UNIVERSITÄT DUISBURG-ESSEN
FAKULTÄT FÜR INGENIEURWISSENSCHAFTEN
ABTEILUNG INFORMATIK UND ANGEWANDTE KOGNITIONSWISSENSCHAFT

Bachelorarbeit

An Algorithm for Dependency-Preserving Smart Home Updates

Koray Uzun
Matrikelnummer:3081690
Angewandte Informatik (Bachelor)

Fachgebiet Verteilte Systeme, Abteilung Informatik
Fakultät für Ingenieurwissenschaften
Universität Duisburg-Essen

5. September 2021

Erstgutachter: Prof. Dr-Ing. Torben Weis
Zweitgutachter: Peter Zdankin
Zeitraum: 1. September 2042 - 1. Januar 2043

Abstract

Smart Home Systeme sind Netzwerke elektronischer Geräte, welche die Funktion haben die Wohn- und Lebensqualität ihrer Nutzer im eigenen Heim zu erhöhen. Diese Systeme unterliegen, wie jede andere Technologie auch, einem ständigen Wandel. Während jedoch viele elektronische Geräte, seien es Smartphones, Tastaturen oder auch Kopfhörer meist nach einigen wenigen Jahren ausgetauscht werden, ist die Erwartungshaltung gegenbüber Smart Home Systemen eine andere. Aufgrund der hohen Anschaffungskosten erwarten Käufer eine hohe Lebensdauer. Gleichzeitig müssen sich die Systeme den Gegebenheiten der Zeit anpassen. Neue Funktionen oder oft auch Sicherheitslücken erfordern ein ständiges updaten der einzelnen Geräte in einem System. Mit jedem Update geht jedoch ein großes Risiko einher, da Hersteller nicht garantieren können, dass ein Update keine Schäden im System anrichtet. Dies liegt daran, dass Updates lediglich für das betroffene Gerät auf mögliche Fehlerquellen überprüft werden, nicht jedoch das Zusammenspiel des Updates mit anderen Geräten. Aufgrund der Diversität an Smart Home Systemen und Geräten wäre dies für den Hersteller gar nicht umsetzbar. Daher bedarf es einer Möglichkeit Updates präventiv zu überprüfen, um so größere Schäden zu meiden und die Langlebigkeit von Smart Home Systemen zu gewährleisten zu können. Diese Arbeit beschäftigt sich mit der Implementation und Evaluation eines Algrorithmus, welcher Abhängigkeiten zwischen Geräten untersucht und basierend auf diesen Abhängigkeiten Updatekonfigurationen ermittelt, welchen den größten Nutzen für den Nutzer haben. Der Algorithmus basiert auf dem Paper "An Algorithm for Dependency-Preserving Smart Home Updates" von Peter Zdanking, Matthias Schaffeld, Marian Waltereit, Oskar Carl und Torben Weis.

1

[1] Wikipedia: `https://en.wikipedia.org/wiki/Abstract_(summary)`

Contents

Chapter 1

Einführung

In den letzten Jahren stieg die Anzahl der Smart Home Geräte stark. Laut Verified Market Research wird der gloable Smart Home Markt im Jahr 2019 auf bereits 80 Milliarden Dollar geschätzt. Prognostiziert werden 207.88 Milliarden Dollar im Jahr 2027. Dies entspricht einer jährlichen Wachstumsrate von 13.52 Prozent [2]. Zudem ist es so, dass innerhalb von Smart Home Systemen die Anzahl der Geräte ebenfalls stetig steigt. Während 2016 durchschnittlich 5.8 Geräte über ein Smart Home System vernetzt waren, waren es 2018 schon bereits 8.1 Geräte [3]. Es ist zu vermuten, dass sich dieser Trend vorerst so weiterentwickeln wird und somit Smart Home Systeme immer größer werden. Wie jede andere Technologie müssen Smart Home Systeme regelmäßig geupdatet werden, um zum Beispiel Sicherheitslücken zu schließen oder neue Funktionalitäten hinzuzufügen. Mit der steigenden Anzahl an Geräten, steigt in einem Smart Home System auch logischerweise die Anzahl der Updates und somit insgesamt die Komplexität von Smart Home Systemen. Das Problem hierbei ist, dass mit steigender Komplexität zum einen automatisch eine höhere Fehleranfälligkeit bezüglich Updates einhergeht, zum anderen wird es schwieriger Fehler zu beheben, die durch Updates verursacht werden. Updates im Einzelnen besitzen bereits viele Fehlerquellen, wie zum Beispiel Sicherheitslücken oder generell fehlerhafte Implementationen. Dies sind jedoch Aspekte die bereits vom Entwickler weitestgehend vermieden werden. Viel Problematischer ist das Zusammspiel verschiedener Geräte und den dazugehörigen Updates. Ein an sich fehlerfreies Update kann bei der Installtion in einem Smart Home System, trotz "fehlerfreier" Programmierung große Probleme verursachen. Was ist wenn zum Beispiel ein Update eine Funktion eines Geräts so verändert, dass ein anderes Gerät nicht mehr darauf zugreifen kann. Durch solche kleine Änderungen können Abhängigkeiten zwischen Geräten zerstört werden, wodurch ganze Teile eines Smart Home Systems ausfallen können. Solch ein Ausfall bedeutet für den Nutzer meist ein großer finanzieller Schaden, da Updates oft nicht rückgängig gemacht werden können und somit Ersatz beschafft werden muss. Aus diesen Gründen sind präventive Maßnahmen notwendig, um die Langlebigkeit von Smart Home Systemen gewährleisten zu können. Langlebigkeit wird nämlich von den meisten Usern erwartet, wie man an der Resonanz zum Abschalten der HueBridge V1 von Philips sehen kann [4].

1.1 Ziel der Arbeit

Eine Steigerung der Lebensqualität ist das Kernziel von Smart Home Systemen. Dies erreicht man zum Beispiel durch die Konfiguration von automatisierten Prozessen. Dadurch können Nutzer nämlich Zeit, Energie und in gewissen Belangen auch Geld sparen. Laut Umfragen geben 57 Prozent der Nutzer von Smart Home Systemen in Amerika an jeden Tag durchschnittlich 30 Minuten Zeit einzusparen [4]. Zusätzlich lassen sich nach Berechnungen des Fraunhofer Instituts für Bauphysik durch Automatisierungen bis zu ca. 40 Prozent Heizkosten sparen. [5]. Solche Vorteile zeigen, dass sich die anfangs teure Anschaffung eines Smart Home Systemes auf lange Zeit sogar rentieren kann. Sicherlich ist dies der Idealfall, aber dennoch zeigen diese Zahlen, dass Smart Home Systeme viele Potenziale bieten. Es hängt jedoch von vielen Faktoren ab, ob sich die Anschaffung eines Smart Homes lohnt und objektiv ist dies nicht zu beurteilen. Was jedoch objektiv gesagt werden kann, ist dass in den meisten Fällen Nutzer aufgrund der teuren Anschaffungskosten eines Smart Homes davon ausgehen, dass ihr System viele Jahre erhalten bleibt. Zum Beispiel gibt es Smart Home Systeme, welche beim Bau eines Hauses direkt mit geplant werden, sodass man, wie zum Beispiel bei KNX Systemen alle Geräte über ein permanent installiertes Bus System verbinden kann [1]. Selbstverständlich gibt es auch günstigere Varianten, aber dennoch ist die generelle Erwartungshaltung, dass wenn man ein funktionierendes System hat, dieses auch über viele Jahre hinweg weiter funktionieren wird. Daher is das Ziel dieser Arbeit die Implementation und Evaluation eines Algrorithmus, welcher Abhängigkeiten zwischen Geräten untersucht und basierend auf diesen Abhängigkeiten Updatekonfigurationen ermittelt, welchen den größten Nutzen für den Nutzer haben. Mit diesem Algorithmus wäre es für Smart Home Nutzer möglich Updates bereits vor der Installation zu überprüfen, um so sicherzustellen, dass sie keine Ausfälle verursachen. Die Anwendung dieses Algorithmus sollte für den Nutzer möglichst einfach sein. So könnte man zum Beispiel ein Smartphone nutzen, welches als zentrales Gerät im System dient. Auf diesem zentralen Gerät wäre es dann möglich den Algorithmus laufen zu lassen und sich über das Smartphone dann für die passende Updatekonfiguration zu entscheiden. Smartphones bieten sich besonders an, da sie eine intuitive Bedienung für den Nutzer ermöglichen und im Normalfall jeder Nutzer eines Smart Homes ein Smartphone besitzt. So umgeht man das Problem, dass es, wie bereits erwähnt, etliche Smart Home Anbieter mit verschiedener Software und Geräten gibt.

Chapter 2

Related Work

In diesem Kapitel wird das Thema Langlebigkeit im Bereich Software Engineering thematisiert. Infolgedessen werden Smart Home Systeme unter dem Aspekt der Langlebigkeit genauer beleuchtet und das Thema Dependency Management aufgegriffen, wobei bereits bestehende Ansätze/Lösungen diesbezüglich genauer betrachtet. Zusätzlich wird der aktuelle Stand des Dependency Managements im Bereich Smart Home untersucht.

2.1 Smart Homes

Es existieren viele verschiedene Anbieter mit verschiedenen zugrundle liegenden Architekturen für Smart Home Systeme . Im Rahmen dieser Arbeit ist es jedoch nicht notwendig die verschiedenen Smart Home Architekturen genauer zu beleuchten. Dennoch schadet es nicht sich einen groben Überblick über die aktuellen Technologien zu verschaffen. Grob kann man die verschiedenen Architekturen darin unterscheiden, ob sie eine Verknüpfung zu einer Cloud besitzen. Cloud gebundene Architekturen bieten dem Nutzer meist eine einfachere Installation und Wartung, indem zum Beispiel Updates automatisch installiert werden.Gleichzeitig bringen sie aber auch potenzielle Gefahren mit sich. Die Abhängigkeit von einer Cloud macht Smart Home Systeme sehr verwundbar. Wird ein externer Service, wie zum Beispiel die Cloud abgeschaltet, kann es zu starken Ausfällen im System führen. [5](Towards Longevity of Smart Homes). Des Weiteren sind Geräte, die mit einer Cloud verbunden sind anfälliger gegenüber Angriffen von Außenstehenden, die sich Zugriff auf das System beschaffen wollen. Nicht Cloudgebundene Architekturen haben diese Nachteile nicht. Es ist jedoch meist schwieriger eine solche Architektur aufzubauen, da sie zum Beispiel, wie im Falle von OpenHab[6], viel manuelle Konfiguration benötigt. Zusätzlich bieten nicht Cloud-gebundene Architekturen mehr Sicherheit, da sie keine starken Abhängigkeiten zu anderen Dienstleistern benötigen. Natürlich könnte man Smart Home Systeme noch weiter spezifizieren, indem man einzelne Komponenten genauer beleuchet und verschiedene Smart Home Anbieter miteinander vergleicht, jedoch ist dies, wie bereits erwähnt, im Rahmen dieser

Arbeit nicht notwendig. Der zu entwickelnde Algorithmus soll nämlich später theoretisch Platform- und Architekturunabhängig anwendbar sein. Zu diesem Zweck werden im Folgenden einige Aspekte bezüglich Smart Homes definiert und vorausgesetzt. [Paper 12]

Im Rahmen dieser Arbeit soll die ausgewählte Architektur keinen Einfluss auf den zu implementierenden Algorithmus haben. Daher wird die für den Algorithmus genutzte Architektur möglichst simpel gehalten.
A smart home has a set of devices that are connected through a platform. Each device has a certain software version and a set of available updates. Each software version has a set of available predefined services. Devices can use services of other devices which creates dependencies. An update configuration is one of the finite states of the nondeterministic finite automaton (NFA) that can be constructed by using the configurations as states and connecting them using individual updates as transitions.[4] Welche Geräte solch eine Architektur beinhaltet oder durch was für eine Platform genau die Geräte miteinander verbunden sind spielt zunächst keine Rolle. Wichtig sind nur die genannten Komponenten:

- Device (Gerät): Geräte sind die Komponenten, die die eigentliche Funktionalität anbieten. Sie besitzen eine aktuelle Firmware Version mit entsprechenden Diensleistungen, potenzielle Updates und Abhängigkeiten zu anderen Geräten, indem sie auf deren Dienstleistungen zugreift.
- Firmware Version: Die Firmware Version gibt an, welche Dienstleistungen ein Gerät anbietet
- Service (Dienstleistungen): Eine Dienstleistung ist eine Funktionalität, die ein Gerät anbietet. Zum Beispiel bietet ein Thermostat die Dienstleistungen "Temperatur messen" an.
- Update: Eine Aktualisierung der Firmware eines Geräts. Durch ein Update können sich angebotenen Diensleistungen eines Geräts ändern.
- Platform: Die Komponente, die alle Geräte miteinander vernetzt.
- Dependency (Abhängigkeiten): Eine Abhängigkeit zwischen zwei Geräten besteht, wenn ein Gerät Zugriff auf die Dienstleistungen eines anderen Geräts benötigt.
- Update Configuration: Eine Updatekonfiguration beschreibt die aktuelle Firmware aller Geräte in einem System. Sie bietet also eine Art Übersicht aller Geräte.

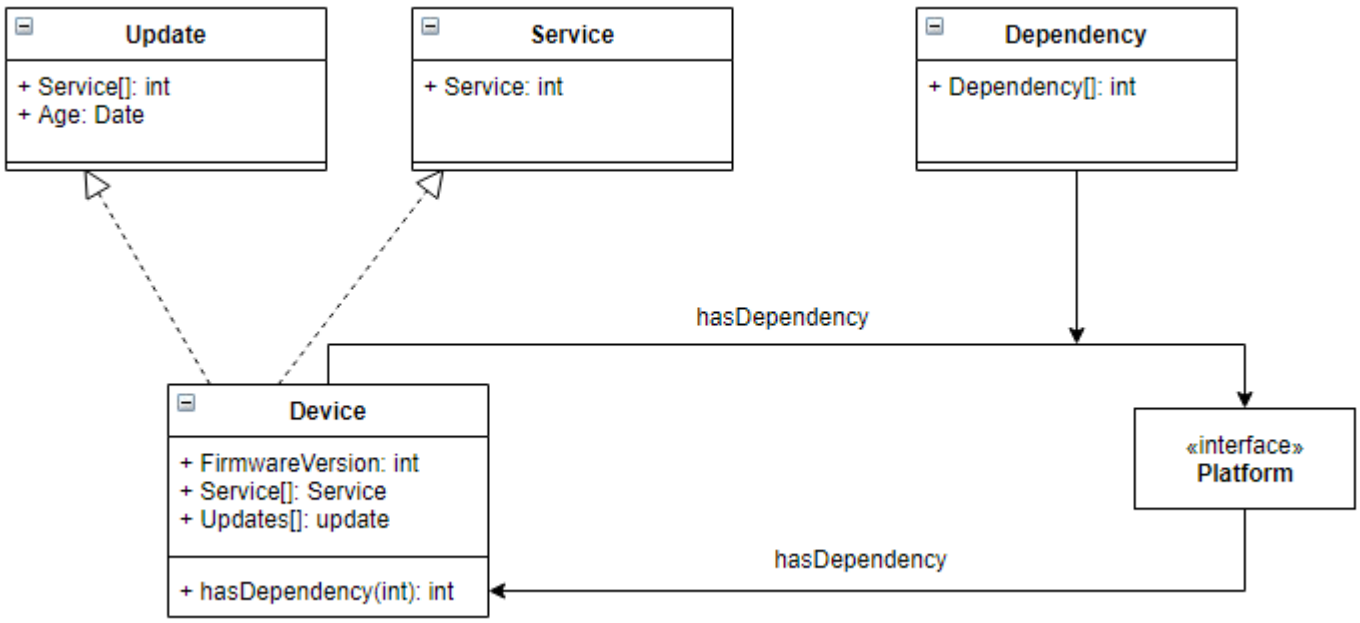

Figure 2.1: Caption

2.1.1 Prescriptive vs Descriptive Standards

Für die Kommunikation der Geräte untereinander existieren viele Kommunikationsprotokolle wie zum Beispiel Bluetooth, Wi-Fi oder auch XMPP. Diese sind bereits im großen Umfang in Verwendung und stark erforscht, weshalb es keiner weiteren Untersuchung dieser Protokolle im Rahmend dieser Arbeit bedarf. Kommunikation allein recht jedoch nicht, die Geräte müssen nicht nur untereinander kommunizieren, sondern sie müssen sich auch verstehen. Dafür ist ein Standard notwendig, welcher die Funktionen/Dienstleistungen der Geräte beschreibt. Die zwei führenden Ansätze im Bereich IoT/Smart Homes sind desktiptive und preskriptive Standards. [4] Preskriptive Standards definieren alle Geräte- und Dienstleistungstypen, die in einem Smart Home System vorkommen können. Durch diese "Vordefinition" kann es zu keinen Redundanzen kommen, was zum Beispiel bedeuten würde, dass zwei verschiedene Anbieter für den gleichen Gerätetyp zwei verschiedene Definitionen verwenden. Ein preskriptiver Standard ermöglich damit eine einfache Kommunikation zwischen den Geräten. Ein deskriptiver Standard hingegen erlaubt es dem Verkäufer seine Geräte und die dazugehörigen Dienstleistungen selbst zu definieren. Für Verkäufer bedeutet dies eine einfachere Entwicklung von Geräten und Dienstleistungen, sobald jedoch Geräte von verschiedenen Anbietern miteinander kommunzieren wollen, kann es aufgrund von Redundanzen nötig sein, dass eine Art Überset-

zung zwischen den Geräten stattfindet. In der Realität bieten sich hybride Standards vermutlich am meisten an, da sowohl deskriptive als auch preskriptive Standards Vor- und Nachteile besitzen.

2.1.2 Update Planing

Oft werden Smart Home Geräte automatisch über eine aktive Internetverbindung geupdatet oder auch durch andere Geräte wie zum Beispiel einem Hub. Dies bietet dem Nutzer mehr Komfort, da kein manueller Aufwand für Updates ensteht. Solches "Remote Updating" sollte aber eigentlich vermieden werden, da dadurch zum Beispiel schädliche Software installiert werden kann. Außerdem müsste der externe Diensleister, der das Updaten der Geräte übernimmt, viele Informationen über das System sammeln, um so ideale Updatekonfigurationen zu finden. Das Preisgeben aller Informationen sollte nicht leichtsinnig getan werden, da es viele potenzielle Gefahren birgt. Allein die Information, wann eine automatisierte Heizung anfängt zu heizen kann potenziellen Kriminellen Informationen darüber geben, wann jemand zu Hause ist oder wann ein Haus leer steht. Idealerweise sollten Nutzer daher das Updaten lokal selbst übernhemen. Dies bedeutet selbstverständlich mehr manuelle Konfiguration, vor jedem Update, da der Nutzer gefragt werden sollte, ob er ein installieren möchte. Diese Frage kann ein Nutzer jedoch ohne weitere Informationen gar nicht plausibel entscheiden. Es wäre also von Vorteil, wenn es für den Nutzer eine Übersicht über alle möglichen Updates geben würde, in der er sich dann für passende Updates entscheiden kann. Für einen solchen Zweck bietet sich Smartphone als zentrales Gerät an um eine Übersicht über alle im System vorkommenden Geräte zu erstellen. Dazu benötigt man eine Möglichkeit die Informationen über die Services und Updates eines Geräts leicht abfragen zu können. Eine solche Abfrage müsste in der Realität für jedes Gerät individuell angepasst werden. Dies ist jedoch nicht umsetzbar, weswegen es nötig ist einen einheitlichen Weg zu finden. Geräte besitzen üblicherweise Metadaten, welche Informationen über Software Version, Speicherbedarf usw. enthalten. Diesen Metadaten könnte man zusätzlich Informationen über Dienstleistungen und Updates hinzufügen, sodass man einen schnellen Überblick über alle Geräte innerhalb eines Netzwerks erhalten kann.

2.2 Langlebigkeit

Im Bereich Software Engineering ist Langlebigkeit ein bekanntes Problem. Es wurden bereits Untersuchungen über die durchschnittliche Lebensdauer von Software durchgeführt. Für kleine Projekte im Bereich von unter 100.000 Zeilen Code wird eine Lebensdauer von circa 6-8 Jahren erwartet. Dabei ist zu beobachten, dass mit steigender Größe eines Projekts die erwartete Lebensdauer im Durchschnitt ebenfalls steigt. Dies hängt vermutlich mit den hohen Kosten zusammen, die für größere Projekte aufkommen, weswegen

man daran interessiert ist lange einen Nutzen von der Software zu haben. Gründe für das Altern von Software sind vielseitig, zum Beispiel kann die Wartung von Software einfach zu teuer werden, so dass es sich lohnt neue Software zu entwickeln oder die Entwicklungsumgebung wird nicht mehr unterstützt und es lohnt sich nicht die Software zu portieren. Andere Gründe können Sicherheitslücken oder auch fehlerhafte Updates sein. Die German Research Foundation hat ein Projekt namens Design for Future ins Leben gerufen, um das Problem der Langlebigkeit von Software zu untersuchen. Ziel ist es fundamentale neue Ansätze im Bereich langlebiger Software zu finden, um Probleme mit Legacy-Software oder der Adaption von Software auf neue Plattformen zu lösen und eine kontinuierliche Weiterentwicklung von Software Systemen gewährleisten zu können [6]. Im Bereich Smart Home existieren keine solche Programme. Das Problem der Langlebigkeit wird in diesem Kontext unterschätzt oder einfach ignoriert. Aufgrund der Komplexität der Systeme besitzen sie viele Schwachstellen die zu einem frühzeitigen Ausfall des Systems führen können. Dazu gehören:

Threat
Discontinued External Services
Breaking Updates
New class of devices
Trade Conflict
Abandoned Protocol
Forced Incompatibility
Second Hand Smart Homes

Figure 2.2: Caption

Die Gefahr, dass ein externer Service, der für ein Smart Home System notwendig ist, offline geht (Discontinued External Servic) ist eine Gefahr die man als Käufer aktiv eingeht, wenn man sich zum Beispiel ein Cloud-gebundenes Smart Home System kauft. Das Ausmaß der Gefahr lässt sich jedoch weitesgehend vermeiden, indem man bei etablierten Anbietern einkauft oder einfach komplett auf die Abhängigkeit zu einem externen Service verzichtet. "Breaking Updates" können Abhängigkeiten im System zerstören und sind somit eine Gefahr, gegen die man als Nutzer nichts unternehmen kann, außer Updates generell nicht zu installieren. Dadurch muss jedoch auf neue Funktionalitäten verzichtet werden, während sich gleichzeitig das Risiko erhöht, dass Sicherheitslücken im System aufkommen. Updates also einfach strikt zu ignorieren ist nicht empfehlenswert, weswegen es, wie bereits in Kapitel 2 erwähnt, einer Möglichkeit bedarf Updates vor ihrer Installation zu überprüfen.[5]

2.3 Dependency Managment

Unter Dependency Management versteht man im Allgemeinen das Strukturieren von Abhängigkeiten verschiedener Systeme/Programme. Es ist oft so, dass Programme nur in Abhängigkeit von anderen Programmen starten können. Dies stellt auch kein Problem dar, solange die Abhängigkeiten in ihrer Anzahl überschaubar bleiben. Realität ist jedoch, dass Programme meist von mehreren Programmen abhängig sind und diese Prorgramme wiederum abhängig von anderen Programmen. Bei zu vielen Abhängigkeiten verliert man als Entwickler und auch als Nutzer schnell den Überblick. Dieses Wirrwarr an Abhängigkeiten bezeichnet man umgangssprachlich als "Dependency Hell", denn sobald man einmal den Überblick verloren hat, wird es äußerst schwierig weitere Änderungen am System vorzunehmen.

2.3.1 Probleme

Es folgt eine kleine Zusammenfassung, der am häufigsten auftretenden Formen der "Dependency Hell":

- Long Chain of Dependencies: Ein Programm A ist abhängig von einem Programm B. Das Programm B wiederum ist abhängig von einem Programm C. Möchte man nun Programm A nutzen benötigt man zusätzlich Programm B und damit auch Programm C. Solche lange Ketten können Konflikte verursachen. (Siehe Conflicting Dependencies)
- Conflicting Dependencies: Sei ein Programm A abhängig von einem Programm B1.1, ein anderes Programm B ist abhängig von Programm B1.2. Programm B kann aber nicht gleichzeitig mit der Version 1.1 und der Version 1.2 installiert sein. Dies bedeutet, Programm A und B können nicht gleichzeitig in einem System laufen.
- Circular Dependencies: Sei ein Programm A abhängig von einer spezifischen Version von Programm B. Programm B wiederum ist abhängig von einer spezifischen Version von Programm A. Beide Programme können nun nicht geupdatet werden, da sie die Abhängigkeiten kaputt verletzen würden.
- Version Lock: Durch einen Version Lock ist man dazu gewungen beim updaten einer Software alle abhängigen Programme ebenfalls upzudaten. (semver.org)
- Version Promiscuity: Werden Abhängigkeiten zu locker betrachtet können zukünftige Updates als kompatibel angesehen werden, obwohl sie es nicht sind.

Für die genannten Probleme existieren bereits viele Ansätze, um gegen sie vorzugehen oder sie zu vermeiden. Dazu gehören:

2.3.2 Lösungen

- Semantic Versioning: Semantic Versioning versioniert Software nach dem Format "MAJOR.MINOR.PATCH", um Probleme wie Version Lock oder Version Promiscuity zu vermeiden. Dafür benötigt man eine Schnittstelle der man die Änderungen an der Software mitteilt. Bei Änderungen, die die API betreffen und nicht abwärtskompatibel sind muss die MAJOR Nummer erhöht werden. Bei Änderungen, die abwärtskompatibel sind, muss die MINOR Nummer erhöht werden. Änderungen, die die API nicht betreffen erhöhen die Patch Nummer. Es gibt noch viele weitere Spezifikationen bezüglich Semantic Versioning, das grobe Konzept sollte jedoch verstanden sein.
- Package Manager: Package Manager sind Tools die sich um das installieren, updaten und konfigurieren von Programmen automatisch kümmern, wodurch eine manuelle Konfiguration nicht mehr nötig ist. [Wikipedia]
- Portable Applications: Portable Applikationen umgehen die Probleme, die Abhängigkeiten mit sich bringen. Sie funktionieren selbstständig, indem alle nötigen Komponenten bereits vorhanden sind.

Die Auflistungen sollen zeigen, dass Dependency Management ein allgegenwärtiges Thema ist, welches in den meisten Bereichen bereits viel Aufmerksamkeit bekommt. Dependency Management umfasst natürlich noch viele weitere Probleme und dazugehörige Lösungen und Lösungsansätze, im Bereich Smart Home exisitiert jedoch nahezu keine Forschungsgrundlage. Als Nutzer kann man nur hoffen, dass neue Updates keine Schäden anrichten, da es keine Möglichkeit gibt sich davor zu schützen.

2.4 Fazit

Im Bereich Software Engineering wird Langlebigkeit viel Aufmerksamkeit geschenkt. Smart Home Systeme sind selbstverständlich Teil des Software Engineerings, aber schaut man sich den aktuellen Stand der Forschung der Langlebigkeit explizit im Bereich der Smart Home Systeme an, wird deutlich das diesbezüglich kaum Forschungsgrundlagen bestehen. Dabei werden Smart Home Systeme als Luxusgüter angesehen von denen man aufgrund der hohen Preise eine lange Lebensspanne erwartet. Um dies zu gewährleisten werden generelle Ansätze des Software Engineerings bezüglich Langlebigkeit genutzt, ausreichend ist dies jedoch nicht. Ohne vernünftiges Dependency Management kann nicht gewährleistet werden, dass Updates keine Ausfälle in Smart Home Systemen verursachen. Dies sieht man am Beispiel des Logitech Harmony Hub's. Logitech hatte eine neue Firmware Version für das Hub veröffentlicht, welche angeblich nur Sicherheitslücken schließen und Bugs fixen sollte. Nachdem Update kam es jedoch vermehrt zu Ausfällen zwischen dem Hub und third-party Geräten. Den Nutzern waren die Hände gebunden,

sie konnten nur darauf hoffen, dass Logitech die Probleme wieder behebt. Dies hat Logitech in diesem Fall auch getan, dennoch zeigt dieses kleine Beispiel, wie leichtsinnig Updates von Nutzern installiert werden. Eine andere Möglichkeit besitzt man auch als Nutzer eines Smart Homes nicht wirklich, da es wie eingangs erwähnt, zur Zeit keine Möglichkeiten gibt Updates vor der Installation zu überprüfen.

Chapter 3

Design und Implementation

Dieses Kapitel beschreibt das Design der Implementation des Algorithmus und erklärt Entscheidungen, die im Laufe der Implementation getroffen wurden.

3.1 Design

Zunächst folgt eine grobe Beschreibung aller Schritte des Algorithmus, um so als Leser genauere Entscheidungen im Laufe der Arbeit besser nachvollziehen zu können. Der Algorithmus basiert auf dem Paper "An Algorithm for Dependency-Preserving Smart Home Updates" (Peter Zdankin, Matthias Schaffeld, Marian Waltereit, Oskar Carl, Torben Weis, 2020). Ziel ist es ist, wie Bereits in Kapitel 1.1 erwähnt, Abhängigkeiten zwischen Geräten zu untersuchen und basierend auf diesen Abhängigkeiten Updatekonfigurationen zu ermitteln, welchen den größten Nutzen für den Nutzer haben.

1. Im ersten Schritt wird eine Übersicht über die aktuelle Updatekonfiguration erstellt. Eine Updatekonfiguration ist eine Zusammenfassung aller Geräte und ihrer aktuellen Firmwareversion. Dadurch weiß man, welche Dienslteistungen die Geräte aktuell anbieten. Zusätzlich wird eine Liste aller möglichen Updates eines Geräts erstellt.

2. Der zweite Schritt ist eine Optimierung bei der versucht wird die Anzahl der interessanten Updates zu verringern. Dazu werden dominierte Updates herausgefiltert und im weiteren Verlauf nicht mehr in Betracht gezogen. Wie genau dies geschieht und welche Updates dominiert sind, wird in einem späteren Kapitel erläutert.

3. Im dritten Schritt wird mit Hilfe der Übersicht über alle Geräte und Updates ein Updatekonfigurationsgraph erstellt. Diesen Graphen kann man sich als nichtdeterministischen endlichen Automaten vorstellen bei dem jeweils ein Zustand eine Updatekonfiguration darstellt. Gegeben seien drei Geräte mit jeweils zwei Updates, dann bedeutet der Start Zustand, dass sich Gerät 1 auf Version 0, Gerät 2 auf Version 2 und Gerät 3 auf Version 4 befindet. Die anderen Zustände bilden alle anderen Möglichkeiten. Siehe Abbildung 3.1.

Figure 3.1: Caption

4. Im vierten Schritt werden alle Updatekonfigurationen dahingehen überprüft, ob sie eine Abhängigkeit verletzen. Ist dies der Fall wird diese Konfiguration aus dem Updatkonfigurationsgraphen entfernt.

5. Im fünften Schritt werden alle nicht Pareto-optimalen Konfigurationen herausgefiltert. Pareto-optimal wird im späteren Verlauf definiert.

6. Im sechsten Schritt werden die Updatekonfigurationen in Subnetzwerke unterteilt. Besteht zum Beispiel keinerlei Abhängigkeit zwischen zwei Gruppen von Geräten in einem System, kann man diese Gruppen in Subnetzwerke unterteilen. Für diese Gruppen kann man später unabhängig von anderen Gruppen die passende Updatekonfiguration auswählen.

7. Im letzten Schritt wird jeder Updatekonfiguration ein Rating zugewiesen mit dem Ziel dem Nutzer die Entscheidung für die passende Updatekonfiguration zu erleichtern.

Basierend auf den Erkenntnissen aus den Kapiteln 1 und 2 können nun Designentscheidungen bezüglich des Algorithmus getroffen werden. Ziel ist es den Prototypen eines zukunftsfähigen Tools zu entwickeln, welcher Smart Home Nutzern die Möglichkeit bietet Updates vor ihrer Installation zu überprüfen. Dabei zielt das Design darauf ab, unnötige architekturspezifische Regularien zu vermeiden, um so möglichst simpel gestaltet werden zu können. Nichtsdestotrotz sollte es möglichst einfach sein den Algorithmus in verschiedene Smart Home Architekturen zu integrieren. Um solch eine einfache Integration zu gewährleisten wird die Implementation daher in Java sehr modular gehalten. Java bietet sich dafür als objektorientierte Programmiersprache sehr an. Des Weiteren wird der Algorithmus nicht auf der Basis eines realen Smart Home Systems aufgebaut. Dies wäre kontraproduktiv, da man als Entwickler eher dazu neigen würde eine systemspezifische Implementation zu entwickeln. Anstatt also ein reales Smart Home Systems als Grundlage zu nehmen werden im Folgenden andere Möglichkeiten genauer beleuchtet. Eine Möglichkeit ist es eine Liste mit Geräten, aktueller Firmware Version, möglichen Updates und allen dazugehörigen Dienstleistungen zu erstellen. Fügt man dieser Liste dann noch Abhängigkeiten hinzu erhält man alle wichtigen Informationen

über ein Smart Home System, die man benötigt. Vorteil einer solchen Liste ist, dass es sehr einfach wäre sie zu erstellen. Für die spätere Evaluation des Algorithmus wäre solch eine statische Liste jedoch kontraproduktiv, da man den Algorithmus somit immer am gleichen System testen würde. Dadurch könnte man keine validen Aussagen über die Performance des Algorithmus treffen. Aus diesem Grund wird ein Generator genutzt, welcher in Abhängigkeit von verschiedenen Parametern Smart Home Systeme kreiert. So können die Anzahl der Geräte, die Anzahl der möglichen Updates pro Gerät, die Anzahl der Dienstleistungen pro Gerät oder auch die Anzahl der Abhängigkeiten zwischen den Geräten variiert werden. Dies ermöglich den später implementierten an Algorithmus an verschiedenen Systemen zu evaluieren, um so ein gutes Fazit über Effiziens und Lauftzeit treffen zu können.

Das Design eines solchen Generators lässt sich in Java sehr leicht gestalten. Der Generator erzeugt Geräte, welche in Form von Objekten gespeichert werden. Diese Objekte enthalten Informationen über die Version, die Dienstleistungen und die möglichen Updates des Geräts. Zusätzlich wird noch das Alter der Updates für spätere Zwecke gespeichert. Idealerweise sollten Updates nach den Spezifikationen des Semantic Versioning definiert werden, jedoch

```
public class Device {
    int version;
    ArrayList<Integer> services;
    ArrayList<ArrayList<Integer>> updates;
    ArrayList<ArrayList<LocalDate>> updateAge;
}
```

Nun stellt sich die Frage wie die Geräte sich selbst definieren, da ohne klare Definitionen eine Kommunikation zwischen den Geräten nicht möglich wäre (siehe Kapitel 2.1.1). Es wird davon ausgegangen, dass Geräte und Dienstleistungen unter einem preskriptiven Standard definiert sind. Die Geräte können also untereinander kommunzieren ohne, dass eine Übersetzung nötig ist, wie es unter einem deskriptiven Standard der Fall wäre. Rein für die Implementierung des Algrorithmus ist es von Vorteil sich an preskriptiven Standards zu orientieren, da durch die Homogenität der Geräte und die redundanzfreie Architektur die Komplexität des Algorithmus gemindert werden kann. Nachdem die Geräte erstellt wurden, müssen Abhängigkeiten zwischen diesen Geräten kreiert werden, um so ein Smart Home System zu simulieren. Eine Abhängigkeit herrscht zwischen zwei Geräten, wenn ein Gerät Zugriff auf die Dienstleistungen eines anderen Geräts benötigt. In Abbildung 3.2 sieht man die Abhängigkeit zwischen einer Heizung und einem Thermostat. Die Heizung greift periodisch auf die Dienstleistung des Thermostats zu, indem es das Thermostat nach der aktuellen Temperatur fragt. Liegt die Temepratur nun unter einer bestimmten Schwelle würde die Heizung automatisch anspringen und so gewährleisten, dass immer eine Mindesttemperatur vorhanden ist.

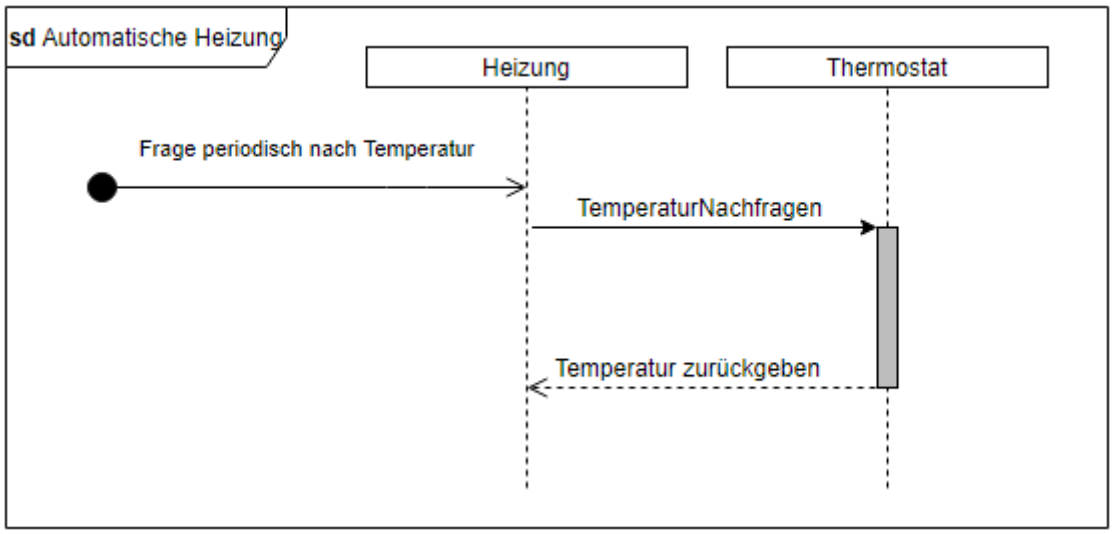

Figure 3.2: Caption

Diese zwei Komponenten, also Geräte und Abhängigkeiten, bilden bereits ein simples Netzwerk, welches für die Anwendung des Algorithmus völlig ausreicht. Des Weiteren wird vorausgesetzt, dass innerhalb eines solchen Systems keine Updates automatisch installiert werden und es möglich ist, wie in Kapitel 2.1, beschrieben, sich eine Übersicht über alle Geräte und deren Informationen zu beschaffen. Bei realen Systemen müssten diese Informationen in den Metadaten der Geräte gespeichert werden, um sie leicht abfragen zu können. Da Metadaten bei den meisten Geräten bereits existieren, sollte es für die Entwickler dieser Geräte kein großes Problem darstellen die Metadaten um diese Informationen zu ergänzen. Genauere Details, wie zum Beispiel, ob die Kommunikation über eine Cloud läuft sind nicht interessant und werden dementsprechend vernachlässigt. Nun steht die Basis für die Implementation des Algorithmus. Im ersten Schritt werden die Updates aller Geräte untersucht. Ziel dieses Schrittes ist es, dominierte Updates zu finden und diese zu löschen, um so Rechenpower einzusparen. Updates sind dominiert, wenn ein anderes Update exisitiert, welches mindenstens die gleichen Dienstleistungen anbietet und zusätzlich aktueller ist.

Version	Services
1.0	1,2
1.1	1,2
2.0	1,3
2.1	1,3
3.0	3,4

Figure 3.3: Caption

In Abbildung 3.3 ist ein Gerät mit 5 verschiedenen Versionen zu sehen. In diesem Beispiel ist die Version 1.0 von der Version 1.1 dominiert, da Version 1.1 die gleichen Dienstleistungen wie Version 1.0 anbietet und gleichzeitig aktueller ist. Das Gleiche gilt für Version 2.0 und 2.1. Für dieses Gerät bleiben also nur noch drei potenzielle Updates, nämlich Version 1.1, 2.1 und 3.0 übrig. Diese Optimierung an allen Geräten im System durchzuführen spart eine Menge Rechenzeit, da nachdem alle dominierten Updates entfernt wurden das kartesische Produkt zwischen den restlichen Updates gebildet wird.

The evaluation usually consists of three main steps: first it defines the goals intended to be achieved by the software in detail; next is a description of the methodology used to measure the satisfaction of the software in relation to these goals; finally, the measurements are depicted and assessed.

Chapter 4

Conclusion

The conclusion quickly summarizes the results of the paper in relation to the previously defined goals and hypotheses. It usually also includes some information on next steps and further research that might be required or possible on the presented subject.

4.1 Bibliograhpy

Bibliography

[1] https://www.verifiedmarketresearch.com/product/global-smart-home-market-size-and-forecast-to-2025/

[2] https://www.homeandsmart.de/

[3] Peter Zdankin, Matthias Schaffeld, Marian Waltereit, Oskar Carl, Torben Weis, "An Algorithm for Dependency-Preserving Smart" Home Updates"

[4] (https://policyadvice.net/insurance/insights/smart-home-statistics/)

[5] (https://www.ibp.fraunhofer.de/content/dam/ibp/ibp-neu/de/dokumente/sonderdrucke/bauphysik-gertis/6-einsparpotenziale-intelligente-heizungsregelung.pdf

4.1 Code Listings

If there is some interesting code you would like to show in order to ease the understanding of the text, you can just include it using the `lstlisting` environment. Have a look at the source of this page to see how this is included:

```
x := from(42);
```

You could also put the code into an external file and include it in this document using the `lstinputlisting` command:

```
1 var x = new Node
2 if luck() {
3     var y = new Node
4     x.next = y
5 }
6 return
```

Be careful not to include large files as it hampers readability. If there is a short excerpt from a large file you would like to show, you can also extract an explicit range of lines from it without the need to modify the source file. This next listing only shows the conditional from the previous code:

```
2 if luck() {
3     var y = new Node
4     x.next = y
5 }
```

To use more advanced syntax highlighting have a look at the available options of the *listings* package or use the *minted* package[1], which has more extensive language support and additional themes. Both can be configured in the main file.

4.2 Math

In case you need to include some math, the *amsmath* package[2] is already included in this document.

To properly display some short formula like $e^{i\pi} = -1$, you can use the `\( \)` inline command. For larger formulas, the `math` environment is more appropriate. If you need to reference the formula multiple times, e.g. in case it is used in theorems, you should use the `equation` environment:

$$\vec{\nabla} \times \vec{B} = \mu_0 \vec{j} + \mu_0 \varepsilon_0 \frac{\partial \vec{E}}{\partial t} \tag{4.1}$$

To reference it as 4.1 using the `\ref{}` command, remember to use a `\label{}`.

4.3 Miscellaneous

You can use the `\todo{}` command to put obvious reminders on the side of the document.

TODO: like this!

[1] `https://texdoc.net/texmf-dist/doc/latex/minted/minted.pdf`
[2] `https://texdoc.net/texmf-dist/doc/latex/amsmath/amsmath.pdf`

Versicherung an Eides Statt

Ich versichere an Eides statt durch meine untenstehende Unterschrift,

- dass ich die vorliegende Arbeit - mit Ausnahme der Anleitung durch die Betreuer - selbstständig ohne fremde Hilfe angefertigt habe und
- dass ich alle Stellen, die wörtlich oder annähernd wörtlich aus fremden Quellen entnommen sind, entsprechend als Zitate gekennzeichnet habe und
- dass ich ausschließlich die angegebenen Quellen (Literatur, Internetseiten, sonstige Hilfsmittel) verwendet habe und
- dass ich alle entsprechenden Angaben nach bestem Wissen und Gewissen vorgenommen habe, dass sie der Wahrheit entsprechen und dass ich nichts verschwiegen habe.

Mir ist bekannt, dass eine falsche Versicherung an Eides Statt nach §156 und §163 Abs. 1 des Strafgesetzbuches mit Freiheitsstrafe oder Geldstrafe bestraft wird.

Duisburg, 5. September 2021
(Ort, Datum)

(Vorname Nachname)