, that debugging is a distinct skill from general programming ability, and should therefor be given more of a focus in the education of software developers.

Zeller [45] suggests, that debugging can be approached using the scientific method. As such, debugging could be described in the following four steps: (1) observing a failure; (2) inventing a hypothesis; (3) using the hypothesis to make predictions; (4) testing the hypothesis by experiments and further observations. By refining the hypothesis and repeating step 4, it will eventually become focused enough, that further refinement is no longer possible. Such hypothesis which explains the earlier observations and can predict future observations (e.g., the failure no longer being present, after fixing the proposed issue) would – in the context of debugging – be called a diagnosis. According to the study performed by Perscheid [30], professional developers can often use a very similar approach, without being consciously aware of the steps. O'Dell [29] concurs, that the process of debugging employed by many software developers can be described using the steps of the scientific method. But they write, that often very little attention is paid to the formation of a hypothesis, leading to wasted effort. Thus emphasizing the need to treat debugging as a systematic process, rather than an art, based solely on intuition.

Several tools and techniques exist to help with the task of debugging. Adragna [1] lists the following types of tools and techniques:

- **Static analysis provided by the compiler** – many compilers offer static analysis as a part of the compilation process. This is possible since static analysis is performed without running the analyzed code. These features attempt to point out possible defects or suspect operations. Detected issues can range from basic seman-

tic problems (e.g., type mismatch) to more complex issues, like memory safety, or deadlock detection [46].

- **Reading the correct documentation** – Adragna points out, that an important part of debugging is having easy access to documentation for all the tools and libraries used.

- **`printf` debugging** – sometimes also called `cout` debugging, takes its name from the statements used to display textual output in languages like `C` and `C++`. Print statements are inserted in points of interest within the program, with the purpose of creating a log of the program execution. They can be used to show the values of variables at certain places in the program or signal that a point in the program has been reached and executed. These statements are removed from the program after the debugging session is over.

- **Logging** – contrary to the temporary nature of `printf` debugging, logging statements are meant to be kept in the program even during production usage. Logging output is saved into a log file, which can later be used to assist in diagnosing problems which happened in the past. Individual logs usually carry a message describing a particular event or unexpected behavior that occurred, alongside a timestamp and some additional information, like the name of the function where the log originated, or the log's severity.

- **Explaining the problem** – trying to explain the problem out loud, either to someone else, to an inanimate object (so called "rubber duck debugging") or to no one in particular, can be beneficial in the debugging process [43].

- **Reading through the code** – trying to fully understand the relevant parts of the code without executing it can help build a better mental model of the program, which in turn can help with formulating hypotheses.

- **Debuggers** – interactive debuggers allow for exploring and interacting with the program in its dynamic form, as opposed to the static form presented by the programs source code.

Developers present in the survey performed by Perscheid [30] also used other tools to aid in the process of debugging, like text search tools, code navigation tools and version control related tools. While non of these tools are strictly related to the process of debugging, they allow for greater speed in formulating new hypotheses and in gathering necessary context, such as examining the latest modifications in a suspect part of the system.

According to Rosenberg [35], interactive debuggers can be classified based on the following criteria:

- **Source–level (symbolic) or machine–level** – Source–level debuggers map the instructions which are being executed on the CPU back to the original source code. Thus making it possible to treat the statements in the higher–level language as a single execution unit, even though it can be comprised of multiple machine instructions. Machine–level debuggers do not make this mapping (either by design or because they are unable to).

- **Standalone or integrated** – Standalone debuggers are used outside of the users standard development environment. This has the advantage of not being tied to a single development environment. On the other hand, it forces the user to switch between multiple tools during the debugging session. An example of an integrated debugger can be seen in Figure 4.1.
- **Interpreted or compiled** – Compiled languages require the debugger to be able to understand the mapping of the compiled machine instructions to the original source code. They also need to rely solely on the debugging interfaces offered by the operating system. While languages which run in an interpreter provide tailored interfaces for execution control and inspecting (and potentially modifying) the program state.

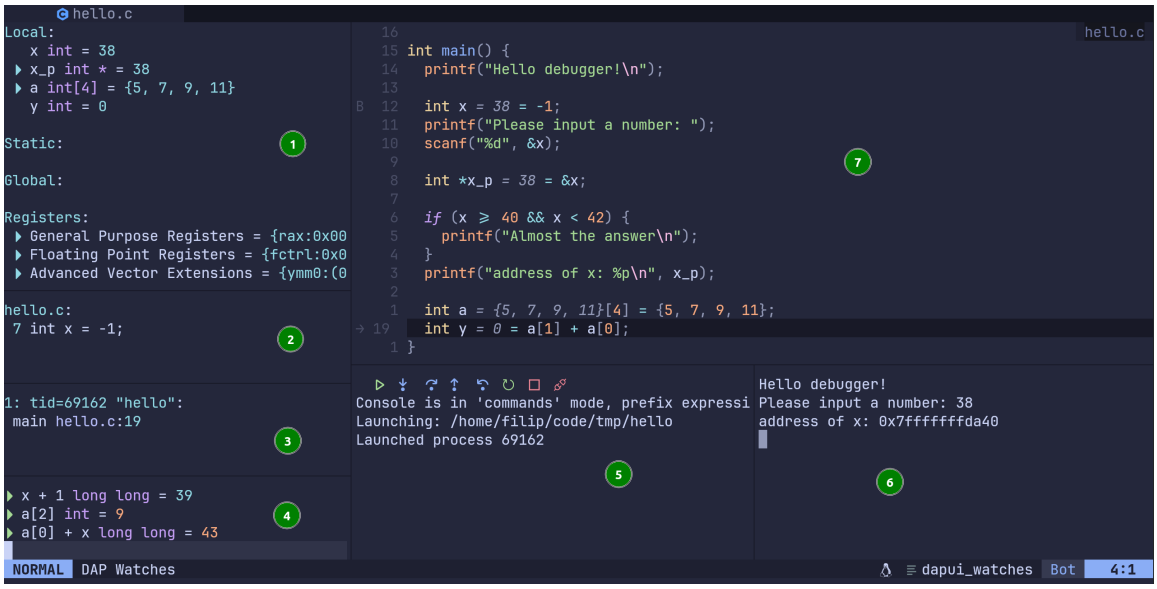


Figure 4.1: Screenshot of an integrated source–level debugger. The debugger presents a user interface inside the development environment. It is split into multiple parts: (1) view of variables and their scope, together with the current state of registers, (2) list of set breakpoints, (3) function call stack, (4) the watches window, (5) buttons for execution control and a debugger console, for issuing commands, which are not made available through the UI, (6) output and input of the program, (7) view of the source code, annotated with breakpoint positions, variable values and an indicator of which line will be executed next (line 19).

One of the main capabilities of interactive debuggers is control of the execution of the program. This capability allows them to pause the execution and then step through the statements of the program. Pausing can usually be done either manually, by sending a command while the program is running and the debugger is attached, or by setting a breakpoint somewhere in the program. A breakpoint is a marker placed on a specific statement or instruction in the program, that tell the debugger to stop execution before that point in the program is executed [35]. If the program is a compiled binary executable, breakpoints typically result in an interrupt instruction being placed where the user whishes to stop the execution [45]. This makes the execution stop and transfers control to the debugger. If the program is running in an interpreter, no such mechanism is needed [37].

The interpreter exposes many powerful debugging features to debuggers or already comes with a builtin debugger. Such tools can benefit from the fact, that the entire program state is already stored in the interpreter and the source code usually doesn't differ much from the expressions being executed, unlike compiled languages.

Besides simple breakpoints, debuggers offer additional ways to trigger a pause in program execution, like the *data breakpoint* or *conditional breakpoint* [45]. Data breakpoints watch a specified variable and trigger when they detect a change in its value. Data breakpoints can be expensive in terms of execution speed, since the debugger has to check for modifications of every watched variable after every instruction. Many CPUs offer *hardware data breakpoints*, allowing for a limited number of watched variables, which do not require constant monitoring by the debugger. Conditional breakpoints can specify a condition, which if met, will trigger the breakpoint.

When the execution of the program is paused, there are usually several options available for stepping the program: *step-into*, *step-over*, *step-out* [1]. The *step-into* command will try to step inside the body the function which is to be executed, stopping on its first instruction. But the *step-over* command will execute the function in its entirety stopping on the next statement bellow the function call.

4.1 VPL Debuggers

Visual programming languages often try to make programming accessible to novice users [11], for example by allowing for creation of the program by composition of blocks, which makes syntactical errors impossible. However, users can still implement programs which behave incorrectly, because of logical errors. Debuggers can help users track down and understand, why their programs are not performing as they expected. This section discusses examples of available debuggers for various VPLs grouped by the language type. All VPLs mentioned in this section are described in Chapter 3. No debuggers for form and icon-based VPLs were found.

Block VPLs

The Scratch visual programming language does not provide a tool for debugging ones code in the standard editor. Deiner [10] introduces their debugger, designed to work with the Scratch programming environment. Their debugger offers execution control, reverse execution and interrogative debugging. Execution control can be achieved either by manually stopping the Scratch VM using a button in the user interface or by setting a breakpoint on a block. After pausing the execution, the program can be stepped. Only the `step-over` command out of the typical commands seen in source-level debuggers is available. It informs the Scratch VM to execute the next block. It is not possible to skip stepping into child blocks of the currently paused block. To enable reverse execution of the program, they instrument the Scratch VM with a custom tracer, which records information about each step of the execution. This trace is then used to implement the `step-back` command, which rewinds the execution by one step. This is possible since the debugger can record all necessary information needed to restore the programs state at any given point in its execution. Instead of the `step-back` command, a timeline slider can also be used to navigate to any historical step of the execution. When viewing a past state of execution, the `step-over` command does not execute the block, it simply moves forward in the execution trace. But resuming execution using the resume button in the user interface when viewing

a past state discards all following trace information and starts executing the block from the current point.

Storing the execution trace also allows the debugger to provide an interrogative debugging system. Allowing the user to select from a list of questions, generated based on all the attributes and variables in the program. Questions take the form of either "Why did..." or "Why didn't..." inspired by the paper Whyline [19]. For every question, the system also generates an answer, in a textual and/or visual format. Example of a question and its answer is seen in Figure 4.2.

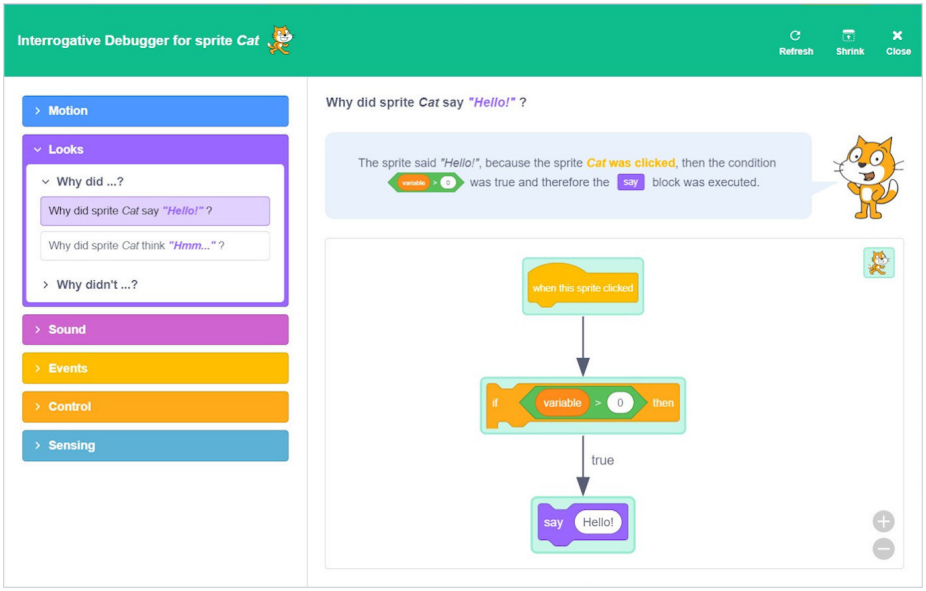


Figure 4.2: Interrogation of the Scratch debugger. Taken from [10]. The left part of the screen shows a list of possible questions, grouped by type. The answer to the selected question is displayed on the right part of the screen. The answer is given in the form of text, which explains why this particular event happened, as well as an excerpt from the program, which was responsible for the event.

Like Srcatch, the Blockly visual programming language does not provide a debugger by default. Savidis [38] described a novel block-level debugger for the Blockly visual programming language. Their debugger has the ability to set breakpoints on any individual block. This can be viewed as an analogous operation to inserting a breakpoint on a particular line of source code in a traditional source-level debugger. Block-level breakpoints technically offer a more fine-grained control of stopping the execution flow, since blocks can be nested in other blocks, forming what would be a single line in a textual programming language. When a breakpoint is hit and the exectuion is paused, a user can choose to step the program. Typical *step-in*, *step-out* and *step-over* options are implemented. The *step-in* command will try to step to the next child block, while *step-over* command will continue executing the next sibling block.

It is also possible to inspect the state of the program variables, through the *watches* window, which displays the current state of all previously used variables, since every variable in Blockly is global. The debugger is implemented by instrumenting the code of every block in the user program with a wait loop, which can be used by the execution control logic to either stop or step the program.

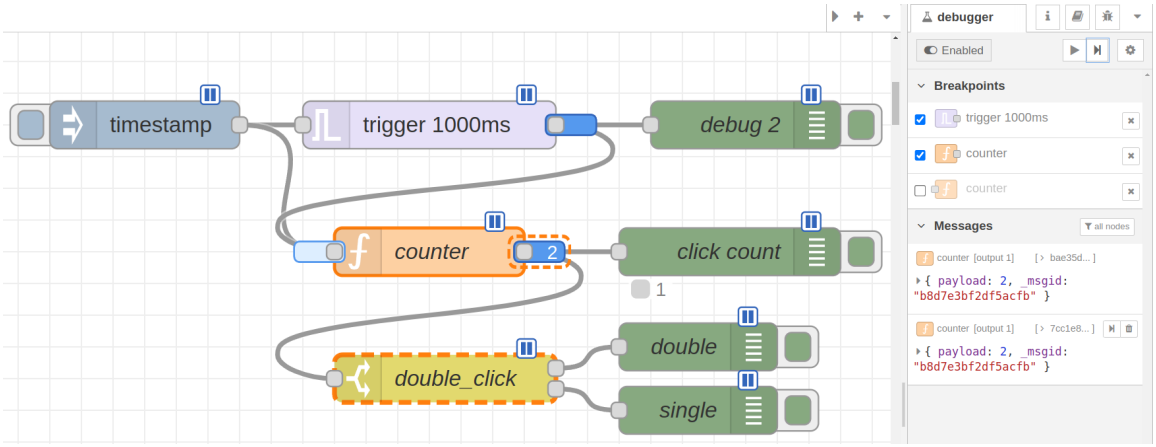


Figure 4.3: Screenshot of Node-RED flow debugger. On the left is a program for detecting double or single clicks. Blue rectangles attached to the outputs and inputs of nodes signify breakpoints. On the right is a panel, showing all messages in the system and a list of active and inactive breakpoints. The position and destination of a single message is highlighted with an dashed orange border.

Diagram VPLs

In the Node-RED VPL, which is a representative of the diagram–based VPLs, debugging is officially only supported using the `printf` debugging method. Special debugging nodes can be inserted into the flow, which simply copy each message from the node's input onto its output, but they also print the message in a debug log window. If the debug node is given a name, the message will be printed alongside the name, to make it clear, which debug node printed it.

There exists a plugin[1] for Node-RED, which adds some interactive debugger functionality. Breakpoints can be set on the inputs and outputs of nodes and are indicated by a blue rectangle attached to the input or output. Any incoming or outgoing message will trigger a breakpoint, pausing the entire flow. Since Node-RED does not have a simulator and can only run in real-time, any messages received from outside of the paused flow (e.g., a message from a physical sensor received over MQTT) will be put into a queue of the output of the node which produced that message. Same applies for messages, which triggered a breakpoint. They are also put into a queue of the respective input or output. Messages in queues can be inspected and discarded, preventing them from propagating through the system. Queued messages can be "stepped", causing them to be processed. The message to be processed will either be chosen from the top of the global runtime queue, or it can be chosen by the user. This could be considered an analogue to the `step over` functionality seen in regular source-level code debuggers. However, it can also act as the `step in` command, since sub-flows are not treated as a new node, but only a grouping of nodes. Finally, the flow can be unpaused and all the queued messages will be processed by the runtime. An example of a paused flow is shown in Figure 4.3. In this example, two breakpoints are set and active, while one is inactive. The right pannel shows a list of messages in the system. Hovering over a message highlights its queue and its destination.

[1] https://flows.nodered.org/node/node-red-debugger

Another example of a diagram–based VPL debugger is the Unreal Engine Blueprints[2] debugger. Unlike the Node-RED debugger, breakpoints are placed on an entire node, instead of specific inputs or outputs. When a node with an active breakpoint is about to be executed, the program is paused and the node is highlighted in the editor. An example of a program with a set breakpoint is shown in Figure 4.4. Since UE Blueprints allow for `pinned` Blueprint variables, the debugger offers a way to mark these variables to be *watched*. If a watched pin has already been executed, it will display the latest value it has been assigned in a *watches* window. Hovering over any pin can also display this information, given the pin has already been executed. Execution controls offer the standard set of actions: *pause*, *resume*, *step-over*, *step-into* and *step-out*.

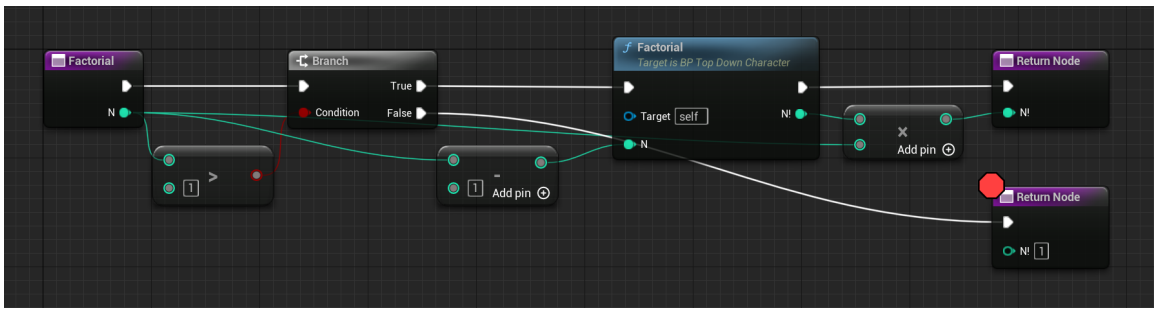


Figure 4.4: Example of a UE blueprints program[3], which defines a function for computing the factorial. The right-most bottom node has an active breakpoint, which will pause the execution, before that node is executed.

It also offers two ways of inspecting the flow of execution. One of them being a *call stack*, which shows a stack of the functions, which are currently being executed. Each function in the stack has been called by the function directly below it. A function represents a group of nodes. To see the flow of execution on the level of individual nodes, the *execution trace stack* can be used. It shows a history of the executed nodes, with the most recent one being on the top. This list updates as the user steps through the node graph.

4.2 Software Simulators

Simulators are software, which acts as another system, platform or architecture for purposes of executing code, in an environmnet which is as close as possible to the actual deployment environment. Simulation of the execution environment is needed because some functionality can only be tested on specific platforms, which have functionality not present on the development machine, or the functionality assumes a system of distributed devices, which all communicate and react to messages.

As an example, if code for a microcontroller counts on being able to set and read the state of input and output PINs, the environment either has to have actual PINs which can be used, or they have to be simulated. In the same manner, if a piece of code is designed to run in an environment, in which it communicates with external devices, it might not

[2] https://dev.epicgames.com/documentation/en-us/unreal-engine/blueprint-debugger-in-unreal-engine

[3] Originally from https://dev.epicgames.com/documentation/en-us/unreal-engine/blueprint-debugger-in-unreal-engine

be possible to run the the program which runs on those devices locally, thus requiring the system to simulate such external dependencies.

Depending on the purpose of the simulator, the simulation can be more or less complex. It might not be desirable to simulate the complete original system, but only a small subset, relevant to the task. External dependencies don't have to be simulated in their entirety. Even if the process of reacting to a message is a complex task for an external device, to the outside world, the only thing visible is the change in the devices reported state. Such a device could then be simulated simply by changing its state to a predefined value as a response to a given message, or by calculating the new state using a simplified formula.

An example of such simulators is the Wokwi[4] project, which can simulate various micro-controllers and peripherals. This allows it to run the original firmware, intended for the real device. It can simulate interaction between the MCU and the external devices connected to its pins. There is only a limited number of external devices (i.e. sensors, displays, memory modules) in the default library. However, it is possible to define custom simulation models, which can then be added to the simulated hardware setup. This ability makes the simulator useful for real-world hardware projects, which may use less common components. Custom chip definitions consist of a JSON file, which contains the chip's interface definition and a implementation file, containing the code responsible for reading the communications from the outside and generating responses on the chip's pins. The implementation file can be written in any programming language capable of being compiled to WebAssembly[5].

The simulator offers a web interface, that contains a code editor, together with a visual editor, in which it is possible to place and connect components. The web interface can be seen in Figure 4.5. Not all features of a given micro-controller can be simulated, like the wireless communication capabilities of some MCUs. The exception is WiFi on the ESP platform, for which the simulator has implemented support. This makes it possible to simulate even devices, which might require communicating with servers or other devices over the internet, such as IoT devices. Another interesting feature of the simulator is the built-in logic analyzer component. This functions like a real logic analyzer, allowing the user to view the traces of digital signals from selected pins.

Savidis et al. [39] designed an IoT system, with integrated simulator and debugger for end-user created automations. The system is built on the IoTivity[6] middleware, which handles the smart objects in the system. They also created an API abstraction layer which sits on top of the IoTivity middleware, to allow them to switch between real and simulated devices, for the purposes of testing and debugging.

Each device in their system is described by a definition file, containing details about the device, its operations and attributes. Using this description, their system is able to generate UI widgets for each device automatically.

End-user automations are programmed in a VPL based on the Blockly library. They provide blocks related to the automation tasks that users will perform, as well as the runtime which executes the generated code from the Blockly program. When running in simulation mode, no communication with real devices is permitted. When a command is sent to a virtual device, it needs to know how would the real device respond to it – how would it change its state in the system. This logic has to be provided by the user, in the form of visual programs, which specify what should happen as a result of a command to that particular device. This needs to be implemented for both simple and complex operations.

[4] https://wokwi.com/
[5] https://webassembly.org/
[6] https://iotivity.org/

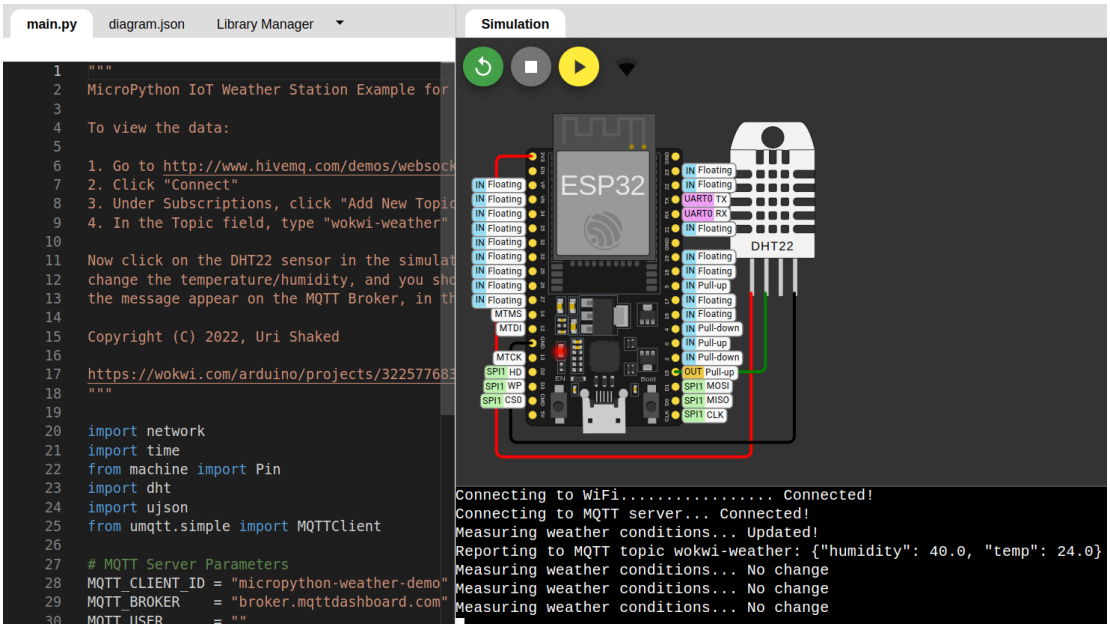


Figure 4.5: Wokwi web interface with the code editor on the left, a virtual workspace on the right and console output on the bottom. The virtual workspace is used for placing and connecting external components to the main MCU. The image shows a paused simulation, in which case the state of each MCU PIN is displayed near it.

To enable proper testing of automations, when running in simulation mode, their system can change the passage of time for all simulated devices and programs. Time can be slowed, sped up, or paused entirely. Since some devices or programs might also behave differently according to the day and month, the user can change these values while running the simulation.

Users can use the simulator by creating test-suites, which consist of predefined changes to device properties at the specified time in the test. This way, a user can create a scenario, which will then play out in the simulator. Another type of supported test allows the user to setup monitoring for certain device properties, which when set to a particular state will warn the user, or trigger a pause in the simulator execution.

Chapter 5

Solution Proposal

When designing the system for testing and debugging of end–user IoT automations I took inspiration from the works of Deiner [10] and Savidis [38] for the debugger, as well as the UE Blueprints debugger. The simulator was inspired by the Wokwi system and the IoT system simulator designed by Savidis et al. [39]. Based on the review of literature and existing solutions, I decided to utilize a VPL for the user created automations, as it is both usable by inexperienced users with no software development skills and experienced users, without being too limiting.

It will allow users simulate different conditions and scenarios for their IoT systems, providing them with a clear view of how their programs react to device attribute changes and how multiple programs interact together.

The word *automation* will be used to describe an end–user program, enhanced with metadata required for its execution and debugging. The metadata includes:

- the automation name, which uniquely identifies it
- a list of device attributes, whose changes trigger this automation (cause it to be executed by the system)
- breakpoints in the program
- a list of errors and warnings

The VPL editor will be based on the work of Podvojský [31], which presents a web–based VPL editor for a block–based VPL. The language consist of form blocks, which can be chained or nested. These form block represent basic programming constructs such as if statements, loops and similar. They also represent actions available in the system, like sending a notification to a user or device specific actions, which can tell the device to perform some action. The editor and VPL are designed with mobile devices in mind, making them perfect for use in home automation systems, where mobile devices are very common. An example screenshot of the selected VPL and its editor can be seen in Figure 5.1. It was chosen instead of a more mature library such as Blockly, primarily because this style of a VPL is better suited for mobile devices and it is tailored for use in IoT systems.

The system will be developed as a web application, since it enables the application to be usable on both larger and smaller screens. Web technologies offer a robust development platform, which can be used to create applications for a wide variety of devices and platforms. It also enables the possibility of embedding the applications as a plugin or a component into other IoT systems, a lot of which are also implemented as web applications.

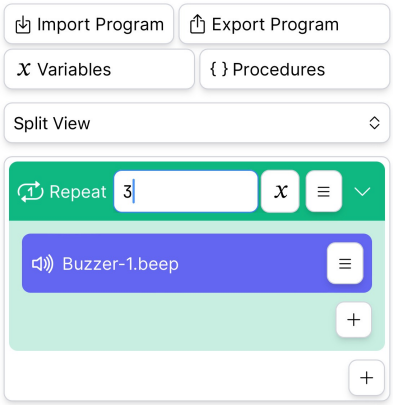


Figure 5.1: Example of the selected VPL and its editor being used on a mobile device. Taken from [31].

Special care should be taken to make sure the system is usable on small screen devices, as many users expect to interact with their IoT systems from their mobile devices.

5.1 System Architecture

The core of the system can be split into three main parts, which can be seen in Figure 5.2. All parts should be implemented as separate modules, providing well designed APIs for interaction. The three parts have the following purposes:

- **VPL Interpreter and Debugger** – a custom, specification compliant interpreter for the selected VPL, specially designed with debugging and tracing in mind.
- **Simulator** – for simulating external devices and other capabilities of the main system. No interaction with any IoT system backend should be necessary, all devices are purely virtual.
- **Scenario Runner** – provides the user–facing interface, through which both the VPL execution and the simulator are controlled. Allows for running test scenarios, which simulate external changes to devices, based on which user programs are triggered.

All necessary data is loaded from the backend of the IoT system, in which the simulator and debugger is to be used. To make it possible to integrate the simulator and debugger into different IoT systems, all data coming from the backend is first transformed into a unified internal representation. Adding support for a new backend can be accomplished by writing a converter from the backend representation to the internal representation. For automations this means parsing the serialized program into an internal abstract syntax tree. Device internal representation consists of the device's name, its current attribute values and its device model. A device model specifies what attributes and functions this device has. Multiple devices can share the same device model. To make a device model fully compatible with the simulator, the backend needs to also provide some simulator specific data for each device model. This is then referred to as the extended device model. It is described in more detail later in this section, when the simulator design is described.

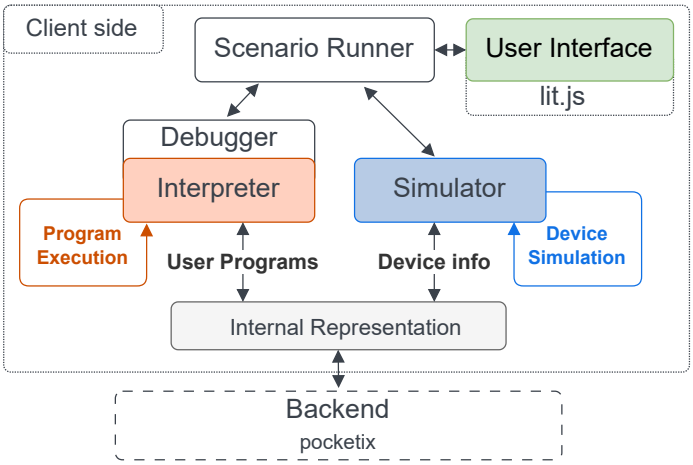


Figure 5.2: Proposed system architecture.

VPL Interpreter and Debugger

The interpreter works with an internal representation of the VPL program. It is separate from any interpreter used on any backend implementing this language, so that it is not tied down to a particular backend implementation. Instead, it will adhere to the language specification and be thoroughly tested.

Before a program can be ran, it needs to be parsed from its native format to the internal representation. As part of this process it is also validated. If the validator produces any errors or warnings, they are added to the automation's metadata. The rest of the system then works with the internal representation. This way, the system is more isolated from changes to the format of the language. Adapters can be created for different versions, as long as they can translate the input program into the standardized internal representation.

Each VPL program is a sequence of blocks, where each block can have other blocks nested inside of it. The execution will proceed sequentially through the list of blocks, moving to the next after the current block is fully evaluated, including any nested blocks and its arguments.

The interpreter will provide an API for stopping execution, which can then be used by the user to control the execution from the UI, or it can be used by the debugger module to stop the program when necessary. It will also provide an API for setting breakpoints. Breakpoints will be set on the level of form blocks. The breakpoints will be checked by the interpreter during execution, which will be paused when a breakpoint is encountered. This event will also be reported to the debugger module. The interpreter should not be able to resume execution on its own, rather it has to wait for a command from the debugger module. All breakpoints will be managed externally to the interpreter module, but it will check if a breakpoint is triggered and stop the execution. It will also offer APIs for tracing of the program execution.

Automations can only affect the rest of the system by sending commands to devices. These commands are passed on to the simulator, which processes them and reacts correspondingly. When an automation is running, it also gets a snapshot of the simulator state at the time of the automation's invocation. Device attributes can then be read as device variables in the program.

Simulator

This module handles the external devices, as well as the passage of time. The simulator will run in discrete time steps, not tied to the real passage of time. This way, the system can simulate faster or slower passage of time compared to reality, which is useful for testing time based programs. It also provides an API, which the Debugger can use for collecting necessary information about all the operations and changes in the system.

Each device present in the system has a virtual twin, which is used in the simulation. Changes made to the virtual device have no effect on the real device, so that the simulation is completely isolated from the real system. Each device belongs to some device model, which specifies what attributes and operations the device has. The virtual device has the same attributes and presents the same operations as the real device, but does not have the same internal logic for those operations, since that is implemented on the actual device hardware. How a device responds to different commands will instead be part of the extended device model specification, which should be provided by the backend of the system, in which the simulator is utilized. The device model specification including implementations for its operations will be serializable and exportable, so that they can be shared between users and so that systems utilizing this debugger can provide specifications for all devices with built-in support. They can offer community repositories for other devices. This should minimize the friction of debugging and testing automations for less experienced users.

The simulator should contain an internal priority queue of events, each scheduled to happen at a given point in time. Stepping through these events will then consist of finding the next event scheduled to happen, setting the internal time to that of the event and processing said event. The event queue will not be static, as processing of an event might cause other events to be scheduled. Another way events might be added to the simulator queue is scheduling and executing commands sent to devices from some running program. Sending device commands is the only way automations have of modifying the simulator state. How devices react to a command is determined by the previously mentioned extended device model specification.

Changes to the simulator state, such as device attribute changes, new events being added to the event queue and similar should be emitted as events, or propagated to the rest of the system by invoking a given callback. The simulator should never advance through the event queue without outside instruction to do so.

Scenario Runner

Providing the user-facing interface will be the role of the Scenario Runner module. It coordinates operations between the VPL Interpreter and the Simulator. It manages breakpoints set by the user, informing the Interpreter about them, which in turn informs the Debugger when a breakpoint is hit and VPL execution is stopped.

The entire system is always started by the Scenario Runner module, by starting a test scenario. A test scenario contains a set of device state changes, scheduled to happen in specific times. It also contains assertions about the system and device state. If some assertion fails, either the entire test can fail and stop execution, or it will just be marked as failed, but execution will continue. If the execution is stopped by a failed assertion, the entire state of the system is visible to the user. The program execution is stopped at the block which was executed last before the assertion. A scenario is the same as a test case, but does not need to include assertions. It can be used during development of the program for quick debugging or stepping through the program and observing the state changes.

The Scenario Runner module collects tracing information from both the Interpreter and Simulator modules, for use in the advanced debugging techniques this module offers, namely stepping backwards in time in the scenario timeline, to get a better understanding of what effect each step has on the state of the devices and what automations are triggered and when. Stepping in reverse should be particularly useful, when the execution is stopped by a failed assertion, and the user wants to replay the steps that led to the failure, without the need of rerunning the entire test.

5.2 User Interface

The user interface is built around the selected VPL editor, extending it in several ways. Controls for program and scenario stepping are added to the bottom of the screen when viewed on a mobile device and to the top when viewed on a larger screen. On larger screens, variable watches, breakpoint list and simulator state windows are added alongside the editor, while on mobile devices these windows are collapsed and can be opened upon clicking their respective buttons, in order to save space when they are not needed.

The timeline of the currently running test scenario is displayed in the top half of the screen on mobile devices, while on larger screen devices it is displayed to the left of the program display area. This timeline starts with the test scenario events, and is dynamically updated with new events, which arise as the test scenario is being executed. It should also display what automations were triggered by what device attribute change. It should also be clear what, if any, commands were sent by each execution of a given automation. The event which will be evaluated when the next step is taken is highlighted using a border.

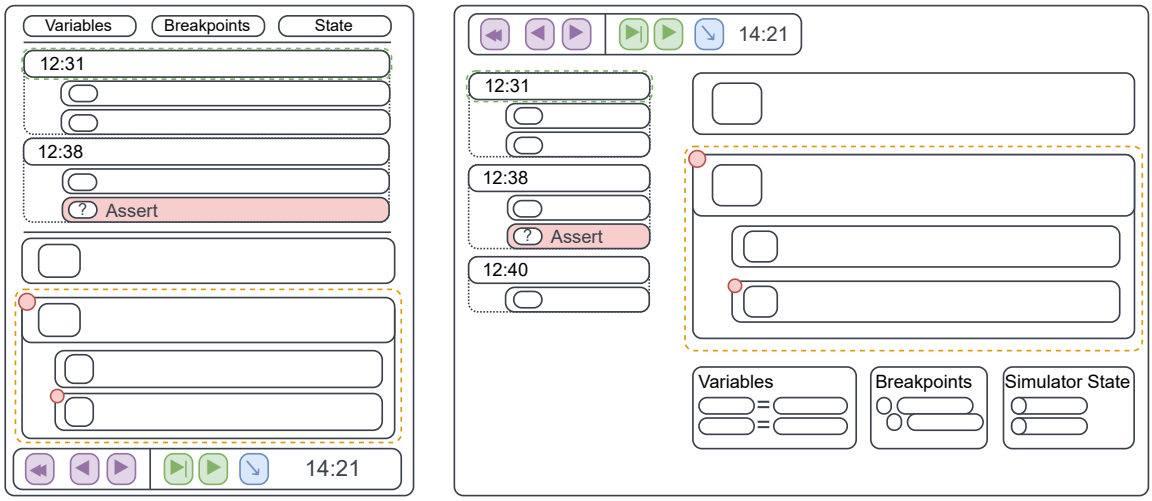


Figure 5.3: Mockup of the scenario runner user interface. Mobile version can be seen on the left and a version for wider screen is on the right. There are two main elements, the scenario timeline and the running program area. Dashed borders are used to highlight the timeline event (or the VPL block) which will be executed next. Red dots on VPL blocks mark breakpoints. Bottom panel (or top panel in the right version) includes buttons for program and scenario stepping. The mobile version has additional windows hidden by default, but they can be opened in a pop-up window by clicking on their associated button.

When an automation was triggered by some change to a device attribute, its program is displayed in the program area (bottom on mobile devices, left of the screen on larger

devices). To display the program, the VPL editor The block about to be executed is highlighted by adding a border to it. Breakpoints are marked with a red dot in the upper left corner of the block. They are filled when active, or only displayed as an outline when inactive. A mockup of this interface can be seen in Figure 5.3.

Test and scenarios are constructed by adding actions like property modifications or assertions to a list, similar to how the topmost form blocks are added to the programs. A mockup of this interface can be seen in Figure 5.4 The difference being, that actions in tests and scenarios can't be nested in one another, instead, they are associated with a given point in time. This association should be denoted by nesting the action in a pseudo-block, representing the desired point in time. If multiple actions are to happen at the same time, they are simply nested in the same time block.

The main event type used in the test scenarios denotes a change in a device attribute. This simulates user interactions with devices (e.g. pressing a button on a device) or a device reporting a change in the state of the system (e.g. thermometer measuring a change in the room's temperature). This basic event is enough to simulate any change in the simulated environment. The other available event is the assert event. This event can be used to add checks of the simulated environment.

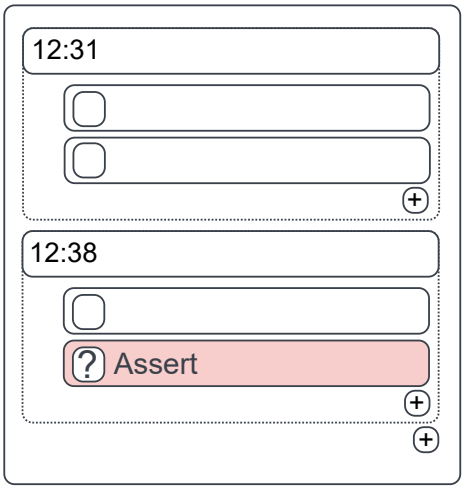


Figure 5.4: Mockup of the test case editor interface. Time blocks can be added using the outer-most plus icon. Each time block has to include at least one action or assertion. Editing the time blocks associated time is done by clicking on the time text. Time blocks can be reordered by holding and dragging.

Chapter 6

Implementation

As stated in the solution proposal, web technologies are a very good choice for implementing a system like this. They provide a great platform, enabling the system to run on a wide range of devices, without any need of platform specific code. One goal of this work is to provide a library, which can be used in different IoT frontend applications (such as the Pocketix[1] IoT ecosystem), web technologies are a natural fit, since a lot of these applications are also web applications.

This decision narrows down the possible options when it comes to picking a programming language. At the time of writing, the only supported programming language for all major browsers is JavaScript[2]. It, unfortunately, does not support static typing, which I find very important in any sizable code base. This is the main reason why I chose to use TypeScript[3] instead, which is a very popular language, that builds upon and transpiles down to JavaScript.

To make the resulting library usable in any frontend application, it can't be tied to a UI framework, which is not easily composable with other frameworks. These requirements make web components[4] an ideal technology for this work. They are a standardized web API, which allows for the creation of custom HTML tags, that encapsulate all their custom functionality and/or styling. A web component can be created without the need for any additional libraries. It can be represented by a single JavaScript file, which if then imported into an HTML document, make the custom HTML tag available. Such a tag can then be used in normal HTML code, making it possible to use without any additional JavaScript code.

Even though it is possible to create web components without any additional libraries, for more complex components (such as the proposed VPL simulator and debugger), it is beneficial to use a library, which provides a convenient reactive framework, while still adhering to the web component specification. One such library is Lit[5]. The decision to use this library was also supported by the fact, that the VPL Editor [31], which will be used and modified as part of this work is built using the same technology.

This chapter will describe the implementation of the solution proposed in Chapter 5. First the specification of semantics for the existing VPL is described in Section 6.1. Then in Section 6.2, the process of parsing this language into an inner representation meant for

[1] https://github.com/pocketix
[2] https://developer.mozilla.org/en-US/docs/Web/JavaScript
[3] https://www.typescriptlang.org/
[4] https://www.webcomponents.org/introduction
[5] https://lit.dev/

the purposes of later interpreting this program is described. This section also talks about the implementation decisions made to make the language interpreter capable of pausing execution when running the program inside the debugger. The simulator and device model specification is presented in Section 6.3. Section 6.4 describes in detail how the library of UI components was designed and what changes were made to the existing VPL Editor.

6.1 Language Specification

The work of Podvojský [31] focused on creating an extensible and flexible VPL editor. The language designed in this work includes some syntactical rules, but no semantic rules. Before I could start implementing an interpreter for this language, a specification of its semantics had to be created.

Types and Type Conversion

The language supports the following basic types: `number`, `string` and `boolean`. The number type encompasses both integers and floating point numbers. Since the target users are both people with programming experience and novices with little to no experience, the type system has to be intuitive and not too constrictive. For this reason, I decided to allow for implicit type conversions, making the language weakly typed. Enforcing explicit type conversions would require adding functions to facilitate such conversions. Considering the target users, I think this would negatively impact their ability to write and read the code. Also considering the focus on use on mobile devices, as specified in Chapter 5, adding more visual clutter to the user program would negatively impact readability given the limited space on small screens.

The way implicit type conversions are done is shown in Table 6.1. The only potentially problematic conversion is from string to number. In many languages, TypeScript included, trying to convert a string value into some numerical type is the same as calling a function which tries to parse said string into that type. In TypeScript, since there is only one main numerical type (excluding `BigInt`, as its usage is usually rare), we can use the builtin `Number()` function[6] to perform the parsing. This way, we can take advantage of the existing string to number conversion[7], instead of creating a custom implementation. The conversion of a string to a number will fail, if the entire string can't be interpreted as a numerical value.

	number	string	bool
number	n/a	`.toString()`	`n != 0`
string	`Number()`	n/a	`string.length > 0`
bool	T:1, F:0	`"true"/"false"`	n/a

Table 6.1: Conversion matrix from a given type on the left, to a given type on the top. Conversion can be attempted for every pair of types. Trying to convert a string to a number might fail, if the string can't be interpreted as a numerical value. All other conversions will always succeed.

[6] Calling `new Number()` (instead of just `Number()`) calls the `Number` constructor and returns a `Number` object instead of a `Number` primitive value, which is usually not desired.

[7] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Number#number_coercion

All other conversions will always be successful. Any number value can always be encoded as a string. For this conversion I also decided to use the built in number to string conversion available in TypeScript. Number values have a `toString()` method, which is wholly sufficient for this use case. When converting a boolean value to a number, `true` values are always converted to the value `1` (one), and `false` values always to a `0` (zero). Converting a number to a boolean is similar. A `0` (zero) number value is always `false`, any other value is converted to `true`. When converting strings to boolean values, the length of the string is used as the determining factor. If the given string has length of zero, it is converted to `false`. If the string is non-empty, it will result in `true`. Boolean values are simply converted to strings containing the words `true` of `false`.

Operators and Implicit Type Conversion

Each operator is assigned a default operand type. A list of all supported operators and their default types can be seen in Table 6.2. This is the type that all of its operands will be converted to. For example, if one wanted to compare a number and a boolean value like this: `12 > true`, both of the operands will get converted to a number. The first operand is already a number, so no conversion is needed. The seconds operand is converted using the rules discussed above. This expression then becomes `12 > 1`. It makes sense to define other type combinations, which are naturally supported by a given operator. For example when testing if one string is greater than another string, it is logical to expect that the strings will be compared based on their lengths. Similarly, the expression `"a" + "b"` should be interpreted as string concatenation. These alternative operator overloads can be seen in the column "Overloads" in Table 6.2. Operators `==` and `!=` for testing equality are defined for any pair of matching types. In case of the operands having different types, the operands will always be evaluated as not equal.

Operator	Default operand type	Overloads
`>`	number	(string, string)
`<`	number	(string, string)
`>=`	number	(string, string)
`<=`	number	(string, string)
`==`	any pair of (T, T)	
`!=`	any pair of (T, T)	
`+`	number	(string, string)
`-`	number	
`*`	number	
`/`	number	
`%`	number	
`and`	bool	
`or`	bool	
`not`	bool	

Table 6.2: Supported type combinations for each operator. Each operator has an associated default type. If the operator is used with any other type combination, the operands will be converted to the default type. The behavior is different only if the operator supports different type combinations, listed in the Overloads column. If both operands match the overload specification, no conversion takes place.

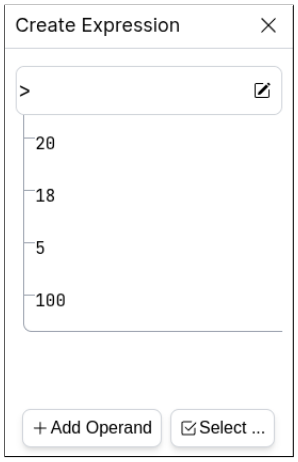


Figure 6.1: Example of an expression with operator chaining as implemented in [31]. The resulting expression in infix notation would be written as `20 > 18 > 5 > 100`.

The VPL Editor [31] allows for unlimited chaining of the same operator. Meaning that one can write the equivalent of this expression `80 > 51 > 30 > 21` by inserting just one `>` (greater than) operator node. An example of such operator use can be seen in Figure 6.1. This functionality can be useful, to make creating expressions easier, especially on mobile devices. But in my opinion, allowing unlimited operator chaining could lead to confusing expressions. I believe it is worth the extra steps of having to explicitly create multiple sub-expressions and chain their results together with different operators. To strike a good balance between convenience and understandability, I limited most comparison operators sense to maximum of three operands. For equality testing operators, the limit is not imposed since I think it is clear what longer chained expressions mean. For other types of operators, I also think that there is no reason to limit the maximum number of operands. The exact number of allowed operands for each operator are shown in Table 6.3. This table also shows examples of how each chained operator is to be evaluated internally.

When it comes to types and type conversion in expressions with operator chaining, the rules are the same as for expressions without chaining. If any operand is not of the default type for the given operator, it will be converted. Unless all operands satisfy some overload of the given operator. In that case, this overloaded operator will be used instead.

Variables and Device Attributes

The user has the ability to create variables, which can later be modified and read in the program. Device attributes are treated almost the same as user variables by the VPL Editor. That is, if a user wants to compare the value of a device attribute in an expression, the resulting program can't distinguish between user variables and device attributes (called "device variables" in the VPL Editor). This is problematic, since for one, device attributes can't be modified in the same way that user variables can, but more importantly, the VPL Editor does not prevent the user from creating a user variable with the same name as some device attribute. The VPL editor itself is able to distinguish the two, but the moment the program is serialized and exported for further processing, this ability is lost. One possible solution would be to create two different tags, and assign the tag `variable` to user variables and `device-attribute` to device attributes. This would require significant changes to the VPL Editor. Instead, I chose to add a prefix to any device attribute (when

Operator	Arity	Example	Transforms into
`>`	2-3	`5 > 8 > 2`	`(5 > 8) and (8 > 2)`
`<`	2-3	`5 < 8 < 2`	
`>=`	2-3	`5 >= 8 >=2`	
`<=`	2-3	`5 <= 8 <= 2`	
`==`	2+	`5 == 8 == 7 == 6`	`(5 == 8) and (8 == 7) and (7 == 6)`
`!=`	2+	`5 != 8 != 7 != 6`	`(5 != 8) and (8 != 7) and (7 != 6)`
`and`	2+	`5 and 8 and 2`	evaluation short-circuits when operand is `false`
`or`	2+	`5 or 8 or 2`	evaluation short-circuits when operand is `true`
`not`	1	`not True`	
`+`	2+	`5 + 8 + 7 + 8`	`(((5 + 8) + 7) + 8)`
`-`	2+	`5 - 8 - 7 - 8`	`(((5 - 8) - 7) - 8)`
`*`	2+	`5 * 8 * 7 * 8`	`(((5 * 8) * 7) * 8)`
`/`	2+	`5 / 8 / 7 / 8`	`(((5 / 8) / 7) / 8)`
`%`	2	`5 % 8`	

Table 6.3: Operator arity and chaining specification. The arity column specifies how many operands each operator can be used with. The "Transforms into" column shows an example of how operators with arity bigger than two are actually evaluated.

used as a variable) and prohibit user variable identifiers from starting with this prefix. The prefix I chose is the `$` (dollar) symbol. Making sure the VPL Editor complies with this modification was simple and it did not require making any changes to the syntax for the existing language. Other than this one limitation, user variable identifiers can be an arbitrary string.

Blocks with Limited Expression Types

For modifying the value of user variables, the language includes the `SetVar` statement. This statement is given an identifier (`lval`) and some arbitrary expression (`rval`) which, after being evaluated, is assigned as the new value associated with the identifier.

The `Repeat` statement serves as a very simple looping construct. Its argument can be only a numeric literal value. Which is used as the counter for the number of iterations performed by this statement. For more complex loops, the `While` statement should be used. It allows an arbitrary expression as its argument. Same applied to the `If` and `ElseIf` statements, which also support arbitrary expressions as their conditions.

The `Switch` statement can be used as a more streamlined branching construct. It requires a variable (user or device) as its argument (selector). Only `Case` statements can be nested inside of a `Switch` statement's body. For simplicity, only literal values are allows as the argument (case expression) for a `Case` statement. Some languages allow arbitrary expressions in place for a case expression. But in order to keep the `Switch` statement simple, if more complex branching conditions are needed, the user should use a the combination of `If`/`ElseIf`/`Else` statements.

6.2 Interpreter and Debugger

The first step of taking an exported program from the VPL Editor and executing it, is parsing and validation. Right now, the VPL Editor exports the program as a JSON document,

which forms the source text of the program. Since the entire pocketix ecosystem is in active development, the format and style of the source text might change with newer versions of the VPL Editor or other parts of the ecosystem. For this reason, I wanted to design the Interpreter in such a way, where parsers for specific versions of the VPL could be written, which would emit the same standardized internal representation used in the rest of the Interpreter and Debugger. Later if the serialized representation of the program changes, it will be possible to adapt only the parser, while the rest of the system stays the same. Of course if the language goes through a more substantial change, modifications might need to be made to the internal representation, so that the new features and semantics can be properly executed. But having the separation of external and internal representation should make this process easier.

The current parser takes in the raw JSON representation of the program in the VPL version 1 format and produces a list of `ASTNode` objects, where each node represents a statement from the original source. Alongside the internal representation, the parser can also optionally produce:

- a list of parsed and validated user variables,
- a list of all device attributes read in the program
- and a list of warnings produced in the process of parsing.

If an error is encountered while parsing the input program, the list of `ASTNode`s is not produced, instead the call of `parse()` throws an error. In order to provide the best experience for the user, the parser tries to find as many errors as possible, even after the first error is found. The parser recovers at the nearest place to the found error and continues with normal parsing. After the entire program is parsed, the list of all errors is compiled and thrown as a custom exception type `ParseError`. This custom error can contain a list of other errors as its body. In that case, the outer error is marked as a container error. This nested structure corresponds to where in the source structure the error was found. If for example an error in the condition expression of a `While` statement was found, the resulting nested error structure would look like this: `InvalidProgram > ParseStatementListError > InvalidWhile > ParseExpressionError > UnknownVarIdentifier`. Only the last nested error will not be marked as a container. This nested structure can be used to gain additional information about the exact details of the found error. When displaying the errors in the VPL Editor, only the non-container errors are displayed, since they contain most of the information.

In order to be able to associate each error with the correct visual block in the VPL editor, errors also have a `path` property. The type of this property can be seen in Listing 1. The path consists of a list of steps, which represent the steps needed to be taken from the root of the program, all the way to the offending part of the program. Each step in the path has a type, which can either be `statement` or `expression`. This determines whether to step to the next nested statement, or the expression associated with the current statement. To reach the condition expression of a top level `If` statement, the two steps would be `statement > expression`. The numerical value `pos` present in each step represent the position of the target in its containing list. If a statement has two nested children, the first nested statement will have `pos: 0`, the second will have `pos: 1`.

The `BlockPath` type can be extended in the future, to support different types of path steps. The path is generated by the language specific parser and the language specific editor, never the interpreter itself.

```
type BlockPath = {
    type: "statement" | "expression";
    pos: number;
}[];

// could point to While > Expression
let p1 = [{type: "statement", pos: 0}, {type: "expression", pos: 0}];
// could point to If > While > Expression
let p2 = [{type: "statement", pos: 0}, {type: "statement", pos: 1},
          {type: "expression", pos: 0}];
```

Listing 1: Type used to associate errors and other metadata with the original (external) source text. The association is done by constructing a path, where each new item in the list is a new level of nesting. Each step of the path is marked as either a statement or an expression, along with its position index in the parent list. Below the type are examples of two paths and the possible target they could represent.

Each type of statement from the VPL version 1 specification is represented by a class, which implements the `ASTNode` interface. I decided to implement the parsing in such a way, where each statement class knows how to construct itself given the corresponding raw JSON program. Each class knows what is required for its construction and can report errors it encounters to the parser object. This object serves mainly to find the correct statement class which should try to parse the next statement, based on keywords and other syntax of the JSON program. By moving all the actual parsing logic into the statement class itself, it will be very simple to add a new kind of statement. After the class which represents this statement is implemented, it only needs to be added as an option to the `parseStatement()` method of the parser.

Some statements like `ElseIf` or `Case` are special, in that they can only appear after or nested inside other specific statements. This is implemented by passing a context object to each statement class parser, which contains information about what is the current parent and previous statement. If a statement requires a specific parent or predecessor, it can validate if that condition was met, or throw an error.

The interpreter can be used to evaluate a tree of nodes, which implement the `Evaluable` interface, which can be seen in Listing 2. This interface prescribes a method `evaluate()` which takes in the current environment, evaluates the given statement and returns the result. The type also specifies other potential arguments to this functions, this is so this interface can be extended later, when defining interfaces for the debugger and it will be discussed more later.

In my current implementation of the individual statement classes, they not only know how to construct themselves from the raw JSON program, they also implement the `Evaluable` interface and know how to evaluate themselves. This design keeps both the parsing and evaluation logic in one place, making it easy to keep in mind what each statement needs, which attributes it has and similar. One disadvantage of this design choice is that if someone was only interested in validating a given input program, the parser would produce a list of nodes which also include the evaluation logic. The downside is that the size can be unnecessarily large when used in situations where the evaluation logic is not needed.

```
interface Evaluable {
    evaluate(env: Env, ..._: any[]): Promise<EvaluateResult>;
}
```

Listing 2: The `Evaluable` interface, which must be implemented by nodes meant to be ran by the interpreter.

When running, the program can modify its variables, which are stored in the environment object. An environment implements the `Env` interface, which is shown in Listing 3. It offers set and get operations for user created variables, as well as for internal bindings, which is used for example by the `Repeat` statement, in order to keep track of the loop counter. Environments can be nested. Nesting environments corresponds to creating a new scope in the program. When looking for a binding given its identifier, the current environment is checked, if no binding is found, the parent environment is checked. This continues until the top-most environment is reached. If even this environment does not contain the request binding, the binding is not defined.

```
interface Env {
    parent: Env|null;
    get(identifier: string): Binding|null;
    set(identifier: string, value: TaggedBindingValue|null): void;
    getInternal(identifier: string): Binding|null;
    setInternal(identifier: string, value: TaggedBindingValue|null): void;
    getLocalBindings(): {[ident: string]: Binding};
    getInternalBindings(): {[ident: string]: Binding};
}
```

Listing 3: The `Env` interface, which is used by the interpreter to keep track of identifiers and their associated values.

Debugging of a program is facilitated by the `Debugger` class. It takes nodes which implement the `Debuggable` interface, seen in Listing 4. This interface requires the implementing objects to define a similar `evaluate()` method as is prescribed by the `Evaluable` interface. Main difference being, that this method takes a debug hook callback function, which will be called inside the `evaluate()` method in places, where execution should be paused and handed over to the debugger. This typically happens right at the start of the method, just before the normal evaluation logic is run. This callback can then be used by the debugger to block the execution, until some signal (such as a step requested by the user) is received.

```
interface Debuggable extends Omit<Evaluable, "evaluate"> {
    evaluate(env: Env, hook: DebugHook): Promise<EvaluateResult>;
}
```

Listing 4: `Debuggable` interface, which needs to be implemented by a the internal program representation, so it can be used in the debugger.

The debugger hook receives information about the current environment and the node currently being executed. This information can then be presented to the user, or used in other ways to decide how the execution should continue.

All other debugging functionality is built around this hook callback function. For example, in my current implementation, you can pause execution and step only on statement nodes, not expression nodes. Since there is not really a good way to visualize that to the user, and does not provide any benefit for understanding the program better. The debug hook is called even for the expression nodes, but when the debugger sees what kind of node is about the be executed, it lets it continue, since it should never stop here. Other functionality is implemented in a similar manner. For running until a breakpoint is hit, the debugger simply checks if the current node has a breakpoint set. If not, it lets it continue, just as if the user requested a step forward.

6.3 Simulator

The simulator consists mainly of the device state and the event queue. The queue contains events scheduled to be processed in the future. Stepping the simulator consists of getting the next event from the queue, setting the simulator time to that of the event which is to be processed and processing the event. A simplified representation of the event type can be seen in Listing 5. There are three possible event types:

- `SetAttribute`, which immediately changes a device attribute value. This event is set by the user, when creating a test scenario.
- `ReceiveCommand` is an event, which is added to the event queue, when some running automation sends an event. Since transmitting and processing an event on the actual device hardware always takes some time, the simulator does not process the command immediately, but schedules its processing for a future time, to simulate network and processing delays. After a command is actually processed, it can result in new `SetAttribute` events being added to the timeline – this is how the simulated device responds to a command, which results in the device changing state.
- `Assert` events can only be added by the user when creating a test scenario. This event contains a condition, which is evaluated when the event is processed. If the condition is true, the assertion is satisfied. Otherwise the assertion fails.

```
type TimeEvent = {
    id: string;
    time: Date;
    type: "set-attribute" | "receive-command" | "assert";
    payload: SetAttributePayload | ReceiveCommandPayload | AssertPayload;
};
```

Listing 5: A simplified version of the simulator queue event type. In the actual implementation, the type is implemented as a tagged union, so each payload type is conditioned by its respective type value.

The simulator provides an API for creating snapshots of its state and then restoring the state from those snapshots. This is utilized when stepping backwards in time. The snapshot includes both the event queue and the device data.

Each device in the system is uniquely identified by its name. The attributes and commands available on the device are determined by witch device model the device belongs to. If two devices belong to the same device model, they have the same attributes and commands. Of course the actual state of the devices (values of their attributes) can be different.

The device model structure is designed in such a way where it is compatible with the equivalent data type used in the backend of the pocketix system. It includes additional values, specifically needed for use in the simulator. To distinguish between the two, I am calling my device models with my additions the extended device model. The first addition is that of a `deviceClass` attribute, which is of type `string[][]`. The device class information is used to provide a more tailored UI for that particular type of device. An example of a device class is `[["sensor", "temperature"]]`, which identifies this device model as a sensor which measures temperature. The inner string array consists of keywords, which increase in specificity as their index increases. If a device was a sensor, but measured a property which is not known by the system, it could still be classified as a sensor, which would allow the system to display its data differently, compared to for example a button. The outer string array enables each device to have multiple different classes. If for example a device is both a switch and a sensor of some kind.

The other addition concerns the logic for handling received commands and for displaying the commands in the block pallet inside the VPL Editor. The command specification datatype is shown in Listing 6.

```
export type DeviceCommandSpec = {
    name: string;
    arguments: DeviceCommandArgSpec[];
    handler: (deviceName: string, args: DeviceCommandArg[],
              time: Date,
              attributes: DeviceAttributes) => TimeEvent[];
    displayProps: {icon: string;
                   backgroundColor: string,
                   foregroundColor: string};
}
```

Listing 6: Type describing the command specification required for each extended device model.

The attribute `displayProps` controls what colors and what icon is used to represent this particular command in block form in the VPL Editor. The attribute `handler` specifies a function, which will be called when a device command is processed by the simulator. It defines how the device reacts to a given command.

From the signature of the `handler` function, we can see that it takes the name of the device, the current time, the arguments with which the command was called and the current state of this device's attributes as input. Based on these input values, it returns an array of `TimeEvents`, which were described above (see Listing 5). Handlers will typically return events of type `SetAttribute`, with their time set to the same time as when the

command was processed. This way, the changes in the device attributes is immediate – this is ok, since the transmit and processing delay is already built into the `ReceiveCommand` event. If no action is to be taken, the handler can simply return an empty array. If the device responds to a particular command in a more complex way, another `ReceiveCommand` scheduled for later can be returned. This way a device can simulate some command, which would normally trigger a multi step process.

6.4 User Interface

As mentioned in the solution proposal (see Chapter 5), the library Lit was chosen for the user interface development. It provides a reactive framework for building web component based UIs. Web components are especially good for building components which are meant to be used in different frontend applications, utilizing different technologies.

Lit allows for constructing components using a reactive approach, where changes to a component's reactive property or state attributes causes the component to re-render, thus making the UI a function of the application state. Lit components are created as TypeScript classes, with reactive properties marked using decorators. Components have different life-cycle hooks, which allow us to react property changes, setup/tear down event listeners and more.

Component properties can either be reflected in actual HTML element attributes, or they can be hidden, which is necessary especially if a property has a more complex type, which is not serializable to a string. Reflecting property changes to HTML attributes makes development and debugging easier, since one can simply inspect the HTML document using the browser developer tools to see what values are set for which properties.

A common problem in component frameworks is getting data to deeply nested components. If a component needs data which is only available to a component many levels above it, all the components in between would need to accept this data as their property and pass it along the their children, so that it can finally be given to the nested component, which might be the only one which needed this data. This problem is usually referred to as prop-drilling. A common way to prevent this problem is the use of contexts. A context has a provider and consumers. The provider registers the context and sets the current value. Components nested anywhere in the provider can access this context by consuming it, without any elements between the provider and the consumer having to know about the context. Lit allows for a very flexible way to use contexts, since any properties marked as being consumed from a context can also simply be passed in by a parent component as regular properties. I utilize the context approach in many places in the UI components, in order to not clutter the properties of components and to make composition simpler and more flexible.

Communication from child elements to their parents is done via custom JavaScript events. This makes composition very easy, as the event doesn't have to be caught and processed by the immediate parent of the component, but can be listened for by any parent. Events can contain arbitrary payloads, allowing for sending any kind of data.

VPL Editor Modifications

Before starting the development of the Debugger and Simulator user interface, I needed to make some modifications to the VPL Editor developed by Podvojský [31].

Firstly I modified the custom button component utilized throughout the editor in order to make it usable by keyboard control. This makes it simpler to use by experienced users and also more accessible.

Next I added the UI components for displaying error and warning messages on blocks. Each message includes a block path attribute, which is described in more detail in Section 6.2. Using this path, I am able to find which block caused the error or warning and display it appropriately. The number of errors and warnings is displayed as an icon badge in the upper right corner of the given block. An example of a warning badge can be seen in Figure 6.2b. Clicking the badge opens a detail, which includes the actual text of the message.

I also modified the responsive styles of the editor so it looks and operates better when used on a mobile device. A comparison between the original and the modified editor can be seen in Figure 6.2. These modifications include rearranging the block header to make better use of space and to allow the expression field to take up more of the block's width as well as grow in height.

To help users understand the at a glance what variables and device attributes are used in expressions, I added an icon and a colored background to them, in order to distinguish them from other operands. I also reduced the number of parentheses, so that they only show when necessary.

(a) The original VPL Editor, which struggles on small devices, when device names and commands are long.

(b) The modified VPL Editor, with slightly reorganized block layout to better fit on small screens.

Figure 6.2: A comparison between the original VPL Editor (on the left) and the modified version (on the right). Modifications include better small screen layout, removed unnecessary parentheses in expressions, highlighted device and user variables in expression.

Breakpoint markers and controls for managing breakpoints were also added. Breakpoints can be set by clicking the top left corner of any block. Clicking the marker again disables the breakpoint, but keeps it on the block. A disabled breakpoint wont pause the program execution, but it can be more easily reactivated, then if it was completely removed. Removing a breakpoint is done by dragging in to the side. This more involved interaction was chosen to prevent completely removing breakpoints on accident. These controls might be uncomfortable on devices with small screens, so I also added the ability to add, disable

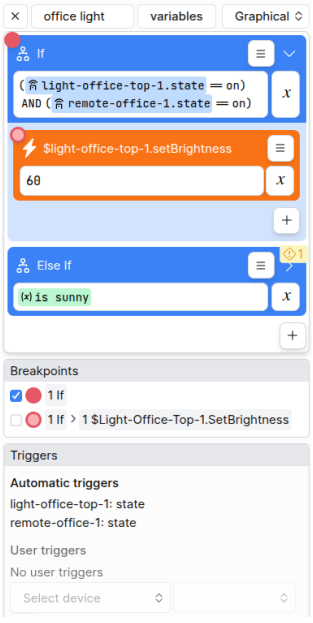
and remove breakpoints to the statement controls menu, which can be opened by clicking the button with three horizontal lines.

Breakpoints are associated with a particular block in the same way as errors and warnings. Each breakpoint has a block path, which uniquely locates the block to which the breakpoint belongs. When there is a change in any breakpoint, the list of breakpoints and their paths is emitted as an event. Components outside of the VPL Editor can then listen to this event and react accordingly.

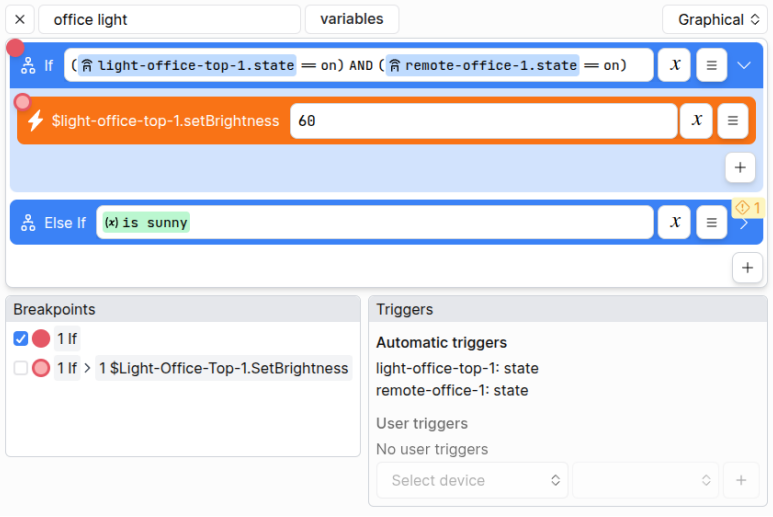
Another required feature which needed to be added is the ability to highlight a particular block. This feature is used to let the user know, which block is about to be executed next. The highlighted block again utilizes a property of type `BlockPath`. If no suitable block for the given path is found, no block is highlighted. When the requested block is found, a dashed outline is drawn around it. It is also scrolled into view if it is not already visible.

Automation Editor

The VPL Editor allows users to create main program of an automation, but they also need to be able to manage other metadata such as its name, breakpoints and triggers. The automation editor can be seen in Figure 6.3. Like all the components and UI layouts developed as part of this work, the automation editor is fully responsive.



(a) Mobile layout of the automation editor



(b) Desktop layout of the automation editor.

Figure 6.3: The automation editor, which includes the VPL Editor, automation name input, list of all breakpoints and a list of automatically extracted triggers and user specified triggers.

Changes to the automation are validated and saved by the `AutomationManager` object. It keeps a list of all available automations in the system and it runs them through the validator when there is a change.

Any changes made in the automation editor are emitted as events, which are then caught in the root component. This component hosts the `AutomationManager` instance, informs

it of any changes received via events and provides a context with a list of all available automation data.

Test Scenario Editor

Test Scenarios are created and edited using the test scenario editor. Just like the automation editor, any changes to a scenario are emitted as events, which are caught in the root component, which forwards the changes to a `ScenarioManager` object. Its function is virtually the same as of the `AutomationManager`, only instead of automation data and automation specific operations, it deals with test scenario data.

The test scenario editor can be seen in Figure 6.4. It is made up of three main areas. The top are displays metadata such as the scenario name, the start time of the simulation, whether or not a specific day should be set for the simulation and the simulated packet (network transmission) delay. In the middle the user can set the initial state of devices in the simulator. Device attributes which are not specified here are set to their default values.

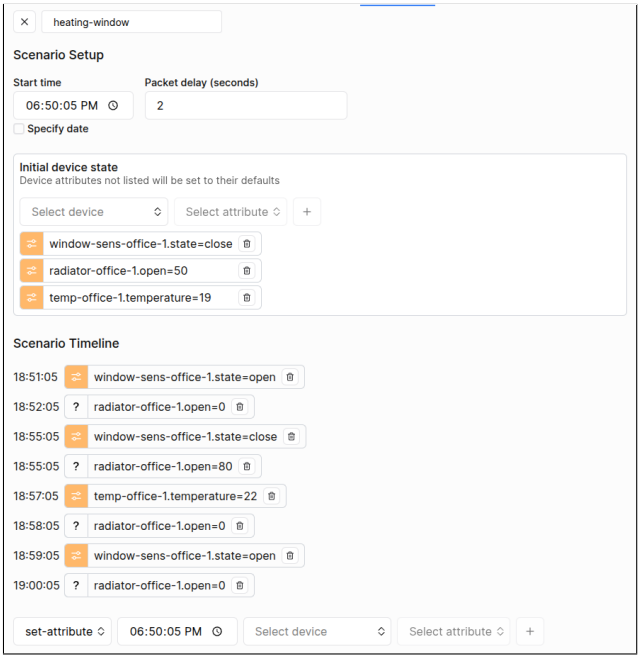


Figure 6.4: Test scenario editor. Scenario metadata such as its name, simulator start time and packet delay is set at the top. The middle section contains device initialization data and the actual scenario timeline is displayed in the bottom portion of the screen. The timeline consists of events scheduled at specific times.

The bottom part of the editor then displays the scenario timeline. At the bottom of the timeline is a form for adding new events to it. The possible event types are `SetAttribute` events and `Assert` events. Their function is described in more detail in Section 6.3.

Data Coordinator

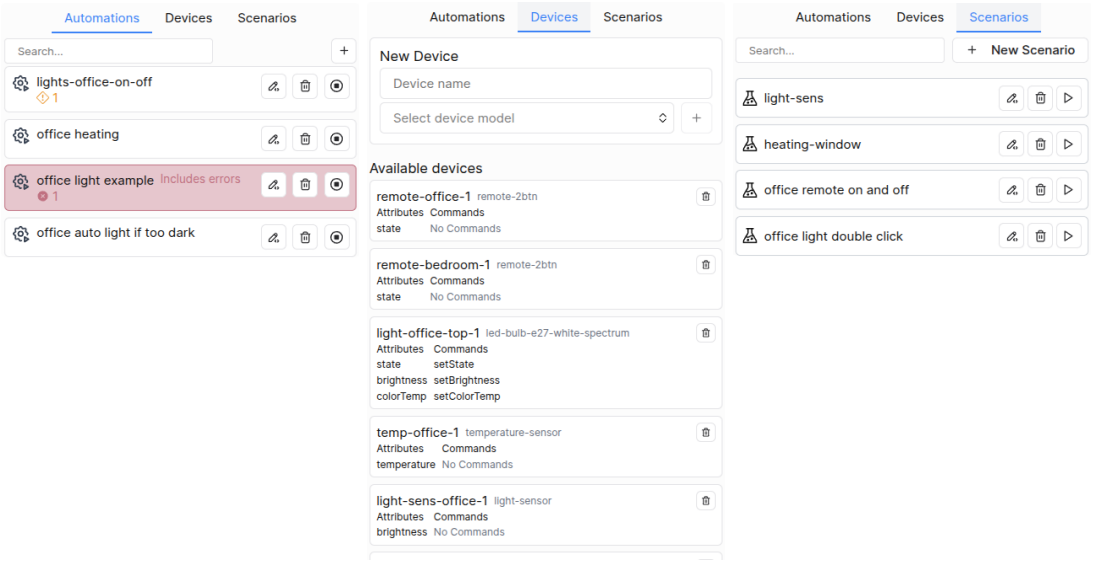
The root component provides all automation, device and scenario data. It setups all necessary contexts and adds event listeners, which listen for events emitted by the editors

discussed in previous subsections. The previously mentioned manager objects are instantiated here.

While idle, it displays a tab component, which provide Automation, Devices and Scenario tabs. These tabs can be seen in Figure 6.5. The automation editor is opened by clicking the pencil icon button of the desired automation in the list. Each automation also has a delete button and a disable button, which will disable the automation and prevent it from running, even if the trigger conditions were met. This way, users can select which automations will be triggered and they can test their interactions in a controlled manner.

The devices tab shows all devices available in the system. Each device has a name and an assigned device model. Each device also shows what attributes and commands it supports. Creating a new device is done in the top part of this tab.

The scenarios tab shows a simple list of all test scenarios available in the system. The scenario editor can be opened by clicking the pencil icon button of the desired scenario. Clicking the play icon button of a given scenario starts this scenario. The root component switches from showing the data tabs to showing the test scenario runner UI, which is described in more detail in the next subsection.



(a) Automation list (b) Device list (c) Scenario list

Figure 6.5: Tabs containing lists of automations, devices and scenarios.

Test Scenario Runner

When a test scenario is started, the test scenario runner screen is shown to the user. This screen, and the set of UI components it is comprised of, was the main focus of this work in terms of UI development. Great attention was given to the responsiveness of this UI, as one of the goals of this work is to make the testing and debugging process manageable from a mobile device. Many users with less technical knowledge expect to use their mobile devices to control their IoT systems.

The desktop view of this screen can be seen in Figure 6.6. The screen is divided into three different parts. On the left the scenario timeline is shown. In the beginning it only shows the timeline, as it was constructed by the user, when creating this test scenario. During the

scenario execution, the timeline gets updated with new events when they happen. When an automation is triggered by a change in a device attribute, it gets added to the timeline. The automation is nested inside of the `SetAttribute` event which triggered it.

If executing the automation results in sending any device commands to devices, the commands are also added to the timeline. To make it clear what automation sent what commands, the command is added to the timeline twice. Firstly it is nested under automation which sent it. Secondly it is added to the appropriate place in the timeline for when the command is scheduled to be executed by the device. Since sending a command to a device is never instantaneous, the simulator models this by scheduling the execution of the command in the future. How far in the future is determined by the 'packet delay' value configured for each particular test scenario. Hovering over a scheduled command will highlight the automation which sent it.

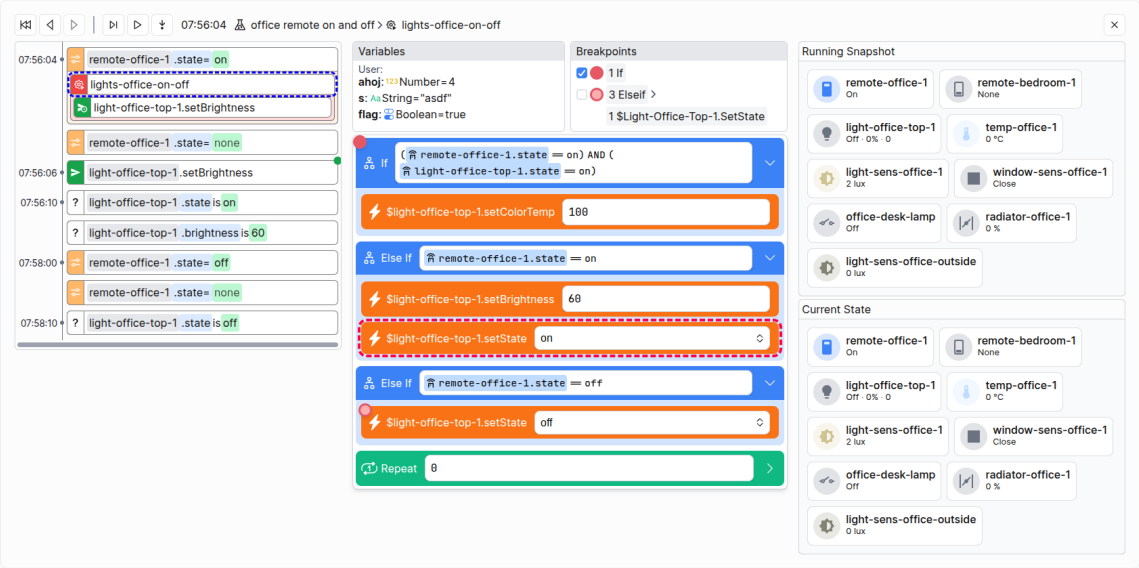


Figure 6.6: Scenario runner UI on a desktop computer. The scenario timeline is shown on the left side of the screen. The right side shows the current device state. The middle portion of the screen shows the currently running automation program.

The middle part of the screen shows the currently running automation. On the top there are two windows showing the current state of user variables in the program and a list of all breakpoints in the program. Below is the VPL Editor showing the automation program. To prevent the user from modifying the program while it is running, the editor is opened in a view only mode, which prevents any program modifications to be made.

On the right the device state is shown. When an automation is running, there are two copies of the device state. The one on the bottom shows the current state, while the one on the top show the state snapshot, which is what the running automation sees. If multiple automations are triggered by the same events, they all run with the same state snapshot.

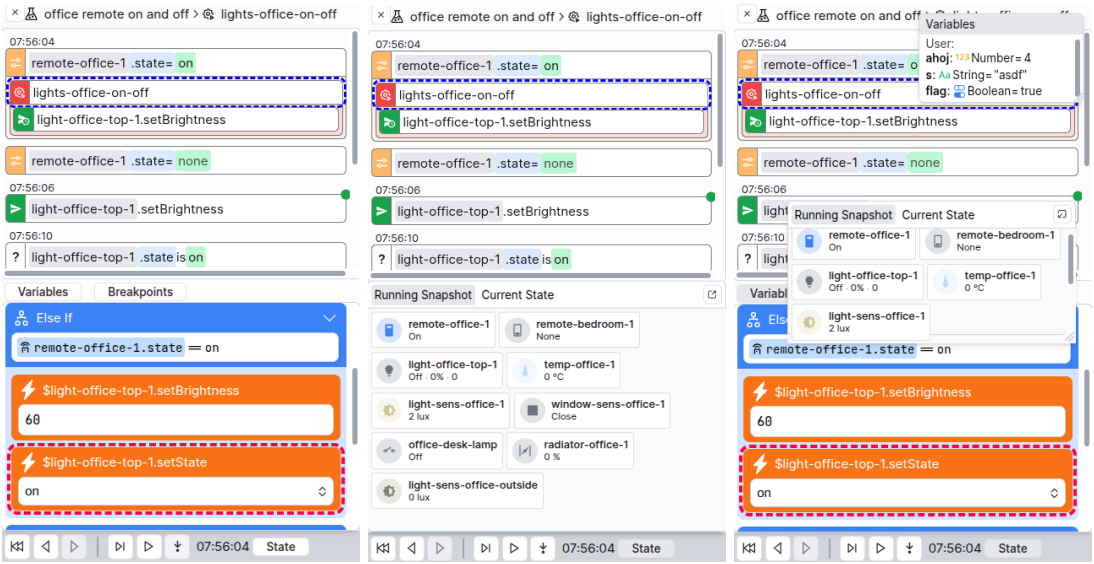
The state display shows each device as a small card, which includes the name of the device, the device icon and the current value of the device attributes. The icon and the layout of the device card is determined by the `DeviceClass` value of each device (see Section 6.3). If the device does not belong to a supported class, a generic card is shown instead.

The toolbar at the top of the screen includes the scenario and debugger controls, the current simulation time and the name of the running scenario and automation. Scenario stepping is controlled by the three left-most buttons. The buttons are (from left to right): step back to the previous top-most event in the timeline, step back to the previous event in the timeline and step forward. The debugger control buttons are (from left to right): execute the entire program, step to the next breakpoint, step to the next statement.

Layout for mobile devices is shown in Figure 6.7. Figure 6.7a shows the default view which is shown when an automation is triggered. The screen is split into two parts. The top part always shows the scenario timeline, which works the same as on larger screens. The bottom part shows the automation, when an automation is running, or the device state if no automation is running.

Since there is limited space on mobile devices, it is difficult to show both the VPL Editor with the running automation and the device state on one screen. For this reason, the device state is hidden by default when an automation is running. If the user wants to look at the device state, it can be overlaid on top of VPL Editor by clicking the 'State' button in the bottom toolbar, as seen in Figure 6.7b. If the user wishes to see both the VPL Editor and the device state at the same time, it is possible to display the device state in a floating window, which can be dragged around the screen and resized. Variable and breakpoint windows are also displayed in floating windows, when the user requests them. This is shown in Figure 6.7c.

The scenario and debugger controls are moved from the top of the screen to the bottom toolbar, to make it more convenient to use on mobile devices. The scenario and automation name is kept at the top of the screen.



(a) Default running automation view

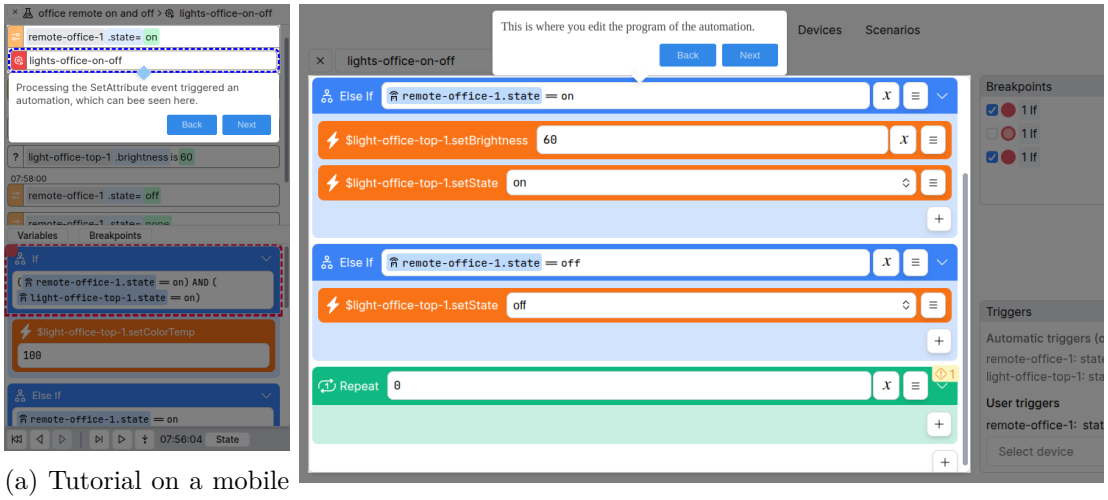
(b) Device state overlaid on top of the running automation

(c) Variable and device state floating windows

Figure 6.7: Tabs containing lists of automations, devices and scenarios.

Interactive Tutorial

In order to make the system more approachable by novice users, I implemented an interactive walkthrough tutorial, which shows the user everything they need to know to start using the application. The tutorial works on both mobile and desktop devices. Examples of the tutorial are shown in Figure 6.8.



(a) Tutorial on a mobile device

(b) Tutorial on a desktop computer

Figure 6.8: Interactive tutorial highlighting a part of the screen, with a message dialog attached to the highlighted element.

To implement the tutorial, I chose the library shepherd.js[8]. It has a very simple, but configurable interface. When defining steps of the tutorial, each step can either be set to display in the middle of the screen, or attached to some element on the page. Since I chose to use web components to write this application, querying for the different elements is slightly more complicated.

Web components are displayed in a shadow root, which isolates them from the rest of the document. This is good for keeping CSS definitions from bleeding through into the components, but it makes selecting elements nested inside shadow roots more difficult. When not using a shadow root, the method `querySelectorAll()` can be used to find any element by its CSS selector. However, this method is not able to penetrate into a shadow root. Therefore a more complicated function was used, which is capable of traversing shadow roots. I came across this function in the work of Podvojský [31], who found the source for this function in a Github Gist[9]. I then modified the function to match the code style of the rest of the project and I added proper TypeScript types. This was enough to adapt the shepherd.js library for use with Lit, since shepherd.js makes it possible to specify the element to which the step should be attached to via a JavaScript function returning an element.

[8] https://www.shepherdjs.dev/
[9] https://gist.github.com/Haprog/848fc451c25da00b540e6d34c301e96a

Chapter 7

Testing

Testing is an integral part of any software project. This work was tested in two parts, depending on the type of testing required. Some parts of the library can be tested by automated tests, which provide some input to the module under test and compare its output with the expected output. When it comes to the user interface, the most type of testing involves actual users, who give their feedback on the intuitiveness and ease of use. Section 7.1 describes what automated testing was performed and Section 7.2 describes the user testing and its results.

7.1 Automated Testing

The most important parts, which need to behave exactly as specified are the VPL parser and interpreter. Since both of them are meant to be run on the client side, they need to maintain compatibility with their backend counterparts. Theoretically, if this library were to be used in an IoT system, whose backend was written in TypeScript, these two components could be reused directly, to avoid duplicating code. This wont be always possible, in which case, there needs to exist a set of tests, which verify that both implementations behave according to the specification.

For this reason, I implemented a suite of parser and interpreter tests, which can be used to ensure compatibility of systems written in different languages. The test consist of JSON files, where one file specifies a single test. Each file includes the test input and the expected output. The test data also specifies, if this test expects the module under test to fail or succeed. It is useful to test for failure, to determine, if the parser can successfully detect malformed programs, or if the interpreter can correctly signal incorrect types of operands and similar runtime errors.

This test data is then loaded by a simple test executor, which simply performs each test as specified, collects all results and returns them. Because the test data is independent of any specific programming language, a similar test executor can be written for any other implementation of the VPL parser and interpreter and the same library of tests can be used to ensure parity between both implementations.

7.2 User Testing

To see how intuitive and usable the implemented user interface is, user testing was performed with users of different technical abilities. Users who took part in the testing can be grouped

into three groups, by their technical skill and domain specific knowledge. I would like to sincerely thank all participants for giving me their time and valuable feedback.

Group one consists of three novice users, who have little to no experience with IoT systems. Each user was first familiarized with general IoT concepts and the system to be tested. Then they were asked to complete three tests, where each one was designed to test the usability of a different part of the system. The first test involved stepping through an existing test scenario and the goal was to determine, if they are able to explain what is happening in the system at each step. All participants were able to correctly explain what is happening on the screen with each step of the simulation. The only part which was not immediately clear to two participants was why commands are not immediately executed, but after explaining the purpose of the packet delay setting, it became clear and they had no issues in explainig it again in future tests.

The second test introduced a faulty automation, which caused unwanted behavior under certain conditions. The participants were first asked, if they can spot any issues with the automation. One was able to correctly identify the condition, which would cause a loop in the automation triggering it self. Two other were able to explain what the purpose of the automation was, but were unable to see any issues. Then they were instructed to use a prepared test scenario, during which the unwanted behavior would be visible. They were asked to step through the test scenario and try to find the unwanted behavior. All three were able to spot the issue. One of the two participants who were not able to see the issue before was able to correctly describe the reason for this behavior and what should be added to the condition to prevent it. The other was able to explain why the behavior occurs in words, but was not able to formulate how the condition should be fixed.

The third and final test asked the participants to construct a test scenario, which would test the provided automation. The function of the provided automation was to turn on the light if a light sensor inside the room reported a brightness below some level and to turn it off, if a light sensor outside the window of that room reported a brightness above some level. All users were able to construct a test scenario, which tested the given automation. A common issue between all three participants was forgetting to set the reported indoor brightness up after setting the outdoor brightness above the "turn off" threshold. This resulted in the simulated light being turned on right after turning off. This is not an issue with the automation, but rather a shortcoming of the simulator. Since it does not know how any devices are related, it can't automatically set the indoor brightness higher, when the outdoor brightness changes. All users were able to identify the mistake in the test scenario. Two were able to find it right away, one was able to identify it after stepping back and forward through the scenario multiple times.

All participants were also asked to explain in their own words what is the purpose of the provided automation before every test. The most common complaints were about the device naming, which would normally not be an issue, if they were using their own IoT system. The second common feedback was that even though devices in the expressions are highlighted, they all look the same, making it difficult to distinguish them at a glance. One participant found it especially difficult to distinguish two devices, which both had a `brightness` attribute, but one was a light sensor, which reported its value in `lux`, while the other was a light bulb, whose brightness was represented as a percentage. Other common feedback includes:

- inability to click on the name of an automation or scenario to open the associated editor

- it should be possible to click on an event in the simulator timeline to jump back to that point in time, instead of having to use the "step back" button
- the result of an `Assert` event could be more pronounced
- ability to see the automation code when constructing a scenario, so it is not necessary to memorize threshold values and similar

The second group of users consists of five university students, who study computer science and have moderate understanding of IoT systems. I demonstrated the system in a group call and then asked them to the go through the first prepared test on their own and give me their feedback as they used the system. One participant mentioned they would prefer to have a single button, which would step through both the simulator and the automation, since now they have to move their mouse between two relatively distant buttons, when they want to quickly step through. Another participant mentioned wanting keyboard support, to make using the system easier and quicker. A common feedback was displaying the command payload in the scheduled command event in the timeline. This was also mentioned by two participants from the first group. All participants were able to use the system on their own after the quick introduction, which was very similar as the introduction available in the interactive tutorial.

The third round of user testing included two IoT industry experts from the company Logimic[1], namely Ing. Michal Valný, Ph.D. and Ing. Ondřej Šulc. Their feedback included adding the ability to simulate that a command from an automation does not reach the target device, limiting the devices displayed to only the ones which would be used in the simulation, as larger systems can have many devices. It was not immediately clear to them, if device state reflects only the confirmed state of the device, or if it also somehow incorporates information about what state the device should be in, based on the sent commands. They also suggested that a system like this could serve as regression tests, where all test scenarios get run to see, if modifying some automation, or adding a new automation did not break any previously working tests. They also suggested running these test automatically, possibly as part of a build pipeline, for which a standardized output of the test results would be useful. They expressed interest about integrating both the modified VPL Editor and the debugger and simulator into their existing system.

7.3 Future Enhancements

Based on the automated testing results, the parser is perfectly capable of being used as a single source of truth for program validation across the Pocketix ecosystem. If it was ever to be extracted into a more slimmed down standalone library, it might be a good idea to consider separating the parsing and evaluation logic. It would mean separated closely related logic into different objects, but it would enable the parser to produce smaller results and make serialization of that result easier, if it was ever needed.

The results of the user testing show, that the system is fairly intuitive and usable even by novice users. Small UI improvements were suggested, which will be simple to implement. Other more involved changes would require a substantial amount of work. These changes would no doubt be useful for making the system more user friendly, but they need to be thoroughly thought out, before being implemented, as to not make the system too complex and impossible to maintain.

[1] Logimic makes bespoke IoT solutions for a wide range of applications. https://www.logimic.com/

Chapter 8

Conclusion

With the rising number of smart devices and the increasing popularity of the Internet of Things across many fields, it is clear that more specialized quality tools are needed to make the use of such devices simpler and more approachable. This is particularly true when it comes to creating and testing user defined programs, which add user-tailored interactions to their smart devices. This master's thesis explores how to design tools, which empower both novice and experienced users to utilize their smart devices to their full potential.

The main goal of this thesis was to design and implement a library that provides an IoT focused debugger and simulator for user defined programs, constructed using a visual programming language (VPL). To achieve this goal, a VPL had to be chosen, based on analysis of existing solutions and uses of VPLs in IoT systems. During the implementation, changes had to be made to the existing editor of this VPL, to add support for new features and to make it better suited for mobile devices.

The entire system was designed in a modular manner, so it can be easily adapted for use with different IoT systems. The implemented VPL parser can also be used as a validator for user programs in other parts of the user interface of systems, which utilize this library. The library also provides visual components, which can be used to construct a rich user interface for the implemented debugger and simulator. Particular focus was given to the responsiveness of the various visual components, to make the entire system usable on a mobile device. An example application was developed, to showcase the implemented library.

The implemented application was tested by a group of users with a wide range of technical skills, as well as industry experts, with an overall positive result. Showing that this design is ready to be integrated into an existing system, where users will be able to utilize it to test their automations and safely explore new possibilities.

Based on user feedback, some modifications and enhancements are proposed. Such as changes to the user interface, to make it more intuitive and easy to use. The proposed changes include merging some buttons used to control the simulator and debugger to streamline the user experience or adding keyboard control. Some more complex changes were proposed, such as making the simulator aware of logical relationships between certain devices and warning the user when it detects that a change to one device should also cause a change in the other device. This would be quite a large undertaking, which would nevertheless make the simulator more user-friendly.

Bibliography

[1] Adragna, P. Software debugging techniques, 2008. Available at: http://cds.cern.ch/record/1100526.

[2] Ahmadzadeh, M.; Elliman, D. and Higgins, C. An analysis of patterns of debugging among novice computer science students. In: *Proceedings of the 10th annual SIGCSE conference on Innovation and technology in computer science education*. Caparica Portugal: ACM, June 2005, p. 84–88. ISBN 9781595930248. Available at: https://dl.acm.org/doi/10.1145/1067445.1067472.

[3] Al Qaseemi, S. A.; Almulhim, H. A.; Almulhim, M. F. and Chaudhry, S. R. IoT architecture challenges and issues: Lack of standardization. In: *2016 Future Technologies Conference (FTC)*. San Francisco, CA, USA: IEEE, 2016, p. 731–738. ISBN 9781509041718. Available at: http://ieeexplore.ieee.org/document/7821686/.

[4] Ashton, K. *That "Internet of Things" Thing* online. June 2009. Available at: https://www.rfidjournal.com/expert-views/that-internet-of-things-thing/73881/. [cit. 2024-12-05].

[5] Bandyopadhyay, D. and Sen, J. Internet of Things: Applications and Challenges in Technology and Standardization. *Wireless Personal Communications*, 2011, vol. 58, no. 1, p. 49–69. Available at: http://link.springer.com/10.1007/s11277-011-0288-5.

[6] Barricelli, B. R.; Cassano, F.; Fogli, D. and Piccinno, A. End-user development, end-user programming and end-user software engineering: A systematic mapping study. *Journal of Systems and Software*, 2019, vol. 149, p. 101–137. Available at: https://linkinghub.elsevier.com/retrieve/pii/S0164121218302577.

[7] Bottcher, A.; Thurner, V.; Schlierkamp, K. and Zehetmeier, D. Debugging students' debugging process. In: *2016 IEEE Frontiers in Education Conference (FIE)*. Erie, PA, USA: IEEE, 2016, p. 1–7. ISBN 9781509017904. Available at: http://ieeexplore.ieee.org/document/7757447/.

[8] Burnett, M. M. Visual programming. *Wiley Encyclopedia of Electrical and Electronics Engineering*, 1999.

[9] Burnett, M. M. and Baker, M. J. A classification system for visual programming languages. *Journal of Visual Languages and Computing*. London; San Diego: Academic Press, c1990-, 1994, vol. 5, no. 3, p. 287–300.

[10] Deiner, A. and Fraser, G. NuzzleBug: Debugging Block-Based Programs in Scratch. In: *Proceedings of the IEEE/ACM 46th International Conference on*

Software Engineering. Lisbon Portugal: ACM, February 2024, p. 1–13. ISBN 9798400702174. Available at: https://dl.acm.org/doi/10.1145/3597503.3623331.

[11] DeLozier, C. and Shey, J. Using Visual Programming Games to Study Novice Programmers. *International Journal of Serious Games*, june 2023, vol. 10, no. 2, p. 115–136. Available at: https://journal.seriousgamessociety.org/index.php/IJSG/article/view/577.

[12] Delsing, J.; Eliasson, J.; Van Deventer, J.; Derhamy, H. and Varga, P. Enabling IoT automation using local clouds. In: *2016 IEEE 3rd World Forum on Internet of Things (WF-IoT)*. Reston, VA, USA: IEEE, 2016, p. 502–507. ISBN 9781509041305. Available at: http://ieeexplore.ieee.org/document/7845474/.

[13] Fitzgerald, S.; Lewandowski, G.; McCauley, R.; Murphy, L.; Simon, B. et al. Debugging: finding, fixing and flailing, a multi-institutional study of novice debuggers. *Computer Science Education*, 2008, vol. 18, no. 2, p. 93–116. Available at: http://www.tandfonline.com/doi/abs/10.1080/08993400802114508.

[14] García, C. G.; Meana Llorián, D.; G Bustelo, C. P. and Cueva Lovelle, J. M. A review about Smart Objects, Sensors, and Actuators. *International Journal of Interactive Multimedia and Artificial Intelligence*, 2017, vol. 4, Special Issue on Advances and Applications in the Internet of Things and Cloud Computing, p. 7–10. Available at: https://www.ijimai.org/journal/bibcite/reference/2560.

[15] Greengard, S. *The internet of things*. Cambridge, Massachusetts: MIT Press, 2015. MIT press essential knowledge series. ISBN 9780262527736 9780262328937.

[16] Gupta, B. and Quamara, M. An overview of Internet of Things (IoT): Architectural aspects, challenges, and protocols. *Concurrency and Computation: Practice and Experience*, november 2020, vol. 32, no. 21, p. e4946. Available at: https://onlinelibrary.wiley.com/doi/10.1002/cpe.4946.

[17] Halpern, M. Computer programming: the debugging epoch opens. *Computers and Automation*, 1965, vol. 14, no. 11, p. 28–39.

[18] Hu, M.; Winikoff, M. and Cranefield, S. Teaching novice programming using goals and plans in a visual notation. In: *IFAC Symposium on Advances in Control Education*. 2012. Available at: https://api.semanticscholar.org/CorpusID:7323544.

[19] Ko, A. J. and Myers, B. A. Designing the whyline: a debugging interface for asking questions about program behavior. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. New York, NY, USA: Association for Computing Machinery, April 2004, p. 151–158. CHI '04. ISBN 9781581137026. Available at: https://dl.acm.org/doi/10.1145/985692.985712.

[20] Kuhail, M. A.; Farooq, S.; Hammad, R. and Bahja, M. Characterizing Visual Programming Approaches for End-User Developers: A Systematic Review. *IEEE Access*, 2021, vol. 9, p. 14181–14202. Available at: https://ieeexplore.ieee.org/document/9320477/.

[21] Li, S.; Xu, L. D. and Zhao, S. The internet of things: a survey. *Information Systems Frontiers*, 2015, vol. 17, no. 2, p. 243–259. Available at: http://link.springer.com/10.1007/s10796-014-9492-7.

[22] Lieberman, H.; Paternò, F.; Klann, M. and Wulf, V. End-user development: An emerging paradigm. In: *End user development*. Springer, 2006, p. 1–8.

[23] Liu, Y.; Qiu, B.; Fan, X.; Zhu, H. and Han, B. Review of Smart Home Energy Management Systems. *Energy Procedia*, 2016, vol. 104, p. 504–508. Available at: https://linkinghub.elsevier.com/retrieve/pii/S1876610216316435.

[24] Merriam Webster (n.d.). *Debug* online. Available at: https://www.merriam-webster.com/dictionary/debug. [cit. 2024-11-08].

[25] Murphy, L.; Lewandowski, G.; McCauley, R.; Simon, B.; Thomas, L. et al. Debugging: the good, the bad, and the quirky – a qualitative analysis of novices' strategies. *ACM SIGCSE Bulletin*, february 2008, vol. 40, no. 1, p. 163–167. Available at: https://dl.acm.org/doi/10.1145/1352322.1352191.

[26] Myers, B. A. Taxonomies of visual programming and program visualization. *Journal of Visual Languages Computing*, 1990, vol. 1, no. 1, p. 97–123. Available at: https://linkinghub.elsevier.com/retrieve/pii/S1045926X05800369.

[27] Navani, D.; Jain, S. and Nehra, M. S. The Internet of Things (IoT): A Study of Architectural Elements. In: *2017 13th International Conference on Signal-Image Technology & Internet-Based Systems (SITIS)*. Jaipur, India: IEEE, 2017, p. 473–478. ISBN 9781538642832. Available at: http://ieeexplore.ieee.org/document/8334789/.

[28] Noone, M. and Mooney, A. Visual and textual programming languages: a systematic review of the literature. *Journal of Computers in Education*, 2018, vol. 5, no. 2, p. 149–174. Available at: http://link.springer.com/10.1007/s40692-018-0101-5.

[29] O'Dell, D. H. The debugging mind-set. *Communications of the ACM*, may 2017, vol. 60, no. 6, p. 40–45. Available at: https://dl.acm.org/doi/10.1145/3052939.

[30] Perscheid, M.; Siegmund, B.; Taeumel, M. and Hirschfeld, R. Studying the advancement in debugging practice of professional software developers. *Software Quality Journal*, 2017, vol. 25, no. 1, p. 83–110. Available at: http://link.springer.com/10.1007/s11219-015-9294-2.

[31] PODVOJSKÝ, L. *Vizuální programování IoT zařízení*. 2024. Dissertation. Brno University of Technology. Faculty of Information Technology.

[32] Qadri, Y. A.; Nauman, A.; Zikria, Y. B.; Vasilakos, A. V. and Kim, S. W. The Future of Healthcare Internet of Things: A Survey of Emerging Technologies. *IEEE Communications Surveys Tutorials*, 2020, vol. 22, no. 2, p. 1121–1167. Available at: https://ieeexplore.ieee.org/document/8993839/.

[33] Qazi, S.; Khawaja, B. A. and Farooq, Q. U. IoT-Equipped and AI-Enabled Next Generation Smart Agriculture: A Critical Review, Current Challenges and Future Trends. *IEEE Access*, 2022, vol. 10, p. 21219–21235. Available at: https://ieeexplore.ieee.org/document/9716089/.

[34] Ray, P. P. A Survey on Visual Programming Languages in Internet of Things. *Scientific Programming*, 2017, vol. 2017, p. 1–6. Available at: https://www.hindawi.com/journals/sp/2017/1231430/.

[35] Rosenberg, J. B. *How debuggers work: algorithms, data structures, and architecture*. USA: John Wiley & Sons, Inc., october 1996. ISBN 9780471149668.

[36] Sanchez, L.; Muñoz, L.; Galache, J. A.; Sotres, P.; Santana, J. R. et al. SmartSantander: IoT experimentation over a smart city testbed. *Computer Networks*, 2014, vol. 61, p. 217–238. Available at: https://linkinghub.elsevier.com/retrieve/pii/S1389128613004337.

[37] SATTERTHWAITE, J. *SOURCE LANGUAGE DEBUGGING TOOLS*. 346 p. Dissertation. ISBN 9781392405192. Available at: https://www.proquest.com/dissertations-theses/source-language-debugging-tools/docview/302762122/se-2. Copyright - Database copyright ProQuest LLC; ProQuest does not claim copyright in the individual underlying works; Last updated - 2023-02-23.

[38] Savidis, A. and Savaki, C. Complete Block-Level Visual Debugger for Blockly. In: Ahram, T.; Karwowski, W.; Pickl, S. and Taiar, R., ed. *Human Systems Engineering and Design II*. Cham: Springer International Publishing, 2020, vol. 1026, p. 286–292. ISBN 9783030279271 9783030279288. Available at: http://link.springer.com/10.1007/978-3-030-27928-8_43.

[39] Savidis, A.; Valsamakis, Y. and Linaritis, D. Simulated IoT Runtime with Virtual Smart Devices: Debugging and Testing End-user Automations:. In: *Proceedings of the 17th International Conference on Web Information Systems and Technologies*. SciTePress, 2021, p. 145–155. ISBN 9789897585364. Available at: https://www.scitepress.org/DigitalLibrary/Link.aspx?doi=10.5220/0010714400003058.

[40] Shu, N. C. Visual programming: Perspectives and approaches. *IBM Systems Journal*, 1999, vol. 38, 2.3, p. 199–221. Available at: http://ieeexplore.ieee.org/document/5387104/.

[41] Sinha, S. *State of IoT 2024: Number of connected IoT devices growing 13September 2024. Available at: https://iot-analytics.com/number-connected-iot-devices/. [cit. 2024-12-02].*

[42] Statista. IoT connections worldwide 2022-2033 *online. Available at: https://www.statista.com/statistics/1183457/iot-connected-devices-worldwide/. [cit. 2024-12-02].*

[43] Whalley, *J.;* Settle, *A. and* Luxton Reilly, *A. A Think-Aloud Study of Novice Debugging.* ACM Trans. Comput. Educ., *june 2023, vol. 23, no. 2, p. 28:1–28:38. Available at: https://dl.acm.org/doi/10.1145/3589004.*

[44] Wu, *M.;* Lu, *T.-J.;* Ling, *F.-Y.;* Sun, *J. and* Du, *H.-Y. Research on the architecture of Internet of Things. In:* 2010 3rd International Conference on Advanced Computer Theory and Engineering(ICACTE). *2010, vol. 5, p. V5–484–V5–487.*

[45] Zeller, *A.* Why programs fail: a guide to systematic debugging. *2nd edth ed. Amsterdam; Boston: Elsevier/Morgan Kaufmann, 2009. ISBN 9780123745156.*

[46] Zhang, *Y.;* Zhang, *K. and* Liu, *G. Static Deadlock Detection for Rust Programs. arXiv, january 2024, arXiv:2401.01114. Available at:* http://arxiv.org/abs/2401.01114. *ArXiv:2401.01114 [cs].*

Appendix A

Contents of the External Attachment

- This document as a PDF file
- text/ – Source code for this document
- implementation/ – Implementation files
  - README.md – File explaining the repository structure and usage
  - demo/ – Files for building a Docker image of the example application
  - src/ – Source files for the entire library
    - debugger/ – Source files for the visual components
    - lib/ – Source files for the parser, interpreter, debugger and simulator