Institut für Parallele und Verteilte Systeme

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

Aggregation von Sensordaten in modellgetriebenen IoT-Anwendungen

Linus Fischer

Studiengang:	Informatik
Prüfer/in:	Prof. Dr. rer. nat. habil. Holger Schwarz
Betreuer/in:	Daniel Del Gaudio, M.Sc.
Beginn am:	1. Oktober 2022
Beendet am:	30. März 2023

Kurzfassung

In vielen Internet-of-Things (IoT) Anwendungen werden Daten von Sensoren generiert und in einem zentralen System verarbeitet, um mit diesen Daten Reaktionen in der realen Welt automatisch anzustoßen. Die Verarbeitung, Aggregation und Filterung der Daten nahe an den Datenquellen durchzuführen spart Zeit und nutzt besser die verfügbaren Ressourcen der IoT-Geräte.

Das IoT Application Modeling Tool (IAMT) ist eine Software, um IoT-Anwendungen zur Verarbeitung von Sensordaten mit einem grafischen Interface zu modellieren und auf IoT-Geräten bereitzustellen. Es wurde an der Universität Stuttgart entwickelt und ist abhängig von der, ebenfalls an der Universität Stuttgart entwickelten Multi-purpose Binding and Provisioning Platform (MBP). Um die im IAMT erstellten Anwendungsmodelle dynamisch ausführen zu können, kann die Messaging Engine (ME2) verwendet werden. Diese wird auf den beteiligten IoT-Geräten installiert und realisiert dann die Anwendungslogik des erstellten Modells. In dieser Arbeit wird die ME2 in ihrer Funktionalität so erweitert, dass es ihr ermöglicht wird, auch komplexere Modelle von IoT-Anwendungen zu realisieren. Zusätzlich werden ausgewählte Aggregationsmethoden implementiert, mit welchen die ME2 Nachrichten in der IoT-Anwendung im Sinne des Complex-Event-Processings verarbeiten kann. Um diese Aggregationsmethoden über das IAMT konfigurierbar zu machen, wird auf Basis von JSON ein Verarbeitungsmodell erstellt. Diese neuen Funktionen werden in einem Anwendungsszenario vorgeführt. Außerdem wird ein Vorschlag und erste Ansätze geliefert, wie sich die ME2 mit ihren erweiterten Funktionen in das IAMT integrieren lässt.

Inhaltsverzeichnis

Abbildungsverzeichnis

Verzeichnis der Listings

Abkürzungsverzeichnis

API Application Programming Interface. 21

CEP Complex Event Processing. 13

EPL Event Processing Language. 47

FIFO First in first out. 27

IAMT IoT Application Modeling Tool. 13

ID Identifier. 20

IoT Internet of Things. 13

IPVS Institut für parallele und verteilte Systeme. 13

JSON JavaScript Object Notation. 14

MBP Multipurpose Binding and Provisioning Platform. 13

ME2 Messaging Engine 2. 13

SSH Secure Shell. 19

1 Einleitung

In den vergangenen Jahren haben Anwendungen des 'Internet of Things' (IoT) zunehmend an Bedeutung erlangt und sind bereits zu einem wichtigen Bestandteil des Alltags vieler Menschen geworden [MLLL+15]. Die Technologie des Internets der Dinge bezieht sich auf die Vernetzung von verschiedensten physischen Objekten. Dabei können Haushaltsgeräte wie Drucker, Lampen, Kaffeemaschinen und Rollläden, aber auch andere Dinge, wie Fahrzeuge, Produktionslinien und Überwachungskameras miteinander verbunden werden. Häufig ist für die Steuerung dieser Geräte kein menschlicher Akteur mehr nötig. Über Sensoren, welche ebenfalls als IoT-Geräte agieren, können Daten in Echtzeit gesammelt und analysiert werden. Aus den Datenströmen dieser Sensoren, ist es möglich, dass die digitalen Geräte selbstständig gewünschte Handlungen ableiten [GBB18; LL12]. Ein viel genutztes Konzept, um solche Datenströme zu verarbeiten ist das Complex Event Processing (CEP) [BD15; CFS+14; RBS21]. Dabei werden den Geräten vordefinierte Muster an Ereignissen oder Messergebnissen gegeben. Diese Muster sollen dann erkannt werden, um ein etwas komplexeres Ereignis feststellen zu können. Treten solche komplexeren Ereignisse auf, so kann eine damit verbundene Reaktion der IoT-Anwendung festgelegt werden. Besonders bei der Verarbeitung von kontinuierlichen, unvorhersagbaren und hochfrequenten Datenströmen kommen die Stärken von CEP zum Vorschein. Darum eignet es sich bestens dazu, Sensordatenströme effektiv zu verarbeiten.

Die Entwicklung von IoT-Anwendungen kann mithilfe von grafisch zu bedienenden Werkzeugen zur Modellierung erheblich vereinfacht werden [HLR17; NTBG15]. Dazu entwickelte das Institut für Parallele und Verteilte Systeme (IPVS) das sogenannte IoT Application Modeling Tool (IAMT), mit dessen Hilfe IoT-Anwendungen zur Verarbeitung von Sensordaten modelliert werden können. Das IAMT ermöglicht es, die einzelnen Operationen und Geräte der IoT-Anwendung als Knoten in einem Graphen zu verknüpfen. Um die verschiedenen Geräte und Operationen verwalten zu können, wird die Multipurpose Binding and Provisioning Platform (MBP) [SHS+20] über ihre REST-Schnittstelle verwendet. Um die modellierten Anwendungen dynamisch ausführen zu können, wurde vom IPVS die Messaging Engine (ME) entwickelt [DH20]. Deren neuere Version (ME2) ermöglicht es, einfache Datenströme auf mehreren Geräten verteilt zu verarbeiten und kann dabei dynamisch neu konfiguriert werden.

Ziel dieser Arbeit ist, Möglichkeiten zu Aggregation von Sensordaten im Sinne des CEP in der ME2 zu implementieren. (Unter dem Begriff der Aggregation werden in dieser Arbeit noch weitere CEP-Methoden als in der Literatur zusammengefasst.) Die dabei ausgewählten Möglichkeiten sollten zuvor, mithilfe von geeigneten Anwendungsfällen und einer Literaturrecherche festgelegt werden. Diese neuen Möglichkeiten sollten dem IAMT in sinnvoller

Weise zur Verfügung gestellt werden, sodass ein Nutzer des IAMT diese einfach bei der Erstellung des Modells verwenden kann. Dabei soll dem Nutzer die Freiheit gewährt werden, die Aggregationen selbst auszuwählen und zu konfigurieren.

Im Kapitel 2 wird eine Einführung in das CEP gegeben und es werden die einzelnen Softwarekomponenten wie die ME2, das IAMT und die MBP einzeln und im Zusammenspiel vorgestellt. Im Kapitel 4 wird darauf eingegangen, wie die ME2 verändert werden muss, um mit deutlich komplexeren Sensordatenströmen umgehen zu können. Es wird außerdem ein Sprachkonzept auf der Basis von JSON[1] entwickelt, mit dessen Hilfe die Aggregationsoperation der ME2 geeignet konfiguriert werden kann. Anschließend wird eine Möglichkeit zur Integration der ME2 in das IAMT vorgeschlagen. Die Implementierung dieser Konzepte wird in Kapitel 5 beschrieben. Dabei wird auch auf die Probleme eingegangen, die bei der Integration der ME2 in das IAMT auftraten. In Kapitel 6 wird ein Anwendungsfall beschrieben, um die Möglichkeiten, welche sich mit der neuen ME2 anbieten, darzustellen. Dieser wird ohne dabei das IAMT zu verwenden durchgeführt und anschließend diskutiert. Zum Ende der Arbeit wird in Kapitel 7 eine Zusammenfassung der Arbeit gegeben und in Kapitel 8 ein Ausblick für mögliche anknüpfende Arbeiten gegeben.

[1] JSON Erläuterung: https://www.w3schools.com/whatis/whatis_json.asp

2 Grundlagen

In diesem Kapitel wird eine kurze Einführung in die Themengebiete gegeben, welche in dieser Arbeit später vorkommen werden. Dazu zählt das Internet of Things im allgemeinen, sowie dessen Nutzung für Sensordatenströme im spezifischen und dessen Entwicklung mithilfe von Modellen. Außerdem werden die Konzepte des Complex-Event-Processings in einer gekürzten Form präsentiert und anschließend werden die einzelnen Software-Tools, welche in dieser Arbeit verwendet werden, vorgestellt.

2.1 Internet of Things

Das Internet der Dinge beschreibt ein modernes Konzept in der Nutzung von elektronischen Geräten. Die zugrunde liegende Idee besteht darin, dass reale physische Objekte wie Sensoren, Computer, Haushaltsgeräte, Produktionsmaschinen und anderes an ein Netzwerk angeschlossen werden. Dabei können sie miteinander interagierten und kooperieren, um gemeinsame Ziele zu erreichen. Sie können Daten austauschen und sammeln. Über das Netzwerk lassen sie sich unter Umständen von Menschen auch fernsteuern. Grundsätzlich aber spielt bei dieser Kommunikation der Mensch oft keine große Rolle mehr. Es handelt sich um eine Kommunikation von Maschine zu Maschine. Dies kann zu einer verbesserten Effizienz bei der Erreichung der angestrebten Ziele führen. Das Internet der Dinge hat schon jetzt einen großen Einfluss im Leben vieler Personen. Ein großer Anwendungsaspekt dieses Konzepts befindet sich im häuslichen Umfeld. Durch Smarthomes lassen sich viele Alltagsprobleme voll automatisiert von verschiedenen IoT-Geräten lösen, oder die Bedienung wird erheblich erleichtert. Aber auch im in der Wirtschaft gibt es viele Anwendungsbereiche für IoT. Ein großer Aspekt ist die Automatisierung von Produktionslinien in der Industrie. Aber auch in der Logistik und Management und dem intelligenten Waren-Transport oder Personentransport wird die IoT einen großen Einfluss haben [AIM10; GBB18].

2.2 Modellgetriebene IoT-Anwendungen

Dieser Abschnitt beschreibt die Motivation für modellgetriebene IoT-Anwendungen und deren zugrundeliegendes Konzept. Durch die immer steigende Zahl an IoT-Geräten, welche an das Internet und andere Netzwerke angeschlossen werden, wirft sich immer mehr die Frage auf,

wie eine solche Menge an Geräten verwaltet werden kann. Dabei steht nicht nur im Fokus, wie ein System von IoT-Geräten entwickelt werden kann, sondern vermehrt auch, wie mit sich ständig ändernden Umgebungen und Geräten umgegangen werden soll. Ein IoT-System sollte fähig sein, sich an hinzukommende und wegfallende Knoten und Änderungen in der Netzwerktopologie schnell anpassen zu können, ohne dass komplizierte Änderungen der Entwickler vorgenommen werden müssen. [HLR17] Mithilfe von Modellen, soll es ermöglicht werden, rasche Anpassungen an einem IoT-Netzwerk vorzunehmen. Hierbei wird auf höheren Abstraktionsebenen festgelegt, was die vorhandenen Ressourcen sind, welche Ziele man erreichen will oder auch welche Operationen in welcher Reihenfolge durchgeführt werden sollen, und welche Einschränkungen die einzelnen Geräte haben. Da es auf dieser Abstraktionsebene nicht mehr eine Rolle spielt, wie ein Ziel erreicht wird, sondern nur noch, dass es erreicht wird, wird die Komplexität stark verringert. Entscheidend ist dabei, wie ein Modell eines IoT Netzwerks auf die tatsächliche Welt abgebildet werden kann. [CPD+16]

2.3 Sensordatenströme

Sensordatenströme sind ein wichtiger Teilbereich der IoT. Schließt man Sensoren an ein Netzwerk an, so liefern diese oft einen kontinuierlichen Strom an Messdaten. Die einzelnen Messungen können unter Umständen hochfrequent sein und müssen in Echtzeit verarbeitet werden. Dabei kann, bei konventionellen Technologien, eine hohe Netzwerk- und Geräteauslastung entstehen. Als Beispiele für solche Sensordaten lassen sich Umweltdaten, Wetterdaten, Positionsdaten, Messungen in Versuchslaboren und Echtzeitdaten in Produktionslinien nennen. Liefert beispielsweise ein Sensor 20 Messdaten pro Sekunde und sendet diese in einer nur 0,5 Kilobyte großen Nachricht, so entsteht innerhalb von einem Tag eine Datenmenge von etwa 800 MB. Bei vielen Sensor wird schnell klar, dass eine solche Datenmenge nicht dauerhaft gespeichert werden kann. Stattdessen müssen die Daten in Echtzeit ausgewertet werden. Nur die daraus gewonnenen relevanten Daten sollen dann weiterverarbeitet werden. [KL09; Nit15] Um diesen kontinuierlichen Strom an Messergebnissen, welcher oft nur aus einzelnen Werten und dem Zeitpunkt der Messung besteht, auswerten zu können, wird ein Datenstrom-Management-System benötigt. Im folgenden Abschnitt wird Complex Event Processing als ein solches System näher erläutert.

2.4 Complex Event Processing

Dieser Abschnitt stammt fast ausschließlich aus dem Buch Complex Event Processing [BD15] von Prof. Dr. Ralf Bruns und Prof. Dr. Jürgen Dunkel. Die angegebenen Seitenzahlen beziehen sich immer auf dieses Buch.

Eine Methode um effektiv Datenströme dynamisch in Echtzeit dynamisch zu analysieren ist das Complex Event Processing [Luc02]. Obwohl CEP ein breites Anwendungsspektrum für

die Verarbeitung von Datenströmen hat, wird in dieser Arbeit das CEP nur im Hinblick auf die Verarbeitung von Sensordatenströmen betrachtet. Bei der Verarbeitung von Sensordatenströmen ist davon auszugehen, dass die einzelnen Sensordaten von geringer Komplexität und feingranular sind. Allerdings ist der produzierte Datenstrom kontinuierlich und teilweise hochfrequent. Da die Datenströme auch nicht zeitlich begrenzt sind, ist es unmöglich, mit einem begrenzen Speicher alle Daten zu speichern. [BD15](S.2) Jede einzelne Messung eines Sensors stellt ein eigenständiges Ereignis (Event) [Glo11] dar. Die Messung führt unmittelbar zu einer generierten Nachricht, welche im restlichen Teil der Arbeit auch synonym als **Ereignis** bezeichnet wird.

2.4.1 Sliding Windows

Da wir, wie eben genannt, nicht in der Lage sind alle Ereignisse in einem kontinuierlichen Strom an Ereignissen dauerhaft zu speichern, benötigen wir eine Möglichkeit um dennoch die relevanten Ereignisse zu speichern. Um dies zu erreichen, wird die vereinfachte Annahme getroffen, dass nur die aktuellsten Ereignisse eine hohe Relevanz besitzen. Es sollen also immer die aktuellsten Ereignisse gespeichert werden und alle Ereignisse, welche durch ihr Alter an Relevanz verloren haben, verworfen werden. Um diese Grenzen zu definieren, stehen zwei verschiedene Möglichkeiten zur Verfügung. Bei Längenfenstern wird im Voraus die maximale Anzahl an Ereignissen schon klar definiert. Es können also, bei einem Längenfenster der Länge n immer nur die letzten n Ereignisse gespeichert werden und alle vorherigen werden verworfen. Bei Zeitfenstern ist die maximale Anzahl an Ereignissen nicht definiert, aber dafür wird genau bestimmt, ab welchem Alter Nachrichten verworfen werden sollen [BD15](S.19).

2.4.2 Unterscheidung der Wertigkeit von Ereignissen

Ein einzelnes Ereignis genügt nur selten, um daraus eine sinnvolle Handlung ableiten zu können. Vielmehr muss ein Ereignis in seinem Kontext betrachtet werden. Dabei kann man es in Beziehung zu Ereignissen, welche einen anderen Ursprung haben, oder man vergleicht es mit den vorhergegangenen Ereignissen desselben Ursprungs. Erst durch diese Beziehungen kann ein wirklicher Informationswert abgeleitet werden. Findet man durch das Untersuchen der Beziehung einen Zusammenhang oder ein Muster, so nennt man diesen Zusammenhang nun ein höherwertiges, komplexes Ereignis. Während die vorherigen niederwertigeren Ereignissen Vorkommnissen in der realen Welt entsprachen, entspricht das komplexe Ereignis einem imaginären Ereignis, welches eine Bedeutung für die Anwendung hat. Dieser Prozess, welcher zu einer Generierung von höherwertigen Ereignissen führt, kann auch mehrfach auf den neu generierten Ereignissen ausgeführt werden und führt zu immer höherwertigeren und komplexeren Ereignissen. Es wird also eine Art von Abstraktionshierarchie gebildet.[BD15](S.5)

2.4.3 Architekturansatz zur Verarbeitung von Sensordatenströmen

Da für das Auftreten der Ereignisse sich nicht bereits im Voraus ein Ablauf definieren lässt, muss die Architektur der Software zur Verarbeitung des Datenstroms unabhängig von festen Abläufen sein, sondern stattdessen Ereignisorientiert (Event-Driven) sein [BD15](S.7). Der Grundzyklus beim CEP besteht aus 3 Schritten: Dem Erkennen, dem Verarbeiten und dem Reagieren. Beim Schritt Erkennen wird hier eine Messung bzw. eine Nachricht eines Sensors erzeugt. Dieses Ereignis wird im Schritt Verarbeiten weiter zu höherwertigen Ereignissen verarbeitet. Dabei wird es zusammen mit anderen Ereignissen aggregiert, abstrahiert und korreliert und in manchen Fällen verworfen. Im letzten Schritt können aufgrund von erzeugten komplexen Ereignissen verschiedene Handlungen ausgelöst werden. Wie zum Beispiel das Auslösen von Alarmen, das Informieren von Personal oder auslösen eines Prozesses. [BD15](S.6)

Die Nachricht zu einem Ereignis muss immer einige Metadaten enthalten, welche bei der Verarbeitung von Bedeutung sind. Dazu gehört ein Zeitstempel, eine ID, der Ereignistyp, die Ereignis-Quelle und natürlich auch die Nutzdaten des Sensors [BD15](S.10). Der **Ereignistyp** beschreibt, von welcher Klasse ein Ereignis ist und legt damit dessen Struktur der Attribute fest. Messungen desselben Sensors sind also immer vom selben Ereignistyp und werden Ereignisinstanzen genannt.

2.4.4 Verarbeitung über Ereignisregeln

Für die Verarbeitung von Ereignissen werden spezielle Ereignisregeln definiert. Diese Regeln setzen sich aus zwei Teilen zusammen. Dem Bedingungsteil und dem Aktionsteil. Im Bedingungsteil wird die Erkennung von gewissen Mustern im Ereignisstrom vorausgesetzt. Es können auch mehrere solche Muster logisch über boolesche Operatoren wie 'und', 'oder' und 'nicht' verknüpft werden (Ereignisalgebra). Wird diese Bedingung erfüllt, so wird die Aktion im Aktionsteil ausgeführt. Der Aktionsteil besteht aus den Reaktionen, welche beim Erkennen eines gesuchten Musters erfolgen sollen. Meisten ist es das Erzeugen eines neuen Ereignisses oder das Anstoßen eines Dienstes eines Akteurs. Mit diesen Regeln, welche in der CEP Komponente ausgeführt werden, können die Ereignisse übersetzt, gefiltert, zu komplexeren Ereignissen aggregiert, angereichert und synthetisiert werden.

- Bei der **Translation** wird der Ereignistyp geändert, aber die Daten werden behalten [BD15](S.27).
- Entfernt man alle Ereignisse aus dem Ereignisstrom und lässt nur solche passieren, welche als relevant betrachtet werden, so nennt man dies **Filterung**. Dies ist äußerst wichtig, um Effizienzsteigerungen zu ermöglichen [BD15](S.28).
- Unter der **Aggregation** wird in diesem Kontext das Zusammenfassen von mehreren korrelierten Ereignissen verstanden. Dabei kommen mathematische Funktionen wie der Durchschnitt oder die Summe zum Einsatz [BD15](S.29). Im Rahmen dieser Arbeit wird der Begriff Aggregation allerdings viel allgemeiner verwendet. Er bezieht sich auf die

gesamte Idee, Ereignisse zusammenzuführen, damit nur relevante Ereignisse weitergeleitet werden. Diese Benennung wird auch beibehalten, selbst wenn die Aggregation so konfiguriert wird, dass gar keine Zusammenfassung stattfindet.

- Wenn für eine Ereignisregel Informationen benötigt werden, welche nicht aus den Ereignissen selbst abgeleitet werden können, so benötigt man eine **Anreicherung mit Kontextwissen**. Dieses Wissen könnte z. B. aus dem Internet abgefragt werden [BD15](S.29).
- Bei der **Synthese komplexer Ereignisse** wird eine Bedeutung aus mehreren einfacheren Ereignissen abgeleitet, welche für das Anwendungssystem relevant sein könnte. Es wird also ein neues Ereignis erzeugt, sobald ein kausaler oder semantischer Zusammenhang zwischen mehreren Ereignissen festgestellt werden kann [BD15](S.30).

Für die Notation dieser Ereignisregeln wird im hierfür referenzierten Buch [BD15] eine eigene Syntax verwendet. Es existieren allerdings noch weitere Event-Processing-Languages wie zum Beispiel die Esper-Query-Language, welche von der Open-Source-CEP-Engine Esper verwendet wird. Deren Syntax ist sehr ähnlich zu den bekannten SQL-Datenbank abfragen.

2.5 IoT Application Modeling Tool

Das vom IPVS entwickelte IoT Application Modeling Tool ermöglicht es einfach IoT-Anwendungen zu erstellen, ohne, dass ein Nutzer viel Vorwissen benötigt. Speziell wird es ermöglicht, IoT-Anwendungen zur Verarbeitung von Sensordaten zu erstellen. Dazu wird ein grafisches Interface wie in Abbildung 6.4 dargestellt angeboten. Der dort dargestellte Graph modelliert die Abfolge an Operationen, welche ausgeführt werden sollen, um einen Sensordatenstrom zu verarbeiten. Dieser Graph kann beliebig vom Nutzer konfiguriert werden. Nachdem dieses Modell erstellt wurde, können die einzelnen Operationen tatsächlichen IoT-Geräten zugewiesen werden. Dies wird in Abbildung 6.5 dargestellt. Da das IAMT stark von der MBP abhängig ist, wird in den folgenden Abschnitten erst auf die anderen nötigen Softwarekomponenten eingegangen, ehe das IAMT näher erläutert wird.

2.5.1 Multipurpose Binding and Provisioning Platform

Die MBP ist eine IoT-Plattform, welche eine große Auswahl verschiedener und damit verbundener Dienste anbietet. Generell kann man mit ihr IoT-Anwendungen erstellen und überwachen. Verschiedene IoT-Geräte lassen sich an die MBP anbinden. Dazu wird eine Secure-Shell-Verbindung (SSH) hergestellt. Um auf diesen Geräten Anwendungen zu installieren und zu verwalten, werden diese Anwendungen in Operatoren verpackt. Diese Operatoren können auf der MBP registriert werden. Sie müssen dabei jeweils eine Reihe von Shell-Skripten anbieten, mit deren Hilfe, die MBP die Anwendungen installieren, starten, überwachen und stoppen

kann. Wie genau diese Operatoren aussehen, wird in späteren Kapiteln noch genauer betrachtet. Zusätzlich bietet die MBP Dashboards an, mit denen denen sich gemessene Daten der IoT-Geräte geeignet visualisieren lassen. Auch Regeln aus dem Bereich der CEP lassen sich auch auf der MBP erstellen definieren. Diese Regeln können abhängig von eingegangenen Sensordaten Aktionen auslösen [Sch] [SHS+20].

2.5.2 Messaging Engine 2

Die Messaging Engine 2 (ME2) ist eine Python-basierte Software, die auf verschiedenen IoT Geräten installiert werden kann, um diese zu steuern und miteinander zu verknüpfen. Sie kann auf so konfiguriert werden, dass sie eine Auswahl vorgegebener Sensoren und Aktuatoren ansteuern kann. *Der Begriff 'Aktuator' wird in dieser Arbeit dazu verwendet, etwas zu beschreiben, das in der realen Welt Veränderungen herbeiführen kann. Damit wird zum Beispiel eine Lampe oder ein Buzzer gemeint.* Eine große Stärke der ME2 ist es, dass sie dynamisch, zur Laufzeit auf ein verändertes Modell der IoT-Umgebung angepasst werden kann.

Über die Konfigurationsdatei `config.yaml` kann der ME2 mitgeteilt werden, an welche Sensoren und Aktuatoren sie angeschlossen wird. Dazu steht eine Menge an Sensortypen und Aktuatortypen zur Auswahl, welche bereits vorimplementiert wurden. Dazu muss lediglich angegeben werden, an welchen Pin (z. B. bei einem Raspberry-Pi) diese angeschlossen werden und in welchem Zeitintervall eine Messung durchgeführt werden soll.

Um auf die Verarbeitung von bestimmten Sensordatenströmen (Flow) angepasst zu werden, bietet die ME2 eine Konfiguration zur Laufzeit über eine JSON-Datei an. Diese flow.json-Datei kann über eine Netzwerkschnittstelle empfangen werden. In dieser Datei werden alle Operationen des Flows in ihrer Reihenfolge angegeben. Zu jeder Operation wird dabei eine ID mitgegeben, welche angibt, welche ME2 diese Operation durchführen soll. Die Operationen sind alle Teil einer vorimplementierten Menge an Operationen, welche aus Sensormessungen, dem Aktivieren von Aktuatoren und Zwischenoperationen wie z. B. Logging bestehen. Dabei steht immer zu Beginn eines Flows eine Sensormessung und zum Ende eines Flows das Aktivieren eines Aktuators.

Wird eine Sensormessung getätigt, so startet die ME2 die erste Operation des Flows. Diese generiert eine Nachricht, welche den Messwert und einige Metadaten enthält und sendet diese an die ME2, welche die im Flow darauffolgende Operation ausführen soll. Dabei kann es sich auch um die eigene ME2 handeln. Bei Empfang einer solchen Nachricht führt die ME2 die entsprechende Operation aus und sendet, falls es sich nicht um die letzte Operation des Flows handelt, eine Nachricht an die darauffolgende Operation aus. Dies wiederholt sich so lange, bis die letzte Operation eines Flows erreicht wird, welche einen Aktivator auslösen sollte.

Will man das Verhalten der verschiedenen ME2s zur Laufzeit ändern, so kann man eine veränderte flow.json Datei an alle ME2s senden. Diese passen sich daraufhin dynamisch an die neuen Anforderungen an.

Im Abschnitt 4.1 von Kapitel 4 wird noch genauer darauf eingegangen, wie ein solcher Flow aussehen kann.

2.5.3 Integration der ME2 und der MBP

Um das Zusammenspiel der verschiedenen Komponenten zu verdeutlichen, wird in Abbildung 2.1 ein Diagramm gegeben. Das IAMT ist für die Ausführung sehr abhängig von der MBP. Alle verfügbaren Geräte und Operatoren werden mithilfe der MBP erstellt, konfiguriert und verwaltet. Damit das IAMT darauf Zugriff hat, muss man sich mit seinem Benutzeraccount der verwendeten MBP-Instanz beim IAMT anmelden. Daraufhin kann man ein neues Projekt erstellen, oder ein altes Projekt laden. Das IAMT fragt über die API der MBP nun alle verfügbaren Operatoren ab und erfasst alle Geräte, welche bei der MBP registriert wurden (1). Mit den geladenen Operatoren kann ein Modell erstellt werden. Dazu wählt man die gewünschten Operatoren aus, konfiguriert diese mit den angegebenen Parametern und platziert sie an den gewünschten Stellen im Graphen (2). Nachdem man das Modell im Frontend des IAMT erstellt hat, kann man dieses auf den verfügbaren Geräten deployen (3). Dazu wird der MBP mitgeteilt, auf welchen Geräten welcher Operator installiert werden soll (4). Das Deployment wird dann von der MBP mittels der vorgegebenen Installations- und Start-Scripten des Operators vorgenommen (6). Die Kommunikation mit den einzelnen Operatoren wird dabei auch von der MBP übernommen.

Um die ME2 in diesem Szenario verwenden zu können, muss sie als Operator auf der MBP registriert werden (5). Sie kann dann auf allen benötigten Geräten mithilfe der MBP installiert und verwaltet werden. Damit sie allerdings die gewünschte Funktionalität erfüllen können, muss das IAMT eine Flow-Datei erstellen, welche aus dem Modell abgeleitet werden kann. Diese Flow-Datei sollte daraufhin an alle ME2s gesendet werden. Zusätzlich muss jeder ME2-Instanz die IP-Adressen der Nachbarn mitgeteilt werden, damit diese direkt kommunizieren können (7). Diese Funktionalität ist allerdings noch nicht vollständig vorhanden. Darum wird darauf erst im Kapitel 4 weiter eingegangen.

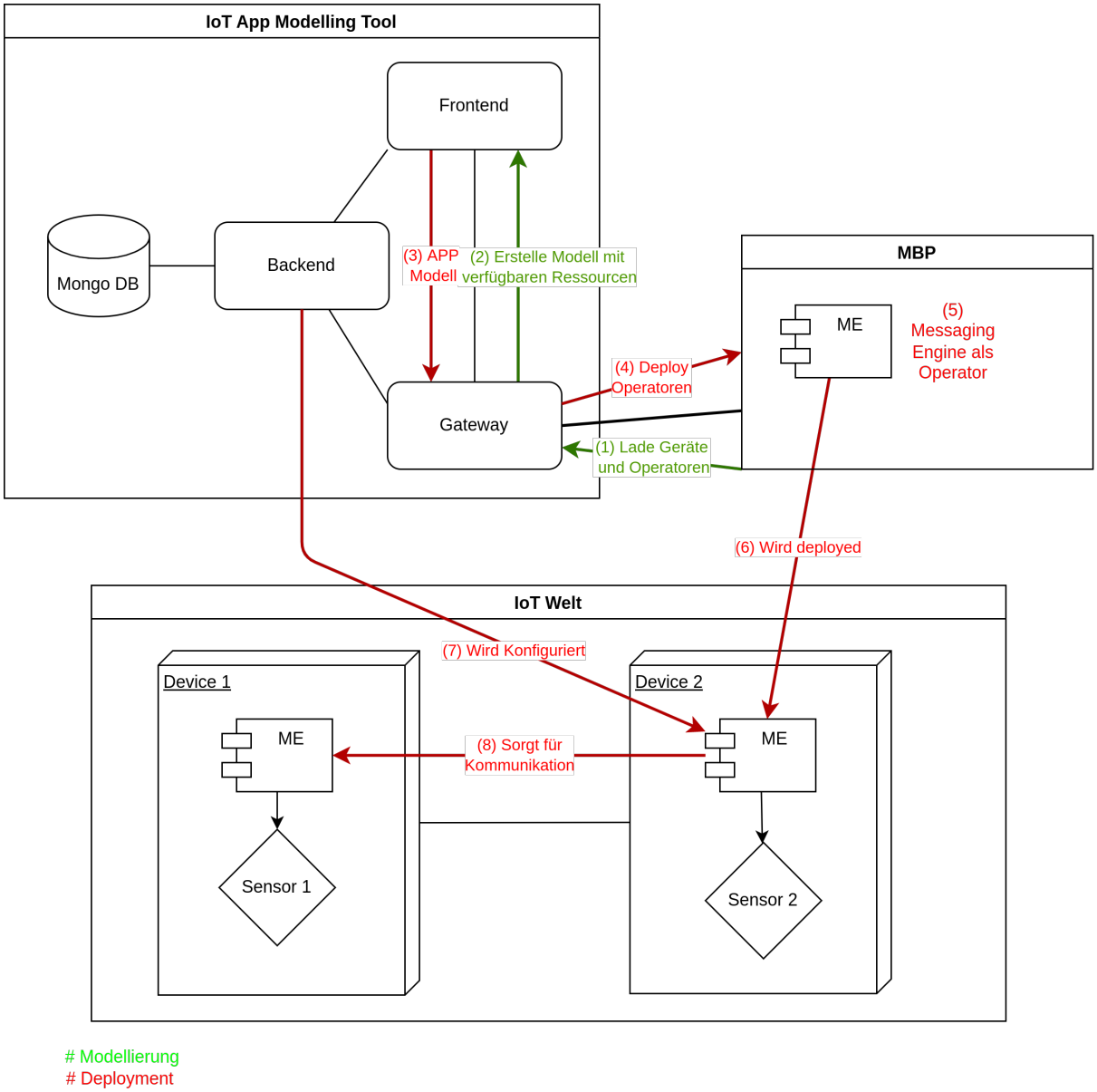

Abbildung 2.1: Diagramm zur Verdeutlichung, wie das IAMT, die MBP und die ME2 im Zusammenspiel funktionieren. Die grünen Pfeile sind dazu da, um das Modell zu erstellen. Die roten Pfeile verdeutlichen den Ablauf, wie das Modell auf den verschiedenen IoT-Geräten deployt werden kann.

3 Verwandte Arbeiten

In diesem Kapitel werden drei Paper vorgestellt, welche ähnliche Zielsetzungen wie diese Arbeit haben. Nachdem kurz zusammengefasst wird, worum es in diesem Paper geht, wird erklärt, was die wesentlichen Unterschiede in Abgrenzung zu dieser Arbeit sind.

3.1 P4CEP: Die Nutzung von CEP für Berechnungen in einem Netzwerk

Um sogenannte 'In-Network' Berechnungen durchzuführen, also die Berechnungen nicht mehr auf einem Server, sondern auf Hardware, welche Teil eines Netzwerkes ist, durchzuführen, gibt es eine 'Data-plane'-Programmiersprache namens P4. In dem Paper **P4CEP: Towards In-Network Complex Event Processing** [KMD+18] wird CEP als eine Möglichkeit vorgeschlagen, um nötige Berechnungen schon direkt auf den Kommunikationspfaden durchzuführen. Um zu demonstrieren, dass CEP hierbei die Latenzzeiten verringert und man auf geeignete Weise die verfügbare Hardware nutzen kann, wurde eine Sprache für Ereignisregeln P4CEP entwickelt. Mithilfe eines endlichen Automaten sollen so die in der Ereignisregel definierten Ereignismuster erkannt werden. In einem Proof-of-Concept wird gezeigt, dass selbst die noch unoptimierte Implementierung einer CEP-Engine schon sehr gute Performanz liefert.

Auch wenn in dieser Arbeit ein sehr ähnliches Ziel erreicht werden sollte und ebenfalls eine Sprache für Ereignisregeln entwickelt wurde, gibt es dennoch zu dieser Bachelorarbeit. Der Schwerpunkt der Implementierung liegt in dem Paper darauf, Ereignismuster zu erkennen, also ob in einer gewissen Abfolge Ereignisinstanzen von bestimmten Ereignistypen auftreten. Für die Implementierung der ME2 liegt der Schwerpunkt allerdings an anderer Stelle. Da die ME2 Sensordatenströme verarbeiten soll und alle Sensoren in regelmäßigen Abständen Messungen durchführen, ist aus der Reihenfolge der auftretenden Ereignisse kaum eine verwertbare Information zu gewinnen. Vielmehr müssen die einzelnen Messwerte mit mathematischen Operationen verarbeitet werden.

3.2 Eine verteilte CEP-Engine für IoT-Anwendungen

Bereits 2014 erschien ein Paper mit dem Ziel eine verteilte CEP-Engine zu entwerfen, welche im Bereich des IoT eingesetzt werden kann [CFS+14]. Dabei wurden die Aufgaben zur Verarbeitung

der Nachrichten aus der IoT-Anwendung in zwei Bereiche aufgeteilt. Ein Teil soll die Daten zuerst aufbereiten und der zweite Teil soll daraus Handlungen oder verwertbare Daten ableiten. Diese Aufgaben werden als separate Komponenten betrachtet, die auf verschiedenen Servern laufen können. Die verschiedenen CEP-Engines können dabei von mehreren Nutzern immer neu konfiguriert werden. Um dem Nutzer das Verändern der Ereignisregeln möglichst einfach zu machen, wird ein Web-Interface angeboten, mit dessen Hilfe es ermöglicht wird, per 'Drag-and-drop' Regeln neu anzuordnen.

Dieses Konzept ist in vielerlei Hinsicht sehr ähnlich zu der Zielsetzung dieser Arbeit, da auch ein einfaches Web-Interface angeboten wird, um dadurch Regeln zu definieren, wie mit Sensordaten umgegangen werden soll. Der große Unterschied dabei ist allerdings, dass die einzelnen CEP-Engines nur auf verteilten Servern laufen, aber nicht auf den IoT-Geräten selbst, welche die Sensordaten selbst generieren, was den Vorteil hätte, dass die Sensordaten frühzeitig gefiltert werden, wodurch sich die Netzwerkauslastung verringern würde.

3.3 Query-Shipping mithilfe der MBP in verteilten IoT-Anwendungen

Um die Netzwerkauslastung bei der Nutzung von CEP im Bereich des IoT zu reduzieren, wird im Paper **An Approach for CEP Query Shipping to Support Distributed IoT Environments** [FHPM18] eine verteilte Datenverarbeitung vorgeschlagen. Dazu sollen die einzelnen Ereignisregeln des CEPs nicht erst auf einem Server verarbeitet werden, sondern auf den jeweiligen IoT Geräten, damit die Verarbeitung so nah wie möglich an der Datenquelle stattfinden kann. Dazu soll zu jeder CEP-Query die benötigte Hardware-Leistung zu deren Ausführung notiert werden. Zu jedem verfügbaren Gerät werden die verfügbaren Leistungsressourcen notiert. Durch das Zusammenführen dieser Informationen kann der ideale Ausführungsort bestimmt werden. Das Konzept, wie die CEP-Queries auf die verschiedenen IoT-Geräte verfrachtet werden, gliedert sich dabei in drei verschiedene Schichten. In der Modellierungsschicht werden die Anforderungen und Ressourcen definiert. Mithilfe der MBP wird die zweite Schicht 'Shipping' realisiert. Diese kombiniert das erstellte Modell mit der realen Welt der IoT-Geräte. Die letzte Schicht 'Hardware' repräsentiert die physische Welt der IoT-Geräte.

Das Paper überschneidet sich in vielen Bereichen mit dieser Arbeit. Man kann diese Arbeit als eine Fortsetzung der dort entwickelten Konzepte betrachten.

Der Gebrauch der ME2, um die Ereignisregeln der CEP auszuführen, ermöglicht es, das Modell sehr dynamisch über eine festgelegte Schnittstelle anzupassen. Die ME2 liefert einige ausgewählte vorgefertigte Funktionen, welche den Konfigurationsaufwand deutlich reduzieren können. Des Weiteren bietet das Zusammenspiel mit dem IAMT eine sehr nutzerfreundliche Benutzeroberfläche.

4 Konzept

Ziel dieser Arbeit ist es, geeignete Methoden zur Aggregation von Sensordaten auf Basis der Messaging Engine zu implementieren. Diese Methoden sollen, auf praktische Weise in das IAMT integriert werden, sodass eine Konfiguration der ME2 von dort aus ermöglicht wird. Dabei sollte der Anwendungsmodellierer bei der Konfiguration des Modells, insbesondere der Aggregation, auf geeignete Weise einbezogen werden.

In diesem Kapitel wird das entwickelte Konzept zum Erreichen dieser Ziele beschrieben. Dabei wird in Abschnitt 4.1 darauf eingegangen, wie der Umgang der ME2 mit Sensordatenströmen (Flows) angepasst werden muss, damit sie nun auch komplexere Konstruktionen, auf welchen Aggregationen durchgeführt werden können, ermöglicht. In Abschnitt 4.2 wird dabei speziell darauf eingegangen, wie die ME2 mithilfe einer modifizierten Flow-Datei in JSON-Format so konfiguriert werden kann, dass man sie auf die erforderten Flows anpassen kann. Zum Schluss wird noch in Abschnitt 4.3 diskutiert, wie die veränderte ME2 in das IAMT integriert werden kann, sodass der Anwender möglichst einfach IoT-Systeme modellieren kann und die erstellten Modelle dann verarbeitet werden können.

4.1 Veränderung der Flows

In diesem Abschnitt wird diskutiert, wie die Struktur des Flows welcher von den verschiedenen ME2s ausgeführt werden kann, angepasst werden muss, damit es ermöglicht wird auch komplexere IoT-Systeme zu erzeugen.

4.1.1 Bisherige Struktur des Flows

Um einen Sensordatenstrom zu erzeugen, welcher es ermöglicht, die gemessenen Daten der verschiedenen Sensoren in Beziehung zu setzen, benötigen wir einen anderen Sensordatenstrom (Flow) als den bisherigen Flow der ME2. Bisher lief der Flow, wie diesem Abschnitt beschrieben wird, ab.

```
{
  "name": "test_flow",
  "ops": [
    {
      "name": "sensor_read",
      "input": "sensor_hall",
      "where": "me2_first"
    },
    {
      "name": "filter",
      "process": "filter_intercept",
      "where": "me2_second"
    },
    {
      "name": "log",
      "process": "log",
      "where": "me2_second"
    },
    {
      "name": "beep_call",
      "output": "actuator_speaker",
      "where": "me2_third"
    }
  ]
}
```

Listing 1: Beispiel eines Flows, wie er vor dieser Arbeit war. Dabei werden die Operationen in einer Liste in ihrer Reihenfolge angegeben. Zu jeder Operation wird der Name der Operation, von welchem Typ diese Operation ist und auf welchem Gerät diese ausgeführt werden darf.

Ein IoT-Gerät empfängt ein Sensor-Messergebnis. Dieses Gerät sucht, ob zu diesem Messergebnis ein Flow, beziehungsweise eine Datei, welche diesen Flow beschreibt, existiert, der als erste Operation einen passenden `Sensor-Input` hat (In dem beispielhaften Flow von Listing 1 ist in der ersten Operation 'sensor_read' der `Sensor-Input` als 'sensor_hall' gewählt). Wenn dies der Fall ist, so wird anschließend im Flow überprüft, ob diese Operation auch auf diesem Gerät, welches das Messergebnis empfing, durchgeführt werden darf. Man überprüft dies, indem man schaut, ob der String im Parameter 'where' der `InstanceID` des eigenen Geräts übereinstimmt. Diese ID wird in der Konfigurationsdatei zu jeder ME2 angegeben. Sind alle Voraussetzungen erfüllt, wird dieser Flow gestartet. Da die erste Operation, also das Einlesen der Sensordaten, bereits erfüllt wurde, wird eine Nachricht für die zweite Operation in diesem Flow erstellt. Die Flows weisen eine lineare Struktur auf, wie in Abbildung 4.1 zu sehen ist, womit klar definiert ist, was die darauf folgende Operation ist. Ist die `InstanceID` der nächsten Operation die eigene ID, so wird die Nachricht Geräte-intern weitergeleitet. Ist die `InstanceID` der nächsten Operation jedoch die ID eines anderen Geräts, so wird unter den bekannten

Nachbarn geschaut, ob ein Nachbar eben jene `InstanceID` besitzt. Zu diesem Gerät wird die Nachricht dann weitergeleitet. Eine Nachricht wird immer zuerst in die `PushQueue` geschoben und anschließend verschickt. Die `PushQueue` ist dabei nur eine FIFO-Queue[1], welche die Nachrichten puffert um ein asynchrones Versenden zu ermöglichen. Existiert kein Gerät, zu dem die Nachricht geschickt werden kann, so tritt ein Fehler auf und der Flow wird an dieser Stelle dann beendet.

Wird eine Nachricht empfangen, unabhängig davon, ob es eine interne Nachricht ist oder ob sie von einem anderen Gerät empfangen wurde, so wird diese Nachricht in der `Pull Queue`, welche ebenfalls nur zum Puffern der Nachrichten dient, eingereiht. Diese Nachricht wird dann, sofern die Kapazitäten vorhanden sind, entnommen und verarbeitet. Zuerst wird dabei erfasst, zu welchem Flow die Nachricht gehört. Dies kann mit der `FlowID` und dem `FlowName` geschehen, welcher in der Nachricht enthalten ist. Anschließend wird bestimmt, welche der Operation des Flows als Nächstes ausgeführt werden soll. Dies kann über das Attribut `LastOperation` bestimmt werden, indem man diese Operation im Flow sucht und dann die darauf folgende Operation auswählt. Zusätzlich wird der Nachricht auch der `payload` entnommen. Dieser enthält die tatsächlichen Nutzdaten, also die gemessenen Sensorwerte. Nun wird überprüft, ob die eigenen `InstanceID` zu der für die Operation angegebenen `InstanceID` passt und entsprechend die Operation mit den Nutzdaten ausgeführt oder entsprechend verworfen. Zu jeder Operation können noch optional im Flow Argumente hinzugefügt werden, welche bei der Ausführung beachtet werden müssen.

Nach Beendigung der Operation wird, wie schon im ersten Schritt, eine Nachricht verfasst, welche zum Gerät der nächsten Operation geschickt wird. Dieses Verfahren wird so lange fortgesetzt, bis alle Zwischen-Operationen ausgeführt wurden und die letzte Operation des Flows erreicht wurde. Die letzte Operation sollte sinngemäß eine Aktivator-Operation sein. Also eine Operation, die etwas in der realen Welt bewirkt. Dies kann ein LED- oder Buzzer-Signal sein, ein Schalter oder auch eine Nachricht, die an ein externes Tool geschickt wird. Ist diese Operation beendet, wird auch der laufende Flow beendet.

4.1.2 Zielsetzung für die neue Struktur der Flows

Ziel dieser Arbeit sollte es sein, nun die Daten mehrerer Sensoren, von eventuell unterschiedlichen Geräten, in Beziehung zueinander setzen zu können und so komplexere, zusammenhängende Ereignisse verarbeiten zu können. Intuitiv wäre es wünschenswert, sämtliche Operationen, in beliebiger Anordnung, miteinander verbinden zu können und so jegliche gewünschte Konfigurationen erreichen zu können. Die Topografie würde sich also von einem linearen Strang hin zu einem verzweigten Netzwerk ändern. Ein Knoten könnte nun also von mehreren Operationen Nachrichten empfangen und selbst auch an eventuell mehrere Operationen

[1]Queue Datenstrukturen: https://www.geeksforgeeks.org/queue-data-structure/

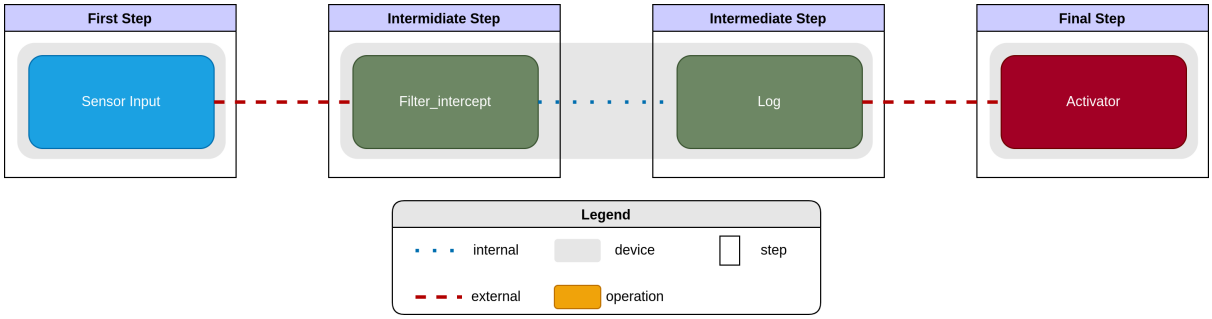

Abbildung 4.1: Beispiel-Diagramm für den alten Flow aus Listing 1. Dabei werden die Operationen farbig in ihrem jeweiligen Step angegeben, um die Reihenfolge zu verdeutlichen. Mit den grauen Kästen werden die verschiedenen Geräte dargestellt, auf denen die Operationen ausgeführt werden. Die Verbindungen zwischen den Operationen stellen dar, ob die Nachrichten Geräte-intern oder extern verschickt werden.

senden. Im Folgenden wird ein Ansatz vorgestellt, welcher diese Ziele adressieren soll, aber auch einige, der damit einhergehenden, Schwierigkeiten umgehen soll.

4.1.3 Vermeidbare Probleme

In diesem Abschnitt werden einige Probleme aufgezählt, welche beim Konzipieren beachtet werden müssen.

- Schlechter Überblick des Ablaufs:

  Läuft ein Sensordatenstrom nicht mehr von einer Informationsquelle zu einem Informationsausgang, sondern kann viele Verzweigungen unterwegs nehmen, so ist schon bei kleineren Flows der Ablauf für Menschen schwer nachzuvollziehen.

- Auftreten von Zyklen innerhalb des Graphen:

  Ein unstrukturierter Graph ermöglicht es, Zyklen innerhalb des Flows einzubauen. Eine solche Konstruktion sollte zwar im Normalfall dem Anwender auffallen, ist allerdings ein Problem, welches beachtet werden muss.

- Technische Umsetzung:

  In der gegebenen Version der ME2 ist schon vollständig vorgegeben, wie eine Nachricht den gegebenen Flow traversiert. Ein Abwenden von dieser Architektur hätte viele weitreichenden Folgen für die Gesamtanwendung der ME2.

4.1.4 Umsetzung

In der Implementierung werden die Schritte des Flows 'Step' genannt, weshalb hier diese Benennung übernommen wird. In diesem Abschnitt soll nun ein mögliches Konzept vorgeschlagen werden, welches möglichst alle Anforderungen erfüllt und dabei die eben genannten Schwierigkeiten vermeidet.

Steps mit mehreren Operationen

In der neuen Architektur der Flows bleiben die einzelnen Steps erhalten. Der Flow läuft also immer noch vom ersten Step bis zum letzten Step. Allerdings sind dabei in jedem Step mehr als eine Operation möglich. Die einzelnen Steps sind also als Menge definiert. Eine Nachricht darf nur von dem eigenen Step zu Operationen des darauf folgenden Steps übermittelt werden. Sie darf also weder an einen vorherigen Step noch an eine Operation im eigenen Step gesendet werden. Auch darf kein Step übersprungen werden. Dieser Aufbau garantiert einen übersichtlichen Ablauf des Sensordatenstroms, da sofort erkennbar ist, in welche Richtung die Nachrichten gesendet werden. Dem Entstehen von Zyklen ist damit auch automatisch vorgebeugt. Es ist außerdem einfach in der Implementierung herauszufinden, an welcher Stelle im Flow man sich gerade befindet.

Trotz dieser Einschränkungen ist es weiterhin möglich, alle vorstellbaren, sinnvollen Konfigurationen eines Flows zu erreichen. Ein Ausnahmefall muss allerdings berücksichtigt werden. Hat man zum Beispiel die Sensoren S1, S2, die Zwischen-Operationen O1, O2 und den Aktivator A1 und möchte folgendes Netzwerk bauen: `A1 -> O1, A2 -> O1, O1 -> O2, A2 -> O2, O2 -> A1`. So fällt auf, dass A2 die Nachricht an O1, als auch an die Operation O2 schicken will, welche allerdings ein Step hinter O1 angesiedelt sein muss. Dies ist mit der Einschränkung, dass man Nachrichten an den direkt darauf folgenden Step schicken muss, nicht möglich. Um diesen Fall zu adressieren, wird eine Operation 'pass' eingeführt, welche die Nachricht einfach an den nächsten Step weiterleitet. Ein solches Umgehen der eigenen Regeln erscheint auf den ersten Blick etwas widersprüchlich. Es ergibt allerdings aus programmiertechnischer Sicht durchaus Sinn. So muss beispielsweise zum Senden einer Nachricht nicht der gesamte Flow ab dem jetzigen Step durchsucht werden. Stattdessen reicht es nur die Operationen des direkt darauffolgenden Steps zu überprüfen. Dies kann zu einer verbesserten Performance führen, da davon auszugehen ist, dass von der Operation 'pass' nur selten Gebrauch zu machen ist, aber die Flows recht groß werden können.

4.1.5 Verknüpfen von Sender und Empfänger

Um zu bestimmen, welche Operation an welche nachfolgende Operationen Nachrichten sendet, wurde ein Konzept ähnlich dem Publish-Subscribe Konzept gewählt [Tar12]. Dabei gibt jede

Operation an, welche Nachricht von ihr gesendet wird und welche Nachrichten sie von vorhergehenden Operationen empfangen möchte. Mittels der Parameter `give` und `take` können die Namen der Nachrichten angegeben werden, welche man empfangen oder senden kann. Die ME2 registriert dann beim Senden einer Nachricht, wie diese Nachricht genannt wird und ob sich im nächsten Step eine oder mehrere Operationen befinden, welche diese Nachricht empfangen wollen. Dadurch muss, wenn man an mehrere Nachfolger senden möchte, dennoch nur einmal die Nachricht benannt werden. Wenn man von mehreren Vorgängern Nachrichten empfangen möchte, ist es möglich, als `take` mehrere Nachrichten mit einem Komma separiert anzugeben. Mehrere Nachrichten als `give` ist nicht möglich, da dies auch wenig Sinn ergibt. Das erzeugte Ergebnis einer Operation ist immer vom selben Nachrichten-Typ. Will man dennoch verschiedene Nachrichten erzeugen, kann man einfach eine weitere Operation, welche den alternativen Nachrichten-Typ erzeugt, selben Step einfügen.

Dieser Ansatz ermöglicht es, einzelne Operation wie unabhängige Komponenten zu betrachten und sie beliebig im Flow anzuordnen und wiederzuverwenden. Fügt man beispielsweise weitere Operationen in den Flow ein, so muss keine der anderen Operationen angepasst werden, da die Verknüpfungen von Nachrichten alleine auf den `give` und `take` Parametern basiert. Für den Anwender ist es bei einer sinnvollen Benennung der Nachrichten schnell klar, was die einzelnen Nachrichten bedeuten.

Ein Nachteil, der dadurch entsteht, ist, dass die Verbindung doppelt geschrieben werden muss. Also sowohl als `give` des Senders als auch als `take` des Empfängers. Ein alternativer Ansatz wäre, direkt bei der sendenden Operation zu bestimmen, an welche nachfolgende Operation gesendet werden soll. Dadurch verlieren die einzelnen Operationen aber an Unabhängigkeit und der Flow wird schlechter verständlich und wartbar.

Um zu erkennen, woher eine Nachricht kommt, also welche Operation sie als letztes behandelt hat und wie sie verarbeitet werden soll, wurde der Nachricht, welche in Listing 2 zu sehen ist, ein weiteres Attribut `name` hinzugefügt. Der `name` entspricht immer dem `give` der sendenden Operation.

In der aktuellen Implementierung der ME2 wird an jede Nachricht ein kompletter Verlauf dieser Nachricht angeheftet. Dadurch lässt sich mithilfe des `name` Attributs beim Untersuchen des Verlaufs leicht nachvollziehen, welchen Pfad die Nachricht durch das Netzwerk an Operationen genommen hat.

Um die Übersichtlichkeit des Flows weiter zu erhöhen, wurde noch zusätzlich die Regel eingeführt, dass alle Sensoren im ersten Step aufgelistet sein sollen. Auch müssen alle Aktuatoren im letzten Step angesiedelt sein. Dies führt zu keiner Einschränkung der Möglichkeiten, kann allerdings die Anzahl der benötigten Pass-Operationen erhöhen.

Um diese neue Version des Flows geeignet zu visualisieren, wird in Abbildung 4.2 ein Beispiel des Flows als Diagramm gegeben.

```
{
  "id": "message_uuid",
  "name": "name_of_the_message",
  "flow_name": "flow_name",
  "flow_id": "flow_id",
  "src_created_at": "datetime",
  "last_operation": "last_op_name",
  "sent_at": "datetime",
  "payload": "payload_base64",
  "original_payload": "payload_base64",
  "history": FlowMessage[]
}
```

Listing 2: Veranschaulichung einer Flow-Message. Dabei ist auch das neue Attribut 'name' zu sehen. Der Payload wird hierbei zusätzlich im Base-64 Format codiert.

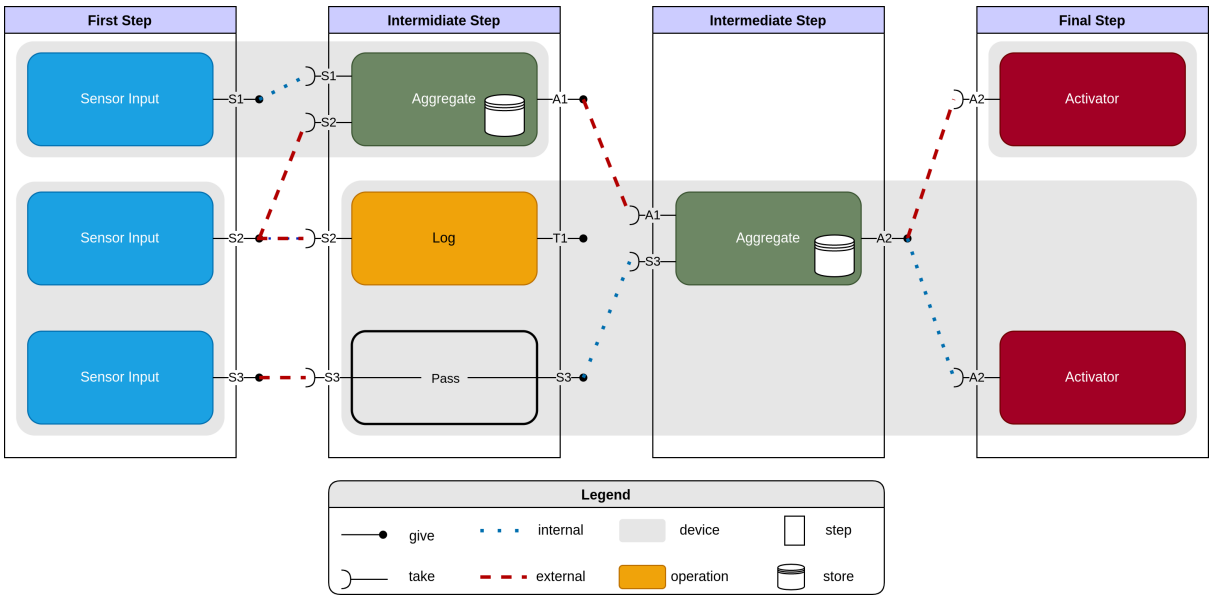

Abbildung 4.2: Beispiel-Diagramm für den neuen Flow. Dabei können mehrere Operationen in einem Step sein. Jede Operation kann Nachrichten anbieten und aufnehmen. Die ME2 erkennt dadurch, an welche Operationen eine Operation senden möchte. Dies wird mit den gestrichelten Linien verdeutlicht. Die grauen Kästen verdeutlichen auf welchem Gerät eine Operation ausgeführt wird.

4.1.6 Nachteile dieses Konzepts

Das gewählte Konzept zur Verarbeitung des Flows bringt natürlich nicht nur Vorteile. Hier werden einige der Nachteile genannt.

- Kein Überspringen von Steps:

  Wenn in einem Flow eine Nachricht erst in einem Step, welcher etwas später im Flow vorkommt, verarbeitet werden soll, so muss diese Nachricht erst durch mehrere Pass-Operationen geleitet werden. Dies kann zu unnötigem Rechenaufwand führen.

- Automatisierung erübrigt die Übersichtlichkeit:

  Die verbessere Verständlichkeit des Flows wird, in einem automatisierten System keine Vorteile mehr erbringen. Wenn man den Flow später nur noch über das Frontend des IAMT konfiguriert, so ist es für den Anwender belanglos, ob der Flow übersichtlich zu lesen ist.

- Vergrößerte Nachrichten:

  Jede Nachricht, die von einer ME gesendet wird, hat nun noch einen etwas vergrößerten Header. Dieser Unterschied ist allerdings verschwindend gering im Vergleich zur Größe der Nachrichten-'History'.

- Doppelte Benennung einer Verbindung:

  Soll Operation A an Operation B eine Nachricht senden, so muss der Name der Nachricht sowohl im Parameter `give` der Operation A als auch im Parameter `take` der Operation B vorkommen.

Bei dem gewählten Konzept fällt auf, dass an vielen Stellen ein leicht ineffizienterer Ansatz als theoretisch möglich gewählt worden ist. Diese Nachteile wurden stets in Kauf genommen, da sie dazu führten, dass aus Sicht des Anwenders und Programmierers ein wesentlich übersichtlicheres, intuitiveres, wartbareres und wiederverwendbareres Nutzen möglich ist. Aus dem Code der ME geht hervor, dass genau dies der Design-Philosophie entspricht. An vielen Stellen ist bei Durchlesen des Codes aufgefallen, dass die ME möglichst verständlich sein soll und dafür auch kleinere Nachteile in Kauf genommen wurden. Beispielsweise wurden interne Nachrichten nicht direkt der nächsten Operation übergeben, sondern über die Push-Queue in die Pull-Queue als vollständige Nachricht übermittelt. Dies wäre in der Theorie wesentlich effizienter möglich, indem der Payload der Nachricht direkt der nächsten Operation übergeben worden wäre.

4.2 Struktur der Flow JSON

In diesem Abschnitt wird nun darauf eingegangen, wie die `flow.json`-Datei aufgebaut sein soll, damit dazu verwendet werden kann, die ME2 so zu konfigurieren, dass sie die den gewünschten Flow mit den dazugehörigen Aggregationsoperationen ausführen kann.

Insbesondere wird darauf eingegangen, wie die ME2 mithilfe einer modifizierten Flow-Datei in JSON-Format so konfiguriert werden kann, dass man sie auf die erforderten Flows anpassen kann

Hinweis: Im Folgenden wird der Begriff 'Nachricht' als Synonym für den Begriff 'Event' verwendet, da er in diesem Szenario die gleiche Bedeutung auf einer anderen Abstraktionsebene hat.

Im Abschnitt der Grundlagen wurde bereits auf Sprachkonzepte zur Ereignisverarbeitung eingegangen. Das Ziel der dort aufgeführten Sprachkonzepte war es, mittels menschlich verständlichem Text der Maschine mitzuteilen, was sie tun soll bzw. wie die Aggregation ablaufen soll. Dies ist ein ähnliches Ziel wie bei gängigen Programmiersprachen. Diese natürliche Sprache muss immer zuerst in für Maschinen verständlichen Code umgewandelt werden. Dabei wird mittels eines Parsers zuerst die Struktur der Eingabe ermittelt. Es wird dabei zwischen Variablen, Operatoren, Verzweigungen, Datentypen und anderem unterschieden. Oft wird die abstrahierte syntaktische Struktur mit einem Abstract-Syntax-Tree dargestellt. Mittels dieser Abstraktion kann nun entweder Maschinencode erzeugt werden oder Befehle für die zur Sprache zugehörigen Laufzeitumgebung generiert werden. In dieser Arbeit soll mithilfe des Frontends des IAMT die Aggregation der ME konfiguriert werden. Wie die Aggregation ablaufen soll, wird der ME2 also nicht von einem Menschen mitgeteilt, sondern direkt von einer anderen Maschine. Wir benötigen also ein Sprachkonstrukt, was hauptsächlich von Maschine zu Maschine verwendet wird, allerdings für Menschen dennoch nachvollziehbar sein soll. Um dieses Ziel zu erreichen, wird ein Sprachkonzept verwendet, welches bereits auf einer anderen syntaktischen Abstraktionsebene liegt. Das Sprachkonzept kann damit ohne viele Zwischenschritte von der ME2 verarbeitet werden.

Da der Flow bereits im JSON-Format geschrieben ist und die Aggregationsoperation auch darin spezifiziert wird, eignet es sich ideal, das Sprachkonzept auch im JSON-Format anzuwenden. Dabei wird die gewünschte Ereignisregel im Parameter 'args' einer Aggregationsoperation übergeben.

Die Ereignisregeln, welche im Buch zu CEP [BD15] erläutert werden, gliedern sich in einen Bedingungsteil und einen Aktionsteil. Diese Aufteilung wird hier übernommen, wobei die Aggregationsfenster zusätzlich aus dem Bedingungsteil herausgelöst werden. Der Aktionsteil wird dahingehend vereinfacht, dass als einzige Aktion 'Create', also das Erschaffen einer neuen Nachricht zur Verfügung steht.

In der JSON-Datei könnte dies dann wie in Listing 3 aussehen. Hierbei ist erkennbar, dass die Steps als Menge an Operationen angegeben werden und der Flow als Menge an Steps. Auch

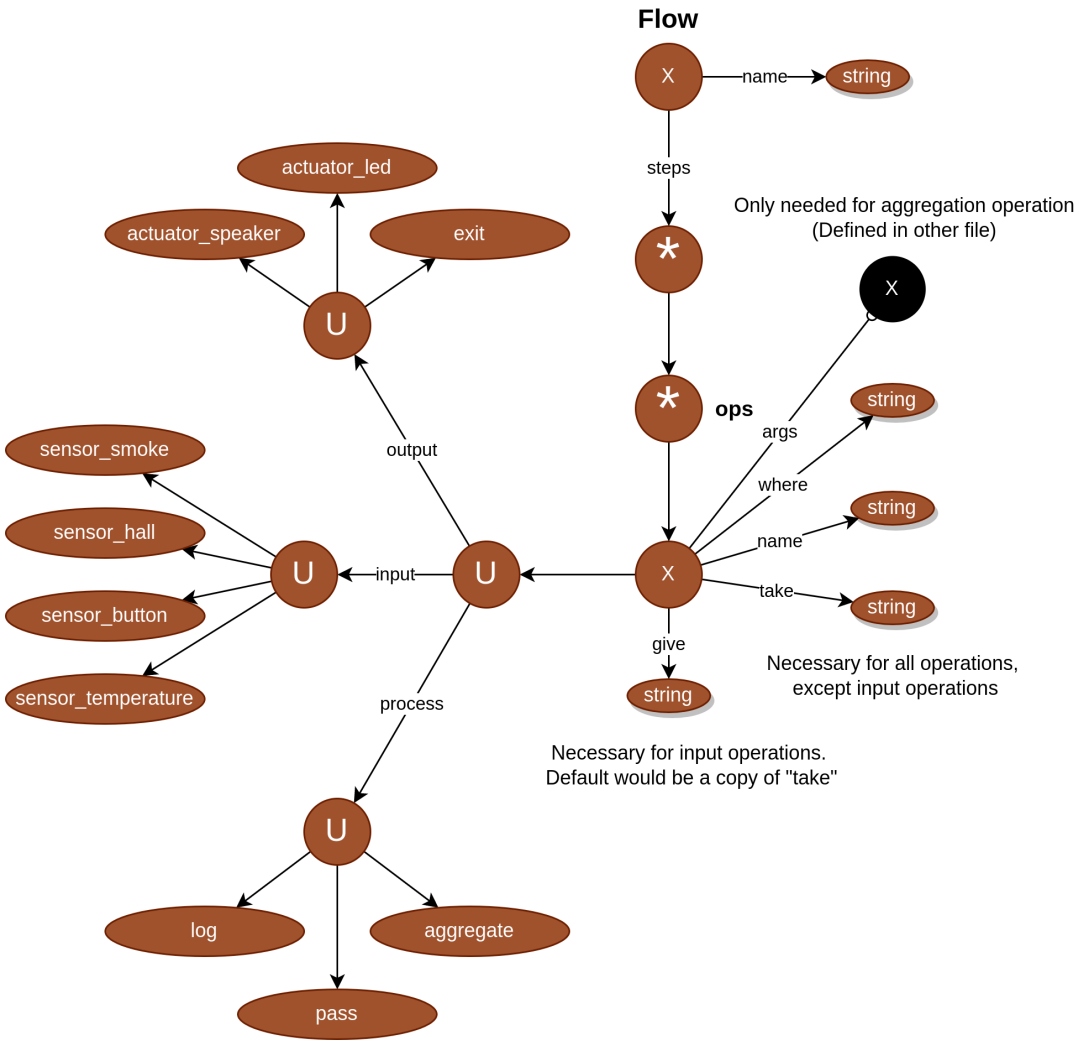

Abbildung 4.3: Complex-Object Notation für den neuen Flow. Die zugehörige Legende wird in Abbildung 4.5 gegeben.

die Parameter give und take werden hier angegeben. Die mittlere Operation ist eine Aggregationsoperation. Dies wird als Diagramm in der Complex-Object-Darstellung in Abbildung 4.3 dargestellt. Dabei wird außerdem veranschaulicht, wie die Argumente angegeben werden können. Die eckigen Klammern stehen dabei für Objekte, welche an anderer Stelle genauer spezifiziert werden.

Um den Erklärungsaufwand zu reduzieren, wird im folgenden Teil die Syntax spezifiziert und gleichzeitig dazu erklärt, was deren Bedeutung und Funktion ist. Diese Event-Processing-Language (EPL) ist im Wesentlichen eine Anwendung der EPL welche im Buch [BD15] spezifiziert wird auf die spezielle Anwendung in der ME2.

4.2.1 JSON-Format der Aggregationsargumente

Wie bereits im oberen Abschnitt erwähnt, können jeder Operation des Flows Argumente übergeben werden. Zur Anwendung einer Aggregationsoperation sind die übergebenen Argumente essenziell, um das Verhalten der Operation festzulegen. Die Argumente bestimmen, welche Daten aggregiert werden sollen, wie groß die Aggregations-Fenster sein sollen, welche Nachricht erzeugt werden soll und welche Bedingungen erfüllt sein müssen. Um diese Informationen zu übermitteln, lassen sich drei wesentlich Argumente bestimmen:

- Die **Bedingung** um eine neue Nachricht zu erstellen:

  Hier legt man fest, wie die ankommenden Messergebnisse zueinander in Verhältnis stehen müssen, damit eine neue Nachricht ausgesendet wird.

- Das **Aggregationsfenster**:

  Das Aggregationsfenster legt fest, welche eingehenden Nachrichten gespeichert werden sollen und wie lange oder wie viele davon zur Laufzeit behalten werden sollen.

- Die **Erstellung einer neuen Nachricht**:

  Hierbei wird festgelegt, was im Falle einer erfüllten Bedingung geschehen soll, also welche Art von Nachricht, mit welchen Werten weiter geschickt werden soll.

Diese Parameter möchten wir uns nun genauer ansehen.

4.2.2 Aggregations-Bedingung (Condition)

Um zu bestimmen, ob das Aufrufen der Aggregationsoperation (durch das Eintreffen einer Nachricht) zur Erzeugung einer neuen Nachricht führen soll, benötigt es eine Aggregations-Bedingung.

Würde man zum Beispiel ein Feueralarmsystem konstruieren, könnte man die Anforderung so formulieren: "Wenn die Temperatur über 40 °C steigt, erzeuge eine Hitzewarnung". Also wäre hier die Bedingung: "Temp > 40.0". Eine Bedingung wird immer zu einem booleschen Wert `True` oder `False` evaluiert. Dieser kann rekursiv aus der Verknüpfung zweier Bedingungen und der Negation einer Bedingung aufgebaut sein.

Logische Verknüpfung

Eine logische Verknüpfung besteht aus zwei Bedingungen `c1` und `c2` und einem logischen Operator `operator` der entweder die Konjunktion `&&` oder die Disjunktion `||` darstellt. Ein Beispiel für die Schreibweise wird in Listing 4 gegeben. Die booleschen Werte der zwei Bedingungen werden dann dementsprechend zusammengefasst.

```
{
    "name": "my_flow",
    "steps": [
        [
            {
                "name": "senor_1",
                "input": "sensor_temperature",
                "where": "111111111",
                "give" : "temp"
            },
            ...
        ],
        [
            {
                "name": "aggr_1",
                "process": "aggregate",
                "where": "222222222",
                "take": "temp, button",
                "give" : "temp_aggr",
                "args": {
                    "condition": <condition>,
                    "windows": <windows>,
                    "create": <create>
                }
            },
            {
                "name": "pass_1",
                "process": "pass"
                "where": "333333333",
                "take" : "temp"
            },
            ...
        ],
        ...
        [
            {
                "name": "act_1",
                "output": "actuator_led",
                "where": "555555555",
                "take" : "temp_aggr"
            },
            ...
        ],
    ]
}
```

Listing 3: Beispiel eines Flows in JSON-Format. Die Werte in den eckigen Klammer stehen dabei für JSON-Objekte, welche an anderer Stelle genauer spezifiziert werden.

```
"condition": {
    "logic": {
        "c1": <condition>,
        "c2": <condition>,
        "operator": "||"
    }
}
```

Listing 4: Flow-Condition einer logischen Verknüpfung von Bedingungen in JSON-Format

```
"condition": {
    "negation": {
        "c": <condition>
    }
}
```

Listing 5: Flow-Condition einer Negation einer Bedingung in JSON-Format

Negation

Eine Negation (`negation`) besteht nur aus einer Bedingung `c`, deren boolescher Wert negiert wird und zurückgegeben wird, wie in Listing 5 dargestellt.

Boolesche Vergleiche

Boolesche Vergleiche bilden die atomaren Bedingungen. Dabei werden zwei Werte 'v1' und 'v2' gewählt und mittels eines Vergleichsoperators `operator` [2] zu einem booleschen Wert ausgewertet (dargestellt in Listing 6). Die Operatoren bestehen aus den in vielen Programmiersprachen gängigen Vergleichsoperatoren `<, >, >=, <=, = , !=`. Um Syntaxfehler zu vermeiden, werden = und == äquivalent behandelt.

In der Complex-Object Notation, wie sie in Abbildung 4.5 beschrieben wird, ergibt sich daraus das Diagramm in Abbildung 4.4. Dabei wurden bei der Notation leichte Änderungen vorgenommen, welche aber intuitiv verständlich sein sollten. Es wurden etwa in den Basiskonstruktoren teils schon konkrete Werte vorgegeben.

[2] Beispiel Java Operatoren: https://www.w3schools.com/java/java_operators.asp

```
"condition": {
    "comparison": {
        "comparator": ">",
        "v1": <value>,
        "v2": <value>
    }
}
```

Listing 6: Flow-Condition eines Vergleichs von zwei Werten in JSON-Format

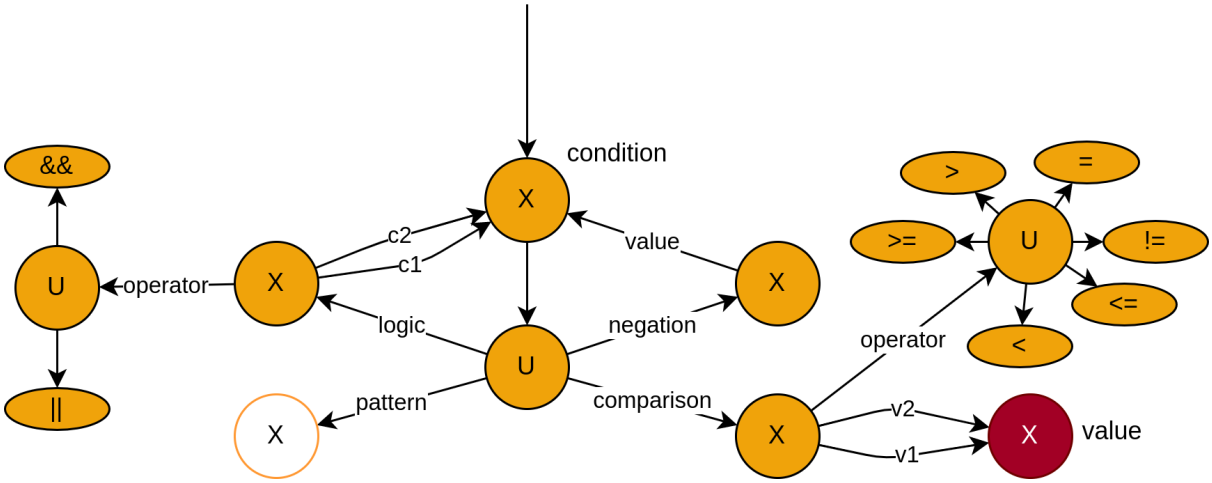

Abbildung 4.4: Complex-Object Notation für die Aggregationsbedingung

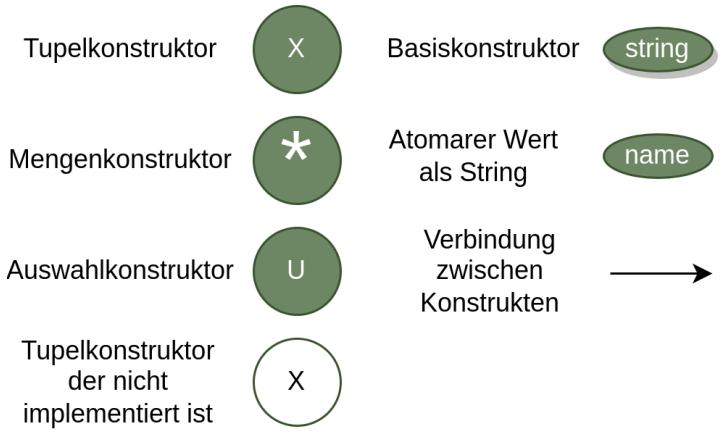

Abbildung 4.5: Legende für die Complex-Object Notation

```
"value": {
    "const": "my_string",
    "type": "string"
}
```

Listing 7: Flow-Value einer Konstante in JSON-Format

4.2.3 Werte ermitteln (Value)

Um in der Aggregations-Bedingung einen Vergleich zwischen zwei Werten machen zu können, muss zuerst festgelegt werden, woher diese Werte ermittelt werden sollen. In der EPL des Buches [BD15] stehen die Attribute der Events und Konstanten zur Verfügung. Außerdem lassen sich Funktionen auf diesen Werten ausführen und die Werte können miteinander verrechnet werden. Bei dieser Notation ist es erst im Kontext ersichtlich, dass bei einer Funktion wie `sum()` alle Events desselben Typs verrechnet werden sollen. Es wird also nicht direkt unterschieden, ob man vom aktuellsten Event dieses Typs ausgeht oder ob man sich auf alle Events dieses Typs bezieht. Darum werden im Folgenden die Werte direkt an die zugrunde liegende Funktion gekoppelt.

Damit bei der Implementierung immer klar ist, von welchem Datentyp der aktuelle Wert ist, wird dies immer mit angegeben. Möglich ist dabei `int, float, string`.

Hier werden nun die vier verschiedenen Arten beschrieben, wie man zu einem Wert gelangen kann.

Konstanten

Konstanten werden benötigt, um allgemeingültige Werte festzulegen. Zum Beispiel darf die Temperatur in einem Szenario nie höher als 20 °C sein. Diese Konstanten werden direkt mit einer festgelegten Zahl oder String in den Flow geschrieben (dargestellt in Listing 7).

Variablen

Variablen beziehen sich auf den Payload der aktuellsten Nachricht. Mit `from` gibt man an, von welchem Nachrichtentyp man eine Nachricht auslesen will. Der Payload dieser Nachricht könnte dann folgendermaßen aussehen: `{'result': 20.0, 'temp': 15.4}`. Um nun noch zu bestimmen, welches Attribut des Payloads gesucht ist, muss dieses in `name` angegeben werden (dargestellt in Listing 8). Da für jeden empfangenen Nachrichtentyp immer ein Fenster der Mindestgröße 1 besteht, kann man diese Variable auch auslesen, selbst wenn man gerade einen anderen Nachrichtentyp empfängt. In diesem Fall wird einfach die neueste Nachricht des Fensters dieses Nachrichtentyps entnommen.

```
"value": {
    "var": {
        "from": "sensor_1",
        "name": "result"
    },
    "type": "float"
}
```

Listing 8: Flow-Value einer Variablen in JSON-Format.

```
"value": {
    "compute": {
        "func": "add",
        "v1": <value>,
        "v2": <value>
    }
    "type": "float"
}
```

Listing 9: Flow-Value eines Wertes, welcher durch eine mathematische Operation auf zwei Werten erhalten wird in JSON-Format

Berechnungen

Um einfache Berechnungen wie Addition, Subtraktion und Multiplikation durchführen zu können, kann man mittels `compute` zwei Werte miteinander verrechnen. Dafür muss man die zwei Werte in `v1` und `v2` angeben und den Funktionstyp in `func` [9]. Dabei ist `v1` immer der erste Wert der Berechnung. Aktuell sind nur die naheliegenden Berechnungen wie `add, sub, mult, div` möglich. Es lassen sich aber unkompliziert weitere Berechnungen wie die Exponentialfunktion und die Modulo-Operation hinzufügen. Diese wurden aber für diese Arbeit als irrelevant betrachtet.

Funktionen

Um eine Aggregation auf demselben Nachrichtentyp durchzuführen, wird auch die Möglichkeit geboten, eine Funktion auf allen Nachrichten des Fensters eines Nachrichtentyps anzuwenden. Also bei einer Fenstergröße 14 zu einem Nachrichtentyp B würde `Sum(B)`, die Summe aller Werte der 14 letzten Nachrichten von B berechnen. Falls es erlaubt wird, auch mit unvollständigem Fenster Berechnungen durchzuführen, kann die Funktion auch auf weniger Werten ausgeführt werden. Die gewünschte Funktion kann unter `func` angegeben werden. Dabei stehen die Funktionen `sum, avg, max, min, count` zur Verfügung. All diese Funktionen

```
"value": {
    "calc": {
        "func": "avg",
        "var": {
            "from": "sensor_1",
            "name": "result"
        }
    }
    "type": "float"
}
```

Listing 10: Flow-Value eines Wertes, welcher durch eine Funktion auf allen konkreten Messergebnissen eines Nachrichtentyps erhalten wird, in JSON-Format

betrachten nur die Werte selbst und nicht, zu welchen Zeitpunkten diese Werte aufgetreten sind. Für komplexere Funktionen, wie etwa die Steigung, könnte man von einer Gleichverteilung ausgehen. So ließen sie sich ohne zusätzliche Modifikationen im Code einfach einfügen. Alternativ kann man auch die Ankunftszeiten der Nachrichten im Fenster mit abspeichern.

Im Parameter `var` wird noch der Nachrichtentyp und der Name des Werts angegeben (dargestellt in Listing 10). Hierbei ist zu beachten, dass `var` hierbei anders interpretiert wird als im Teil Variablen. Dort wurde nur nach der aktuellsten Nachricht geschaut, während hier alle Nachrichten des Fensters betrachtet werden.

Stellt man dies in der Complex-Object Darstellung dar ergibt sich das Diagramm in Abbildung 4.6.

4.2.4 Aktion, Nachricht erzeugen (Create)

Wenn der Bedingungsteil einer Ereignisregel erfüllt worden ist, so wird der Aktionsteil ausgeführt. Da wir in unserem Flow alle Aktuatoren als separate Operationen anlegen, benötigen wir an dieser Stelle keine speziellen Aktionen. Es genügt, wenn es möglich ist, die Aktuatoren zu aktivieren. Dies ist möglich, indem man Nachrichten generiert, welche von den Aktuatoren empfangen werden können. Um eine Nachricht zu generieren, wird der Name der Nachricht benötigt, nach der sie benannt werden soll. Um zusätzliche Informationen an nachfolgende Operationen übermitteln zu können, benötigen wir einen Payload, der übermittelt werden soll. Dieser lässt sich in `param` angeben, wie in Listing 11 dargestellt wird. Dabei kann eine Menge an Parametern angegeben werden. Jeder dieser Parameter benötigt einen Namen und einen Wert. Die Notation dieses Werts ist dieselbe, wie sie im oberen Teil beschrieben wurde. Es können also auch Berechnungen angegeben werden, welche evaluiert werden, bevor eine Nachricht generiert wird (das dazu passende Diagramm wird in Abbildung 4.7 gezeigt).

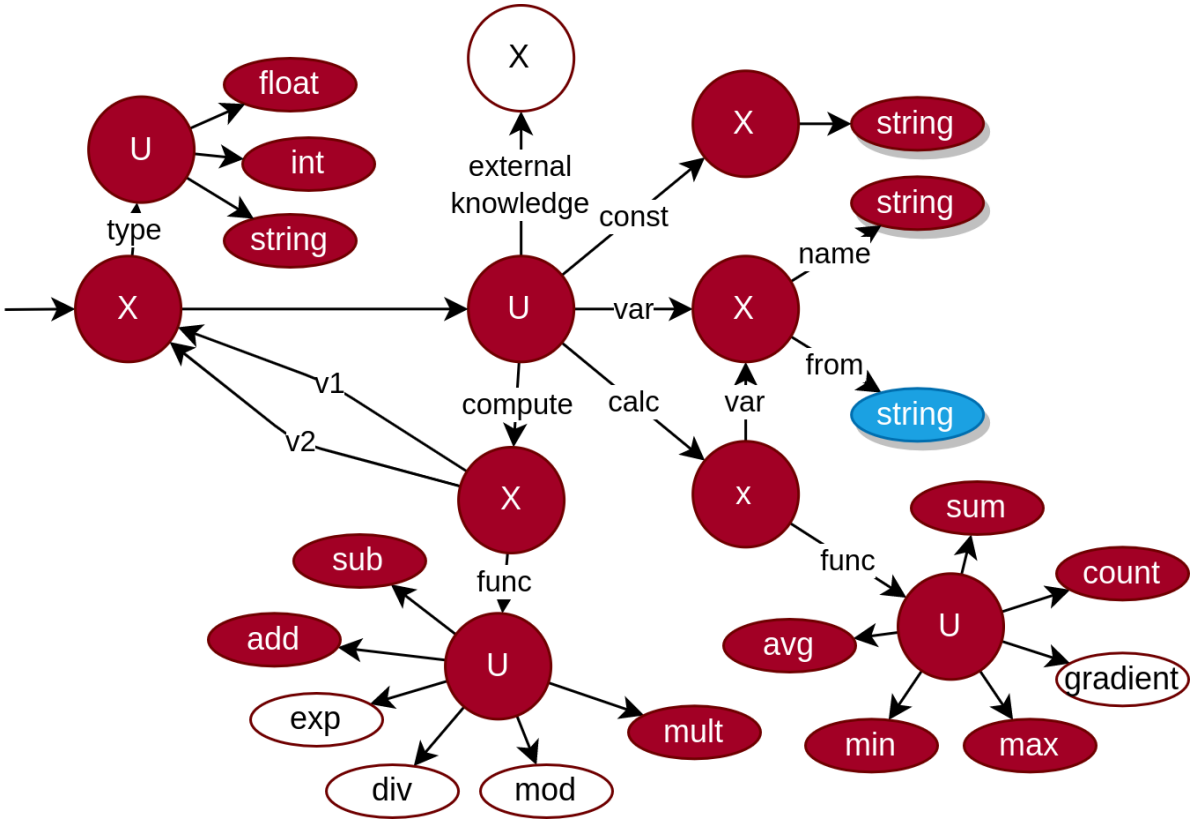

Abbildung 4.6: Complex-Object Notation für die Werte in der Aggregation

```
"create": {
    "name": "my_event",
    "param": [
        {
            "name": "param_one",
            "value": <value>
        },
        ...
    ]
}
```

Listing 11: Flow-Create zur Erzeugung einer neuen Nachricht mit den spezifizierten Parametern in JSON-Format

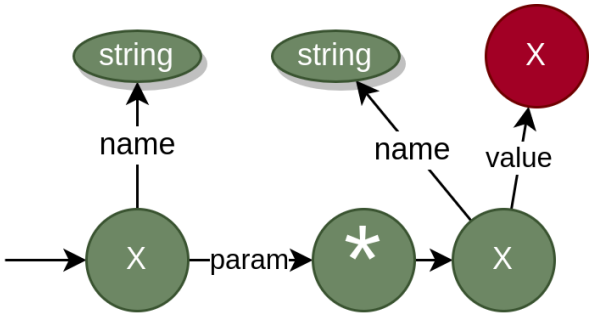

Abbildung 4.7: Complex-Object Notation für die Erzeugung neuer Nachrichten

4.2.5 Fenster festlegen (Windows)

Wie im Grundlagenteil erklärt wurde, ist es beim CEP nicht möglich, alle Nachrichten zur Weiterverarbeitung zu speichern. Dies hängt damit zusammen, dass der Nachrichtenfluss ohne zeitliche Begrenzung ist. Darum müssen ältere Nachrichten immer wieder verworfen werden. Man erhält ein Fenster, welches eine begrenzte Menge an aktuellen Nachrichten speichert. Es lassen sich im Wesentlichen zwei gängige Fensterarten unterscheiden. Zeitfenster verwerfen alle Nachrichten, deren Empfang eine bestimmte Zeit zurückliegt. Längenfenster verwerfen immer die älteste Nachricht, wenn eine neue Nachricht hinzukommt. Nur zu Beginn werden sie einmal komplett nach der festgelegten Größe aufgefüllt.

Um mehrere Nachrichtentypen aggregieren zu können, werden auch verschiedene Fenster benötigt. Diese Fenster werden durch einen Namen unterschieden. Neben den Angaben von `name` und `type` muss auch noch die Größe `size` angegeben werden (dargestellt in Listing 12). Diese ganze Zahl gibt für Längenfenster an, wie viele Nachrichten immer gleichzeitig behalten werden können. Für Zeitfenster gibt diese Zahl an, nach wie vielen Sekunden eine Nachricht verworfen werden darf. Um mögliche fehlerhafte Werte zu Beginn zu vermeiden, kann im Parameter `require_full` angegeben werden, ob das Längenfenster Werte liefern darf, wenn es noch nicht komplett gefüllt wurde. Will man zum Beispiel eine Veränderung eines Messwertes erkennen und Alarm schlagen, sobald sich dieser Wert signifikant ändert, so würde bei einem leeren Fenster, immer direkt zu Beginn ein Alarm ausgelöst werden.

Versucht man mehrere Nachrichten zu einer Nachricht zusammenzufassen, um die Auslastung des Netzwerkes zu reduzieren, so muss man feststellen, dass bei jeder empfangenen Nachricht eine neue Nachricht erzeugt wird. Die Anzahl der Nachrichten reduziert sich also durch die Aggregation nicht. Um eine solche Reduzierung zu ermöglichen, gibt es die Möglichkeit, ein `Batch-Window` zu erzeugen. In [ACL18] wird die Bedeutung eines Batch-Streams zur Effizienzsteigerung näher erläutert. Wird `batch` auf `true` gesetzt, so wird die Aggregation nur durchgeführt, sobald das Fenster seit der letzten Aggregation komplett neu aufgefüllt wurde. Bei einem Längenfenster der Größe 10 wäre das immer nach 10 Nachrichten und bei einem Zeitfenster der Länge 20 Sekunden wäre das bei einer Nachricht, welche mindestens 20 Sekunden nach der letzten Aggregation empfangen wurde. Alternativ ließe sich beim

```
"windows": [
    {
        "name": "window_one",
        "type": "length",
        "size": 5,
        "batch": false,
        "require_full": true
    },
    ...
]
```

Listing 12: Flow-Window zur Spezifizierung aller Fenster, welche bei der Ereignisregel benötigt werden in JSON-Format

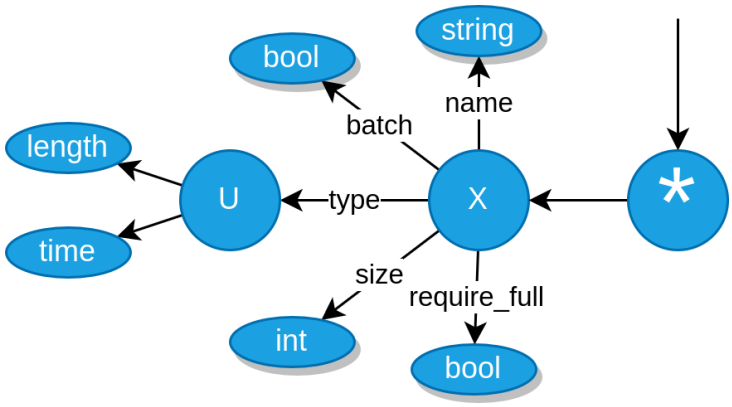

Abbildung 4.8: Complex-Object Notation für die Konfiguration der Fenster

Zeitfenster auch ein Timer setzen, welcher alle 20 Sekunden die Aggregation mit der aktuellsten Nachricht startet (dargestellt in Abbildung 4.8).

4.2.6 Zusammenfassung

Fasst man alle Teildiagramme zu einem Diagramm zusammen, erhält man das Diagramm in Abbildung 4.9 Zusammenfassend, ist es nun möglich, die flow.json-Datei mit einem Standard JSON-Parser einzulesen. Das erstellte Objekt kann direkt als ausführbare Ereignisregel betrachtet werden. Dazu muss das JSON-Objekt nur noch Schritt für Schritt traversiert werden. So lässt sich nun genau standardisiert vom IAMT mit der ME2 kommunizieren. Es muss nur eine JSON-Datei erstellt werden, welche alle Operationen enthält und die Ereignisregeln für Aggregationsoperationen genau spezifiziert.

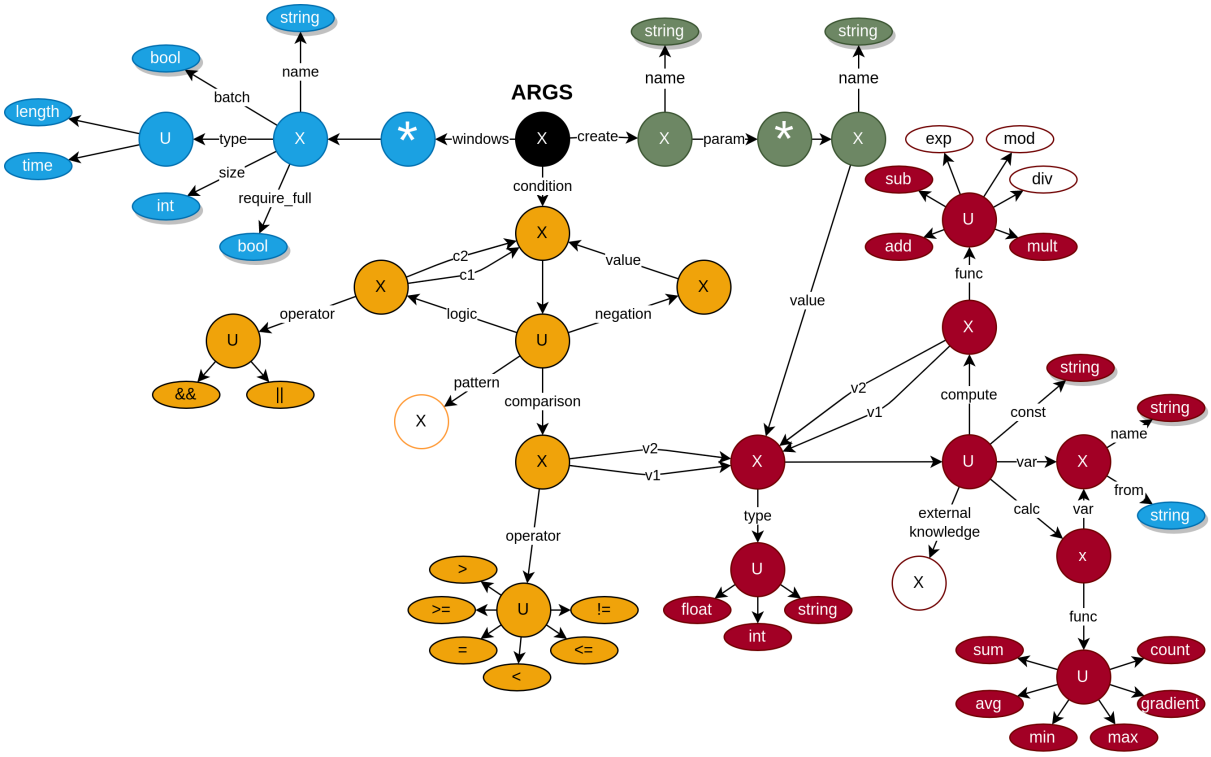

Abbildung 4.9: Complex-Object Notation für die gesamte Ereignisregel

4.3 Konzept zur Integration der ME2 in das IAMT

Nach der Implementierung der obigen Änderungen an der ME2 sollte ein Benutzer schon in der Lage sein, mittels mehrerer ME2s ein IoT-Netzwerk, welches Sensordatenströme und Aggregationen ermöglicht, zu erzeugen. Das Verfahren, alle Konfigurationsdateien und die Flows manuell zu erstellen und auf allen Geräten zu installieren, ist allerdings sehr aufwendig. Außerdem bedürfen kleine Änderungen wieder großem Aufwand und es ist dabei schwierig, den Überblick zu behalten. Das IAMT strebt, wie bereits in den Grundlagen in Kapitel 2 erwähnt, einen modellbasierten Ansatz an, um IoT-Netzwerke und Sensordatenströme zu realisieren. Dabei soll der Anwender sich nicht mehr viele Gedanken machen müssen, wie er seine konkreten Ziele erreichen will, sondern soll nur noch seine Ziele formulieren müssen. Dies ist möglich, indem der Anwender ein Modell erstellt, welches dem gewünschten Netzwerk entspricht. Dieses Modell wird anschließend vom IAMT interpretiert und realisiert. Um dies zu ermöglichen, muss das IAMT in der Lage sein, auf den verschiedenen Geräten vorkonfigurierte ME2s installieren zu können. Zudem muss es zu einem gegebenen Modell eine entsprechende flow.json-Datei generieren können. Diese soll dann auf die gegebenen Geräte angepasst werden können. Zu guter Letzt muss sie die einzelnen Geräte miteinander verbinden und sie starten können.

4.3.1 Modell erstellen

Die Erstellung des Modells läuft im IAMT über einen visuellen Graphen ab. Dies geschieht in zwei separaten Phasen ab. In Phase 1 wählt der Nutzer alle gewünschten Operationen aus und konfiguriert diese mit den gewünschten Parametern. Die Operationen werden als Knoten in dem visuellen Graphen im Frontend dargestellt. Durch gerichtete Kanten, welche die Knoten verbinden, wird festgelegt, in welcher Reihenfolge die Operationen ablaufen, beziehungsweise welche Nachrichten wie gesendet werden. In der zweiten Phase werden die Operationen den gegebenen Geräten zugeordnet. Dabei können mehrere Operationen auf demselben Gerät verwendet werden.

Um nun die gewünschte Funktionalität zu ermöglichen, müssen an diesem Konstrukt mehrere Änderungen vorgenommen werden. Zuerst muss das IAMT zwei verschiedene Arten an Operatoren erlauben. Zu Beginn dieser Arbeit werden alle Operatoren von der MBP geladen. Sie können dort erstellt werden, indem man die benötigten Start-Scripte, Installations-Scripte und andere Dateien hoch lädt und man die benötigten zusätzlichen Angaben tätigt. Ein Screenshot des Interfaces wird in Abbildung 5.1 gegeben. Um das Modell zu erweitern, müssen nun auch die Operationen, welche auf der ME2 ausgeführt werden können, als Operatoren angeboten werden. Man kann sie auch als virtuelle Operatoren bezeichnen, da sie nicht auf einem Gerät installiert werden, sondern stattdessen in den Flow der ME2 eingebaut werden und dann zur Laufzeit verwendet werden. Diese virtuellen Operatoren müssen nicht von der MBP erstellt oder geladen werden. Man kann sie direkt im IAMT zur Verfügung stellen und bei Gebrauch jeweils konfigurieren. Dabei kann auch die Aggregationsoperation genauer spezifiziert werden. Diese virtuellen Operatoren können nicht schon in Phase zwei direkt den jeweiligen Geräten zugewiesen werden. Zuvor muss für jedes der Geräte eine ME2 als realer Operator zugewiesen werden. Diese Operatoren werden so konfiguriert, dass ihnen Sensoren und Aktuatoren zugewiesen werden. Erst danach können die virtuellen Operatoren auf die realen Operatoren verteilt werden. Es muss also noch eine weitere Phase in die Erstellung des Modells eingefügt werden.

4.3.2 ME2 Konfigurieren und Deployen

Das Deployment der ME2 ist nur mit der Hilfe der MBP möglich. Wird ein Gerät auf der MBP registriert, so erhält die MBP Zugriff auf das Gerät über eine SSH-Verbindung [BBSS01]. Die ME2 wird auf der MBP wie ein gewöhnlicher Operator verwendet. Das bedeutet, sie enthält vier verschiedene Shell-Skripte, welche von der MBP aufgerufen werden können. Zum Installieren werden alle Dateien, welche zu dem Operator dazugehören, auf das Gerät geladen und das Installations-Script wird aufgerufen. Dieses Script sollte die ME2 insofern vorbereiten, damit sie nun einfach aufgerufen werden kann. Um die ME2 zu starten, wird das Start-Script aufgerufen. Dabei werden alle zuvor angegeben Parameter dem Script als Argumente mit übergeben. Das Start-Script soll nun die ME2, wie durch die Parameter spezifiziert, konfigurieren und anschließend starten. Die MBP muss noch zusätzlich zwei Scripte erhalten, von denen eines

überprüft, ob die ME2 gerade ausgeführt wird und eines, um die ME2 zu stoppen. Um sie zu deinstallieren, entfernt die MBP einfach wieder das erstellte Verzeichnis auf dem Gerät.

4.3.3 Interface für die Aggregationsoperation

Um eine Aggregationsoperation genauer zu spezifizieren, benötigt es ein Interface, mit dem der Benutzer dem IAMT die geforderten Eigenschaften der Aggregationsoperation mitteilen kann. Für dieses Interface lassen sich viele verschiedene Ansätze finden.

Der erste Ansatz wäre, dass alle Parameter für die Aggregationsoperation in **JSON-Format** eingegeben werden müssen. Dies wäre die einfachste Herangehensweise, welche gleichzeitig eine sehr hohe Flexibilität liefert. Der Nachteil dabei wäre, dass der Nutzer genau verstehen muss, wie dieses Format verwendet wird. Dabei muss sehr viel Text geschrieben werden. Der geschriebene Text ist aufgrund dessen Größe wahrscheinlich nicht sehr übersichtlich und es erhöht sich die Wahrscheinlichkeit für Fehler.

Eine weitere Methode, welche den Nutzer einen Text schreiben ließe, wäre den Nutzer seine Eingaben mittels einer Event-Processing-Language (EPL) tätigen zu lassen. Dabei ließe sich die EPL welche im Buch [BD15] vorgestellt wird, oder auch Esper-EPL[3] verwenden. Diese Methode ist eine deutlich übersichtlichere und verständlichere Weise, wie man die Ereignisregel für die Aggregationsoperation formulieren kann. Allerdings gehört auch hierzu ein gewisses Vorwissen der Syntax der EPL. Für die Implementierung wird außerdem ein Parser erfordert, welcher die Eingaben in der EPL Syntax in das gewünschte JSON-Format umwandelt.

Um die Eingabe einer Ereignisregel in das Interface möglichst intuitiv, benutzerfreundlich und übersichtlich zu gestalten, kann auch ein Ansatz gewählt werden, welcher recht ähnlich zu dem bekannten Bildungstool im Bereich Informatik, 'Scratch'[4] ist. Dabei werden bausteinartig Codeblöcke zusammengebaut, um eine Art von Programmiersprache zu bilden. Dies ermöglicht es Fehler in der Syntax zu vermeiden, da zuvor festgelegt wird, welche Blöcke wie zusammengefügt werden dürfen. Dieses Interface klingt nach einem sehr hohen Implementierungsaufwand. Da die Auswahl an Optionen in jedem Fall allerdings begrenzt ist, wird hier behauptet, dass der Aufwand dafür nicht zu groß ist. Wie aus Abbildung 4.9 ersichtlich wird, müssen nur 11 verschiedene Blöcke angeboten werden, da lediglich die Objekte von einem Block repräsentiert werden müssen, nicht aber deren Parameter. Die Parameter der Blöcke sind in dem Diagramm schon klar definiert und müssten nur als Eingabefelder oder Auswahlfelder dargestellt werden. Ein sehr ähnliches Interface ist in der MBP bereits implementiert worden und wird in Abbildung 4.10 gezeigt. Dieses Interface dient zur Eingabe von Bedingungen von Ereignisregeln. Dabei werden die einzelnen Teilbedingungen als separate Blöcke dargestellt, welche miteinander kombiniert werden können. In der MBP werden die Eingaben des Nutzers in eine Ereignisregel in der Schreibweise, welche von Esper verwendet wird, übersetzt. Damit

[3] Esper Event Processing Language: https://www.espertech.com/esper/
[4] Scratch: https://scratch.mit.edu/projects/editor/?tutorial=getStarted

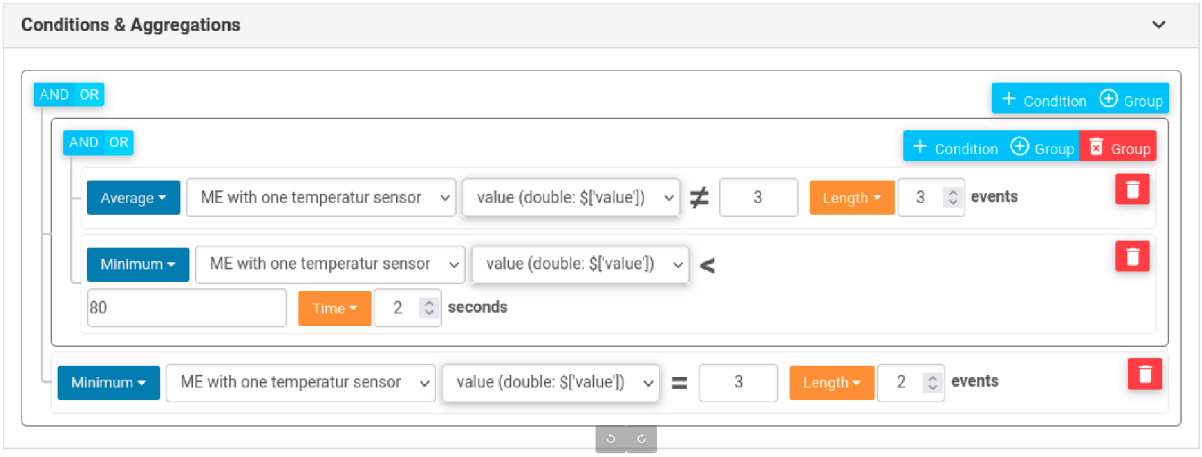

Abbildung 4.10: Ein Screenshot des Interfaces der MBP zur Erstellung einer Bedingung für eine Ereignisregel. Dabei wird die Bedingung mit mehreren verschachtelten Blöcken visualisiert.

ist auch schon eine grobe Vorlage vorhanden, wie diese Eingaben in tatsächliche Ereignisregeln umgewandelt werden können. Zusätzlich existiert in der MBP noch ein weiteres Interface, wie in Abbildung 4.11 zu sehen ist, welches auch eine ähnliche Bausteinstruktur aufweist, aber die Eingaben des Nutzers schon direkt in ein gültiges JSON-Format umwandelt.

4.3.4 Erstellen der flow.json-Datei

Nachdem das Modell im IAMT erfolgreich erstellt wurde, muss es in ein lauffähiges System umgewandelt werden. Dazu werden zuerst die benötigten ME2-Operatoren mittels der MBP auf den gegebenen Geräten installiert. Diese werden dann entsprechend der Parameter konfiguriert und gestartet. Nun ist es Aufgabe des ME-Konfigurators, welcher Teil des IAMT ist, zu dem Flow, welcher im Modell gegeben ist, eine flow.json-Datei zu erstellen und allen beteiligten ME2s zu schicken.

Die Parameter 'name' und 'args' können aus den Parametern der Operationen aus dem Modell übernommen werden. Die Parameter 'input', 'process' oder 'output' werden aus dem Typ der Operation gefolgert. Um den Parameter 'where' festlegen zu können, wird die ID des Geräts betrachtet, zu welchem diese Operation hinzugefügt worden ist. Um die Operation in eine valides JSON-Format zu schreiben, müssen noch die Parameter 'give' und 'take' belegt werden. Dazu können einfach die Pfeile, welche im Modell die Operationen miteinander verbinden, benannt werden. Diese Benennung kann entweder durch eine hochgezählte ID erfolgen oder manuell vom Nutzer benannt werden, um den Flow übersichtlicher zu gestalten. Dabei ist nur wichtig, dass der 'give' Parameter dem 'take' Parameter der nachfolgenden Operation entspricht.

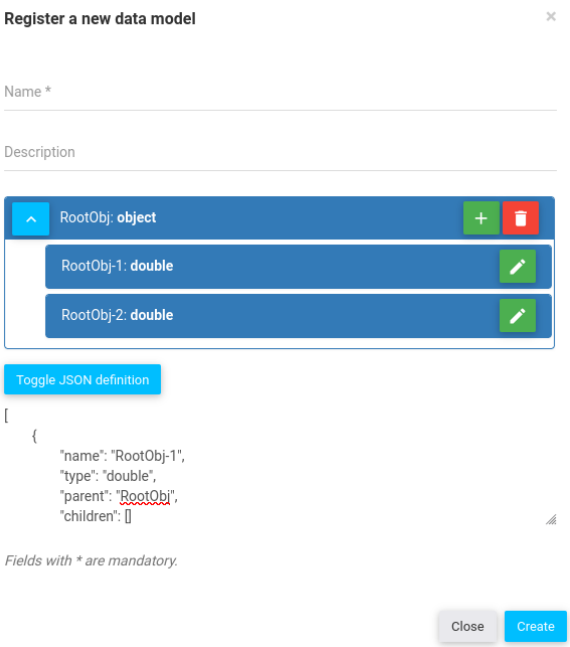

Abbildung 4.11: Ein Screenshot des Interfaces der MBP zur Erstellung eines Datamodels. Im oberen Teil kann das Modell mithilfe visueller Blöcke erstellt werden. Im unteren Teil wird das Modell im JSON-Format beschrieben.

Um zu bestimmen, wie viele Steps im Flow benötigt werden, ist es sinnvoll, zuerst den Graphen, welcher durch das Modell gegeben ist, auf den längsten Pfad hin zu untersuchen. Alle Operationen, welche auf diesem längsten Pfad liegen, können daraufhin ihrem jeweiligen Step im JSON-Format zugefügt werden. Anschließend können alle Input- und Output-Operationen dem ersten beziehungsweise letzten Step hinzugefügt werden. Die Process-Operationen welche nicht auf dem längsten Pfad liegen, müssen nun noch den richtigen Steps hinzugefügt werden. Diese Operationen sollten immer ein Step nachdem alle benötigten Nachrichten gesendet worden sind, eingefügt werden. Wenn also zum Beispiel die Operation 'Alarm' die Nachrichten 'Rauch' und 'Hitze' empfangen kann, und 'Rauch' in Step drei und 'Hitze' in Step fünf erzeugt wird, so muss die Operation 'Alarm' in Step sechs eingefügt werden. Zum Schluss müssen noch benötigte Pass-Operationen eingefügt werden. Diese gewährleisten, dass Nachrichten eine Operation erreichen können, welche mehrere Steps weiter hinten im Flow angesiedelt sind.

Nachdem die flow.json-Datei erstellt wurde, wird sie an alle ME2s über die verfügbare API übermittelt. Außerdem werden den ME2s die benötigten Nachbarknoten mitgeteilt, sodass diese über ihre IP-Adresse erreicht werden können.

5 Implementierung

Die Implementierung der entwickelten Konzepte lässt sich grob in 4 Teile gliedern. Dabei ist zu beachten, dass der vierte Teil in dieser Arbeit nicht mehr bearbeitet werden konnte.

- Abschnitt 5.1 handelt von der Anpassung der ME2 auf den veränderten Flow. Dies erforderte zahlreiche kleinere Änderungen an vielen verschiedenen Dateien der ME2.
- Abschnitt 5.2, welcher mit Abstand den Großteil des Aufwands dieser Arbeit ausmachte, beschreibt die Implementierung der Aggregationsoperation. Dabei wurden einige neue Dateien erstellt, welche die Verarbeitung der Ereignisregeln in JSON-Format ermöglichen.
- In Abschnitt 5.3 soll die ME2 so in die MBP integriert werden, dass sie als Operator verwendet werden kann. Dieser Teil war mit viel Konfigurationsaufwand und Problemen verbunden, weshalb er nicht vollständig fertiggestellt werden konnte.
- Der letzte Teil in Abschnitt 5.4 beinhaltet die Integration der ME2 in das IAMT. Dies gelang allerdings nicht, da der dritte Teil der Implementierung dazu benötigt worden wäre und viele anderer nötige Softwarekomponenten nicht vorhanden waren.

5.1 Anpassungen an der ME2

Dieser Abschnitt handelt davon, welche Änderungen an der ME2 vorgenommen werden müssen, um mit den neuen Flows umgehen zu können, welcher nun nicht mehr linear, sondern verzweigt sind.

5.1.1 Veränderung im Umgang mit den Flows

In einem verzweigten Flow müssen die passenden Operationen zu einer empfangenen Nachricht aufwändiger gesucht werden.

Um zu erkennen, welche Operation bei Empfang einer Nachricht als Nächstes ausgeführt werden soll, wird die letzte vollendete Operation betrachtet. Von dort aus wird nun untersucht, was der nächste Step ist. In diesem Step wird dann geschaut, ob eine der enthaltenen Operationen im Parameter `take` einen Namen hat, der dem Namen der empfangenen Nachricht entspricht. Existiert eine solche Operation, wird noch untersucht, ob die ID im Parameter

'where' der Operation der eigenen Geräte ID entspricht. Erst dann werden die gefundenen Operationen ausgeführt. Ausnahmen bilden dabei die ersten und letzten Operationen des Flows. Das Versenden einer Nachricht läuft sehr ähnlich ab. Nachdem alle Operationen des nächsten Steps gefunden wurden, welche den benötigten 'take' Parameter haben, wird die Nachricht an alle Geräte geschickt, deren ID bei den gefundenen Operationen verwendet wird. In der bisherigen Implementierung gab es nur eine Operation in jedem Step. Dadurch bliebe das Suchen der passenden Operationen des Steps erspart.

Diese Implementierung, welche es erfordert bei jeder Nachricht den Flow zu durchsuchen, bietet sowohl Vorteile als auch Nachteile. Einerseits ist die Komplexität des Codes sehr gering und der Code ist insgesamt sehr robust gegen alle möglichen Arten von unerwartetem Verhalten. Bei oft wechselnden Flows und wenig Nachrichten könnte diese Implementierung schneller und effizienter sein. In einem realen Szenario, ist allerdings davon auszugehen, dass der Flow sich selten ändert und viele Nachrichten empfangen werden. Darum wäre für zukünftige Implementierungen sinnvoll, einen Flow nur zu Beginn einmal zu durchsuchen und alle relevanten Daten so abzuspeichern, dass Nachrichten schneller verarbeitet werden können. Dies ist allerdings nicht Teil dieser Arbeit.

5.1.2 Besser verständliche Ausgaben

Bei dem erstmaligen Testdurchlauf der ME2 wurde das Terminal mit derartigen Ausgabe-Texten überflutet, sodass es beinahe unmöglich wurde, zu verstehen, was die ME2 gerade tatsächlich tut. Zwar gab es eine Einteilung in verschiedene Loglevels, wie 'Debug', 'Info', 'Warning' und 'Error', allerdings wurden diese Einteilung teils unterschiedlich bewertet. Docker, welches zum einfacheren simulieren des IoT-Netzwerkes verwendet wurde, ist so konfiguriert, dass mit jeder Ausgabe viele Informationen zu dem virtuellen Gerät mitgeliefert werden. Dies verbrauchte zusätzlichen Platz im Terminal und verschlechterte damit die Lesbarkeit. Um die Ausgaben besser verstehen zu können, wurden viele unnötige Ausgaben entfernt. Andere Ausgaben wurden neu geschrieben, sodass nur noch die wesentlichen Informationen kompakt ausgegeben werden. Um besser zu verstehen, in welchem Zustand sich die Nachricht der ME2 gerade befindet, wurde die Regel eingeführt, dass nur die Nachricht, welche angibt, welcher Step gerade ausgeführt, wird in einem groß geschriebenen Format ausgegeben wird. Alle anderen Nachrichten müssen eingerückt werden. Dadurch ist bei jeder Ausgabe erkennbar, wo man sich gerade im Flow befindet. Dies ist in der beispielhaften Terminal-Ausgabe in Abschnitt 6.5 ersichtlich.

5.1.3 Sonstige Änderungen

Des Weiteren wurden viele kleine Änderungen vorgenommen und einige Syntaxfehler behoben, welche durch neuere Python-Schreibweisen entstanden sind.

- Als Beispiele wären zu nennen, dass die Nachrichten nun einen zusätzlichen Namen erhalten haben. Dieses zusätzliche Attribut bedurfte zahlreicher kleinerer Änderungen am Code.
- Um die ME2s zu Testzwecken einfach miteinander verbinden zu können wurde ein zusätzlicher Parameter in der Konfigurationsdatei angegeben, mit welchem man die IP-Adressen der Nachbarn schon fest vorgeben kann.
- Es musste das Execution-Repository angepasst werden, damit es die korrekten Operationen abspeichern und finden kann.
- Die Entitäten, welche von der ME2 verwendet werden, mussten angepasst werden.

5.2 Parsen des Flow JSON und Ausführen der Aggregation

Die wichtigste Änderung des Flows besteht darin, dass eine Aggregationsoperation eingefügt worden ist. Bei der Ausführung dieser Operation wird im Wesentlichen die flow.json Datei als Objekt eingelesen und anschließend traversiert und dabei ausgeführt. Nur um die angegebenen Fenster zu realisieren, muss von diesem Konzept abgewichen werden.

5.2.1 Überprüfen der Korrektheit des Flows

Vorherige Implementierungen haben beim Durchlaufen eines Flows immer erst den gesamten Flow überprüft. Dabei wurde überprüft, ob dieser syntaktisch korrekt aufgebaut ist und ob alle semantischen Bedingungen erfüllt werden. Ein solches Überprüfen für die Aggregationsoperation ist wenig sinnvoll. Die syntaktische Korrektheit dieser Operation ist rekursiv definiert. Es muss also in einer komplexen rekursiven Art Schritt für Schritt jeder Teil der Operation überprüft werden. Beim Entdecken eines Fehlers führt dieser zum Abbruch des aktuellen Sensordatenstroms. Daraufhin sollte eine aussagekräftige Fehlermeldung generiert werden, welche dem Nutzer Informationen liefert, wie er den Flow anpassen muss. Dieses Verhalten kann auch erreicht werden, ohne dass der Flow zuvor komplett untersucht werden muss. Dabei kann der Flow auch direkt ohne eine vorherige Überprüfung ausgeführt werden. Tritt ein Fehler während der Ausführung auf, so wird der Flow abgebrochen und eine möglichst aussagekräftige Fehlermeldung wird ausgegeben. Dies liefert für den Nutzer dasselbe Ergebnis, wie wenn der Flow zuvor untersucht worden ist.

5.2.2 Bedingungen

Um die im Flow für eine Aggregationsoperation angegebenen Bedingungen zu überprüfen, müssen alle Teilbedingungen ausgewertet werden. Es wird davon ausgegangen, dass die Anzahl der Teilbedingungen gering ausfällt. Diese Annahme ermöglicht es einen simplen rekursiven Algorithmus zu wählen, um alle Bedingungen zu überprüfen. So wird bei 'konjunktion', 'disjunktion' oder 'negation' jeweils die Evaluierung der Teilbedingung rekursiv aufgerufen. Vergleiche zwischen Werten stellen dann die atomaren Bedingungen, und damit auch das Ende der Rekursion, dar. Diese können einfach ausgewertet werden, indem der angegebene Vergleichsoperator mit dem entsprechenden Operator in Python ersetzt wird. Zuvor müssen allerdings die beiden angegebenen Werte in tatsächliche Zahlenwerte, boolesche Werte oder Strings umgewandelt werden. Wie diese Werte evaluiert werden, wird im nächsten Teil betrachtet.

5.2.3 Werte

Werte können einen ganz unterschiedlichen Ursprung im Programm haben, werden aber immer nach dem Verarbeiten als einfache Strings, Zahlenwerte oder boolesche Werte repräsentiert. Der Ursprung kann aus dem Flow, der letzten Nachricht, aus dem Aggregations-Fenster oder einer Kombination dieser sein. Bei allen Werten muss der gewünschte Datentyp mit angegeben werden. Der ermittelte Wert wird dann nach dessen Bestimmung zu diesem Typ explizit gecastet.

Konstanten

Konstanten werden im Flow direkt mit angegeben. Dabei der als String mitgegebene Wert einfach zu dem angegebenen gewünschten Typ gecastet.

Variablen

Variablen stellen den mitgelieferten Wert des letzten Events dar. Wird also zum Beispiel ein Temperatur-Messung-Event mit der Temperatur 35 °C aggregiert, wird dann 35 °C als Wert der Variable verwendet. Die Variable muss allerdings dennoch benannt werden, da ein Event auch mehrere Werte transportieren kann. Um eine Variable zu laden, wird einfach die letzte empfangene Nachricht aus dem Fenster, dessen Implementierung später erläutert wird, geladen und der gesuchte Wert entsprechend des Namen der Variablen ausgelesen.

Mathematische Funktionen

Mathematische Funktionen werden über allen empfangenen Werten eines Event-Typs ausgeführt, die in einem Fenster gespeichert sind. Dabei sind nur Funktionen möglich, welche von mehreren Werten auf nur einen Wert abbilden. Das wären zum Beispiel: Minimum, Maximum, Summe, Produkt und Durchschnitt. Deren Implementierung ist trivial. Für die benötigten Einzelwerte, werden alle Werte des Fensters, entsprechen dem Variablennamen in eine Liste geladen und der Funktion übergeben.

Mathematische Berechnungen

Um einfache mathematische Operationen zu implementieren, wurde ein rekursiver Ansatz gewählt. Dabei werden die zwei gegebenen Werte zuerst in einer niedrigeren Rekursion zu konkreten Zahlenwerten, Strings oder auch booleschen Werten ausgewertet und dann die gewählte mathematische Operation ausgeführt.

5.2.4 Neue Nachrichten

Wird die Aggregationsbedingung erfüllt, so wird ein neues Event erzeugt. Dabei wird der Payload der empfangenen Nachricht überschrieben und die Nachricht wird weitergeleitet. Ist die Bedingung nicht erfüllt oder es tritt ein Fehler auf, so wird keine neue Nachricht gesendet und der Flow endet an dieser Stelle. Um zu bestimmen, an welche Knoten die Nachricht gesendet wird, werden alle Operationen des nachfolgenden Steps durchsucht. Bei denjenigen Operationen, bei welchen der String im Parameter 'take' mit dem String aus dem Parameter 'give' der jetzigen Operation übereinstimmt, wird eine neue Nachricht gesendet. Dabei wird, wie schon in der vorigen Implementierung, die Nachricht selbst in die Nachrichten Historie angeheftet. Zusätzlich wird noch der neue Parameter 'name' mit dem gegebenen String aus 'give' ausgefüllt. Durch die Historie kann nun auch noch am Ende des Flows nachvollzogen werden, welchen Weg die Nachricht durch den Flow genommen hat.

5.2.5 Aggregationsfenster

Aggregationsfenster stellen den komplexesten Teil der Implementierung dar. Diese müssen Nachrichten speichern und sind somit nicht zustandslos. Darum werden sie innerhalb einer Klasse gespeichert.

Die Nachrichten werden also immer im flüchtigen Speicher des jeweiligen Python-Objekts gehalten. Dies verringert natürlich die Robustheit des Netzwerks gegenüber Abstürzen von einzelnen Geräten. In unserer Anwendung sollte dies allerdings kein Problem sein. Es wird davon ausgegangen, dass alle Nachrichten, die im Aggregationsfenster gehalten werden, nur

innerhalb einer kurzen Zeitspanne empfangen wurden. Bei einem Absturz gehen alle diese Nachrichten verloren. Dies führt allerdings nur so lange zu einem abweichenden Verhalten, bis das Fenster wieder vollständig aufgefüllt worden ist. Nach unserer Annahme ist dies nur eine kurze Zeitspanne und sollte deshalb unproblematisch sein, da die Probleme, welche durch den Absturz verursacht werden, nur unwesentlich vergrößert werden. Darüber hinaus lässt sich die Speicherung wesentlich effizienter vornehmen, wenn die Nachrichten nicht auf einen festen Speicher geschrieben werden müssen.

Die Implementierung der zwei verschiedenen Fensterarten, welche in den Grundlagen in Abschnitt 2.4.1 erläutert werden, unterscheidet sich teils fundamental. Deshalb werden sie in folgenden Abschnitten separat betrachtet.

Längenfenster

Längenfenster sollen immer eine gewisse Anzahl an Nachrichten im Speicher halten. Erst wenn mehr Nachrichten als die vorgegebene Anzahl empfangen werden, sollen die ältesten Nachrichten entfernt werden. Diese Art von Speicher lässt sich auch als Ringspeicher bezeichnen. Die Implementierung dieses Ringspeichers wird mittels eines Arrays vorgenommen. Dieser Array wird zu Beginn mit der Länge des Fensters initialisiert. Ein Pointer zeigt dabei immer auf die aktuelle Stelle, an der neue Nachrichten eingefügt werden sollen. Nach jeder Nachricht wandert der Pointer um einen Schritt weiter im Array. Überschreitet er die Array-Größe fängt er wieder von vorne an. Eine Nachricht wird mit dem Erstell-Datum der Nachricht und allen Parametern des Payloads gespeichert. Alle anderen Informationen der Nachricht werden verworfen. Für zukünftige Aggregationen sind diese nicht mehr relevant.

Um mit Nachrichten, welche in einer falschen Reihenfolge empfangen werden, umgehen zu können, wurde ein Reparatur-Algorithmus eingeführt. Dieser sortiert die später empfangenen Nachrichten an der richtigen Stelle ein und passt dann die Position der anderen Nachrichten und der Pointer entsprechend an. In speziellen Fällen könnte dies zu unerwartetem Verhalten führen. Will man beispielsweise die Sensordaten auf Schwankungen untersuchen, so kann es zu Problemen führen, wenn im Nachhinein die vorherigen Werte angepasst werden können. Im Allgemeinen ist allerdings davon auszugehen, dass das gewünschte Verhalten erreicht wird.

Bei der Erstellung des Fensters stehen dem Nutzer zwei Auswahlmöglichkeiten zur Verfügung, welche bei der Implementierung beachtet werden müssen:

- Nur mit vollem Fenster arbeiten:

  Wird diese Option gewählt, so wird nur etwas zurückgegeben, sobald das Fenster einmal gefüllt worden ist. Dies wird mittels des Zählers der Gesamtzahl aller empfangenen Nachrichten überprüft.

- Nur einmal pro Fenster auswerten:

Diese Option bestimmt, ob nach jeder Nachricht eine Auswertung stattfindet oder nur nach jedem mal, wenn das Fenster komplett neu aufgefüllt worden ist. Dabei wird überprüft, ob der Pointer auf die letzte Position im Array zeigt. Nur dann wird eine Aggregation eingeleitet.

Zeitfenster

Der große Unterschied bei der Implementierung von Zeitfenstern ist, dass die Anzahl der gespeicherten Nachrichten nicht im Voraus festgelegt werden kann. Es wird lediglich bestimmt, wie lange Nachrichten im Speicher gehalten werden. Darum ist es sinnvoller, diese Speicherung mittels einer Linked-List vorzunehmen. Dabei können neue Nachrichten einfach am Ende angehängt werden und alte Nachrichten am Anfang wieder entfernt werden. Das Entfernen der Nachrichten passiert nicht aktiv, nachdem ein Timer abgelaufen ist, sondern passiv, bevor die neueste Nachricht verarbeitet wird. Sobald also eine neue Nachricht eingefügt wird, wird überprüft, ob es noch Nachrichten gibt, welche zu alt sind. Diese werden dann anschließend entfernt. Da Python keine Linked-List zur Verfügung stellt, welche für genau die benötigten Funktionen optimiert ist, wurde die Linked-List selbst implementiert.

In Listing 13 wird ein Python-Dictionary dargestellt, welches helfen soll, zu verstehen, wie die Fenster abgespeichert werden. Wichtig ist dabei zu beachten, dass dies nicht exakt der wirklichen Implementierung entspricht. Die Namen der Fenster werden gleich gewählt, wie die Namen der Eventtypen, die sie speichern sollen. Im Dictionary 'types' wird zu jedem Fenster abgespeichert, ob es vom Typ Längenfenster oder Zeitfenster ist. Im Dictionary 'queues' werden die Fenster dann tatsächlich gespeichert. Dabei wird für jedes Fenster unter 'messages' die Liste an gespeicherten Nachrichten gespeichert. Diese Liste ist entweder ein Array oder eine Linked-List. Für die Nachrichten werden nur die Attribute des Payload 'param' und die Zeitpunkte des Empfangs der Nachrichten gespeichert. Alle weiteren Angaben wie die Größe, der Zähler, der Pointer und ob es ein Batch-Fenster ist, werden auch in dem Fenster gespeichert.

5.2.6 Probleme

Bei der Implementierung der ME2 sind sehr oft Schwierigkeiten aufgetreten, welche mit der Nutzung der ME2 im Zusammenspiel zusammenhängen. Die wichtigste Funktion der ME2 besteht darin, dass sie, zusammen mit anderen ME2s, gut kommuniziert. Dass die Protokolle zur Kommunikation wie geplant ablaufen, ist also von zentraler Bedeutung. Darum konnte man in nur sehr wenigen Bereichen die Implementierung mit einer einzelnen ME2 oder einem einzelnen Operator auf der ME2 testen. Gängige Methoden um Komponenten in stark zusammenhängenden Architekturen zu testen bzw. zu integrieren, ist es einzelne Komponenten mit Test-Dummys oder Test-Treibern zu ersetzen. Dabei lässt sich der Top-Down oder Bottom-Up-Approach unterscheiden. Bei der Top-Down Herangehensweise, werden schrittweise alle Komponenten erst mit Dummys simuliert und anschließend Stück für Stück durch

```
queues: {
  "name_of_first_length_window": {
    "messages": [
      {
        date: datetime                  #message create at
        param: {}                       #data sent with the message
      },
    ]
    "pointer": int,                     #position in array
    "counter": int,                     #total number of inserted messages
    "is_batch": bool,                   #if true only create new message if
                                        #last create message was max_len ago
    "require_full": bool,               #if true don't send messages until
                                        #the window was filled once completely
    "max_length": int                   #size of window
  },

  "name_of_first_time_window": {
    "messages": [                       #linked-list
      {
        date: datetime                  #message create at
        values: {}                      #data sent with the message
      },
    ]
    "counter": int,                     #total number of inserted messages
    "is_batch": bool,                   #if true only create new message if
                                        #last create message was max_len ago
    "max_time": int                     #size of window in microseconds
  }
}


types: {
  "name_of_first_length_window": bool   #true if the window is time_window
                                        #instead of length_window
}
```

Listing 13: Python-Dictionaries zur beispielhaften Demonstration der Datenstruktur der Fenster. Man muss dabei beachten, dass es nicht der tatsächlichen Implementierung entspricht, da die Listen unterschiedlich implementiert wurden.

funktionierende Komponenten ersetzt. Bei der Bottom-Up Herangehensweise werden erst die einzelnen Komponenten vollständig implementiert und von einfachen Treibern bedient. Nach und nach werden dann die Treiber durch tatsächliche Komponenten ersetzt. Beide Ansätze wären bei der ME2 kaum hilfreich, da die einzelnen Komponenten nicht in einer Hierarchie zueinander stehen, sondern in beide Richtungen voneinander abhängen. Das Entwickeln der Testumgebung mit Test-ME2s wäre unverhältnismäßig zum Implementierungsaufwand selbst. Der Grund dafür ist, dass für beinahe jeden Testfall eine neue Testumgebung aufgebaut werden müsste. Diese Testumgebung müsste immer die genau passenden Nachrichten senden bzw. empfangen, wie es vom Protokoll vorgegeben ist. Darum wurde stattdessen der primitive Ansatz gewählt, das Programm für jeden Test neu auf mehreren ME2s durchlaufen zu lassen, und anhand der Terminal-Ausgabe zu erkennen, wie das Programm abgelaufen ist, bzw. welche Fehler aufgetreten sind. Dafür wurde die Terminal-Ausgabe gänzlich umstrukturiert und übersichtlicher gemacht, um den Programmablauf klar erkennen zu können. Im Folgenden werden noch zwei weitere kleinere Schwierigkeiten bei der Implementierung genannt.

- Kein Auffinden der Nachbarn:

  Um einen Flow auf mehreren Geräten durchzuführen, müssen die einzelnen Geräte einander im Netzwerk finden. Dafür gab es schon eine implementierte Methode, welche allerdings nicht funktionierte. In Zukunft soll die Verbindung der ME2s über einen externen Konfigurator hergestellt werden. Als Übergangslösung wurde die config.yaml Datei dahingehend erweitert, dass man die IP-Adressen der Nachbarn fest angeben kann. Dadurch wurde es ermöglicht, die ME2s in Beziehung zueinander zu testen.

- Docker Konfiguration:

  Um die Testausführungen deutlich zu beschleunigen, wurde Docker und Docker-Compose verwendet. Dabei wurde ein virtuelles Netzwerk erzeugt, in dem dann die virtuellen MEs einander auffinden sollen. Dies war bereit vorhanden, allerdings war diese Konfiguration fehlerbehaftet. Das Ermitteln der Probleme hierbei nahm einiges an Zeit in Anspruch.

5.3 Erstellen der ME als MBP-Operator

Um die ME2 über die MBP verwendbar zu machen, muss sie als Operator registriert werden. Das dazu gegebene Interface wird in Abbildung 5.1 gezeigt. Neben dem Namen und der Beschreibung können hierbei auch das Data-Model und die Access-Control-Policies angegeben werden. Diese spielen aber für die Verwendung der ME2 keine Rolle. Unter Operator-Scripts werden alle für die Installation und Bedienung nötigen Dateien hochgeladen.

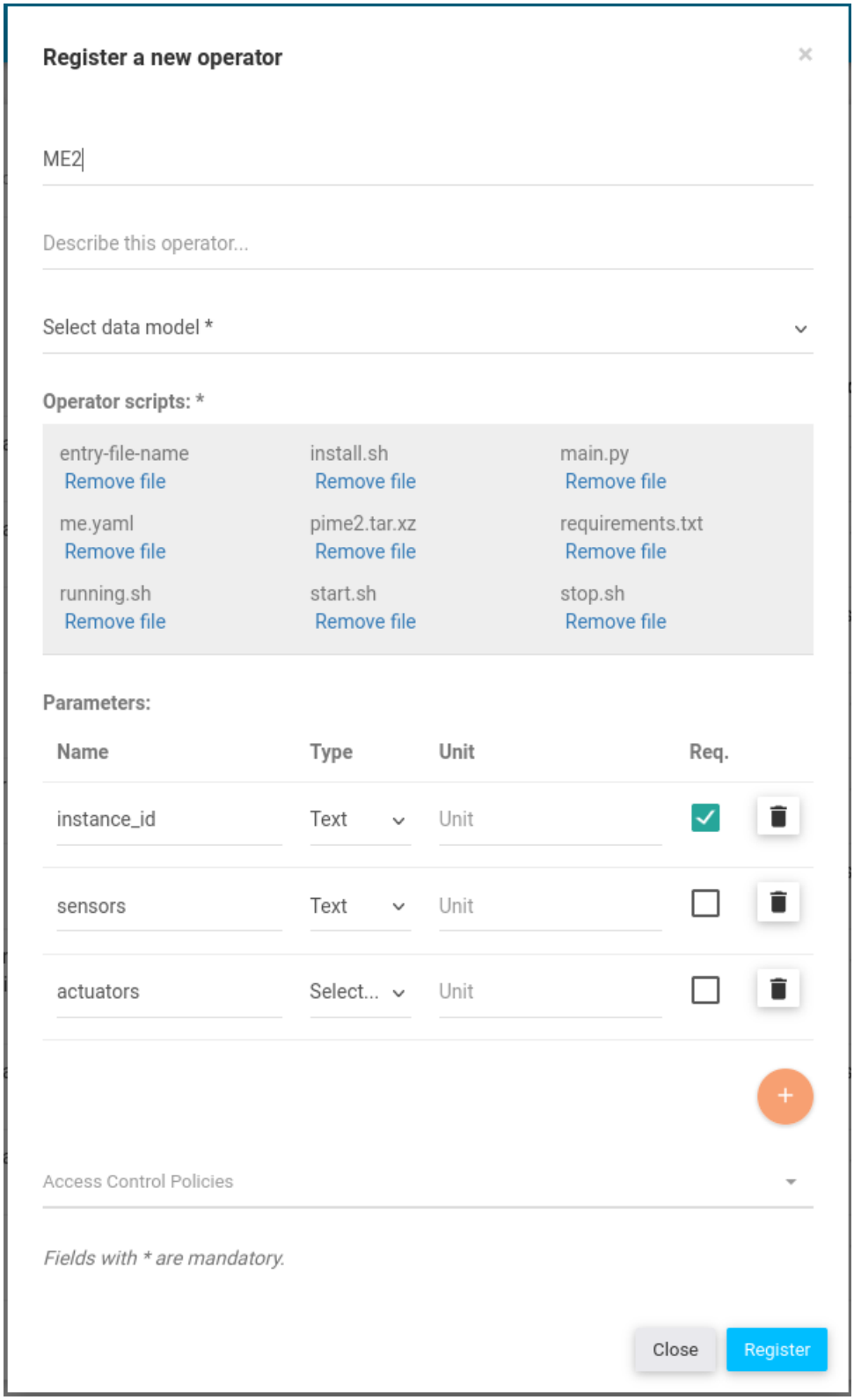

Abbildung 5.1: Interface zum Registrieren eines neuen Operators auf der MBP. Dabei werden alle benötigten Dateien per Drag-and-drop in das mittlere Feld geladen. Im unteren Teil können beliebig viele Parameter angegeben werden.

5.3.1 Installation über die MBP

Um die ME2 auf einem Gerät zu installieren, werden alle Dateien in ein Verzeichnis auf dem Gerät kopiert. Die MBP ruft dann das `install.sh` Script auf. Dieses Script entpackt im ersten Schritt die komprimierte pime2 Datei zu einem neuen Unterverzeichnis. Darin befindet sich der gesamte Code für die ME2. Anschließend updatet das Script das Gerät und installiert Python3 und PIP3. Mithilfe von pip3 werden dann alle Python-Bibliotheken, welche in der Datei requirements.txt angegeben werden, installiert. Danach ist die ME2 vollständig installiert und kann nun mit dem `start.sh` Script gestartet werden.

5.3.2 Konfiguration mittels Parameter

Die MBP sieht vor, dass man Operatoren mittels festgelegter Parameter genauer spezifizieren kann. Die angegebenen Parameter werden, beim Aufrufen des `start.sh` Scripts als Argumente mit übergeben. Dieses Script ist identisch, wie die anderen Start-Scripte der Beispieloperatoren der MBP implementiert. Dabei wird das Python-Script, welches in der `entry-file-name` Datei referenziert wird, mit den Parametern aufgerufen. Die Logs werden in ein Datei `start.log` geschrieben, und die Prozess-ID wird in der Datei `pid.txt` gespeichert. Da in der `entry-file-name`-Datei `main.py` referenziert wird, wird dieses Python-Script mit den Parametern aufgerufen. In der `main.py` Datei sollen nun die übergebenen Parameter ausgewertet werden und die ME2 entsprechend neu konfiguriert werden. Dazu werden erst alle Parameter in ein Python-Dictionary (dict) [1] eingelesen. Außerdem wird die `config.yaml` Datei ebenfalls in ein `dict` eingelesen. Diese Konfigurations-Datei ist für alle Parameter mit Default-Werten schon initialisiert. Danach wird für jeden initialisierten Parameter der entsprechende Parameter im `dict` der eingelesenen `config.yaml` überschrieben. Sind alle Parameter verarbeitet, so wird das überarbeitete `dict` mittels `yaml.dump()` wieder zurück in die Datei geschrieben. Da nun die Konfigurationsdatei den neuen Bedürfnissen angepasst ist, kann nun die ME2 regulär gestartet werden. Dazu wird `main.py` welches sich im entpackten Verzeichnis `pime2` befindet, aufgerufen und bekommt die neue Konfigurationsdatei übergeben.

Es gilt hierbei zu beachten, dass wenn ein ME2-Operator ein weiteres Mal initialisiert wird, so sind die Standardwerte bereits überschrieben. Dies bedeutet insbesondere, dass Änderungen an der Konfiguration nicht die letzten Änderungen rückgängig machen. Will man aber die letzten Änderungen rückgängig machen, so müssen die Parameter einfach erneut überschrieben werden.

Im Abschnitt 5.3.4 wird beschrieben, warum es nicht möglich war, die Parameterübergabe der MBP zu testen. Da somit nicht getestet wurde, wie sich verschieden komplexe Parameter auf die ME2 übertragen lassen, wird erwartet, dass es bei einer möglichen zukünftigen Ausführung zu Schwierigkeiten kommen wird. Konkret wäre dies, wenn man mehrere Sensoren auf der

[1] Python Datenstrukturen und das Dictionary: https://docs.python.org/3/tutorial/datastructures.html

ME2 initialisieren will, die dazu benötigten Argumente müssen als String in den Parametern angeben werden. Dieser String wird beim Verarbeiten der Argumente als eine Zeichenfolge im JSON-Format interpretiert. Wenn diese Argumente aber als Zeichenfolge im JSON-Format übergeben werden, so enthalten diese Leerzeichen, Zeilenumbrüche und Sonderzeichen. Dies könnte zu Fehlverhalten führen, wenn die Argumente über SSH der ME2 übergeben werden. Um dieses mögliche Problem zu vermeiden, wird vorgeschlagen, dass im IAMT bereits beim Ausfüllen der Parameter eine zusätzliche Codierung vorgenommen wird. Der MBP wird also ein codierter String übergeben, welcher keine problematischen Zeichen enthält. Die ME2 kann dann die Parameter einfach wieder decodieren.

5.3.3 Scripte zum Überwachen und Stoppen

Damit die MBP die ME2 stoppen kann, oder überprüfen kann, ob sie noch am Laufen ist gibt es die Scripte `stop.sh` und `running.sh`. Beide Scripte sind identisch zu den Scripten der Beispieloperatoren der MBP. Das Script `running.sh` überprüft, ob die Prozess-ID welche in `pid.txt` gespeichert wurde noch am Laufen ist und gibt entsprechend `true` oder `false` zurück. Das Script `stop.sh` stoppt den Prozess aus der `pid.txt`-Datei und entfernt anschließen diese Datei.

5.3.4 Probleme

- Fehlermeldung `unauthorized` beim Verbinden mit der MBP:

  Wenn man das IAMT nutzt, so verbindet dieses sich mit der verknüpften MBP Instanz. Über die MBP werden verfügbare Operatoren geladen und angebotene Funktionen über die API verwendet. Dabei muss man sich im Frontend anmelden, welches sich mit diesen Eingabedaten selbst bei der MBP einloggt. Zu Beginn lieferte das IAMT immer die aussage lose Fehlermeldung `unauthorized`. Dieses Problem wurde relativ zügig behoben.

- Erreichen der VMs:

  Anfangs, war es nicht möglich die virtuellen Maschinen zu erreichen, welche zum Testen der ME2 und der MBP benötigt wurden. Dies lag unter anderem an der Firewall und Netzwerk-Konfiguration des internen Informatiknetzwerkes der Universität Stuttgart.

- Verbinden der VMs mit der MBP:

  Um die VMs der MBP verfügbar zu machen, müssen diese der MBP Zugriff über eine SSH-Verbindung gewährleisten. Wie man diese SSH-Verbindung herstellt, wird in der Dokumentation der MBP ausführlich beschrieben. Obwohl dieser Beschreibung genauestens gefolgt wurde, gelang es nicht eine solche Verbindung herzustellen. Tatsächlich ist es der MBP nicht möglich, über private und öffentliche Keys eine SSH Verbindung

```
ubuntu@ba-linus-1:~/scripts/mbp6412fdeccd992758c685b349$ ls
entry-file-name  main.py         me.yaml       requirements.txt  start.sh
install.sh       mbp.properties  pime2.tar.xz  running.sh        stop.sh
ubuntu@ba-linus-1:~/scripts/mbp6412fdeccd992758c685b349$ sudo bash install.sh
ubuntu@ba-linus-1:~/scripts/mbp6412fdeccd992758c685b349$ ls
entry-file-name  main.py         me.yaml  pime2.tar.xz      running.sh  stop.sh
install.sh       mbp.properties  pime2    requirements.txt  start.sh
```

Listing 14: Ausgaben eines Terminals auf einer virtuellen Maschine, nachdem ein Operator von der MBP installiert wurde. Diese Verzeichnisstruktur wurde durch die MBP angelegt. Erst durch manuelles Starten der Installationsskripts wird pime2 entpackt.

zu starten. Nur mithilfe von Passwörtern für die SSH-Verbindung, war es möglich, die Verbindung herzustellen.

- MBP ruft Scripte nicht auf:

  Das während der Entwicklung schwerwiegendste Problem war, dass die MBP die Scripte des Operators nie aufgerufen hat. Wählt man zum Beispiel einen Sensor aus, welcher die ME-2 als Operator hat, so kann man diesen installieren. Nach dem Installieren kann man auf 'Starten' klicken. Daraufhin erhält man eine Benachrichtigung, die besagt: 'Component started successfully'. Gleichzeitig ändert sich die Statusleiste zu 'Running' und man erhält am unteren linken Rand, die Option 'Stop operator'. Ein Screenshot wird in Abbildung 5.2 gegeben. Es deutet also alles daraufhin, dass die ME2 erfolgreich installiert und gestartet wurde. Wenn man nun aber das Verzeichnis auf der VM, welches die MBP angelegt hat, genauer betrachtet, so fällt auf, dass das komprimierte Unterverzeichnis pime2 nicht entpackt wurde. Führt man das `install.sh` Script auf, so wird das Verzeichnis entpackt. In Listing 14 wird demonstriert, wie pime2 erst nach dem manuellen Aufruf von install.sh entpackt wurde. Durch mehrere Tests wurde so festgestellt, dass keines der Scripte jemals von der MBP aufgerufen wurde, aber die MBP immer positive Rückmeldungen herausgab.

Viele der hier genannten Probleme waren nur schwer zu beheben, da kein Zugriff auf die laufenden Instanzen vorhanden war und die Fehlermeldungen wenig bis keine Hinweise zur Problemursache lieferten. Ein selbständiges Behandeln dieser Probleme, hätte es erfordert, große Softwarekomponenten zu bearbeiten, welche außerhalb der Aufgabenbereiche dieser Arbeit lagen und hätte somit unverhältnismäßig viel Zeit in Anspruch genommen. Meist wurden die Probleme, vonseiten des Betreuers und dessen Assistenten recht schnell behoben, nachdem das Problem richtig identifiziert wurde. Allerdings war in der Zwischenzeit ein Weiterarbeiten an der eigentlichen Aufgabe nur sehr erschwert oder gar nicht möglich. Es ist nicht gelungen in dieser Arbeit ein vollständig funktionierenden MBP Operator zu erstellen und zu verwenden.

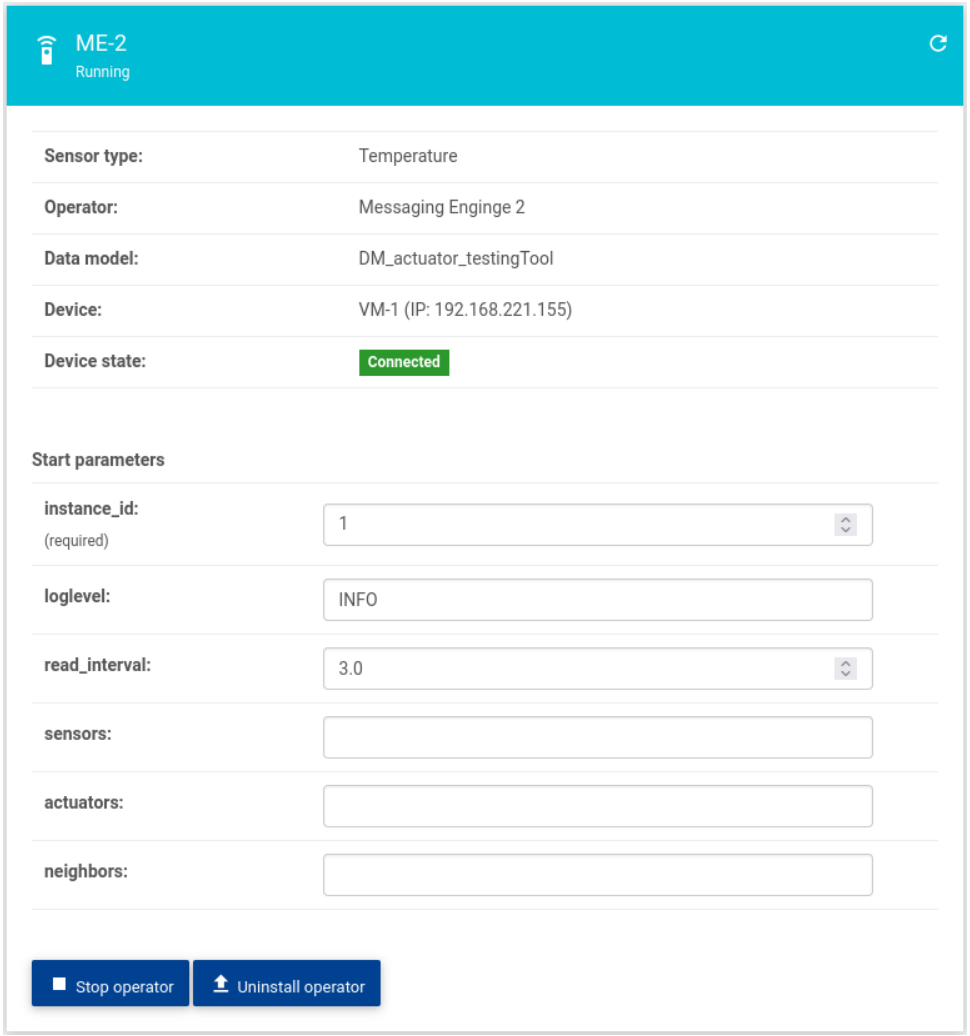

Abbildung 5.2: Screenshot des MBP Interfaces, nachdem eine ME2 gestartet wurde. Am oberen linken Rand ist dabei zu sehen, dass der Operator am Laufen ist. Der Operator kann am unteren linken Rand gestoppt oder deinstalliert werden.

5.4 Integration der ME2 in das IAMT

Zum Zeitpunkt der Bearbeitung dieser Bachelorarbeit, war der ME2-Konfigurator des IAMT noch in Bearbeitung. Darum konnte dieser noch nicht verwendet oder angepasst werden. Die Konzepte, wie die ME2 mit der neuen Aggregationsfunktion in das IAMT integriert werden kann, wurden im Kapitel 4 bereit benannt. Es wurde noch kein Implementierungsversuch zu diesen Konzepten unternommen. Ein Grund dafür ist auch, dass es nicht gelungen ist, die ME2 als Operator auf der MBP zu registrieren und es somit nicht möglich war diesen im IAMT zu verwenden.

6 Anwendungsfall

Um die Möglichkeiten und Grenzen der Aggregation mittels der Messaging Engine darzustellen, wird in diesem Kapitel ein mögliches Szenario einer Nutzung durchgespielt.

6.1 Szenario für die Überwachung von Saunen

Man stelle sich folgendes Szenario vor: Eine Therme hat zwei neue Saunen. Eine davon ist eine Dampfsauna mit einer Temperatur von 75 °C und die andere ist eine gewöhnliche Sauna mit 85 °C. Nun möchte die Therme ein Überwachungssystem einbauen, um mögliche Übertemperaturen in den Saunen festzustellen. Die Saunen dürfen nie ihre maximale Temperatur überschreiten. Außerdem darf die Dampfsauna nie wärmer als die normale Sauna sein, da sonst das Beheizungssystem fehlerhaft wäre. Zur Verfügung stehen ihr drei IoT-Geräte, in diesem Fall Raspberry-PIs[1]. Um den Alarm auszulösen gibt es ein einfaches LED-Licht, welches im Zimmer des Bademeisters blinken soll. Für die Messung der Temperatur in der Dampfsauna steht ein Temperatursensor zur Verfügung, welcher feuchte resistent ist. Allerdings hat dieser oft Schwankungen in den Messergebnissen und ist zusätzlich falsch kalibriert, sodass das Ergebnis immer 5 °C zu viel anzeigt. Für die normale Sauna steht ein Temperatursensor zur Verfügung, welcher immer präzise Ergebnisse liefert. Die einzelnen Geräte werden so verkabelt, dass es sinnvoll ist, die Nachrichten der Dampfsauna von Gerät-1 an das Gerät-2 der 85° Sauna zu schicken, und von dieser aus an Gerät-3 in das Zimmer des Bademeisters weiterzuleiten. Als Diagramm kann man den Aufbau also wie in Abbildung 6.1 darstellen.

6.2 Konzept zur Überwachung der Saunen

Um nun mittels dieser Geräte vernünftige Ergebnisse zu erzielen, müssen die Daten folgendermaßen aggregiert werden: Für den Temperatursensor, welcher starke Schwankungen in seinen Messergebnissen hat, ist es naheliegend, die Daten erst zu bereinigen, bevor diese weiter gesendet werden. Ein möglicher Ansatz dafür wäre es, hochfrequente Messungen durchzuführen und dann immer nur den Durchschnitt der Messung als Ergebnis weiterzuschicken. In diesem Beispiel werden alle drei Sekunden Messungen vorgenommen. Diese Messungen

[1] Raspberry Pi: https://www.raspberrypi.com/documentation

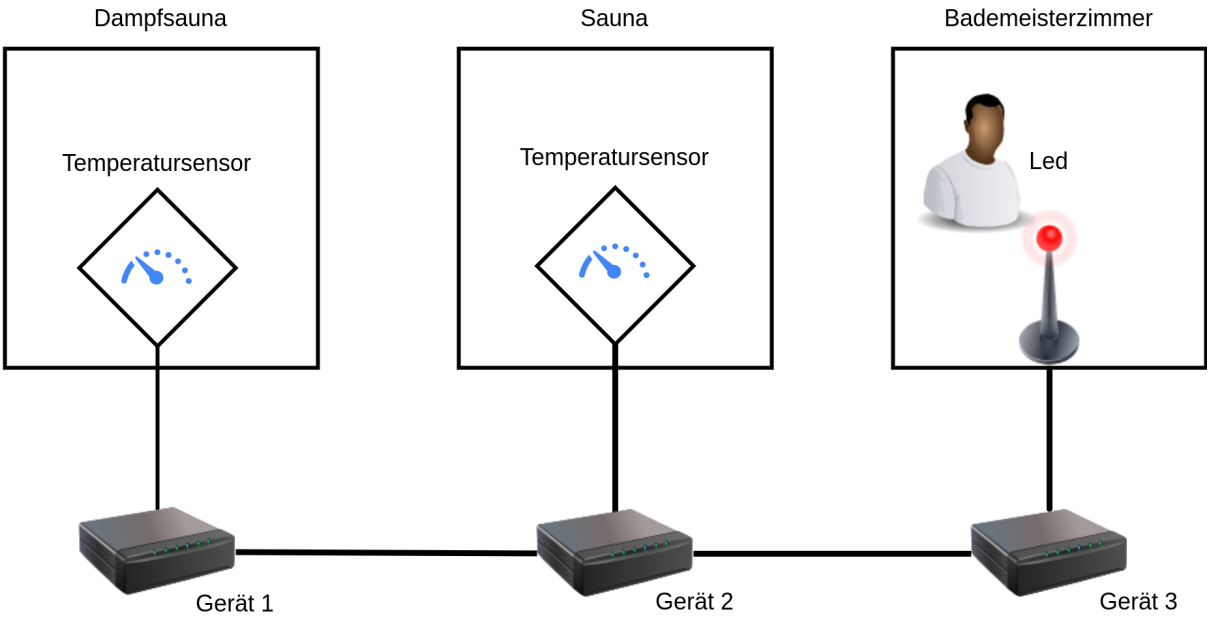

Abbildung 6.1: Diagramm des Anwendungsfalls zur Überwachung von Saunen. Dabei werden die 3 IoT-Geräte mit ihren zugehörigen Temperatursensoren oder LEDs gezeigt und wie diese auf die verschiedenen Räume verteilt sind.

werden dann aggregiert und nur alle 5 Messungen wird eine neue Nachricht ausgesendet. Also wird insgesamt alle 15 Sekunden eine Messung an das nächste Gerät gesendet. Dies ist mittels einer Aggregation innerhalb desselben Geräts möglich. Bevor nun aber das Ergebnis weitergeschickt wird, muss erst der Messfehler des Temperatursensors behoben werden. Wir wissen also, dass immer 5 °C zu viel angezeigt werden. Darum müssen wir nur diese 5 °C wieder abziehen. Um zu überprüfen, ob ein Alarm ausgelöst werden soll, müssen wir nur für beide Sensoren überprüfen, ob sie ihren jeweiligen Maximalwert überschreiten und ob die Dampfsauna wärmer ist als die normale Sauna. In natürlicher Sprache lässt sich die Aggregation also folgendermaßen formulieren: 'Wenn die Temperatur der Dampfsauna 75 °C überschreitet oder die Temperatur der normalen Sauna 85 °C überschreitet oder die Temperatur der Dampfsauna höher ist als die der normalen Sauna, dann löse ein Alarm-Signal aus.' Diese Aggregation wird am besten noch auf dem Gerät 2 ausgeführt, da sonst unnötig viele Nachrichten gesendet werden müssen. Gerät 3 hat dann nur noch die Aufgabe, bei Empfang eines Alarm-Signals die LED anzuschalten. Dieser Flow lässt sich als Diagramm wie in Abbildung 6.2 visualisieren.

6.3 Vorbereitung

Um die drei Geräte einzurichten, muss die Messaging Engine auf jedem Gerät installiert werden. Für jedes Gerät muss die Messaging Engine beim Start so konfiguriert werden, dass sie

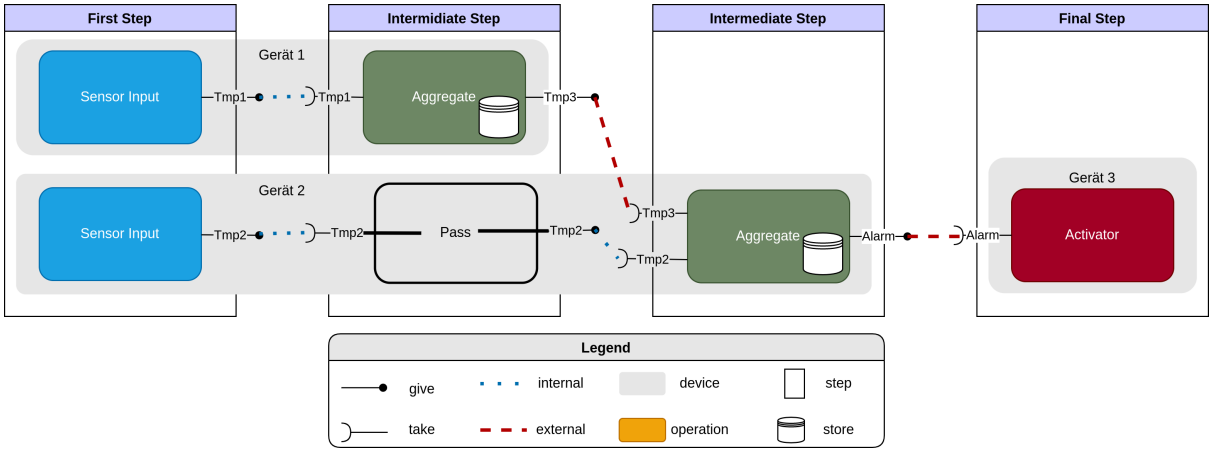

Abbildung 6.2: Diagramm für den gewünschten Flow.

```
instance_id: 111111111
loglevel: INFO
database: me2.db
host: 10.5.0.2
port: 5683
read_interval: 3.0
sensors:
  - name: "Temperature-Sensor"
    type: temperature
    gpio1: 12
```

Listing 15: Die Datei 'config.yaml' von Gerät 1. Wichtig ist dabei die instance_id, das read_interval und der Temperatursensor

die richtigen Sensoren und Aktuatoren bedienen kann. Beispielsweise muss dem ersten und zweiten Gerät mitgeteilt werden, dass sie einen Temperatursensor angeschlossen bekommen haben und an welchem Pin des Raspberry-PIs er anliegt. Das dritte Gerät muss so konfiguriert werden, dass es die LED als Aktuator kennt. Bei beiden Temperatursensoren muss noch jeweils das Lese-Intervall festgelegt werden. Bei Gerät 1 wird dieses auf 3 Sekunden gelegt und bei Gerät 2 auf 20 Sekunden. Außerdem muss allen Geräten eine ID gegeben werden. Diese ID muss mit der ID des Flows übereinstimmen.

Alle diese Informationen sollen beim Hochfahren der ME in der config.yaml Datei geschrieben werden. Diese Datei ist bei allen ME mit Standardwerten bereits initialisiert. Zu einem späteren Zeitpunkt soll es möglich sein, diese Informationen in der MBP als Operator-Parameter einzugeben. Jeder ausgefüllte Parameter wird dann den vorgegebenen Parameter der config.yaml Datei überschreiben. Die fertig erstellten Konfigurationsdateien könnten wie in folgenden Listings aussehen [15, 16, 17]. Die dazugehörige Erklärung kann in Kapitel 4 gefunden werden.

```
instance_id: 222222222
loglevel: INFO
database: me2.db
host: 10.5.0.4
port: 5683
read_interval: 20.0
sensors:
  - name: "Temperature-Sensor"
    type: temperature
    gpio1: 12
```

Listing 16: Die Datei 'config.yaml' von Gerät 2

```
instance_id: 333333333
loglevel: INFO
database: me2.db
host: 10.5.0.6
port: 5683
actuators:
  - name: "Led"
    type: led
    gpio1: 14
```

Listing 17: Die Datei 'config.yaml' von Gerät 3

Diese Konfiguration wurde in diesem Beispiel manuell vorgenommen. Das bedeutet, die config.yaml-Datei wurde mit einem Editor bearbeitet. Sobald alle Komponenten dieser Software lauffähig sind, soll die Konfiguration Frontend des IAMT stattfinden. Dabei registriert man für jedes der Geräte eine generische ME und konfiguriert diese mittels vorgegebener Parameter. In Abbildung 6.3 sieht man, wie die Parametereingabe funktioniert. Das Bild ist ein Screenshot der aktuellen MBP mit einer ME2 als Operator. Zwar ist im Frontend der MBP diese Konfiguration schon möglich, aber die dazugehörige Backend-Funktionalität der ME2 ist noch nicht lauffähig. Mehr dazu steht im Implementierungsteil Kapitel 5.

6.4 Erstellung des Flows

Die Erstellung des Flows wurde ebenfalls manuell vorgenommen. Dabei wurde die Datei flow.json in einem Editor erstellt. Auch hier soll durch zukünftige Arbeiten eine Konfiguration mittels des Frontends möglich sein. Dabei werden alle Operatoren, welche im Flow erlaubt

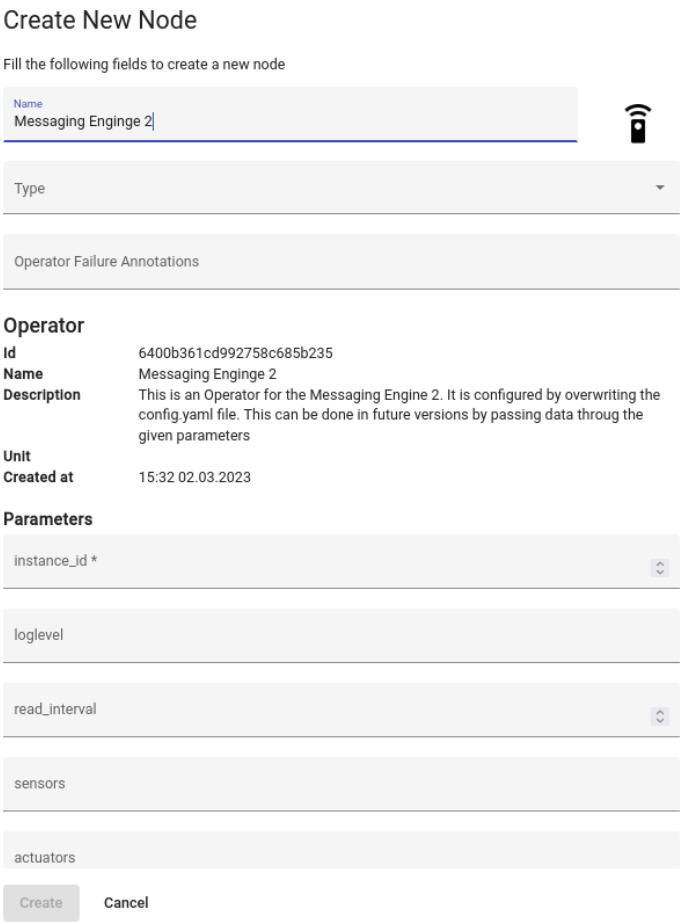

Abbildung 6.3: IAMT Interface des ME2 Operators. Dabei können die verschiedenen Parameter mit Strings oder Zahlen ausgefüllt werden.

sind als virtuelle Operatoren angeboten. Im Folgenden wird beschrieben, wie der Ablauf sein sollte, sobald alle Software Komponenten funktionsfähig sind.

Im ersten Schritt wählt man alle nötigen Operationen aus, welche für das geplante Netzwerk nötig sind. Dies wäre in unserem Fall zweimal eine Sensor-Read-Operation, zweimal eine Aggregationsoperation und einmal eine Aktuator-Operation. Diese werden im Frontend des IAMTs am linken Bildschirmrand zur Verfügung gestellt, wie in Abbildung 6.4 zu sehen ist. Bei der Auswahl der jeweiligen Operationen werden direkt die nötigen Parameter und Konfigurationen der Operationen in einem Interface abgefragt. Dabei wird zum Beispiel bei den Sensor-Read-Operationen abgefragt, wie die gesendete Nachricht heißen soll und um welchen Sensor es sich handelt. Bei den Aktuator Operationen wird gefragt, welche Nachricht empfangen werden soll und welcher Aktuator ausgelöst werden soll. Die Aggregationsoperationen benötigen ein etwas komplexeres Interface. Sind alle Operationen angegeben und verknüpft worden, so müssen diese noch, in einem weiteren Schritt, den verschiedenen ME2s,

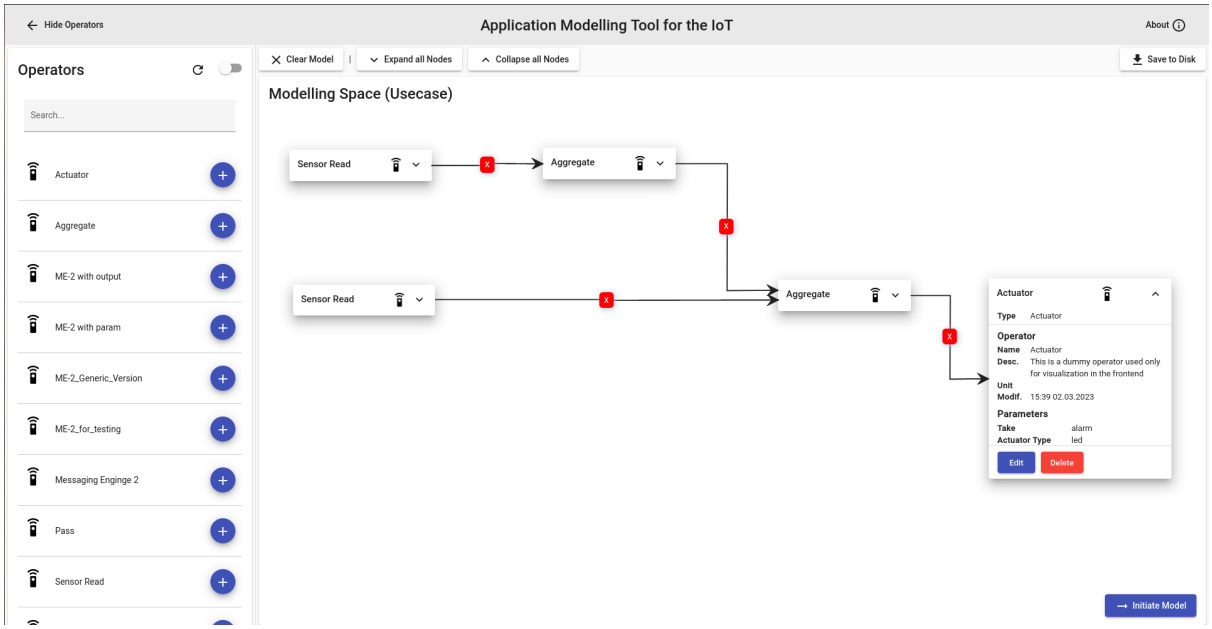

Abbildung 6.4: Interface des IAMT Frontends. Die verfügbaren Operationen am linken Bildschirmrand können ausgewählt werden und dann im Graphen platziert und verbunden werden. Klickt man auf den Button am unteren rechten Rand, so kann man die Operationen den verschiedenen Geräten zuweisen.

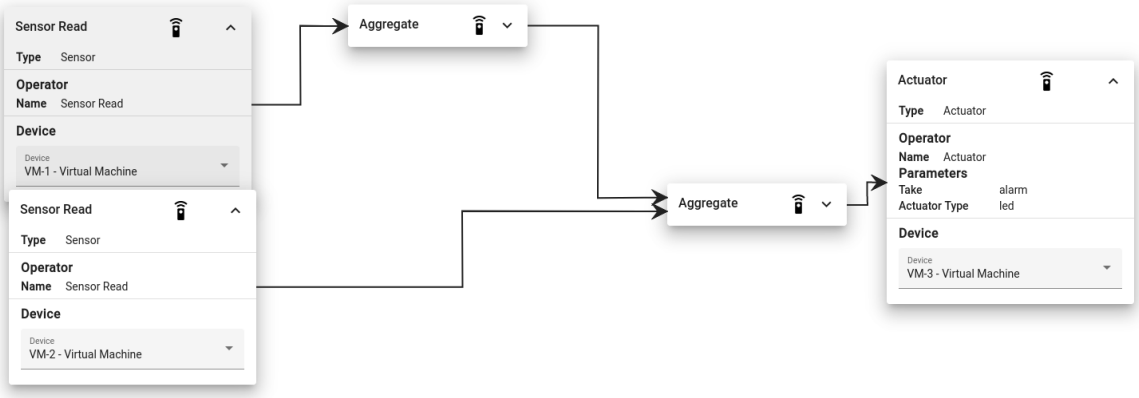

Abbildung 6.5: Interface IAMT Frontends zur Auswahl der Geräte für die gewählten Operationen. Dabei kann jeder Operation im Dropdown-Menü ein Gerät zugewiesen werden.

auf den jeweiligen Geräten, zugewiesen werden. Dies lässt sich mit dem Klick auf 'Initiate Model' erreichen. In dem folgenden Interface in Abbildung 6.5 kann für jede Operation im Dropdown-Menü das zugehörige Gerät ausgewählt werden. Klickt man anschließend auf 'Deploy' so wird mittels des ME-Konfigurators die flow.json Datei automatisch erstellt und zu den verschiedenen Geräten geschickt. Diese werden ihren jeweiligen Nachbarn bekannt gemacht und können somit ihre Aufgabe in diesem Flow erfüllen.

Listing 20 stellt den gesamten Flow dar, welcher alle gewünschten Anforderungen dieses Anwendungsfalls erfüllt. Die beiden Aggregationen wurden, zur besseren Lesbarkeit in Listing 18 und Listing 19 ausgelagert.

6.5 Ausführung

Um das Szenario testen zu können, benötigen wir zuerst die drei IoT-Geräte. Da die MBP und der Konfigurator zum Zeitpunkt dieses Experiments noch nicht voll funktionsfähig waren, wurde die Konfiguration der Geräte manuell vorgenommen. Dabei wurde Docker-Compose verwendet, um besser auswertbare Terminal-Ausgaben, welche in Abschnitt 6.5 zu sehen sind, zu erhalten. Die einzelnen Geräte werden mit Docker in einem virtuellen Netzwerk simuliert. Dabei werden in der Konfigurationsdatei schon zu Beginn die IP-Adressen der jeweiligen Nachbarn bekannt gegeben. Die Sensoren und Aktuatoren werden im Test-Modus registriert.

6.6 Auswertung der Ergebnisse

Die in Abschnitt 6.5 dargestellte Terminal-Ausgabe kombiniert die Ausgabe aller drei Geräte. Jede Zeile ist dabei als separate Ausgabe zu betrachten. Die erste Spalte gibt an, von welchem Gerät die Ausgabe stammt. Die zweite Spalte gibt an, zu welchem Zeitpunkt die Ausgabe getätigt worden ist. In der dritten Spalte wird die ID der Instanz ausgegeben. Diese stimmt mit der ID in der Konfigurationsdatei der jeweiligen ME2 überein. Sie ist also immer genau einem Gerät zugeordnet. Das Schlüsselwort'root' gibt an, dass die Meldung vom Teil des Codes der ME2 stammt, welcher die Hauptfunktionalität implementiert. Es könnte passieren, dass der CoAP-Server [BCS12], welcher von der ME2 zur Kommunikation verwendet wird, Fehlermeldungen ausgibt. In diesem Fall würde in der vierten Spalte 'coap' stehen. Die fünfte Spalte gibt das Log-Level der Ausgabe an. Da in der Konfigurationsdatei 'INFO' gewählt worden ist, werden keine 'DEBUG' Nachrichten angezeigt, welche zusätzliche Informationen liefern würden. In der letzten Spalte wird dann der eigentliche Inhalt geschrieben. Der dargestellte Ausschnitt der Terminal-Ausgabe wurde etwas später im Verlauf des Testfalls ausgewählt. Direkt beim Starten der verschiedenen Geräte werden üblicherweise viele Ausgaben getätigt, welche mit dem Hochfahren der ME2 und dem Auffinden der Nachbarn zusammenhängen.

```
{
  "name": "clean_temp_values",
  "process": "aggregate",
  "where": "111111111",
  "take": "temp_1",
  "give": "temp_3",
  "args": {
    "create": {
      "name": "temp_3",
      "param": [
        {
          "name": "result",
          "value": {
            "compute": {
                "v1": {
                    "calc": {
                        "func": "avg",
                        "var": {"from": "temp_1", "name": "result"}
                    } , "type": "float"},
                "v2": {"const":5.0, "type":"float"},
                "func": "sub"
            },
            "type": "float"
          }
        }
      ]
    },
    "windows": [
      {
        "name": "temp_1",
        "type": "length",
        "size": 5,
        "batch": true,
        "require_full": false
      }
    ]
  }
}
```

Listing 18: Erste Aggregations-Operation des Flows in der JSON Datei. Dies ist ein Ausschnitt der gesamten Datei in Listing 20.

```
{
"name": "check",
"process": "aggregate",
"where": "222222222",
"take": "temp_2, temp_3",
"give": "alarm",
"args": {
"create": {
      "name": "alarm",
      "param": []
    },
    "condition": {
      "logic": {
        "c1":{
          "comparison": {
            "comparator": ">",
            "v1": {"var": {"from": "temp_2", "name": "result"}, "type": "float"},
            "v2": {"const": 85.0, "type":"float"}
          }
        },
        "c2": {
          "logic": {
            "c1": {
              "comparison": {
                "comparator": ">",
                "v1": {"var": {"from": "temp_3", "name": "result"}, "type": "float"},
                "v2": {"const": 75.0, "type":"float"}
              }
            },
            "c2": {
              "comparison": {
                "comparator": ">",
                "v1": {"var": {"from": "temp_3", "name": "result"}, "type": "float"},
                "v2": {"var": {"from": "temp_2", "name": "result"}, "type": "float"}
              }
            },
            "operator": "||"
          }
        },
        "operator": "||"
      }
    },
    "windows": [
      {
        "name": "temp_2", "type": "length",
        "size": 1, "batch": false,
        "require_full": false
      },
      {
        "name": "temp_3", "type": "length",
        "size": 1, "batch": false,
        "require_full": false
} ] } }
```


Listing 19: Zweite Aggregations-Operation des Flows in der JSON Datei. Dies ist ein Ausschnitt der gesamten Datei in Listing 20.

```
{
    "name": "test_flow_usecase",
    "steps": [
      [
        {
          "name": "temp_sensor_normal_sauna",
          "input": "sensor_temperature",
          "where": "222222222",
          "give" : "temp_2"
        },
        {
          "name": "temp_sensor_steam_sauna",
          "input": "sensor_temperature",
          "where": "111111111",
          "give" : "temp_1"
        }
      ],
      [

        Listing 18
        {
          "name": "pass",
          "process": "pass",
          "where": "222222222",
          "take": "temp_2"
        }
      ],
      [
        Listing 19
      ],
      [
        {
          "name": "alarm_light",
          "output": "actuator_led",
          "where": "333333333",
          "take": "alarm"
        }
      ]
    ]
  }
```

Listing 20: flow.json Datei, welche den gesamten Flow darstellt. Die beiden Aggregationsoperation wurden der Übersichtlichkeit wegen ausgelagert.

```
1:  me2_first     | 11:23:57 - on:111111111 - root - INFO -   ----Read sensor data: {'result': 60.61902950423255}
2:  me2_first     | 11:23:57 - on:111111111 - root - INFO - START FLOW: 'test_flow_usecase' with message 'temp_1': {'result': 60.61902950423255}
3:  me2_first     | 11:23:57 - on:111111111 - root - INFO - EXECUTE OPERATION clean_temp_values(aggregate) with input 'temp_1: {"result": 60.61902...
4:  me2_first     | 11:23:57 - on:111111111 - root - INFO -       Stopping flow bacause: Aggregation condition was not statisfied or batch not full
5:  me2_first     | 11:23:57 - on:111111111 - root - INFO - CANCEL FLOW test_flow_usecase
6:  me2_first     |  ********************************************************************************
7:  me2_first     | 11:24:00 - on:111111111 - root - INFO -   ----Read sensor data: {'result': 40.038026222429814}
8:  me2_first     | 11:24:00 - on:111111111 - root - INFO - START FLOW: 'test_flow_usecase' with message 'temp_1': {'result': 40.038026222429814}
9:  me2_first     | 11:24:00 - on:111111111 - root - INFO - EXECUTE OPERATION clean_temp_values(aggregate) with input 'temp_1: {"result": 40.03802...
10:  me2_first    | 11:24:00 - on:111111111 - root - INFO -       Stopping flow bacause: Aggregation condition was not statisfied or batch not full
11:  me2_first    | 11:24:00 - on:111111111 - root - INFO - CANCEL FLOW test_flow_usecase
12:  me2_first    |  ********************************************************************************
13:  me2_first    | 11:24:03 - on:111111111 - root - INFO -   ----Read sensor data: {'result': 30.594027943412563}
14:  me2_first    | 11:24:03 - on:111111111 - root - INFO - START FLOW: 'test_flow_usecase' with message 'temp_1': {'result': 30.594027943412563}
15:  me2_first    | 11:24:03 - on:111111111 - root - INFO - EXECUTE OPERATION clean_temp_values(aggregate) with input 'temp_1: {"result": 30.59402...
16:  me2_first    | 11:24:03 - on:111111111 - root - INFO -       Stopping flow bacause: Aggregation condition was not statisfied or batch not full
17:  me2_first    | 11:24:03 - on:111111111 - root - INFO - CANCEL FLOW test_flow_usecase
18:  me2_first    |  ********************************************************************************
19:  me2_second   | 11:24:05 - on:222222222 - root - INFO -   ----Read sensor data: {'result': 88.73168017329405}
20:  me2_second   | 11:24:05 - on:222222222 - root - INFO - START FLOW: 'test_flow_usecase' with message 'temp_2': {'result': 88.73168017329405}
21:  me2_second   | 11:24:05 - on:222222222 - root - INFO - EXECUTE OPERATION pass(pass) with input 'temp_2: {"result": 88.73168017329405}'
22:  me2_second   | 11:24:05 - on:222222222 - root - INFO - EXECUTE OPERATION check(aggregate) with input 'temp_2: {"result": 88.73168017329405}'
23:  me2_second   | 11:24:05 - on:222222222 - root - INFO -       Evaluated condition to true
24:  me2_second   | 11:24:05 - on:222222222 - root - INFO -       Created event 'alarm' with payload {}
25:  me2_second   | 11:24:05 - on:222222222 - root - INFO - SUCCESS Sending FlowMessage to 10.5.0.6:5683/flow-messages
26:  me2_third    | 11:24:05 - on:333333333 - root - INFO - EXECUTE OPERATION alarm_light(None) with input 'alarm: "{}"'
27:  me2_third    | 11:24:05 - on:333333333 - root - INFO -       Started 1 actuators
28:  me2_third    | 11:24:05 - on:333333333 - root - INFO -       Dummy led: True
29:  me2_third    | 11:24:05 - on:333333333 - root - INFO -       Triggered 1 actuators of type LED
30:  me2_third    | 11:24:05 - on:333333333 - root - INFO - FINISHED FLOW test_flow_usecase:3137f503b1314110b0822935db44eb42, result: "{}"
31:  me2_third    |  ********************************************************************************
32:  me2_first    | 11:24:06 - on:111111111 - root - INFO -   ----Read sensor data: {'result': 20.918072182442273}
33:  me2_first    | 11:24:06 - on:111111111 - root - INFO - START FLOW: 'test_flow_usecase' with message 'temp_1': {'result': 20.918072182442273}
34:  me2_first    | 11:24:06 - on:111111111 - root - INFO - EXECUTE OPERATION clean_temp_values(aggregate) with input 'temp_1: {"result": 20.9180...
35:  me2_first    | 11:24:06 - on:111111111 - root - INFO -       Stopping flow bacause: Aggregation condition was not statisfied or batch not full
36:  me2_first    | 11:24:06 - on:111111111 - root - INFO - CANCEL FLOW test_flow_usecase
37:  me2_first    |  ********************************************************************************
38:  me2_first    | 11:24:09 - on:111111111 - root - INFO -   ----Read sensor data: {'result': 83.93123838566862}
39:  me2_first    | 11:24:09 - on:111111111 - root - INFO - START FLOW: 'test_flow_usecase' with message 'temp_1': {'result': 83.93123838566862}
40:  me2_first    | 11:24:09 - on:111111111 - root - INFO - EXECUTE OPERATION clean_temp_values(aggregate) with input 'temp_1: {"result": 83.931...
41:  me2_first    | 11:24:09 - on:111111111 - root - INFO -       Evaluated condition to true
42:  me2_first    | 11:24:09 - on:111111111 - root - INFO -       Created event 'temp_3' with payload {'result': 42.22007884763717}
43:  me2_first    | 11:24:09 - on:111111111 - root - INFO - SUCCESS Sending FlowMessage to 10.5.0.4:5683/flow-messages
44:  me2_second   | 11:24:09 - on:222222222 - root - INFO - EXECUTE OPERATION check(aggregate) with input 'temp_3: "{'result': 42.22007884763717}"'
45:  me2_second   | 11:24:09 - on:222222222 - root - INFO -       Evaluated condition to true
46:  me2_second   | 11:24:09 - on:222222222 - root - INFO -       Created event 'alarm' with payload {}
47:  me2_second   | 11:24:09 - on:222222222 - root - INFO - SUCCESS Sending FlowMessage to 10.5.0.6:5683/flow-messages
48:  me2_third    | 11:24:09 - on:333333333 - root - INFO - EXECUTE OPERATION alarm_light(None) with input 'alarm: "{}"'
49:  me2_third    | 11:24:09 - on:333333333 - root - INFO -       Started 1 actuators
50:  me2_third    | 11:24:09 - on:333333333 - root - INFO -       Dummy led: True
51:  me2_third    | 11:24:09 - on:333333333 - root - INFO -       Triggered 1 actuators of type LED
52:  me2_third    | 11:24:09 - on:333333333 - root - INFO - FINISHED FLOW test_flow_usecase:9272b8cafcb64fe09d5e0908f755a22e, result: "{}"
53:  me2_third    |  ********************************************************************************
```

Listing 21: Terminal-Ausgabe des Anwendungsfalls mit Docker. Dies stellt sämtliche Terminal-Ausgaben von mehreren virtuellen IoT-Geräten in einem kurzen Zeitausschnitt dar.

In Zeile 1 wird, auf Gerät 1, von Sensor-Listener eine Sensor-Messung empfangen. Das Gerät startet dann den angegebenen Flow mit der generierten Nachricht 'temp_1' welche als Parameter 'result' mit dem Wert von 60.6 °C erhält (Zeile 2). Daraufhin wird die Aggregation durchgeführt (Z.3). In diesem Schritt sollte immer der Durchschnittswert von 5 Messungen genommen werden und die fehlerhaften 5 °C ausgeglichen werden. Allerdings ist zu diesem Zeitpunkt erst eine der fünf Messungen erfolgt. Darum wird der Flow mit der Begründung 'batch not full' abgebrochen. (Z.6-18) Dieses Szenario wiederholt sich noch zweimal mit den Werten 40 °C und 31 °C (Z.4-5).

Anschließend misst Gerät 2 eine Temperatur von 88.7 °C (Z.19). Es startet den Flow mit der Nachricht 'temp_2' (Z.20). Daraufhin wird diese Nachricht an den nächsten Schritt weitergeleitet. Dies ist in unserem Flow die Operation 'pass', welche die Nachricht sofort weiterleitet

(Z.21). Die darauffolgende Operation ist die Aggregation, welche bestimmen soll, ob ein Alarm ausgelöst werden soll (Z.22). Da die Temperatur von 88.7°C die maximale Temperatur von 85°C übersteigt, wird ein Alarm-Event erzeugt (Z.23-24). Dieses wird nun an Gerät 3 gesendet (Z.25). Gerät 3 startet nun die Operation 'alarm_light' welche die LED zum Leuchten bringt (Z.26-29). Dies beendet den Flow (Z.30-31).

Daraufhin bekommt Gerät 1 wieder eine Temperaturmessung von 21 °C, welche nun insgesamt die vierte Messung ist, also immer noch nicht ausgewertet werden kann (Z.32-37). Die nächste Messung von Gerät 1 liefert eine Temperatur von 84 °C und macht damit das Fenster voll (Z.38-39). Die Aggregationsbedingung ist automatisch erfüllt, da keine Bedingung angegeben wurde (Z.40-41). Der Durchschnittswert der Werte: [60.6, 40.0, 30.6, 20.9, 83.9] liegt bei etwa 47,2 (Z.42). Dieser wird um 5 °C korrigiert und man erhält die Temperatur von 42,2 °C. Das bereinigte Messergebnis wird nun an Gerät 2 gesendet (Z.43). Dieser Temperaturwert sollte keinen Alarm auslösen (Z.44-45). Der letzte Messwert von Gerät 2, welcher sich noch nicht geändert hat, ist allerdings immer noch zu hoch. Da immer alle Bedingungen ausgewertet werden, wird die LED so lange weiter blinken, bis keine der Bedingungen für einen Alarm mehr erfüllt ist (Z.46-53).

Dies war nur ein sehr kurzer Ausschnitt der Terminal-Ausgaben. Von der ersten bis zur letzten Nachricht vergingen gerade einmal 12 Sekunden. Ein dauerhaft laufendes Programm würde also Unmengen an Text generieren. In dem wirklichen Szenario sind wir allerdings nicht an den Terminal-Ausgaben interessiert, sondern nur daran, dass die LED blinkt, sobald fehlerhafte Temperaturen auftreten. Setzt man das LOG-Level auf eine höhere Ebene, so wird im Normalfall kaum ein Text generiert.

6.7 Fazit

Dieses Anwendungsszenario demonstriert gut, welche vielfältigen Möglichkeiten sich mit der neuen Aggregationsmethode der ME2 ergeben. So lässt sich durch das Konfigurieren der flow.json Datei auf sehr viele diverse Probleme reagieren. Es fällt auch auf, dass durch die vielen gebotenen Möglichkeiten sich der Aufwand um einen eigenen Flow zu erstellen stark erhöht. Dieser Nachteil wird aber im Hinblick auf künftige Arbeiten am IAMT nicht mehr relevant sein. Durch übersichtlich gestaltete Interfaces soll die Erstellung eines Modells auch für unerfahrene Nutzer einfach und schnell möglich sein.

7 Zusammenfassung

In dieser Arbeit wurde die ME2 so erweitert, dass es ihr nun möglich ist, in deutlich komplexeren IoT-Anwendungen als zuvor, effektiv Sensordatenströme in Zusammenarbeit mit anderen ME2s zu verarbeiten. Dabei wurde auch das Hauptziel dieser Arbeit erfüllt, welches es erforderte, eine geeignete Auswahl an Aggregationstechniken für Sensordatenströme in der ME2 zu implementieren.

Dazu wurde die ME2 so optimiert, dass sie in der Lage ist komplexere Datenströme zu verarbeiten, in welchen die Datenströme nicht mehr linear von einem Sensor zu einem Aktivator durch eine Reihe von ME2s geleitet werden müssen. Stattdessen ist es jetzt möglich, mithilfe von ME2s mehrere verzweigte und teils überlappende Sensordatenströme zu realisieren. Dies erlaubt es einem Anwender, beinahe alle möglichen Konfigurationen von IoT-Netzwerken zu erstellen.

In diese nun deutlich potenteren Sensordatenströme wurde eine Möglichkeit implementiert, um die Daten der Sensordatenströme zusammen aggregieren zu können. Diese Aggregation macht davon Gebrauch, dass in den veränderten Sensordatenströmen die Daten von mehreren Sensoren zueinander in Beziehung gesetzt werden können. Um diese Aggregationsoperation möglichst vielfältig konfigurierbar und einsetzbar zu gestalten, ist es möglich, deren Funktionsweise mithilfe von Ereignisregeln, welche aus dem Bereich des CEP stammen, festzulegen. Um diese Ereignisregeln möglichst verständlich, in einer effizient zu verarbeitenden Form und in den Flow der ME2 leicht zu integrierenden Art zu notieren, wurde ein eigenes Sprachkonstrukt auf Basis von JSON entwickelt. Dieses Sprachkonstrukt ermöglicht es, die Aggregationsoperationen so zu konfigurieren, dass festgelegt werden kann, unter welchen Bedingungen eine neue Nachricht erstellt werden kann. Es kann bestimmt werden, welche Informationen in einer neu generierten Nachricht enthalten sein sollen und wie diese benannt werden soll. Außerdem kann eine große Auswahl an Berechnungen durchgeführt werden und miteinander kombiniert werden, um auch komplexere Auswertungen zu ermöglichen. Dem Anwender steht dabei auch offen, auf welcher Menge an Nachrichten Aggregationen durchgeführt werden sollen, indem er die Fenstergrößen von Zeitfenstern oder Längenfenstern festlegt. Diese hohe Flexibilität ermöglicht unbegrenzt viele Nutzungsszenarien, wie die Aggregationsoperation eingesetzt werden kann. Dies kommt allerdings mit dem Nachteil einer eher aufwendigeren Bedienung daher.

Die Möglichkeiten, welche sich nun durch die modifizierte ME2 ergeben, wurden in einem Anwendungsbeispiel dargestellt. In diesem Beispiel wurde der nötige Bedienungsaufwand,

aber auch die Flexibilität der ME2 geeignet dargestellt. Anschließend wurde demonstriert, wie mehrere ME2s in diesem Szenario im Zusammenspiel die Anforderungen erfüllten.

Um diese Verbesserungen der ME2 für die Anwendungsmodellierung im IAMT verfügbar zu machen, sollte die ME2 als Operator auf der MBP registriert werden. Dieser Operator konnte allerdings nie ganz getestet werden, da viele Probleme bei der Verwendung der MBP auftraten. Dadurch war es auch nicht möglich, die ME2 in einer brauchbaren Weise in das IAMT zu integrieren. Stattdessen wurde ein Konzept erläutert, wie die ME2 sich ideal in das IAMT integrieren lassen könnte, sobald alle dafür nötigen Komponenten voll funktionsfähig sind.

8 Ausblick

Die in dieser Arbeit entwickelten Möglichkeiten zur Aggregation von Sensordatenströmen mithilfe der ME2 ermöglichen ein breites Anwendungsspektrum in den verschiedensten IoT-Netzwerken. Um allerdings dieses Potenzial auszuschöpfen und dabei das Modellierungstool des IAMT zu verwenden, gibt es noch einige Aufgaben, die bearbeitet werden sollten, bevor dieses Tool wirklich sinnvoll nutzbar wird. Dies beinhaltet Optimierungen an der ME2, Reparaturen und Anpassungen an der MBP sowie neue Funktionalitäten im IAMT.

8.1 Optimierungen an der ME2

Um die ME2 in einer tatsächlichen Anwendung über länger Zeiträume zu verwenden, wäre es nötig Effizienzsteigerungen vorzunehmen, welche es ermöglichen, die ME2 ressourcenschonend einzusetzen.

8.1.1 Flows nicht bei jeder Nachricht traversieren

Die gegebene Implementierung der ME2 erfordert es, dass bei jeder empfangenen Nachricht der aktuelle Flow komplett neu verarbeitet wird. Ein effizienteres System, welches den Flow nur beim ersten Empfang verarbeitet und dann die relevanten Teile so speichert, dass dieser Flow bearbeitet werden kann ohne die Flow-Datei neu traversieren zu müssen, wäre günstiger.

8.1.2 Interne Nachrichten nicht über Queues weiterleiten

In der gegebenen Implementierung werden alle versendeten Nachrichten in eine Push-Queue geschoben. Auch wenn Nachrichten geräte-intern verarbeitet werden, werden sie erst in die Push-Queue geschoben und direkt anschließend der Pull-Queue entnommen. Dieser Ablauf ermöglicht es zwar, dass der Code sehr konsistent ist, führt aber einen nicht zu vernachlässigenden zusätzlichen Aufwand ein. Rein interne Nachrichten könnten auch direkt in der nächsten Operation verarbeitet werden und dann erst weiter geschickt werden. Durch eine solche Modifikation würde auch die neu eingeführte Pass-Operation der ME2 kaum zusätzlichen Rechenaufwand kosten.

8.1.3 Zusätzliche Aggregationsmöglichkeiten

In dieser Arbeit wurden viele der Möglichkeiten, welche das CEP zu Verarbeitung von Datenströmen bietet, implementiert. Dabei wurden solche Aggregationsmöglichkeiten gewählt, welche als sinnvoll für die Einsatzgebiete der ME2 erachtet wurden. Um aber noch ein breiteres Anwendungsspektrum bieten zu können, wären noch einige zusätzlich Funktionen denkbar. Dazu zählt das Erkennen von Ereignismustern, also zu erkennen, ob die empfangenen Nachrichten einer vordefinierten Reihenfolge entsprechen. Um komplexere Entscheidungen treffen zu können, welche nicht nur von Sensormessungen abhängen, wäre es denkbar, die Möglichkeit hinzuzufügen, externes Wissen (etwa aus dem Internet) in die Aggregation einfließen zu lassen. Außerdem lassen sich noch einige mathematische Operationen hinzufügen, wie z. B. die Steigung von Messwerten zu berechnen.

8.2 Veränderung an der MBP

In dieser Arbeit ist es nicht gelungen, einen Operator von der MBP aus zu installieren und zu starten. Es ist davon auszugehen, dass dies an Fehlern der MBP lag. Diese Fehler sollten auf jeden Fall in zukünftigen Arbeiten behoben werden.

Um bei der Konfiguration der Operatoren der MBP noch mehr Möglichkeiten zu bieten, ist es erforderlich komplexere Parameter für Operatoren zu ermöglichen. Eine Möglichkeit wäre es, Objekte mit mehreren Attributen als Parameter zu ermöglichen. Dies ließe sich zum Beispiel mit Objekten im JSON-Format realisieren. Um diese dann über die SSH-Schnittstelle übermitteln zu können, sollten diese beim Senden kodiert werden.

8.3 Veränderung am IAMT

Da es in dieser Arbeit nicht gelang, eine Integration der ME2 in das IAMT durchzuführen, bleibt dies die Aufgabe zukünftiger Arbeiten.

8.3.1 Anpassung des Modells

Um die Nutzung der ME2 geeignet im Anwendungsmodell modellieren zu können, muss dessen Struktur grundsätzlich geändert werden. Dabei sollte unterschieden werden, zwischen Operatoren, welche real auf den verfügbaren Geräten installiert werden können und Operationen, welche auf den verschiedenen ME2s ausgeführt werden können. Ein dazu mögliches Konzept wurde in dieser Arbeit in Abschnitt 4.3.1 vorgestellt. Dieses überarbeitete Modell muss dann auf geeignete Weise an die beteiligten ME2s übermittelt werden.

8.3.2 Interface zur benutzerfreundlichen Erstellung der Flow-JSON

Um dem Nutzer des IAMT die Verwendung der Aggregationsoperation der ME2 möglichst angenehm und einfach zu gestalten, ist es erforderlich, ein dafür zugeschnittenes Interface zu entwerfen. Über dieses Interface sollte es dem Nutzer ermöglicht werden, möglichst intuitiv und fehlerresistent Ereignisregeln für die Aggregationsoperation zu definieren. Diese Eingaben müssen dann in einem zweiten Schritt in das benötigte JSON-Format der ME2 umgewandelt werden. Auch hierzu wurde in Abschnitt 4.3.3 ein Konzept vorgestellt.

8.3.3 Einfacheres Einrichten der Software

Da die Verwendung der ME2 über das IAMT drei verschiedene Softwarekomponenten erfordert, wird dies für einen Nutzer schnell zu einem hohen Konfigurationsaufwand. Um die Nutzerfreundlichkeit zu erhöhen, wäre es angebracht eine Installationsmöglichkeit zu schaffen, welche die MBP und das IAMT zusammen installiert und verknüpft, sowie die ME2 gleich als Operator registriert.

Literaturverzeichnis

[ACL18] T. Akidau, S. Chernyak, R. Lax. *Streaming systems: the what, where, when, and how of large-scale data processing*. Ö'Reilly Media, Inc.", 2018 (zitiert auf S. 43).

[AIM10] L. Atzori, A. Iera, G. Morabito. „The internet of things: A survey". In: *Computer networks* 54.15 (2010), S. 2787–2805 (zitiert auf S. 15).

[BBSS01] D. J. Barrett, D. J. Barrett, R. E. Silverman, R. Silverman. *SSH, the Secure Shell: the definitive guide*. Ö'Reilly Media, Inc.", 2001 (zitiert auf S. 46).

[BCS12] C. Bormann, A. P. Castellani, Z. Shelby. „Coap: An application protocol for billions of tiny internet nodes". In: *IEEE Internet Computing* 16.2 (2012), S. 62–67 (zitiert auf S. 71).

[BD15] R. Bruns, J. Dunkel. *Complex event processing: komplexe Analyse von massiven Datenströmen mit CEP*. Springer-Verlag, 2015 (zitiert auf S. 13, 16–19, 33, 34, 39, 47).

[CFS+14] C. Y. Chen, J. H. Fu, T. Sung, P.-F. Wang, E. Jou, M.-W. Feng. „Complex event processing for the Internet of Things and its applications". In: *2014 IEEE International Conference on Automation Science and Engineering (CASE)*. 2014, S. 1144–1149. DOI: 10.1109/CoASE.2014.6899470 (zitiert auf S. 13, 23).

[CPD+16] B. Costa, P. F. Pires, F. C. Delicato, W. Li, A. Y. Zomaya. „Design and Analysis of IoT Applications: A Model-Driven Approach". In: *2016 IEEE 14th Intl Conf on Dependable, Autonomic and Secure Computing, 14th Intl Conf on Pervasive Intelligence and Computing, 2nd Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress(DASC/PiCom/DataCom/CyberSciTech)*. 2016, S. 392–399. DOI: 10.1109/DASC-PICom-DataCom-CyberSciTec.2016.81 (zitiert auf S. 16).

[DH20] D. Del Gaudio, P. Hirmer. „A lightweight messaging engine for decentralized data processing in the Internet of Things". In: *SICS Software-Intensive Cyber-Physical Systems* 35.1-2 (2020), S. 39–48 (zitiert auf S. 13).

[FHPM18] A. C. Franco da Silva, P. Hirmer, R. K. Peres, B. Mitschang. „An Approach for CEP Query Shipping to Support Distributed IoT Environments". In: *2018 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*. 2018, S. 247–252. DOI: 10.1109/PERCOMW.2018.8480241 (zitiert auf S. 24).

[GBB18] P. Gokhale, O. Bhat, S. Bhat. „Introduction to IOT“. In: *International Advanced Research Journal in Science, Engineering and Technology* 5.1 (2018), S. 41–44 (zitiert auf S. 13, 15).

[Glo11] E. P. Glossary–Version. „Event Processing Technical Society“. In: (2011) (zitiert auf S. 17).

[HLR17] M. Hussein, S. Li, A. Radermacher. „Model-driven development of adaptive IoT systems.“ In: *MODELS (Satellite Events)*. 2017, S. 17–23 (zitiert auf S. 13, 16).

[KL09] A. Klein, W. Lehner. „How to optimize the quality of sensor data streams“. In: *2009 Fourth International Multi-Conference on Computing in the Global Information Technology*. IEEE. 2009, S. 13–19 (zitiert auf S. 16).

[KMD+18] T. Kohler, R. Mayer, F. Dürr, M. Maaß, S. Bhowmik, K. Rothermel. „P4CEP: Towards In-Network Complex Event Processing“. In: *Proceedings of the 2018 Morning Workshop on In-Network Computing*. NetCompute '18. Budapest, Hungary: Association for Computing Machinery, 2018, S. 33–38. ISBN: 9781450359085. DOI: 10.1145/3229591.3229593. URL: https://doi.org/10.1145/3229591.3229593 (zitiert auf S. 23).

[LL12] T. Liu, D. Lu. „The application and development of IoT“. In: *2012 International symposium on information technologies in medicine and education*. Bd. 2. IEEE. 2012, S. 991–994 (zitiert auf S. 13).

[Luc02] D. Luckham. *The power of events*. Bd. 204. Addison-Wesley Reading, 2002 (zitiert auf S. 16).

[MLLL+15] S. Madakam, V. Lake, V. Lake, V. Lake et al. „Internet of Things (IoT): A literature review“. In: *Journal of Computer and Communications* 3.05 (2015), S. 164 (zitiert auf S. 13).

[Nit15] S. Nittel. „Real-time sensor data streams“. In: *Sigspatial Special* 7.2 (2015), S. 22–28 (zitiert auf S. 16).

[NTBG15] X. T. Nguyen, H. T. Tran, H. Baraki, K. Geihs. „FRASAD: A framework for model-driven IoT Application Development“. In: *2015 IEEE 2nd world forum on internet of things (WF-IoT)*. IEEE. 2015, S. 387–392 (zitiert auf S. 13).

[RBS21] A. M. Rahmani, Z. Babaei, A. Souri. „Event-driven IoT architecture for data analysis of reliable healthcare application using complex event processing“. In: *Cluster Computing* 24 (2021), S. 1347–1360 (zitiert auf S. 13).

[Sch] T. Schneider. „Verarbeitung komplexer loT-Daten in der loT-Plattform MBP“. Bachelor's Thesis, S. 19 (zitiert auf S. 20).

[SHS+20] A. C. F. da Silva, P. Hirmer, J. Schneider, S. Ulusal, M. T. Frigo. „MBP: Not just an IoT platform“. In: *2020 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*. IEEE. 2020, S. 1–3 (zitiert auf S. 13, 20).

[Tar12] S. Tarkoma. *Publish/subscribe systems: design and principles*. John Wiley & Sons, 2012 (zitiert auf S. 29).

Alle URLs wurden zuletzt am 28. 03. 2023 geprüft.

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Ort, Datum, Unterschrift