Großer Beleg

Construction of Variable App-Level Process Chains in Android

submitted by

Sophie Ziemann
born 12.02.1996 in Chemnitz

Technische Universität Dresden

Faculty of Computer Science
Institute of Applied Computer Science
Chair of Human-Computer Interaction

Supervisors:
David Gollasch, M. Sc.

Professor:
Prof. Dr. rer. nat. habil. Gerhard Weber

Submitted April 20, 2020

Abstract

In an ageing society with an understaffed health sector there is a growing demand for adaptive assistance robots helping to ease the workload of staff and enabling individuals to stay autonomously at home for longer. The need for adaptability includes the demand for the execution of variable tasks that are adapted to user needs and preferences. Means from software variability, as used in software product lines and software ecosystems, can be borrowed to construct such tasks.

This thesis proposes a way of executing those tasks on the Android based adaptive assistance robot platform Loomo. The premise is that tasks can be split into atomic subtasks, which can then be combined to form complex process chains. In the context of Android each subtask is executed by an app.

In the concept suggested in this thesis a task object, which holds a list including all information related to all tasks in the process chain and may have been constructed using means of software variability, is handed to a central manager app. This app retrieves the first subtask app required for the task execution from the task object and starts this app's execution via an Android Intent, which also contains the task object. Upon finishing, each subtask app starts the next subtask app on the task list. As the manager app is the last entry on the task list it is restarted after the last subtask has been executed.

To prove the feasibility of the concept a prototype was implemented, consisting of a manager app and five subtask apps. In the prototype scenario the robot's task is to find the user's glasses. The execution of the task can be adapted to the user's preferences regarding the output modality, as the robot will either communicate with the user via speech or by displaying text on its display.

Though some open challenges remain, namely the need for controllability while executing a task and the handling of unexpected situations at runtime, the concept developed in this thesis was found to be a feasible option for the execution of variable app-level process chains in Android.

Contents

The Need for the Construction of Variable App-Level Process Chains in Android

Our population is ageing, due to longer life expectancy and declining birth rates. At the same time, the nursing sector suffers from being understaffed [Koc15]. Smart assistance robots can help to improve this situation by assisting caregivers in tasks such as lifting people or entertaining patients, but also by giving old people the opportunity to remain living autonomously at home longer[Pol05]. While in times of Industry 4.0 manufacturing without robots seems unimaginable, the development of robots in the health care sector can't keep up despite the demand. While there are some robots already in use in the health care domain, like the therapy robot seal Paro[1] or the robot bear RIBA[2], who can lift people in and out of bed, most of them are quite restricted in their range of functions.

However the requirements for robots working in the nursing sector are manifold. They include being able to answer questions, recognising and displaying emotions for natural interaction, monitoring the user, communicating information to secondary users like relatives or caretakers, helping to navigate spaces and many more [Pol05; Bzu+12; OO12]. It is crucial that one robot can fulfil all or at least most of these requirements, making the investment worthwhile for individuals and nursing facilities alike. Besides having a great variety of services it can provide to its users, an assistance robot must also be able to cope with a variety of potentially unknown situations and adapt to the specific needs of the current user. It should for instance be capable of adapting its behaviour to user preferences or disabilities, the environment, its own state or cultural norms.

As each task the robot can perform must be adaptable to user needs, variants of this task emerge that each match a specific configuration of user requirements. The need for adaptability calls for ways to handle the arising variability. This can be accomplished through the application of means from the domain of software variability in the domain of robotics. To achieve this, software development plays a key role, but is proving to be the "bottleneck of robotic systems engineering" [Gar+19].

Problem Description

This thesis uses the Android-based assistance robot Loomo as a platform for a prototype showcasing the concept that is going to be developed. It must however be noted, that the considerations made in this thesis are not solely applicable to Loomo, but can be applied to every Android based robot.

The execution of tasks is usually implemented statically for robots: For each task the robot is able to perform, there is some separate fixed code. In Android the execution of a task would correspond to the execution of a single app.

[1]`http://www.parorobots.com/`
[2]`http://rtc.nagoya.riken.jp/RIBA/index-e.html`

With an increasing function range however, the code should still remain maintainable, expandable and exchangeable. Code reuse can also help decrease complexity. Applying a modular approach helps to fulfil these requirements and to manage the code more efficiently. Moreover, it facilitates the introduction of variability mechanisms.

Applying this concept to Android, a modular task can be divided into several atomic subtasks, each of which the robot can perform through the execution of a separate app. To perform the modular task (i.e. the process chain), the necessary subtasks (apps) must be executed in the right order. Furthermore, there must be a task object that holds the task list with information on the order of the necessary apps for the task. The modular approach allows an easy exchange of parts of the task list or to add new apps to the pool of available subtask apps. This enables a flexible composition of components without the need to touch the source code of the existing components, which can be beneficial when wanting to introduce variability to make the robot adaptable to user needs.

To handle variability in software systems, a variability management can be deployed (see Section 1.2). Given a set of requirements it produces a concrete configuration of components. This concept can be applied to the construction of variable app-level process chains in Android to produce the task object holding a sequence of subtasks.

The goal of this thesis is to construct the task object, which holds all information about the apps to be executed, for the Android platform and to formulate a concept on how to start the execution of those process chains, so that the apps on the task list are executed consecutively and automatically.

Research Question and Goals

The discussed research question of this thesis is the following:

How can a job sequence that has been created with means of software variability be executed in Android?

To answer this question, we must achieve the following goals:

Goals

- Describe how adaptive assistance robots currently execute tasks.
- Explain software variability, software variability management and existing methods.
- Identify software variability management methods that are suitable for the use case.
- Design a suitable concept for executing variable process chains in Android.
- Create a Prototype for the concept to prove it is suitable.
- Evaluate the prototype in terms of whether it implements the concept and answers the research question.
- Give a summary of the findings of the thesis and an outlook on further possible extensions of the concept.

Structure of the work

- **Background** This chapter will talk about the need for assistance robots and gives some examples for assistance robots as a research object, as well as introduce the notion of software variability and why this can be relevant in the robotics domain.
- **Concept** In this chapter we will design a running example and use it to evaluate the previously presented variability modelling and realisation approaches regarding the problem description. We will design a concept for the execution of process chains, introduce the assistance robot platform Loomo and provide Android basic principles required for the prototype.
- **Prototype** In this chapter the components of the prototype will be described and documented.
- **Evaluation** This chapter will evaluate the functionality of the prototype.
- **Summary and Conclusion** This chapter will summarise the findings of the thesis.

1 Assistance Robots and Software Variability

This chapter will give an overview of the range of adaptive assistance robots and why they are so important. Furthermore examples for social robots in education and health care that are already commercially available while also being relevant to researchers are given. Later on the principles of software variability are presented and the need for software variability in the robotics domain is explained.

1.1 Adaptive Assistance Robots

One of the main purposes of robots, besides entertaining, is to replace humans in various tasks. This might be because some tasks are too dangerous to be performed by humans, such as the defusing of bombs, or too demanding, such as lifting heavy manufactured parts. Robots are also more reliably precise than humans, a trait which is essential in manufacturing, where every piece must turn out precisely as designed.

But aside from replacing humans, robots may also assist humans. Though there are cases of human-robot interaction in manufacturing, assistance robots usually appear in non-industrial contexts. In those cases robots usually perform tasks that could also be done by humans. This may happen out of convenience to facilitate people's jobs and lifes, but it might also happen if there are no humans available for the job, for instance due to the staff shortage in the health care sector.

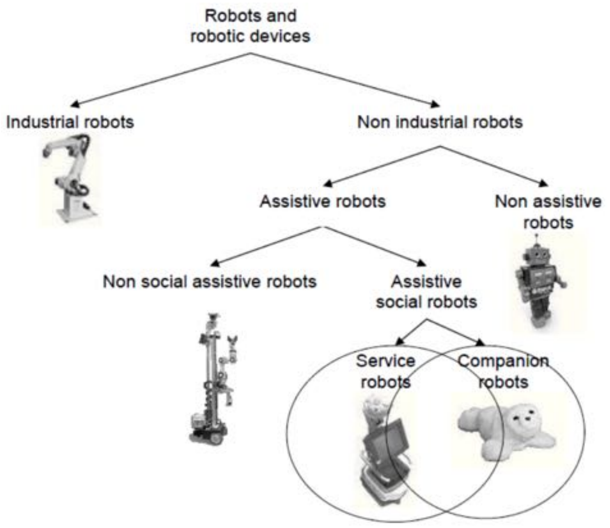


Figure 1.1 – A classification of robots and robotic devices [Hee+10; OO12]

Definition In her report Neumann [Neu16] states that there is no universal definition for the term assistance robot, however she does provides the following description: Firstly, assistance robots are *service robots*, which, as defined in ISO 13482, refers to robots performing ”useful tasks for humans or equipment excluding industrial automation applications” [Ii14]. Secondly, human assistant robots are used for welfare purposes and livelihood support (excluding medical applications), provide physical and/or mental support (e.g. in the form of social interaction, communication and monitoring) and typically support care patients, the elderly and caregivers. [Neu16]

Furthermore, Heerink et al. [Hee+10] and Orha et al. [OO12] provide a categorisation of robot technologies (Figure 1.1). According to them, assistive robots can be divided into *social* and *non social assistive robots*. The latter refers to physical assistive technology that is not in any way socially interactive. An application for this kind of robot is the assistance in rehabilitation [OO12].

Research on *assistive social robots* on the other hand can be found in two areas, from which two subcategories emerge: The first one regards *companion robots* providing social support. They are used for social companionship or robot assisted therapy. The robot seal Paro[1] falls under this category, as it is used in therapy for dementia patients.

The second kind of assistive social robots are socially interactive *service robots*, offering physical and cognitive assistance. Their purpose is mainly to carry out service tasks of any kind for the user. A representative of this category is the German Care-O-bot, a robot with advanced assistive functionalities intended to provide for instance walking assistance and butler services. It will be presented in more detail in the next section, as well as the humanoid assistance robot Pepper, which also can be put in the same category.

Heerink et al. attribute certain characteristics to social assistive robots, such as being able to express and/or perceive emotions. The former can be done by using facial expressions, sounds or speech, the latter for instance by analysis of facial expressions or sounds. They are able to communicate using high level dialogue, for instance through asking questions or using dialogue to solve problems mutually. Furthermore, they can learn and recognise models of other agents, establish and maintain social relationships, use natural cues such as gaze or gestures, exhibit a distinctive personality and character and may learn or develop social competencies. Though their research focuses on the domain of elder care, this categorisation can be applied to robots in all domains. [Hee+10; OO12]

Combining Neumann’s description of assistance robots [Neu16] and the characterisation of assistive social service robots by Heerink et al. and Orha et al. [Hee+10; OO12] leads to proposing this general definition of the term assistance robot:

Definition 1: Assistance robot
An assistance robot is a robot performing useful tasks for humans in a non-industrial context with the ability for natural social interaction.

In this thesis the term *assistance robot* is going to be used according to this definition.

1.1.1 Requirements for Adaptive Assistance Robots in Health Care

In the previous section, assistance robots have been characterised. In the process of deriving a definition, features that qualify a robot as an assistance robot were named, such as providing physical and mental aid or the ability to show social behaviour. The core requirements for assistance robots that emerge from those features can be found in the definition: the *ability for*

[1] `http://www.parorobots.com/`

natural social interaction and providing *physical and mental assistance* (phrased as ”performing useful tasks for humans”). However, when it comes to becoming a successful and widely adapted assistance robot in the health care domain, a robot must not only fulfil the core requirements, but also meet the specific needs of the user group.

Though smart and flexible robots capable of human-computer interaction are getting increasingly important in industrial robotics technology [RCW18], the requirements for robots in health care are quite different. Hence the health care sector can't fully benefit from the advances made in the field of industry robots. Big companies can afford to assign one robot to one task, as the time they save and the improved accuracy will make the investment worthwhile. This is not the case in the health care sector, as there is not much profit to be expected from the employment of robots, but rather an increased quality of life. Furthermore, the tasks a single robot should be able to perform might be more manifold and complex than in a factory environment, as health care facilities and individuals usually aren't able to afford multiple robots. This leads to the need for affordability and a broad function range that makes up for the cost of an assistance robot.

Another difficulty poses the fact, that when directly interacting with humans outside of a predictable factory environment, a robot must be able to cope with many different scenarios and open ended environments [Sch+15; Gar+19; Ing+18]. Often in manufacturing, one robot is specialised in doing one particular task under fixed environmental conditions. For robotic applications outside of industrial applications, adaptability, i.e. the ability to change in order to suit different conditions[2], plays a big role. Cully et al. [Cul+15] compare robots to animals, which can quickly adapt to injuries. Many robots could benefit greatly from this skill, as their fragility hinders the adoption in complex environments outside factories. The ability to adapt to new situations can help robots to function even when they are broken.

Being able to cope with unknown and uncertain situations is essential for many kinds of robots, such as those working in space [LNS13], e.g. the robot Justin designed by the DLR to repair satellites[3]. Robots directly interacting with humans must be able to adapt to a social context instead of a physical environment. During interaction users might show unexpected behaviour, different user groups might prefer different interaction modalities, or cultural norms in different countries require the robot to behave differently[4]. This makes adaptability another crucial requirement for a successful assistance robot.

After conducting interviews and listening to focus groups Bzura et al. [Bzu+12] came up with a set of user requirements for a personal health care robot. They concluded that an ideal robot must provide the primary user with independence, emergency support and a sense of security. The robot must be affordable or provide some value to secondary users that may compensate for some of its cost. Lastly, the robot must have an intuitive user interface to provide a seamless and convenient user experience. [Bzu+12]

To sum it up, the requirements for assistance robots in the context of health care include:

- Adaptability, i.e. the ability to adapt to unknown situations
- A broad function range and/or also providing value to secondary users
- Affordability

[2]https://dictionary.cambridge.org/dictionary/english/adaptability
[3]https://www.dlr.de/rm/en/desktopdefault.aspx/tabid-11427/#gallery/29202
[4]http://caressesrobot.org/en/project/

- Providing the primary user with independence, emergency support and a sense of security
- An intuitive user interface to make for a seamless user experience

While all of them are necessary for making a successful assistance robot, this thesis focuses on how adaptability can be achieved.

1.1.2 Assistance Robots in Practice

To give an idea of what contemporary robots are capable of, two assistance robots, as defined in Definition 1, that are suited to be employed in the health care domain and each fulfil most of the requirements specified in section 1.1, will be presented. The robots chosen are among the best known and most capable adaptive assistance robots and have been successfully employed in real life scenarios in many different areas. They are also highly relevant to researchers, serving as platforms to evaluate concepts from various domains or extending the function range to meet user needs.

Furthermore the assistance robot Loomo will be introduced in Section 2.4, as it will serve as a platform for a protoype, showcasing the concept that is going to be developed in this thesis.

Pepper Pepper[5] is a humanoid robot created by the French robotics company Aldebaran Robotics, which in 2015 was acquired by the Japanese telecommunications and media group SoftBank and rebranded as SoftBank Robotics in 2016[6]. Being launched in 2014 and introduced to the market in 2015, Pepper was the world's first full-scale social humanoid robot to be offered to consumers that could recognise faces and basic human emotions. It is optimised for human interaction which is enabled through conversation and its touch screen[7].

Pepper (Figure 1.2) stands 120cm tall and weighs 28kg. Its 20 motors provide 20 degrees of freedom and with its 3 omnidirectional wheels it can move at 3km/h maximum velocity. It is equipped with touch sensors, LEDs and microphones for multimodal interaction as well as infrared sensors, bumpers, an inertial unit, 2D and 3D cameras, and sonars for omnidirectional and autonomous navigation. Perception modules help recognising people talking to it and it can interact in 15 languages. The battery lasts for 12 hours and it is based on Aldebaran's open and fully programmable Linux-based platform NAOqi OS.

Though initially only being designed for a business-to-business (B2B) application in SoftBank stores, it has since been adapted for business-to-consumer (B2C), business-to-academics (B2A) and business-to-developers (B2D) applications [PG18]. Out of the 10,000 Peppers sold so far mainly in Japan, only about 3,000 serve in B2B applications. Besides being employed to greet customers and provide information about products and services by over 2000 companies, among them big brands such as Nestlé, Renault or Uniqlo[7], successful trials have also been performed in Europe at health care and elderly care facilities and on Costa Cruise ships. About 2,000 robots have been provided to educational institutions in Japan to support the teaching of robot programming, as robotics knowledge will become increasingly important in the future. Apart from being deployed at schools and homes, it has also been selected as the robotic platform for the

[5]https://www.softbankrobotics.com/emea/en/pepper
[6]https://www.softbankrobotics.com/emea/en/company
[7] https://robots.ieee.org/robots/pepper/

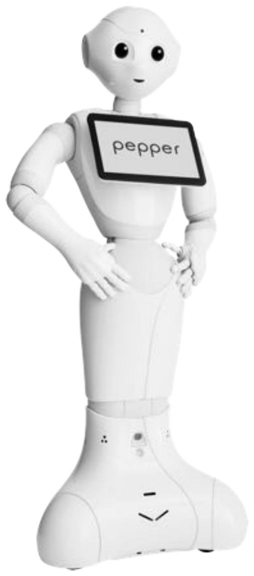


Figure 1.2 – The first social humanoid robot able to recognise faces and basic human emotions.[5]

RoboCup@Home[8] Social Standard Platform League (SSPL) competitions [PG18]. This results in teams relying on the Pepper robot in competitions for the years to come and due to its comparatively low price and low required maintenance and prior knowledge regarding for instance mechanical engineering, it is attractive to academic institutions and researchers [LW18].

Researchers are for instance developing extensions on the platform, such as simulation tools [LW18; BC19] or localisation and navigation systems [Góm+19; AKP19]. Some publications emerged from the RoboCup@Home competition [LW18; Lie+19]. Furthermore, authors survey Pepper's social acceptance in various scenarios [TTZ17; De +18; Bec+19] or use it as a platform to develop cultural competent robots[9].

CARE-O-BOT Care-O-bot [Kit+15] is a line of mobile robot assistants developed by the Fraunhofer Institute for Manufacturing Engineering and Automation in Germany. The first generation was introduced in 1999 [SM99], the second followed in 2004 [GHS04] and the third in 2009 [Rei+09] (Figure 1.3). While the first two generations focused on developing the technological foundation mainly in the areas of navigation, manipulation and perception, Care-O-bot 3 was already designed as a product vision for a future household assistant. The latest generation, Care-O-bot 4, was introduced in 2015 and "was designed as a gentleman robot that allows the users to relate emotionally and socially with it" [Kit+15] and, in contrast to earlier generations mainly acting as technology development platforms, Care-O-bot 4 is "providing the basis for commercial service robot solutions".[10]

Advancements from the earlier generations are the modular approach enabling a cost efficient, distributed and customised research, a higher agility (e.g. through the spherical joints), options for multimodal interactivity integrated in neck and head and better presence [Kit+15]. The modular approach implies that the Care-O-Bot 4 is actually not a fully configured product but rather

[8]`http://www.robocupathome.org`
[9]`http://caressesrobot.org/en/project/`
[10] `https://spectrum.ieee.org/automaton/robotics/home-robots/care-o-bot-4-mobile-manipulator`

a modular product family. It consists of 5 modules, a base, a torso, up to two arms, a sensor ring and a head, different options being available for each of them. Depending on the intended functionality and budget the customer can choose a configuration[11].

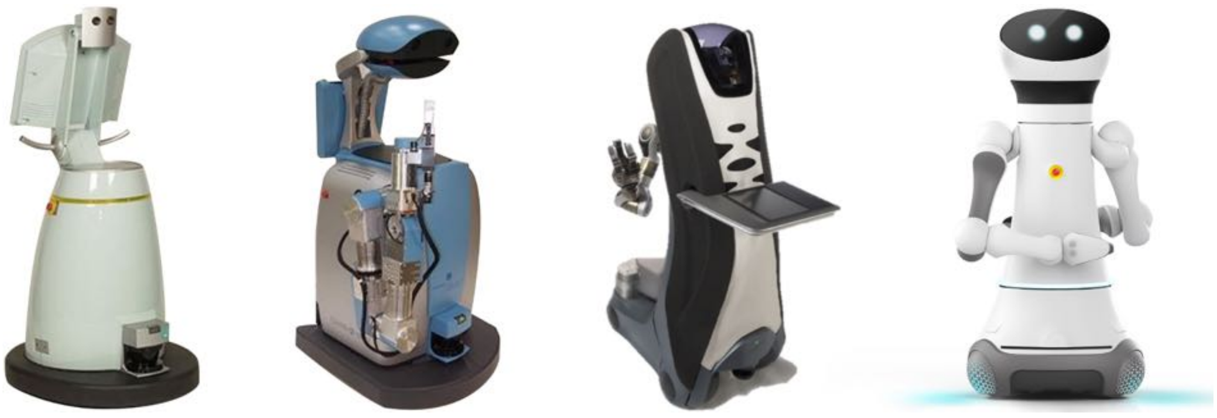


Figure 1.3 – The Evolution of Care-o-Bot. [Kit+15]

The hardware depends on the configuration, but fully configured the Care-O-bot 4 has up to 31 degrees of freedom, a 7" touchscreen, LEDs, speakers and microphones for multimodal interaction as well as optional sensors and cameras. The fully configured robot is 158cm tall, weighs 140kg and can move with 4,3 km/h maximum velocity. Developers can use the robot's APIs to extend its functionality and have access to open source repositories for different robotic frameworks (currently ROS and Orocos) containing all relevant hardware drivers, simulation models for the hardware components and application examples[12].

In 2015 the product design of the Care-O-bot 4 has been awarded with the renowned Red Dot Award: Product Design. As only 1,6 percent of all applications, Care-O-bot 4 received the recognition "Best of the Best" [13].

Its intended application domains include service (e.g. supporting patients and personnel in health care institutions, delivering orders in restaurants or providing reception and room service in hotels or for entertainment), industry (e.g. shelf-picking and commissioning in warehouses, loading and unloading of machines in manufacturing environments; the base module can also be used for transportation tasks) and home (so far still being a product vision of a mobile robot assistant, it could be employed to actively support humans in their daily life, e.g. by delivering food and drinks, by assisting with cooking or by cleaning) [11]. The first Care-O-bot 4 Paul was employed in retail and is still deployed at the electronics store Saturn in Ingolstadt since the end of October 2016[14]. The robot is also currently stationed at 3 more electronics stores in Germany and Switzerland as well as at the "Haus der Geschichte" in Bonn[15].

Furthermore, Care-O-bot 4 is available as a high-tech research platform. Two distributions can currently be found at Fraunhofer IPA in Stuttgart and one at the Korea Institute of Science

[11] `https://www.care-o-bot-4.de/`
[12] `https://www.care-o-bot.de/en/research.html`
[13] `https://www.care-o-bot.de/content/dam/careobot/de/documents/Pressemitteilungen/PM_COb4_RedDotDesignAward.pdf`
[14] `https://idw-online.de/en/pdfnews662791`
[15] `https://www.care-o-bot.de/de/care-o-bot-4.html`

and Technology KIST in Seoul[16]. Just as its predecessors, Care-O-bot 4 is relevant to researchers and academic institutions, being used as a platform to demonstrate concepts developed in the robotics domain [Ang+18; LVK18; Ham+19], to evaluate the user experience with the robot and its acceptance [SSB17] or for students' masters theses [Sep18].

1.2 Software Variability

As robots get increasingly important in all aspects of our lives, they need to fulfil many different requirements and are expected to have a great range of functions. Hence the hardware and software requirements have increased, but while hardware components continuously get more performant and efficient, software is turning out to be a roadblock, being described as ”the bottleneck of robotic systems engineering” [Gar+19] or ”a show-stopper towards the next level of robotic applications” [Sch+15].

Garcia et al. constitute three factors for this development: Firstly, there are no key stakeholders coordinating robotics software development, as it is just based on community efforts. Secondly, the high variability in robot technologies makes software reuse challenging and thirdly, many robotic developers are not specifically trained in software engineering. It can therefore be observed that there is a growing need for software engineering practices in the robotics domain.

It is elusive to define a general reference architecture, as robotic applications are so manifold. A robotic system has to serve many purposes and therefore has to be equipped with a mix of functionalities depending on factors such as the robot's mechanical structure, the task to be performed or the environmental conditions. This multitude of factors produces a multitude of variability dimensions calling for an appropriate variability management. [Gar+19]

In this section the basic concepts for software variability management will be presented.

1.2.1 The Notion of Software Variability

First of all to explain what variability management is and what it consists of, we must define variability in software systems. According to Svahnberg et al., ”[v]ariability is the ability to change or customize a system” and ”[i]mproving variability in a system implies making it easier to do certain kinds of changes” [SGB01]. They state that ”it is possible to anticipate some types of variability and construct a system in such a way that it facilitates this type of variability.”

Pohl et al. agree with this, stating that colloquially ”the term variability refers to the ability or the tendency to change” [PBL05]. They pose 3 questions to help define variability: ”What does vary?”, ”Why does it vary?” and ”How does it vary?”. They answer the first question by introducing the term *variability subject*.

Definition 2: Variability Subject

”A variability subject is a variable item of the real world or a variable property of such an item.” [PBL05]

A variable item could be the method of payment offered by a shop, while a variable property might be the colour of a car.

[16]`http://wiki.ros.org/Robots/Care-O-bot/distribution`

To answer the second question they explain causes for variability, such as different stakeholder needs, different country laws or technical reasons, as well as interdependent items, where the reason for an item to vary is the variation of another item.

Lastly, to identify the different shapes of a variability subject, they define the term *variability object*.

Definition 3: Variability Object

”A variability object is a particular instance of a variability subject.” [PBL05]

For the variability subject ”payment method” the variability objects are the concrete payment methods, such as payment by card, by bill or by cash. Variability objects for the subject ”colour” are for instance red, green or blue. [PBL05]

In the software domain, variability occurs for instance in the context of software families, which comprise a set of closely related software systems in terms of configurable functionality [SSA14]. Software product lines (SPL) are a type of software family (see Section 1.2.4), the products of a product line have shared commonalities and variable configurable features.

In software product line engineering the variability subjects and corresponding variability objects are embedded into the context of a software product line. They represent a subset of all possible variability subjects and objects from the real world respectively, that are needed to realise a particular software product line [PBL05]. To reflect this Pohl et al. define the term *variation point* accordingly.

Definition 4: Variation Point

”A variation point is a representation of a variability subject within domain artefacts enriched by contextual information. [PBL05]

Another way to describe variation points would be as ”one or more locations at which the variation will occur” [JGJ97].

Analogous to the definition of a variation points, Pohl et al. define the term *variant* as ”a representation of a variability object within domain artefacts” [PBL05], i.e. one concrete option at a variation point, such as red in the colour example. There is however no universal definition for the term variant. Besides the definition Pohl et al. and other authors [PBL05; SGB01; BB01; Anq+10] use, other publications [AK09; Ros+11; Hab+13; SSA14] use the term to describe a concrete, fully configured member of a software family, synonymous to the term product.

Definition 5: Variant

”A variant of a software family denotes the realisation of one member of the set of software systems encompassed by the family that is valid with regard to the configuration rules governing variability.” [SSA14]

The term variant will be used according to this definition. In the example of the car colour, a red car is a valid variant, for which the option red was chosen for the subject ”colour”.

Software product line engineering aims at deriving a valid variant of the software family, which in the case of SPLs is called a product. In order to derive a valid variant of a software family, the

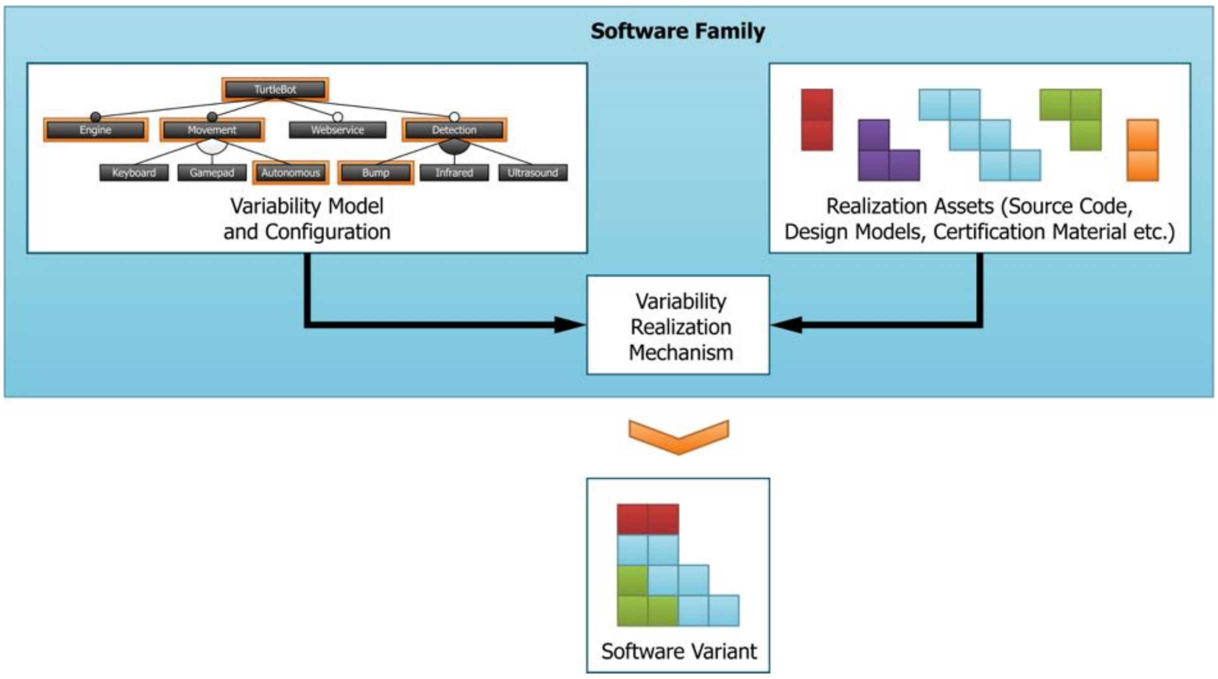

Software Family
TurtleBot
Engine
Movement
Webservice
Detection
Keyboard
Gamepad
Autonomous
Bump
Infrared
Ultrasound
Variability Model and Configuration
Realization Assets (Source Code, Design Models, Certification Material etc.)
Variability Realization Mechanism
Software Variant

Figure 1.4 – The variant derivation process [SSA14]

knowledge about the requirements on the product and the possible configurations must first be captured in a variability model (see Section 1.2.2). Realisation assets, such as source code fragments, act as building blocks that can be used to implement the variant. In the variant derivation process, a variability realisation mechanism (see Section 1.2.3) accepts a conceptual variability model and realisation assets as input to produce the realisation level counterpart, the variant (see Figure 1.4).

Problem Space, Solution Space and Configuration Knowledge Czarnecki et al. explain, that once the components (i.e. assets or artefacts) of a system are known, the abstract requirements can be mapped onto appropriate configurations of concrete components. The assembly of components to form a variant could even be automated using tool support. To do this configuration knowledge is required, mapping the problem space to the solution space (see Figure 1.5). [CE99; CE00]

The *solution space* contains the elementary implementation components for all possible systems, such as source code, design patterns, frameworks or documentation and certification material [CE99; CE00; SSA14]. They are designed to maximise their combinability, minimise redundancy and maximise reuse [CE99]. The elements of the solution space are required in order to assemble an executable software system, which is why they are mostly relevant to the technical stakeholders concerned with implementing the system [SSA14].

The *problem space* on the other hand consists of domain-specific, application oriented concepts and features as well as analysis methods, expressing the requirements of the different stakeholders. The conceptual configuration options can be captured in a variability model [CE99; CE00; SSA14]. Those elements are more suitable for non-technical stakeholders such as managers or end-customers, to help them get an overview of the configuration options or to perform configurations on a conceptual level without needing deeper knowledge of the realisation.

Instead of *problem and solution space*, Pohl et al. use the terms *problem and solution view* to refer to the requirements and the architecture respectively, pointing out the interrelation between domain requirements engineering and domain design [PBL05].

Lastly the *configuration knowledge* contains the configuration rules, which hold information about illegal feature combinations, default settings, default dependencies, construction rules and optimisations [CE99]. It spans both the problem and the solution space, but is being represented differently in each. It can be captured within a variability model in the problem space or as customisation of realisation artefacts in the solution space. To build an executable software system from the conceptual configuration of a variability model, a variability realisation mechanism has to be applied within a variant derivation process. This process assembles the realisation artefacts according to the conceptual configuration. [SSA14]

To manage the variability of a system, the knowledge one has about the system and the configuration rules must first be captured in a variability model. Secondly a realisation mechanism must be chosen to assemble the realisation artefacts. The following sections will present the most important variability modelling approaches as well as variability realisation mechanisms.

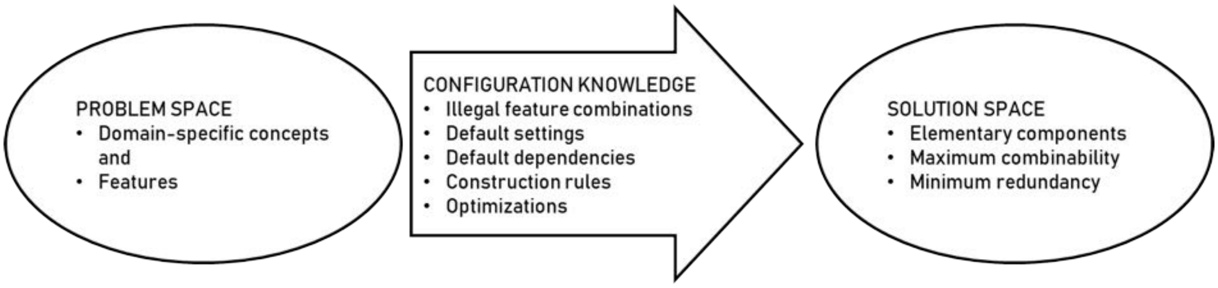


Figure 1.5 – The configuration knowledge maps the problem space to the solution space [CE99]

1.2.2 Variability Models

Variability models capture the knowledge about the commonalities and variability of the system's artefacts including all possible variants and the properties and dependencies specific to the organisation and the domain [Sch+12]. While they are generally described as being part of the problem space [CE99], Schaefer et al. are of the opinion that they can cover the problem space regarding stakeholder needs and desired features, as well as the solution space including the architecture and components of the technical solution [Sch+12]. Hence it is an individual decision whether the model ought to be truly independent of the realisation or whether it can contain information related to the architecture.

There are many approaches to modelling variability, a few of them were examined by Chen et al. in their systematic review of existing variability modelling approaches proposed until 2009 [CAA09]. They found, that out of the 33 examined approaches, 14 used feature modelling and 6 used decision modelling, making them the most influential concepts. Additionally the OVM approach proposed by Pohl et al. will be presented, as it differs significantly from the other two approaches in that it defines the variability information in a separate model [PBL05].

Despite the approaches being distinct, it can be found that the same system can be expressed by various models and that those models can be converted into one another. El-Sharkawy et al.

[EDS12] claim that decision models have an equivalent expressive power as feature models and Roos-Frantz et al. [RBR11] propose a transformation between Feature Models and Orthogonal Variability Models, which allows interoperability between modelling tools, particularly analysis tools.

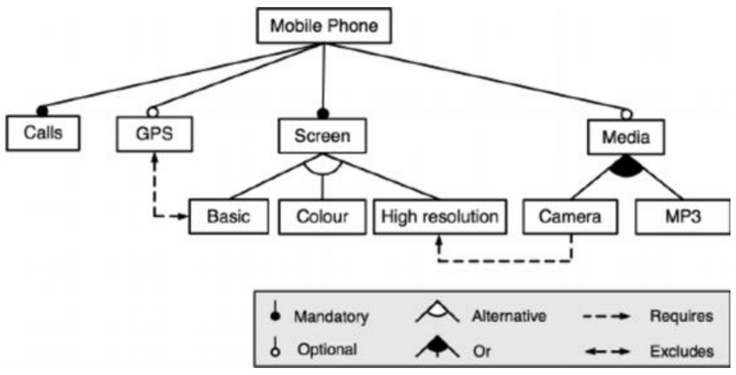


Figure 1.6 – A sample feature model [Ben+13]

Feature Models Feature models [Kan+90; CE00] are one of the most popular and best researched variability modelling languages with many extensions and analysis techniques built on top of them [Ber+13; CAA09; BSR10; HCH]. They were introduced in 1990 as part of the feature-oriented domain analysis (FODA) methodology [Kan+90] and gained popularity mainly because of their simple and intuitive notation [Ber+13]. A feature model contains all common and variable features of a set of elements as well as the dependencies between the variable features. It creates a hierarchical tree structure, making it easy to mirror the potentially also hierarchical structure of a software architecture. Cross-tree constraints among the features define all valid configurations, thus capturing the configuration knowledge. They can be described as tree-like menus of configuration options [Ber+13] and are therefore well suited to assist in the configuration process by displaying all available options.

Decision Name	Description	Type	Range	Cardinality	Rule
GPS	Should GPS be included?	Boolean	True | False		IF (GPS) THEN Screen != Basic;
Screen	Which screen should be used?	Enum	Basic | Colour | High Resolution	1:1	IF (Screen == Basic) THEN GPS = false;
Media	Are any media wanted?	Boolean	True | False		
Media_Types	Which types of media do you want to use?	Enum	Camera | MP3	1:2	IF (Camera) THEN Screen = High Resolution;

Table 1.1 – A sample decision model

Decision Models Decision models [MA02] focus on the process of configuration, which involves making decisions at each variation point in order to derive a specific variant. Decisions are expressed as a set of interconnected questions that need to be answered by choosing one (or more) out of multiple options and represent the configuration knowledge (see table 1.1). A valid variant that has been derived by applying the decision making process obeys the configuration rules captured in the underlying model. Being another popular variability model and serving as the base for many extensions [CAA09], decision models express similar configuration knowledge as feature models [EDS12] and just as FMs represent a valid variant space.

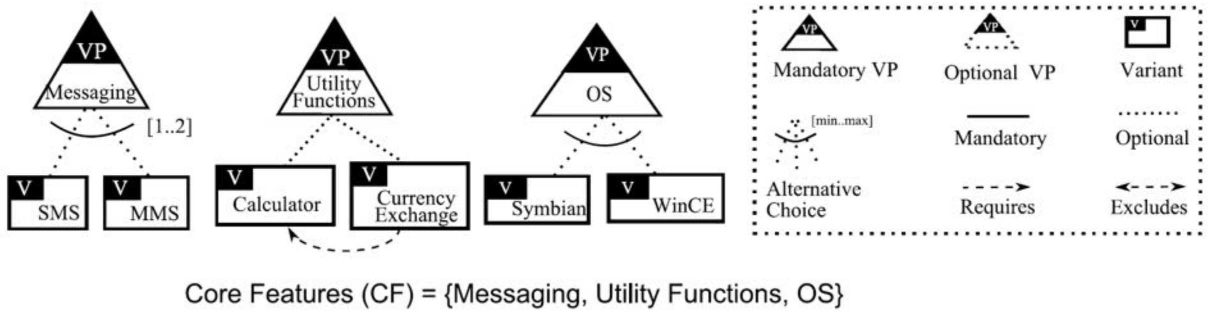


Figure 1.7 – A sample orthogonal variability model of a mobile phone product line [RBR11]

Orthogonal Variability Models (OVMs) Orthogonal Variability Models (OVMs) [PBL05] are based on the understanding that there are multiple dimensions of potentially orthogonal concerns involved in variability. For instance concerns regarding the functionality of a software family may be orthogonal to those regarding the platform (e.g., Windows or Linux) [SSA14]. Pohl et al. therefore propose OVMs, where variability information is defined in a model separate from the software development model. In contrast to feature models they only focus on variability and don't consider commonalities at all. They constitute a structure of variation points, specifying independent places for variation, and the constraints between them. A variation point specifies a list of all possible configurations for this point, all variation points and their respective options making up the variability model. The term variant is used for a specific configuration option, which is different to the use of the term in this thesis.

1.2.3 Variability Realisation Mechanisms

To assemble an executable software system from the conceptual configuration and to implement the configuration knowledge, a variability realisation mechanism [Sch+12] has to be employed. This results in one concrete variant of the software family. Three general types of variability realisation mechanisms can be distinguished: annotative, compositional and transformational. They will be introduced in this section regarding their basic principles as well as benefits and drawbacks.

Annotative Variability Realisation Mechanisms Annotative approaches include all possible variants of a product line within a single model consisting of realisation assets such as source code fragments. This model contains more than the elements required by any individual variant and is therefore often referred to as a 150% model. In order to derive a concrete variant, parts of the model have to be removed, as described in Figure 1.8, which is why these approaches are sometimes called subtractive or negative variability realisation mechanisms. Annotations on source code level regarding the product features describe which parts of the model have to be removed depending on the feature configuration. [Sch+12; KAK08; SSA14]

Benefits of annotative realisation mechanisms according to Seidl et al. are that no specific requirements on the structure of the realisation assets are posed and the specification of variations for a realisation asset in the same language as the asset itself is allowed. [SSA14] Additionally they can implement fine-grained extensions, meaning features with fine-grain granularity such as adding a statement in the middle of a method, better than compositional approaches. [KAK08]

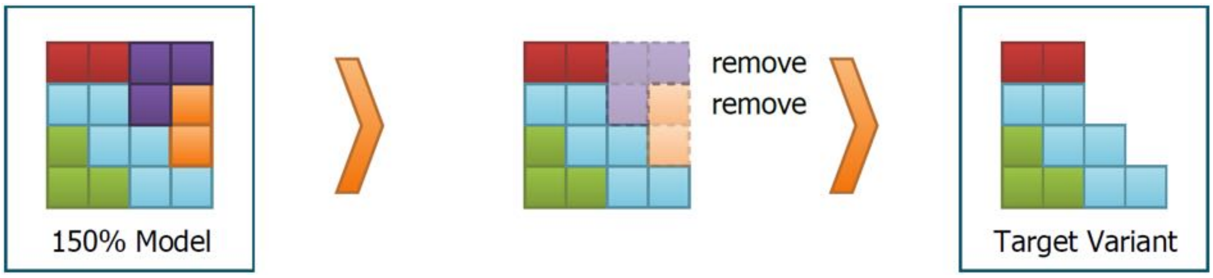


Figure 1.8 – ”Annotative variability realisation mechanisms assemble all possible variations of a realisation asset in a “150%” model for the software family and remove the parts that are not required by a particular configuration.” [SSA14]

On the other hand, there are also drawbacks. Due to the fact that the 150% model contains all possible elements, full knowledge of all possible configuration options is required and the model itself must be altered to add new options. Moreover, conflicting information that are correct for the software family but not for a concrete artefact of the realisation language can occur. It may also be difficult to express some variations such as alternate configuration options while satisfying the well-formedness rules of the language which would cause the model to be rejected by most tools. [SSA14] Additionally, Kästner et al. claim that annotations introduce readability problems by obfuscating the source code and are error-prone which raises the complexity. According to them annotative approaches also provide no perceptible form of modularity. [KAK08]

Compositional Variability Realisation Mechanisms In compositional approaches features are modelled as distinct modules that refine a core architecture. To generate a valid variant, a set of modules is added to a core model according to the chosen feature configuration, as seen in Figure 1.9. Therefore compositional approaches are also referred to as additive or positive variability realisation mechanisms. The modules correspond to realisation assets and are associated with the product features. The core model contains only artefacts that are common to all possible variants of a software family, hence it’s not necessarily a complete representation of a variant nor valid regarding syntax and semantics of the language, as missing pieces may only be added during the variant derivation process.

In contrast to annotative approaches not all possible variants have to be known in advance as new ones may be added as additional units of composition, provided that they can be implemented by more compositions and not by removal of artefacts. On the other hand this approach might pose requirements on the structure of the realisation assets, if validity regarding syntax and static semantics should be maintained, otherwise the units of composition or the common core model are invalid in isolation. This prevents the use of standard tools which may not be able to deal with the occurring conflicts and may also lead to comprehension problems for the developers. Splitting up functionality into small units increases scattering which may increase the necessary maintenance effort. [SSA14] Regarding granularity, Kästner et al. state, that existing composi-

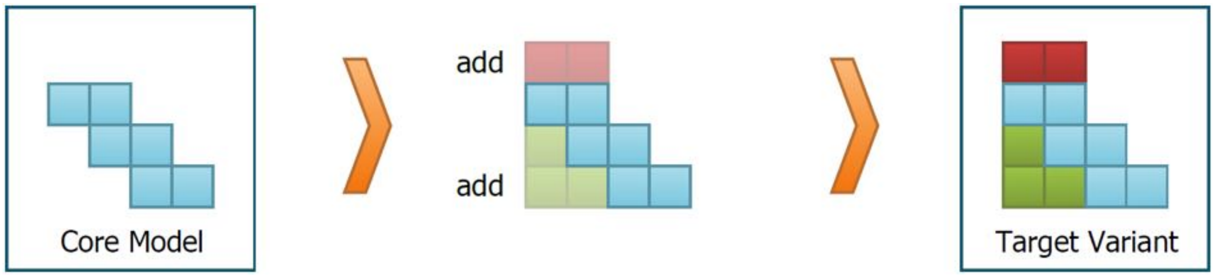


Figure 1.9 – "Compositional variability realisation mechanisms capture variations related to individual configuration options in separate units of composition external to the realisation assets and combine them with the core model when deriving a variant." [SSA14]

tional techniques usually only allow coarse-grained extensions, such as adding entire methods. For fine-grain extensions, workarounds are required which may raise the implementation's complexity. [KAK08]

Transformational Variability Realisation Mechanisms Transformational approaches create a variant by transforming a base variant into a target variant as seen in Figure 1.10. Those transformation operations can be described as adding, modifying and removing individual elements of the underlying language. They can also be grouped into transformation modules, which enables the use of control flow logic, but generally a list of operations will be executed. Each feature might be realised in a module, but modules may also be more fine-grained representing a fraction of a feature, which is useful when different features require the same transformations. To derive a valid target variant the transformation modules required for the chosen configuration must be applied to the base variant, which is a valid variant of the software family. Deciding which variant is going to be the base variant is basically an arbitrary choice, though different factors might be considered: Firstly, the chronology of development may cause one variant being developed first, acting as a base for future variants. Secondly, the variant with the most significant overlap with the other variants may be chosen as the base variant while the others are specified using transformation modules. Thirdly one might also choose the variant as the base variant that is the most suitable for the job. [SSA14; Sch+12]

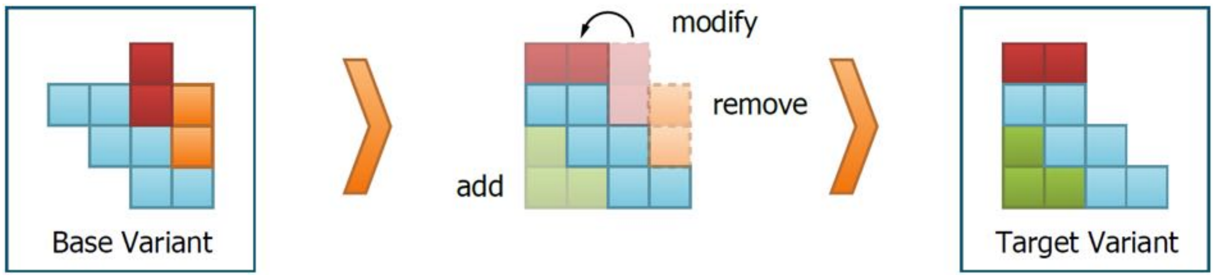


Figure 1.10 – "Transformational variability realisation mechanisms transform a base variant of the software family to a target software system by adding, modifying and removing parts." [SSA14]

Transformational approaches don't require all features to be known in advance and allow for features to be added or altered which makes them more flexible than compositional and annotative approaches, where only addition or removal are allowed respectively. Additionally, standard tools can be used for inspecting the artefacts, if the configuration knowledge is sound, as in that case the input and the output of the variability realisation mechanism are valid regarding the syntax of their respective language. Lastly the process of deriving additional variants by transforming an existing valid variant aligns well with the practice of companies of first developing an individual software solution on request and later on transforming it into a configurable software family that can be sold as customised products off the shelf. [SSA14]

1.2.4 Software Product Lines and Ecosystems

This section presents the most important concepts of software families, a term which is often used as an umbrella term for the different types of variable software systems. Parnas defines them ”as a set of programs whose common properties are so extensive that it is advantageous to study the common properties of the programs before analyzing individual members”. He concludes that ”software family engineering assumes that there exists more commonality than variability in a family of software systems”. [Par76]

Software Product Lines One concept of software families are software product lines, which Bosch describes as ”a set of systems which share a common software architecture and set of reusable components” [Bos00]. The goal of a software product line (SPL) is to maximise the benefits of the commonalities between the products in the family while providing sufficient variability for each family member [Bos00] and to minimise the cost of developing and evolving software products that are part of a product family [SGB01]. Pohl et al. [PBL05] define SPLs using the concepts of platforms and mass customisation. They draw parallels between the development of the production of manufactured goods and software: Formerly goods were handcrafted individually for the customer, but the increasing wealth meant that more people were able to afford those goods and the rising demand could not be met. In the automobile domain Ford introduced production lines enabling the production of affordable cars for the mass market. This however reduced the possibilities for diversification. Customised and mass produced products can be found in the software domain as well, being denoted as individual software and standard software. Each has its drawbacks, as the former is quite expensive and the latter lacks sufficient diversification. Taking into account the demand for individualised product means finding a middle ground between mass production and individualisation – mass customisation.

Pohl et al. use the definition provided by Stanley Davis in 1987:

Definition 6: Mass customisation

Mass customisation is the large-scale production of goods tailored to individual customers' needs. [Dav87]

Similarly Tseng and Jiao state that: “Mass customization aims to provide customer satisfaction with increasing variety and customization without a corresponding increase in cost and lead time” [TJM96].

Furthermore Pohl et al. present platforms as a prerequisite for mass customisation.

Definition 7: Platform

A platform is any base of technologies on which other technologies or processes are built. [PBL05]

In the automobile domain manufacturers started introducing common platforms for the different type of cars they were offering, by planning in advance which parts would be used in which car model. The platform provided a structure for the major components, which were usually the most expensive subsystem. This approach enabled companies to offer a larger variety of products while reducing production costs at the same time. Combining the concepts of mass customisation and common platforms enables the reuse of a common base of technology while offering products close to customer requirements. [PBL05]

Software product line engineering is defined accordingly.

Definition 8: Software Product Line (Engineering)

“Software product line engineering is a paradigm to develop software applications (software-intensive systems and software products) using platforms and mass customisation.” [PBL05]

SPLE follows a centralised approach. One single company maintains the product line, having control over which asset combinations are considered valid variants [Bos09; PBL05]. Extensions are developed in-house or by sub-contractors. This results in the end user not necessarily being aware of the product line nature of their software, as they purchase it as an individual product [Bos00; PBL05]. An exception would be the involvement of the customer in the configuration process if this brings value to the owner of the SPL [SSA14]. As the product line is maintained by a single vendor, this vendor has full knowledge about the variant space at any given time. This enables the construction of a central variability model and the use of all presented variability realisation mechanisms [SSA14].

Examples for the use of software product lines include the Gasoline Systems Engine Control Software from Bosch[17] and Boeing’s Bold Stroke Avionics Software Family[18]. The company Philips even employs multiple product lines, as their Product Line of Medical Systems[19], their Product Line of Software for Television Sets[20] or their Telecommunication Switching System[21].

Software Ecosystems Another concept to model software families are software ecosystems (SECOs) [Bos09]. Similarly to SPLs they model commonalities and variabilities of a software family. However, in contrast to the centralised approach of the SPL, a SECO has in its centre a technological platform which is maintained by a single company and that can be combined with various extensions by third-party contributors to form a variant of the software family [Bos09; McG09; Ber+14]. The combinations of assets that are considered valid variants are captured in the configuration knowledge, which for SECOs is usually not represented in a central variability model on a conceptual level [Ber+14] but is part of the realisation assets themselves [Lun09].

[17]`https://splc.net/fame/bosch/`
[18]`https://splc.net/fame/boeing/`
[19]`https://splc.net/fame/philips-medical-systems/`
[20]`https://splc.net/fame/philips-software-for-television-sets/`
[21]`https://splc.net/fame/philips-telecommunication-switching-system/`

Due to the open concept and there not being a central variability model, the variant space is usually not fully known [Bos09]. The open variant space affects which variability realisation mechanisms can be implemented [Ber+14]. As annotative variability realisation mechanisms require full knowledge of the variant space they can't be implemented for SECOs. Compositional and transformational approaches however don't have this restriction and can thus be applied. [SSA14]

Berger et al. present some well-known examples of SECOs, such as the Eclipse IDE and its extensions, the Android platform and its apps for mobile devices as well as the Linux kernel with its user-contributed extensions [Ber+14].

Bosch [Bos09] explains that for a transitioning from a software product line to a software ecosystem the company must open up the application for external developers, for instance through provisioning APIs [Bos09].

On a social level, the former product line owner takes the role of the platform leader, being responsible for the success of the SECO by planning future directions of development, bringing together developers for extensions, creating suitable variation points and ensuring economical soundness for all contributing participants [SSA14]. On a technical level the platform leader must ensure that the contribution effort for third-party developers is as low as possible for instance through providing stable and expressive APIs, thorough documentation or allowing the use of generic IDEs [Bos09].

There are numerous reasons for a vendor to be using a SECO instead of an SPL approach. Besides the use of a SECO being an effective mechanism for mass customisation, the maintaining company might not be able to satisfy customer needs on their own anymore. This would be the case if there are too many or too specialised feature requests for the vendor to implement in a reasonable amount of time and with research and development effort offering an acceptable return on investment [PBL05; SSA14]. Lastly there are also social aspects to consider. A higher number and broader diversity of developers increases innovation and thus leads to a higher interest by customers. Additionally a successful platform will make customers as well as developers depend on it, securing long term success. [SSA14]

1.2.5 The Need for Software Variability in the Robotics Domain

According to Schlegel et al., "mastering the software complexity becomes pivotal towards exploiting the capabilities of advanced robotic components and algorithms" [Sch+15], with the robotics domain coming with extra challenges. They claim the major difference to other domains being the dependence on run-time exploitation of (software) variation points in order to manage the robot's resources "to face open-ended environments and to meet non-functional requirements" [Sch+15]. As in an open-ended environment the number of possible situations can't be foreseen, a robot must be able to make appropriate decisions at run-time. This requires the move from a code-driven approach to a model-driven one, which is formalised and can therefore be automatically processed by the robot at run-time. [Sch+15; BCH15b]

Dynamic variability techniques offer a solution for coping with those run-time requirements [BCH15b]. However, up to this point dynamic variability remains an unsolved and ongoing common challenge, though different approaches by different groups exist [Gar+19].

Besides the aspect of run-time exploitation, a model-driven approach can also help reduce the complexity exposed to the designers and system integrators, making it simpler to express variability at design time using easier to understand domain-specific languages (DSLs), enabling tool support for automated consistency checks or constraint fulfilment and providing a black-box view of the software building blocks with left-open bindings. Based on the information available at runtime, the robot can bind the left-open variability to fulfil the to be applied policies. Consequently software models support the robot in it's decision making process [Sch+15; Lot+14].

Another reason why software variability plays a big role, not only in the robotics domain but also in the general area of software development, is that it is expensive to develop from scratch. As software systems get bigger and more complex, reuse of code is necessary to make a system manageable and maintainable for the developers [BCH15a].

1.3 Summary

To summarise, there is a great demand for adaptive assistance robots to support care patients and care givers, as well as help elderly people to maintain their autonomy longer. Some robot solutions are already available, but due to the often limited function range and significant financial investment they are not yet widely adopted in the health care domain.

As requirements in this domain differ from the concerns relevant in other domains where robots already play an important role, it proves to be difficult to benefit from the advances made in those fields. Though hardware developments can be transferred to another domain, the development of appropriate software proves to be a roadblock. It is specific to the context a robot acts in and therefore not reusable in a different context, furthermore it is quite difficult to cope with the variable runtime requirements the interaction with real people and the real world poses.

Applying methods from the software variability domain in the robotics domain can provide the means to handle the requirements that are posed to assistance robots, such as considering user preferences or requirements when executing tasks or reacting to unknown environments at runtime. Considerations regarding this concept exist, but so far coping with variability remains an unsolved challenge.

2 Developing a Concept for Variable Process Chains

In this chapter a running example will be designed to provide a use case motivating this work. The in Chapter 1 presented variability models (Section 1.2.2) and variability realisation mechanisms (Section 1.2.3) are being analysed regarding their potential application for the use case.

Subsequently a concept for the execution of variable process chains is described and the Android based assistance robot Loomo is introduced as a platform for a prototype proving the feasibility of the concept. Lastly the foundation knowledge needed to implement this concept in Android is given.

2.1 Designing a Running Example

A residential group is home to 4 elderly people between the ages of 65 and 85. They live on their own, with a nurse checking in every morning to help with medication and a housekeeper coming in every other day. A catering service will provide them with lunch and dinner while the housekeeper takes care of grocery shopping.

Additionally they are supported by a social assistance robot, helping them out with their everyday tasks. This robot is equipped with wheels, a touch screen, LEDs, speakers, a camera and is based on the Android operating system.

Each of the residents has different preferences and impairments, which means the assistant robot must be able to adapt to everyone's needs accordingly. This involves the use of suitable input and output modalities according to the requirements of each user. Possible input modalities are speech, gestures or the touch screen, while output modalities may include speech, visual effects using the LEDs, the screen as well as movements (e.g. spinning around).

The following resident personas have to be to be considered:

- Charlotte was born mute and uses sign language to communicate. If she is around, the robot must pay attention to her and be ready to accept commands in sign language.
- Kurt is visually impaired, therefore the robot shouldn't rely on visual feedback, but instead provide information using speech.
- Max recently got hearing aids, but he doesn't always wear them. The robot can try to talk to him, if he does not respond, it will try talking louder or attract the user's attention with blinking LEDs or moving in the field of view and showing text on the display.
- Lisa is epileptic, so blinking lights should be avoided when interacting with her.

Lastly, the robot may apply its default interaction modalities when communicating with the staff, which can include a combination of all possible modalities to provide a good user experience.

Examples of tasks The robot's mission is to perform tasks for the residents. In this paragraph three examples for such tasks will be given to provide context and a reference point for the following considerations. It must be noted that the scenarios provided do not include every possible aspect that has to be considered in the implementation of a particular task in a real world scenario to keep the running example manageable and understandable.

The first task an assistance robot might have to perform is to open the door on a user command. The robot might also react directly to the doorbell, but this would raise the complexity of this example and is therefore not considered. The task "*Opening the Door*" is comprised of several subtasks (Figure 2.1). After receiving the command, the robot must drive to the door from its current position, open the door (possibly with the assistance of smart home technology) and analyse the situation on the other side of the door. Multiple scenarios can be imagined. A possible situation is that there is no one on the other side of the door, maybe because the visitor got impatient and walked away or it was a prank. In that case, the robot would close the door and return to the user who gave the command, telling them nobody was at the door. The more likely scenario is that there is a person or a group of people the robot may greet and ask how it can help them. The robot might also try to recognise the visitor(s) to greet them with their name.

From this point on the scenario can continue in many ways. If the visitor is a guest, the robot can let them in, if the visitor turns out to be a package deliverer, the robot will perform the task of accepting and delivering a package (Figure 2.2). However, there might also be potentially unwanted visitors, such as sales people, beggars or even burglars. In those cases the robot must be able to deal with an unexpected request of the visitor or even with an attempted break-in.

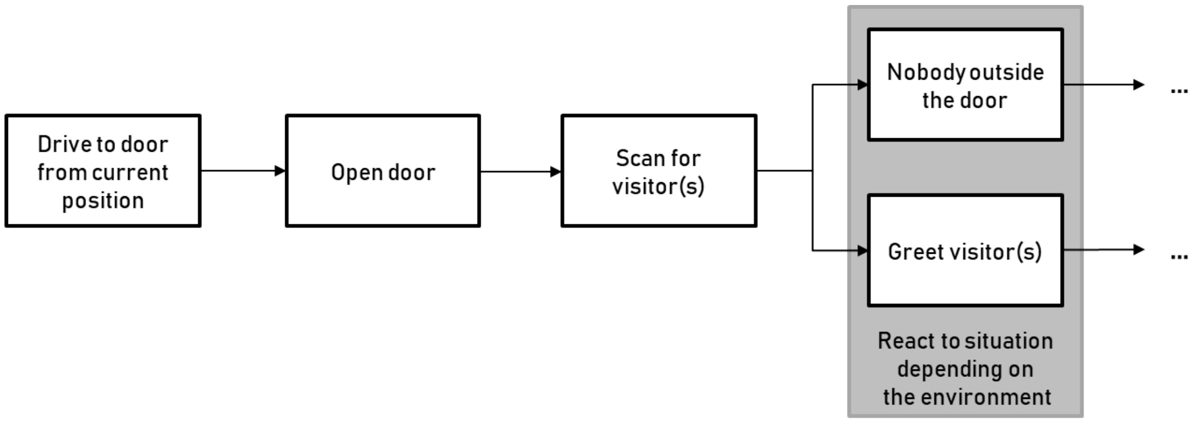

Drive to door from current position
Open door
Scan for visitor(s)
Nobody outside the door
Greet visitor(s)
...
...
React to situation depending on the environment

Figure 2.1 – When opening the door, a robot must be able to cope with uncertain situations.

If the visitor turns out to be a package deliverer, the robot will perform the task "*Delivering a Package*" (Figure 2.2). Firstly it will take the package from the deliverer and then close the door. The robot will then scan the package for the address to determine the receiver of the package. If the address is not readable, the robot might put the package at a designated space for packages, such as the kitchen table. If the address is readable, the robot might know whether the receiver prefers packages to be directly brought to them, if they are home, or whether they like to have them placed in their room. If the former is the case, the robot will try to locate the recipient and, if successful, deliver the package, else it will put the package at a designated place.

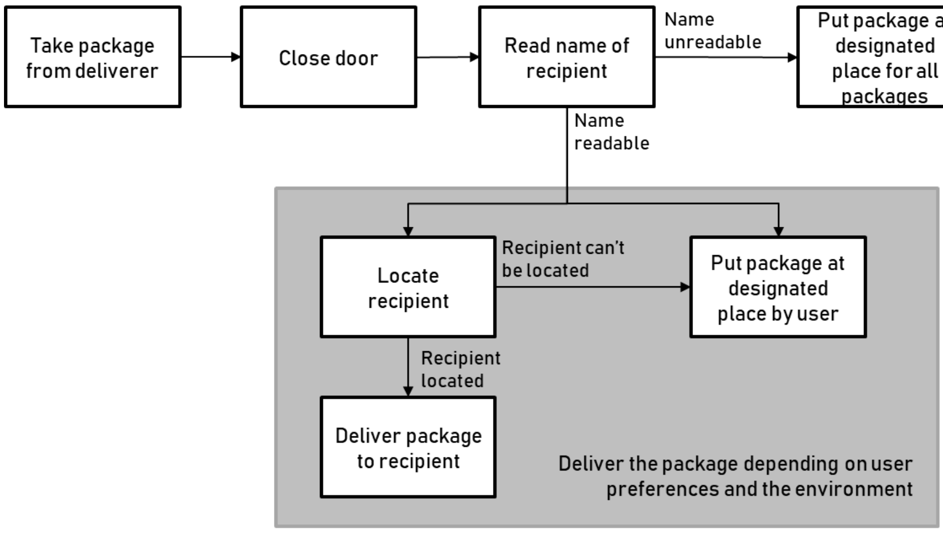

Take package from deliverer
Close door
Read name of recipient
Name unreadable
Put package at designated place for all packages
Name readable
Locate recipient
Recipient can't be located
Put package at designated place by user
Recipient located
Deliver package to recipient
Deliver the package depending on user preferences and the environment

Figure 2.2 – The robot executes the task of delivering a package differently, depending on what the user prefers.

Another task a robot might perform in order to assist people in their daily life is "*Fetching an Item*". This item might be a pair of glasses from the bedside table or a glass of water from the kitchen. Depending on the item there might be additional subtasks necessary to complete the job, for instance filling the glass with water, but those will not be considered in this example. On an abstract level, the first part of this task is for the robot to locate the item and drive to its position. Then the robot has to pick it up and return to the person that gave the order. To get the person's attention and tell them the task has been executed, the robot has different output modalities, including acoustic means, such as speech at different levels of volume, or visual means, such as LED effects and text displayed on the display. When the robot informed the user about the successful execution of the task it can hand over the item and the task is completed.

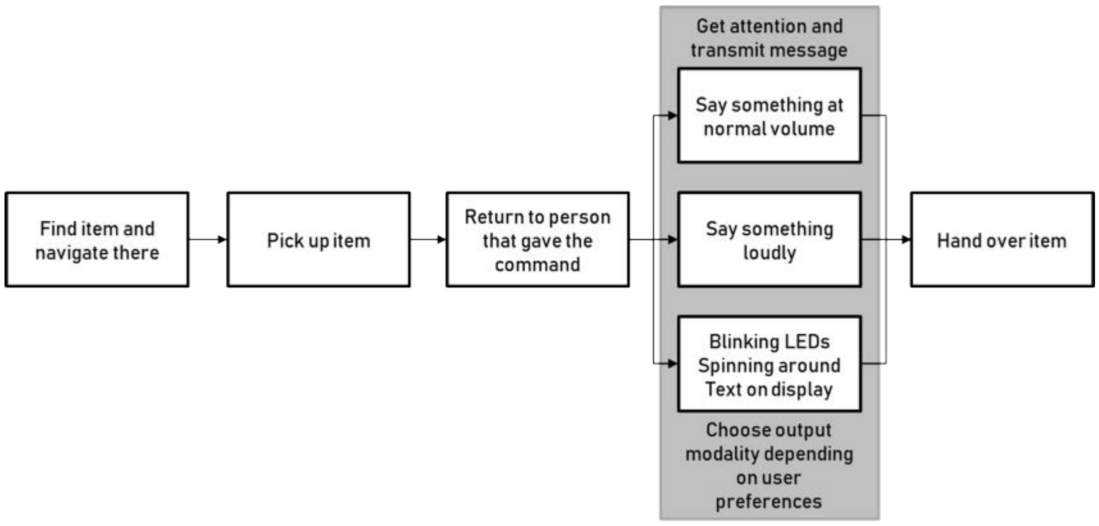

Get attention and transmit message
Say something at normal volume
Find item and navigate there
Pick up item
Return to person that gave the command
Say something loudly
Hand over item
Blinking LEDs Spinning around Text on display
Choose output modality depending on user preferences

Figure 2.3 – When delivering a fetched item the robot gets the user's attention using their preferred output modalities.

The goal of this thesis is to propose a way of performing these processes according to the user's needs and preferences. Each process (i.e. task) can be broken down into atomic subtasks, that need to be assembled in order to fulfil the specified task. During the configuration process the user's preferences must be considered, for instance regarding their preferred output modalities as in the ”Fetching an item” example or where they want their packages to be put down as in the ”Delivering a package” example. Furthermore a robot should be able to react to its environment as when ”Opening the door”.

2.2 Identifying Feasible Variability Methods

Assistance robots must be able to adapt the execution of tasks to a potentially dynamic context, while considering user requirements and/or the environment as the running example showed. A way to cope with this need for adaptability is the employment of a variability management.

Tasks can be modelled as sequences of atomic subtasks as seen in Section 2.1. To build such a sequence, variability management methods, such as the ones presented in Chapter 1, can be applied. The configuration knowledge about which sequences of subtasks are considered valid tasks can be captured in a variability model (see Section 1.2.2). A variability realisation mechanism (see Section 1.2.3) can be applied to derive a concrete sequence of subtasks, i.e. a process chain. The output of this process is an ordered list of concrete subtasks. Applying this concept to the platform Android, the task list is an object containing the order of the apps (and potentially needed arguments, such as the position to go to) that have to be called sequentially in order to perform the task specified. This object is going to be referred to as the *task object*.

A variability management abstractly describes all means of handling variability. This thesis focuses on proposing the structure of a task object and how it can be executed in Android. Additionally, this section discusses, which choices future implementations might make concerning variability models and realisation mechanisms, and which methods can therefore be expected to be used for the construction of process chains for assistance robots.

Variability Model To be considered a feasible variability model in the context of assistance robots a model should fulfil several requirements:

- It should be able to cope with an open variant space, as the Android platform is a software ecosystem which allows the addition of new apps in order to extend the robots function range [Ber+14]. It should therefore be possible to easily add new apps/functions to the robot, without having to restructure the model.
- It should be able to fully model the robot's function range, in order to derive a process chain containing those functions.
- It should be suitable for modelling the architecture of typical software projects and ideally be well established in that area.
- Additionally it would be favourable if the model is already widely used/well-known to developers, to minimise the training period.

Feature Models are the most popular and most used variability models [CAA09]. This implies a high chance for developers already being familiar with the model, moreover it is quite intuitive to read [Ber+13]. Many researchers have adapted it developing extensions to express what they needed the model to express [CAA09]. As they capture all common and variable parts of a system, feature models are well suited to display the full function range of a robot in a single model. However, feature models in their ”pure form” are not suited for open variability, as they require full knowledge over the variants in advance. The concept of *Multi Software Product Lines* [RS10] introduces the opportunity to enable open variability by composing a Multi Product Line of several product lines, each of which can be represented by a feature model. Permitting the exchange of product lines (and feature models respectively) allows for open variability. With this extension feature models are very likely to be used as a variability model, as they are well suited to portray a complete system with all dependencies between its components that is still intuitive to read.

Besides Feature Models, *Orthogonal Variability Models* may also be a plausible choice. They focus on the variation points, which are crucial in the process of configuring a task object. Some parts of a task object may always remain, for instance the basic structure. Other parts, as the specific subtasks (i.e. the apps), will have to be chosen based on user requirements or the current environmental conditions or events. For each variable part of the task object a variation point can be introduced. OVMs have an open structure, consisting of several disjoint tree-structured variation point diagrams. New diagrams can be added without changing the existing model. The focus of OVM on the variable parts doesn't lend itself to representing a complete system as well as Feature Models do, though extensions could be added to include commonalities. They are also not as well-known and widespread as FMs. As they are otherwise suitable, they must not be ruled out.

Decision Models are another widespread modelling approach and have been extended many times [CAA09; Cza+12]. They focus on the process of configuration and don't represent the complete structure of the underlying system. They can be designed to allow the addition of decisions and therefore an open variant space. As OVMs they concentrate on the configurable (variable) parts of a system and don't include commonalities [Cza+12]. The approach to modelling is completely different to the other models presented, focusing on the process instead of the features. Decision models can therefore oftentimes not be implemented well in typical software projects, in contrast to FM and OVM which both focus on the available components of a system. Instead of being used to realise the software architecture, they are often employed to assist decision making processes in other parts of the software development process, for instance in software development project management [Ngu06], software maintenance and optimisation [BKM16], testing [RS18] or business process modelling [Wes19].

	supports open variant space	**can model full function range**	**suitable for architecture modelling**	**widely known (optional)**
FM	yes, with extension (MSPL)	yes	yes	yes
OVM	yes	no extension possible	yes	no
DM	yes	no	no	yes

Table 2.1 – Comparison of variability models

Variability Realisation Mechanism The need for an open variant space remains when choosing a variability realisation mechanism. Therefore the use of an *Annotative Realisation Mechanism* can be ruled out, as it requires full knowledge over the variant space. A *Compositional Realisation Mechanism* however doesn't require all variants to be known in advance and aligns well with the concept of the task object in Android. The core model would be the empty task object, which is filled with the chosen apps as the task sequence is constructed. The apps, or rather the information about the apps, correspond to the units of composition.

Using a *Transformational Realisation Mechanism* is another option, as it also doesn't require full knowledge about the variant space. A ready configured task object would serve as the base variant and depending on the final configuration certain subtasks would have to be exchanged. If a task object has to be modified to represent the same task as the base variant, only with a few modifications such as exchanging a few subtasks due to different user requirements, this mechanism lends itself. If it however needs to be transformed to represent a different task, presumably all subtasks need to be exchanged. As this is not very straightforward, a compositional realisation mechanism may be preferred, still the possible use of a transformational realisation mechanism should be kept in mind.

2.3 Describing a Concept for Executing Process Chains

The previous section talked about suitable variability models and realisation mechanisms for constructing process chains. The goal of the thesis however is to describe a concept for executing those process chains in Android. In this section, a high-level concept for executing process chains on a platform allowing the use of modular components is going to be developed.

As described in the running example (see Section 2.1) a task can be broken down into reusable subtasks. In a technical system a subtask can be executed by a *module*, each of which is independent from the remaining modules to facilitate the addition, removal and exchange of subtask modules. To fulfil a task, the necessary subtask modules need to be executed in the correct order. This sequence of subtasks is referred to as a *process chain*.

The concept of process chains in Android has characteristics of a product line as well as of a software ecosystem approach. On one hand, the SECO element is introduced by the Android platform and the requirement, that it should be easy to add, exchange or delete subtask modules, which leads to the need for an open variant space. This is typical for SECOs and encourages innovation by allowing third-party developers to add to the pool of available components.

On the other hand, the actual configuration of a process chain is more similar to a product line approach, the process chain being the product that is going to be derived. All in all one can say, that creating process chains for an assistance robot in Android relates to deriving a product like in a software product line approach, but in an ecosystem environment.

The running example showed the modularity of complex tasks an assistance robot is expected to perform. In this section, an abstract concept for the execution of process chains is going to be developed, which is independent of the technical realisation. This may allow it to be adapted for Android in a different way than described later in this thesis or even be used for a different system. Moreover, a more abstract concept is easier to grasp.

The first thing that is needed for executing a process chain is a data structure that holds all necessary information about it. The process chain is basically a list of modules. This list may include the names or paths of the modules through which they must be addressed, as well as potentially more information that needs to be passed to the module in order for it to execute its task correctly. An example would be a module responsible for finding the robot's path to a certain destination. In order to do so, the module needs to know the destination. For the system to keep that information, the task list holding the module paths and parameters must be encoded in some data structure that may easily be passed from component to component. This is the *task object*.

In order to react to a user command, a component that listens for commands is required. Commands may come in different forms through different input modalities, such as speech, gestures or through the robot's touch screen interface. Ideally the listener component accepts all of those modalities, as this makes employing the robot accessible for everyone. When a command is received it must be analysed for information regarding the task and user requirements. The users might phrase their requirements explicitly, or the robot may recognise the user and has information about the user's preferences in its database.

The next step is to use the information about which task should be performed and the user preferences to construct a suitable process chain which fulfils the task as well as the requirements. Speaking in terms of software variability, a *task configurator component* takes the configuration knowledge, which includes the information specific to the analysed command as well as the knowledge about valid component combinations, and the realisation assets, which in this case are the modules or even more specific, the names or paths of the modules, and applies a variability realisation mechanism in order to construct a task object (c.f. Figure 1.4). It can be assumed that a compositional variability realisation mechanism is employed, as it is intuitive to have the structure of an empty task object as the core component whose slots are filled with the chosen components.

The configurator must have knowledge about all available components and how they can be combined to form valid process chains. This configuration knowledge can be captured in a feature model or an orthogonal variability model. Using this knowledge the configurator derives a variant, which in this case is the task object. The need for an open variant space demands that the pool of components may be changed at any time. However, in order to include a new component in a process chain, the component must be included in the variability model. This implies that a new component must be registered with the maintainer of the configurator module.

After the task object was constructed, the task should be executed. In order to not have too many components and keep the system as small as possible, the previously described listener component is the one that receives the task object back from the task configurator and starts the first component on the list. As this component is now the central component of this concept managing incoming commands, it is from now on referred to as the *manager component*. In theory the configurator may also be a part of the manager, however, as the functionality of the configurator is very complex and must potentially be touched often if new assets or configuration knowledge arise, it is better to practice separation of concerns.

To execute a task, the manager must read the first entry from the task list. As the process chain should be executed automatically, each module must know its successor and be able to call it after finishing its own subtask. The information is passed in form of the task object from module to module. As a module might appear multiple times in the task list, representing different subtasks with different parameters, the information which one is the current entry on the task list must be passed alongside the task object.

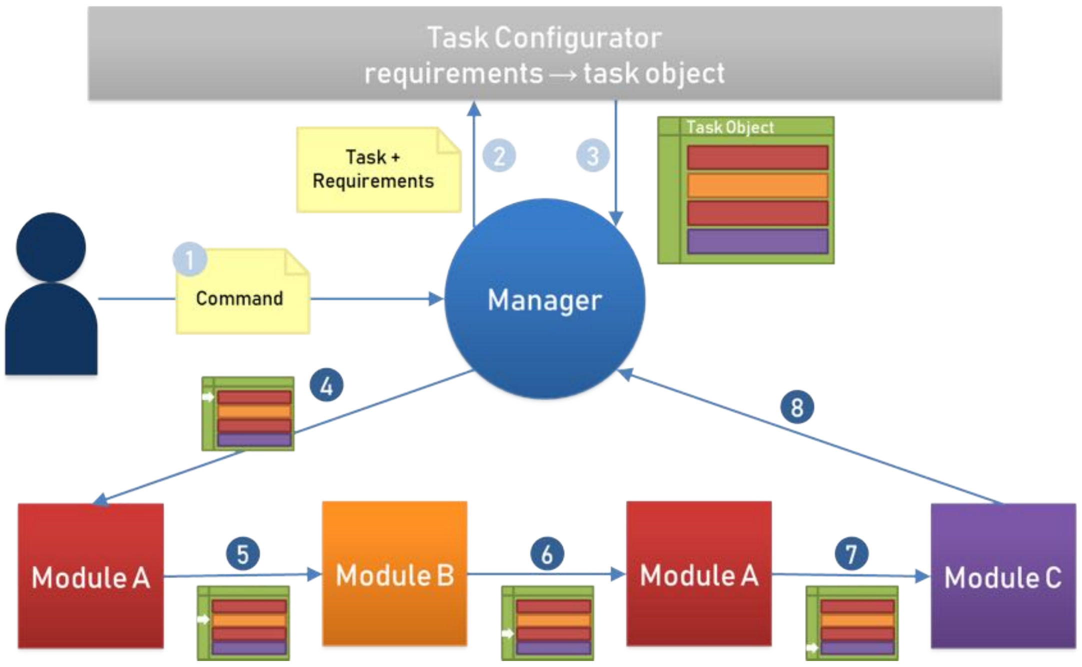


Figure 2.4 – A concept for executing process chains

The Execution of a Process Chain Figure 2.4 visualises the concept that is going to be explained in more detail now. The process starts with the user giving a command to the robot (Step 1). This includes the task the robot should perform and possibly explicitly phrased requirements, such as the desired speech volume. The manager receives the command and analyses it for those information. Additionally, the manager may enrich the information with details from its database, such as user specific needs as described in the running example, or from its sensors, for instance regarding environmental conditions. The manager then takes these information and hands them to the *task configurator* (Step 2).

According to the given information, the configurator will assemble a configuration of concrete subtask (*modules*). It can use software variability techniques to do this. The result of this process (Step 3) is a *task list* encoded in a *task object* containing a list of concrete modules (and information on how they can be addressed as well as further details needed for their execution) that must be executed to fulfil the task.

In the example in Figure 2.4 the task list comprises four modules, Module A is included twice. The manager will start the execution of the first module on the list. At the same time it passes the task object to the module, as well as information as to at which position on the task list the module is located (Step 4). The position on the task list is depicted by the white arrow on the task object. This is critical, as a module can be in the task list multiple times but with different details. A speech synthesis module for instance might be called multiple times for a task but is required to output a different text each time.

When the module completed its subtask, it will start the next module on the list and hand over the task object (Step 5). Before doing so it updates the pointer to the next module in the task object. This process is continued until the last subtask was executed, the last module will restart the manager so a new command can be accepted (Step 8).

While the configuration part of this concept (Steps 1-3) is crucial for the implementation of variable process chains, it is not the focus of this thesis. Instead the focal point is how the result of the configuration process, the task object, can be executed (Steps 4-8) in the context of assistance robots. The Android based assistance robot Loomo platform will be used as a prototype for a proof of concept.

2.4 Introducing the Assistance Robot Loomo Platform

The feasibility of the concept developed in this thesis will be shown on the assistance robot Loomo Platform (Figure 2.5), which resulted from a collaboration of Segway Robotics and Intel. The American manufacturer Segway Inc. is best known for their two-wheeled, self-balancing personal transporter of the same name and was acquired by the Chinese company Ninebot in 2015. Their first robot Loomo was initially shown in 2016 at CES (Consumer Electronics Show)[1] and a developer version was published subsequently[2]. After a very successful crowdfunding campaign it became commercially available in 2018[3].

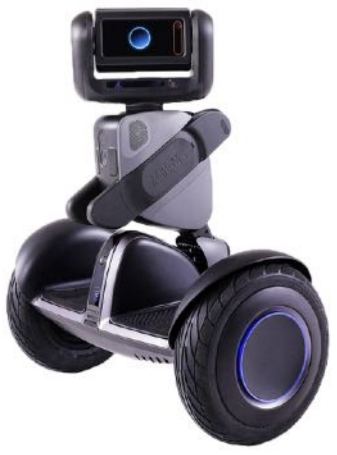


Figure 2.5 – The self balancing transporter and personal robot Loomo[5]

Loomo's hardware consists of a Segway/Ninebot Self-Balancing Vehicle combined with an Intel Atom-based computing unit and various sensors and actuators to sense and interact with the environment. It is equipped with an Intel RealSense ZR300 depth-sensing camera system enabling 3D perception, human detection and tracking as well as obstacle avoidance and mapping techniques such as SLAM. An HD camera is mounted next to the robots face providing 104 degrees field of view and a microphone array helps to localise voice sources and reduce background noises during voice recognition. An ultrasonic sensor enables object detection and collision prevention. Further sensors include touch sensors on the head for natural interactions, hall sensors in the wheel motors providing stable odometry data, and IMUs (inertial measurement units) in the robot's body and head measuring the robots pose and head orientation. A hardware extension bay gives the user direct access to an USB port and an additional power supply that can be

[1]https://techcrunch.com/2016/01/07/segway-has-created-a-robot-that-connects-to-your-two-wheeled-scooter/
[2]https://techcrunch.com/2017/01/05/segways-first-robot-launches-to-developer-partners/
[3]https://www.intelrealsense.com/loomo/

used as a power source for attached external devices.[4] Large anti-slip tires prevent sliding and make conquering uneven territory possible. Loomo can move at up to 18km/h in self-balancing mode or at 8km/h in following mode[5]. At £2000 (~€2300)[5] it is quite affordable compared to other assistance robot technologies.

Android is used as an open software platform, an SDK is available for building extensions and controlling the sensors and actuators. Functions for specific use cases can be implemented in the form of Android apps with access to the control SDK. Android being a widespread and popular operating system for mobile devices facilitates extending Loomo's function range.

2.5 Android Basic Principles

Android is a free operating system for mobile devices by the Open Handset Alliance founded by Google[6]. It is a software ecosystem (see Section 1.2.4) with a common technological platform maintained by its supplier. The non-technical end users create their own individual instances by installing components (apps) offered by third parties. The service oriented architecture of Android encourages variability by making the extension of the instance easy. It is open by design as well as dynamic, allowing the installation and removal of apps from the free market at runtime. While the variability management of the main platform is centralised and fully controlled by the supplier, there is no comprehensive centralised and integrated variability model, as variability information is described decentralised within each app. [Ber+14]

The execution of process chains as described in Section 2.3 is part of a software variability management. Building a prototype that implements the concept in Android requires an understanding of the basic features of an Android application.

An *activity* is a component of an app that provides a user interface (UI), i.e. a screen. An app contains at least one activity, the main activity with the home screen, which is usually started when the app is called, e.g. when the user taps the app's icon. With mobile apps however the user doesn't necessarily enter the application through the home screen. The home screen of an email app for instance might show a list of recently received emails, however if the user clicks a mailto link in their browser they will be redirected to the email app's screen for writing an email.[7]

Each activity of an app can be called by another activity of the same app or by another app. This is done through *Intents*, which, as the name says, express an app's intention to do something. They provide a runtime binding between separate components, such as activities, services (which perform background operations) or broadcasts (messages apps can receive). There are two types of Intents: *Explicit Intents* specify which concrete application satisfies the Intent, supplying the component's detailed package or class name. This type of Intent is mostly used to navigate between the components of an app, as the class names of the activities and services are known, which might not be the case with external apps. *Implicit Intents* on the other hand don't specify a component, but define a general action to be performed, allowing a component from another app to handle it. For instance, when wanting to display a location on a map, the developers don't have to implement this feature themselves, but they can use an implicit Intent to request that a

[4]https://developer.segwayrobotics.com/developer/documents/segway-robot-overview.html
[5]https://shop.segway.com/uk-en/57/-segway-loomo
[6]https://developer.android.com/legal.html
[7]https://developer.android.com/guide/components/activities/intro-activities

capable app fulfils this task.[8]

To give the component called by the Intent something to operate on, data can be included in the Intent, for instance in the form of *extras*. An extra is a *Bundle*, which is a key/value store for specialized objects, of any additional information. When for instance wanting so send an email message via an Intent, the extra can be used to include details of the message, such as a subject line or the email body.[9]

The user interface associated with each activity consists of a hierarchical structure of *layouts* and *widgets*. Layouts are containers controlling their children's position on the screen and are called *ViewGroup* objects. Widgets refer to UI components such as buttons or text boxes and are called *Views*. The menus, styles, colours and layouts of an UI are defined in XML files.[10]

2.6 Summary

In the running example a senior living group is assisted with daily tasks by a robot. As there are multiple users with diverse preferences and impairments, the robot has to adapt to the different needs of each person. Example tasks the robot might perform for the users include opening the door, accepting and delivering a package, and fetching an item for the user. It can be observed that each complex task can be broken down to a sequence of subtasks.

Afterwards the in Chapter 1 presented variability models and variability realisation mechanisms were analysed regarding the probability of them being used for the construction of variable process chains for robots. In terms of variability models, feature models can be assumed to be most likely used, as they are the most common and the best suited. OVMs only model variability and not commonalities, which makes it harder to capture the whole function range of a robot. They may therefore be less likely to be used. Decision models however focus solely on the process of configuration instead of the features of the product and are therefore not well suited to model the software architecture or the features of a robot respectively. Regarding variability realisation mechanisms, the need for an open and scalable variant space leaves only the compositional and transformational approach. Both are imaginable, but the compositional one might lend itself better to the concept of modular process chains.

Subsequently a concept for the execution of variable process chains was described. The concept involves a manager component that receives the user command, analyses it and instructs the task configurator component to create a task list from the given command and requirements. The task list is stored in a task object. After receiving the task object, the manager starts the first module on the list and hands it the task object. After fulfilling their task, each module calls the next module on the list and hands over the task object, until the last module calls the manager again.

The assistance robot Loomo could be employed in the scenario from the running example as it offers most of the features of the robot described. Wheels allow it to move around and multiple sensors and cameras help analyse the environment. Furthermore it is equipped with a touch screen and speakers providing output modalities. Solely the LEDs are missing from the robot described in the running example. Loomo is also based on the popular Android operating system. As it is widely used for mobile devices, there is a high chance that potential developers already

[8] `https://developer.android.com/guide/components/intents-filters`
[9] `https://developer.android.com/reference/android/content/Intent`
[10] `https://developer.android.com/training/basics/firstapp/building-ui`

have experience with the platform. Moreover developing for Android doesn't prove to be too difficult as java can be used as a programming language, which even more developers might be familiar with and which is quite similar to other object-oriented programming languages. The operating system makes the development of extensions quite accessible which together with the hardware features provides many possibilities to extend the feature range of the robot. Hence it is suitable to act as the platform for a prototype showing how process chains created with means of software variability can be executed in the context of an assistance robot.

Lastly, some foundation knowledge needed to implement this concept in Android was given. Android is a software ecosystem and therefore encourages variability. An Android application consists of at least one activity associated with a screen. The UI of the screen consists of View-Groups (layouts) and Views (widgets). App components can call each other through Intents, of which explicit and implicit ones exist.

3 Demonstrating the Concept of Variable App-Level Process Chains in Android

In the previous chapter, a concept for executing process chains was presented in a platform-independent and technology-neutral manner. In this chapter, the concept is going to be adapted for the Android platform and a prototype demonstrating the concept will be implemented for the assistance robot Loomo platform.

3.1 Adapting the Concept for the Android Platform

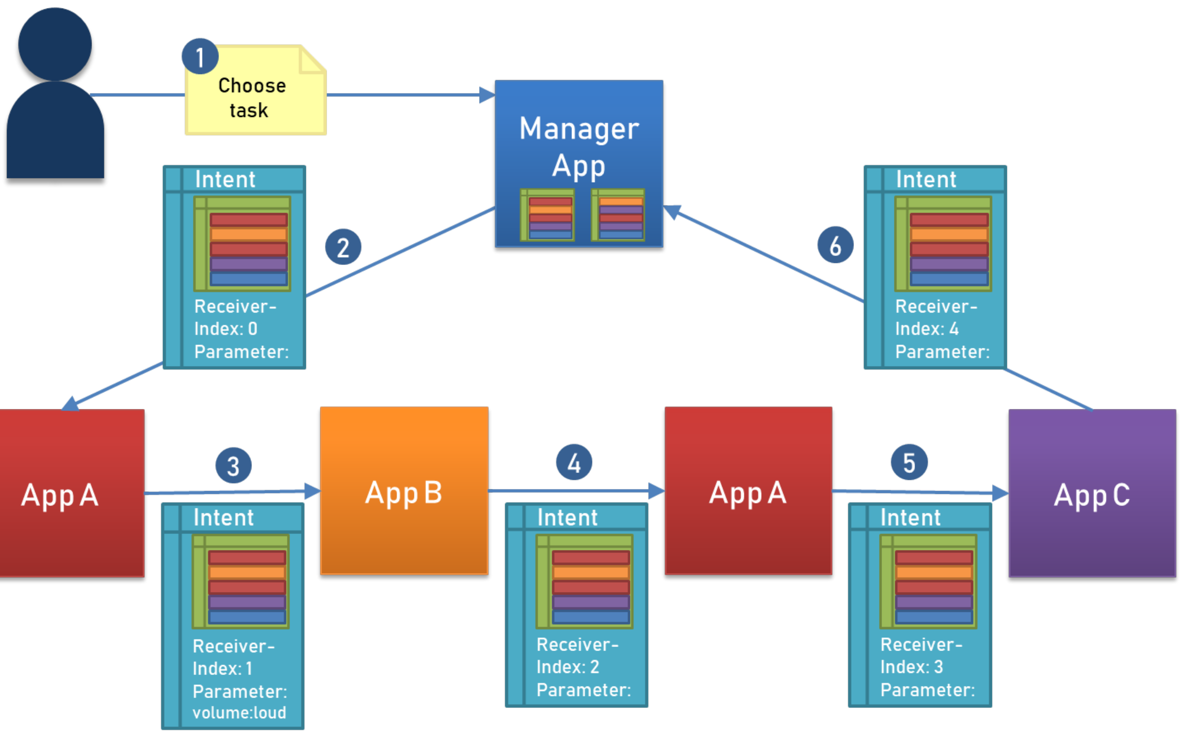


Figure 3.1 – An adapted concept for executing process chains in Android

In Section 2.3 a concept for executing modular process chains has been developed (c.f. Figure 2.4). The necessary knowledge to adapt this concept to the Android platform was given in Section 2.5. In this section, a concept adapted to the Android platform will be presented (see Figure 3.1).

The *modules* described in the original concept correspond to the top-level components in Android: *apps*. Each module representing a subtask can be implemented as an app which must support the task object as a form of data interchange format. The manager can be a standalone app or part of another app. It takes the user command, obtains the task object and starts the first app on the task list. While the robot is waiting for a command, the manager app is running and listening for input, but if the robot is currently executing a task, ideally it should still be able to receive another command or abort instructions. This introduces the need for handling concurrency (see Section 3.5).

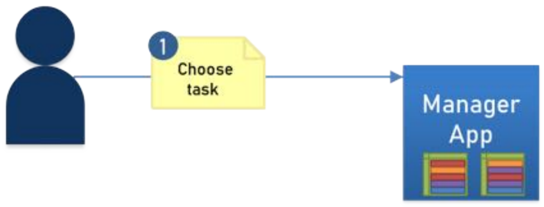


Figure 3.2 – The manager app receives a user command and obtains a suitable task object.

When the robot receives a command in a real fully-functional scenario, the manager analyses it regarding which task should be executed and which requirements the user imposes. A suitable task object is then constructed by the manager or a separate configurator app, possibly using means of software variability. As the implementation of the task configurator is not part of this thesis, the manager in the prototype and in Figure 3.2 is given two ready configured task objects to choose from. The prototype manager UI contains two buttons, each representing one task object, and pressing them corresponds to choosing a task object.

To start the execution of the encoded process chain, an Intent object (see Figure 3.3) must be created to start the first app on the task list. The Intent takes the task object, the receiver index (the list position of the app receiving the Intent on the task list) and optionally parameters as extras. After an app is started through the Intent, it must extract the extras from the Intent and store the receiver index (as the current index) and the task object in variables (see Section 3.4). If parameters are provided, the app will process them right away to execute the subtask it is specified for.

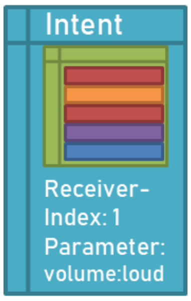


Figure 3.3 – The Intent containing the task object, the receiver index and a parameter "volume".

When the app fulfilled its subtask, it must access the stored index and task object to look up the next app on the task list via the incremented current index as list position. It will then, from the list entry, extract the package name of the next app's main activity and parameters needed for its execution. Subsequently a launch Intent for the package is constructed, putting the task object as an extra, the incremented current index as the new receiver index extra and if present the parameter as the parameter extra. Finally the task object and the current index variables are reset and the next app is started via the constructed Intent (see Figure 3.4).

This process is repeated for every app on the task list. To restart the manager app when the task is finished, so that another task may be executed, the last entry on the task list will contain the manager's package name. That way, the apps fulfilling the subtasks don't need to know the manager, which makes the changes to the manager easy and even enables the employment of multiple manager apps.

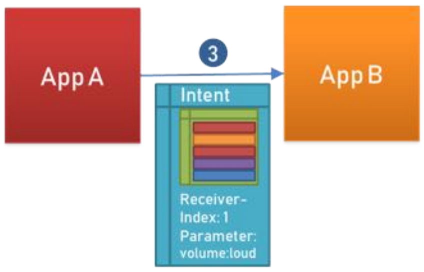


Figure 3.4 – Upon finishing, app A starts app B using an Intent containing the task object.

The concept described in this thesis has the subtask apps start each other, which requires each app to have knowledge about the whole process chain and subsequently some code to handle the task object (see Section 3.4). Though the code is reusable, it is hard to maintain and subtask app developers must be trusted to implement it correctly. Hence one should consider offering the process chain functionality in form of a library to developers, as this ensures the correct use of the interface and facilitates updates of the code.

Other than the decentralised approach used in this thesis, a centralised implementation is also imaginable, where the manager starts all subtask apps. Upon finishing, a subtask app restarts the manager, which then starts the next app on the list. That way, the manager is the only one to have access to the task object, which would not only protect it from being manipulated, but would also require less process chain specific functionality in the subtask apps. They could be addressed in the same way as when being called outside a process chain, though they would still be required to restart the manager upon finishing which requires some specific code. However, as in Android an active application is always visible in the UI, the user would see the manager app always being restarted in between steps, which doesn't look very seamless and could lead to confusion. Hence the decision for the decentralised approach was made, as it also closely resembles the original abstract concept.

3.2 A Process Chain for the Prototype

The scenario implemented in the prototype is a modified variant of the running example ”Fetching an item” (Figure 2.3). As it is implemented on the robotic platform Loomo, which has no built-in arms allowing it to actually pick up an item, the scenario is modified to not have the robot fetch an item, but rather to have it turn its head to observe the environment and report back the retrieved information to the user. As the robot's cameras are positioned at the head, this is a plausible use case. However, for the sake of this prototype the ”looking around” part of the process chain is merely symbolic and while turning the head no actual information is recorded. It must be noted, that the robot can be enabled to fulfil the task from the running example, for instance through the extension with a robotic arm via the hardware extension bay or through the cooperation with other devices able to fulfil this subtask, such as a drone. Nevertheless, the modified task suffices to show the feasibility of the concept.

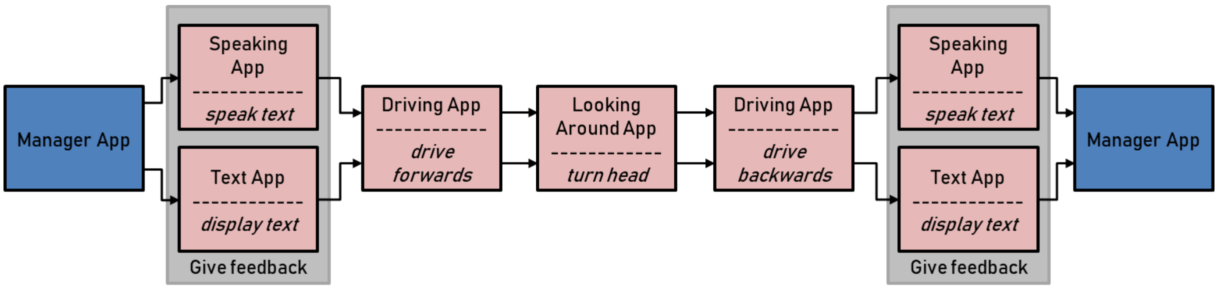


Figure 3.5 – The process chain implemented in the prototype

The scenario implemented in the prototype is that the user asks the robot to find their glasses. The robot will then confirm the command using the user's preferred output modality, drives a small distance, turns its head symbolising the scanning of the environment and drives back to its starting position. Finally, the robot tells the user, using the same output modality as before, that it unfortunately could not find the glasses. Figure 3.5 visualises the modules of the process chain. The robot receives the command in the form of having the user manually choose a deposited task object in the manager app, through pressing a button in the UI. Depending on which one was chosen, a different path is taken during the execution of the process chain. In this use case the two process chains only differ at the variation points where feedback from the robot to the user is required.

3.3 The Task Object in Android

The task object plays a crucial role in the execution of process chains, as it holds the task list with the information about the subtasks. JSON was chosen as a data interchange format to encode the task list. Other file formats, such as XML, could also be used, as they can encode the same structured information and are as common as JSON. As the task object is going to be passed around using Intents, the maximum Intent size of ~500kB[1] must be kept in mind. JSON is more lightweight, produces less overhead and is easier human-readable, which is why it was chosen to encode the task object in this case.

JSON is built on two structures: One is an unordered collection of key-value pairs, while the other contains an ordered list of values. JSON is language independent, therefore those two structures are implemented differently in the specific languages. In Android they are realised as JSON*Object*[2] (containing key-value pairs) and JSON*Array*[3] (ordered list of values) respectively.

The outer structure of the task object is a JSONArray, containing the subtasks as JSONObjects. As the order of the subtasks matters, they can be addressed sequentially by their position in the array. As described in Section 3.1, the index of the next app is stored as an Intent extra and not in the task object itself. This makes accessing this information easier. Each JSONObject representing a subtask contains the package of the app and optionally parameters as key-value-pairs. The value

[1]https://www.neotechsoftware.com/blog/android-intent-size-limit
[2]https://developer.android.com/reference/org/json/JSONObject
[3]https://developer.android.com/reference/org/json/JSONArray

of the parameter pair is another JSONObject. That way multiple parameters can easily be encoded, while keeping the same structure on the first level of each subtask object. When analysing the task object to address the next app, an object is addressed by its index in the array and the two values are extracted. The package name is used to construct the Intent, while the parameter JSONObject is put as an Intent extra. The app receiving the Intent knows if it expects any parameters and how many, and can now further unravel the JSONObject parameters. This enables the reuse of the code snippet for constructing an Intent for the next app on the list (see Section 3.4).

```
[
  {
    "package": "com.example.speech",
    "parameter": {
      "text": "Hello World!",
      "volume": 50
  }
  },
  {
    "package": "com.example.navigation",
    "parameter": {
      "destination": "kitchen"
    }
  },
  {
    "package": "com.example.manager"
  }
]
```

Listing 3.1 – A sample task object in JSON format.

There is also the possibility to not put the parameters as an Intent extra, but to have the receiving app extract its parameters itself. This is not done because, firstly, an app might be called by other Intents not containing a task object, which would require a different method of extracting the parameters. Having the parameters as an Intent extra can be declared as the only valid way of addressing the app, making the implementation simpler. Secondly, as part of a process chain, each app must access the task object after fulfilling its task to extract the information on how to address the next app on the list. Having the app access the task object only once further simplifies the code.

3.4 Documentation of Core Components

This section includes the most important components needed to implement the concept described in Section 3.1.

The Intent The Intent that starts the next app on the task list contains the following extras:

- `taskObject`: The task object as a stringified JSONArray
- `receiverIndex`: The list position of the receiving app as an Integer
- `parameter` (optional): Parameters for the receiving app as a stringified JSONObject

App Components The subtask apps require the following components:

- Instance variables to cache the received task object, current index and potential parameters needed for the task execution:

```java
private String taskObject = null;
private Integer currentIndex = null;

// optional
private String text = null;
```

- An `evaluateIntent()` method, which is executed in the `onCreate()` or `onStart()` method: Gets the Intent, extracts the data and stores the index, the task object and potential parameter

- A `taskFinished()` method, which is called when the subtask is fulfilled:
  Extracts the next task on the task list and builds the Intent with the necessary data from the task object, resets `taskObject` and `currentIndex` variables and finally calls the next app via the constructed Intent

The code snippets for the `evaluateIntent()` and `taskFinished()` methods can be found below. As every subtask app uses these code snippets, only slightly adapted for the different parameters that each app might accept, it is quite easy to adapt a new app to be part of a process chain. Furthermore, being able to address all apps that require the same parameters with the same task list entry (only adapting the package name in the task list), makes them easily exchangeable. This mutual interface is useful when wanting to realise a variability management to construct a task object. When implementing this concept in real life, one should consider creating a library holding these code snippets. This ensures the correct use of the interface and enables easier maintainability, updates and version control.

The manager app doesn't require the `evaluateIntent()` method, as it is at the end of the process chain and requires no further information from it. As is starts the execution of the process chain it however requires an adapted version of the `taskFinished()` method (for more details see below).

Receiving the Intent This code snippet shows the `evaluateIntent()` method, which should be executed right when the app is started, in the `onCreate()` or `onStart()` method. Firstly the Intent which called the app is fetched and - if present - the extras are extracted and stored in the instance variables. Moreover, it is shown how a parameter `text` can be extracted from the JSONObject. The code snippet can easily be adapted to accommodate multiple or no parameters at all. Depending on how the subtask application is implemented, the parameters might be handled differently. Instead of storing them in instance variables too, they might be returned as the JSONObject and extracted in another method. When considering offering this method in a library, the way of returning the parameters should be standardised, so that the method accepts all multiplicities of parameters while still being reusable.

```java
private void evaluateIntent() {
    if (taskObject == null) {
        // get Intent and passed Parameters
        Intent intent = getIntent();
        Bundle extras = intent.getExtras();
        if (extras != null) {
            // store taskObject and currentIndex
            currentIndex = intent.getIntExtra("receiverIndex", 0);
            taskObject = intent.getStringExtra("taskObject");
            // get expected parameter "text"
            String parameterString = intent.getStringExtra("parameter");
            if (parameterString != null) {
                try {
                    // read parameters
                    JSONObject parameter = new JSONObject(parameterString);
                    text = parameter.getString("text");
                } catch (JSONException e) {
                    e.printStackTrace();
                }
            }
        }
        Log.i(TAG, "Evaluate intent: currentIndex "+ currentIndex +"; text = \"" + text + "\"")
            ↪ ;
    }
}
```

Listing 3.2 – The evaluateIntent() method

After this method has been executed the attribute `taskObject` can be checked for being null and if it is not, the app is called in a process chain and needs to call the `taskFinished()` methods after fulfilling its purpose.

Starting the next App The function taskFinished() is called when the app completed its subtask and wants to start the next app on the task list.

```java
public void taskFinished(){
    // if part of process chain
    if (taskObject != null && currentIndex != null) {
        try {
            // get details of next app
            JSONArray taskList = new JSONArray(taskObject);
            JSONObject nextTask = taskList.getJSONObject(currentIndex + 1);
            String packageName = nextTask.getString("package");
            // create Intent
            Intent intent;
            if (packageName.equals(getPackageName())){
                // app has to call itself
                intent= new Intent(MainActivity.this,MainActivity.class);
            }else {
                PackageManager pm = getPackageManager();
                intent = pm.getLaunchIntentForPackage(packageName);
            }

            if (intent != null) {
```

```
20                 // put intent extras
21                 intent.putExtra("taskObject", taskObject);
22                 intent.putExtra("receiverIndex", currentIndex + 1);
23                 try{
24                     intent.putExtra("parameter", nextTask.getString("parameter"));
25                 } catch(JSONException e) {
26                     // no parameters for next app available
27                 }
28                 Log.i(TAG, "Start next app \"" + packageName + "\"");
29
30                 try {
31                     startActivity(intent);
32                 }catch(Exception e){
33                     Log.i(TAG, "Could not start Activity. " + e);
34                     e.printStackTrace();
35                 }
36             } else {
37                 Log.i(TAG, "Couldn't get an intent for the next app, check if the package name
                     ↪ exists.");
38             }
39             // reset
40             taskObject = null;
41             currentIndex = null;
42
43             finish();
44         } catch (JSONException e) {
45             e.printStackTrace();
46             Log.i(TAG, "Log.i(TAG, "Something went wrong when trying to convert the taskObject
                 ↪ to a JSONArray: " + e);
47         }
48     }
49     Log.i(TAG, "Task object is null.");
50 }
```

Listing 3.3 – The taskFinished() method

First of all one needs to make sure that the app is currently part of a process chain before extracting information in the next app (Line 3). Then in a try-catch block the stringified JSONArray `taskObject` is converted to a real JSONArray (Line 6) and the next task is fetched (Line 7). As the Intent is created differently when calling another app compared to when an app calls itself, this condition must be checked (Line 12). If the Intent was successfully created, the `taskObject` and `receiverIndex` are put as extras. As parameters are optional and therefore might be non-existent, putting them as extras requires a try-catch block to prevent errors. Finally the next application on the task list is called via the Intent.

Regardless of whether the Intent is successfully created and the next app is started or not, the last step is to reset the task object and current index and terminate the app via `finish()`. In case the Intent can't be created, which might happen due to a non-existent package name in the task object, the application is finished nonetheless. As all subtask apps are terminated upon finishing but the manager application is not, it is on top of the stack of paused applications and is resumed automatically by the Android system if a subtask app is terminated without starting another app. One could also consider setting the manager application as the home screen using a launcher, that way one would return to the manager under all circumstances.

The reason why it is important to terminate each subtask app is that when an application has previously been part of a process chain, because it appears multiple times in the same task or it was part of a task that has been executed earlier, upon being resumed it remembers the Intent that created it and doesn't recognise the new Intent which has different extras. Furthermore terminating each app upon finishing helps with binding the robot's services, as the services are bound each time on creation and automatically unbound on termination of the app, making sure the services are bound correctly when (re)starting an app.

The manager app calls the first app on the list and therefore also requires a code snippet such as this one. However as it knows it is starting the first app on the list it may set the `receiverIndex` to `-1` and only check if the task object in not null, as the current index can't be. Moreover, the manager app is not finished when starting the process chain, but can be resumed if another app fails to continue the process chain.

3.5 Handling Concurrency

Ideally the robot's user interface should conform to the dialogue principles [ISO06]. This implies that the robot's behaviour should *conform to user expectations* and remain *controllable* while the robot is currently executing the task. When for instance the robot is currently on its way to open the door, the user might want to know the time and expects the robot to answer right away. The user might even give abort instructions. This introduces the need for the parallel execution of tasks.

When introducing parallel task execution, it must be kept in mind that not all combinations of tasks can be executed concurrently, for instance if they use the same assets of the robot. Giving the robot conflicting commands, such as driving in two different directions at the same time, could lead to the application(s) crashing or in the worst case the robot exhibiting unpredictable and potentially dangerous behaviour. Therefore rules on which combinations of tasks are allowed to be executed simultaneously and which tasks can only be executed sequentially must be established. In case of an abort command, these considerations are naturally not necessary.

In the concept that has been developed in this thesis, tasks are split into subtasks, which can each be executed by an Android app. So in order to execute tasks concurrently, apps need to run concurrently. In Android 7.0 and higher, devices can display multiple apps simultaneously using multi-window[4], for instance in split screen mode or picture-in-picture mode. However, in Android 9 and lower, apps could not actually be run concurrently. Only one of the displayed apps could be active at any time, the other app(s) being in the paused state. Though developers could adapt their app to handle the paused state, not everyone did, which could lead to a poor user experience, such as messages not being sent and videos stopping, when apps entered the paused state. In 2019 Google introduced the "multi-resume" feature for Android 10, allowing multiple apps to maintain the resumed state at the same time[5]. When wanting to implement a parallel listener component in the form of an app (e.g. the manager app), those features are essential. As the Loomo platform is running on Android 5.1 (API Level 22)[6], unfortunately neither of them is currently available for the robot. If in the future Loomo's Android version gets updated, tasks

[4]`https://source.android.com/devices/tech/display/multi-window`

[5]`https://www.xda-developers.com/android-q-splitscreen-multitasking-multi-resume/`
`https://source.android.com/devices/tech/display/multi_display/multi-resume`

[6]`https://developer.segwayrobotics.com/developer/documents/setup-developing-environment.html`

might be executed in parallel. Until then, a new task can be either put in a queue to be executed after the first task is finished, or, if the new task has a higher priority than the one currently running, the first task can be put on hold and is resumed after the new one is finished. The currently running task might also be stopped entirely in favour of another task or due to an abort command.

To detect a new command, a parallel listener component should be installed. It should always run while a task is executed and be able to accept commands, for instance via a voice UI. It must analyse new commands regarding the next steps (i.e. should the task be queued or executed immediately). In case the new task should be executed immediately, a new task object must be constructed, which the listener component may request from the task configurator component. In short one could say it is a parallel running, listening manager component. If the parallel execution of apps was possible, one could consider using the existing manager app as the parallel component. As this is currently not the case, another way of implementing a concurrent component to the subtask apps must be found.

Android services are application components performing long-running operations in the background without a user interface[7]. Such a service could be implemented as the parallel listener component. When starting the execution of a process chain, the manager would start the service to listen in its stead. Upon recognising a new command the service would, as described previously, either store the command in a queue which is returned to the manager app after the task execution, or the currently running task is interrupted and the service either resumes the old task itself (if it should be resumed) or passes this information to the manager app.

The issue that arises with the use of services lies within the limitations of the Loomo SDK: Though a service can bind another service[8], making it possible for a background service to theoretically bind the speech SDK of the robot, the Loomo platform only supports binding to services that run in the foreground. If the application is switched to the background, the service is disconnected automatically[9]. This means, that when for instance the base SDK is currently moving the robot, the speech SDK can't be bound in another component running in the background. If the robot SDKs are required to run parallelly in separate components, it has to be tested if there is a way to use foreground services to bind the robot services. Finding an alternative to the robot voice SDK that can be run in the background might also be an option.

In case this does not provide the desired solution, the final suggestion is to include the listener component in each subtask app. A queue may be passed with the Intent along the process chain, so that after the task is finished, the manager app knows which process chain to execute next. If the current task should be interrupted for the new task, the information if the old task needs to be resumed after the new one is finished must be kept. It can be passed in the Intents of the new process chain.

To sum it up, to ensure controllability the implementation of a concurrent listener asset is recommended. This asset can be a separate app or service, or be integrated in each subtask app. While the parallel execution of process chains is not currently possible, as in Android 5.1 only one app can be displayed and active at any given moment, one can either queue incoming process chains or interrupt a running process chain for another one with a higher priority. To accept

[7]https://developer.android.com/guide/components/services

[8]https://developer.android.com/reference/android/content/Context.html#bindService(android.content.Intent,%20android.content.ServiceConnection,%20int)

[9]https://developer.segwayrobotics.com/developer/documents/segway-robots-sdk.html

incoming requests, the asset is required to listen for new commands and decide what happens with the recognised task. Not strictly sticking to the concept of process chains may allow simple requests, such as telling the time, to be handled immediately by the listener component itself without the need to start subtask apps.

Having the concurrent listener asset in a separate component would be preferred for maintainability and reduced overhead in the subtask apps. However, even though reusing the existing manager app would require minimal extra effort, a concurrent listener app was ruled out for technical reasons. Implementing a listener service called by the manager app before starting the execution of the process chain could work, but may cause problems if accessing the robot SDKs in a background component is necessary. Finally, an integrated listener can most likely be implemented successfully, using the robot SDKs to listen for new commands. As code must be deployed in every subtask app this feature would however be very hard to maintain. Providing the listener functionality in a library offers a solution to this problem, as it not only facilitates updates to the code, but it also ensures the correct use of the listener component.

3.6 Summary

In this chapter the platform-independent concept from the previous chapter was adapted for the Android platform as the assistance robot Loomo, which serves as a platform for the prototype demonstrating this concept, is based on Android. The adapted concept includes a manager app listening for user commands and subtask apps each fulfilling an atomic subtask. On a user command the manager app gets a task object, which may be constructed with means of software variability. After retrieving the first app on the task list, the manager creates an Intent which starts this app and contains the task object, the started app's list position and optional parameters as extras. Upon receiving the Intent, a subtask app analyses the Intent and stores the Intent extras in instance variables. When an app has finished its task it analyses the stored task object for the next app on the list using the incremented current index as the list position. An Intent is created and the task object, the receiver's index and optional parameters are put as Intent extras. Finally the Intent is called which starts the next app on the task list.

Moreover a use case is derived from the running example to be demonstrated in the prototype: The user chooses their preferred output modality by pressing a button which triggers the execution of a process chain containing the chosen output modality. The robot may give feedback via speech or displaying some text on its display. The scenario is that the user asks the robot if it can find the user's glasses. The robot answers using the chosen output modality, drives a bit in one direction, looks around turning its head, drives back and tells the user that it couldn't find the glasses. For each of these steps a separate app is used. After the task is finished the manager app is restarted.

The task object will be implemented as a JSONArray containing a JSONObject entry for each app that is in the task list. Using the JSON format makes the task object easily human-readable and reduces the size of the Intent. Furthermore the core components that need to be present in each Android app to act as part of a process chain were given. They include instance variables to store the Intent extras `taskObject` and `currentIndex` as well as any parameter. Furthermore each subtask app requires a code snippet that evaluates the Intent and one that creates the Intent for the next app on the task list.

Finally, some considerations were made regarding concurrency in Android. Having a listener component running concurrently while the task is executed can solve the problem of the robot not reacting to new commands while currently executing a task. The listener component might be an app, which is not currently possible for Loomo's Android version, a service or be integrated in each subtask app. Implementing it as a service may be complicated by the fact that the robot SDK components can only be accessed by Android components running in the foreground. Lastly integrating the listener into each subtask app is the option that seems most likely to be successful. This makes the code harder to maintain as each app must be accessed when something in the source code of the listener component changes, which is why outsourcing the code to a central library should be considered.

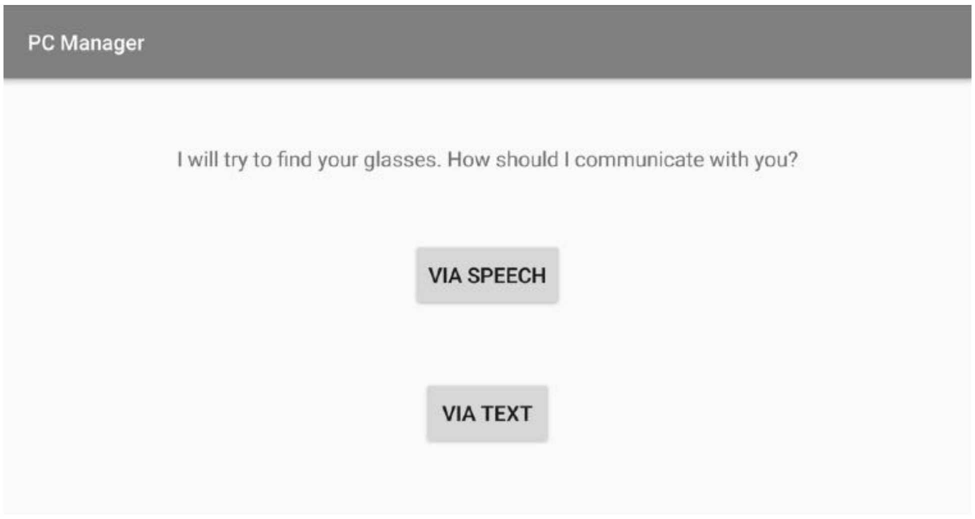
4 Evaluating the Protoype

In this chapter, the functionality of the prototype is going to be evaluated regarding whether it sufficiently showcases the feasibility of the previously developed concept for the execution of process chains in Android. To do this, tests need to be performed proving that providing a task object to the manager app results in the correct execution of the tasks on the task list.

4.1 Components to Test

The prototype consists of five individual Android apps, each consisting of a single activity.

- **Manager App** (see Figure 4.1): Holds two task objects, when a button is pressed it starts the execution of the first app from the associated task object.
- **Speaking App**: Is given a String `text`, which is output via Loomo's text-to-speech module.
- **Text App**: Is given a String `text`, which is displayed on the screen.
- **Driving App**: Is given a Float `direction`, which is used to set the x-coordinate of a checkpoint for the robot's base module to navigate to.
- **Looking Around App**: When started turns the robot's head around the z-axis (Yaw axis)


PC Manager

I will try to find your glasses. How should I communicate with you?

Figure 4.1 – The start screen of the manager app

It has to be noted, that for the apps that use the Loomo SDK, namely the Speaking App, Driving App and Looking Around App, a delay has been built in to give the app time to bind to the services.

The services should be initiated right when starting the app. Between connecting to the services and using them, around 5 ms must have passed. The exact amount of time may vary, therefore tests are indispensable. When waiting for user input to use a service, for instance having the user press a button on the screen to start voice recognition, this issue should not arise. However as the goal was for process chains to be executed automatically, user input should not be required during the execution of a process chain.

In order to implement a delay, the Android Handler `postDelayed()` method[1] and Android CountdownTimer[2] were used successfully. Using `Thread.sleep()` was not successful, as the binding of the service did not progress in the sleeping thread. Moreover binding the service in the `onCreate()` method and accessing it in the `onStart()` method failed as well. In some circumstances if enough code is processed in between binding and accessing, the service might be bound by the time it is accessed. Nonetheless it is inadvisable to have the implementation depend on that possibility.

4.2 The Testing Process

The testing process in this thesis aims at proving the functionality of the prototype as well as its robustness against edge cases. The test cases are derived from the system specification and requirements, rather than from the code, making this a case of black-box testing. Hence code coverage is not considered.

In order to test the correct execution of process chains, test cases must be defined. For each test case a task object, which is used to start the process chain in the manager, is going to be provided as input. For each test case the expected outcome is described. If the anticipated result occurs, the test is considered successful. The test cases were chosen to show the applicability of the concept for the use case described in Section 3.2, as well as to demonstrate how the system handles edge cases, such as a faulty or extremely long input (i.e. task object).

The execution of the tests takes place as follows: For each test case, a task object in JSON format (see below), will be hardcoded manually in the manager app. On the press of a button, the manager app fetches the task object and starts the execution of the associated process chain. Each app that can be part of the process chain, the manager as well as the apps each representing a subtask, output log statements at the crucial parts of their lifecycle:

- When the app is being started
- After the Intent was evaluated; print values of `currentIndex` and parameters
- When the services are bound successfully
- When the main action of the subtask app is currently being executed (in most cases when the robot SDK is used)
- Before starting the next app

Furthermore log statements are put wherever something can go wrong.

[1] `https://developer.android.com/reference/android/os/Handler.html#postDelayed(java.lang.Runnable,%20java.lang.Object,%20long)`
[2] `https://developer.android.com/reference/android/os/CountDownTimer`

4.2.1 Test Cases

Prototype Scenario First of all it needs to be tested if the scenario that has been described in Section 3.2 can be executed correctly. To do this, two task objects each representing one variant of the process chain visualised in Figure 3.5 are tested:

```
[
    {
        package: "com.example.processchaintext";
        parameter: {
            "text": "Okay, I will have a look for you."
        }
    },
    {
        package: "com.example.processchaindrive";
        parameter: {
            "direction": "0.5f"
        }
    },
    {
        package: "com.example.processchainlookaround"
    },
    {
        package: "com.example.processchaindrive";
        parameter: {
            "direction": "-0.5f"
        }
    },
    {
        package: "com.example.processchaintext";
        parameter: {
            "text": "Hey, unfortunately I couldn't find your glasses."
        }
    },
    {
        package: "com.example.processchainmanager"
    }
]
```

Listing 4.1 – *TC01-TextFeedback*: The task object for the text feedback variant.

```
[
    {
        package: "com.example.processchainspeech";
        parameter: {
            "text": "Okay, I will have a look for you."
        }
    },
    {
        package: "com.example.processchaindrive";
        parameter: {
            "direction": "0.5f"
        }
    },
    {
        package: "com.example.processchainlookaround"
16     },
17     {
18         package: "com.example.processchaindrive";
19         parameter: {
20             "direction": "-0.5f"
21         }
22     },
23     {
24         package: "com.example.processchainspeech";
25         parameter: {
26             "text": "Hey, unfortunately I couldn't find your glasses."
27         }
28     },
29     {
30         package: "com.example.processchainmanager"
31     }
32 ]
```

Listing 4.2 – *TC02-SpeechFeedback*: The task object for the speech feedback variant.

It is to be expected, that for both task objects the log shows that each of the specified apps is started, the Intents are evaluated and for the apps using the Loomo SDK the services are bound successfully. Furthermore the main action (e.g. speaking or displaying the text parameter) should be executed before the next app on the list is started successfully. Lastly the manager app should be restarted.

This test also proves the fact that it should not be a problem if the same app is in a task object multiple times with different parameters. Furthermore it shows that the interface can handle task list entries without any parameters, such as the entry for the manager (Line 30) and the Look Around App (Line 15).

App is in the task list twice in a row This test case shows that the interface is able to handle the situation that an app needs to call itself with different parameters. It is expected that the Text App is executed three times, each time with a different `text` parameter.

```
[
    {
        package: "com.example.processchaintext";
        parameter: {
            "text": "One"
        }
    },
    {
        package: "com.example.processchaintext";
        parameter: {
            "text": "Two"
        }
    },
    {
         package: "com.example.processchaintext";
        parameter: {
            "text": "Three"
        }
    },
    {
        package: "com.example.processchainmanager"
    }
]
```

Listing 4.3 – *TC03-AppSeries*: The task object including the Text App three times in a row with different parameters.

Wrong package name This test case shows what happens when there is a non-existent package name in the task object. This may be due to a spelling error or because the package name of the app was updated. It is to be expected that the app trying to create an Intent for the non-existent package name logs an error and finishes, afterwards the manager is automatically restarted.

```
[
    {
        package: "com.example.processchaintext";
        parameter: {
            "text": "One"
        }
    },
    {
        package: "com.example.processchaintext";
        parameter: {
            "text": "Two"
        }
    },
    {
        package: "com.example.processchaintest";
        parameter: {
            "text": "Three"
        }
    },
    {
        package: "com.example.processchainmanager"
    }
]
```

Listing 4.4 – TC04-*WrongPackageName*: A task object with a wrong package name.

Faulty task object If the task object in the manager is faulty, i.e. not a real JSONArray or null, the manager application should log an error.

```
{}
```

Listing 4.5 – TC05-*FaultyTaskObject*: A faulty task object.

Large task object An Intent can be at most around 500 kB in size[3]. If the process chain is too long, this should result in an error.

[3]https://www.neotechsoftware.com/blog/android-intent-size-limit

```
[
    {
        package: "com.example.processchaintext";
        parameter: {
        // The extremely long value for the text parameter was shortened here
            "text": "Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy
                ↪ eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam..."
        }
    },
    {
        package: "com.example.processchainmanager"
    }
]
```

Listing 4.6 – TC06-*LargeTaskObject*: A very large task object.

4.3 Evaluating the Test Execution

In this table, the test cases are going to be named and it will be marked whether the test was successful or not. Furthermore, the log is linked.

Test Case	Task Object	Log	Test successful?
TC01-TextFeedback	Listing 4.1	A.1	yes
TC02-SpeechFeedback	Listing 4.2	A.2	yes
TC03-AppSeries	Listing 4.3	A.3	yes
TC04-WrongPackageName	Listing 4.4	A.4	yes
TC05-FaultyTaskObject	Listing 4.5	A.5	yes
TC06-LargeTaskObject	Listing 4.6	A.6	yes

Table 4.1 – Overview of test results

As Table 4.1 shows, the tests were successful, which shows the feasibility of the concept as well as its robustness.

4.4 Observations and Boundaries Regarding the Implementation

There are some things to be considered when applying this concept to a real life scenario. These considerations include boundaries as well as things future developers should simply be aware of.

In the running example in Section 2.1 three use cases were presented. The ”Fetching an Item” example (Figure 2.3) is very similar to the use case on which the prototype is based. Variability appears in form of different output modalities that depend on user preferences. This kind of variability can be anticipated before constructing the task object. The opposite is true for the use case ”Opening the Door” (Figure 2.1). Here, parts of the modular task vary because the robot needs to adapt its behaviour to unexpected situations. In the ”Delivering a Package” example variability occurs due to expected user preferences (where they like their packages to be put) and unexpected situations (whether the recipient of the package can be located). In a real life scenario an assistance robot must always expect the unexpected and be able to react to unforeseen situations. The concept of variable process chains can be applied when the steps of a task can be predicted, as the task object is created before processing it. If unknown variability is to be expected, the process chain might be constructed as far as the subtasks are known, the last module on the list being an application that can handle the unknown situation and either edit the task object or create a new one.

Furthermore, in the concept as described, the robot can't receive another command while it is executing a task. Only when the task is fulfilled and the manager module/app is restarted, a new task can be commissioned. To avoid the robot "being in the zone", a way to accept commands while a task is running should be found. This requires implementing concurrency, while keeping in mind that the services of the Loomo SDK can only be bound in applications running in the foreground, and has been discussed in Section 3.5. The implementation of a parallel listener component was recommended to solve this issue.

As mentioned in Section 4.1, when using the Loomo SDK, the developer must schedule enough time for the application to bind to the services before using them. This can be achieved by employing a delay period after initiating the services.

The apps created for the prototype consist each of only one activity for the sake of simplicity. However, in practice an app fulfilling a subtask might be more complex and consist of multiple activities. In that case the developer must make sure the task object and current index don't get lost, either by passing them with the Intent or by making them accessible from other activities, for instance by declaring them as instance variables or creating getter and setter methods. When this functionality is standardised it can also be offered in a library.

Something that has been observed, but was not part of the test cases as it doesn't play a role for the intended execution of process chains as defined in the concept is the following: When the execution of a process chain is finished, and the user manually resumes an app that has been part of said process chain from the history (recent apps), the application, even though it has been terminated via `finished()`, remembers the Intent that created it the last time and proceeds to act like it was still in a process chain. This only happens when the app is manually resumed immediately after being in the process chain. If this behaviour should be forbidden, the developer is advised to look into marking the Intent as handled before finishing the app, to avoid having it handled again on resume[4].

4.5 Summary

In this chapter the prototype that has been developed in Chapter 3 is tested.

Firstly the components of the prototype are listed. The prototype consists of five Android apps: The *Manager App*, the *Speaking App*, the *Text App*, *the Driving App* and the *Looking Around App*. These apps can be used to implement the scenario described in Section 3.2.

Furthermore the testing process is described. For each test case a task object is given which is deposited in the manager app for testing purposes. To track the correct execution of the process chain, each app contains log statements at crucial steps of its life cycle. First of all the scenario described in Section 3.2 is tested, as the prototype has been designed to implement it. The two

[4]Possible solution: `https://stackoverflow.com/a/25535915`

task objects are the same except for the app that outputs the task parameter. One process chain uses the Text App while the other uses the Speech App. Further test cases include a task object where the same task is included multiple times in a row, a task object including a non-existent package name, a faulty task object and a very large task object.

All the tests were executed successfully. Finally some concluding observations were made regarding the implementation of the prototype, for instance that it is in the nature of the concept to construct the process chains before executing them, which makes reacting dynamically to the environment difficult. Additionally, to avoid the robot not reacting to new commands while currently executing a task, a way to implement concurrency in Android should be found, which was described in detail in Section 3.5.

5 Conclusion

This chapter will summarise the thesis, discuss the found solution, answer the research question and give an outlook.

5.1 Summary

There is a growing demand for adaptive assistance robots, as more people are living longer and the health care sector is understaffed. Robots can assist people in nursing homes as well as help individuals to stay autonomously at home longer and improve their quality of life. A definition for assistance robots has been provided and requirements for a successful assistance robot in the health care domain were gathered. They include adaptability and a broad function range followed by affordability, providing the user with independence, a sense of security and emergency support as well as having an intuitive user interface. Additionally the humanoid assistance robot Pepper and the mobile robot assistant CARE-O-BOT 4 were presented.

Furthermore, the notion of software variability was explained and three variability models (feature models, decision models and orthogonal variability models) as well as three variability realisation mechanisms (annotative, compositional and transformational) have been introduced. The concepts of software product lines and software ecosystems were described as well as the need and the roadblocks for the application of software variability techniques in the robotics domain.

The goal of the second chapter was to develop a concept for executing variable process chains. Firstly a running example was designed. It revolves around 4 elderly people living in a residential group mostly on their own with the help of an assistance robot. Each of the residents has a different impairment which requires the robot to adapt its way of communicating for each individual user. Examples of tasks that the robot should perform were given, each of which could be broken down into atomic reusable subtasks. Subsequently the previously presented variability models and variability realisation mechanisms were evaluated in terms of their suitability for constructing modular tasks. The favoured variability models were feature models and potentially OVMs, while the most suitable variability realisation mechanism is the compositional approach, the transformational approach potentially also being feasible.

The chapter's focus was to develop a high-level concept for the execution of variable process chains. The concept contains a manager component in its centre, which upon receiving a user command analyses it and instructs the task configurator component to create a task list from the given command and requirements and return it in form of a task object. After receiving the task object, the manager starts the first module on the list and hands it the task object. After fulfilling their subtask, each module calls the next module on the list and hands over the task object. The last module calls the manager again.

The assistance robot platform Loomo was introduced afterwards, as it serves as a platform for the prototype demonstrating the concept. The robot consists of a self-balancing vehicle combined with a computing unit and various sensors and is based on the Android operating system.

To give the reader some foundation knowledge needed later for the implementation of the prototype, the basic principles of Android were introduced.

In the third chapter the previously developed concept was elaborated and adapted for the Android platform to be implemented for the assistance robot Loomo. As the construction of the process chains is not implemented in this thesis, the manager app in the prototype holds a fixed task object. On a user command the manager starts the execution of the given process chain, which is encoded in a task object, by looking up the first subtask app on the task list and creating an Intent for this app, putting the stringified task object, stringified parameter JSONObject and the receiving app's list index as Intent extras. Each subtask app analyses its starting Intent's extras, stores the task object and the current index and uses the parameters to execute its task. After finishing, an Intent for the next app on the task list is created using the stored task object and the current index and the next application is started via this Intent.

Subsequently a concrete use case, which was derived from the running example, was described to be implemented in the prototype: The user chooses their preferred output modality by pressing a button in the manager app which triggers the execution of a process chain containing the selected output modality. The robot will give feedback either via speech or by displaying some text on its display.

Furthermore, the concrete implementation of the concept's components is described in detail. The task object is implemented in JSON format, as a JSONArray that contains a JSONObject entry for each app that is in the task list. This makes the task object easily human-readable and reduces the size of the Intent. The core components required in each Android app which may be part of a process chain include instance variables to store the Intent extras `taskObject`, `currentIndex` and any parameter. Moreover each subtask app requires a code snippet that evaluates the Intent and one that creates the Intent for the next app on the task list.

Finally the problem of the robot not being able to react to commands when currently executing a task was discussed. To solve the problem, the implementation of a concurrent listener component is recommended. Implementing a listener component in each subtask app is the most promising option, even though it is harder to maintain than a parallel service or app. This disadvantage may be compensated when offering the listener functionality in form of a library.

The fourth chapter documented the testing process for the prototype showcasing the concept developed in this thesis. After listing the components of the prototype, which consists of a manager app and four subtask apps, the testing process was described. Each app contains log statements at crucial points of its life cycle. For each defined test case the log is printed after conducting the test and it is put in the appendix. Each test case contains a short description and expected outcome, as well as a task object with which the test will be run. The outcome of the test was documented in Table 4.1. All tests yielded the expected outcome. Finally some observations regarding the boundaries of the prototype were made.

5.2 Discussion

It will be briefly discussed whether the proposed concept is a suitable solution for a use case such as the one described in the Running Example in Section 2.1.

In Section 1.1.1 requirements for a successful assistance robot have been described: adaptability, a broad function range, affordability, providing the main user with independence, support and a sense of security, as well as an intuitive user interface. The Loomo robotic platform can be used to develop a robot that fulfils these requirements. At ~€2300 it's realistic for individuals and health care facilities alike to be able to afford the assistance robot Loomo, though it is still an investment. A rental model could be put in place to make this technology accessible for a broader audience. The fact that the Loomo platform is based on Android, a software ecosystem, implies that the extension of the robot's function range is fairly easy. Equipping the robot with an intuitive user interface and the ability to provide the user with independence, support and a sense of security is the task of an UX designer. However, the concept and the prototype developed in this thesis were developed to enhance the robot's adaptability. Using a software variability management poses the possibility to cope with the need for adaptability by enabling the construction of tasks that are adapted to user requirements. The aim of this thesis was to propose a concept for the execution of variable app-level process chains, which have been created using software variability methods.

The concept proposes breaking a task down into reusable subtask modules. Combining subtasks to complex tasks leads to solving more problems with a smaller set of modules and further facilitates the extension of the function range, as new modules may be added or old ones may be swapped out. Additionally the system is easier to maintain. The prototype implements this concept, showing that given a list of subtask apps encoded as a task object, the robot is able to execute the apps autonomously in the specified order. The modules "Speaking"/"Displaying Text", "Driving" and "Looking Around" are used to symbolically fulfil the task of searching the user's glasses. Swapping out the "Speaking" module for the "Displaying Text" module shows that a process chain can easily be adapted to user requirements.

While the prototype successfully demonstrates the concept for executing variable tasks in Android, there are some considerations that have to be made regarding whether this is the right way to solve the problem of adaptability in assistance robots. There is, for instance, the need to execute a task while listening for new commands and, ideally, being able to fulfil the command or abort instruction immediately. This is crucial as users expect a robot to react to their commands at any given time. In this thesis ways to implement concurrency in Android were proposed to solve this problem (see Section 3.5). Having a listener component in each app that can be part of a job sequence seems the most promising. When offering a library containing the listener functionality maintainability can be sustained and the correct use of the listener is facilitated.

Another important issue is the requirement for an adaptive assistance robot to cope with unknown situations at run-time. This plays a role for all robotic applications coping with variability and remains an ongoing problem [Gar+19]. Process chains as described in this thesis are constructed fully before being executed. This does not leave room for unexpected situations, as the environment can only be taken into account when constructing a process chain. Hence a way to dynamically update or create process chains should be found. The need for a parallel listener component that makes reacting to or at least taking in new commands during task execution has already been established. A very basic approach for this component could entail only queuing the received commands to execute them after the current task is finished. Abort commands should of course be executed immediately. One could however extend the functionality of this component, enabling it to dynamically update the current process chain according to environmental conditions and unexpected situations.

Additionally the concept developed in this thesis was designed for the Android platform. Many robotic applications are however not based on Android but on the Robot Operating System ROS. The concept developed here can't easily be adapted for ROS and is therefore only applicable for a narrow set of robotic applications. However, for Android based robotic application the concept provides the opportunity to use modular process chains to implement the execution of variable tasks, if the issues mentioned are resolved.

5.3 Answer to the Research Question

The research question that has been posed in the beginning of this thesis was: *How can a job sequence that has been created with means of software variability be executed in Android?*

This question has been answered by proposing to encode the job sequence in a JSONArray object (the task object), which includes the single tasks encoded as JSONObjects. Each JSONObject representing a task contains the key-value pair package name as well as an optional key-value pair parameter. Moreover, explicit Intents are used to start the apps in the task object, containing the task object, the position of the application on the task list and optional parameters as extras.

To be part of a job sequence, applications require a common interface, which also makes apps easily exchangeable. This interface requires two code snippets, that could be made available to developers in a library. The first one evaluates the Intent that started the app and stores the Intent extras. It is executed right when the app is started. The second one is executed when the application finished its subtask. It extracts the next task in the job sequence from the task object, creates an Intent for the specified package and starts the next application.

A manager application obtains the task object in the beginning, for instance using means of software variability, and starts the first app on the task list. It is also the last app on the task list, to ensure it is started again when the task is finished. While all subtask apps are terminated upon finishing, the manager is not, so if during the execution of the job sequence an app can't be started, for instance due to a wrong package name, the app that tried to start is terminated and the manager is automatically resumed by Android.

5.4 Outlook

In the running example in Section 2.1 a scenario with four elderly people living in a residential group with the help of an assistance robot was described. The robot helps them keep their autonomy longer by assisting them in simple tasks in their daily lifes. The tasks provided include opening the door, accepting and delivering a package and fetching an item. The prototype that was developed in this thesis implemented the task "Search a missing item", which is another useful scenario for the running example. Almost every task an assistance robot should be able to perform can be broken down to subtasks. Exceptions might be simple question-answer scenarios, where a voice assistant module would be sufficient.

Before implementing the concept in a real life robotic application, the remaining problems mentioned earlier must be resolved. Most importantly this includes a parallel listener component for controllability and the ability to react to unexpected situations at run-time by dynamically creating or altering task objects.

As mentioned, one could try to use the parallel listener component to not only react to direct commands, but also to react to unexpected situations and environmental conditions by updating the task object. This means each subtask app would contain its own parallel manager component. To standardise this component and make it reusable and maintainable, it is advisable to offer it in form of a library to developers, alongside the functionality necessary for an app to be part of a process chain (see Section 3.4). Regarding the latter, the advantage of putting the functionality in a library also entails that subtask app developers don't need to directly handle the task object, which should ensure the correct use of the interface. Developers would still be required to actively include the process chain functionality in their app, in form of library function calls, but they are not required to fully understand it.

With the issues resolved, the concept developed in this thesis could be part of a system that uses software variability management methods to construct process chains in a task configurator component. Though the task configurator should be cared for by a single maintainer, the open Android ecosystem allows the easy addition of new components. New components should be registered with the maintainer of the system, who must include the components in the underlying model and define configuration rules. By offering details on the interface third party developers can contribute to the pool of subtask applications, like in a software ecosystem, however, a maintainer controls the rules that construct an actual process chain, as in SPLE.

A Appendix

A.1 Text Feedback Log

```
2020-02-07 11:09:19.004 10846-10846/? I/Loomo-PC Manager: Application is being executed
2020-02-07 11:09:21.999 10846-10846/com.example.processchainmanager I/Loomo-PC Manager: Create Task Object:[{"package":"com.example.processchaintext","parameter":{"text":"Okay, I will have a look for you."}},{"package":"com.example.processchaindrive","parameter":{"direction":"0.5f"}},{"package":"com.example.processchainlookaround"},{"package":"com.
example.processchaindrive","parameter":{"direction":"-0.5f"}},{"package":"com.example.processchaintext","parameter":("text":"Hey, unfortunately I couldn't find your glasses."}},{"package":"com.example.processchainmanager"}]
2020-02-07 11:09:22.001 10846-10846/com.example.processchainmanager I/Loomo-PC Manager: Start Intent
2020-02-07 11:09:22.097 10866-10866/? I/Loomo-PC Text: Application is being executed.
2020-02-07 11:09:22.157 10866-10866/? I/Loomo-PC Text: Evaluate intent: currentIndex 0; text = "Okay, I will have a look for you."
2020-02-07 11:09:22.157 10866-10866/? I/Loomo-PC Text: Set text: "Okay, I will have a look for you."
2020-02-07 11:09:27.165 10866-10866/com.example.processchaintext I/Loomo-PC Text: Start next app "com.example.processchaindrive"
2020-02-07 11:09:27.335 10886-10886/com.example.processchaindrive I/Loomo-PC Drive: Application is being executed
2020-02-07 11:09:27.408 10886-10886/com.example.processchaindrive I/Loomo-PC Drive: Evaluate intent: currentIndex = 1; direction = 0.5
2020-02-07 11:09:27.442 10886-10886/com.example.processchaindrive I/Loomo-PC Drive: Listener bound: true
2020-02-07 11:09:28.421 10886-10886/com.example.processchaindrive I/Loomo-PC Drive: Driving to (x:0.5, y:0)
2020-02-07 11:09:32.422 10886-10886/com.example.processchaindrive I/Loomo-PC Drive: Start next app "com.example.processchainlookaround"
2020-02-07 11:09:32.543 10907-10907/? I/Loomo-PC Look Around: Application is being executed.
2020-02-07 11:09:32.603 10907-10907/? I/Loomo-PC Look Around: Evaluate intent: currentIndex = 2
2020-02-07 11:09:32.638 10907-10907/? I/Loomo-PC Look Around: Listener bound: true
2020-02-07 11:09:33.609 10907-10907/com.example.processchainlookaround I/Loomo-PC Look Around: Looking right.
2020-02-07 11:09:38.611 10907-10907/com.example.processchainlookaround I/Loomo-PC Look Around: Looking left.
2020-02-07 11:09:42.618 10907-10907/com.example.processchainlookaround I/Loomo-PC Look Around: Task finished.
2020-02-07 11:09:42.626 10907-10907/com.example.processchainlookaround I/Loomo-PC Look Around: Start next app "com.example.processchaindrive"
2020-02-07 11:09:42.684 10886-10886/com.example.processchaindrive I/Loomo-PC Drive: Application is being executed
2020-02-07 11:09:42.732 10886-10886/com.example.processchaindrive I/Loomo-PC Drive: Evaluate intent: currentIndex = 3; direction = -0.5
2020-02-07 11:09:42.753 10886-10886/com.example.processchaindrive I/Loomo-PC Drive: Listener bound: true
```

```
2020-02-07 11:09:43.744 10886-10886/com.example.processchaindrive I/Loomo-PC Drive: Driving to
    ↪ (x:-0.5, y:0)
2020-02-07 11:09:47.756 10886-10886/com.example.processchaindrive I/Loomo-PC Drive: Start next
    ↪ app "com.example.processchaintext"
2020-02-07 11:09:47.827 10866-10866/com.example.processchaintext I/Loomo-PC Text: Application
    ↪ is being executed.
2020-02-07 11:09:47.857 10866-10866/com.example.processchaintext I/Loomo-PC Text: Evaluate
    ↪ intent: currentIndex 4; text = "Hey, unfortunately I couldn't find your glasses."
2020-02-07 11:09:47.857 10866-10866/com.example.processchaintext I/Loomo-PC Text: Set text: "
    ↪ Hey, unfortunately I couldn't find your glasses."
2020-02-07 11:09:52.865 10866-10866/com.example.processchaintext I/Loomo-PC Text: Start next
    ↪ app "com.example.processchainmanager"
2020-02-07 11:09:52.929 10846-10846/com.example.processchainmanager I/Loomo-PC Manager:
    ↪ Application is being executed
```

A.2 Speech Feedback Log

```
2020-02-07 11:03:39.517 10608-10608/? I/Loomo-PC Manager: Application is being executed
2020-02-07 11:03:40.517 10608-10608/com.example.processchainmanager I/Loomo-PC Manager: Create
    ↪ Task Object: [{"package":"com.example.processchainspeech","parameter":{"text":"Okay, I
    ↪ will have a look for you."}},{"package":"com.example.processchaindrive","parameter":{"
    ↪ direction":"0.5f"}},{"package":"com.example.processchainlookaround"},{"package":"com.
    ↪ example.processchaindrive","parameter":{"direction":"-0.5f"}},{"package":"com.example.
    ↪ processchainspeech","parameter":("text":"Hey, unfortunately I couldn't find your glasses
    ↪ ."}},{"package":"com.example.processchainmanager"}]
2020-02-07 11:03:40.518 10608-10608/com.example.processchainmanager I/Loomo-PC Manager: Start
    ↪ Intent
2020-02-07 11:03:40.609 10627-10627/com.example.processchainspeech I/Loomo-PC Speech:
    ↪ Application is being executed.
2020-02-07 11:03:40.668 10627-10627/com.example.processchainspeech I/Loomo-PC Speech: Evaluate
    ↪ intent: currentIndex 0; text = "Okay, I will have a look for you."
2020-02-07 11:03:40.704 10627-10627/com.example.processchainspeech I/Loomo-PC Speech: Listeners
    ↪  bound true
2020-02-07 11:03:41.676 10627-10627/com.example.processchainspeech I/Loomo-PC Speech: Speaking:
    ↪  "Okay, I will have a look for you."
2020-02-07 11:03:45.682 10627-10627/com.example.processchainspeech I/Loomo-PC Speech: Start
    ↪ next app "com.example.processchaindrive"
2020-02-07 11:03:45.826 10649-10649/com.example.processchaindrive I/Loomo-PC Drive: Application
    ↪  is being executed
2020-02-07 11:03:45.922 10649-10649/com.example.processchaindrive I/Loomo-PC Drive: Evaluate
    ↪ intent: currentIndex = 1; direction = 0.5
2020-02-07 11:03:45.976 10649-10649/com.example.processchaindrive I/Loomo-PC Drive: Listener
    ↪ bound: true
2020-02-07 11:03:46.938 10649-10649/com.example.processchaindrive I/Loomo-PC Drive: Driving to
    ↪ (x:0.5, y:0)
2020-02-07 11:03:50.955 10649-10649/com.example.processchaindrive I/Loomo-PC Drive: Start next
    ↪ app "com.example.processchainlookaround"
2020-02-07 11:03:51.146 10671-10671/com.example.processchainlookaround I/Loomo-PC Look Around:
    ↪ Application is being executed.
2020-02-07 11:03:51.227 10671-10671/com.example.processchainlookaround I/Loomo-PC Look Around:
    ↪ Evaluate intent: currentIndex = 2
2020-02-07 11:03:51.266 10671-10671/com.example.processchainlookaround I/Loomo-PC Look Around:
    ↪ Listener bound: true
```

```
2020-02-07 11:03:52.234 10671-10671/com.example.processchainlookaround I/Loomo-PC Look Around:
     ↪ Looking right.
2020-02-07 11:03:57.233 10671-10671/com.example.processchainlookaround I/Loomo-PC Look Around:
     ↪ Looking left.
2020-02-07 11:04:01.238 10671-10671/com.example.processchainlookaround I/Loomo-PC Look Around:
     ↪ Task finished.
2020-02-07 11:04:01.246 10671-10671/com.example.processchainlookaround I/Loomo-PC Look Around:
     ↪ Start next app "com.example.processchaindrive"
2020-02-07 11:04:01.317 10649-10649/com.example.processchaindrive I/Loomo-PC Drive: Application
     ↪  is being executed
2020-02-07 11:04:01.340 10649-10649/com.example.processchaindrive I/Loomo-PC Drive: Evaluate
     ↪ intent: currentIndex = 3; direction = -0.5
2020-02-07 11:04:01.352 10649-10649/com.example.processchaindrive I/Loomo-PC Drive: Listener
     ↪ bound: true
2020-02-07 11:04:02.349 10649-10649/com.example.processchaindrive I/Loomo-PC Drive: Driving to
     ↪ (x:-0.5, y:0)
2020-02-07 11:04:06.353 10649-10649/com.example.processchaindrive I/Loomo-PC Drive: Start next
     ↪ app "com.example.processchainspeech"
2020-02-07 11:04:06.413 10627-10627/com.example.processchainspeech I/Loomo-PC Speech:
     ↪ Application is being executed.
2020-02-07 11:04:06.427 10627-10627/com.example.processchainspeech I/Loomo-PC Speech: Evaluate
     ↪ intent: currentIndex 4; text = "Hey, unfortunately I couldn't find your glasses."
2020-02-07 11:04:06.444 10627-10627/com.example.processchainspeech I/Loomo-PC Speech: Listeners
     ↪  bound true
2020-02-07 11:04:07.432 10627-10627/com.example.processchainspeech I/Loomo-PC Speech: Speaking:
     ↪  "Hey, unfortunately I couldn't find your glasses."
2020-02-07 11:04:11.436 10627-10627/com.example.processchainspeech I/Loomo-PC Speech: Start
     ↪ next app "com.example.processchainmanager"
2020-02-07 11:04:11.480 10608-10608/com.example.processchainmanager I/Loomo-PC Manager:
     ↪ Application is being executed
```

A.3 App Multiple Times in a Row Log

```
2020-02-07 11:12:49.737 11101-11101/? I/Loomo-PC Manager: Application is being executed
2020-02-07 11:12:51,078 11101-11101/com.example.processchainmanager I/Loomo-PC Manager: Create
     ↪ Task Object: [{"package":"com.example.processchaintext","parameter":{"text":"One"}},{"
     ↪ package":"com.example.processchaintext","parameter":{"text":"Two"}},{"package":"com,
     ↪ example.processchaintext","parameter":{"text":"Three"}},{"package":"com.example.
     ↪ processchainmanager"}]
2020-02-07 11:12:51,079 11101-11101/com.example.processchainmanager I/Loomo-PC Manager: Start
     ↪ Intent
2020-02-07 11:12:51,180 11121-11121/? I/Loomo-PC Text: Application is being executed,
2020-02-07 11:12:51.240 11121-11121/? I/Loomo-PC Text: Evaluate intent: currentlndex 0; text =
     ↪ "One"
2020-02-07 11:12:51.240 11121-11121/? I/Loomo-PC Text: Set text: "One"
2020-02-07 11:12:56,242 11121-11121/com.example.processchaintext I/Loomo-PC Text: Start next
     ↪ app "com.example.processchaintext"
2020-02-07 11:12:56,269 11121-11121/com.example.processchaintext I/Loomo-PC Text: Application
     ↪ is being executed.
2020-02-07 11:12:56,304 11121-11121/com.example.processchaintext I/Loomo-PC Text: Evaluate
     ↪ intent: currentlndex 1; text = "Two"
2020-02-07 11:12:56,304 11121-11121/com.example.processchaintext I/Loomo-PC Text: Set text: "
     ↪ Two"
```

```
2020-02-07 11:13:01,309 11121-11121/com.example.processchaintext I/Loomo-PC Text: Start next
     ↪ app "com.example.processchaintext"
2020-02-07 11:13:01,328 11121-11121/com.example.processchaintext I/Loomo-PC Text: Application
     ↪ is being executed.
2020-02-07 11:13:01,339 11121-11121/com.example.processchaintext I/Loomo-PC Text: Evaluate
     ↪ intent: currentIndex 2; text = "Three"
2020-02-07 11:13:01,339 11121-11121/com.example.processchaintext I/Loomo-PC Text: Set text: "
     ↪ Three"
2020-02-07 11:13:06,347 11121-11121/com.example.processchaintext I/Loomo-PC Text: Start next
     ↪ app "com.example.processchainmanager"
2020-02-07 11:13:06.396 11101-11101/com.example.processchainmanager I/Loomo-PC Manager:
     ↪ Application is being executed
```

A.4 Wrong Package Name Log

```
2020-02-07 11:19:27.632 11578-11578/? I/Loomo-PC Manager: Application is being executed
2020-02-07 11:19:30,122 11578-11578/com.example.processchainmanager I/Loomo-PC Manager: Create
     ↪ Task Object: [{"package":"com.example.processchaintext","parameter":{"text":"One"}},{"
     ↪ package":"com.example.processchaintext","parameter":{"text":"Two"}},{"package":"com.
     ↪ example.processchain","parameter":{"text":"Three"}},{"package":"com.example.
     ↪ processchainmanager"}]
2020-02-07 11:19:30,123 11578-11578/com.example.processchainmanager I/Loomo-PC Manager: Start
     ↪ Intent
2020-02-07 11:19:30,149 11465-11465/com.example.processchaintext I/Loomo-PC Text: Application
     ↪ is being executed.
2020-02-07 11:19:30,160 11465-11465/com.example.processchaintext I/Loomo-PC Text: Evaluate
     ↪ intent: currentlndex 0; text = "One"
2020-02-07 11:19:30,160 11465-11465/com.example.processchaintext I/Loomo-PC Text: Set text: "
     ↪ One"
2020-02-07 11:19:35,165 11465-11465/com.example.processchaintext I/Loomo-PC Text: Start next
     ↪ app "com.example.processchaintext"
2020-02-07 11:19:35,179 11465-11465/com.example.processchaintext I/Loomo-PC Text: Application
     ↪ is being executed.
2020-02-07 11:19:35,190 11465-11465/com.example.processchaintext I/Loomo-PC Text: Evaluate
     ↪ intent: currentIndex 1; text = "Two"
2020-02-07 11:19:35,190 11465-11465/com.example.processchaintext I/Loomo-PC Text: Set text: "
     ↪ Two"
2020-02-07 11:19:40,192 11465-11465/com.example.processchaintext I/Loomo-PC Text: Failed to
     ↪ start next app.
2020-02-07 11:19:40,232 11578-11578/com.example.processchainmanager I/Loomo-PC Manager:
     ↪ Application is being executed
```

A.5 Faulty Task Object Log

```
2020-02-07 11:37:36.082 14126-14126/com.example.processchainmanager I/Loomo-PC Manager:
     ↪ Application is being executed
2020-02-07 11:37:37.830 14126-14126/com.example.processchainmanager I/Loomo-PC Manager: Create
     ↪ Task Object: {}
2020-02-07 11:37:37.837 14126-14126/com.example.processchainmanager I/Loomo-PC Manager:
     ↪ Something went wrong when trying to convert the taskObject to a JSONArray: org.json.
     ↪ JSONException: Value {} of type org.json.JSONObject cannot be converted to JSONArray
```

A.6 Large Task Object Log

```
2020-02-07 11:50:30.508 29239-29239/? I/Loomo-PC Manager: Application is being executed
2020-02-07 11:50:33.758 29239-29239/com.example.processchainmanager I/Loomo-PC Manager: Create
     ↪ Task Object: [{package: "com.example.processchaintext"; parameter: {"text": "Lorem ipsum
     ↪  dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut
     ↪ labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo
     ↪  duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem
     ↪ ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam
     ↪ nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua
     ↪ . At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no
     ↪  sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet,
     ↪ consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore
     ↪ magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et
     ↪ ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit
     ↪ amet. Duis autem vel eum iriure dolor in hendrerit in vulputate velit esse molestie
     ↪ consequat, vel illum dolore eu feugiat nulla facilisis at vero eros et accumsan et iusto
     ↪  odio dignissim qui blandit praesent luptatum zzril delenit augue duis dolore te feugait
     ↪  nulla facilisi. Lorem ipsum dolor sit amet, consectetuer adipiscing elit, sed diam
     ↪ nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat. Ut wisi
     ↪ enim ad minim veniam, quis nostrud exerci tation ullamcorper suscipit lobortis nisl ut
     ↪ aliquip ex ea commodo consequat. Duis autem vel eum iriure dolor in hendrerit in
     ↪ vulputate velit esse molestie consequat, vel illum dolore eu feugiat nulla facilisis at
     ↪ vero eros et accumsan et iusto odio dignissim qui blandit praesent luptatum zzril
     ↪ delenit augue duis dolore te feugait nulla facilisi. Nam liber tempor cum soluta nobis
     ↪ eleifend option congue nihil imperdiet doming id quod mazim placerat facer possim assum.
     ↪  Lorem ipsum dolor sit amet, consectetuer adipiscing elit, sed diam nonummy nibh euismod
     ↪  tincidunt ut laoreet dolore magna aliquam erat volutpat. Ut wisi enim ad minim veniam,
     ↪ quis nostrud exerci tation ullamcorper suscipit lobortis nisl ut aliquip ex ea commodo
     ↪ consequat. Duis autem vel eum iriure dolor in hendrerit in vulputate velit esse molestie
     ↪  consequat, vel illum dolore eu feugiat nulla facilisis. At vero eos et accusam et justo
     ↪  duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem
     ↪ ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam
     ↪ nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua
     ↪ . At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no
     ↪  sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet,
     ↪ consetetur sadipscing elitr, At accusam aliquyam diam diam dolore dolores duo eirmod eos
     ↪  erat, et nonumy sed tempor et et invidunt justo labore Stet clita ea et gubergren, kasd
     ↪  magna no rebum. sanctus sea sed takimata ut vero voluptua. est Lorem ipsum dolor sit
     ↪ amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod
     ↪ tempor invidunt ut labore et dolore magna aliquyam erat. Consetetur sadipscing elitr,
     ↪ sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam
```

```
      ↪  voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd
      ↪ gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit
      ↪  amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et
      ↪ dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo
      ↪ dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum
      ↪ dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy
      ↪  eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At
      ↪ vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea
      ↪ takimata sanctus. Lorem ipsum dolor sit amet, conset
2020-02-07 11:50:34.536 29239-29239/com.example.processchainmanager I/Loomo-PC Manager: Start
      ↪ next app "com.example.processchaintext"
2020-02-07 11:50:34.549 29239-29239/com.example.processchainmanager I/Loomo-PC Manager: Could
      ↪ not start Activity. java.lang.RuntimeException: Failure from system
```

List of Definitions

List of Figures

List of Tables

List of Listings

Bibliography

[AK09] Sven Apel and Christian Kästner. “An Overview of Feature-Oriented Software Development”. In: *Journal of Object Technology* (JOT) 8 (July 2009), pp. 49–84. DOI: `10.5381/jot.2009.8.5.c5` (cit. on p. 12).

[AKP19] Paola Ardón, Kaisar Kushibar, and Songyou Peng. “A Hybrid SLAM and Object Recognition System for Pepper Robot”. Mar. 2019 (cit. on p. 9).

[Ang+18] A. Angleraud, Q. Houbre, V. Kyrki, and R. Pieters. “Human-Robot Interactive Learning Architecture using Ontologies and Symbol Manipulation”. In: *2018 27th IEEE International Symposium on Robot and Human Interactive Communication* (RO-MAN). Aug. 2018, pp. 384–389. DOI: `10.1109/ROMAN.2018.8525580` (cit. on p. 11).

[Anq+10] Nicolas Anquetil, Uirá Kulesza, Ralf Mitschke, Ana Moreira, Jean-Claude Royer, Andreas Rummler, and André Sousa. “A model-driven traceability framework for software product lines”. In: *Software & Systems Modeling* 9.4 (Sept. 2010), pp. 427–451. DOI: `10.1007/s10270-009-0120-9`. URL: `https://doi.org/10.1007/s10270-009-0120-9` (cit. on p. 12).

[BB01] Felix Bachmann and Len Bass. “Managing Variability in Software Architectures”. In: *SIGSOFT Softw. Eng. Notes* 26.3 (May 2001), pp. 126–132. DOI: `10.1145/379377.375274`. URL: `http://doi.acm.org/10.1145/379377.375274` (cit. on p. 12).

[BC19] Maxime Busy and Maxime Caniot. “qiBullet, a Bullet-based simulator for the Pepper and NAO robots”. Sept. 2019 (cit. on p. 9).

[BCH15a] Jan Bosch, Rafael Capilla, and Rich Hilliard. “Trends in systems and software variability”. In: *IEEE Software* 32.3 (2015), pp. 44–61. DOI: `10.1109/MS.2015.74` (cit. on p. 22).

[BCH15b] Davide Brugali, Rafael Capilla, and Mike Hinchey. “Dynamic variability meets robotics”. In: *Computer* 48.12 (2015), pp. 94–97. DOI: `10.1109/MC.2015.354` (cit. on p. 21).

[Bec+19] Lucile Bechade, Guillaume Dubuisson-Duplessis, Gabrielle Pittaro, Mélanie Garcia, and Laurence Devillers. “Towards Metrics of Evaluation of Pepper Robot as a Social Companion for the Elderly: 8th International Workshop on Spoken Dialog Systems”. In: Jan. 2019, pp. 89–101. DOI: `10.1007/978-3-319-92108-2_11` (cit. on p. 9).

[Ben+13] David Benavides, Alexander Felfernig, José Galindo, and Florian Reinfrank. “Automated Analysis in Feature Modeling and Product Configuration”. In: vol. 7925. June 2013 (cit. on p. 15).

[Ber+13] Thorsten Berger, Steven She, Rafael Lotufo, Andrzej Wasowski, and Krzysztof Czarnecki. “A Study of Variability Models and Languages in the Systems Software Domain”. In: *IEEE Transactions on Software Engineering* 39.12 (2013), pp. 1611–1640. DOI: `10.1109/TSE.2013.34` (cit. on pp. 15, 27).

[Ber+14] Thorsten Berger, Rolf Helge Pfeiffer, Reinhard Tartler, Steffen Dienst, Krzysztof Czarnecki, Andrzej Wąsowski, and Steven She. “Variability mechanisms in software ecosystems”. In: *Information and Software Technology* 56.11 (2014), pp. 1520–1535. DOI: `10.1016/j.infsof.2014.05.005` (cit. on pp. 20 sq., 26, 32).

[BKM16] Mohammed Ben␣Daya, Uday Kumar, and D.N. Murthy. “Maintenance Decision Models and Optimization”. In: Feb. 2016, pp. 299–299. DOI: `10.1002/9781118926581.part3` (cit. on p. 27).

[Bos00] Jan Bosch. *Design and Use of Software Architectures – Adopting and Evolving a Product Line Approach*. Pearson Education, Jan. 2000 (cit. on pp. 19 sq.).

[Bos09] Jan Bosch. “From software product lines to software ecosystems”. In: *Proceedings of the 13th International Software Product Line Conference* Splc (2009), pp. 111–119. URL: `http://dl.acm.org/citation.cfm?id=1753251` (cit. on pp. 20 sq.).

[BSR10] David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. “Automated Analysis of Feature Models 20 Years Later: A Literature Review”. In: *Inf. Syst.* 35.6 (Sept. 2010), pp. 615–636. DOI: `10.1016/j.is.2010.01.001`. URL: `http://dx.doi.org/10.1016/j.is.2010.01.001` (cit. on p. 15).

[Bzu+12] Conrad Bzura, Hosung Im, Kevin Malehorn, and Wan Liu. “The Emerging Role of Robotics in Personal Health Care: Bringing Smart Health Care Home”. In: Worcester Polytechnic Institute, 2012 (cit. on pp. 1, 7).

[CAA09] Lianping Chen, Muhammad Ali Babar, and Nour Ali. “Variability Management in Software Product Lines: A Systematic Review”. In: *Proceedings of the 13th International Software Product Line Conference*. SPLC ’09. San Francisco, California, USA: Carnegie Mellon University, 2009, pp. 81–90. URL: `http://dl.acm.org/citation.cfm?id=1753235.1753247` (cit. on pp. 14–16, 27).

[CE00] Krzysztof Czarnecki and Ulrich W. Eisenecker. “Generative programming - methods, tools and applications”. In: 2000 (cit. on pp. 13, 15).

[CE99] Krzysztof Czarnecki and Ulrich W. Eisenecker. “Components and Generative Programming”. In: *Software Engineering — ESEC/FSE ’99*. Ed. by Oscar Nierstrasz and Michel Lemoine. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 2–19 (cit. on pp. 13 sq.).

[Cul+15] Antoine Cully, Jeff Clune, Danesh Tarapore, and Jean Baptiste Mouret. “Robots that can adapt like animals”. In: *Nature* 521.7553 (2015), pp. 503–507. DOI: `10.1038/nature14422` (cit. on p. 7).

[Cza+12] Krzysztof Czarnecki, Paul Grünbacher, Rick Rabiser, Klaus Schmid, and Andrzej Wąsowski. “Cool features and tough decisions: A comparison of variability modeling approaches”. In: *ACM International Conference Proceeding Series* (2012), pp. 173–182. DOI: `10.1145/2110147.2110167` (cit. on p. 27).

[Dav87] Stanley M. Davis. “Future perfect”. In: (1987) (cit. on p. 19).

[De +18] Laurens De Gauquier, Hoang-Long Cao, Pablo Gomez Esteban, Albert De Beir, Stephanie Van de sanden, Kim Willems, Malaika Brengman, and Bram Vanderborght. “Humanoid Robot Pepper at a Belgian Chocolate Shop”. In: Mar. 2018, pp. 373–373. DOI: `10.1145/3173386.3177535` (cit. on p. 9).

[EDS12] Sascha El-Sharkawy, Stephan Dederichs, and Klaus Schmid. “From Feature Models to Decision Models and Back Again an Analysis Based on Formal Transformations”. In: *Proceedings of the 16th International Software Product Line Conference - Volume 1*. SPLC ’12. Salvador, Brazil: ACM, 2012, pp. 126–135. DOI: `10.1145/2362536.2362555`. URL: `http://doi.acm.org/10.1145/2362536.2362555` (cit. on pp. 15 sq.).

[Gar+19] Sergio García, Daniel Strüber, Davide Brugali, Alessandro Di Fava, Philipp Schillinger, Patrizio Pelliccione, and Thorsten Berger. “Variability Modeling of Service Robots”. In: (2019), pp. 1–6. DOI: `10.1145/3302333.3302350` (cit. on pp. 1, 7, 11, 21, 57).

[GHS04] Birgit Graf, Matthias Hans, and Rolf D. Schraft. “Care-O-bot II — Development of a Next Generation Robotic Home Assistant”. In: *Autonomous Robots* 16.2 (Mar. 2004), pp. 193–205. DOI: `10.1023/B:AURO.0000016865.35796.e9`. URL: `https://doi.org/10.1023/B:AURO.0000016865.35796.e9` (cit. on p. 9).

[Góm+19] Cristopher Gómez, Matías Mattamala, Tim Resink, and Javier Ruiz-del-Solar. “Visual SLAM-Based Localization and Navigation for Service Robots: The Pepper Case”. In: Aug. 2019, pp. 32–44. DOI: `10.1007/978-3-030-27544-0_3` (cit. on p. 9).

[Hab+13] Arne Haber, Katrin Hölldobler, Carsten Kolassa, Markus Look, Bernhard Rumpe, Klaus Müller, and Ina Schaefer. “Engineering Delta Modeling Languages”. In: *Proceedings of the 17th International Software Product Line Conference*. SPLC ’13. Tokyo, Japan: ACM, 2013, pp. 22–31. DOI: `10.1145/2491627.2491632`. URL: `http://doi.acm.org/10.1145/2491627.2491632` (cit. on p. 12).

[Ham+19] N. Hammoudeh Garcia, M. Lüdtke, S. Kortik, B. Kahl, and M. Bordignon. “Bootstrapping MDE Development from ROS Manual Code - Part 1: Metamodeling”. In: *2019 Third IEEE International Conference on Robotic Computing (IRC)*. Feb. 2019, pp. 329–336. DOI: `10.1109/IRC.2019.00060` (cit. on p. 11).

[HCH] Arnaud Hubaux, Andreas Classen, and Patrick Heymans. “P.: A preliminary review on the application of feature diagrams in practice”. In: *In: VaMoS’10* (2010 (cit. on p. 15).

[Hee+10] Marcel Heerink, Ben Kröse, Vanessa Evers, and Bob Wielinga. “Assessing Acceptance of Assistive Social Agent Technology by Older Adults: the Almere Model”. In: *International Journal of Social Robotics* 2.4 (2010), pp. 361–375. DOI: `10.1007/s12369-010-0068-5` (cit. on pp. 5 sq.).

[Ii14] ISO/TC 184 Automation systems ISO/TC 184 Industrielle Automatisierungssysteme and ISO/TC 184 Systèmes d’automatisation industrielle et intégration integration. ISO 13482 *Robots and robotic devices - Safety requirements for personal care robots*. Standard. 2014 (cit. on p. 6).

[Ing+18] Juan F. Inglés-Romero, Juan Manuel Espín, Rubén Jiménez-Andreu, Roberto Font, and Cristina Vicente-Chicote. “Towards the use of Quality-of-Service Metrics in Reinforcement Learning: A Robotics Example”. In: *MODELS Workshops*. 2018 (cit. on p. 7).

[ISO06] ISO/TC 159 Ergonomics ISO/TC 159 Ergonomie. ISO 9241-110 *Ergonomics of human-system interaction - Part 110: Dialogue principles*. Standard. 2006 (cit. on p. 43).

[JGJ97] Ivar Jacobson, Martin Griss, and Patrik Jonsson. *Software reuse architecture, process and organization for business success ; [conforms to the unified modeling language]*. 1. print. New York, NY [u.a.] , Harlow [u.a.]: Addison-Wesley, 1997. URL: `http://slubdd.de/katalog?TN_libero_mab2)51315` (cit. on p. 12).

[KAK08] Christian Kästner, Sven Apel, and Martin Kuhlemann. "Granularity in software product lines". In: *Proceedings - International Conference on Software Engineering* (2008), pp. 311–320. DOI: `10.1145/1368088.1368131` (cit. on pp. 16–18).

[Kan+90] Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. "Feature-Oriented Domain Analysis (FODA) Feasibility Study". In: November (1990). DOI: `CMU/SEI-90-TR-21` (cit. on p. 15).

[Kit+15] Ralf Kittmann, Tim Fröhlich, Johannes Schäfer, Ulrich Reiser, Florian Weisshardt, and Andreas Haug. "Let me Introduce Myself: I am Care-O-bot 4, a Gentleman Robot". In: Sept. 2015 (cit. on pp. 9 sq.).

[Koc15] Robert Koch-Institut. "Welche Auswirkungen hat der demografische Wandel auf Gesundheit und Gesundheitsversorgung?" In: *Gesundheit in Deutschland*. Kapitel 9. Robert Koch-Institut, Epidemiologie und Gesundheitsberichterstattung, 2015. DOI: `10.17886/rkipubl-2015-003-9` (cit. on p. 1).

[Lie+19] Florian Lier, Johannes Kummert, Patrick Renner, and Sven Wachsmuth. "ToBI - Team of Bielefeld Enhancing the Robot Capabilities of the Social Standard Platform Pepper". In: Aug. 2019, pp. 524–535. DOI: `10.1007/978-3-030-27544-0_43` (cit. on p. 9).

[LNS13] Przemyslaw Lasota, Stefanos Nikolaidis, and Julie A. Shah. "Developing an Adaptive Robotic Assistant for Close Proximity Human-Robot Collaboration in Space". In: (2013), pp. 1–8. DOI: `10.2514/6.2013-4806` (cit. on p. 7).

[Lot+14] Alex Lotz, J Inglés-Romero, Dennis Stampfer, Matthias Lutz, Cristina Vicente-Chicote, and Christian Schlegel. "Towards a Stepwise Variability Management Process for Complex Systems – A Robotics Perspective". In: *Int. Journal of Information System Modeling and Design (IJISMD)* 5 (Nov. 2014), pp. 55–74. DOI: `10.4018/ijismd.2014070103` (cit. on p. 22).

[Lun09] Mircea F. Lungu. "Reverse engineering software ecosystems". English. PhD thesis. Faculty of Informatics, University of Lugano, Switzerland, Nov. 2009 (cit. on p. 20).

[LVK18] J. Lundell, F. Verdoja, and V. Kyrki. "Hallucinating Robots: Inferring Obstacle Distances from Partial Laser Measurements". In: *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. Oct. 2018, pp. 4781–4787. DOI: `10.1109/IROS.2018.8594399` (cit. on p. 11).

[LW18] Florian Lier and Sven Wachsmuth. "Towards an Open Simulation Environment for the Pepper Robot". In: Mar. 2018, pp. 175–176. DOI: `10.1145/3173386.3177088` (cit. on p. 9).

[MA02] Dirk Muthig and Colin Atkinson. "Model-Driven Product Line Architectures". In: *Software Product Lines*. Ed. by Gary J. Chastek. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 110–129. DOI: `10.1007/3-540-45652-x_8` (cit. on p. 15).

[McG09] John McGregor. "Ecosystems." In: *Journal of Object Technology* 8 (Sept. 2009), pp. 7–16. DOI: `10.5381/jot.2009.8.6.c1` (cit. on p. 20).

[Neu16] Dana Neumann. *Human Assistant Robotics in Japan - Challenges and Opportunities for European Companies-*. 2016, pp. 8–9 (cit. on pp. 5 sq.).

[Ngu06] Thang Nguyen. “A decision model for managing software development projects”. In: *Information & Management* 43 (Jan. 2006), pp. 63–75. DOI: `10.1016/j.im.2005.01.006` (cit. on p. 27).

[OO12] Ioan Orha and Stefan Oniga. “Assistance and telepresence robots: a solution for elderly people”. In: *Carpathian Journal of Electronic and Computer Engineering* 5.1 (2012), pp. 87–90 (cit. on pp. 1, 5 sq.).

[Par76] D. L. Parnas. “On the Design and Development of Program Families”. In: *IEEE Transactions on Software Engineering* SE-2.1 (Mar. 1976), pp. 1–9. DOI: `10.1109/TSE.1976.233797` (cit. on p. 19).

[PBL05] Klaus Pohl, Günter Böckle, and Frank van der Linden. *Software product line engineering - Foundations, Principles, and Techniques*. Springer Berlin Heidelberg, 2005. DOI: `10.1007/978-3-642-36583-6_1` (cit. on pp. 11 sq., 14, 16, 19–21).

[PG18] Amit Kumar Pandey and Rodolphe Gelin. “A Mass-Produced Sociable Humanoid Robot: Pepper: The First Machine of Its Kind”. In: *IEEE Robotics & Automation Magazine* PP (July 2018), pp. 1–1. DOI: `10.1109/MRA.2018.2833157` (cit. on pp. 8 sq.).

[Pol05] Martha E. Pollack. “Intelligent Technology for an Aging Population: The Use of AI to Assist Elders with Cognitive Impairment”. In: *AI Magazine* 26.2 (June 2005), p. 9. DOI: `10.1609/aimag.v26i2.1810`. URL: `https://www.aaai.org/ojs/index.php/aimagazine/article/view/1810` (cit. on p. 1).

[RBR11] Fabricia Roos-Frantz, David Benavides, and Antonio Ruiz-Cortés. “Feature Model to Orthogonal Variability Model Transformation towards Interoperability between Tools”. In: (May 2011) (cit. on pp. 15 sq.).

[RCW18] Zhang Ruishu, Zhang Chang, and Zheng Weigang. “The status and development of industrial robots”. In: *IOP Conference Series: Materials Science and Engineering* 423 (Nov. 2018), p. 012051. DOI: `10.1088/1757-899X/423/1/012051` (cit. on p. 7).

[Rei+09] Ulrich Reiser, Christian Connette, Jan Fischer, Jens Kubacki, Alexander Bubeck, Florian Weisshardt, Theo Jacobs, Christopher Parlitz, Martin Hagele, and Alexander Verl. “Care-O-bot ®3 - Creating a product vision for service robot applications by integrating design and technology”. In: Nov. 2009, pp. 1992–1998. DOI: `10.1109/IROS.2009.5354526` (cit. on p. 9).

[Ros+11] Marko Rosenmüller, Norbert Siegmund, Mario Pukall, and Sven Apel. “Tailoring Dynamic Software Product Lines”. In: *SIGPLAN Not.* 47.3 (Oct. 2011), pp. 3–12. DOI: `10.1145/2189751.2047866`. URL: `http://doi.acm.org/10.1145/2189751.2047866` (cit. on p. 12).

[RS10] Marko Rosenmüller and Norbert Siegmund. “Automating the Configuration of Multi Software Product Lines.” In: *VaMoS* 10 (2010), pp. 123–130 (cit. on p. 27).

[RS18] Fabrizio Ruggeri and Refik Soyer. “Decision Models for Software Testing”. In: July 2018, pp. 277–286. DOI: `10.1002/9781119357056.ch11` (cit. on p. 27).

[Sch+12] Ina Schaefer, Rick Rabiser, David Clarke, Lorenzo Bettini, David Benavides, Goetz Botterweck, Animesh Pathak, Salvador Trujillo, and Karina Villela. "Software Diversity – State of the Art and Perspectives". In: *International Journal on Software Tools for Technology Transfer* October (2012), pp. 477–495. DOI: `10.1007/s10009-012-0253-y` (cit. on pp. 14, 16, 18).

[Sch+15] Christian Schlegel, Alex Lotz, Matthias Lutz, Dennis Stampfer, Juan F. Inglés-Romero, and Cristina Vicente-Chicote. "Model-driven software systems engineering in robotics: Covering the complete life-cycle of a robot". In: *it - Information Technology* 57.2 (2015), pp. 85–98. DOI: `10.1515/itit-2014-1069` (cit. on pp. 7, 11, 21 sq.).

[Sep18] Mikko Seppälä. "A secure and conflict free control platform for Care-O-Bot 4". MA thesis. Aalto University School of electrical engineering, Finland, 2018 (cit. on p. 11).

[SGB01] Mikael Svahnberg, Jilles Van Gurp, and Jan Bosch. "On the Notion of Variability in Software Product Lines By On the Notion of Variability in Software Product Lines". In: *Science* 02 (2001). URL: `www:http://www.ipd.hk-r.se/[msv%7B%5C%%7D7Cjvg%7B%5C%%7D7Cbosch]` (cit. on pp. 11 sq., 19).

[SM99] C. Schaeffer and T. May. "Care-O-bot: A System for Assisting Elderly or Disabled Persons in Home Environments". In: *Proceedings of AAATE-99 Düsseldorf* (1999) (cit. on p. 9).

[SSA14] Christoph Seidl, Ina Schaefer, and Uwe Aßmann. "Integrated management of variability in space and time in software families". In: (2014), pp. 22–31. DOI: `10.1145/2648511.2648514` (cit. on pp. 12–14, 16–21).

[SSB17] Carolin Schmitt, Johannes Schäfer, and Michael Burmester. "Wie wirkt der Care-O-bot 4 im Verkaufsraum? Evaluation der User Experience eines Serviceroboters". In: Sept. 2017. DOI: `10.18420/muc2017-up-0171` (cit. on p. 11).

[TJM96] Mitchell M. Tseng, Jianxin Jiao, and M. Eugene Merchant. "Design for Mass Customization". In: *CIRP Annals* 45.1 (1996), pp. 153–156. DOI: `https://doi.org/10.1016/S0007-8506(07)63036-4` (cit. on p. 19).

[TTZ17] Sofia Thunberg, Sam Thellman, and Tom Ziemke. "Don't Judge a Book by its Cover: A Study of the Social Acceptance of NAO vs. Pepper". In: Oct. 2017, pp. 443–446. DOI: `10.1145/3125739.3132583` (cit. on p. 9).

[Wes19] Mathias Weske. "Business Decision Modelling". In: June 2019, pp. 241–257. DOI: `10.1007/978-3-662-59432-2_5` (cit. on p. 27).