END USER SOFTWARE PRODUCT LINE SUPPORT FOR SMART SPACES

by

Vasilios Tzeremes
A Dissertation
Submitted to the
Graduate Faculty
of
George Mason University
in Partial Fulfillment of
The Requirements for the Degree
of
Doctor of Philosophy
Information Technology

Committee:

___________________________________	Dr. Hassan Gomaa, Dissertation Director
___________________________________	Dr. Jeff Offutt, Committee Member
___________________________________	Dr. Jens-Peter Kaps, Committee Member
___________________________________	Dr. Thomas LaToza, Committee Member
___________________________________	Dr. Stephen Nash, Senior Associate Dean
___________________________________	Dr. Kenneth S. Ball, Dean, Volgenau School of Engineering
Date:______________________________	Fall Semester 2016 George Mason University Fairfax, VA

End User Software Product Line Support for Smart Spaces

A Dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy at George Mason University

by

Vasilios Tzeremes
Masters of Science
American University, 2004
Bachelor of Business
Technological Educational Institute of Athens, 1999

Director: Hassan Gomaa, Professor
Department of Computer Science

Fall Semester 2016
George Mason University
Fairfax, VA

DEDICATION

I dedicate this work to:

My wonderful children Konstantine and Zoe for all the time and attention I took from you to complete this dissertation.

My beautiful wife Dora for your patience, support and motivation. You have been there for me every step of the way in this journey. Without your understanding, love and sacrifices I would have never finished.

To my parents Konstantino and Giannoula for your encouragement, trust and support throughout the years.

ACKNOWLEDGEMENTS

Firstly, I would like to thank my advisor Dr. Hassan Gomaa for his continuous support and the countless hours he spend throughout my PhD study, for his patience, motivation, and extending my academic and technical capabilities beyond what I thought possible. His guidance, thorough feedback and trust enabled me to complete this PhD dissertation. It was an honor having Dr. Gomaa, an authority in the domains of software engineering and software product lines to be my advisor.

Besides my advisor, I would like to thank Dr. João Pedro Sousa for helping me to start this research, making me part of the Team Computing (TeC) research group and expanding my knowledge in the domains of ubiquitous computing, end user development and smart spaces.

I would like to thank the rest of my committee: Dr. Offutt, Dr. Kaps, and Dr. LaToza, for their support and valuable feedback that helped shape this research.

I would also like to thank Xiang Shen for his help, support and work on the TeC Android simulator.

Last but not the least, I would like to thank my sister Despina, parents-in-law George and Georgia, the rest of my family, friends, business partners, coworkers and everyone else throughout the years that helped me to complete this research and my life in general.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

END USER SOFTWARE PRODUCT LINE SUPPORT FOR SMART SPACES

Vasilios Tzeremes, Ph.D.

George Mason University, 2016

Dissertation Director: Dr. Hassan Gomaa

Smart spaces are physical environments equipped with pervasive technology that sense and react to human activities and changes in the environment. End User Development (EUD) skills vary significantly among end users who want to design, develop and deploy software applications for their smart spaces. Typical end user development is opportunistic, requirements are usually unplanned and undocumented, applications are simplistic in nature, design is ad-hoc, reuse is limited, and software testing is typically haphazard, leading to many quality issues. On the other hand, technical end users with advanced EUD skills and domain expertise have the ability to create sophisticated software applications for smart spaces that are well designed and tested.

This research presents a systematic approach for adopting reuse in end user development for smart spaces by using Software Product Line (SPL) concepts. End User (EU) SPL Designers (who are technical end users and domain experts) design and develop EU SPLs for smart spaces whereas less technical end users derive their individual smart

space applications from these SPLs. Incorporating SPL concepts in EUD for smart spaces makes it easier for novice end users to derive applications for their spaces without having to interface directly with devices, networks, programming logic, etc. End users only have to select and configure the EU SPL features needed for their space. Another benefit of this approach is that it promotes reuse. End user requirements are mapped to product line features that are realized by common, optional, and variant components available in smart spaces. Product line features and the corresponding component product line architecture can then be used to derive EU applications. Derived EU applications can then be deployed to different smart spaces, thereby avoiding end users having to create EU applications from scratch. Finally the proposed approach has the potential of improving software quality since testing will be an integral part of EU SPL process.

In particular, this research has: (a) defined a systematic approach for EU SPL Designers to design and develop EU SPLs, (b) provided an EU SPL application derivation approach to enable end users to derive software applications for their spaces, (c) designed an EU SPL meta-model to capture the underlying representation of EU SPL and derived application artifacts in terms of meta-classes and relationships that supports different EUD platforms, (d) designed and implemented an EUD development environment that supports EU SPL development and application derivation, and (e) provided a testing approach and framework for systematic testing of EU SPLs and derived applications.

1 INTRODUCTION

1.1 Background

As computing becomes ubiquitous, software demands are rapidly increasing. Software requirements for end users are becoming personalized and often fluctuate. Professional engineers do not have the capacity and domain knowledge to satisfy all software needs. End users know their own context and needs better than anybody else, and they often have real-time awareness of shifts in their respective domains (Burnett and Myers, 2014). End users are already involved in software development and outnumber professional engineers. For instance, the current ratio of end users to professional engineers is 30-to-1 (Burnett and Scaffidi, 2014). End User Development (EUD) involves a set of methods, techniques, and tools that enable users of software systems, who are acting as non-professional software developers, to create, modify, or extend a software artifact (Lieberman et al., 2006). Examples of EUD are spreadsheet programming, visual programming, email rule filters, web site creation tools, etc.

Another prominent area for end user development is smart spaces. Smart spaces are environments equipped with visual and audio sensing systems, pervasive devices, sensors, and networks that can perceive and react to people, sense on-going human activities and respond to them (Singh et al., 2006). Several End User (EUD) environments for smart spaces have been proposed to assist end users to create applications for their smart environments. EUD environments for smart spaces provide user interfaces for end users to create software applications and interconnect applications with devices deployed in a smart space. Jigsaw (Humble et al., 2003), Puzzle (Danado and Paternò, 2012), PIP (Chin et al., 2010), FedNet (Kawsar et al., 2008), and TeC (Sousa, 2010) are examples of EUD environments. EUD environments enable end users to create their own applications for home security, building automation, space notifications, energy conservation and office ergonomics.

Having end users creating software applications has several benefits. Some of the benefits are that it empowers end users to create software applications, the applications are built to the end user specifications and there is better adaptation of the software applications by end users. Having end users creating software applications also has challenges. End users have different technological backgrounds. Thus not all end users have the same development abilities. Furthermore EUD is more opportunistic than systematic, requirements are usually unplanned and undocumented, reuse is ad-hoc, and software testing is typically haphazard, leading to quality issues (Ko et al., 2011). End User Software Engineering (EUSE) focuses on approaches, techniques and tools to improve the quality of end user software (Burnett, 2009). Software Product Line (SPL) methods can also help end users to reuse work of others and improve the software quality.

This research investigates how SPL concepts can be applied to end user development for smart spaces.

1.2 Motivation

Several EUD environments for smart spaces have been proposed to enable end users to customize their smart spaces. One of the problems with existing solutions is that they either target a specific group of end users or they assume end users have a baseline technical background. In fact, end users have different computer skills, personality characteristics, ages, gender (Beckwith and Burnett, 2004) etc. Technical end users and domain experts have the ability to create sophisticated software for their smart spaces. However, less technical end users find it difficult to create software for their smart spaces due to a lack of technical knowledge, domain expertise, and difficulties using EUD environments for smart spaces (Kawsar et al., 2008). It would be beneficial to enable end users to salvage the work of more technical end users and domain experts to create software applications for their spaces.

Several quality issues have been reported by applications created by end users. Some of these include errors in the logic, compatibility issues etc. (Burnett, 2009). The domain of EUSE is derived from software engineering and provides systematic approaches for end users to create quality software. Reuse is also one of the areas that EUSE identifies as promising for improving end user software quality and promoting end user development. Some of the issues of reuse in EUD is that end users don't design their software applications for reuse and even if they do, other end users have difficulties finding and reusing the software applications to address their needs (Burnett, 2009). SPL technology addresses software reuse of requirements, designs and implementations, and could assist with EUSE. The problem is that SPL methods target professional software engineers rather than end users. SPL creation involves requirements gathering, commonality/variability analysis, feature modeling, variable architecture design, component design and implementation. In an end user environment, the development process is more agile. End users are not familiar with prescriptive SPL methods and therefore modifications are needed to define a SPL method to target end users.

By adopting reuse, end users would not have to duplicate work to create similar applications. In addition, reuse of more sophisticated and stable end user applications can increase the end user satisfaction that could lead to better adoption of EUD for smart spaces.

1.3 Glossary of Relevant Terms

This section provides a common vocabulary for terms used in related literature and throughout this dissertation.

- **End User Development (EUD)** – a set of methods, techniques, and tools that enable users of software systems, who are acting as non-professional software developers, to create, modify, or extend a software artifact (Lieberman et al., 2006) (Chapter 1).
- **Software Product Lines (SPL)** – a set of software intensive systems sharing a common, managed set of features that specify the specific needs of a particular market segment or mission and are developed from a common set of core assets in a prescribed way (Clements and Northrop, 2002) (Chapter 2).
- **Smart Spaces** – ordinary environments equipped with visual and audio sensing systems, pervasive devices, sensors, and networks that can perceive and react to

people, sense ongoing human activities and respond to them (Singh et al., 2006). Smart spaces are also referred to smart environments in part of the literature (Chapter 2).

- **End User Development (EUD) Environments for Smart Spaces** – provide user interfaces for end users to create software applications and interconnect applications with devices deployed in a smart space. (Chapter 2). EUD Environments are also referred as platforms (Chapter 5) and as EUD Tools in literature.
- **End User Product Lines (EU SPL)** – product lines for smart spaces created by technical end users and domain experts (Chapter 4).
- **End User (EU) Application** – software application for smart spaces derived by end users from the EU SPL (Chapter 4).
- **End User Product Line (EU SPL) Process** – a systematic approach for EU SPL designers who are technical end users and domain experts to design and develop end user software product lines for smart spaces that end users can use to derive applications for their smart spaces (Chapter 4). The EU SPL process consists of the End User Product Line Engineering (EUPLE) process and the End User Application Engineering (EUAE) process.
- **End User Product Line Engineering (EUPLE)** – is the process that technical end users and domain experts follow to develop EU SPLs (Chapter 4).
- **End User Application Engineering (EUAE)** – is the process that end users follow to derive applications from EUSPLs for their smart spaces (Chapter 4).

- **EU SPL Meta-model** – captures the underlying representation of end user product lines and end user applications in terms of meta-classes and relationships (Chapter 5).
- **Platform Independent Model** – is an end user application model that is independent of the platform (EUD environment e.g., Jigsaw/TeC) and the hardware/Operating System (OS) (Chapter 5).
- **Platform Specific Model** – is an end user application model that is specific to an EUD environment e.g., Jigsaw/TeC but independent of the hardware/OS platform (Chapter 5).
- **Platform Independent Product Line (PIPL) Meta-model** – captures the underlying representation of EU SPLs in terms of meta-classes and relationships independent of the platform (EUD environment. The meta-model contains representations of EU SPL features, feature dependencies, and the component architecture that realizes each feature. The meta-model is platform independent and contains meta-classes that are common to event-driven EUD environments for smart spaces (Chapter 5).
- **Platform Independent Product (PIP) Meta-model** – provides the underlying representation of end user applications in terms of meta-classes and relationships, which are derived from the PIPL meta-model (Chapter 5).
- **Platform Specific Product Line (PSPL) Meta-model -** similar to the PIPL meta-model but is extended with platform specific meta-classes (Chapter 5). The TeC

PSPL and Jigsaw PSPL are examples of PSPL meta-models for the TeC and Jigsaw EUD environments.

- **Platform Specific Product (PSP) Meta-model** – provides the underlying representation of end user application in terms of meta-classes and relationships, which are derived from the PSPL meta-model (Chapter 5). The TeC PSP and Jigsaw PSP are examples of application models derived for the TeC and Jigsaw PSPLs.

1.4 Problem Statement

End User Development (EUD) skills vary significantly among end users who want to design, develop and deploy software applications for their smart spaces. Typical end user development is opportunistic, requirements are usually unplanned and undocumented, applications are simplistic in nature, design is ad-hoc, reuse is limited, and software testing is typically haphazard, leading to many quality issues. On the other hand, technical end users with advanced EUD skills and domain expertise have the ability to create sophisticated software applications for smart spaces that are well designed and tested. The problem to be solved is (a) enable technical end users and domain experts to design and develop software applications for smart spaces that can be reused, and (b) enable less technical end users to adapt software applications developed by technical end users and domain experts to their spaces.

1.5 Thesis Statement

A systematic design approach and end user development environment can be created to specify, design, implement, test and deploy end user applications for smart spaces by using software product lines concepts. This will enable technical end users and domain experts to utilize the design method and development environment to create end user software product lines for smart spaces, from which end users will be able to derive applications for their spaces.

1.6 Research Focus and Goals

The focus of this research is to develop an End User Software Product Line (EU SPL) approach that extends existing EUD practices for smart spaces. The main concept of this approach is having End User SPL Designers (who are technical end users and domain experts) create EU SPLs for smart spaces and have end users derive their individual smart space applications from these SPLs. Incorporating SPL concepts in EUD for smart spaces makes it easier for novice end users to derive applications for their spaces without having to interface directly with devices, networks, programming logic, etc. End users only have to select and configure the EU SPL features needed for their space. Another benefit of this approach is that it promotes reuse. End user requirements are mapped to product line features that are realized by common, optional, and variant components available in smart spaces. Product line features and the corresponding component product line architecture can then be used to derive EU applications. Derived EU applications can then be deployed to different smart spaces, thereby avoiding end users having to create EU applications from scratch. Finally the proposed approach has the potential of improving software quality since testing will be an integral part of EU SPL process.

The goals of this research are to investigate: (a) a systematic approach for End User SPL designers to design and develop EU SPLs, (b) an EU SPL application derivation approach to enable end users to derive software applications for their spaces, (c) an EU SPL meta-model to capture the underlying representation of EU SPL and derived applications, (d) an EUD development environment that supports EU SPL development and application derivation, and (e) a testing approach and framework for testing EU SPLs and derived applications.

1.7 Research Approach

This research addresses the lack of a systematic approach and development environments to design and develop software applications for smart spaces that can be reused by end users. The research approach is described in detail in Chapter 3. Below is a summary of the research approach:

1. Define a comprehensive EU SPL process for (a) designing, developing and testing end user product lines for smart spaces and (b) deriving applications that can be that can be applied to different end user environments.
2. Define an EU SPL meta-model that extends existing meta-models of EUD environments for smart spaces to provide product line support. The meta-model captures the underlying representation of end user product lines and derived applications in terms of meta-classes and relationships that support different EUD platforms.

3. Develop a proof-of-concept End User Software Product Line Prototype (EUSPLP) development environment based on the EU SPL Process and meta-model. The environment supports the creation of end user product lines and application derivation for smart spaces.
4. Validate this research by applying the EU SPL process and proof-of-concept EUSPLP development environment to the Smart Home EU SPL case study. A testing framework is provided to test the artifacts of the EUSPLP development environment.
5. Deploy and execute TeC applications on the distributed TeC Android simulator (Shen, 2014).

1.8 Importance and Rationale of this Research

The growing adoption of ubiquitous computing and the Internet of Things (IoT) have contributed to the advancement of smart spaces. In the context of smart spaces, ubiquitous computing focuses on the interaction of end users with the environment whereas the IoT focuses on the interconnection of devices and services using the internet for connectivity. End user development environments for smart spaces aim to allow end users to take advantage of the device connectivity and end user friendly user interfaces to create applications for comfort, security, scheduling tasks, convenience through automation, energy management efficiency, health and assisted living (Rashidi and Cook, 2009). Even though EUD environments for smart spaces have made significant contributions for enabling end users to create applications for their spaces, they do not account for reuse and applications developed are platform (EUD environment) specific. For instance, TeC applications can only be deployed to a TeC smart space and Jigsaw applications can only be deployed to a Jigsaw smart space.

This research presents a systematic approach for adopting reuse in end user development for smart spaces by using software product line concepts. Using product line concepts for EUD, platform independent applications can be developed and then adapted for different EUD platforms. This research approach extends EUD environments for smart spaces with EU SPL support.

It should be noted that parts of the research described in this dissertation have been published in international conferences and workshops (Sousa, Tzeremes and Masri 2010; Sousa, Shen, Tzeremes and Hodum 2012; Tzeremes 2015; Tzeremes and Gomaa 2015; Tzeremes and Gomaa 2016a; Tzeremes and Gomaa 2016b).

1.9 Organization

This dissertation is organized as follows. Chapter 2 surveys related work that form the basis for this research. Chapter 3 details the research approach. Chapter 4 describes the end user software product line process, including end user product line development and application derivation. Chapter 5 describes the end user software product line meta-model that captures the underlying representation of end user product lines. Chapter 6 presents the EUSPLP development environment that supports product line development and application derivation. Chapter 7 describes the validation and testing approach of this research. Chapter 8 concludes the dissertation, outlines the contributions of this research, and suggests future work. Finally, Appendix-A presents the complete design of the Smart Home EU SPL case study used to validate this research.

2 RELATED WORK

2.1 Introduction

This chapter presents related research work that is the basis for the research described in this dissertation. Section 2.2 describes ubiquitous computing. Ubiquitous computing concepts and technologies are used to create smart spaces. Section 2.3 discusses the Internet of Things (IoT). IoT utilizes existing internet protocols for the communication of physical objects in smart spaces. Section 2.4 describes different development approaches for end users to create software applications for smart spaces. Section 2.5 provides an overview of software product lines. Section 2.6 discusses meta-modeling approaches for creating software applications. Section 2.7 discusses the extent that software product lines concepts have been applied to end user development. Section 2.8 discusses how this research compares to the related research. Finally section 2.9 provides a summary of this chapter.

2.2 Ubiquitous Computing

The term of ubiquitous, also known as pervasive, computing was first introduced by Mark Weiser in 1991 (Weiser, 1991). Weiser used the word ubiquitous to describe the concept of everywhere computing. Weiser believed that computing should be integrated seamlessly in the background, allowing people to employ it when needed without shifting their focus from their main tasks. The Olivetti Cambridge Research Labs active badge project (Want et al., 1992) that took place between 1990 till 1992 was an example of a ubiquitous computing environment at the time. The active badge project instrumented people working on a building with smart badges. In the building itself a number of sensors were deployed to read the badges. As a result, among others, doors were open to people that were carrying the provisioned badges, rooms were greeting people with their name, phones were transferred to a phone that the badge wearer was close and computers were adjusted to the badge wearer preferences.

Computing has evolved over the years from mainframe computers that were available in specific locations and supported multiple users, to personal computers where each user mainly interacted with one computer, to pervasive computing where technology is everywhere and supports multiple users. Satyanarayanan (Satyanarayanan, 2001) describes the progress of distributed and mobile systems research in relation to pervasive computing. Distributed systems are concerned with issues of remote communication protocols, fault tolerance, high availability, and remote information access and secure remote communication. Mobile computing builds on distributed systems and addresses research problems for mobile networking, mobile information access, adaptive applications, energy-aware systems and location sensitivity. Pervasive computing is the natural progression of both distributed and mobile systems. Some of the main research areas of pervasive technology are: (a) how to creating smart spaces that can react, send and receive information, (b) how technology can be hidden to the background and its available to users when needed, (c) how the environment can distinguish between different users that exist on the same space, and (d) how applications are deployed to smart spaces that have different technology support.

Pervasive environments, also known as smart spaces or smart environments, are composed from devices, networking, middleware and applications (Saha and Mukherjee, 2003). There is a variety of heterogeneous devices available in a pervasive environment, some examples are: mouse and keyboards, sensors and actuators embedded in the environment, cell phones, computers, custom devices developed for a specific purposes etc. Devices exchange data with other devices, software applications and the environment seamlessly. Networks provide communication protocols, auto configuration, quality of service, reliability, failover, lower bandwidths, lower transmission requirements, security and routing algorithms to support pervasive computing. Pervasive middleware can be thought as a distributed operating system. The middleware's responsibilities are to perform I/O operations, facilitate device communication, file system manipulation, application execution, error detection and resource allocation. The middleware essentially needs to present the heterogeneous environment as homogeneous to the applications. Pervasive applications are aware of their environment and are able to recover from device and sensor failures.

There have been several middleware architectures proposed for implementing pervasive environments (Saha and Mukherjee, 2003; Whitmore et al., 2015). Some of those initiatives are the ROS (Quigley et al., 2009), Aura (Sousa and Garlan, 2002), JCAF (Bardram, 2005), Smart Products (Mühlhäuser, 2008), UbiComp (Goumopoulos and Kameas, 2009), ACOCO (Fortino et al., 2013) projects. The Robot Operating System (ROS) is a middleware for creating smart spaces through the use of service robots. The ROS architecture consists of nodes, messages, topics and services. Nodes are processes that communicate with other nodes through messages. Messages can be send through topics for public-subscribe communication or services for point-to-point synchronous communication. The Aura project enables users to preserve continuity of their tasks across environments. The Aura architecture is composed of user tasks, the task manager, the context observer and the environment manager. A user task is composed from a collection of services used to accomplish the task. The task manager is responsible for managing the user tasks. The context observer based on the user context executes the appropriate user task on the target environment. The environment manager keeps track of all the resources in the environment. The Java Context-Awareness Framework (JCAF) is a context-awareness environment with an Application Programming Interface (API) that supports the creation of specific context-aware applications. The Smart Products initiative is based on creating autonomous objects that can communicate with another through peer-to-peer protocols to create smart spaces. The UbiComp middleware creates smart spaces through the composition of artefacts. Artifacts in UbiComp are heterogeneous tangible objects (sensors/actuators/devices) that can be combined together to achieve a task. UbiComp provides an editor for composing and instructing artifacts (Mavrommati et al., 2004) in smart spaces. ACOSO is agent-oriented event-driven architecture that reacts when changes in the environment occur. The middleware supports message passing and publish/subscribe mechanisms for agent communication.

There are several challenges for developing software for pervasive environments. These challenges can be grouped in the following areas: (a) application development, (b) user context, (c) data, (d) configuration, and (g) user interface (Henricksen et al., 2001;

Satyanarayanan, 2001). Application development challenges deal with the application structure, component design and implementation, interaction sequence between components, components states, application lifetime, concurrency, transactionality, device interaction, transmission requirements, workflow, application goals and security. User context gives the ability to applications to infer user activities based on spatio-temporal data (Pereira and Loyola, 2012). For example consider a smart meeting room. When the door is closed and they are people in the room the smart room can infer that there is a meeting in progress. Thus pervasive applications need to capture in their design: time, space, location, proximity to other devices, transition states, events of other applications, and operational history characteristics. Data challenges for creating pervasive applications deal with data storage, data dissemination, data security and data replication issues across environments. Finally configuration challenge research issues deal with how pervasive applications can be dynamically reconfigured based on the presence or absence of certain devices.

2.3 Internet of Things (IoT)

The Internet of Things (IoT) can be thought as a paradigm where every-day physical objects (sensors, devices, vehicles, buildings) can be equipped with identifying, sensing/actuation, storing, networking and processing capabilities that will allow them to communicate with one another and with other devices and services over the Internet to accomplish some objective (Whitmore et al., 2015). These objects are typically referred as smart objects. Smart objects are everyday objects that are equipped with hardware components such as a radio for communication, a CPU to process tasks, sensors/actuators to be conscious of the world in which they are situated and to control it at a given instance (Fortino and Trunfio, 2014). Smart objects can interact with other smart objects and people. The term machine-to-machine (M2M) is used to describe the direct communication protocols between smart objects (Yun et al., 2015). The idea of IoT was first introduced by Kevin Ashton while working on the Auto-ID Center at the Massachusetts Institute of Technology (MIT). Ashton originally used the term “Internet of Things” in 1999 in a presentation he made to Procter and Gamble to introduce RFID technology to the company’s supply chain management (Schneiderman, 2015).

Some of the drivers that contributed to the development of IoT are: (a) uniformity of access, (b) logistics, (c) energy efficiency, (d) physical security and safety, (e) industrial (f) medical, and (g) lifestyle (Kopetz, 2011). The internet provides uniform access to different types of computing devices, with different architectures and communication protocols. IoT takes advantage of the object interoperability over the internet and extends its function to smart objects. Logistics is another driver for IoT. For example retail products go through several steps in the supply chain before they make it to the market. The product is created from raw material, then is transferred to the manufacturing warehouse, then is transferred to the wholesaler warehouse and finally the product arrives at the retailer. This process involved a lot of manual communication between the different business parties in order to coordinate and keep track of the products. With the use of RFID tags on retail products, IoT provides much more meaningful insight to the entire process. Manufacturers, wholesalers and retailers have automated real time views of where products are in the supply chain. In a smart space environment, RFID technology is used to track smart objects throughout the environment. IoT has a major impact on energy efficiency. Smart objects collaborate with each other to ensure that smart environments optimize their energy consumption. Physical security and safety is another problem that IoT addresses. IoT objects work with each other to ensure that smart environments are safe to operate while providing access control to ensure that authorized resources are in the space. IoT plays a significant role in industrial manufacturing process. Smart objects help verifying the quality of manufactured products while monitoring the environment for failed machinery, machinery maintenance etc. There are several medical devices that monitor people's sugar levels, blood pressure, heart rate etc. Medical devices can be in the form of wearable technology or even internal to the patient's body. Extending medical devices with IoT gives the ability for medical devices to work together to diagnose patients and notify additional help if needed. IoT can have a significant effect on people's life styles. Smart objects can collaborate to adjust smart environments based on people's context. For instance, if a home resident goes to sleep, smart objects can notify the environment to adjust the energy and security objects in the environment.

Some of the current technical challenges current IoT research is investigating is (a) Internet Integration, (b) Smart object identification, (c) Near Field communication, and (d) Security (Kopetz, 2011). Adding internet connectivity to smart objects is a challenge. Internet communication requires power and not all smart objects in IoT have the same power capabilities. Furthermore as smart objects move potentially might lose internet connectivity. Current research is working to develop new communication protocols to minimize power consumption and address the offline challenges. For example the Internet

Engineering Task Force (IETF) has initiated a working group on IPv6 over Low Power Wireless Area Networks to find an energy-efficient solution for the integration of the IPv6 standard with the IEEE 802.15.4 wireless near field communication standard. Smart object identification is another challenge for IoT. According to forecasts from Cisco Systems, by 2020 more than 50 billion smart objects will be connected in the IoT (Fortino and Trunfio, 2014). Providing a common ontology to identify all these objects is a challenge. Smart object identification can be even more challenging when you have composite smart objects where a smart object is comprised of other smart objects. Near Field Communication (NFC) (ISO/IEC 18092, 2013) is a high performance communication interface and protocol for devices to communicate over a short range. One of the benefits of NFC is that requires less power compared to Bluetooth and other similar protocols because of the short range. One of the main challenges with NFC is security (Ji and Xia, 2016). Security overall is another challenge for IoT. Some of the main security challenges in IoT are: communication confidentiality and integrity, device availability, device authentication and access control, device computing limitations, heterogeneity in security protocols supported by devices and enforcing security policies in IoT environments (Mahmoud et al., 2015).

Ubiquitous computing and IoT research areas are the building blocks for creating smart spaces. Current ubiquitous computing research is focused on smart space applications and Human-Computer Interaction (HCI) whereas the current IoT research focus is to create the infrastructure and protocols for smart object communication (Ebling, 2016).

2.4 End User Development for Smart Spaces

Smart spaces are ordinary environments equipped with visual and audio sensing systems, pervasive devices, sensors, and networks that can perceive and react to people, sense ongoing human activities and respond to them (Singh et al., 2006). Examples of smart spaces are homes, offices, hospitals, farms equipped with technology to sense and react to environment changes. Applications for a smart home include energy efficiency, security, entertainment, and utility automation. End User Development (EUD) environments for smart spaces provide user interfaces for end users to create software applications and interconnect applications with devices deployed in a smart space. The purpose of EUD environments is to enable end users to develop software applications of their environments to suit their needs. For example, consider an economy laundry end user application that adjusts the operation of the washer and dryer during off peak hours when the power rates are discounted and pause operation during hours that the power rates peak. Current EUD approaches can be summarized in five general categories (Dimitris Kalofonos and Franklin Reynolds, 2006): (1) Programming languages, (2) Natural language processing, (3) Direct manipulation, (4) Programming by example, and (5) Visual programming.

2.4.1 Programming Languages

Programming languages have evolved over the years. Machine specific and assembly languages have given their place to higher level languages that are less demanding, easier to use and provide abstractions that make them almost hardware independent. An example is the JAVA programming language with its motif of “Write once, run anywhere” versus traditional languages that had to be compiled for different environments. In JAVA, programs are compiled one time into byte code. There are different byte code interpreters called Java Virtual Machines (JVMs) for different operating systems this make it easier for end users to run their programs in different platforms. Another advantage of higher level programming languages for end users is that Original Equipment Manufacturers (OEM) expose functionality of their products as programming APIs so end users can code to the API versus the internals of the devices. Programming languages, even though they have become easier to use over the years, still require a significant amount of training and computer science knowledge to be used by end users.

2.4.2 Natural Language Processing (NLP) EUD Environments

Natural Language Processing (NLP) approaches for end users are concerned with enabling end users to program their environments using every day human speaking languages. CAMP (Truong et al., 2004) and InterPlay (Messer et al., 2006) are some examples of end user programming frameworks in this category. CAMP uses a magnetic poetry metaphor for end users to program their environment. In CAMP, words are grouped in the following categories: who, what, where, when and general. End users express tasks by creating *"poems"* by combining words from different categories. An example of a "poem" in CAMP is "Capture Joe's dinner time conversations in the dining room." InterPlay provides the middleware for integrating consumer electronics in a smart home and allows end users to control and coordinate those devices using "pseudo sentences." A "pseudo sentence" is a simpler form of a grammatically correct full sentence. It consists of a verb, a subject and a target. The verb captures the activity that the user wants to perform. The subject captures the content that the user wants to use. The target implies the device that the user wants to perform a task. An example of a “pseudo sentence” is “Play big blue at the home theater.” Even though NLP is very promising, there are limitations to the extent that natural language processors can process complex end user input that deals with programming a smart space.

2.4.3 Direct Manipulation EUD Environments

Direct Manipulation approaches allow end users to directly manipulate objects. The Media Cubes (Blackwell and Hague, 2001) and FedNet (Kawsar et al., 2008) EUD environments provide tangible user interfaces for end users to program their spaces by direct manipulation. In Media Cubes, end users program their environment by manipulating a set of physical cubes. A cube consists of sensors, a processor and batteries. Cubes can be associated with devices and assume their functionality. For example a cube can be associated with a DVD player and assume its “*play”* and *“stop”* functionality. Cubes can sense and interact with other cubes by facing each other. End users can program their environments by grouping cubes together. For example consider two cubes where one of them represents a TV and the other one a DVD player. The DVD player cube, if it faces the TV cube, implies that the DVD player streams its output to that TV. In FedNet, devices and software applications come with RFID cards that embed remote URLs of where device and application binaries can be downloaded. The FedNet deployment tool is used to install, uninstall, start, stop, and associate devices and applications by scanning the corresponding RFID cards. Direct manipulation can be easier to understand, since end users manipulate directly physical objects versus having end users access physical objects through command line or image representations. Similar to NLP, direct manipulations approaches are hard to scale for complex end user applications.

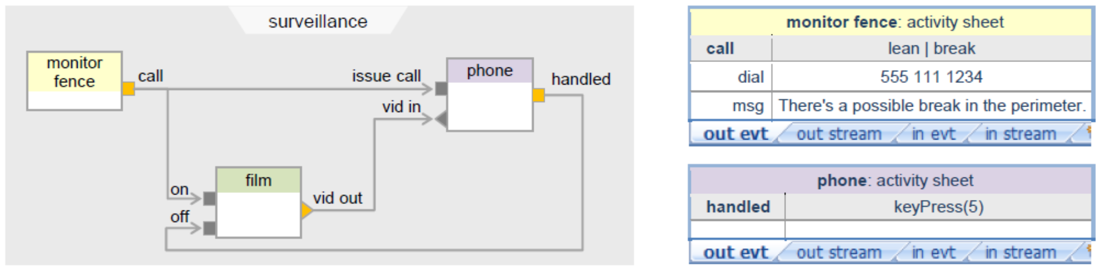
2.4.4 Programming by Example EUD Environments

Programming by Example (PBE) and Programming by Demonstration (PBD) approaches present the computer with examples of data that a program will process and having the system automatically deduce the current program from the examples (Myers, 1990a). In the context of smart spaces end users demonstrate to their environments of how to react when a certain conditions occur. The CAPpella (Dey et al., 2004) and Pervasive Interactive Programming (PiP) (Chin et al., 2010) end user frameworks are examples of this approach. CAPpella enables end users to create context-aware application through programming by example. It uses machine learning and user input to build software applications. End users train their environment with multiple examples that include a situation and a corresponding action. After CAPpella gets trained, it will perform the demonstrated action when the situation occurs. In PIP, the main concept is the *"Deconstructed Model"* where devices advertise services they support. Users can create virtual devices also called a MetaAppliance (MAp) that combine services provided by different devices. End users construct MAps using a graphical user interface and demonstrate behavior by physically interacting with the devices. Demonstrated behavior is stored in the form of rules. During run time, PIP uses a rules engine to evaluate the rules. Programming by example can be transparent to end users but the environment set up can be challenging; also, altering the system behavior will require re-training of the system model, which can be a complex task and time consuming.

2.4.5 Visual Programing EUD Environments

"Visual Programming" (VP) refers to any system that allows the user to specify a program in a two (or more) dimensional fashion (Myers, 1990b). Visual programming uses visual elements (such as icons, drawings or gestures) to create programs. Visual programming provides a natural way to program that helps users conceptualize what they develop.

In the smart space area, several EUD environments have been proposed that use visual programming. Some examples of visual programming frameworks are: Jigsaw (Humble et al., 2003), Puzzle (Danado and Paternò, 2012), GALLAG Strip (Lee et al., 2013), ICAP (Dey et al. 2006) and Team Computing (Sousa, 2010). Jigsaw and Puzzle provide a user interface for reconfiguring and reorganizing devices in a smart space. Devices appear as jigsaw pieces in the Jigsaw and Puzzle editors. End users can dynamically combine the Jigsaw pieces to create applications for their environments. GALLAG Strip enables users to create context aware applications through a sequence of screens in a mobile device. ICAP provides a visual rule building approach for end users to create context aware applications for their spaces. End users can specify rules from simple logic to personal, spatial and temporal relationships. Team designs define teams and are created in the TeC Editor. A team design is a collection of Activity sheets connected together. Activities sheets represent software components, devices, and humans. During the team deployment, activity sheets are mapped to players operating in the smart space. Additional logic, conditions and output events can be added to activity sheets for customization. Activity sheet outputs are connected to inputs of other activity sheets and get activated when their holding condition is true. Activity sheets are visually represented in the TeC Editor as big square boxes, inputs and output are smaller square boxes attached to the activity sheets. TeC also supports input and output streams. Streams are represented with small triangles attached to activity sheets. Figure 2.1 shows a “surveillance” team for a small farm to illustrate the user interface of TeC. The purpose of the team is to monitor the perimeter fence of a small farm. If an animal leans on, or breaks the fence, the owner of the farm gets contacted with a live video stream. The team has three activity sheets “monitor fence”, “film” and “phone.” The monitor fence activity sheet has an output event named “call” that gets triggered when the lean or break event occurs. The “call” output event is connected to the “on” input event of the “film” activity sheet, which turns on the camera and the “issue call” input event of the “phone” activity sheet that calls the farm owner. A video stream is sent from the “film” activity sheet to the phone of the farm owner. Video streams in TeC are represented with triangles. The owner of the farm can press key 5 on the phone, indicating that no further action is required from the system. This will trigger the “handled” output event of the “phone” to be true, which will send the “off” input event to “film” that results in the camera turning off.



monitor fence: activity sheet	
call	lean | break
dial	555 111 1234
msg	There's a possible break in the perimeter.

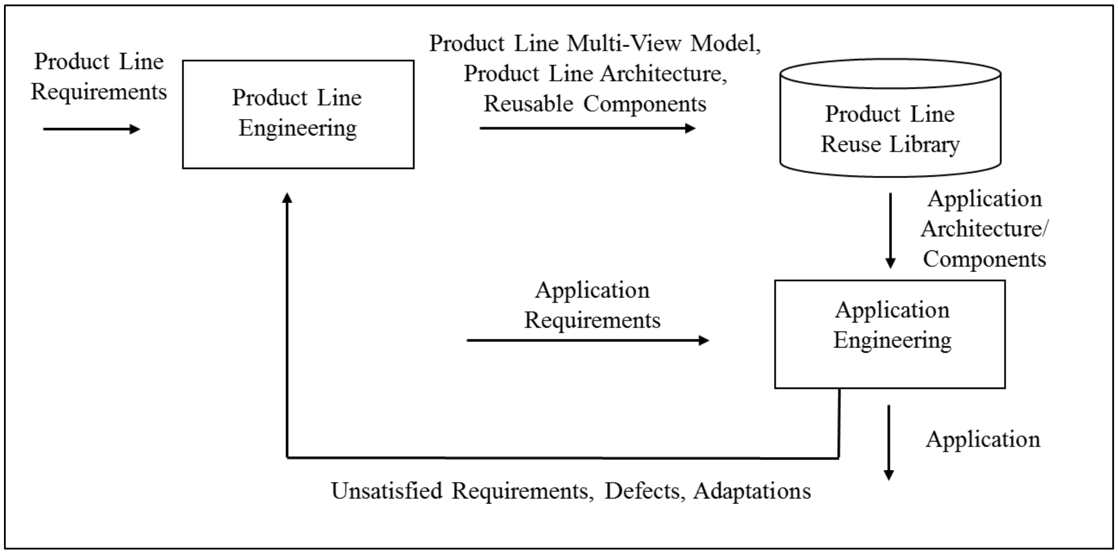
phone: activity sheet	
handled	keyPress(5)

Figure 2.1 TeC User Interface

2.5 Software Product Line Approaches

A Software Product Line (SPL) is a set of software intensive systems sharing a common, managed set of features that specify the specific needs of a particular market segment or mission and are developed from a common set of core assets in a prescribed way (Clements and Northrop, 2002). An SPL consists of a family of systems that share common and variable functionality. Common functionality utilizes reuse among products created from the product line. Variable functionality is what differentiates each of the products. Product lines are ubiquitous and can be found in almost all software applications that are offered in different editions. An example is the windows operating system. Windows is distributed in different editions, home, professional and ultimate. All versions share common features like mail, calendar and messaging but higher priced versions contain additional features like enhanced data protection and remote desktop connection features. Companies that adopted SPLs have experienced improvements in quality, maintainability, productivity and reduced costs (Kakola and Leitner, 2014).

The Software Product Line (SPL) engineering process is the process for creating a product line. Figure 2.2 shows a high level overview of the SPL engineering (Gomaa, 2005a) process. The SPL engineering process consists of two sub-processes: (a) product line engineering (a.k.a. domain engineering) process in which the product line is created and (b) the application engineering process in which software applications are derived. The software product line creation process involves software engineers working with product line stakeholders to define the product line requirements, the product line features. All artifacts created from the product line engineering process are stored in the product line



Product Line Requirements
Product Line Engineering
Product Line Multi-View Model, Product Line Architecture, Reusable Components
Product Line Reuse Library
Application Architecture/ Components
Application Requirements
Application Engineering
Application
Unsatisfied Requirements, Defects, Adaptations

Figure 2.2 Process Model for Software Product Lines

reuse library. The application engineering process is the process for generating applications from the product line features, architecture, and components. A product line feature is realized by one or more components and satisfies a specific user requirement or set of requirements.

Features are categorized as common, optional, alternative and parameterized. Common features exist in all products of the product line. Optional features exist in only certain products of the product line. Alternative features are features that can be selected in place of each other, one of which can be a default feature. Finally, parameterized features are SPL configuration parameters that are set during application derivation or at run time initialization. In application engineering, product engineers specify the feature requirements of the final product. The product line creation process maps the feature requirements to the components that implement them and assembles the final product. Product engineers communicate additional requirements and errors back to the SPL engineers to include them in future releases. Some of the most common SPL approaches are: The Software and systems engineering - Reference model for product line engineering and management (ISO/IEC 26550:2016, 2016), PLUS (Gomaa, 2005a), CVL (Haugen et al., 2013), COPA(America et al., 2000), FAST(Harsu, 2002), and KobrA (Atkinson and Muthig, 2002).

2.5.1 Product line engineering and management (ISO/IEK 26550:2016)

The international standard for Software and systems engineering - Reference model for product line engineering and management (ISO/IEC 26550:2016, 2016), aims to create a common vocabulary and standard process for product line creation. The standard covers domain and application engineering aspects for creating the product line. Domain engineering covers product line scoping, domain requirements engineering, domain design, domain realization and domain validation and verification. During domain engineering, organizational management works with technical management to perform product line scoping. Product line scoping involves identifying market groups, product categories, common and variable features, functional domains for envisioned features that provide sufficient reuse, reusable assets for creating products and cost benefit analysis for each domain asset. After the product line is scoped domain requirements engineering is performed that identifies the product line stakeholders and captures detailed requirements. Domain design is used to perform commonality and variability analysis, feature modeling and define the domain architecture. Domain realization is responsible for component design and implementation. Domain validation and verification provide the quality assurance aspect to the product line. All domain assets defined during domain engineering are stored on the domain asset repository. The application engineering process in the ISO involves application requirements engineering, application design, application realization and application verification and validation. Application requirements engineering develops application-specific requirements reusing common and variable requirements defined during domain requirements engineering. Application design derives the application architecture from the domain architecture in order to meet application requirements. Application realization implements product line members by drawing upon the application requirements and architecture; reusing and configuring domain components and interfaces. Application verification and validation ensures that the right member product and the right application assets have been modeled, specified, designed, built, verified, and validated. All artifacts created by the application engineering process are stored in the application asset repository.

2.5.2 Product Line UML-Based Software Engineering

Product Line UML-Based Software Engineering (PLUS) is defined as a design method for software product lines that describes how to conduct requirements, analysis, and design modeling for software product lines in UML (Gomaa, 2005a). PLUS requirements phase identifies the product line use cases and tags them as kernel optional and variant. Feature analysis identifies the product line features and maps them to the use cases. During the analysis phase, PLUS examines the problem domain and develops the system context diagram, collaboration/sequence diagrams and state diagrams. The analysis phase concludes with feature/class dependency diagrams and tables that show the classes that implement features. In the design phase, PLUS examines the solution domain and develops the product line architecture and structures the system into subsystems and components. The design phase ends with defining the communication interface of each component. In the component implementation phase software engineers select a subset of the designed functionality for development. The product line testing phase performs integration testing among the components developed on the increment with the existing components of the product line and functional testing that test the functionality of the increment. All artifacts generated by PLUS are stored in the software product line repository.

2.5.3 Common Variability Language

The Common Variability Language (CVL) (Haugen et al., 2013) is used to add variability to MDA models. In particular CVL, is a Domain Specific Language (DSL) for modeling variability in models that are based on Meta Object Facility (MOF) standard defined by the Object Management Group (OMG) (Reinhartz-Berger et al., 2014). CVL operates on three models: the base model, the variability model and the resolution model. The base model is a domain model for a particular system. For example a base model can describe a particular train control system deployed to a train station (Svendsen et al. 2010). The variability model describes variations on the system. In the train control system example there might be train stations with different number of tracks, different directions etc. The train control variability model needs to capture different train control systems that can be deployed to train stations with different configurations. The resolution model captures a set of options on the variability model. In the train example a resolution model can be the train control system supporting a train station with two tracks, one track going east and another track going west. To create a new system CVL takes as input the three models and generates new resolved models. Existing DSL tools can operate on the resolved models that can transform them to runnable software.

2.5.4 Component-Oriented Platform Architecting

The Component-Oriented Platform Architecting (COPA) method is a component based product line methodology that provides a set of (component-based) subsystems and interfaces (with their associated processes, documentation and tools) from which a stream of derivative and composite products (families) can be developed and produced according to a domain specific architecture or product family architecture (America et al., 2000). COPA uses the Business-Architecture-Process-Organization (BAPO) model to cover multiple aspects of the product line lifecycle like business drivers, architecture, processes and organization concerns. BAPO starts by identifying the business needs for the product line which might be an improvement of an existing product line or the need for a new one. After the business need gets identified, BAPO defines the product line architecture. There the domain of the product line is defined. Systems and components are defined and structured to fit the product line architecture. The process phase of BAPO creates the architecture previously defined while identifying component dependency, commonality and variability. The organization aspect of BAPO covers organizational support for the product line. It ensures that the product line matches the organization's business needs, it provides management support and defines processes for product line maintainability and evolution.

2.5.5 Family-Oriented Abstraction, Specification and Translation

Family-Oriented Abstraction, Specification and Translation (FAST) is a product line methodology that abstracts the commonality of target software products and creates a common platform for the creation of a family of software systems. Variability is addressed through parameterization or conditional compilation (Harsu, 2002). The methodology has two main phases: (1) Domain qualification (2) Domain Engineering and Application engineering. During domain qualification product families are identified and justification is made for their creation. Domain engineering covers analysis and implementation of the domain. During domain analysis product line functionality is abstracted and a common platform for product line family creation is designed. Domain implementation creates and implements the common platform. Application engineering uses the platform created in domain engineering to create product line family members.

Feature-Oriented Reuse Method for product line software engineering (FORM) is a software product line methodology that supports architecture design and object oriented component development while incorporating a design and analysis marketing perspective (Kang et al., 2002). The FORM process has two sub processes: (1) Asset development and (2) Product development process. The asset development process analyses the commonality and variability of the product line and develops a component based architecture based on the analysis performed. The product development process gathers product requirements, selects features, adopts an architecture, adapts components and generates code for the software product.

2.5.6 KobrA

KobrA (Atkinson and Muthig, 2002) is a component based approach for software product line development. Software elements are created individually and get synthesized in different ways to create different members of the product line. KobrA has two main phases: Framework Engineering and Application Engineering. Framework engineering analyses the commonality and variability of the product line and creates generic framework that represents all variations of the product line while including information about the common and variant features. Application engineering is responsible for instantiating the generic framework and create different product variants based on customer specifications.

2.6 Meta-modeling

A model of a system is a description or specification of that system and its environment for some certain purpose (OMG, 2003). A meta-model is a model that describes a model (Kleppe, 2008). For example in EUD, end user applications created for smart spaces can be thought as application models. Examples of application models are: security, energy efficiency, and economics applications. Internally EUD environments have developed a meta-model to describe the structure of these applications. Meta-modeling is the process for creating a meta-model for a specific domain.

Many software specification and design methods advocate a modeling approach in which, the developed system is represented by means of multiple-view meta-models. Gomaa and Shin (Gomaa and Shin, 2008) proposed a multiple-view meta-modeling approach for software product lines. Abu-Matar and Gomaa (Abu-Matar and Gomaa, 2012) proposed a feature-based variability multi-view meta-modeling approach for service oriented architectures. Model Driven Development (MDD) abstracts software development life cycle by shifting its focus from code to models, metamodels and model transformations. Blanc et al. (Blanc et al., 2005) propose extending MDD approaches with meta-modeling approach for reuse. The UML4SPM (Bendraou et al., 2005) work proposes a new UML based metamodel for software process modelling that support executable models. Model Driven Architecture (MDA) (OMG, 2003) uses meta-modeling to define the underlying representation of platform independent and platform specific architectures.

2.6.1 Model Driven Architecture

Model Driven Architecture (MDA) is a software development framework based on automatic transformations of models (Debnath et al., 2008). The Object Management Group (OMG) promotes model-driven architecture which UML models of the software architecture are developed prior to implementation (Gomaa, 2016). The Unified Modeling Language (UML) is a modeling language used to describe the results of object-oriented analysis and design developed by different methodologies e.g, COMET (Gomaa, 2000), PLUS (Gomaa, 2005a).

MDA separates business and application logic from underlying platform technology, distinguishing the following models: Computation Independent Model (CIM), Platform Independent Model (PIM), Platform Specific Model (PSM) and code. The most common representation of these models is UML. However, other languages can be used if they are based on Meta Object Facility (MOF) meta-model (Abu-Matar, Mohammad Ahmad, 2011). The CIM is used to analyze the problem domain. The CIM captures business processes, system requirements and functions independent of any system implementation. The PIM is used for creating an abstract version of the system independent of technology implementation (Singh and Sood, 2009). The PIM captures different aspects of the system, identifies the system entities and operations needed to satisfy the requirements described in CIM. The PSM augments the PIM with specific platform details and implementation information such as .NET, J2EE, Webservices, etc. To create an application for multiple platforms, a given PIM will have to be mapped to multiple PSMs. The PSM model is used to generate code and deploy the application to the environment.

2.7 Software Product Lines for End Users

Current research on utilizing product lines for end users includes Monaco (Prähofer et al., 2008), SimPL (Malaer and Lampe, 2008), MobiLine (Marinho et al., 2013) and Perez et al. (Perez and Valderas, 2009). Monaco proposes a software development framework for building end user programming environments. The problem that Monaco solves is that Original Equipment Manufacturers (OEMs) were spending significant effort to create end user programming environments for similar products. Monaco abstracts end user programming domain specific functionality and proposes a development framework for the creation of end user programming environments that OEMs can reuse.

SimPL (Malaer and Lampe, 2008) provides product line tools for domain engineers to set up an environment for end users to instantiate product line members. Domain engineers in SimPL define a Domain Specific Modeling Language (DSML) that describes a specific domain. The SimPL editor translates the DSML created by domain engineers to a set of graphical elements that can be grouped together by end users to create software applications. The SimPL approach is closer to the ones examined in the end user programming frameworks section. SimPL does not explicitly model product line variability but it defers it to the DSML creators. DSML dictates which elements can be connected together in the SimPL editor.

MobiLine (Marinho et al., 2013) developed a software product line for the domain of mobile and context-aware applications. MobiLine identified multiple individual mobile applications (games, mobile commerce, mobile guides, mobile learning) that involve mobile devices and user context. MobiLine used existing applications as requirements elicitation and created a domain model for mobile and context aware functionality. The mobile and context aware domain model is combined with specific application domain models (eg., mobile visit guides, financial applications, health care applications) to create mobile and context-aware applications. The benefit of this approach is that mobile and context aware functionality does not have to be replicated across different application domain models.

Perez et al. utilize variability engineering for professional engineers to cooperate with end users to create configurable applications for their smart spaces (Pérez et al., 2009; Pérez and Valderas, 2009). Variability modeling is used as a requirements gathering tool between professional engineers and end users. Based on the variability model, engineers create environments that end users can reconfigure using existing end user programming frameworks like Jigsaw.

2.8 Comparison with Existing Approaches

The following sections discuss how this research relates to the current research in the areas of: (a) EUD environments for smart spaces, (b) Software product lines, (c) Meta-models, and (d) SPL approaches for end users and smart spaces.

2.8.1 EUD Environments for Smart Spaces

The functionality provided by EUD environments for smart spaces can be grouped in two general areas: Smart space configuration and context aware environments. Smart space configuration environments enable end users to control and combine functionality of devices. Jigsaw (Humble et al., 2003), and Puzzle (Danado and Paternò, 2012) are some examples. Context aware environments create rules based on user context (activity, location, identity, time) and device functions. PIP (Chin et al., 2010), FedNet (Kawsar et al., 2008), iCAP (Dey et al., 2006), GALLAG Strip (Lee et al., 2013), and TeC (Sousa, 2010) are some examples. Current EUD environments for smart spaces do not account for reuse. End user applications are created for specific environments and are not portable to other environments. For instance an end user application for TeC is only applicable for the TeC EUD environment and cannot be reused for Jigsaw.

The research described in this dissertation extends existing EUD environments for smart spaces with product line support. Thus, this research extends visual languages used by EUD environments and application models to create product line features. End users can select features from the product line and derive applications for their smart spaces.

2.8.2 Software Product Lines

Software product line methods such as ISO ISO/IEC 26550 (ISO/IEC 26550:2016, 2016), PLUS (Gomaa, 2005b), CVL (Haugen et al., 2013), COPA(America et al., 2000), FAST (Harsu, 2002), and KobrA (Atkinson and Muthig, 2002) address the problem of modeling variability in product lines and provide processes to design SPLs and derive applications from them.

The research described in this dissertation extended current software product line approaches to provide support for EUD development and smart spaces. In particular this research defined a lightweight product line approach for technical end users and domain experts to design and develop EU SPLs that can be used by end users to derive applications for different EUD environments. Furthermore this research extended the design method and modeling techniques defined in PLUS to capture feature and component platform dependencies. The product line design artifacts of the PLUS method were also extended to capture the platform and component / connector architecture information available in smart spaces.

2.8.3 SPL Approaches for End Users and Smart Spaces

Current research on utilizing product lines concepts for end users and smart spaces includes SimPL (Malaer and Lampe, 2008), MobiLine (Marinho et al., 2013) and Perez et al. (Perez and Valderas, 2009). As with this research, SimPL uses components, connectors and triggers to create application logic. In SimPL domain engineers are responsible for providing implementations of the components that realize each feature in the product line.

End users use the DSML to select different components applicable for each feature and connect them together to form application logic.

This dissertation research describes a visual language for technical end users and domain experts to create product lines. The implementation of the components is provided by the EUD environments. In addition, features in this research are realized by components connected together versus having features realized by a set of components that the end user is allowed to connect as proposed in SimPL.

MobiLine extends SPL concepts to reuse mobile and context-aware functionality for different application domains. The SPL process followed by MobiLine is complex and requires the involvement of product line engineers, application engineers and domain experts from different domains to create product lines and derive applications. This dissertation research builds on extending SPL methods to address end user development for smart spaces.

Perez et al. utilize variability engineering for professional engineers to cooperate with end users to capture end user requirements for smart spaces (Perez and Valderas, 2009). Perez provides examples using Jigsaw and programming by demonstration. This dissertation research extends Perez's work beyond requirements elicitation for product lines. This dissertation research utilizes visual languages and application models of EUD environments to create product lines for smart spaces.

2.8.4 Meta-modeling

MDA separates business and application logic from underlying platform technology. This dissertation research is influenced by the CIM, PIM and PSM concepts but was expanded to end user development for smart spaces. Thus, this research investigates the creation of platform independent and platform specific meta-models to capture end user product lines that can be used to derive applications for different EUD environments for smart spaces.

2.9 Summary

This chapter has described related work to this research. The ubiquitous computing section and the Internet of Things sections described how the two concepts can be used to create smart spaces, in particular the components of smart spaces, different smart space initiatives, and challenges for creating software applications for smart spaces. The end user development for smart spaces section covered the evolution of systems that enable end users to develop software for their spaces. The software product line approach section described the concept of software product lines and discussed different approaches for creating software product lines. The software product lines for end users section described current initiatives that show how software product line concepts can be adapted for end users. Finally, this chapter described how the research described in this dissertation compares to existing research on EUD environments, SPL methods, meta-modeling approaches and current SPL approaches for end users.

3 RESEARCH APPROACH

3.1 Introduction

This chapter describes the research approach followed in this dissertation. In summary, this research defines an EU SPL process that supports end user product line development and application derivation for smart spaces. To support the EU SPL process, an EU SPL meta-model is defined to capture the EUSPL meta-classes and relationships. The EUSPLP development environment was created to enable the development of EU SPLs and application derivation. Finally, a Smart Home EU SPL case study was used to validate this research.

The chapter is organized as follows: Section 3.2 describes the overall research approach. Section 3.3 describes the background and artifacts of the EU SPL process defined in this research. Section 3.4 provides an overview of the EU SPL meta-model defined for capturing the underlying representation of end user product lines and end user applications. Section 3.5 describes the proof-of-concept EUSPLP environment created in this research. Section 3.6 describes the validation approach for this research, including the testing approach and framework. Section 3.7 describes the rationale of extending existing EUD approaches for smart spaces with EU SPLs. Finally, section 3.8 summarizes this chapter.

3.2 Research Approach

This research addresses the End User Software Product Line (EU SPL) process and supporting development environment, which are used by technical end users and domain experts to develop end user product lines for smart spaces. End users utilize the same process and environment to derive applications from the EU SPL. The EU SPL process provides: (a) technical end users and domain experts with a systematic approach to develop end user product lines, (b) end users with an approach to reuse end user applications, and (c) testing support to improve the quality of end user applications.

The EU SPL process consists of the (a) End User Software Product Line Engineering (EUPLE), and (b) End User Application Engineering (EUAE) sub-processes. EUPLE defines the process steps and process artifacts to create end user product lines. EUAE defines the process steps for deriving applications from the product line. The EU SPL process is tailored to address end user requirements for smart spaces. The Smart Home EU SPL case study was created to verify each step of the EU SPL process.

The research defined a meta-model that is utilized to capture the underlying representation of EU SPLs and derived EU applications in terms of entities meta-classes and relationships. The EU SPL meta-model was derived from: (a) examining the end user environments for smart spaces described on Chapter 2, and (b) the Smart Home EU SPL case study. As part of this research the meta-models of the TeC and Jigsaw EUD environments were designed. The meta-models contain meta-classes for representing EU applications in the TeC and Jigsaw EUD environments. The TeC and Jigsaw meta-models were used to form the EU SPL meta-model. In particular, the common meta-classes of the TeC and Jigsaw meta-models were extracted to create platform independent meta-models (PIPL/PIP). The PIPL/PIP meta-models provide the underlying representation for end user product lines and derived applications that can be applied to any event driven EUD environment for smart spaces. The meta-model was extended to create platform specific meta-models (PSPL/PSP) that support end user product lines for specific EUD environments for smart spaces.

The EUSPLP development environment was created to enable end users to design product lines and derive applications for smart spaces. The environment was developed based on the EU SPL process and meta-model. As part of the user interface for the EUSPLP environment, a visual language was designed to enable: (a) technical end users and domain experts to design EU SPLs, and (b) end users to derive applications.

To validate this research, a Smart Home case study (a) was created using the EU SPL process, (b) was implemented using the EUSPLP environment, (c) the TeC PSPL was tested using the EU SPL Testing process, (d) derived applications from the EUSPL were tested using the EU Application Testing process, and (e) derived applications were deployed to the TeC Android simulator and tested using the EU Application Deployment Testing process.

3.3 EU SPL Process for Smart Spaces

The EU SPL process described in this research provides a systematic approach for creating end user software product lines and deriving applications for smart spaces. The process is based on the PLUS method (Gomaa 2005) which was extended in this research to include the design of end user product lines for smart spaces. The EU SPL method consists of the End User Product Line Engineering (EUPLE) process in which the end user software product line is designed and developed, and (b) the End User Application Engineering (EUAE) process in which software applications are derived.

The following artifacts are created during the requirements, analysis and design phases of the EUPLE process:

- Use Case Modeling – Use cases are defined to capture end user requirements.
- Feature Modeling – The EU SPL feature model is created.
- Static Modeling – During static modeling, the components needed to realize each feature are defined.
- Dynamic Modeling – Sequence diagrams are defined for each feature defined in the EU SPL.
- Inter-feature Component Communication Modeling – Captures inter-feature component communication.
- Component Modeling – component diagrams and component input/output tables are created for each feature to capture the component communication interfaces.
- Platform Specific Feature/Component Modeling – Captures platform specific component information that applies to platform specific features.
- Feature-based Integration Test Cases – Capture component outputs / inputs / triggering conditions and expected test results for testing the component architecture of individual features and feature combinations.

The following artifacts are created during the requirements, application derivation and testing phases of the EUAE process:

- Application Derivation Feature Modeling – The subset of the feature model that contains the selected features for the application.

- End User Application Architecture Modeling – The derived application component architecture for the target in the end user environment.
- Feature-based Integration Test Cases – The Feature-based Integration Test Cases that apply to the features that comprise the derived application.

From the above artifacts, certain artifacts are designed differently in this research to cover the unique issues related to end user development for smart spaces: feature modeling, static modeling, dynamic modeling, component modeling, platform dependent feature/component modeling, test case format, application derivation feature model and end user application architecture modeling. The Smart Home EU SPL case study was created using the EU SPL process.

3.4 EU SPL Meta-model for Smart Spaces

The EU SPL meta-model designed in this research provides a meta-model for representing end user product lines and derived applications for different EUD environments for smart spaces. The EU SPL meta-model was used to support the EU SPL process. The EU SPL meta-model is composed of the following meta-models:

- Platform Independent Product Line (PIPL) meta-model - provides the underlying representation of EU SPLs independent of any platform (EUD environment).
- Platform Independent Product (PIP) meta-model - provides the underlying representation of end user applications derived from the PIPL meta-model.
- Platform Specific Product Line (PSPL) meta-model - provides the underlying representation of EU SPLs for specific EUD environments.

- Platform Specific Product (PSP) meta-model - provides the underlying representation of end user applications derived from the PSPL meta-model.

The EU SPL meta-model also defines the relationships between the meta-classes representing the different models.

3.5 Proof-of-concept EUSPLP Development Environment

A proof-of-concept End User Software Product Line Prototype (EUSPLP) development environment was created to support this research. The EUSPLP environment provides the functionality to (a) enable EU SPL designers to create end user product lines, and (b) enable end users to derive and deploy applications for their smart spaces. Some of the unique characteristics of the design and development of the EUSPLP environment are that: (a) utilizes end user friendly interfaces for product line creation and application derivation, (b) integrates with the TeC end user environment for application deployment, (c) supports additional end user environments by developing different EUSPLP adaptors, (d) remotely accessible to EU SPL designers and end users through the use of web browsers, and (e) utilizes REST services and JSON format to communicate with remote TeC end user environments. Below is an overview of each subsystem within EUSPLP:

- EU SPL Development Subsystem
  - Allows EU SPL designers to visually create/edit the EU SPL feature model tree and define feature and feature group relationships.
  - Allows EU SPL designers to visually create/edit component architectures and associate them with product line features. A drag and drop interface was created for EU SPL designers to create component architectures.

- Allows EU SPL designers to configure different parameters relating to the feature component architecture.
- Creates PIPL JSON representation of the EU SPL to store the product line visual representation.
- Creates TeC PSPL JSON representation of the EU SPL to store the product line specification for TeC used for application derivation.

- Application Derivation Subsystem
  - Allows end users to visually select different features from the EU SPL.
  - Allows end users to visually configure the component parameters of the selected features.
  - Allows end users to visually derive applications for their spaces. The environment derives a TeC PSP model from the TeC PSPL model based on the end user selections. The TeC PSP model is stored in JSON.
- Application Distributor Subsystem
  - Provides a REST service for distributing the TeC PSP to TeC EUSPLP Adaptors deployed in different TeC environments.
- TeC EUSPLP Adaptor Subsystem
  - Retrieves the TeC PSP specification and stores it in the TeC environment. The end user interacts with the TeC environment to complete the deployment of the application.

3.6 Validation

The validation of this research was performed through (a) the Smart Home EU SPL case study, (b) the EUSPLP environment, (c) the EU SPL Testing framework and (d) the deployment of derived applications to the TeC Android simulator.

The Smart Home EU SPL case study described in Appendix-A was created following the EU SPL method described in Chapter 4. The case study includes features from the domains of home automation, home security, home notifications, home maintenance, resident comfort and energy conservation. Both End User Product Line Engineering (EUPLE) and End User Application Engineering (EUAE) were applied to the case study. EUPLE was applied to develop the EU SPL. EUAE was applied to derive end user applications for two end user platforms, TeC and Jigsaw.

To validate the EUSPLP environment, the Smart Home EU SPL case study was designed and implemented using the prototype environment. In addition, several applications were derived from the Smart Home SPL implementation using the application derivation interface of the prototype.

The EUSPLP environment was also used to validate the EU SPL meta-model and meta-model mappings described in Chapter 5. Thus, the PIPL, TeC PSPL and TeC PSP meta-models defined in the EUSPLP environment were derived from the EU SPL meta-model. The meta-class mappings required by the application derivation process for the conversion of a TeC PSPL model to a TeC PSP model, were derived from the EU SPL meta-model mappings.

The EU SPL Testing, EU Application Testing and EU Application Deployment Testing processes of the EU SPL Testing approach, were used to test TeC SPLs and TeC applications developed using the EUSPLP environment. In particular, the EU SPL Testing process was used to validate that the TeC SPLs developed using the EUSPLP environment: (a) follow the EU SPL consistency rules, and (b) each feature / component architecture executes as it was designed in the EUSPLP environment. The EU Application Testing process was used to validate that the TeC applications derived using the EUSPLP environment: (a) are composed of a valid feature combination, and (b) the application component architecture executes correctly. The EU Application Deployment Testing process was used to test that TeC applications were deployed successfully to the smart space. The TeC Android Simulator created by Shen (Shen, 2014) was used to validate that derived applications from the EUSPLP environment were deployed successfully to a distributed Android platform. Thus, different experimental end user applications including an end user application derived from the Smart Home EU SPL case study were deployed to the TeC Android Simulator.

3.7 Rationale for Extending EUD Approaches with EU SPLs

There are several issues in developing end user applications for smart spaces using current EUD approaches that can be addressed by applying the End User Software Product Line (EU SPL) approach described in this research. Table 3.1 provides a summary of the EUD issues, and compares how each of the issue is addressed utilizing current EUD approaches for smart spaces versus using EU SPLs.

Table 3.1 Benefits of Extending EUD Approaches for Smart Spaces with EU SPLs

EU Development Issue	**Current EUD Approaches for Smart Spaces**	**Utilizing the EU SPL Approach**
EU Application Development Cost	Costs depend on the ability of each end user to develop EUD applications versus outsourcing the development to technical end users and/or domain experts. Higher application development cost, since there is no reuse and applications from the same domain have to be re-developed for different EUD environments and smart spaces.	Initial cost to design and develop the EU SPL. Low EU application development cost after the EU SPL has been created, since applications can be derived from the EU SPL to satisfy end user requirements for individual smart spaces.
EU Technical Background	Does not address variability in end users technical backgrounds and EUD capabilities. EUD environments provide a common user interface for all end users to design and develop applications for smart spaces. Does not address non-technical end users issues in developing EU applications.	The EU SPL development environment provides a different user interface and workflows for technical end users and/or domain experts to create EU SPLs, whereas it provides a simpler user interface for end users to derive applications.
Software Reuse	Software reuse is limited. End users do not develop applications with a goal to reuse and even if they do, current EUD environments do not provide mechanisms for application reuse. End user applications have to be re-developed for different EUD environments and smart spaces.	EU SPLs promote reuse by designing and developing product line features that are realized by common, optional, and variant components and connectors. End user applications are derived by selecting EU SPL features for different EUD environments and smart spaces.
EU Application Requirements	Requirements are usually unplanned and undocumented. End user requirements are too personalized to create applications that can be reused by other end users for different EUD environments and smart spaces.	Requirements are collected and documented through the EU SPL requirements elicitation process. Requirements are used to define the EU SPL features, feature groups and feature dependencies. Features are selected by end users to tailor the EU application to their needs.

	End users focus on implementation without taking the time to document requirements.	
EU Software Design	Software design of EU applications is adhoc. Non-technical end users are not familiar with software design methods.	Software design is an integral part of the EU SPL process. Technical end users and/or domain experts design platform independent and platform specific product line features, feature dependencies, feature groups and reusable components that support different EUD environments and smart spaces. Non-technical end users can utilize software design by selecting features and reusable components to derive applications for their smart spaces.
EU Software Development	EUD is opportunistic. Difficult for non-technical end users to develop applications utilizing existing EUD environments for smart spaces. EUD difficulty increases with the complexity of the EU application.	Software development is performed by technical end users and/or domain experts. End users can derive complex applications for their spaces by selecting and configuring EU SPL features
EU Application Complexity	Applications are simplistic in nature. Limited user interfaces for developing complex applications. Variability in end user application sophistication based on the end user technical background.	Application functionalities are organized as EU SPL features that are realized by common, optional, and variant components and connectors. During application derivation, selected features and their corresponding component/connector architecture can be used to compose a highly complex and configurable application.
EU Application Testing	Software testing is typically haphazard, leading to quality issues in applications developed by end users.	The EU SPL process provides a systematic testing approach that can be used to test EU SPLs, derived applications, and end user application deployment in smart spaces

3.8 Summary

This chapter provides a summary of the research approach followed in this dissertation. The research approach include (a) definition of the EU SPL process, (b) definition of the EU SPL meta-model, (c) design and development of the EUSPLP environment, and (d) a testing process to validate the artifacts of the EUSPLP environment. The Smart Home EU SPL case study was used to validate the different parts of this research.

4 EU SPL PROCESS FOR SMART SPACES

4.1 Introduction

The Software Product Line (SPL) engineering process provides a systematic approach for developing software product lines. The SPL engineering process consists of two sub-processes: (a) the product line engineering (a.k.a. domain engineering) process in which the product line is developed, and (b) the application engineering process in which software applications are derived from the product line. The product line engineering process involves software engineers defining the product line features and developing the product line architecture to support them. The application engineering process involves application engineers deriving applications from the product line features and SPL architecture. The SPL engineering process involves requirements gathering, commonality /variability analysis, feature modeling, variable architecture definition, component design and implementation.

One of the issues with End User Development (EUD) for smart spaces is that there is variability in the EUD environments and the components / devices supported by different smart spaces. The SPL engineering process could be used for EUD but the problem is that the SPL process targets professional engineers and can be complex for end users and domain experts to use. This chapter presents an End User (EU) SPL process for developing end user applications for smart spaces. The EU SPL process was defined as part of this research and extends conventional SPL approaches to support the unique requirements of EUD development for smart spaces. The EU SPL process provides a lightweight product line approach for technical end users and domain experts to design and develop EU SPLs that can be used by end users to derive applications for different EUD environments. As part of the EU SPL process, conventional SPL design artifacts were extended to capture information about platforms and component / connector architectures in smart spaces. The Smart Home EU SPL case study was designed and developed using the EU SPL process described in this chapter.

This chapter is organized as follows. Section 4.2 provides an overview of the EU SPL process. Section 4.3 describes the end user product line engineering process including: end user requirements elicitation, analysis modeling, design modeling, implementation and testing. Section 4.4 describes the end user application engineering process including: end user application requirements, application derivation, testing and application deployment. Section 4.5 describes evolution of end user software product lines. Finally, section 4.6 summarizes this chapter.

4.2 End User SPL Process

End user development for smart spaces has several unique requirements that differentiate it from traditional application development. Some of the differences are that it targets end users to develop software and that applications can be highly personalized with different smart space requirements (Dautriche et al., 2013). The End User SPL process provides a systematic approach for EU SPL designers who are technical end users and domain experts to design and develop end user software product lines for smart spaces that end users can use to derive applications for their smart spaces. Figure 4.1 shows the End User Software Product Line (EU SPL) process. Similar to the conventional SPL

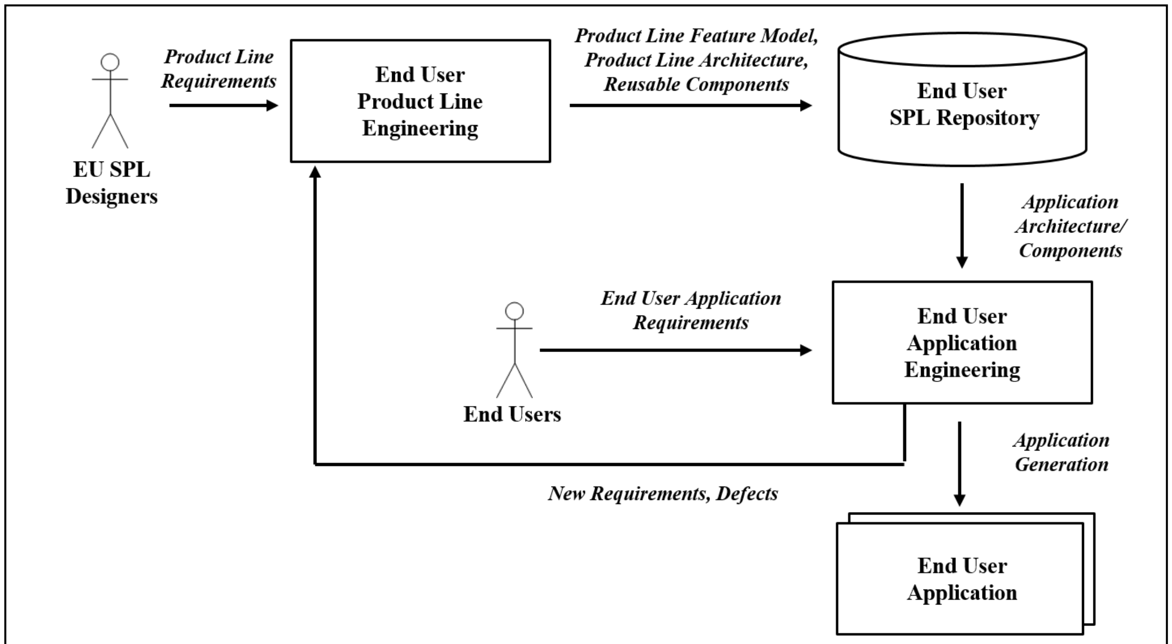

Product Line Requirements
End User Product Line Engineering
Product Line Feature Model, Product Line Architecture, Reusable Components
End User SPL Repository
EU SPL Designers
Application Architecture/ Components
End User Application Requirements
End User Application Engineering
End Users
Application Generation
New Requirements, Defects
End User Application

Figure 4.1 End User Software Product Line Process

engineering process (Gomaa, 2005a), the EU SPL engineering process consists of two sub-processes: (a) the End User Product Line Engineering (EUPLE) process in which the end user software product line is created, and (b) the End User Application Engineering (EUAE) process in which software applications are derived.

Figure 4.2 shows the different phases of the end user product line engineering process. In detail, during end user product line engineering, EU SPL designers work with end users to collect requirements, define the product line scope and create the product line feature model using the EU SPL requirements elicitation process. The feature model captures all the features of the product line and the dependency between them. After the requirements are created, analysis modeling is performed to define: the components needed to implement each feature, the component interactions needed to realize each feature and the component relationships. Components are designed to be reusable to avoid duplication.

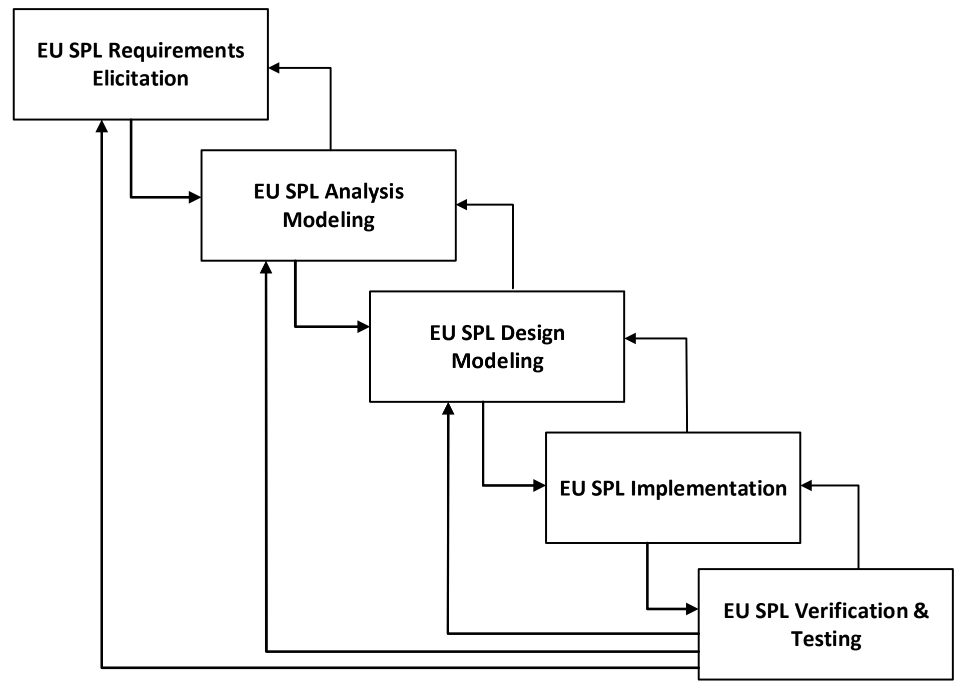


Figure 4.2 End User Product Line Engineering Phases

During design modeling, the EU SPL architecture is created, feature dependency resolution is performed and the component interfaces are defined. During EU SPL implementation the product line components are coded. Finally, during EU SPL testing test cases are defined for the EU SPL features and feature combinations. As shown on Figure 4.2 there is feedback between the different phases of EU Product Line Engineering. In particular, issues and software defects identified during EU SPL testing are communicated to the corresponding phases that the issue was introduced. For example if during testing, a software defect is found that is caused by conflicting features, the issue will be communicated to the EU Analysis Modeling, EU SPL Design Modeling and EU SPL Implementation phases. All artifacts created during the EU SPL engineering are stored in the End User SPL Repository. During end user application engineering, end users select the product line features they need from the EU SPL and derive end user applications for their smart spaces.

Figure 4.3 shows the different phases of End User (EU) Application Engineering. In detail, during the End User Application Requirements Selection phase, end users select the product line features from the EU SPL feature model that they need for their spaces. During the End User Application Derivation phase, the end user application architecture, components and test cases are derived from the EU SPL Repository. The EU Application Testing phase ensures that the test cases are executed successfully against the derived applications. Finally, during the “End User Application Deployment” phase, the derived application is deployed to the end user smart space platform. End users communicate defects and new requirements back to EU SPL designers for future product line releases as shown in Figure 4.3.

4.3 End User Product Line Engineering (EUPLE)

This section describes the End User Product Line Engineering (EUPLE) process. The section starts by discussing different EUPLE strategies for EU SPL designers to develop EU SPLs and then proceeds with describing in detail each of the EUPLE phases.

4.3.1 Forward and Reverse EUPLE Strategy

There are two main EUPLE strategies for creating EU SPLs for smart spaces: (a) forward engineering, and (b) reverse engineering. In the forward engineering strategy, EU

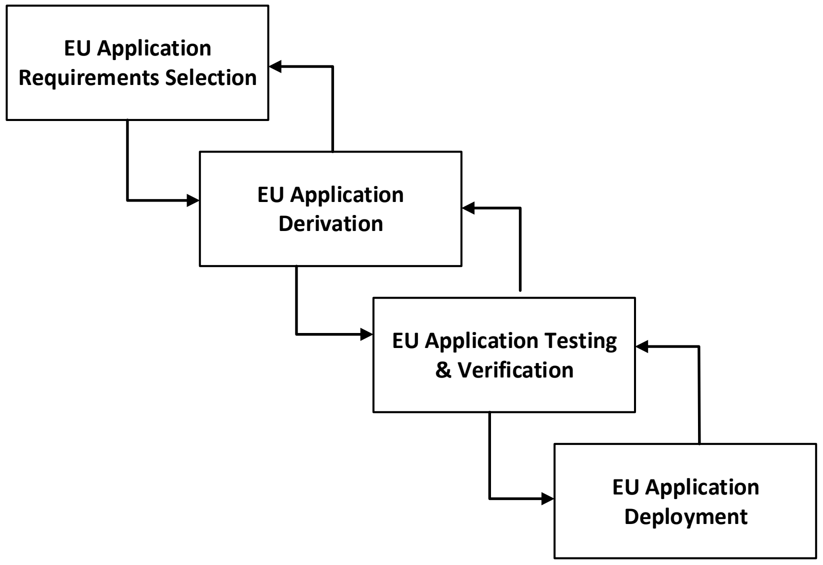

EU Application Requirements Selection
EU Application Derivation
EU Application Testing & Verification
EU Application Deployment

Figure 4.3 End User Application Engineering Phases

SPL designers consider the product line in its entirety. EU SPL designers work with end users to define the product line requirements.

The requirements are classified as kernel, optional or variant. Kernel requirements are implemented by all members of the EU SPL. Optional requirements are implemented by some of the applications derived by the EU SPL. Variant requirements are alternative requirements that can be selected for EUSPL derived applications. An example of a variant requirement is to have derived applications support different languages. Based on the requirements classification: (a) the product line feature model is created, and (b) EUD environment analysis is performed in which EU SPL designers make the determination, based on the feature model, if the EU SPL is going to be applicable to a specific EUD environment, for example Jigsaw or TeC, or if the EU SPL is going to be designed independent of any specific EUD environments. The EU SPL analysis modeling phase involves the creation of the static model, dynamic model and feature/component model. Kernel requirements are considered first and then optional and variant requirements are considered. During design modeling the EU SPL architecture is composed, the design patterns are selected to resolve inter feature component communication, and the component interface is designed. In the design phase kernel features are considered first, and then optional and variant features are added. The implementation phase also starts with the development of kernel features first, and then optional and variant features are implemented. Finally during the testing phase, the product line is tested and verified against the initial requirements.

The reverse engineering approach is used when there are already individual end user applications in place. EU SPL designers derive the EU SPL requirements from the developed end user applications, classify the requirements as common, optional and variant, and create the feature model. The target end user smart space platform determination is also derived by the environments that the existing end user applications are created. During analysis modeling the static model, dynamic model and feature/component model are derived from existing end user applications and requirements. Similar to the forward engineering approach, kernel requirements are considered first followed by optional and variant requirements. Depending on the feature type they realize, components are classified as common, optional or variant. Feature/component modeling is performed to associate features with components they depend on, and these dependencies are depicted in a table view. During dynamic modeling, the product line architecture, design patterns for inter-feature component communication and component interfaces are developed by reverse engineering existing end user applications. Finally, during feature implementation, test cases can also be derived to some extent by reverse engineering and reusing test cases of existing EU applications. The remainder of this chapter will discuss the end user product line engineering from a top down approach.

4.3.2 EU SPL Requirements Elicitation

EU SPL requirements elicitation involves a set of activities to help define the overall scope of the product line. EU SPL designers with domain expertise define the overall road map for the EU SPL. Then EU SPL designers work with end users to collect and document requirements. Based on product line scoping and requirements, the product line feature model is defined. This section describes the end user requirement elicitation process and provides examples for a smart home case study.

4.3.2.1 Use Case Modeling for EU SPL

EU SPL designers can document end user requirements using Use Case modeling. Use Cases describe the interactions between actors which are system external entities and the smart space to achieve a goal. Typical actors in smart spaces are humans, animals, sensors, actuators, devices, and external systems that initiate or detect external events that cause the smart space to react. For example, consider a person entering a smart home. Depending on whether the person is a home resident or an intruder, the smart home can react in different ways. In addition to humans, smart spaces heavily depend on sensors, actuators, devices, and external systems to identify changes to the environment. For instance, a moisture sensor reading might be significant enough to notify a house resident of a possible flood. Use cases for smart spaces should document all the actors that can initiate or detect external events in the smart space. Typical use cases in smart spaces come from the domains of security, automation, space notifications, energy conservation, and ergonomics.

Use case modeling has been extended by the PLUS method to capture product line requirements (Gomaa, 2005a). To document a Use Case for smart spaces using the PLUS method the product line designers need to specify:

- Use Case Name - The name of the use case
- Reuse Category - Specifies weather the use case is kernel, optional or alternative
- Summary - Provides the summary of the use case
- Actors - The actors of the use case (such as humans, animals, sensors, actuators, devices, and external systems)
- Dependency - Use cases that this use case depends
- Preconditions - What conditions need to be true for the use case to execute
- Description - Sequence of events between the actor(s) and the system
- Alternatives - Description of alternatives to the mainstream sequence of events
- Variation Points - Captures places that different functionality can be performed by different members of the product line

- Post Condition - The state of the system after the successful execution of the use case
- Outstanding Questions - Additional questions for end users

EU SPL designers should start documenting the kernel use cases first and then continue with the optional and alternative ones. Table 4.1 shows an example of the Lawn Irrigation Use Case from the Smart Home case study used in this research. The Lawn Irrigation Use Case is part of the smart space automation domain.

Table 4.1 Example of a Lawn Irrigation use case for a smart space

Use Case Name	Lawn Irrigation
Reuse Category	Optional
Summary	The user start/stops the sprinklers to water the lawn. The smart space start/stops the sprinklers and sends outcome notifications
Actors	Home Resident
Dependency	N/A
Preconditions	1. The sprinklers are off 2. The hose is connected to the sprinklers and the water is on
Description	1. The home resident presses the start irrigation button. 2. The smart space starts watering the lawn and sends notifications that is started 3. The home resident presses the stop irrigation button 4. The smart space stops watering the lawn and sends notifications that is stopped
Alternatives	N/A
Variation Points	N/A
Post Condition	The smart space has watered the lawn
Outstanding Questions	Is automation desired? What type of automation is preferred timer of weather sensing?

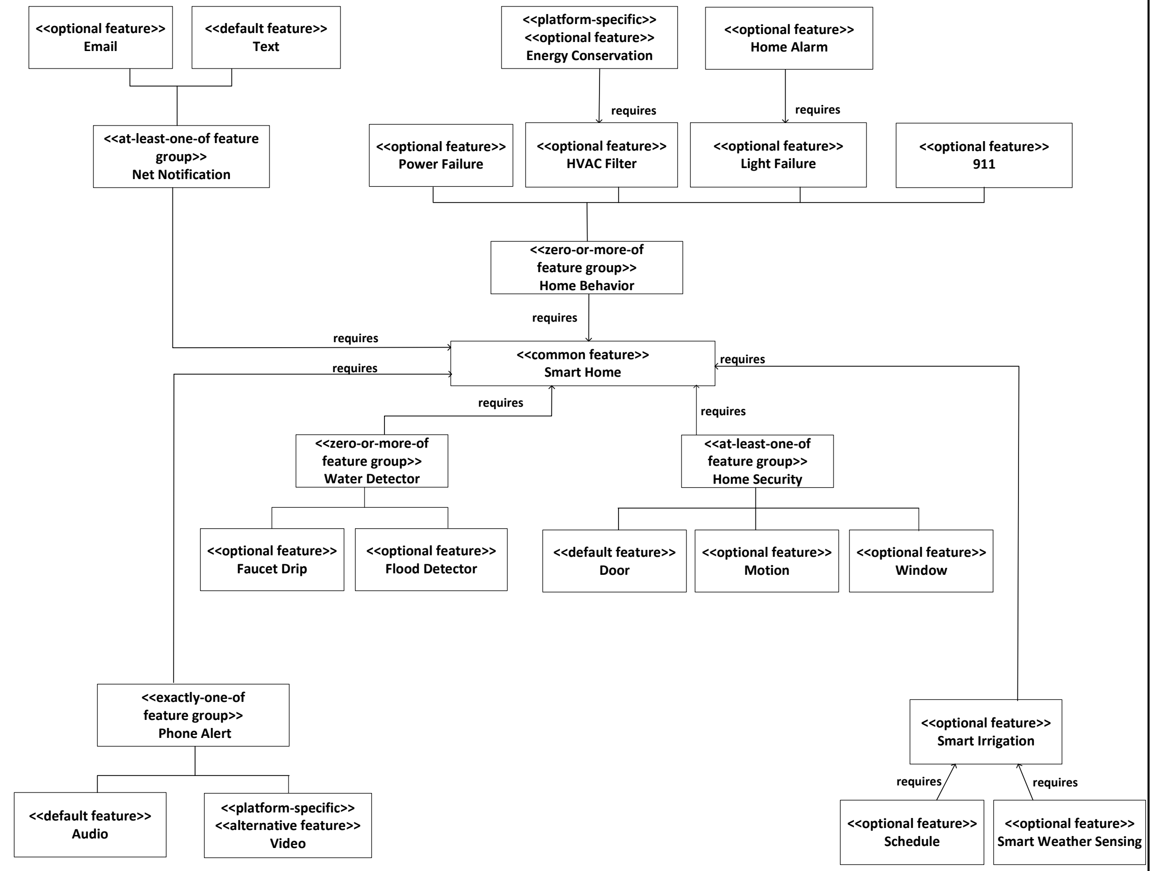
4.3.2.2 Feature Modeling

Product line features are requirements or characteristics that are provided by one or more members of the SPL (Gomaa, 2005a). Feature modeling is used to capture feature commonality / variability and feature dependencies within the EU SPL. In addition, as part of this research, feature modeling was extended to capture feature dependencies in EUD environments (platforms). Product line features can be (a) platform independent to indicate that a feature does not depend on components or functionalities of a specific EUD environment, or (b) platform specific to indicate that a feature depends on components or functionalities of a specific EUD environment e.g, TeC, Jigsaw.

Feature models are derived by use case modeling. In a feature model, features can be organized (a) as common or variable, (b) in feature groups, and (c) as parametrized features. Common features are features that exist in all products derived by the EU SPL. Common features may dependent on other common features. Variable features exist only in some product line members. Variable features can be further categorized as optional or alternative features. Optional features are noncompulsory features that mainly depend on other common or variant features. Alternative features are used to describe mutually exclusive features. Feature groups are used for grouping similar features. Feature groups can be classified as: (a) exactly-one-of, (b) zero-or-one-of, (c) at-least-one-of and (d) zero-or-more-of. Exactly-one-of feature groups indicate that only one feature from a feature group can be present in an end user application derived by the product line. Exactly-one-of feature groups are mainly used to group alternative features, exactly one feature of the group must be selected during application derivation. Zero-or-one-of feature groups are also used to group alternative features but the feature selection from the feature group is optional during application derivation. At-least-one-of feature groups are used to indicate that at least one feature of the feature group must be selected during application derivation. Zero-or-more-of feature groups are used to indicate that zero or more features of the feature group can be selected from the feature group during application derivation. Parameterized features are features that can be configured during the application deployment time. In the feature model, features are decorated with the <<platform-specific>> and <<platform-independent>> UML stereotypes to indicate whether a feature is platform specific or not. If a feature is not decorated with any of the stereotypes, it implies that the feature is platform independent. Figure 4.4 shows the feature model for the Smart Home EU SPL case study developed in this research.

As shown in Figure 4.4 the feature model has one common feature called Smart Home that all other features and feature groups depend on. There is one optional feature Smart Irrigation that depends on the Smart Home feature. The Schedule and Smart Weather Sensing features are also optional and depend on the Smart Irrigation feature. There is one exactly-one-of feature group called Phone Alert that depends on the Smart Home feature. The Phone Alert feature group has two mutually exclusive features Audio and Video. The Audio feature is the default feature and Video is the alternative feature. Default features are selected by default if no other feature in the feature group is selected. The Video feature is platform specific.

The feature model also contains two at-least-one-of feature groups: Net Notification and Home Security. Both of the feature groups depend on the Smart Home



<<optional feature>> Email
<<default feature>> Text
<<platform-specific>> <<optional feature>> Energy Conservation
<<optional feature>> Home Alarm
requires
requires
<<at-least-one-of feature group>> Net Notification
<<optional feature>> Power Failure
<<optional feature>> HVAC Filter
<<optional feature>> Light Failure
<<optional feature>> 911
<<zero-or-more-of feature group>> Home Behavior
requires
requires
requires
<<common feature>> Smart Home
requires
requires
requires
<<zero-or-more-of feature group>> Water Detector
<<at-least-one-of feature group>> Home Security
<<optional feature>> Faucet Drip
<<optional feature>> Flood Detector
<<default feature>> Door
<<optional feature>> Motion
<<optional feature>> Window
<<exactly-one-of feature group>> Phone Alert
<<optional feature>> Smart Irrigation
requires
requires
<<default feature>> Audio
<<platform-specific>> <<alternative feature>> Video
<<optional feature>> Schedule
<<optional feature>> Smart Weather Sensing

Figure 4.4 Smart Home Feature Model

common feature. The Net Notification feature group contains two optional features Email and Text. Text is the default feature. The Home Security feature group contains three optional features: Door, Motion and Window. Door is the default option of the feature group. The Smart Home feature model also contains two zero-or-more-of feature groups: Water Detector and Home Behavior. The Water Detector feature group contains two optional features Faucet Drip and Flood Detector. The Home Behavior feature group contains four optional features: Power Failure, HVAC Filter, Light Failure and 911. In addition the Home Alarm optional feature depends on the Light Failure feature.

Furthermore the Energy Conservation optional feature depends on the HVAC Filter. The Energy Conservation feature also is platform specific.

The Feature group / Feature dependency table is another view that captures the relationship between product line features and feature groups. The Feature group / Feature dependency table assists EU SPL designers to ensure consistency between features and feature groups. As shown on Table 4.2 the table has four columns: (a) Feature Group Name, (b) Feature Group Category, (c) Feature Name, and (d) Feature Category. The Feature Group Category and Feature Category need to be compatible for example exactly-one-of feature group needs to have a set of alternative features since only one can be selected. Table 4.2 shows the Feature Group / Feature dependency table for the Smart Home case study. For example as shown on Table 4.2 the Phone Alert exactly-one-of feature group has two alternative features Audio and Video with the Audio feature being the default option.

Table 4.2 Feature Group / Feature Dependency Table

Feature Group Name	**Feature Group Category**	**Features in Feature Group**	**Feature Category**
Phone Alert	exactly-one-of	Audio Video	default alternative
Home Security	at-least-one-of	Door Motion Window	default optional optional
Water Detector	zero-or-more-of	Flood Detector Faucet Drip	optional optional
Home Behavior	zero-or-more-of	Light Failure HVAC Filter Power Failure 911	optional optional optional optional
Net Notification	at-least-one-of	Text Email	default optional

4.3.3 EU SPL Analysis Modeling

EU SPL Analysis modeling consists of static modeling, component structuring, dynamic modeling and feature/component modeling.

4.3.3.1 Static Model

The EU SPL static model captures the product line components needed to realize the use cases defined and feature model. In addition component structuring is performed to capture the component reuse stereotype, role stereotype and platform dependencies. This research used UML stereotypes to classify the EU SPL components. To capture component reuse characteristics, the following reuse stereotypes are used <<kernel>>, <<optional>>, <<variant>>, <<default>>. This research uses the PLUS method role stereotypes to capture the application purpose of each component (Gomaa, 2005a). For example a component can be <<interface>>, <<entity>>, <<control>>, <<application logic>>, <<timer>>, <<system interface>>, <<coordinator>>, <<device interface>>, <<algorithm>>, <<message-broker>>, <<input/output device interface>>, etc. Components that are only applicable to specific end user environments are annotated with the <<platform-specific>> stereotype.

Figure 4.5 shows the static model and the component structuring for the components used in the Smart Home case study used in this research. For example as shown on the securityAlertHandler component is annotated with the <<kernel>> stereotype to capture reuse category and the <<message-broker>> stereotype to capture the component role category. Similarly the component videoCall is annotated with the <<optional>> stereotype to capture the reuse category, the <<input / output device

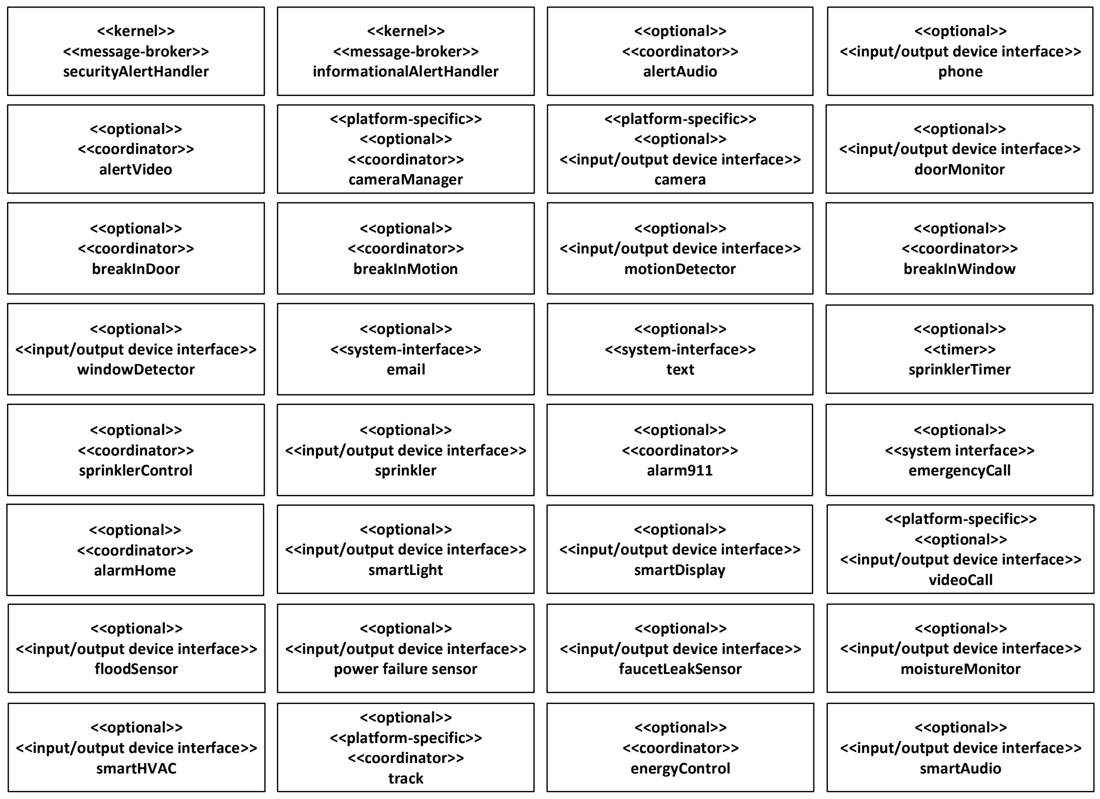

<<kernel>> <<message-broker>> securityAlertHandler
<<kernel>> <<message-broker>> informationalAlertHandler
<<optional>> <<coordinator>> alertAudio
<<optional>> <<input/output device interface>> phone
<<optional>> <<coordinator>> alertVideo
<<platform-specific>> <<optional>> <<coordinator>> cameraManager
<<platform-specific>> <<optional>> <<input/output device interface>> camera
<<optional>> <<input/output device interface>> doorMonitor
<<optional>> <<coordinator>> breakInDoor
<<optional>> <<coordinator>> breakInMotion
<<optional>> <<input/output device interface>> motionDetector
<<optional>> <<coordinator>> breakInWindow
<<optional>> <<input/output device interface>> windowDetector
<<optional>> <<system-interface>> email
<<optional>> <<system-interface>> text
<<optional>> <<timer>> sprinklerTimer
<<optional>> <<coordinator>> sprinklerControl
<<optional>> <<input/output device interface>> sprinkler
<<optional>> <<coordinator>> alarm911
<<optional>> <<system interface>> emergencyCall
<<optional>> <<coordinator>> alarmHome
<<optional>> <<input/output device interface>> smartLight
<<optional>> <<input/output device interface>> smartDisplay
<<platform-specific>> <<optional>> <<input/output device interface>> videoCall
<<optional>> <<input/output device interface>> floodSensor
<<optional>> <<input/output device interface>> power failure sensor
<<optional>> <<input/output device interface>> faucetLeakSensor
<<optional>> <<input/output device interface>> moistureMonitor
<<optional>> <<input/output device interface>> smartHVAC
<<optional>> <<platform-specific>> <<coordinator>> track
<<optional>> <<coordinator>> energyControl
<<optional>> <<input/output device interface>> smartAudio

Figure 4.5 Smart Home Case Study Static Model

interface>> stereotype to capture the role category and the <<platform-specific>> stereotype to indicate that this component only applies to specific platforms.

The Platform Specific Feature / Component relationship table captures the relationship between platform specific features and platform specific components. As shown in Table 4.3 the platform specific feature / component relationship table has 4 columns: (a) Feature Name, (b) Platform Name, (c) Component Name and (d) Platform Specific Name. The Feature Name column captures the name of the feature. The Platform Name column captures the end user platform(s) that the feature applies. The Component Name column captures the component name as it appears on the static model. The Platform

Table 4.3 Platform Specific Feature / Component relationship table

Feature Name	Platform Name	Component Name	Platform Specific Name
Energy Conservation	Team Computing	track	tecTrack
Video	Team Computing	videoCall cameraManager camera	tecVideoCall tecCameraManager tecCamera

Specific Name column captures the actual component name in the specific platform. For example the Energy Conservation feature applies only to the TeC platform. The track component of the Energy Conservation feature would have to be mapped to the tecTrack component of Team computing during the end user application deployment process.

4.3.3.2 Dynamic Modeling

EU SPL designers use dynamic modeling to capture the object interactions needed to satisfy EU SPL features. This research used UML sequence diagrams to model object interactions. Sequence diagrams model the message interaction of objects based on a time sequence (Rumbaugh et al., 2004). Sequence diagrams should be developed for all features defined in the feature model of the EU SPL.

Figure 4.6 shows the sequence diagram for the Video feature that is part of the Phone Alert feature group. The sequence interaction starts with the :alertVideo object that after initialization [init=true] sends a message to the subscribe input of the :securityAlertHandler object to receive security alert notifications. When a security alert is detected by the :securityAlertHandler [sendAlert=true] it sends the security alert message

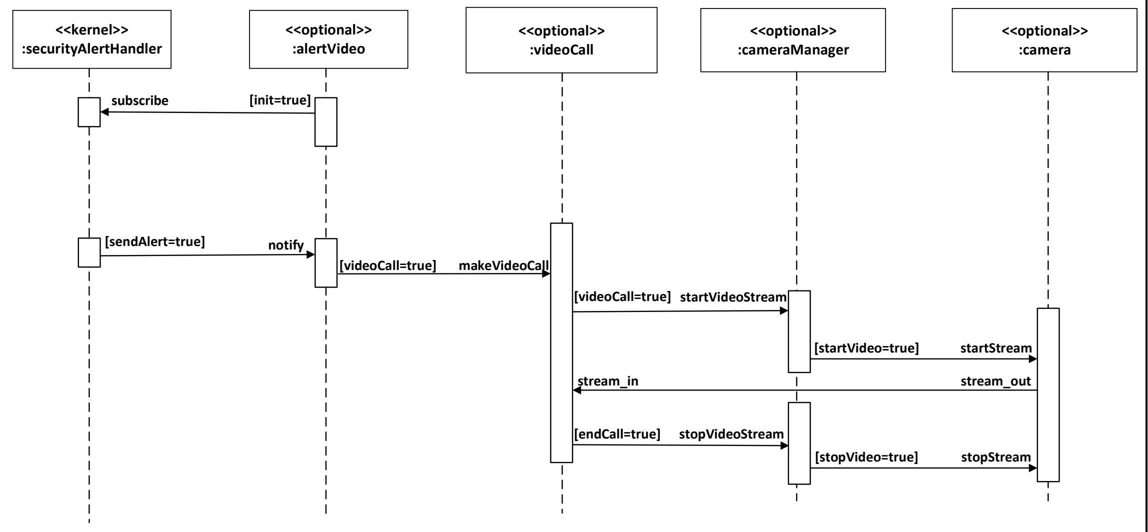


Figure 4.6 Sequence Diagram for the Video Feature

to the notify input event of the :alertVideo object. Upon receiving the security alert message the :alertVideo object evaluates the [videoCall=true] condition and if true sends a message to the makeVideoCall input of the :videoCall object. The :videoCall object represents a smart phone device. When the makeVideoCall input is called a video call is initiated on the smart phone device and the [videoCall=true] condition is evaluated and if true the :videoCall object sends a message to the startVideoStream input of :cameraManager in order to request a video stream. Upon receiving the message the :cameraManager evaluates the [startVideo=true] condition and if true it sends a message to the startStream input of the :camera object with the :videoCall object information. The camera will send the video stream stream_out to the :videoCall object stream_in input in order for the video to be displayed on the device. Upon the end of the phone call the [endCall=true] condition of the :videoCall object evaluates to true and the :videoCall object sends a message to the stopVideoStream input of the :cameraManager object. The stopVideoStream will evaluate the [stopVideo=true] condition and if true the :cameraManager object will send a message to the stopStream input of the :camera object to indicate that the :camera object can stop sending the video stream to the :videoAlert object.

Similarly, Figure 4.7 shows the sequence diagram for the Energy Conservation feature from the Smart Home EU SPL case study. The goal of the feature is to conserve energy when the house residents are away by adjusting the home temperature. The temperature will be adjusted back to normal when residents return home. The sequence diagram starts with the :track object that sends a message to the econ input of the :energyControl object when the house residents are away [away=true]. When the econ input is received the [adjustHvacLevel=true] and [energyLevelNotification=true] conditions are evaluated to true. The :energyControl object sends a message to the setHvacLevel input of the :smartHVAC object with the desired temperature settings. Furthermore the :energyControl objects sends a message to the receiveAlert input of the :informationAlertHandler object with the energy level changes. When the residents are back home the :track object [home=true] condition is evaluated to true, which causes the :track object to send a message to the norm input of the :energyControl object. When the norm input is received the [adjustHvacLevel=true] and [energyLevelNotification=true] conditions are evaluated to true. The :energyControl object sends a message to the setHvacLevel input of the :smartHVAC object to adjust temperature settings back to

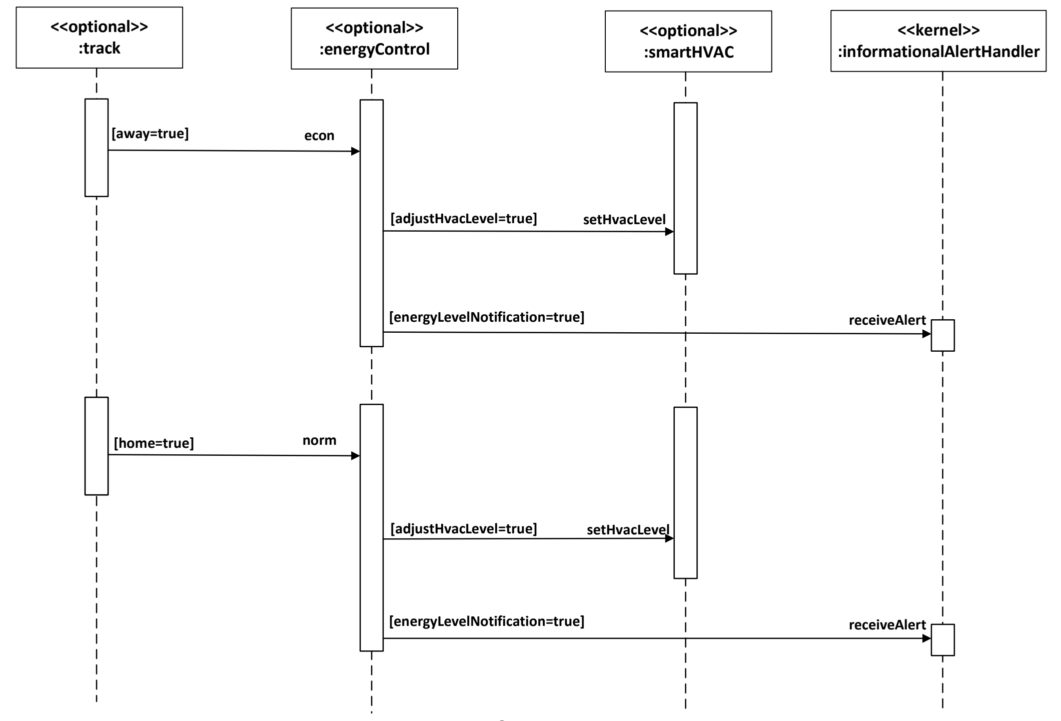


Figure 4.7 Sequence Diagram for the Energy Conservation Feature

normal. In addition, the :energyControl objects sends a message to the receiveAlert input of the :informationAlertHandler object with the energy level changes.

4.3.3.3 Feature / Component Dependency Table

The Feature / Component table describes in detail the EU SPL features and the components needed to support the implementation of each of the features. The purpose of the table is for EU SPL designers to ensure consistency between features and the components that support them. For example a common feature cannot be implemented by optional components. The Feature/Component table contains the following columns:

- Feature Name – The name of the Feature

- Feature Group – The name of the Feature Group that the Feature belongs
- Feature Category – The type of feature (common, optional, etc.)
- Component Name – The components names that implement each feature
- Component Reuse Category – The component type (kernel, optional, etc.)
- Component Parameter – Component Parameters needed by the Feature

Table 4.4 shows the Feature / Component Dependency Table that was developed for the Smart Home EU SPL Case Study used in this research. For example the common feature Smart Home is implemented by the securityAlertHandler and the informationalAlertHandler component that are kernel. Similarly the alternative Video feature is implemented by the alertVideo, videoCall, cameraManager and camera optional components. Since the Video feature depends on the Smart Home feature, the Video feature will also be supported by the securityAlertHandler and informationalAlertHandler kernel components. Finally, the optional Energy Conservation feature is implemented by the optional track and energyControl components. The component parameter residentIDs of the track component indicate the smart home residents that need to be tracked by the component.

4.3.4 EU SPL Design Modeling

While EU SPL Analysis modeling focus on the analysis of the problem domain, EU SPL Design modeling maps the EU SPL Analysis model to the solution domain (Gomaa, 2016). During EU SPL Design modeling the component inter-feature communication, component relationships and component interface models are defined.

Table 4.4 Feature / Component Dependency Table for the Smart Home EU SPL Case Study

Feature Name	**Feature Group Name**	**Feature Category**	**Component Name**	**Component Reuse Category**	**Component Parameter**
Smart Home		common	securityAlertHandler informationalAlertHandler	kernel kernel	
Audio	Phone Alert	default	alertAudio phone	optional optional	
Video	Phone Alert	alternative	alertVideo videoCall cameraManager camera	optional optional optional optional	
Door	Home Security	default	breakInDoor doorMonitor	optional optional	
Motion	Home Security	optional	breakInMotion motionDetector	optional optional	
Window	Home Security	optional	breakInWindow windowDetector	optional optional	
Smart Irrigation		optional	sprinkler sprinklerControl	optional optional	
Schedule		optional	schedule	optional	timetorun : String
Smart Weather Sensing		optional	moistMonitor	optional	
Email	Net Notification	optional	email	optional	
Text	Net Notification	default	text	optional	
Flood Detector	Water Detector	optional	floodSensor	optional	
Faucet Drip	Water Detector	optional	faucetLeakSensor	optional	
Home Alarm	Home Behavior	optional	alarmHome smartAudio smartDisplay	optional optional optional	
911	Home Behavior	optional	alarm911 emergencyCall	optional optional	
Light Failure	Home Behavior	optional	smartLight	optional	
HVAC Filter	Home Behavior	optional	smartHVAC	optional	
Power Failure	Home Behavior	optional	powerFailureSensor	optional	
Energy Conservation	Home Behavior	optional	track energyControl	optional optional	residentIDs: List<String>

4.3.4.1 Inter-Feature Component Communication

As EU SPL designers define features and the components that implement each feature, they might determine situations where components of one feature need to communicate with components of other features to accomplish a task. One solution to this problem is to refactor the feature model to support this. Refactoring will work for smaller feature models but as the model grows that might not be a viable option. This research utilized the subscription/notification (Gomaa, 2016) design pattern for inter feature component communication as an alternative option to feature refactoring. The idea is that instead of components sending messages directly to each other, message broker components are provided as intermediaries. Components can send messages to the message broker, which then notifies components that have registered with the message broker to receive messages. Some benefits of the public / subscribe design pattern for developing EU SPLs are (a) promotes loose coupling between sender and receiver components and (b) better scalability since newly created components can register with existing message brokers to send and receive messages. The inter-feature component communication table captures all product line components that send and receive messages through message broker components. Table 4.5 shows the inter-feature component communication table

Table 4.5 Inter-Feature Component Communication Table for the Smart Home Case Study

Message Broker	**Subscribed Components**	**Message Producer Components**
securityAlertHandler	alertAudio alertVideo alarmHome alarm911 email text	breakInDoor breakInMotion breakInWindow
informationalAlertHandler	email text	schedule sprinklerControl smartLight smartHVAC powerFailureSensor energyControl floodSensor faucetLeakSensor

that was created to support the Smart Home case study. There are following columns in the table:

- Message Broker – The name of the message broker component
- Subscribed Components – The components subscribed to receive messages
- Message Producer Components – The components producing the messages

For example as shown on Table 4.5 the securityAlertHandler is a message broker component. The components alertAudio, alertVideo, alarmHome, alarm911, email, text are subscribed and receive messages from the securityAlertHandler. The breakInDoor, breakInMotion, breakInWindow components send messages to the securityAlertHandler. As shown on the second row of Table 4.5 the text component is also subscribed and receive messages from the informationalAlertHandler to support a different use case.

Figure 4.8 shows an example of component communication using subscription/notification from the Smart Home EU SPL case study. In detail, when the alertAudio component of the Audio feature initializes, it sends a message with its id to the securityAlertHandler message broker component. Components that have subscribed to receive messages from a message broker are shown in the Subscribed Components column in the inter-feature component communication table. When there is a break-in activity, the breakInDoor component of the Door feature sends alerts to the securityAlertHandler. When there is a message available, the securityAlertHandler sends it to the alertVideo component.

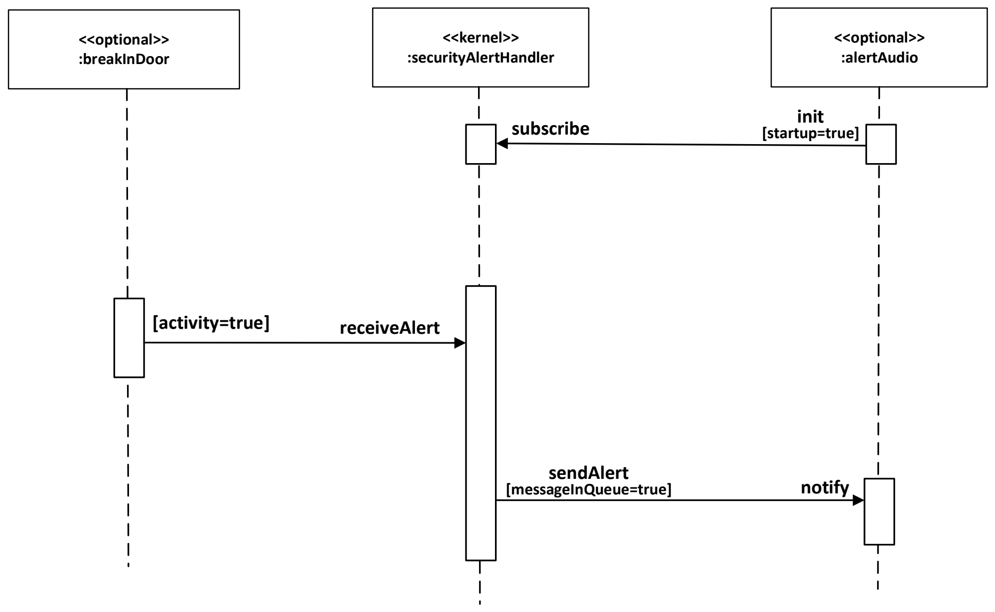


Figure 4.8 Subscribe and Receive Messages to a Message Broker

4.3.4.2 Component Relationships and Interface Design

UML component diagrams can be used by EU SPL designers to capture (a) components available in a smart home, (b) component relationships, and (c) provided and required interfaces needed for components to communicate. In an end user environment, components communicate with other components through output/input ports. Figure 4.9 shows the component diagram of the Home Alarm Feature. The component diagram is composed of the securityAlertHandler, alarmHome, smartAudio, smartDisplay and the smartLight components.

The components are decorated with UML stereotypes to indicate whether a component is kernel, optional, or variant. For example the securityAlertHandler is a <<kernel>> component while alarmHome, smartAudio, smartDisplay and smartLight are <<optional>> components. Furthermore additional stereotypes are used to capture the role

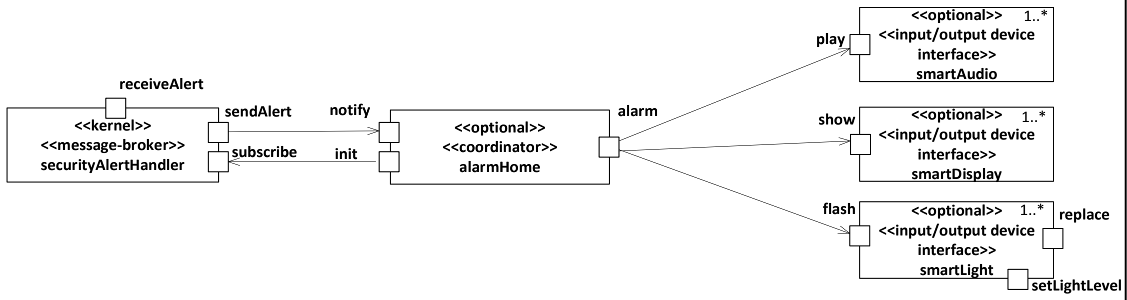


Figure 4.9 Component Diagram for the Home Alarm Feature

of each component. For example securityAlertHandler is a <<message-broker>> component, alarmHome is a <<coordinator>> component while smartAudio, smartDisplay and smartLight are input/output device interface components. Components may also have a multiplicity indicator to indicate the number of component instances in a smart space. For example the smartAudio, smartDisplay and smartLight components have 1…* multiplicity that indicates that there are one or more smartLight, smartAudio and smartDisplay component instance in the smart space. The connections between components also indicate the required and provided interfaces between components.

Table 4.6 shows details about the components input and output messages. For example the init output of the alarmHome component outputs the clientID parameter to indicate the component identification. The securityAlertHandler has a subscribe input and expects as input the clientID parameter to indicate the component that is subscribing to receive messages. Similarly the securityAlertHandler has a sendAlert output that outputs a message parameter to indicate the alert message. As shown on Figure 4.9 the sendAlert output of the securityAlertHandler is connected to the notify input of the alarmHome. The

Table 4.6 Component Input / Output for the Home Alarm Feature

Component Name	Component Input	Component Output	Component Output Triggering Condition

securityAlertHandler	receiveAlert(in message)
	subscribe(in clientID)
		sendAlert(out message)	messageInQueue=true

alarmHome	notify(in message)	init(out clientID)
		alarm(out message)
			startup=true
			message=true

smartAudio	play(in message)		
smartDisplay	show(in message)		

smartLight	flash()
	setLightLevel(in lightLevel)
		replace(out lightID)	light=out

notify input expects as a parameter the message to distribute to the appropriate devices. Parameters sent from outputs to inputs can be ignored by the inputs if the parameters are not relevant. For instance the alarm output of the alarmHome component outputs a message parameter. The play input of the smartAudio and the show input of the smartDisplay use the message parameter to play the message over the house speakers or to display the message to the house monitors. Figure 4.10 shows the component diagram of the Video feature. The diagram contains the components: securityAlertHandler, alertVideo, videoCall, cameraManager and camera that implement the Video feature. The components videoCall, cameraManager and camera are annotated with the <<platform-specific>> stereotype. The <<platform-specific>> stereotype indicates that the components are specific to a specific EUD environment for smart spaces. The Platform Specific Feature table contains additional details about the EUD environment and the applicable components.

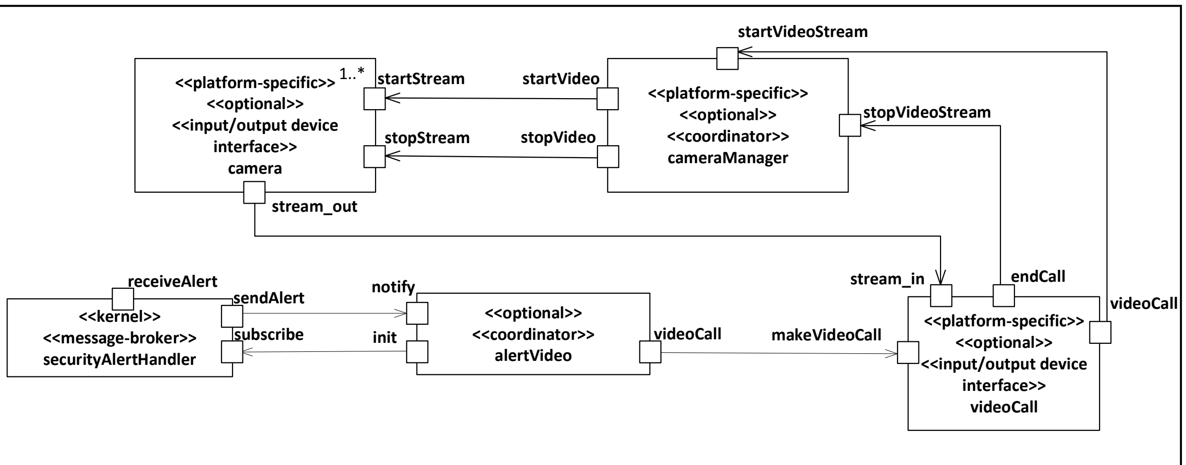


Figure 4.10 Component Diagram for the Video Feature

4.3.5 EU SPL Implementation

EU SPL implementation is the process for implementing the code of the product line components. The EUSPLP development environment created by this research can be used to create EU SPLs. The EUSPLP is described in detail in Chapter 6.

4.3.6 EU SPL Testing

This research developed an EU SPL testing framework for testing end user product lines. The EU SPL testing framework is described in detail in Chapter 7.

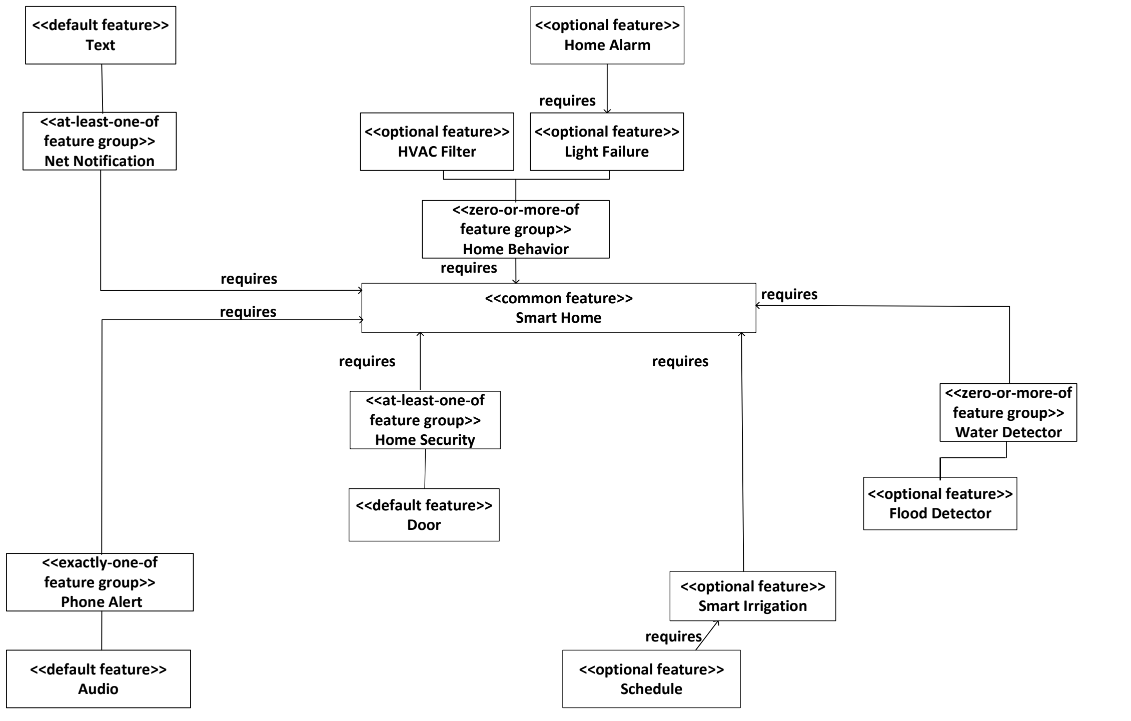
4.4 End User Application Engineering (EUAE)

End User Application Engineering (EUAE) is the process to derive end user applications from the End User SPL and deploy end user applications to end user smart spaces. The EUAE process can be broken down to the (a) End User Application Requirements Selection, (b) End User Application Derivation, (c) End User Application Testing, and (d) End User Application Deployment phases.

4.4.1 End User Application Requirements Selection

During the End User Application Requirements Selection phase end users specify the required EU SPL features for their spaces. The selected features need to be compatible with other features selected from the EU SPL. For instance an end user cannot select two alternative features or select zero features form an at-least-one-of feature group. The outcome of the EU application requirements process is a derived feature model that captures the features that end users selected. Figure 4.11 shows an example of features that an end user selected from the Smart Home case study used in this research.

As shown in Figure 4.11 the following features were selected: Smart Home, Audio, Flood detector, Door, Smart Irrigation, Schedule, Text, HVAC Filter, Light Failure and Home Alarm. The Smart Home is a common feature that all features depend on. The Audio feature was selected as an example of a feature selected from an exactly-one-of feature group (Phone Alert). The text feature was selected as example of a feature selected from an at-least-one-of feature group (Net Notification). Similarly, the door feature was selected from the at-least-one-of Home Security feature group. The HVAC Filter and Light Failure features are selected from the zero-or-more-of Home Behavior feature group. The Flood Detector is another zero-or-more-of feature selected from the Water Detector feature group. The Smart Irrigation feature is an example of an optional feature. Finally the Home Alert and Schedule are examples of optional features that depend on other optional features. The features selected are compatible with each other. For instance there are no mutually exclusive features selected etc.



<<default feature>> Text
<<optional feature>> Home Alarm
requires
<<at-least-one-of feature group>> Net Notification
<<optional feature>> HVAC Filter
<<optional feature>> Light Failure
<<zero-or-more-of feature group>> Home Behavior
requires
requires
requires
<<common feature>> Smart Home
requires
requires
requires
requires
<<at-least-one-of feature group>> Home Security
<<zero-or-more-of feature group>> Water Detector
<<default feature>> Door
<<optional feature>> Flood Detector
<<exactly-one-of feature group>> Phone Alert
<<optional feature>> Smart Irrigation
requires
<<default feature>> Audio
<<optional feature>> Schedule

Figure 4.11 Example of an Instance of the Smart Home Feature Model based on End User Requirements

4.4.2 End User Application Derivation

The application derivation phase is responsible for deriving the end user application based on the end user feature selections. In detail, the components, component connectors, and component configuration parameters that realize the selected features are derived from the EU SPL Repository. Then there is a component mapping process that maps the components derived from the EU SPL Repository to the components of the target EUD environment to create the application architecture. The component mappings from the SPL to the EUD environment are described in detail in Chapter 5.

Table 4.7 shows the application mapping for the Smart Home derived application to the

Table 4.7 Example of Derived End User Application Mapped to Jigsaw

Feature Name	Feature Group Name	JIGSAW PSP Component Name	JIGSAW PSP Component Input	JIGSAW PSP Component Output	JIGSAW PSP Component Output Triggering Condition

Smart Home		securityAlertHandler	receiveAlert(in message)
			subscribe(in clientID)
				sendAlert(out message)	messageInQueue=true

Smart Home		informationalAlertHandler	receiveAlert(in message)
			subscribe(in clientID)
				sendAlert(out message)	messageInQueue=true

Light Failure	Home Behavior	smartLight	flash	replace(out lightID)	light=false
HVAC Filter	Home Behavior	smartHVAC		replace filter(out hvacID)	replaceFilter=true

Home Alarm	Home Behavior	alarmHome	notify(in message)	init(out clientID)
				alarm(out message)
					startup=true
					message=true

Home Alarm	Home Behavior	smartAudio	play(in message)		
Home Alarm	Home Behavior	smartDisplay	show(in message)		

Audio	Phone Alert	alertAudio	notify(in message)	init(out clientID)
				call(out phone_number, out message)
					startup=true
					message=true

Audio	Phone Alert	phone	makeCall(in phone_number, in message)		
Door	Home Security	breakInDoor	action(in deviceID, in deviceType)	activate activity(out message)	startup=true motion=true
Door	Home Security	doorMonitor	on	movement(out deviceID , out deviceType)	move=true
Flood Detector	Water Detector	floodSensor		flood(out location)	moisture=true
Smart Irrigation		sprinkler	startWater stopWater		
Smart Irrigation		sprinklerControl	water	turn on (out message) turn off (out message)	
Schedule		sprinklerTimer		timeAlert(out message)	scheduledTimerAlert=true
Text		text	notify(in message)	init(out clientID)	startup=true

Jigsaw EUD environment based on the features selected in Figure 4.11. The table shows all the components, inputs / outputs and triggering conditions used to react to smart space events. Figure 4.12 visualizes the derived end user application architecture as it would be

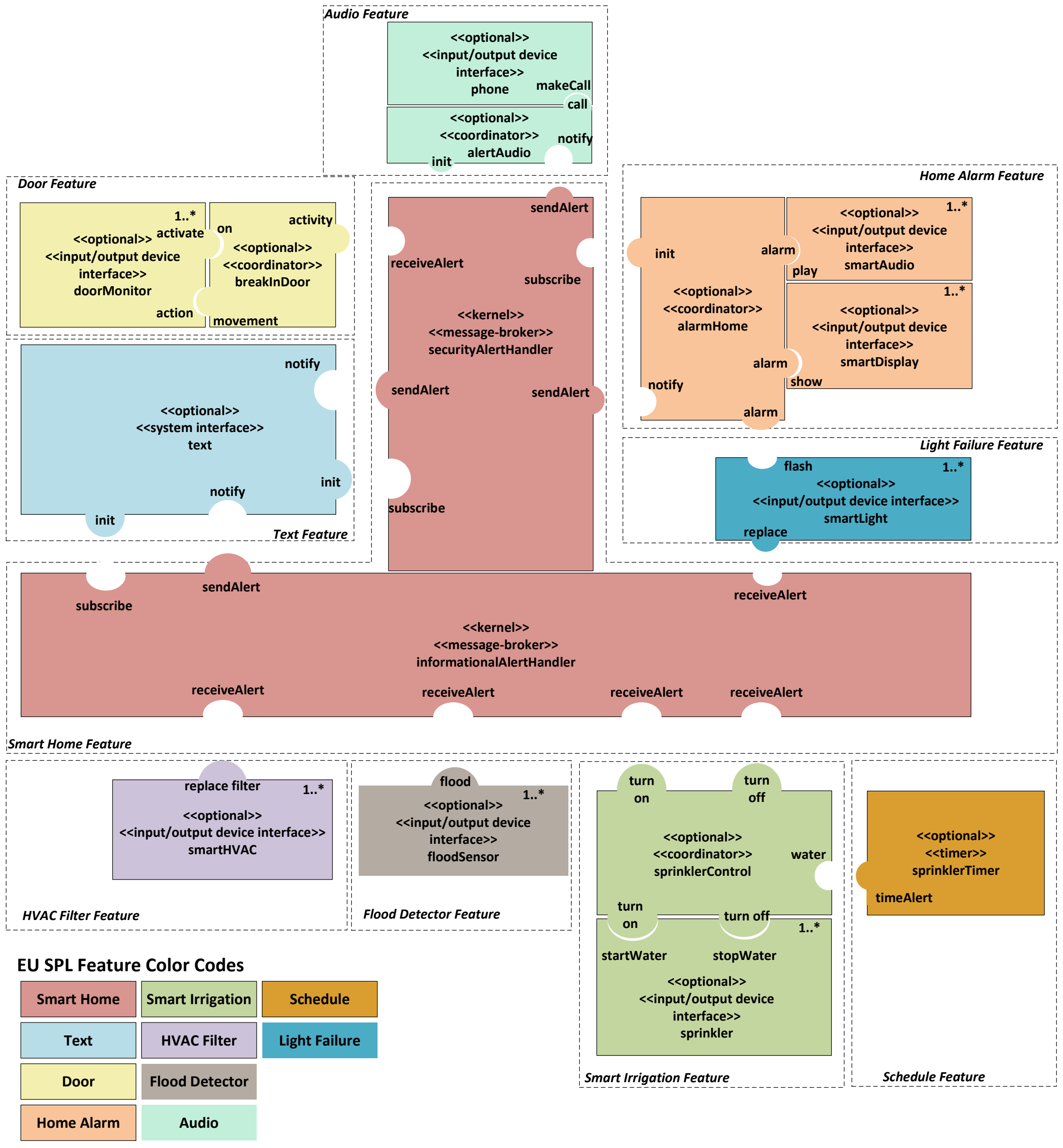

Audio Feature
<<optional>> <<input/output device interface>> phone
makeCall
call
<<optional>> <<coordinator>> alertAudio
notify
init
Door Feature
1..*
<<optional>> <<input/output device interface>> doorMonitor
activate
action
on
activity
<<optional>> <<coordinator>> breakInDoor
movement
sendAlert
receiveAlert
subscribe
<<kernel>> <<message-broker>> securityAlertHandler
sendAlert
sendAlert
subscribe
Home Alarm Feature
init
alarm
<<optional>> <<coordinator>> alarmHome
alarm
notify
alarm
<<optional>> <<input/output device interface>> smartAudio
play
1..*
<<optional>> <<input/output device interface>> smartDisplay
show
notify
<<optional>> <<system interface>> text
init
notify
init
Text Feature
Light Failure Feature
flash
1..*
<<optional>> <<input/output device interface>> smartLight
replace
subscribe
sendAlert
receiveAlert
<<kernel>> <<message-broker>> informationalAlertHandler
receiveAlert
receiveAlert
receiveAlert
receiveAlert
Smart Home Feature
replace filter
1..*
<<optional>> <<input/output device interface>> smartHVAC
HVAC Filter Feature
flood
1..*
<<optional>> <<input/output device interface>> floodSensor
Flood Detector Feature
turn on
turn off
<<optional>> <<coordinator>> sprinklerControl
water
turn on
turn off
1..*
startWater
stopWater
<<optional>> <<input/output device interface>> sprinkler
Smart Irrigation Feature
<<optional>> <<timer>> sprinklerTimer
timeAlert
Schedule Feature
EU SPL Feature Color Codes
Smart Home
Smart Irrigation
Schedule
Text
HVAC Filter
Light Failure
Door
Flood Detector
Home Alarm
Audio

Figure 4.12 Example of Smart Home End User Application Architecture for Jigsaw

displayed to the Jigsaw editor. As shown in Figure 4.12, components are represented as Jigsaw pieces put together to form application logic. Similarly, Table 4.8 shows the

Table 4.8 Example of Derived End User Application Mapped to Team Computing

Feature Name	Feature Group Name	TEC PSP Component Name	TEC PSP Component Input	TEC PSP Component Output	TEC PSP Component Output Triggering Condition

Smart Home		securityAlertHandler	receiveAlert(in message)
			subscribe(in clientID)
				sendAlert(out message)	messageInQueue=true

Smart Home		informationalAlertHandler	receiveAlert(in message)
			subscribe(in clientID)
				sendAlert(out message)	messageInQueue=true

Light Failure	Home Behavior	smartLight	flash	replace(out lightID)	light=false
HVAC Filter	Home Behavior	smartHVAC		replace filter(out hvacID)	replaceFilter=true

Home Alarm	Home Behavior	alarmHome	notify(in message)	init(out clientID)
				alarm(out message)
					startup=true
					message=true

Home Alarm	Home Behavior	smartAudio	play(in message)		
Home Alarm	Home Behavior	smartDisplay	show(in message)		

Audio	Phone Alert	alertAudio	notify(in message)	init(out clientID)
				call(out phone_number, out message)
					startup=true
					message=true

Audio	Phone Alert	phone	makeCall(in phone_number, in message)		
Door	Home Security	breakInDoor	action(in deviceID, in deviceType)	activate activity(out message)	startup=true motion=true
Door	Home Security	doorMonitor	on	movement(out deviceID , out deviceType)	move=true
Flood Detector	Water Detector	floodSensor		flood(out location)	moisture=true
Smart Irrigation		sprinkler	startWater stopWater		
Smart Irrigation		sprinklerControl	water	turn on (out message) turn off (out message)	
Schedule		sprinklerTimer		timeAlert(out message)	scheduledTimerAlert=true
Text		text	notify(in message)	init(out clientID)	startup=true

application mapping for Smart Home derived application to the Team Computing EUD environment based on the feature selections shown in Figure 4.11. Figure 4.13 visualizes

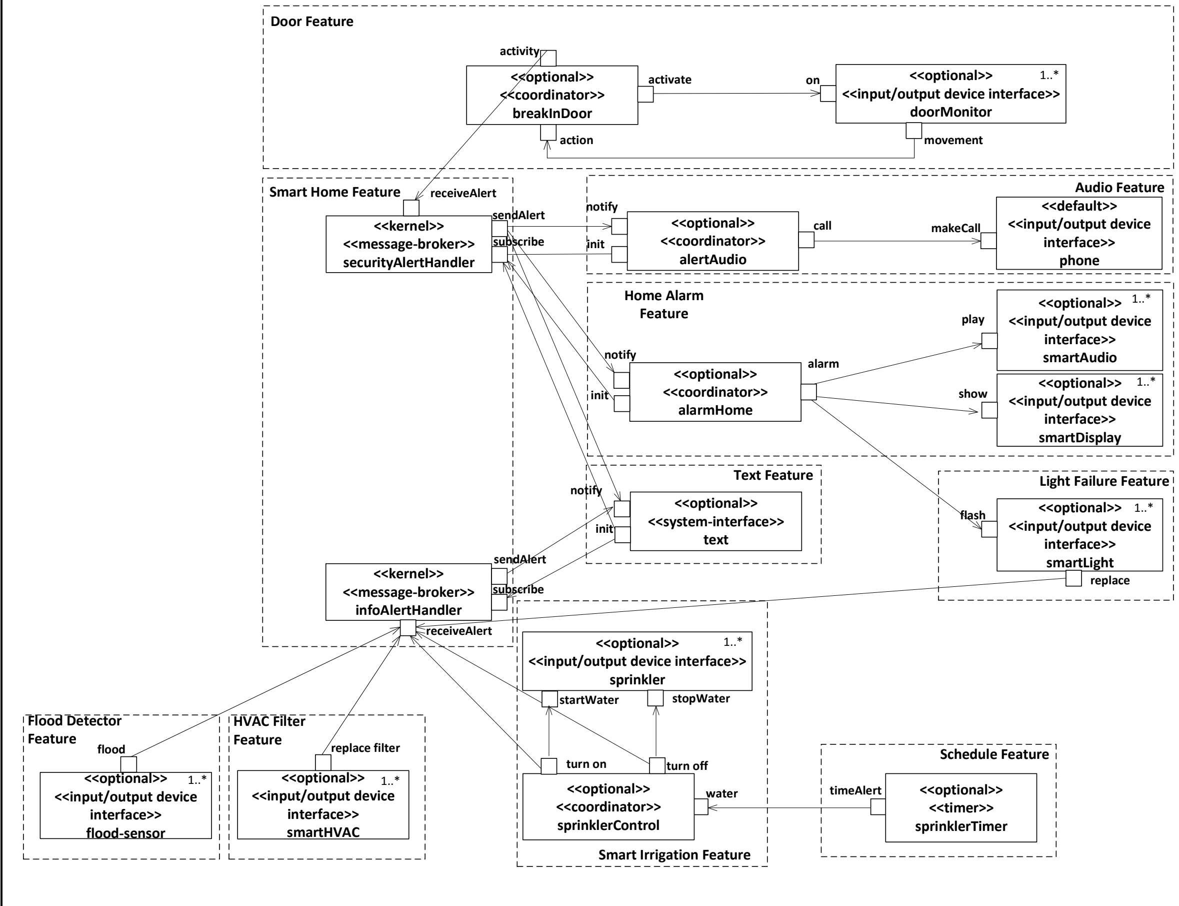

Door Feature
activity
<<optional>> <<coordinator>> breakInDoor
activate
on
<<optional>> <<input/output device interface>> doorMonitor
1..*
action
movement
Smart Home Feature
receiveAlert
<<kernel>> <<message-broker>> securityAlertHandler
sendAlert
subscribe
Audio Feature
notify
init
<<optional>> <<coordinator>> alertAudio
call
makeCall
<<default>> <<input/output device interface>> phone
Home Alarm Feature
<<optional>> <<input/output device interface>> smartAudio
play
notify
init
<<optional>> <<coordinator>> alarmHome
alarm
show
<<optional>> <<input/output device interface>> smartDisplay
Text Feature
notify
init
<<optional>> <<system-interface>> text
Light Failure Feature
flash
<<optional>> <<input/output device interface>> smartLight
replace
<<kernel>> <<message-broker>> infoAlertHandler
sendAlert
subscribe
receiveAlert
<<optional>> <<input/output device interface>> sprinkler
startWater
stopWater
Flood Detector Feature
flood
<<optional>> <<input/output device interface>> flood-sensor
HVAC Filter Feature
replace filter
<<optional>> <<input/output device interface>> smartHVAC
turn on
turn off
<<optional>> <<coordinator>> sprinklerControl
water
Smart Irrigation Feature
Schedule Feature
timeAlert
<<optional>> <<timer>> sprinklerTimer

Figure 4.13 Example of Smart Home End User Application Architecture for TeC

the derived application architecture as it would be displayed to the Team Computing application editor.

4.4.3 End User Application Testing

End User (EU) Application is described in detail in Chapter 7 as part of the EU SPL testing framework.

4.4.4 End User Application Deployment

End user application deployment involves end users deploying the derived applications to their smart spaces. During application deployment, EUD environments map the derived application to a set of devices available in the smart space. EUD environments communicate with devices deployed in the smart space and provide them with application instructions. It is the responsibility of the EUD environment to inform the end user if devices that interface with components are not available during application deployment. After derived applications are successfully deployed to the smart space, end users can use the feature-based integration test cases used for EU application testing to test the successful deployment of the application, as described in Chapter 7.

4.5 End User SPL Evolution

As end users derive and deploy applications to their smart spaces they might identify product line defects and new product line requirements that want for their spaces. End users communicate the new requirements and product line defects to EU SPL designers. Similarly EU SPL designers might have new requirements and product line defects identified by internal testing. All defects and new requirements are added to the EU SPL repository. EU SPL designers prioritize, implement and test the new requirements and/or defects using the EUPLE process shown in Figure 4.2 and Figure 4.3. EU SPL updates can be communicated back to end users.

4.6 Summary

This chapter has described the end user software product line process that is used by EU SPL designers to create end user product lines and end users to derive applications for their spaces. The end user product line process consist of two phases: (a) end user product line engineering, and (b) end user application derivation. During end user product line engineering, end users perform product line requirements elicitation, analysis, design, implementation and testing to develop the EU SPL. End user application engineering involves end users selecting the smart space feature requirements they need, application derivation, application testing, and application deployment to the smart space. This chapter described the EU SPL process by providing examples for each phase from the Smart Home case study developed for this research.

5 END USER SOFTWARE PRODUCT LINE META-MODEL FOR SMART SPACES

5.1 Introduction

End User Development (EUD) environments such as Team Computing (TeC) (Sousa et al., 2010), and Jigsaw (Humble et al., 2003) aim to enable end users to create and deploy software applications for their smart spaces. EUD environments connect software applications and devices deployed in the smart space while providing friendly user interfaces for end users to create software applications. End User Software Product Lines (EU SPLs) extend EUD environments with product line support to promote reuse and software application portability. EU SPLs for smart spaces provide a lightweight approach for SPL development while addressing the dynamic nature of these environments.

This chapter describes a meta-modeling approach for developing EU SPLs for smart spaces. The meta-modeling approach provides platform independent and platform specific EU SPL modeling support. A platform independent model is an end user application model that is independent of the platform (EUD environment e.g., Jigsaw/TeC) and the hardware/Operating System (OS). Platform independent modeling involves EU SPL designers creating platform independent models that can be tailored to different EUD environments through an application derivation process. A platform specific model is an end user application model that is specific to an EUD environment e.g., Jigsaw/TeC but independent of the hardware/OS platform. Platform specific modeling involves EU SPL designers creating platform specific models that are bound to specific EUD environments e.g., Jigsaw/TeC. Platform specific models provide an additional capability, since they have access to platform specific functionality that is not available to the platform independent models.

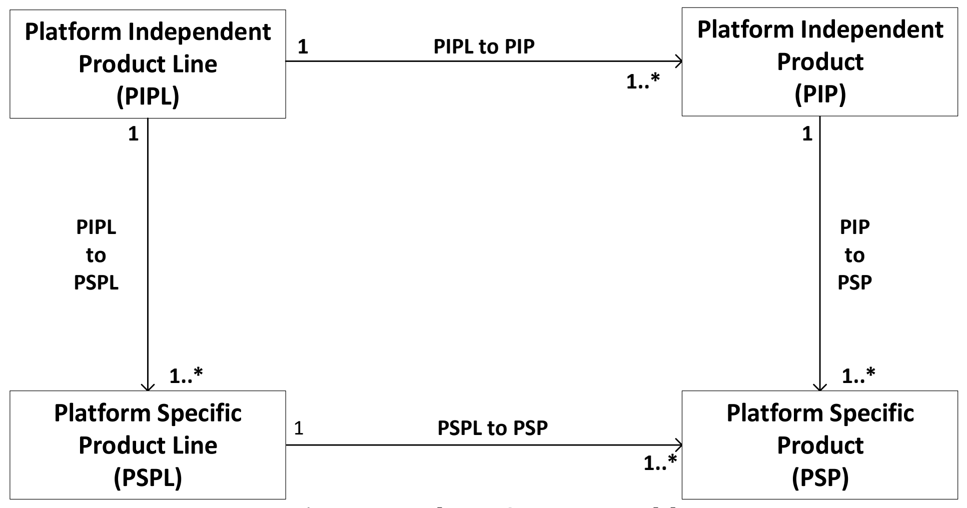
In detail, this chapter presents a meta-model as the basis for developing an EU SPL development environment for creating EU SPLs and deriving End User (EU) applications. The meta-model is composed of platform independent and platform specific meta-models. This chapter describes in detail both parts of the meta-model and discusses the relationships and mappings between them. This chapter is organized as follows. Section 5.2 describes the overall EU SPL meta-modeling approach for smart environments. Section 5.3 presents the platform specific meta-models for the TeC and Jigsaw EUD environments. Section 5.4 introduces the platform independent meta-model. Section 5.5 discusses the mapping of the platform independent meta-model to the TeC and Jigsaw platform specific meta-models. Finally, section 5.6 summarizes this chapter.

5.2 Overview of the EU SPL Meta-model for Smart Spaces

There are several common characteristics across EUD environments for smart spaces. For example all event driven EUD environments consist of components that are abstractions of devices, sensors, actuators, application, services etc. and connections between the components to create application logic. There is also variability between EUD environments. For example some end user environments provide specific functionality to handle user-context, location, and temporal relationships while others do not. To address the commonality and variability of EUD environments and provide a common approach for the development of end user applications for smart spaces, the EU SPL meta-model is designed.

Figure 5.1 shows the EU SPL meta-model for smart spaces. The meta-model consists of platform independent and platform specific meta-models. The platform independent meta-model is composed of the Platform Independent Product Line (PIPL) and the Platform Independent Product (PIP) meta-models. The PIPL meta-model captures the underlying representation of EU SPLs in terms of meta-classes and relationships independent of the platform (EUD environment). The meta-model contains representations of EU SPL features, feature dependencies, and the component architecture that realizes each feature. The component architecture describes the smart space components, connectors and other artefacts that are needed for the feature implementation. The PIP meta-model provides the underlying representation of end user applications in terms of meta-classes and relationships derived from the PIPL meta-model. To derive PIP models end users select product line features from the PIPL model. The components and their relationships that realize the selected features are used to derive the PIP model. Both PIPL and PIP models are platform independent models that can be mapped to different EUD environments e.g., Jigsaw/TeC for smart spaces.

The platform specific meta-model consists of the Platform Specific Product Line (PSPL) and the Platform Specific Product (PSP) meta-models. The PSPL meta-model captures the underlying representation of EU SPLs in terms of meta-classes and relationships specific to platform (EUD environment). The meta-model contains representations of EU SPL features, feature dependencies, and the component architecture


Platform Independent Product Line (PIPL)
Platform Independent Product (PIP)
Platform Specific Product Line (PSPL)
Platform Specific Product (PSP)
PIPL to PIP
PIPL to PSPL
PIP to PSP
PSPL to PSP
1
1..*
Figure 5.1 End User SPL Meta-model

that realizes each feature. The PSPL meta-model is used for creating EU SPL models for specific platforms. PSPL models are derived from PIPL models through meta-class mapping discussed later in this chapter. The PSP meta-model captures the underlying representation of end user application in terms of meta-classes and relationships derived from the PSPL meta-model. As shown in Figure 5.1, PSP models can be derived from PIP models in addition to PSPL models.

There is a one-to-many relationship between the platform independent and the platform specific models. For instance, multiple PSPL models for different platforms can be derived from the PIPL model. EU SPL designers can model platform independent EU SPLs using the PIPL meta-model that can be map to PSPL models for different platforms. Similar multiple PSP models can be derived from the PIP model. End users can derive PIP models that can be mapped to PSP models for different platforms.

The PIPL to PIP and PSPL to PSP model relationships are one-to-many. For instance, multiple PIP models can be derived from one PIPL model. Similar multiple PSP models can be derived from one PSPL model. PSPL and PSP models are bound to a specific EUD platform. For example a PSPL model designed for TeC can derive PSP models that can only be deployed to TeC smart spaces. The following sections of this chapter describe in detail the platform specific and platform independent meta-models.

5.3 Platform Specific Meta-models

This section describes the platform specific meta-models, for the Team Computing (TeC) and Jigsaw end user environments, before describing how they can be integrated into platform independent meta-models in Section 5.4. In particular, the section presents the application meta-models of the (TeC) and Jigsaw end user platforms and explains how they were extended to create platform specific product line meta-models that can be used to create EU SPLs. Furthermore the component mappings of the product line meta-models to the application meta-models needed to derive end user applications from the EU SPL are also discussed for each EUD environment. Section 5.3.1 describes the platform specific meta-models for TeC and section 5.3.2 for the Jigsaw end user environment.

5.3.1 Platform Specific Meta-models for TeC

This section describes the PSP and PSPL meta-models for the TeC end user environment. In particular, section 5.3.1.1 introduces TeC and presents its application (PSP) meta-model, and section 5.3.1.2 explains how the TeC application meta-model was extended to create the TeC PSPL. The TeC PSPL can be used to create EU SPLs for the

TeC platform. Section 5.3.1.3 describes the meta-model component mapping between the TeC PSPL and the TeC PSP meta-models.

5.3.1.1 Platform Specific Product (PSP) for TeC

Team Computing (TeC) is an event driven architectural style that enables end users to design and deploy personalized software for their smart spaces (Sousa et al., 2010). A detailed description of TeC is provided in section 2.4.5 of Chapter 2.

Figure 5.2 shows the application meta-model for TeC. The Team Design meta-class captures information about TeC applications. A Team Design can be deployed to zero or more Locations. The Location meta-class captures location information of a smart space. For example, one Team Design might apply to devices available to the family room of a smart home versus another one that applies to the entire house. A Team Design is realized by one or more Activity Sheets. The Activity Sheet meta-class represents software components, devices, and humans operating in ubiquitous computing environments. Activity Sheets have zero or more Inputs and Outputs. The Input meta-class contains information about the Activity Sheet required interfaces for receiving data. The Output meta-class contains information about the Activity Sheet provided interfaces for sending data. Outputs are bound by triggering conditions that when evaluated to true causes the output to be send to the destination. In TeC, device connectivity can be achieved by having outputs from one Activity Sheet send to inputs of another Activity Sheet. Inputs and Outputs can contain zero or more Payloads. The Payload meta-class contains information

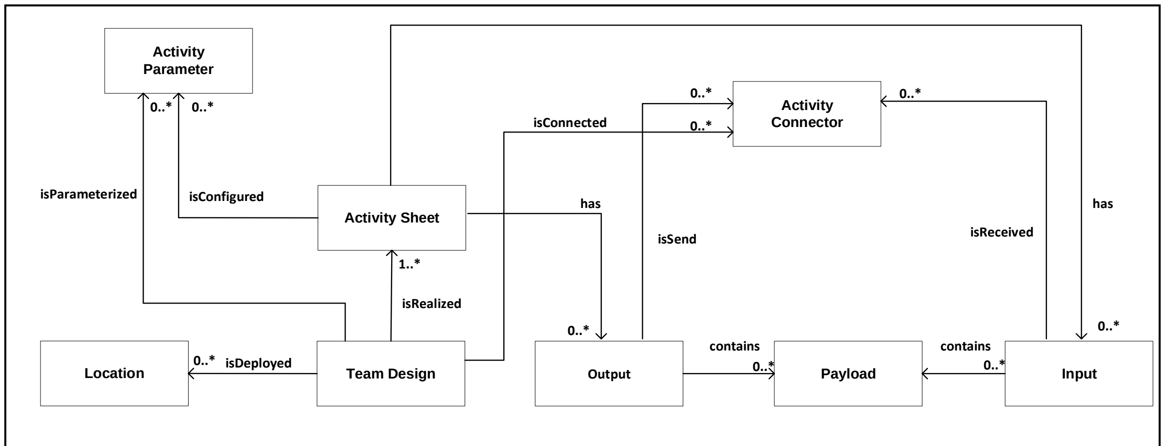
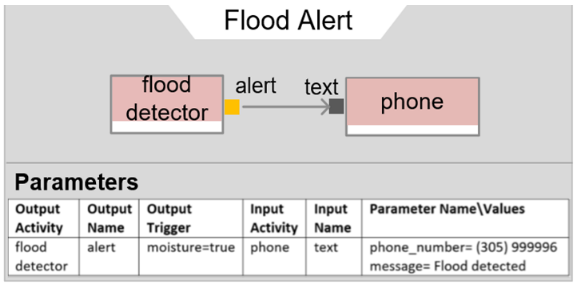


Figure 5.2 TeC Application Meta-model (PSP)

in the form of key-value pairs about the data send from Outputs to Inputs. The Activity Connector meta-class contains information about the Activity Sheet's connectivity within a Team Design. Outputs send data to zero or more Activity Connectors and Inputs receive data from zero or more Activity Connectors.

Figure 5.3 shows the Team Design of a "Flood Alert" TeC application deployed in a smart home. The purpose of the application is to send text alerts to the home residents if a flood is detected. The "Flood Alert" Team Design is realized of a flood detector and a phone TeC Activity Sheets. The flood detector Activity Sheet represents moisture sensors deployed in the smart home, and the phone Activity Sheet a house phone that supports landline messaging. The flood detector Activity Sheet has an Output called "alert" that sends flood notifications to the "text" Input of the phone Activity Sheet. The Activity Connector meta-class for the Team Design contains information about the connection of the "alert" Output and the "text" Input. The "alert" output has a triggering condition that is evaluated to true when the flood detector detects moisture. When moisture is detected,



Output Activity	Output Name	Output Trigger	Input Activity	Input Name	Parameter Name\Values
flood detector	alert	moisture=true	phone	text	phone_number= (305) 999996 message= Flood detected

Figure 5.3 Flood Alert – TeC Team

"alert" sends one message with two Payloads in the form of key-value pairs to the "text" input. The keys of the payloads are phone_number and message. The phone Activity Sheet will interpret the phone_number payload value as the number to text and the message payload value as the contents of the message to send. An Activity Sheet is configured by zero or more Activity Parameters. The Activity Parameter meta-class captures information about configurable internal parameters of Activity Sheets. An example of an Activity Parameter in the "Flood Alert" example are the moisture threshold values for the flood detector Activity Sheet. When the moisture threshold values are exceeded then the sensor can report moisture.

5.3.1.2 Platform Specific Product Line (PSPL) for TeC

To extend TeC with product line support, the TeC PSPL was created. The TeC PSPL is used to derive applications for different TeC environments. In particular, The TeC PSP model was extended with product line support to create the TeC PSPL meta-model shown in Figure 5.4. The objective of the TeC PSPL meta-model is to derive multiple TeC PSP models from one TeC PSPL model. The TeC PSPL meta-model is composed of the Feature and the TeC Product Line (PL) Component meta-models. The Feature meta-model

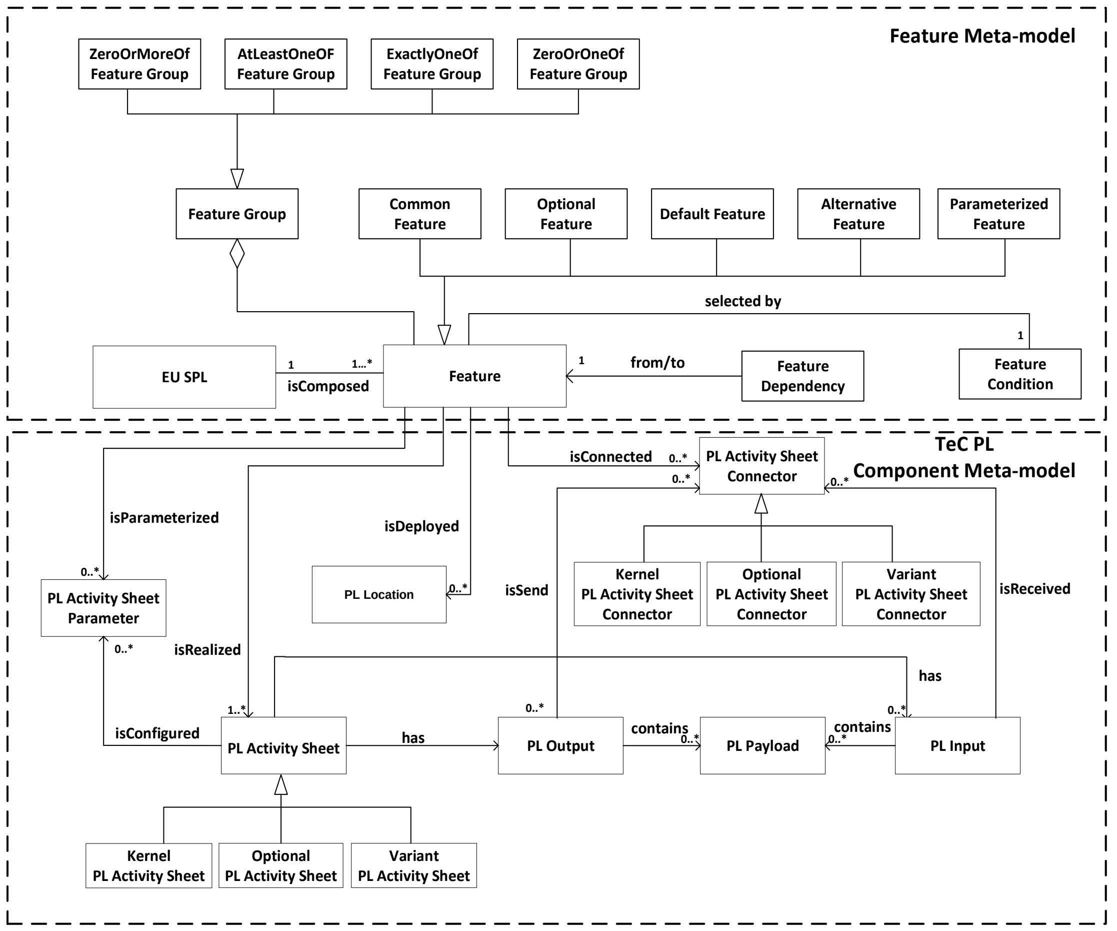


Figure 5.4 TeC Platform Specific Product Line (PSPL) Meta-model

is platform independent and describes the EU SPL and feature relationships. The TeC Product Line Component meta-model is specific to the TeC platform and describes the relationships between product line features and the TeC Product Line (PL) component architecture that realizes each feature. The TeC PL meta-model extends the TeC meta-model with product line support. The remainder of this section describes the meta-model in detail.

As shown in Figure 5.4, an EU SPL is composed of one or more features. Each Feature describes a specific functionality that the EU SPL supports. Features can be common, optional, alternative, default or parameterized. Common are features that exist in all products derived from the product line. Optional features are features that can be found in only some products derived from the product line. Alternative features are features that are mutually exclusive. Default features are one of a group of alternative features that the EU SPL designer has pre-selected for product derivation. Parameterized features are features that can be parameterized by end users during application derivation.

Features can belong to feature groups. Feature groups can be thought as a set of features that share a common set of constraints. There are four types of feature groups: (1) ZeroOrMoreOf, (2) AtLeastOneOf, (3) ExactlyOneOf, and (4) ZeroOrOne. ZeroOrMoreOf is a feature group from which zero or more features can be selected. AtLeastOneOf is a feature group from which more than one feature must be selected. ExactlyoneOf is a feature group from which only one feature can be selected. ZeroOrOne is a feature group from which either no feature or one feature can be selected. Features can be dependent on other features. For example consider three features {A}, {B}, {C} and that $\{C\} \rightarrow \{A\} \wedge \{B\}$, this implies that feature {C} cannot exist if features {A} and {B} do not exist. The Feature Dependency meta-class captures the dependency among features. Feature conditions are an alternative way for expressing feature selection.

Features are realized by one or more PL Activity Sheets and are connected to zero or more PL Activity Connectors. PL Activity Sheets can be kernel, optional or variant. Kernel PL Activity Sheets are available to all PSPs derived from the PSPL. Optional PL Activity Sheets are available to only some derived PSPs. Variant PL Activity Sheets are mutually exclusive PL Activity Sheets. PL Activity Sheets can have zero or more PL Inputs and PL Outputs. PL Inputs and PL Outputs can have zero or more PL Payloads. PL Activity Connectors can also be kernel, optional or variant. A feature is parameterized by zero or more PL Activity Parameters. Finally, a feature is deployed in zero or more PL Locations. For example a product line feature can be applicable to components in a specific location of the smart space. The PL Location meta-class captures the location info.

5.3.1.3 TeC PSPL to PSP Meta-model Mappings

To derive end user applications from the TeC EU SPL, the selected features and components of the TeC PSPL need to be mapped to the features and components of the TeC PSP model. Figure 5.5 shows the high-level meta-class mappings between the TeC PSPL and the TeC PSP meta-models.

In detail, each PL Activity Sheet in the PSPL model is mapped to an Activity Sheet in the PSP model. Similar each PL Activity Connector in the PSPL component model will be mapped to an Activity Connector in the PSP model. PL Activity Parameters will be mapped to Activity Parameters and PL Locations to Locations meta-classes in the TeC component model. The PL Inputs, PL Outputs and PL Payload meta-classes from the TeC PSPL model are mapped to Input, Outputs and Payload meta-classes in the TeC PSP.

5.3.2 Platform Specific Meta-models for Jigsaw

This section describes the PSP and PSPL meta-models for the Jigsaw end user environment. In particular, section 5.3.2.1 introduces Jigsaw and presents its application (PSP) meta-model, section 5.3.2.2 discusses how the Jigsaw application meta-model was extended to create the Jigsaw PSPL. The Jigsaw PSPL can be used to create EU SPLs for

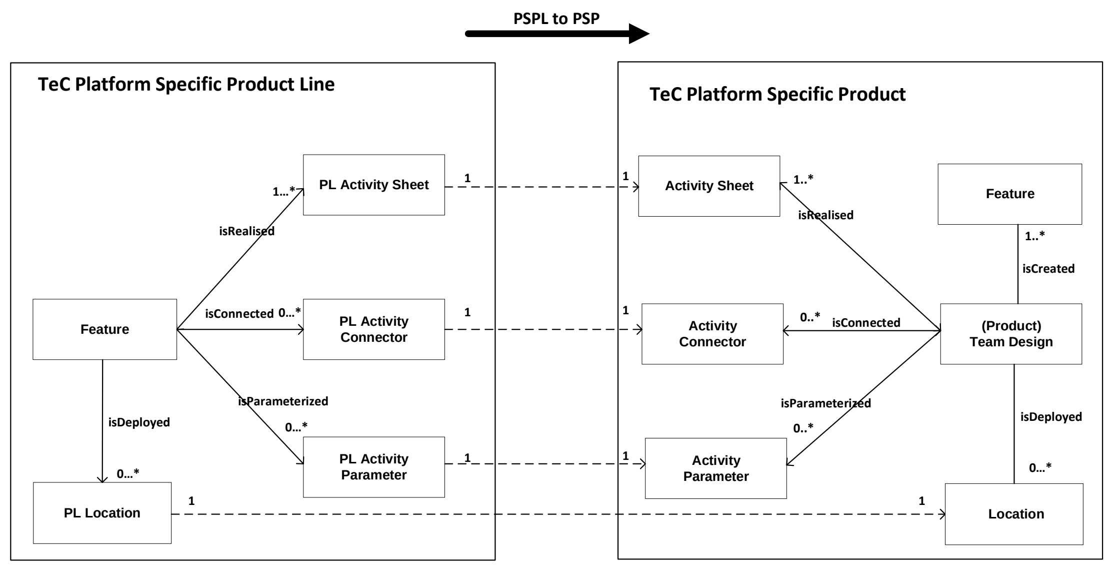


Figure 5.5 PSPL to PSP Meta-model Mapping for the TeC Platform

the Jigsaw platform. Finally, section 5.3.2.3 describes the meta-model mapping between the Jigsaw PSPL and the Jigsaw PSP meta-models

5.3.2.1 Platform Specific Product (PSP) for Jigsaw

Jigsaw (Humble et al., 2003) is an EUD environment that enables end users to configure devices, applications and services available to their smart space through a puzzle like user interface. Figure 5.6 shows the Jigsaw meta-model that was developed as part of this research. In detail, a Jigsaw Puzzle is realized by one or more Jigsaw Pieces. Each Jigsaw Piece represents a device in the smart space. Examples of Jigsaw Pieces are a phone, a doorbell, a camera etc. Each Jigsaw Piece can have zero or more Jigsaw Piece Parameters. Jigsaw Piece Parameters represent device configuration parameters. For example a

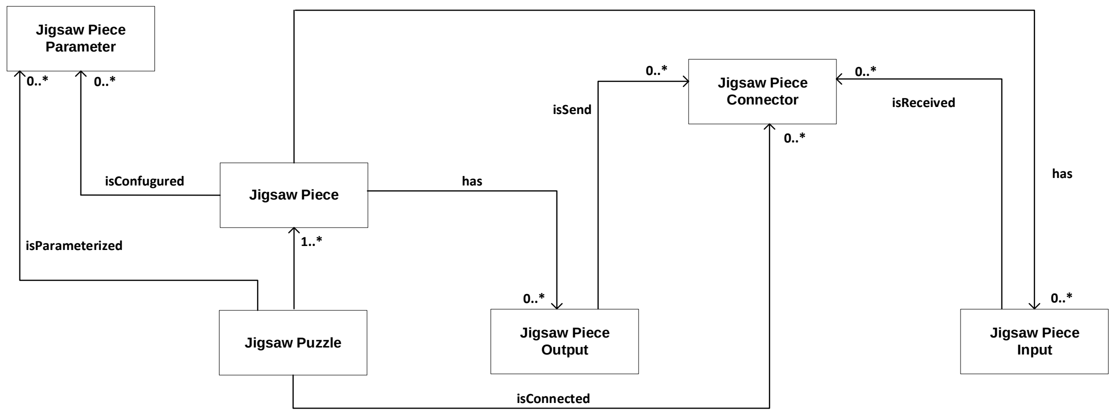


Figure 5.6 Jigsaw Application Meta-model (PSP)

doorbell device might have different ring tones, a photo camera can have different light settings and so on. Jigsaw Pieces have zero or more Jigsaw Piece Inputs and Jigsaw Piece Outputs. Jigsaw Piece Inputs capture device inputs and Jigsaw Piece Outputs capture device outputs. The Jigsaw Piece Connector meta-class captures the connectivity of Jigsaw Pieces. In particular, the Jigsaw Piece Output of one Jigsaw Piece can be connected to the Jigsaw Piece Input of another Jigsaw Piece. The Jigsaw Piece Output to Jigsaw Piece Input relationship is captured by the Jigsaw Piece Connector meta-class. A Jigsaw Puzzle is connected by zero or more Jigsaw Piece Connectors.

Figure 5.7 shows an example of a doorbell application using Jigsaw. The purpose of the application is when a person rings the doorbell, the camera takes a picture and send it to the resident smart phone. To create this application a Jigsaw Puzzle is created. The Jigsaw Pieces of the puzzle are “Door Bell”, “Camera” and “Phone.” The “Door Bell” Jigsaw Piece represents the house door bell device, the “Camera” Jigsaw Piece represents a webcam device installed in the entrance and the “Phone” Jigsaw Piece represents the

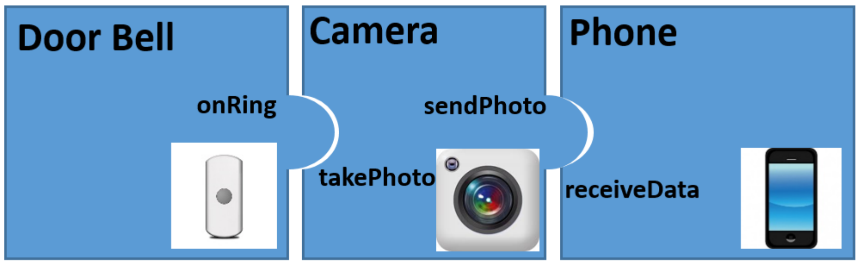


Figure 5.7 Jigsaw Doorbell Application Example

resident's smart phone. The "onRing" output of the "Door Bell" piece captures the event of a person ringing the doorbell. The "onRing" output is connected to the "takePhoto" input of the "Camera" Jigsaw Piece instructing the camera to take a picture. The "onRing" to the "takePhoto" connectivity information is captured by the Jigsaw Activity Connector meta-class. Finally the "sendPhoto" output of the "Camera" Jigsaw Piece is connected to the "receiveData" input of the "Phone" Jigsaw Piece to indicate that the picture taken by the camera needs to be send to the phone.

5.3.2.2 Platform Specific Product Line (PSPL) for Jigsaw

To extend Jigsaw with product line support the Jigsaw PSPL was created. The Jigsaw PSPL is used to derive applications for different Jigsaw environments. In particular, the Jigsaw PSP meta-model was extended with product line support to create the Jigsaw PSPL meta-model. The Jigsaw PSPL meta-model consists of the Feature meta-model and the PL Jigsaw Component meta-model. The Feature meta-model part of the Jigsaw PSPL meta-model is the same as the TeC PSPL shown on Figure 5.4. Figure 5.8 shows the PL Jigsaw Component meta-model part of the Jigsaw PSPL meta-model.

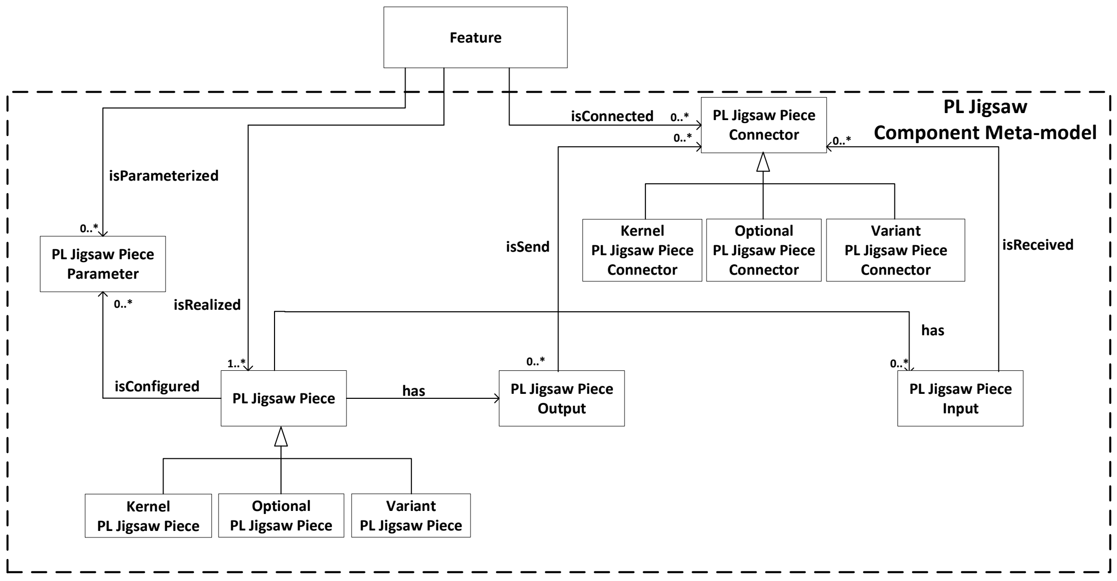


Figure 5.8 PL Jigsaw Component Meta-model

The main meta-classes of the PL Jigsaw Component meta-model are the PL Jigsaw Piece, PL Jigsaw Piece Parameter, PL Jigsaw Piece Connector, PL Jigsaw Input and PL Jigsaw Output. The PL Jigsaw Piece abstracts Jigsaw Pieces that represent different devices in a smart space. PL Jigsaw Pieces can be kernel, optional or variant product line components. A PL Jigsaw Piece is configured by zero or more PL Jigsaw Piece Parameters. PL Jigsaw Piece Parameters contain configuration parameters for the PL Jigsaw Piece. PL Jigsaw Pieces can have zero or more PL Jigsaw Piece Inputs and PL Jigsaw Piece Outputs. The PL Jigsaw Piece Input meta-class contains information about the PL Jigsaw Piece required interfaces and the PL Jigsaw Piece Output meta-class contains information about PL Jigsaw Piece provided interfaces. PL Jigsaw Piece Connector meta-class contains information about the PL Jigsaw Piece’s connectors within a product line feature. PL Jigsaw Piece Outputs send data to zero or more PL Jigsaw Piece Connector Connectors and PL Jigsaw Piece Inputs receive data from zero or more Activity Connectors. PL Jigsaw Piece Connector can be kernel, optional or variant. A product line feature is realized from one or more PL Jigsaw Pieces, is parameterized by zero or more PL Jigsaw Piece Parameters and is connected to zero or more PL Jigsaw Piece Connectors. During application derivation PL Jigsaw meta-classes are mapped to the Jigsaw meta-classes.

5.3.2.3 Jigsaw PSPL to PSP Meta-model Mappings

Figure 5.9 shows the Jigsaw PSPL to PSP high-level meta-class mappings needed to derive Jigsaw end user applications. In detail, all components that realize end user selected features are derived from the Jigsaw PSPL. The derived PSPL components are mapped to PSP components models during application derivation.

As shown in Figure 5.9 each PL Jigsaw Piece in the PSPL model is mapped to one Jigsaw Piece in the PSP model. Similar each PL Jigsaw Piece Connector in the PSPL component model will be mapped to a Jigsaw Piece Connector in the PSP model. PL Jigsaw Piece Parameters are mapped to Jigsaw Piece Parameters. The PL Jigsaw Piece Inputs and PL Jigsaw Piece Outputs are mapped to Jigsaw Piece Input and Jigsaw Piece Outputs.

5.4 Platform Independent Meta-models

In order to develop end user applications that do not depend on any particular EUD environment, the PSPL and PSP meta-models were extended to create the Platform Independent Product Line (PIPL) and the Platform Independent Product (PIP) meta-

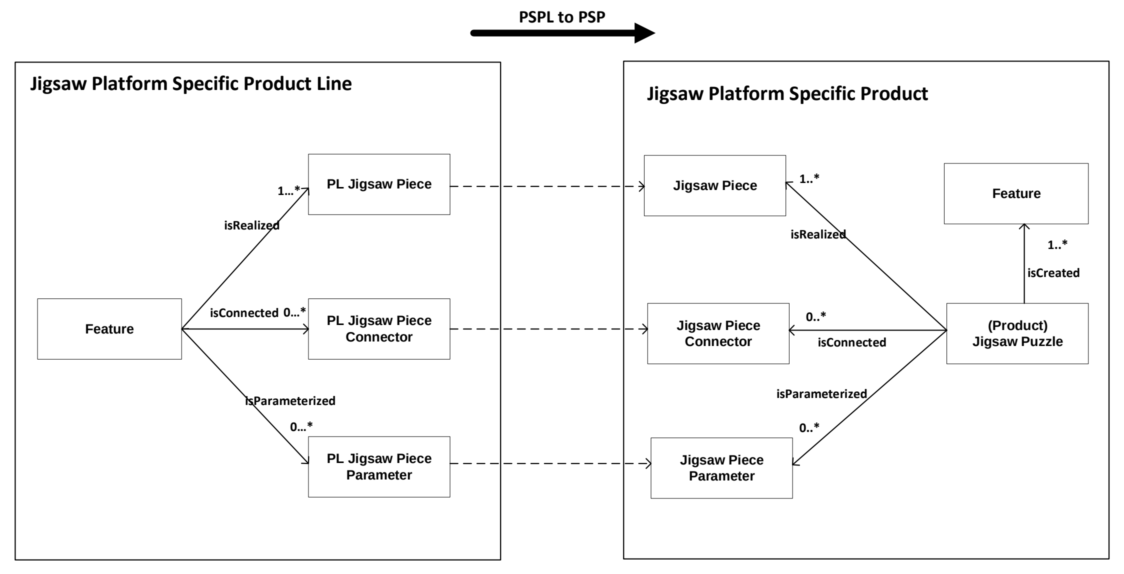


Figure 5.9 PSPL to PSP Mapping for the Jigsaw Platform

models. The platform independent models apply to all EUD environments that support a component and connector architecture.

5.4.1 Platform Independent Product Line (PIPL)

Similar to the PSPL, the PIPL meta-model consists of the Feature and the Component meta-models. The Feature meta-model is the same as the PSPL shown on Figure 5.4. The Component meta-model is designed to support common component connector functionality across different EUD environments.

Figure 5.10 shows the PIPL component meta-model. In detail, each feature in the PIPL is realized by one or more PL Components, is connected by zero or more PL Component Connectors, and is parameterized by zero or more PL Component Parameters. PL Components are similar to PL Activity Sheets in the TeC PSPL and PL Jigsaw Pieces in the Jigsaw PSPL. PL Components represent software applications and devices that are

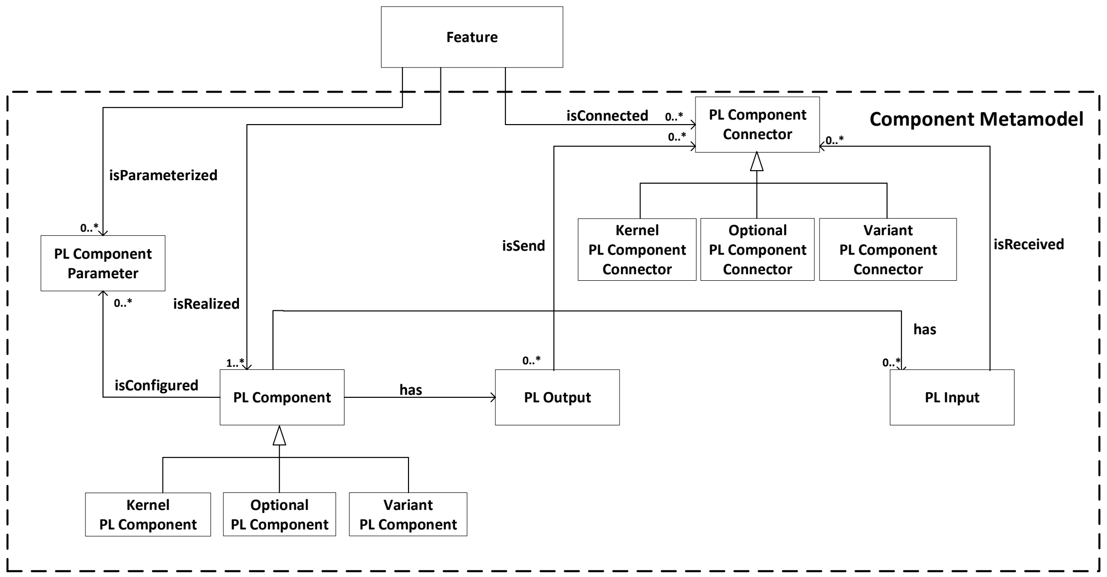


Figure 5.10 Platform Independent Product Line (PIPL) Meta-model

part of the smart space. PL Components can be kernel, optional or variant and they have zero or more PL Inputs and PL Outputs. The PL Input meta-class contains information about the PL Component required interfaces and the PL Output meta-class about the PL Component provided interfaces. PL Component Connectors indicate the way PL Components within a product line feature are connected. For instance, PL Outputs of one PL Component can be connected to PL Inputs of another PL Component. PL Inputs send data to zero or more PL Component Connectors and PL Inputs receive data from zero or more PL Component Connectors. PL Component Connectors can be kernel, optional or variant. Finally, PL Components are configured by zero or more PL Component Parameters.

5.4.2 Platform Independent Product (PIP)

The Platform Independent Product (PIP) meta-model provides the underlying representation of end user applications in terms of meta-classes and relationships, which are derived from the PIPL meta-model. Figure 5.11 shows the PIP meta-model. End user applications in the PIP meta-model are represented by the Product meta-class. A Product in the PIP meta-model is composed of one or more Components. Components represent meta-classes of the smart space (devices, applications, sensors, etc.). Components of a product are connected by zero or more Component Connectors. Components can have zero or more Inputs to receive data and zero or more outputs to send data. The Component connector meta-class contains information about interconnecting Component Outputs and Component Inputs. Finally a Product in the PIP meta-model is parameterized by zero or more Component Parameters.

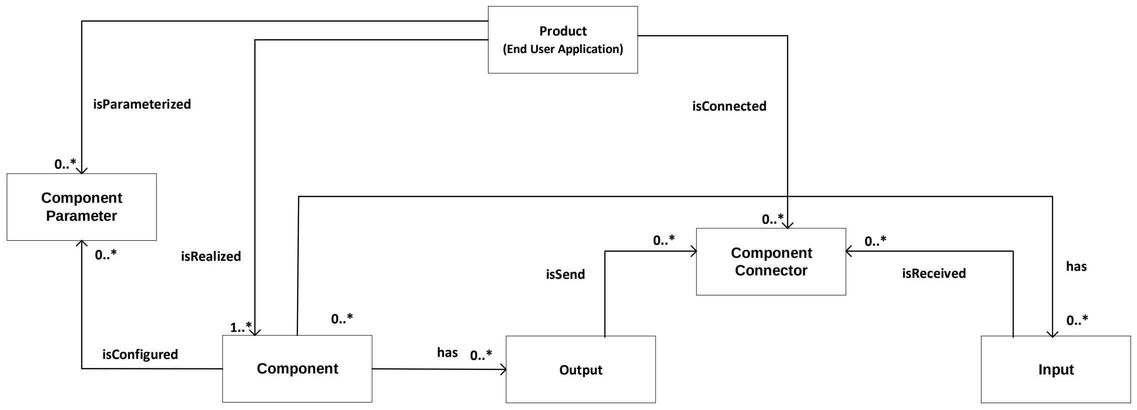


Figure 5.11 Platform Independent Product (PIP) Meta-model

5.4.3 PIPL to PIP Meta-model Mappings

Figure 5.12 shows the PIPL to PIP meta-model component mappings needed to derive end user applications from the product line. Similar to the PSPL to PSP meta-model mappings, the components of the selected features are derived from the PIPL model. The PIPL components are mapped to PIP models following the mappings shown in Figure 5.12. In detail, PL Components that are part of each feature are mapped to Components in the PIP model, PL Component Connectors are mapped to Component Connectors and PL Component Parameters are mapped to Component Parameters in the PIP model. PL Inputs and PL Outputs are mapped to the Input and Output meta-classes in the PIP model.

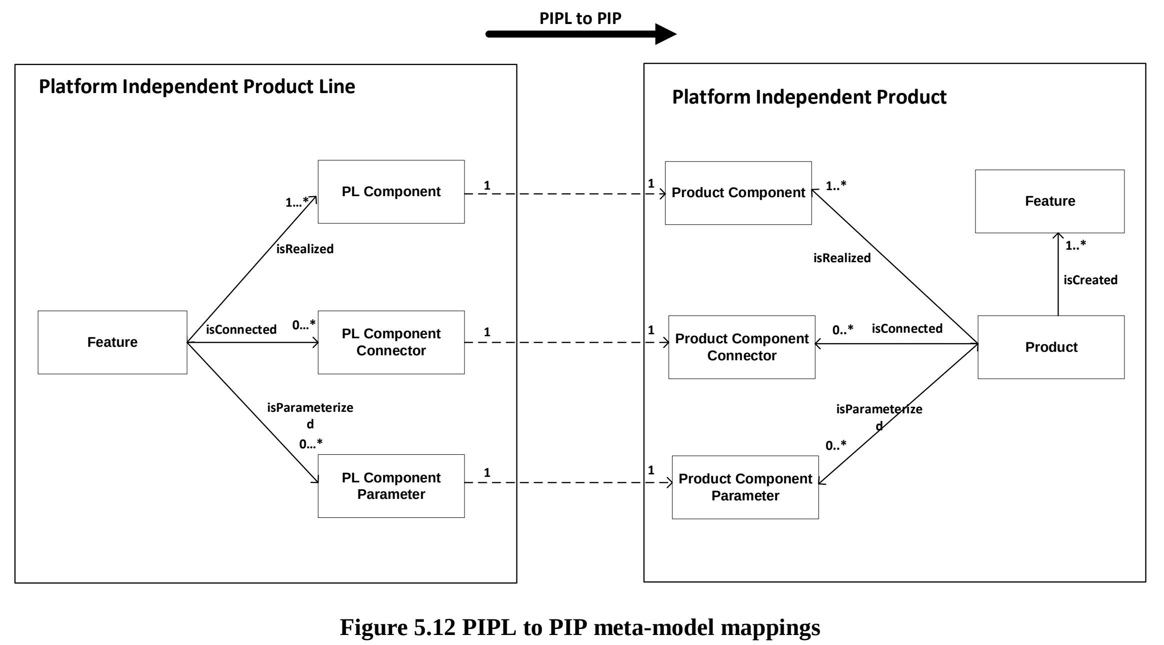


Figure 5.12 PIPL to PIP meta-model mappings

5.5 Platform Independent to Platform Specific Mappings

Platform independent models need to be mapped to platform specific models in order to be deployed to a specific end user environment. This section describes the component meta-class mappings between platform independent and platform specific meta-models. Figure 5.13 shows the platform independent to platform specific mappings in the EU SPL meta-model.

The PIPL to PSPL meta-model mapping shown in Figure 5.13 enables EU SPL designers to develop product lines that can be mapped to EU SPLs for different EUD environments. Another benefit of the PIPL to PSPL meta-model mapping is that EU SPL designers can develop basic EU SPL functionality as platform independent models, and map the EU SPL to a platform specific model. At the platform specific layer the EU SPL designers can extend the EU SPL with platform specific functionality. This allows EU SPL designers to reuse and extend EU SPL models across different platforms. The following

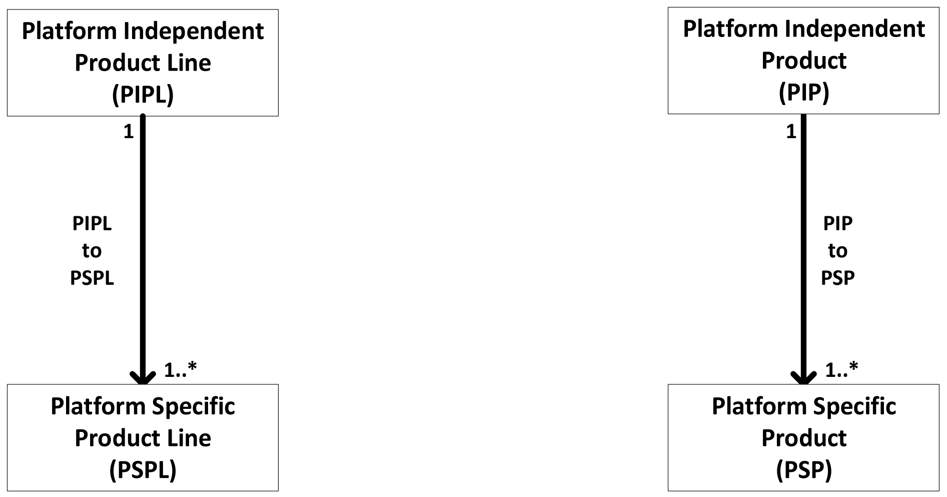

Platform Independent Product Line (PIPL)
1
PIPL to PSPL
1..*
Platform Specific Product Line (PSPL)
Platform Independent Product (PIP)
1
PIP to PSP
1..*
Platform Specific Product (PSP)

Figure 5.13 Platform Independent to Platform Specific Mappings

sections describes the PIPL to PSPL meta-model mappings for TeC and Jigsaw EUD environments. The PIP to PSP meta-model mapping enables end users to derive platform independent application models from the PIPL that are then mapped to specific end user environments. This section describes the PIP to PSP meta-model mappings for the TeC and Jigsaw architectures.

5.5.1 PIPL to TeC PSPL Meta-model Mappings

Figure 5.14 shows the component mapping of the PIPL meta-model to the TeC PSPL meta-model. The component mapping can be used for converting platform independent product line models to TeC platform specific product line models. In detail, there is a one-to-one relationship between PL Components in the PIPL meta-model and the PL Activity sheets in the TeC PSPL meta-model. Similarly, there is a one-to-one relationship between PL Component Connectors and PL Activity Connectors in TeC PSPL,

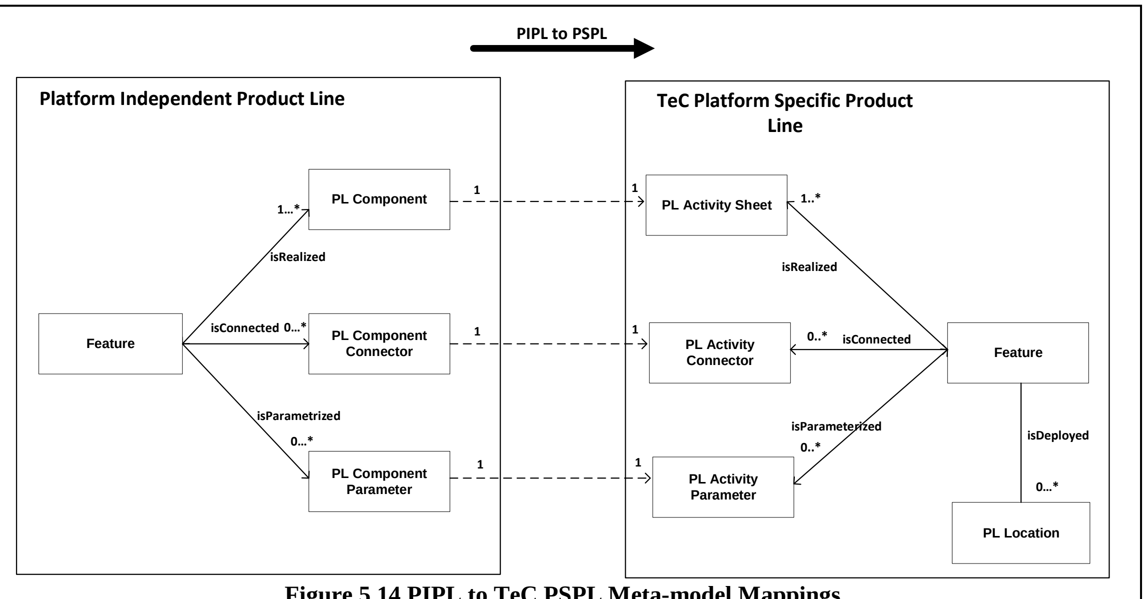


Figure 5.14 PIPL to TeC PSPL Meta-model Mappings

and between PL Component Parameters in PIPL and PL Activity Parameters. PL Inputs and PL Outputs in the in the PIPL meta-model are mapped to PL Inputs and PL Outputs in the TeC PSPL meta-model. The PL Payload and PL Location are specific meta-classes of TeC and do not map to PIPL. The PIPL meta-model was not extended with the PL Payload and PL Location meta-classes because the PIPL to PSPL model mapping will not be successful for EUD environments that do not support these meta-classes.

5.5.2 PIPL to Jigsaw PSPL Meta-model Mappings

Figure 5.15 shows the component mapping of the PIPL to the Jigsaw PSPL meta-model. The component mapping can be used for converting platform independent product line models to Jigsaw platform specific product line models. In particular, there is a one-to-one relationship between PL Components in the PIPL and PL Jigsaw Pieces in the Jigsaw PSPL. There is also a one-to-one relationship between PL Component Connector meta-classes and PL Jigsaw Piece Connectors, and between PL Component Parameters and PL Jigsaw Piece Parameters. PL Inputs, PL Outputs in the PIPL meta-model are mapped to PL Inputs and PL Outputs in the Jigsaw PSPL meta-model.

5.5.3 PIP to TeC PSP Meta-model Mappings

Figure 5.16 shows the platform independent product to TeC platform specific products component meta-model mappings. The component mapping can be used for converting PIP models to TeC PSP models. In detail, there is a one-to-one relationship

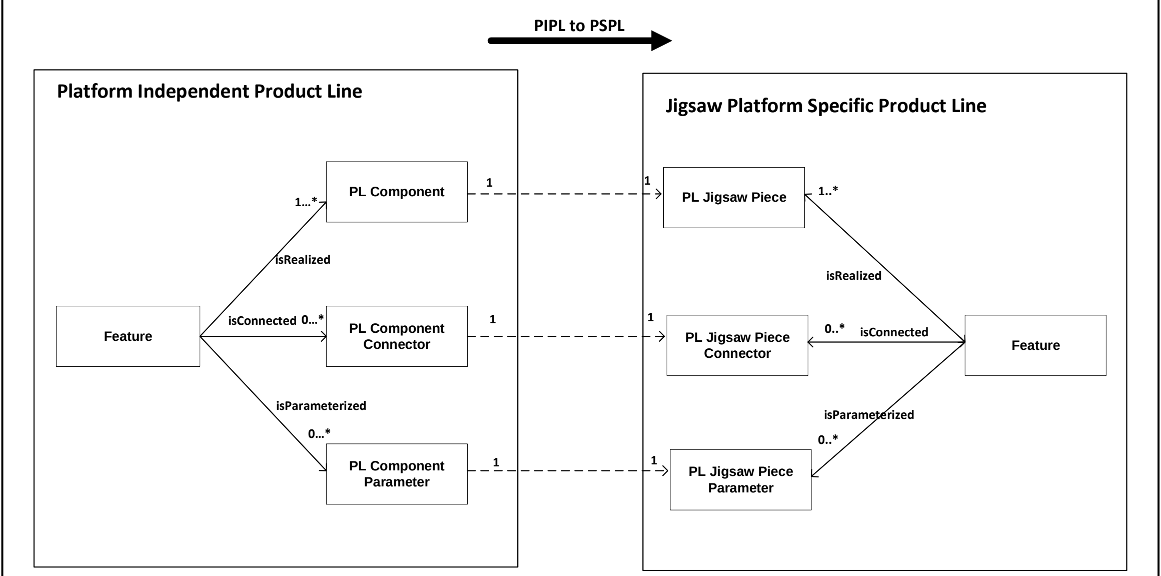


Figure 5.15 PIPL to Jigsaw PSPL Meta-model Mappings

between a Product in the PIP meta-model and a Team Design in TeC. Both the Product and Team Design meta-classes represent end user applications. There is also a one-to-one relationship between Components in PIP and Activity Sheets in the TeC PSP meta-model. Similar there is a one-to-one relationship between Component Connectors and Activity Connectors, and between Component Parameters and Activity Parameters. There is also a one-to-one mapping between Inputs and Outputs in the PIP model and the corresponding Inputs and Outputs in the TeC PSP meta-model. The Payload and Location meta-classes are specific to TeC and there is no mapping to the PIP model.

5.5.4 PIP to Jigsaw PSP Meta-model Mappings

Figure 5.17 shows the platform independent product to Jigsaw platform specific product meta-model mappings. The component mapping can be used for converting PIP models to Jigsaw PSP models. In detail, there is a one-to-one relationship a Product in the

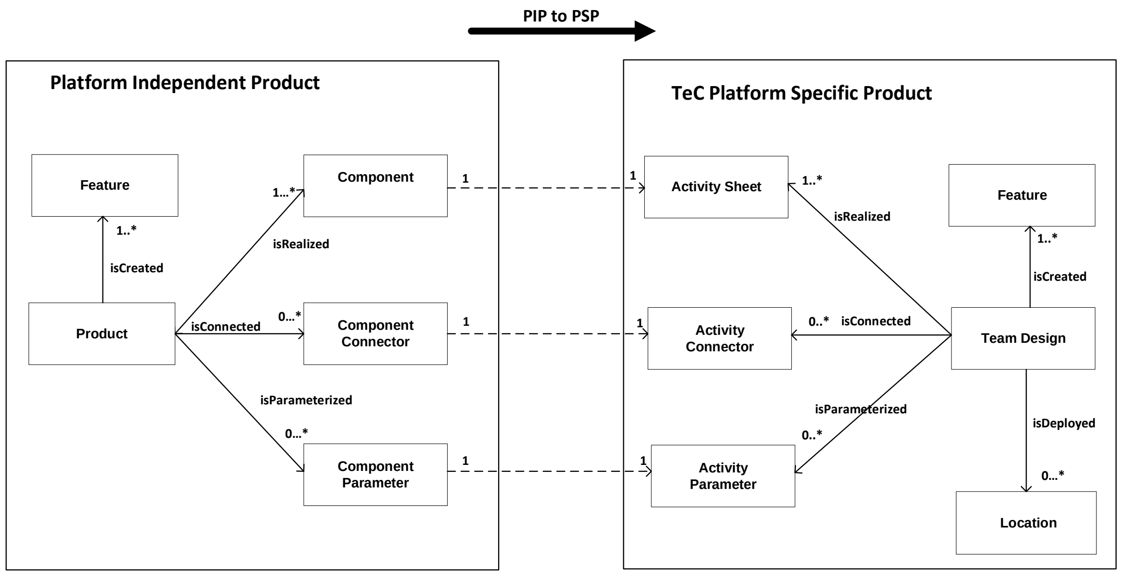


Figure 5.16 PIP to PSP Mapping for the TeC EU Platform

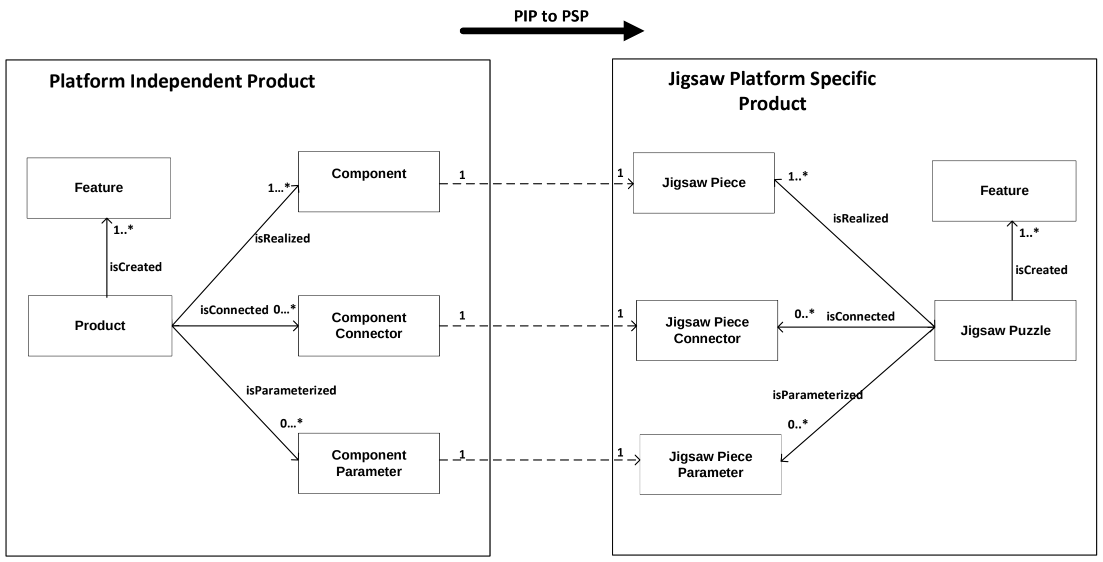


Figure 5.17 PIP to PSP Mapping for the Jigsaw EU Platform

PIP meta-model with a Jigsaw Puzzle in Jigsaw. Both the Product and Jigsaw Puzzle meta-classes represent end user applications. There is also a one-to-one relationship between Components in PIP with Jigsaw Pieces in the Jigsaw PSP meta-model. Similar there is a one-to-one relationship between Component Connectors with Jigsaw Piece Connectors and Component Parameters with Jigsaw Piece Parameters. Finally, there is a one-to-one mapping between Inputs and Outputs in the PIP model and the corresponding Inputs and Outputs in the Jigsaw PSP meta-model.

5.6 Summary

As EUD environments for smart spaces expand, end users will be faced with the challenge of having to develop the same type of applications for different environments. EU SPLs for smart spaces enables end users to derive software applications for their individual spaces. This chapter described the EU SPL meta-model for creating end user product lines. The EU SPL meta-model consists of platform independent and platform specific meta-models. The platform specific meta-models were discussed in context of the TeC and Jigsaw end user environments. The platform independent meta-model is an abstract meta-model for creating product lines for end user environments that supports component and connector architecture. The chapter also presented the meta-model mappings between platform independent and platform specific meta-models to indicate the way platform independent models can be mapped to specific end user environments.

6 END USER SOFTWARE PRODUCT LINE PROTOTYPE (EUSPLP) DEVELOPMENT ENVIRONMENT

6.1 Introduction

This chapter describes the End User Software Product Line Prototype (EUSPLP) development environment created to validate this research. The EUSPLP environment was designed to support end users and extend End User Development (EUD) environments for smart spaces with product line support. The environment provides end user oriented interfaces to enable EU SPL designers to develop End User (EU) Software Product Lines (SPL) and end users to derive applications. The EUSPLP environment was created using the EU SPL process and the EU SPL meta-models described in Chapters 4, and Chapter 5 respectively. In addition, the EUSPLP environment was used to implement the Smart Home EU SPL case study described in the Appendix and to derive end user applications for the TeC EUD environment.

The chapter is organized as follows: Section 6.2 describes the system use cases that the EUSPLP implements. Section 6.3 discusses the overall EUSPLP system architecture. Section 6.4 provides an overview of the physical meta-models developed for the EUSPLP to represent EU SPLs for smart spaces and derived applications. Section 6.5 describes in detail the EUSPLP EU SPL Development subsystem used to develop product lines. Section 6.6 discusses the EUSPLP Application Derivation subsystem used to derive applications from the EU SPL. Section 6.7 describes the process for deploying EU SPL derived applications to the Team Computing (TeC) environment. Finally section 6.8 provides a summary of this chapter.

6.2 EUSPLP System Use Cases

There are five main use cases shown in Figure 6.1 that the EUSPLP development environment supports: (1) Manage EU SPL, (2) Create EU SPL, (3) Edit EU SPL, (4) Derive EU SPL Application, and (5) Import Derived Application. The use cases have two main actors that interact with the EUSPLP environment: (1) the EU SPL designer, and (2) the End user. The EU SPL designer is an advanced end user or domain expert who creates and maintains the EU SPL. The end user derives an EU SPL application and initiates the application deployment process to the EUD environment.

The Manage EU SPL use case captures the interactions between the EU SPL designer and the EUSPLP environment to create and maintain the EU SPL. In particular, during this interaction, EU SPL designers create product line features and develop the component architectures to realize them. The EUSPLP environment stores the EU SPLs created by EU SPL designers for application derivation. The Manage EU SPL use case includes the Create EU SPL and Edit EU SPL use cases.

The Derive EU SPL Application use case captures the interactions between the End user and the EUSPLP environment to derive applications from the EU SPL. In particular, during application derivation, end users select and configure the EU SPL features needed for their spaces. The EUSPLP environment, based on the end user selections, extracts the components and component connectors for the selected features and generates the derived application architecture.

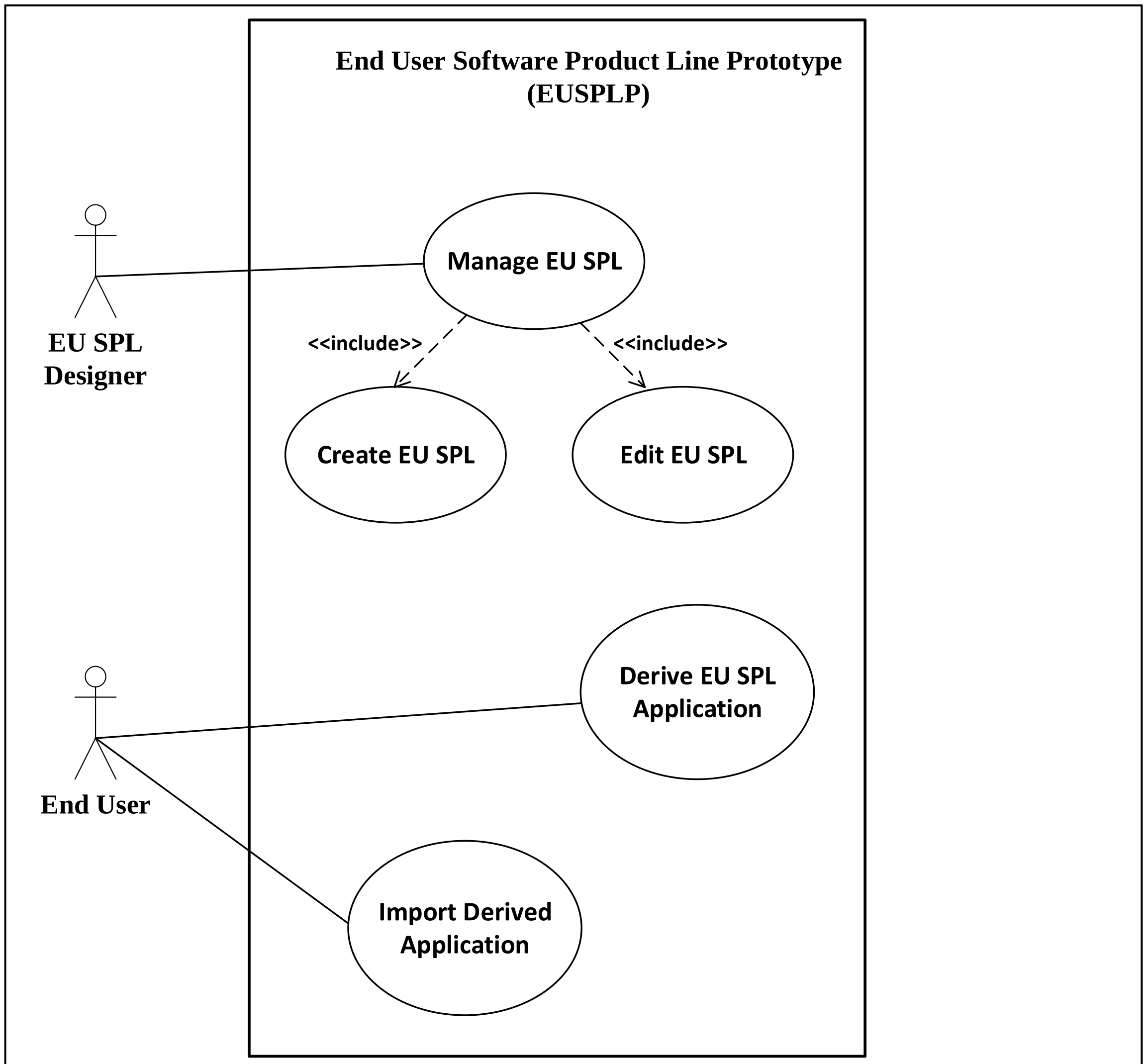

End User Software Product Line Prototype (EUSPLP)
EU SPL Designer
Manage EU SPL
<<include>>
<<include>>
Create EU SPL
Edit EU SPL
End User
Derive EU SPL Application
Import Derived Application

Figure 6.1 EUSPLP Use Cases

The Import Derived Application use case captures the interactions between the End User and the EUSPLP environment to import a derived application to the end user environment. In particular, the end user imports and stores the derived application from the EUSPLP to the EUD environment. End users interact with EUD environment to deploy the end user application to the smart space.

6.3 EUSPLP System Architecture

The EUSPLP development environment was created in Java and is packaged to be deployed in any compatible Java Platform Enterprise Edition (Brock et al., 2014) (Java EE) application server implementing the Java Servlet, Java Server Pages and Java Expression Language specifications. In this research, EUSPLP was deployed in the Java EE Apache Tomcat server that implements the required specifications. The reasons that motivated the selection of Java and the Java EE platform were that the technologies are open source, portable and provide web support.

The open source characteristics makes the EUSPLP environment to not depend on any proprietary technologies. The EUSPLP can be deployed in any operating system that runs Java. The Java EE web support is another feature that the EUSPLP prototype utilizes. The EUSPLP user interface is written using HyperText Markup Language (HTML) (Pilgrim, 2010), and JavaScript (Duckett, 2014) technologies. EU SPL designers and end users, interface with the EUSPLP using web browsers. The EUSPLP user interface communicates with the EUSPLP server using Representational State Transfer (REST) services (Richardson and Ruby, 2007) over the HyperText Transfer Protocol (HTTP) (Totty et al., 2002). REST is a client-server architecture which uses the HTTP protocol. REST services are represented as different URIs in the server that represent different resources. HTTP methods (POST, GET, PUT, DELETE) are used to create, update, modify and delete server resources. JavaScript Object Notation (JSON) (Taylor, 2014) is used as the format for messages exchanged between the EUSPLP client (user interface) and EUSPLP server. JSON is a lightweight human readable data format. Data in JSON are represented as nested key-value pairs. JSON is an alternative format to XML. XML uses a rich markup language for data representation versus JSON that uses a simpler representation. The JSON format is common across JavaScript frameworks used for asynchronous browser / server communication. Similar TeC is communicating with the EUSPLP using REST services and JSON messages over HTTP.

Figure 6.2 shows the EUSPLP subsystem architecture and processes. The EUSPLP subsystem is composed of four subsystems developed as part of this research: (1) EU SPL Development, (2) Application Derivation, (3) Application Distributor, and (4) TeC EUSPLP Adaptor. EU SPL Development subsystem provides the user interface, services and storage mechanisms for EU SPL designers to create and edit end user product lines. The Application Derivation subsystem provides the user interface, services and storage mechanisms for end users to derive TeC applications. The Application Distributor subsystem provides services for external systems to query and retrieve the derived application. . The TeC EUSPLP Adaptor subsystem is responsible for acquiring the application derivation specification from the Application Distribution subsystem and sending it to the target TeC environment to be stored in the TeC database. End users can utilize the TeC environment to complete the application deployment.

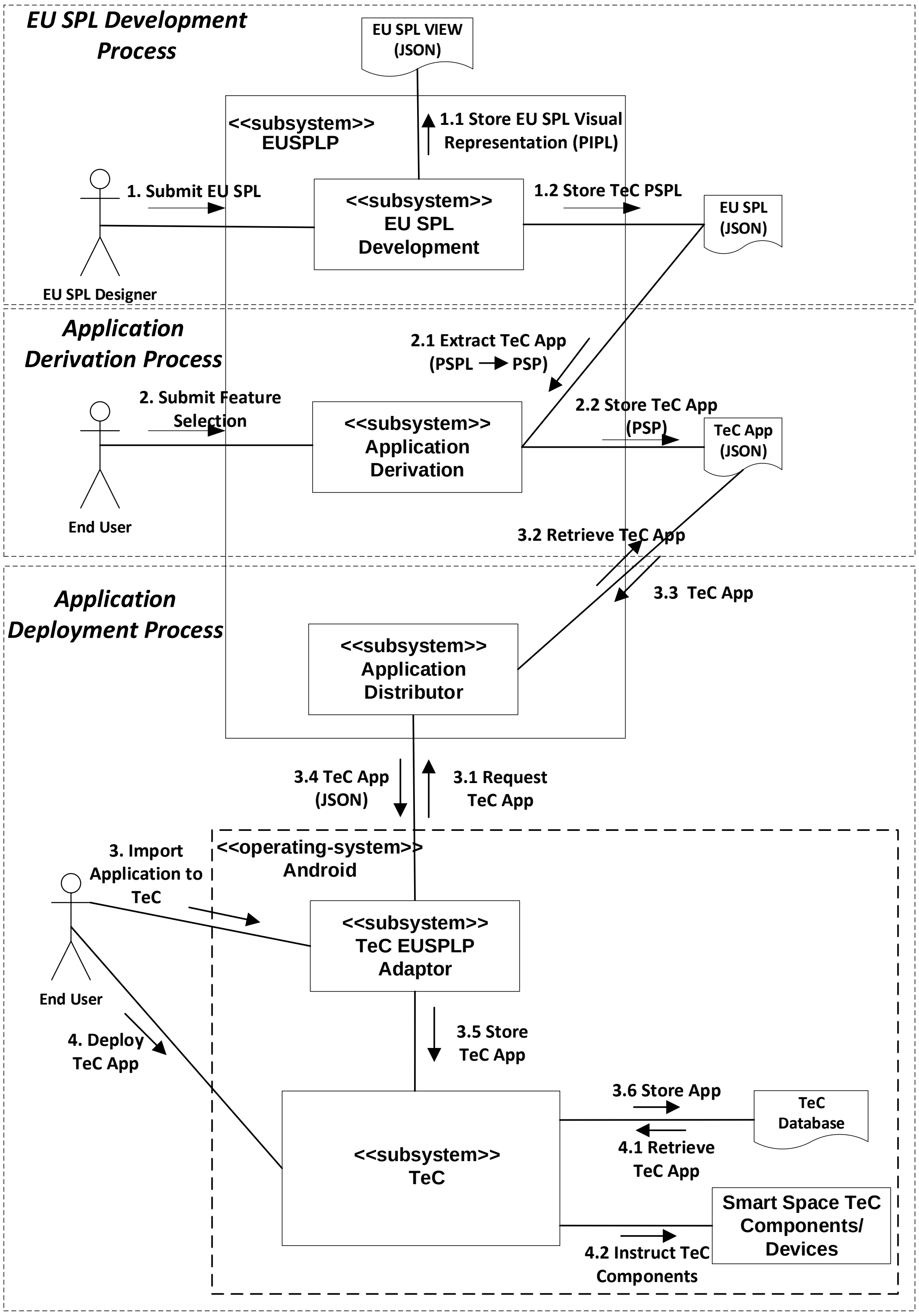

EU SPL Development Process
EU SPL VIEW (JSON)
<<subsystem>> EUSPLP
1.1 Store EU SPL Visual Representation (PIPL)
1. Submit EU SPL
<<subsystem>> EU SPL Development
1.2 Store TeC PSPL
EU SPL (JSON)
EU SPL Designer
Application Derivation Process
2.1 Extract TeC App (PSPL → PSP)
2. Submit Feature Selection
<<subsystem>> Application Derivation
2.2 Store TeC App (PSP)
TeC App (JSON)
End User
3.2 Retrieve TeC App
Application Deployment Process
3.3 TeC App
<<subsystem>> Application Distributor
3.4 TeC App (JSON)
3.1 Request TeC App
<<operating-system>> Android
3. Import Application to TeC
<<subsystem>> TeC EUSPLP Adaptor
End User
4. Deploy TeC App
3.5 Store TeC App
3.6 Store App
TeC Database
4.1 Retrieve TeC App
<<subsystem>> TeC
Smart Space TeC Components/ Devices
4.2 Instruct TeC Components

Figure 6.2 EUSPLP Subsystem Architecture and Processes

The EUSPLP supports three major processes shown in Figure 6.2: (1) EU SPL Development, (2) Application Derivation, and (3) Application Deployment. The EU SPL Development process enables end users to develop and store EU SPLs that can be used for deriving EU applications. In detail, after developing product line features, EU SPL designers submit the EU SPL to the EU SPL Development subsystem for processing (shown in step "1. Submit EU SPL" in Figure 6.2). The EU SPL Development subsystem stores the EU SPL Platform Independent Product Line (PIPL) model (shown in step "1.1 Store EU SPL Visual Representation (PIPL)") in JSON format. The PIPL captures the EU SPL visual representation. The EU SPL Development subsystem converts the PIPL model to the TeC Platform Specific Product Line (PSPL) model. The TeC PSPL is serialized as JSON in the file system for long term storage, as shown in step "1.2 Store TeC PSPL."

The Application Derivation process enables end users to derive applications for their smart spaces. In detail, the Application Derivation process starts with the end user selecting features from the EU SPL and submitting the selections to the Application Derivation subsystem, as shown in step "2. Submit Feature Selection" in Figure 6.2. The Application Derivation subsystem extracts the component architecture of the selected features from the PSPL (shown in step "2.1 Extract TeC App (PSPL → PSP)") and generates the TeC App (PSP). The TeC App is serialized to JSON in the file system, as shown in step "2.2 Store TeC App (PSP)" in Figure 6.2.

The Application deployment process enables end users to import derived applications to the TeC environment and deploy them to their smart spaces. The Application Deployment process starts with the end user interfacing with the TeC EUSPLP

Adaptor deployed in the target TeC platform e.g., Android. In detail, The TeC EUSPLP Adaptor subsystem interacts with the Application Distributor to retrieve the derived application (shown in steps “3. Import Application to TeC” through “3.4 TeC App (JSON)” in Figure 6.2) and stores the derived application to the TeC environment (shown in steps “3.5 Store TeC App” through “3.6 Store Appp” in Figure 6.2). To complete the deployment process of the derived application, the end user interacts with the TeC subsystem, as shown in step “4. Deploy TeC App” in Figure 6.2. The TeC subsystem retrieves the TeC App, as shown in step “4.1 Retrieve TeC App”, decomposes the TeC App into a set of individual instructions for TeC components and devices available in the smart space and communicates with the components / devices to provide them with application instructions as shown in step “4.2 Instruct TeC Components” in Figure 6.2.

6.4 EUSPLP Meta-models

This section describes the physical meta-models created for the EUSPLP development environment to represent: (a) EU SPLs for TeC (TeC PSPL), and (b) TeC applications (TeC PSPs). The EUSPLP physical meta-models are based on the TeC PSPL and PSP meta-models described in Chapter 5. In detail, this section describes how the TeC PSPL meta-model described in Chapter 5 was implemented in the EUSPLP to represent TeC EU SPLs. In addition, the section describes how the TeC PSP meta-model described in Chapter 5 was implemented to represent a TeC application. The TeC PSPL meta-model created for the EUSPLP was created as part of the development of the EUSPLP. The part of the TeC physical meta-model used by the EUSPLP to represent the TeC application, excluding the meta-classes used for TeC application deployment, was implemented as part of this research.

6.4.1 EUSPLP TeC PSPL Meta-model

The EUSPLP TeC PSPL meta-model describes the meta-classes and their relationships used to represent a TeC EU SPL. The EUSPLP TeC PSPL meta-model is divided into three logical areas: (1) Feature meta-model, (2) Feature to TeC EU SPL Component meta-model relationships, and (3) TeC EU SPL Component meta-model. The sections below describe in detail each of the meta-models.

Figure 6.3 shows the TeC PSPL Feature meta-model. In detail, the EUSPL meta-class is used to capture the TeC product line information. The EUSPL meta-class has one or more Features. The Feature meta-class captures information about product line features. Each Feature can be a member of zero-or-one FeatureGroup. The FeatureGroup meta-class is used to group a set of related Features with a particular constraint on their usage in a derived application. The feature group types supported by the EUSPLP are AT_LEAST_ONE_OF_FEATURE_GROUP, EXACLY_ONE_OF_FEATURE_GROUP, ZERO_OR_ONE_OF_FEATURE_GROUP and ZERO_OR_MORE_OF_FEATURE_GROUP. Each Feature meta-class contains exactly one FeatureVariability meta-class to describe the variability of the Feature meta-class. The variability types supported by the EUSPLP are COMMON, OPTIONAL, DEFAULT_OPTIONAL, DEFAULT_ALTERNATIVE, ALTERNATIVE and PARAMETERIZED_FEATURE. A Feature can itself contain zero or more Features. This relationship is shown in Figure 6.3 through the childFeatures attribute attached to the

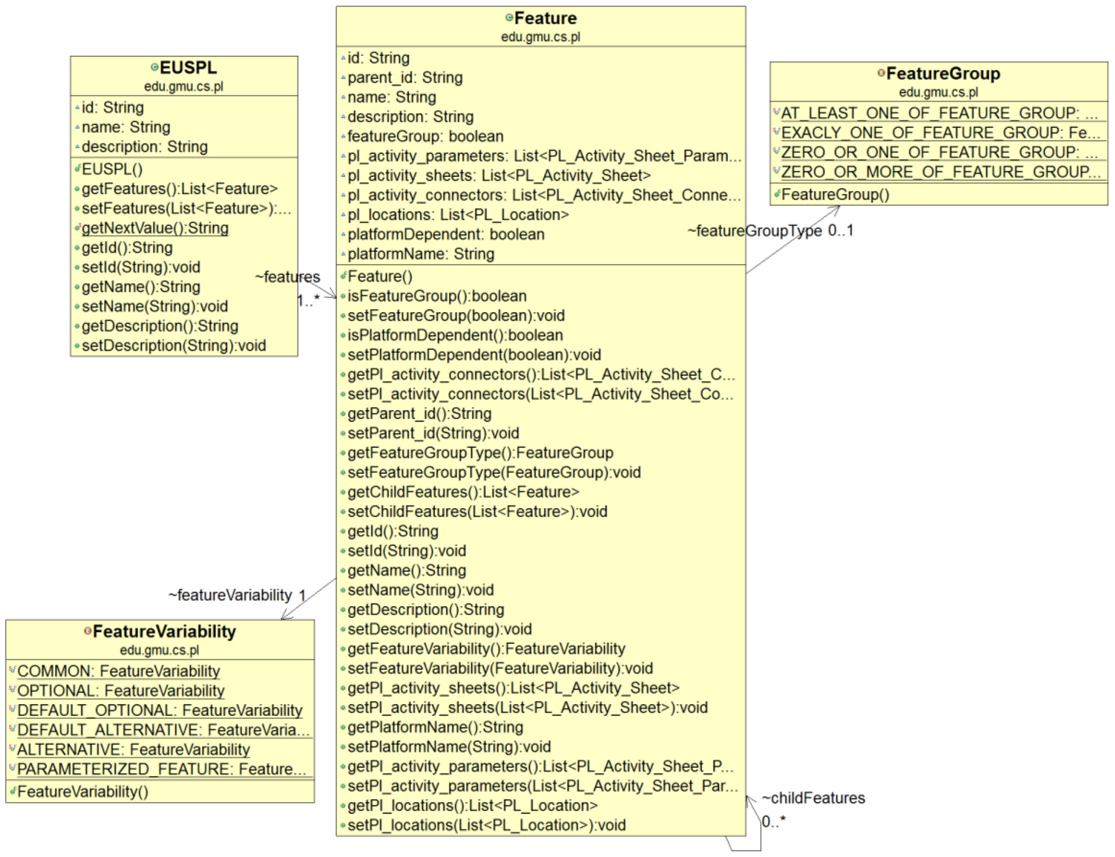
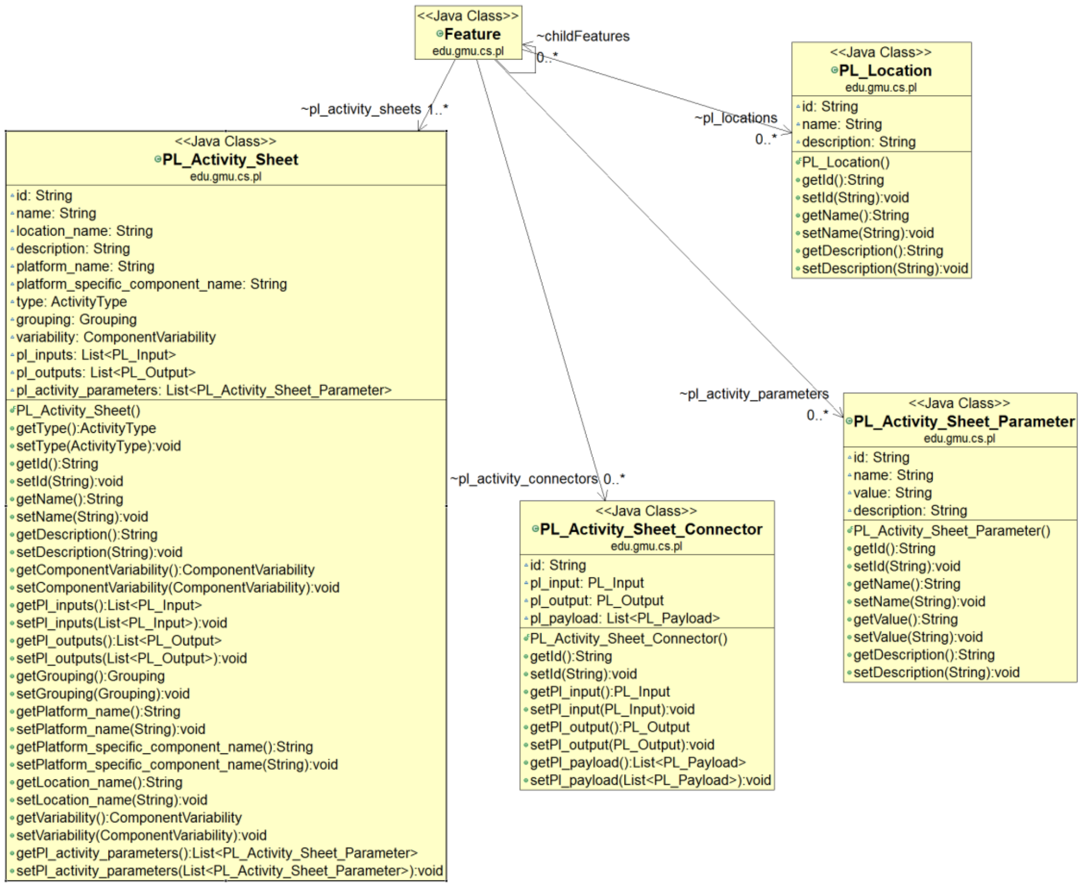

EUSPL
edu.gmu.cs.pl
id: String
name: String
description: String
EUSPL()
getFeatures():List<Feature>
setFeatures(List<Feature>):...
getNextValue():String
getId():String
setId(String):void
getName():String
setName(String):void
getDescription():String
setDescription(String):void
~features
1..*
Feature
edu.gmu.cs.pl
id: String
parent_id: String
name: String
description: String
featureGroup: boolean
pl_activity_parameters: List<PL_Activity_Sheet_Param...
pl_activity_sheets: List<PL_Activity_Sheet>
pl_activity_connectors: List<PL_Activity_Sheet_Conne...
pl_locations: List<PL_Location>
platformDependent: boolean
platformName: String
Feature()
isFeatureGroup():boolean
setFeatureGroup(boolean):void
isPlatformDependent():boolean
setPlatformDependent(boolean):void
getPl_activity_connectors():List<PL_Activity_Sheet_C...
setPl_activity_connectors(List<PL_Activity_Sheet_Co...
getParent_id():String
setParent_id(String):void
getFeatureGroupType():FeatureGroup
setFeatureGroupType(FeatureGroup):void
getChildFeatures():List<Feature>
setChildFeatures(List<Feature>):void
getId():String
setId(String):void
getName():String
setName(String):void
getDescription():String
setDescription(String):void
getFeatureVariability():FeatureVariability
setFeatureVariability(FeatureVariability):void
getPl_activity_sheets():List<PL_Activity_Sheet>
setPl_activity_sheets(List<PL_Activity_Sheet>):void
getPlatformName():String
setPlatformName(String):void
getPl_activity_parameters():List<PL_Activity_Sheet_P...
setPl_activity_parameters(List<PL_Activity_Sheet_Par...
getPl_locations():List<PL_Location>
setPl_locations(List<PL_Location>):void
FeatureGroup
edu.gmu.cs.pl
AT_LEAST_ONE_OF_FEATURE_GROUP: ...
EXACLY_ONE_OF_FEATURE_GROUP: Fe...
ZERO_OR_ONE_OF_FEATURE_GROUP: ...
ZERO_OR_MORE_OF_FEATURE_GROUP...
FeatureGroup()
~featureGroupType 0..1
~featureVariability 1
FeatureVariability
edu.gmu.cs.pl
COMMON: FeatureVariability
OPTIONAL: FeatureVariability
DEFAULT_OPTIONAL: FeatureVariability
DEFAULT_ALTERNATIVE: FeatureVaria...
ALTERNATIVE: FeatureVariability
PARAMETERIZED_FEATURE: Feature...
FeatureVariability()
~childFeatures
0..*

Figure 6.3 EUSPLP TeC PSPL - Feature Meta-Model

Feature meta-class. Nested Features relationships are used in the EUSPLP to represent the EU SPL as a Feature hierarchy.

Figure 6.4 shows the relationships between the Feature meta-class and the TeC EU SPL Component meta-model. The TeC EU SPL Component meta-model contains the meta-classes and relationships needed for the implementation of each Feature. As shown on Figure 6.4 the component meta-classes associated to the Feature meta-class are: PL_Activity_Sheet, PL_Activity_Connector, PL_Location, and PL_Activity_Parameter. The PL_Activity_Sheet meta-class represents TeC components extended with product line



<<Java Class>>
Feature
edu.gmu.cs.pl
~childFeatures
0..*
~pl_activity_sheets 1..*
~pl_locations
0..*
~pl_activity_parameters
0..*
~pl_activity_connectors 0..*
<<Java Class>>
PL_Location
edu.gmu.cs.pl
id: String
name: String
description: String
PL_Location()
getId():String
setId(String):void
getName():String
setName(String):void
getDescription():String
setDescription(String):void
<<Java Class>>
PL_Activity_Sheet
edu.gmu.cs.pl
id: String
name: String
location_name: String
description: String
platform_name: String
platform_specific_component_name: String
type: ActivityType
grouping: Grouping
variability: ComponentVariability
pl_inputs: List<PL_Input>
pl_outputs: List<PL_Output>
pl_activity_parameters: List<PL_Activity_Sheet_Parameter>
PL_Activity_Sheet()
getType():ActivityType
setType(ActivityType):void
getId():String
setId(String):void
getName():String
setName(String):void
getDescription():String
setDescription(String):void
getComponentVariability():ComponentVariability
setComponentVariability(ComponentVariability):void
getPl_inputs():List<PL_Input>
setPl_inputs(List<PL_Input>):void
getPl_outputs():List<PL_Output>
setPl_outputs(List<PL_Output>):void
getGrouping():Grouping
setGrouping(Grouping):void
getPlatform_name():String
setPlatform_name(String):void
getPlatform_specific_component_name():String
setPlatform_specific_component_name(String):void
getLocation_name():String
setLocation_name(String):void
getVariability():ComponentVariability
setVariability(ComponentVariability):void
getPl_activity_parameters():List<PL_Activity_Sheet_Parameter>
setPl_activity_parameters(List<PL_Activity_Sheet_Parameter>):void
<<Java Class>>
PL_Activity_Sheet_Parameter
edu.gmu.cs.pl
id: String
name: String
value: String
description: String
PL_Activity_Sheet_Parameter()
getId():String
setId(String):void
getName():String
setName(String):void
getValue():String
setValue(String):void
getDescription():String
setDescription(String):void
<<Java Class>>
PL_Activity_Sheet_Connector
edu.gmu.cs.pl
id: String
pl_input: PL_Input
pl_output: PL_Output
pl_payload: List<PL_Payload>
PL_Activity_Sheet_Connector()
getId():String
setId(String):void
getPl_input():PL_Input
setPl_input(PL_Input):void
getPl_output():PL_Output
setPl_output(PL_Output):void
getPl_payload():List<PL_Payload>
setPl_payload(List<PL_Payload>):void

Figure 6.4 Feature to TeC EU SPL Component Meta-Model Relationships

semantics to capture variability. TeC components represent devices and software available in a TeC environment. Examples of PL_Activity_Sheet meta-classes are phones, cameras, motion sensors, etc. A Feature can have one or more PL_Activity_Sheet meta-classes. The PL_Activity_Connector meta-class captures connectivity of PL_Activity_Sheet meta-classes related to a Feature. Features can have zero or more PL_Activity_Connector meta-classes. The PL_Location meta-class captures location information applicable to a given Feature. The PL_Activity_Parameter meta-class captures configurable internal parameters

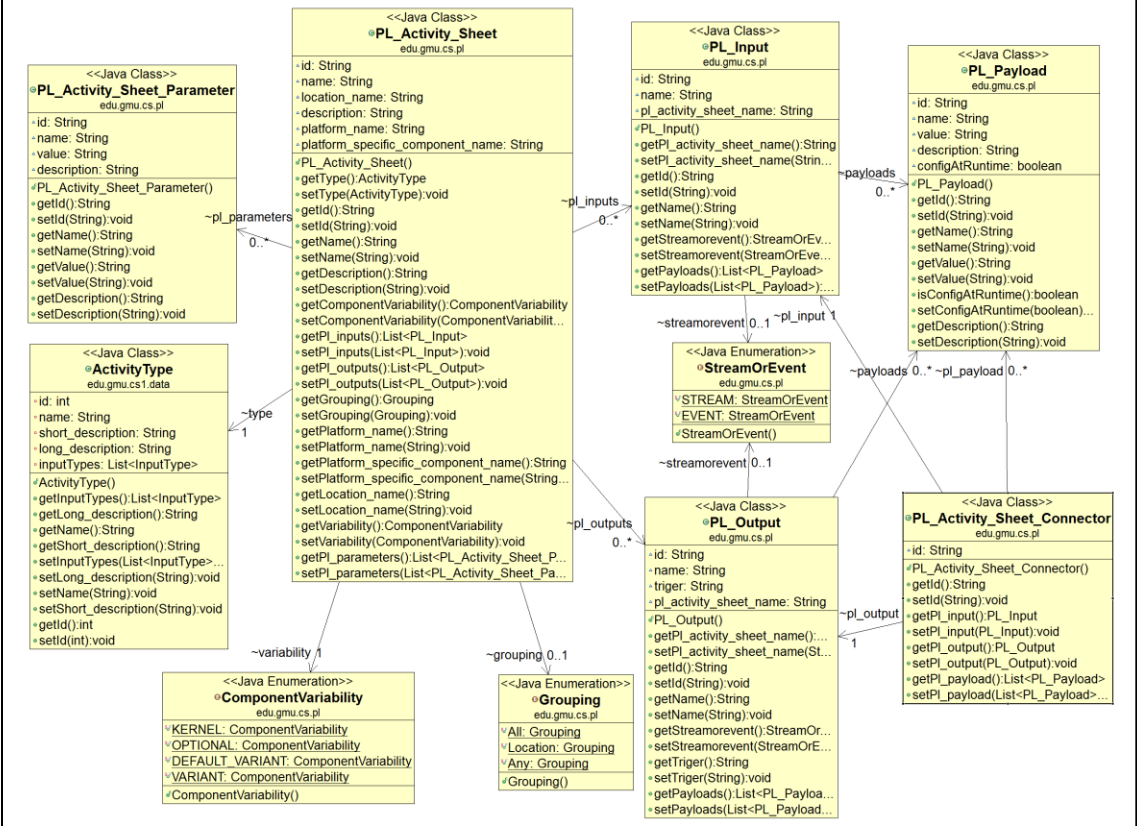


Figure 6.5 TeC EU SPL Component Meta-Model

of the PL_Activity_Sheet. Feature meta-classes can have zero or more PL_Activity_Parameter meta-classes.

Figure 6.5 shows the TeC EU SPL Component meta-model. In detail, the PL_Activity_Sheet meta-class has zero or more PL_Input and PL_Output meta-classes. The PL_Input meta-class captures input events and the PL_Output meta-class captures output events or data streams of the PL_Activity_Sheet meta-class. Examples of data streams can be audio or video data. The PL_Input events capture changes in the environment and based on the input values can modify the internal state of the PL_Activity_Sheet meta-class. The PL_Output events cause output events to be generated when changes occur in the internal state of the PL_Activity_Sheet. The PL_Output events have a triggering condition that is based on the PL_Activity_Sheet internal variables. The PL_Payload meta-class captures the data elements send by output events to inputs.

As shown on Figure 6.5 the PL_Output and PL_Input events have zero or more payload data. The PL_Output is connected to the PL_Input through the PL_Activity_Connector meta-class in order to connect different components. The PL_Activity_Connector meta-class combines inputs, outputs and payloads to ensure data integrity. The PL_Activity_Connector meta-class has one PL_Output to indicate the beginning of the component connection, one PL_Input to indicate the end of the component connection and zero-or-one PL_Payload to indicate the data payload to be used between the PL_Output and PL_Input events. Each PL_Activity_Sheet meta-class can have zero-or-one Grouping. The Grouping meta-class represents the way that the PL_Activity_Sheet is applied to the physical environment. For example a PL_Activity_Sheet with grouping type “All” represents all devices/components in the physical environment that implement the activity type that the PL_Activity_Sheet meta-class represents. The grouping type “Location” represents all devices/components in a given location and “Any” represents any device/component that implement the activity type that the PL_Activity_Sheet meta-class represents. The ComponentVariability meta-class captures the PL_Activity_Sheet variability information. Finally each PL_Activity_Sheet meta-class belongs to one ActivityType. The ActivityType meta-class is used to indicate the type of a PL_Activity_Sheet. For example consider an ActivityType that represents a motion sensor. The ActivityType for the motion sensor exposes an Application Programming Interface

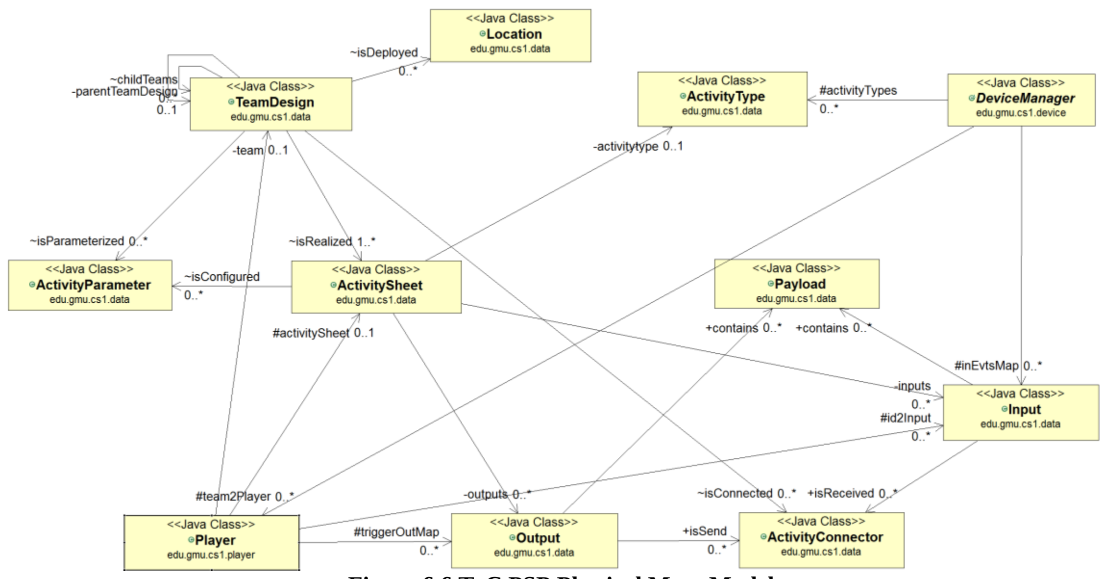


Figure 6.6 TeC PSP Physical Meta-Model

(API) for TeC meta-classes to use. A PL_Activity_Sheet meta-class that belongs to the motion sensor ActivityType represents an instance of the ActivityType and inherits all API functions from the type.

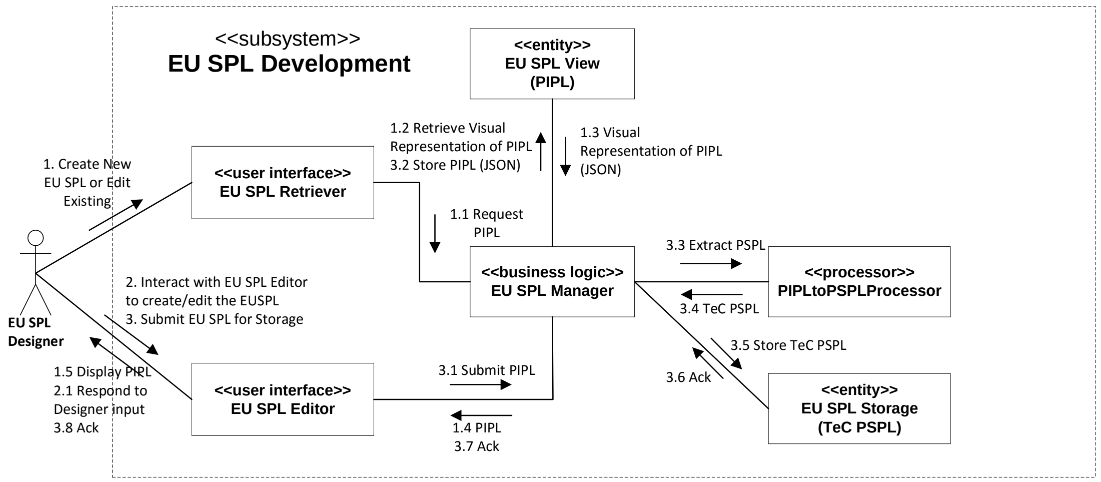
6.4.2 TeC Physical Meta-Model

This section describes the TeC Physical meta-model. The meta-model was used by the EUSPLP development environment to represent derived applications for TeC. The TeC meta-model excluding the DeviceManager / Player meta-classes and their relationships were developed as part of this research. The meta-model is used by the TeC Android simulator (Shen, 2014) (a) to capture the structure of TeC applications developed by end users, and (b) to map TeC application components to devices in the TeC environment during application deployment.

Table 6.1 TeC PSP Physical Meta-Model

Meta-Class Name	Meta-Class Description
ActivityType	Captures the logical component type (phone, moisture sensor, etc).
ActivitySheet	Capture ActivityType instances in TeC applications
TeamDesign	Captures a TeC Application
Location	Captures the location of the TeC Application
Input	Captures the input events of the ActivitySheet
Output	Captures the output events of the ActivitySheet
Payload	Captures the payload send between outputs/inputs
ActivityConnector	Captures the output/input connectivity
ActivityParameter	Captures the parameters of the ActivitySheet
DeviceManager	Captures the device information that implement each ActivityType. Each device has to extend the DeviceManager class. For this research the TeC devices were extended to support the Smart Home case study
Player	Captures device instances of different devices that are part of a TeC Application

The EUSPLP environment during application derivation, retrieves the components and connectors that realize the selected features from the EU SPL and maps them to the TeC meta-model in order to create the TeC application. The TeC application is stored in the TeC environment during the application deployment process. Figure 6.6 shows the meta-classes and relationships of the TeC physical meta-model. The main meta-classes of the TeC meta-model are the: TeamDesign, Location, ActivityParameter, ActivitySheet, ActivityType, Input, Output, Payload, ActivityConnector, DeviceManager and Player. Table 6.1 provides a brief description for each meta-class. The EUSPLP uses the entire TeC physical meta-model shown on Figure 6.6 to represent TeC applications, besides the DeviceManager and the Player meta-classes. The DeviceManager and Player meta-classes are used to capture low level application deployment information in the TeC environment.



<<subsystem>>
EU SPL Development
<<entity>> EU SPL View (PIPL)
<<user interface>> EU SPL Retriever
<<business logic>> EU SPL Manager
<<processor>> PIPLtoPSPLProcessor
<<user interface>> EU SPL Editor
<<entity>> EU SPL Storage (TeC PSPL)
EU SPL Designer
1. Create New EU SPL or Edit Existing
2. Interact with EU SPL Editor to create/edit the EUSPL
3. Submit EU SPL for Storage
1.5 Display PIPL
2.1 Respond to Designer input
3.8 Ack
1.1 Request PIPL
1.2 Retrieve Visual Representation of PIPL
3.2 Store PIPL (JSON)
1.3 Visual Representation of PIPL (JSON)
3.1 Submit PIPL
1.4 PIPL
3.7 Ack
3.3 Extract PSPL
3.4 TeC PSPL
3.5 Store TeC PSPL
3.6 Ack

Figure 6.7 EU SPL Development Subsystem and Component Interactions

6.5 EUSPLP EU SPL Development Subsystem

Figure 6.7 shows the internal composition of the EU SPL Development subsystem and the EU SPL designer interactions. The EU SPL Development subsystem is composed of six components: (1) EU SPL Editor, (2) EU SPL Retriever, (3) EU SPL Manager, (4) EU SPL View, (5) PIPLtoPSPLProcessor, and (6) EU SPL Storage. The EU SPL Editor provides the user interface for developing EU SPLs. The EU SPL Retriever provides the user interface to query existing EU SPLs. The EU SPL Manager provides the services for creating and retrieving EU SPLs. The EU SPL View provides services for storing and retrieving the visual representation (PIPL) of the EU SPL. The PIPLtoPSPLProcessor generates the TeC product line model (TeC PSPL) from the visual representation (PIPL). The EU SPL Storage provides services for storing and retrieving the TeC PSPL. The reason for having different components is to have separation of concerns on the functionality provided by each of the components. With this approach, components can be reused by other subsystems. Another benefit is that internal updates of individual components do not affect the rest of the components.

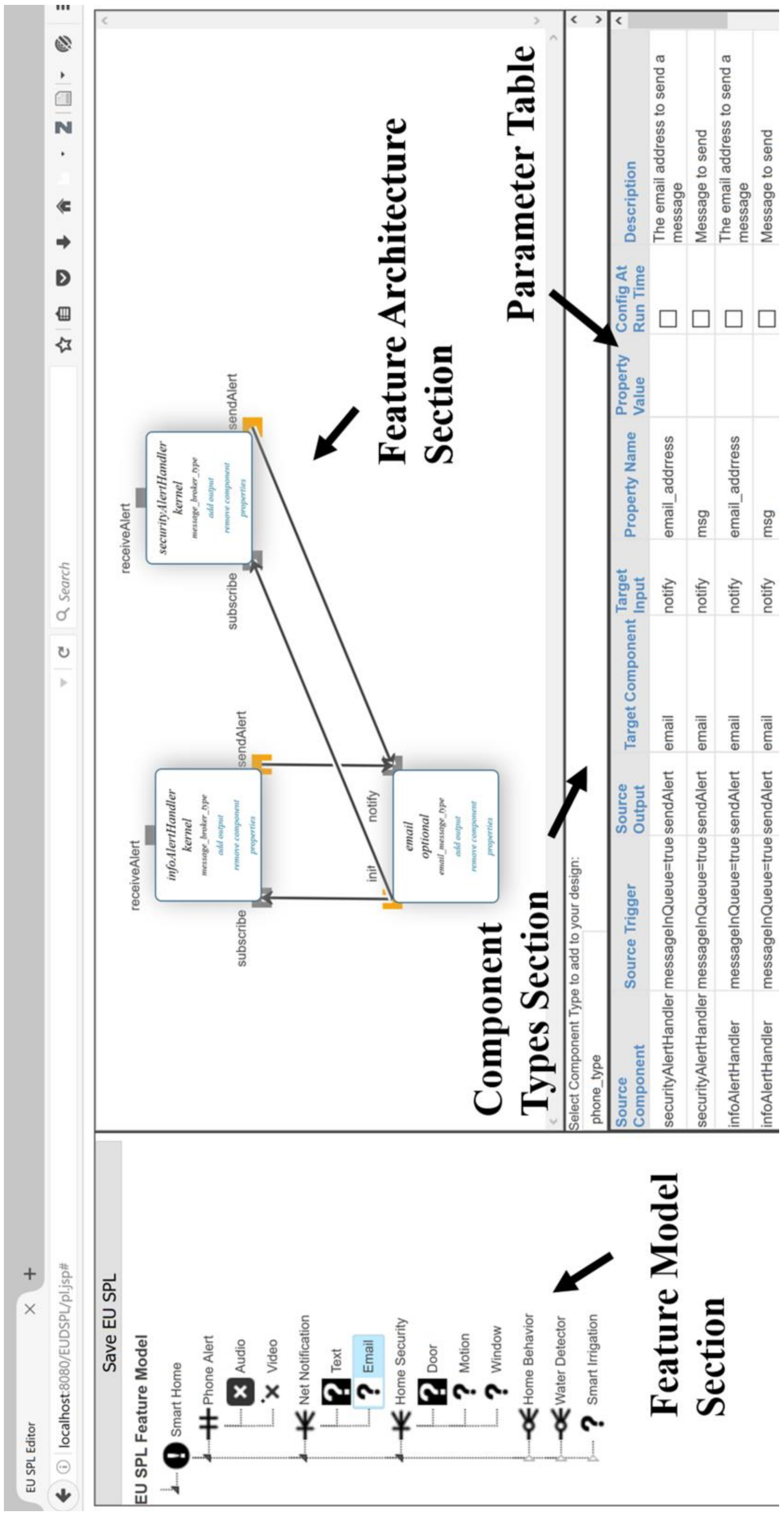
The remainder of this section discusses the EU SPL designer interactions and inner workings of the EU SPL Development subsystem. In detail, the EU SPL designer interacts with the EU SPL Retriever component to retrieve or create a new EU SPL, as shown in steps “1. Create New EU SPL or Edit Existing” in Figure 6.7 through “1.5 Display PIPL.” The EU SPL designer interacts with the EU SPL Editor to create or edit the EU SPL, as shown in step “2. Interact with EU SPL Editor to create/edit the EUSPL” in Figure 6.7. The EU SPL Editor responds to the EU SPL Designer inputs, as shown in step “2.1 Respond to Designer input.”

Next, the EU SPL designer submits the EU SPL to the EU SPL Editor, as shown in step “3. Submit EU SPL for Storage” in Figure 6.7. The EU SPL Editor submits the EU SPL to the EU SPL Manager in JSON format as shown in step “3.1 Submit PIPL.” The communication between the EU SPL Editor and the EU SPL Manager is through REST services. The JSON message that the EU SPL Editor sends contains PIPL with visual representation constructs used by the user interface of the editor. The EU SPL Manager sends the PIPL to the EU SPL View component to store the PIPL shown in step “3.2 Store PIPL.” After the PIPL view is stored, the EU SPL Manager sends the PIPL to the PIPLtoPSPLProcessor shown in step “3.3 Extract PSPL” to convert the PIPL to the TeC PSPL. The PIPLtoPSPLProcessor extracts the TeC PSPL specification, as a Java Object representation, from the PIPL. The PIPLtoPSPLProcessor sends the TeC PSPL specification to the EU SPL Manager as shown in step “3.4 TeC PSPL.” The EU SPL

Manager sends the TeC PSPL representation to the EU SPL Storage component for storage shown in step “3.5 Store TeC PSPL.” The EU SPL Storage component stores the TeC PSPL representation on the file system in JSON format and sends an acknowledgement message to the EU SPL Manager shown in step “3.6 Ack.” Upon successful storage of the TeC PSPL, the EU SPL Manager sends an acknowledgement message to the EU SPL Editor shown in step “3.7 Ack.” The EU SPL Editor shows an acknowledgement message to the EU SPL designer that the EU SPL has been stored successfully, as shown in step “3.8 Ack.” The EU SPL designer can repeat the processes shown in Figure 6.7 to continue evolving the EUSPL.

6.5.1 EU SPL Editor

Figure 6.8 shows the user interface of the EU SPL Editor used to develop EU SPLs. The user interface utilizes an interactive tree structure for representing the EU SPL feature model and a drag and drop interface for component designs to make it more natural for EU SPL designers to use. The user interface is divided in four main sections: (1) The Feature Model section, (2) The Feature Architecture section, (3) The Component Types section, and (4) The Parameter Table.



Select Component Type to add to your design:

phone_type

Source Component	Source Trigger	Source Output	Target Component	Target Input	Property Name	Property Value	Config At Run Time	Description
securityAlertHandler	messageInQueue=true	sendAlert	email	notify	email_addrress		☐	The email address to send a message
securityAlertHandler	messageInQueue=true	sendAlert	email	notify	msg		☐	Message to send
infoAlertHandler	messageInQueue=true	sendAlert	email	notify	email_addrress		☐	The email address to send a message
infoAlertHandler	messageInQueue=true	sendAlert	email	notify	msg		☐	Message to send

Figure 6.8 EU SPL Editor User Interface

6.5.1.1 Feature Model Section

The Feature Model section is responsible for capturing the SPL feature model. The Feature Model section was implemented in JavaScript by customizing and extending the jsTree (Duckett, 2014) tree plugin of the jQuery technology. The EU SPL designer can right click on the feature model section as shown in Figure 6.9 through Figure 6.11 to create new features, platform dependent features and feature groups. The Feature Model is represented as a hierarchical tree structure in the EUSPLP. The reason that a hierarchical tree structure was used to represent the feature model versus a directed acyclic graph normally used in traditional SPLs was to make it simpler for EU SPL designers to visualize the product line features and their dependencies. Furthermore different icons were used as a visual representation of different feature types. The visual representation of feature types was used to simplify the user interface. The remainder of this section describes the visual representations of the feature types.

The Feature Model section supports the creation of (a) common, (b) default optional, (c) optional, (d) default alternative, and (e) alternative features. Common features are represented with the exclamation mark ❗ icon in a black circle and represent features that are required for application derivation. Default optional features are represented with a white question mark ⍰ icon in a black background and represent the default features from a set of optional features. Optional features are represented with a black question mark ? icon and represent features that are optional. Default alternative features are represented with the ⊠ icon and represent the default feature from a set of alternative features.

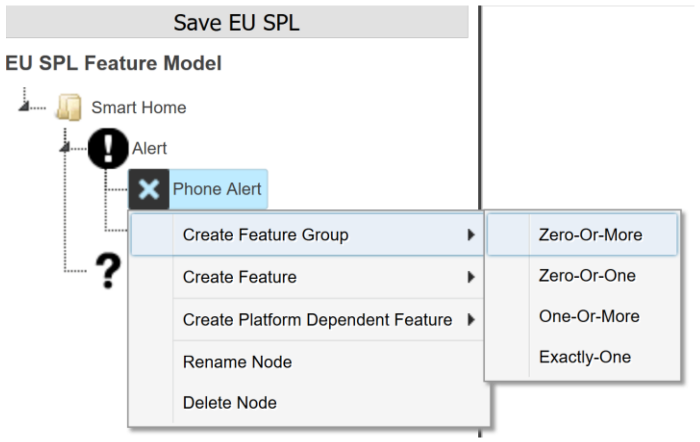


Figure 6.9 Feature Group Menu in the EU SPL Editor

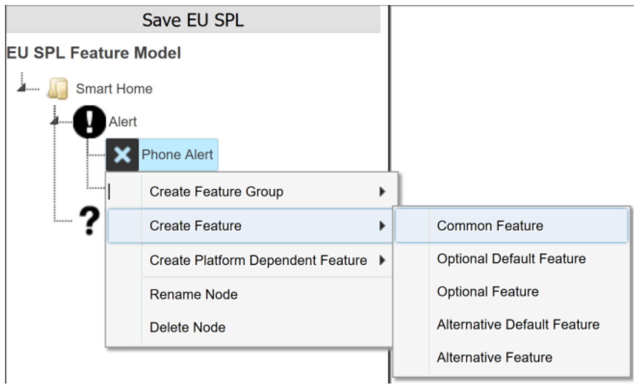


Figure 6.10 Feature Menu in the EU SPL Editor

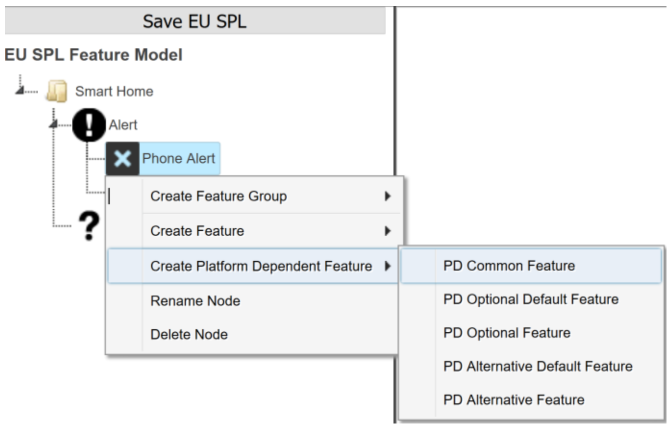


Figure 6.11 Platform Dependent Menu in the EU SPL Editor

Alternative features are represented with × icon and represents mutually exclusive features. The feature model also supports platform dependent features, which are features that are only applicable to a specific end user environment.

The Platform dependent features supported by the prototype are (a) common, (b) default optional, (c) optional, (d) default alternative, and (e) alternative features. The icons representing platform dependent features are similar to regular features but in addition have a dot indicator on the icon left corner. For example platform dependent common features are represented with the exclamation mark ❗ icon having a white dot on the left corner. Platform dependent default optional features are represented with a white question mark in a back background ❓ icon having a white dot on the left corner. Platform dependent optional features are represented with the question mark ? icon having a black dot on the left corner. Platform dependent default alternative features are represented with the ❎ icon having a white dot on the left corner. Alternative features are represented with × having a black dot on the left corner.

The feature groups supported by the prototype are (a) zero-or-more (b) zero-or-one (c) one or more and (d) exactly-one. The EUSPLP is using the crow's foot notation (Barker, 1990) to capture the cardinality of a feature group. The reason that Crow's foot notation was used in the EUSPLP was because the notation is widely used to represent entity relationships in data models. In detail, zero or more feature groups indicate that zero or more features can be selected from the feature group during application derivation. Zero or more feature groups are represented with the following icon -o< that has a circle to indicate zero features connected to three lines to indicate multiple features. Zero-or-one feature groups indicate that zero or one feature can be selected from the feature group during application derivation. Zero-or-one feature groups are represented with the following icon −o| that has a circle to indicate zero features connected to vertical line to indicate one feature. One or more feature groups indicate that one or more features can be selected from the feature group during application derivation. One or more feature groups are represented with the following icon |< that has a vertical line to indicate one feature connected to three lines to indicate multiple features. Exactly-one feature groups indicate that exactly one feature can be selected from the feature group during application derivation. Exactly-one feature groups are represented with the following icon −|| that has two vertical lines to indicate that minimum and maximum feature group cardinality is one. Table 6.2 displays

Table 6.2 EU SPL Editor Feature Model Notation

Feature Model Node Notation	**Feature Model Node Description**
❗	common feature
⍰	optional default feature
?	optional feature
❎	alternative default feature
✕	alternative feature
⁺❗	platform dependent common feature
⁺⍰	platform dependent optional default feature
⁺?	platform dependent optional feature
⁺❎	platform dependent alternative default feature
⁺✕	platform dependent alternative feature
−o<	zero-or-more feature group
−o|	zero-or-one feature group
−|<	one-or-more feature group
−||	exactly-one feature group

a summary of the EU SPL Editor Feature Model Notation used by the EUSPLP environment.

Internally each node on the feature model that describes a feature or a feature group has the following properties: id, icon, and data. The id property captures the unique id of each node on the feature model. The icon property captures the location of the icon representation of the node in the feature model. The data object captures the data needed to realize a product line feature. To create the EU SPL the EU SPL designer submits the feature model including feature nodes with their properties to the EU SPL Manager.

6.5.1.2 Feature Architecture Section

The Feature Architecture section shown in Figure 6.12 is used to capture the component/connector specification that realizes each feature. This section utilizes a drag and drop interface. Drag and drop interfaces are ubiquitous and used daily by end users. For instance drag and drop is used to resize windows in personal computers, tablets, navigate maps, to scroll up and down a document (Appert et al., 2015). Furthermore the What You She Is What You Get (WYSIWYG) principal used for end user development (Burnett, 2009) aims to have end users relate their programs to the end result. By utilizing the drag and drop interface, EU SPL designers can drag and drop components to the feature architecture section and connect them together. The feature architecture section was created in this research by customizing and extending the community edition of the jsPlumb (Porritt 2016) JavaScript Library.

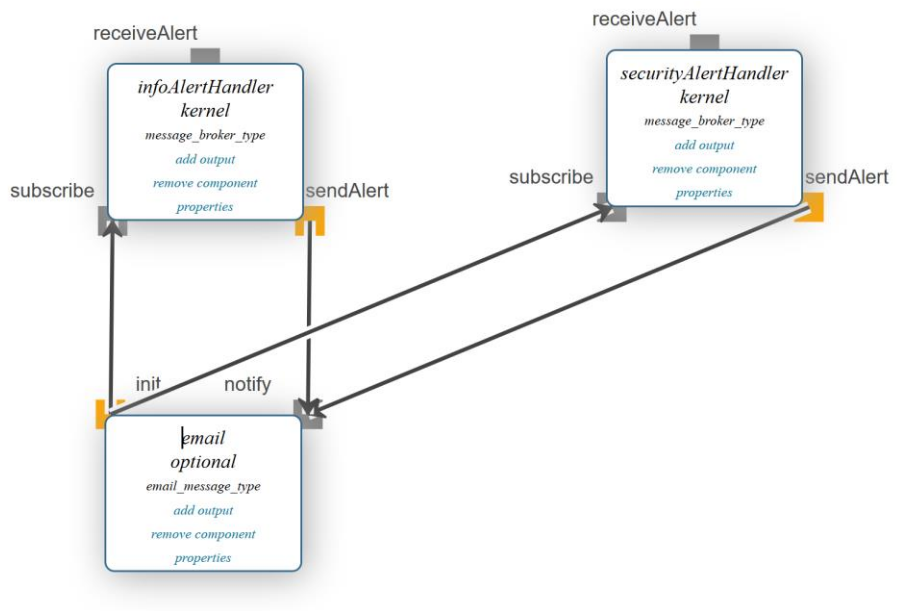


Figure 6.12 EUSPLP Component Example

In detail, the Feature Architecture section contains components and component connectors. The components are instances of TeC activity types. Components are represented as rectangular boxes in the feature architecture section. Inputs of the components are shown as gray boxes attached to the component box and outputs are shown as orange boxes attached to the component box. Figure 6.12 shows an example of a component design from the feature architecture section used to implement a feature. As shown in Figure 6.12, there are three components, the infoAlertHandler, the securityAlertHandler and the email. The infoAlertHandler and securityAlertHandler components have two inputs: subscribe and receiveAlert and one output sendAlert. The email component has one input notify and one output init. The design indicates that during the initialization the email component subscribes to the infoAlertHandler and securityAlertHandler components to receive messages. When a message is available the infoAlertHandler and securityAlertHandler components send alert messages to the notify input of the email component. The email will send email notifications upon the receipt of the alert message.

The component internal representation contains the following properties: comp_name, comp_type, variability_type, location, platform_name, platform_specific_component_name, is_group, inputs, and outputs. The comp_name property captures the component name. The comp_type property captures the activity type of the component. The variability_type property captures if the component is kernel, optional, variant, or default variant. The location property specifies the location name of the component. The platform_name property is applicable if the component is platform specific and indicates the name of the end user environment that the component applies. The platform_specific_component_name property is also applicable if the component is platform specific and indicates the component name in the end user environment that the component applies. The is_group property specifies if the component represents a grouping of components that implement the same activity type. The inputs property of the component is an array and specifies the input events of each component.

Inputs events are component notifications that can cause changes in a component state that can lead to the execution of component outputs. For example consider a component that represents a DVD player. The component can have an input event play that causes the DVD player to play a movie and output a video stream or an error message if there is no DVD in the player. Each object on the input events array contains the following properties: name, type, and a payloadlist. The name property specifies the name of the event type, the type property specifies if the input event is of type event or a video data stream and the payloadlist property specifies the payloads that the component needs to handle an event. Payload objects are mainly name-value pairs. For example consider an input event called “send-text” on a component that represents a cell phone. The send-text event will need to have a payload list that will consist of two payload objects. The first payload object will have a name called “phone_number” and value the actual phone number for example “(999) 999-9999” that the text will be send. The second payload object will have “message” as the name of the payload and the actual text that will be send as value. In the EUSPLP all component inputs are inherited by the components type that they represent. The outputs property of the component is an array and specifies the output events of each component.

Output events are events generated by a component when it’s internal state changes. For example consider a thermometer that makes a sound when a certain temperature gets reached. The sound is the output event of the thermometer. To control output events there are triggering conditions that when they are true the output event gets generated. Output events are connected to input events of other components to create application logic. Each object on the outputs array contain the following properties: name and triggering condition. The name specifies the output name and triggering condition specifies the state of the component that needs to be true in order for the output event to be generated. EU SPL designers can specify component outputs during component designs. The component connector object of the Feature Architecture section encapsulates the information needed to connect two components.

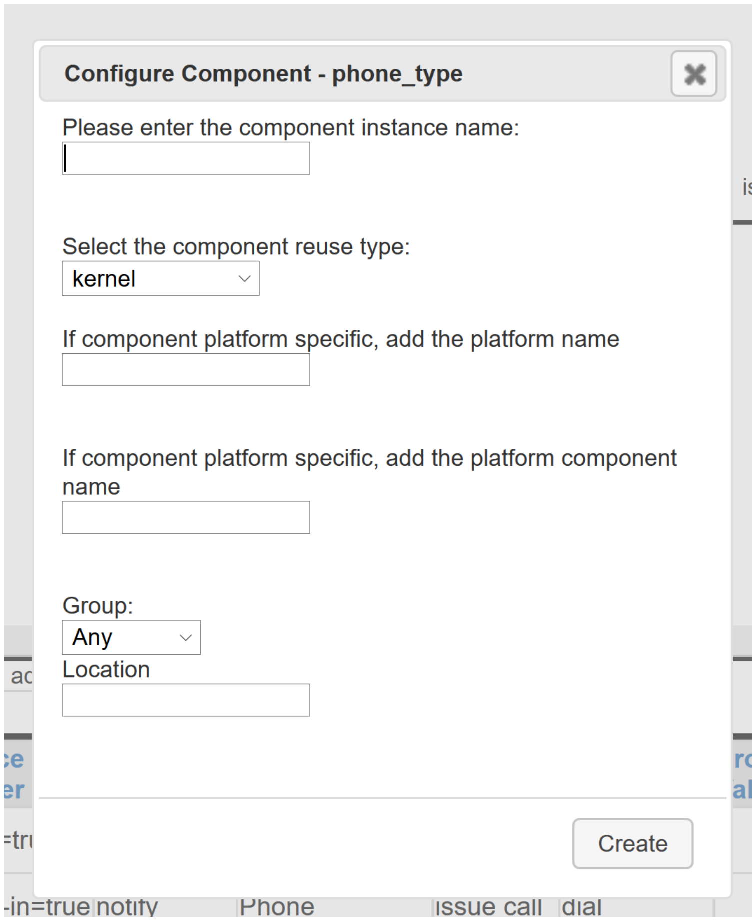
6.5.1.3 Component Types Section

The Components Types section displays all available component types in the EU SPL Editor that EU SPL designers can use to realize features. Since the EUSPLP targets to derive applications that can be deployed to the TeC environment, the component types used in the prototype are TeC activity types.

The properties of the component types are: id, name and inputTypes. The id and name properties specify the id and name of component type. The inputTypes is an array of input objects. To create or edit the component architecture of a feature, EU SPL designers select the component type. Upon the component selection, the EU SPL Editor prompts the EU SPL designer for additional component information (comp_name, variability_type, location, platform_name, platform_specific_component_name, and is_group) needed to create the component instance as shown in Figure 6.13. The EU SPL Editor combines the component type and the EU SPL designer entered information to create the activity instance in the Feature Architecture section. The component type user interface was developed by extending the JQuery UI (Sarrion 2012) JavaScript libraries.

6.5.1.4 Parameter Table Section

The Parameter Table section specifies all parameters that need to be configured either by the EU SPL designer or by the end users during application derivation. The parameter table user interface is created by extending the editablegrid (Máca, 2016) JavaScript libraries. The Parameter Table displays all component connector properties applicable to a selected feature from the feature model. The parameter table gets auto populated as EU SPL designers connect components in the Feature Architecture section to



Configure Component - phone_type

Please enter the component instance name:

Select the component reuse type:
kernel

If component platform specific, add the platform name

If component platform specific, add the platform component name

Group:
Any

Location

Create

Figure 6.13 Component Type Configuration

implement a feature. The internal parameter table representation contains the following properties: sourceactivityname, sourcetrigercondition, sourceoutput, targetactivityname, targetinput, configuredRunTime, propertyname, propertyvalue and description. The sourceactivityname specifies the name of the component name that that is the source of the component connection. The sourcetrigercondition property specifies the triggering condition that is needed for the output event to occur. The sourceoutput property specifies the output name that the component connection starts. The targetactivityname property specifies the target activity of the component connection. The target input property specifies the name of the input of the target activity that the component connection ends. The configuredRunTime property specifies if the parameter need to be defined during application derivation or if the parameter need to be specified by the EU SPL designer during feature creation. The propertyname specifies the name of the input parameter that needs to be specified by the output event for the target component to process the input event. The propertyvalue specifies the value of the propertyname property. The description property provides additional information about the propertyname property. All entries of the parameter table are stored on the feat_properties array specified for each feature.

6.5.2 Feature Creation in the EU SPL Editor

To create a feature, EU SPL designers create a node on the feature model by selecting the appropriate feature type. The Feature Architecture section and the Parameter Table are reset to accommodate the new feature architecture and parameters. Child features can be added under a feature node. Kernel features exist in all products derived by the product line, so they should not depend on non-kernel (optional/variant) features types. Optional and variant features can depend on kernel features or other features. After the feature node is created EU SPL designers select the component types and add them to the Feature Architecture section. As the EU SPL designers connect components to develop the feature architecture, the parameter table gets auto-populated based on the components configuration parameters. EU SPL designers configure the parameter table to complete the feature realization. When EU SPL designers select existing nodes on the feature model, the

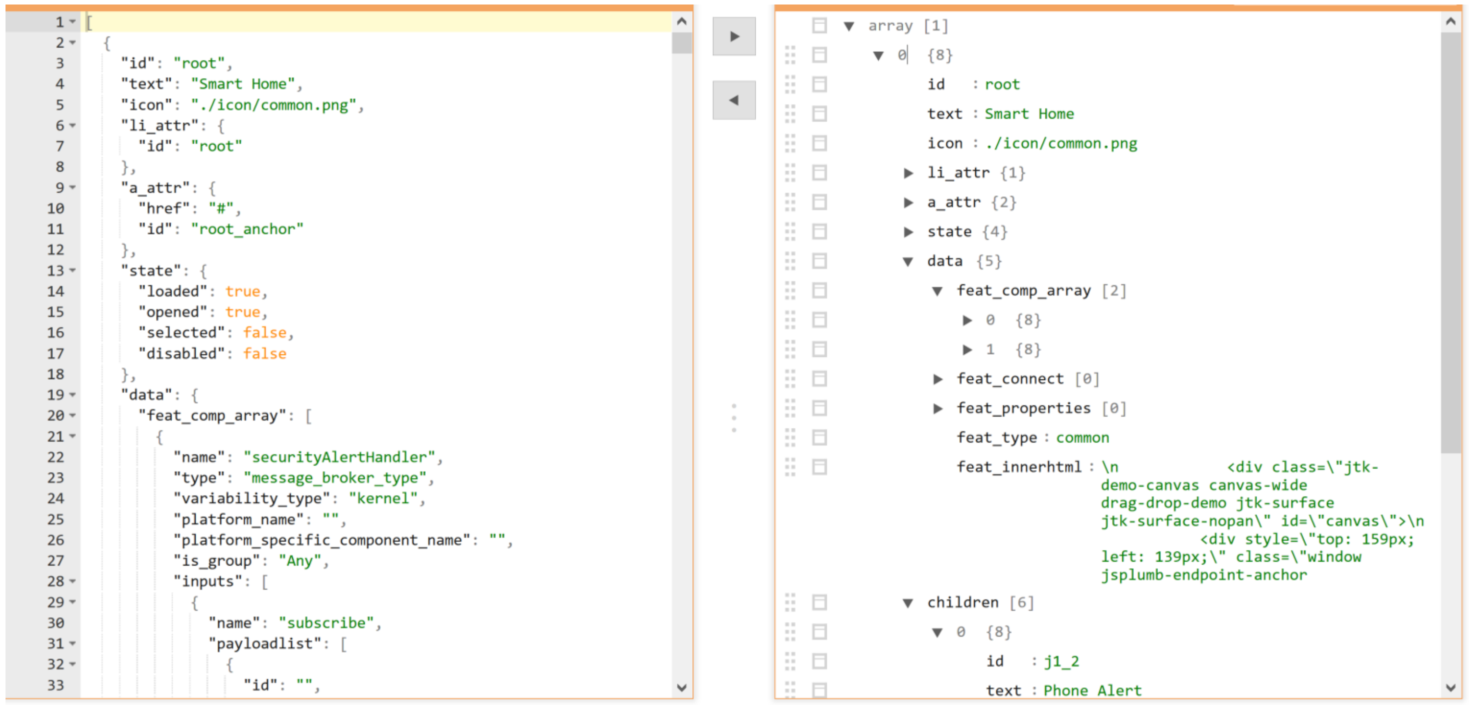
Feature Architecture and the Parameter table sections are restored and show the selected feature design and parameter values.

6.5.3 PIPL JSON Representation

The EU SPL designers select the “Save EU SPL” button in the user interface of the EU SPL Editor, shown in Figure 6.14 to store EU SPLs to the EUSPLP server. Internally the EU SPL Editor extracts from client memory the product line design (PIPL), serializes it to JSON format and sends it to the EU SPL Manager component for processing in the EUSPLP server. The PIPL captures the EU SPL representation in JSON combined with visual elements needed by the EU SPL Editor to display the end user product line.

Figure 6.14 shows part of the PIPL JSON representation created for the Smart Home EU SPL case study. The left side of Figure 6.14 displays the row JSON format of the Smart Home PIPL that was submitted to the EUSPLP for processing. The right side of Figure 6.14 displays the PIPL JSON in a human readable format.

In detail, the right side of Figure 6.14 shows that the PIPL is submitted to the EUSPLP server as an array. The array has one node named Smart Home and it contains eight properties: id, text, icon, li_attr, a_attr, state, data and children. The icon, li_attr, a_attr and state properties capture user interface information needed by the EU SPL editor. The data property captures the feature architecture of the Smart Home feature. The EU SPL Manager sends the PIPL JSON to the EU SPL View meta-class for storage as shown in Figure 6.14. The PIPL will be retrieved and sent to the EU SPL Editor when the EU SPL designers requests to edit the product line.



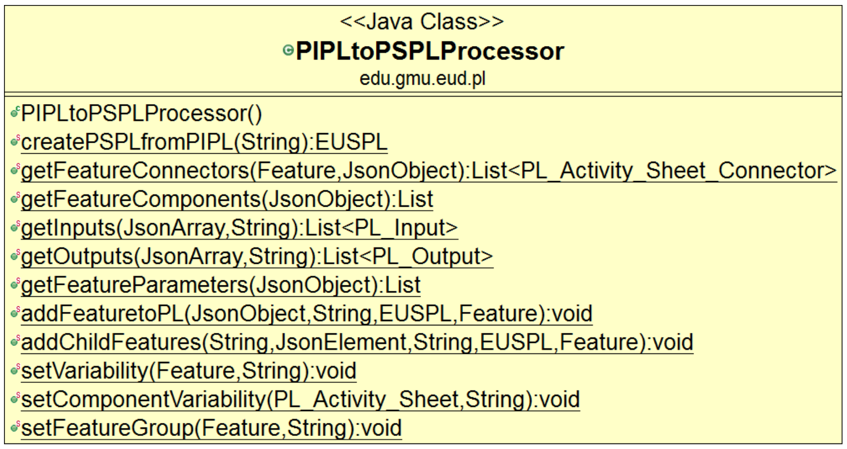
```
[
  {
    "id": "root",
    "text": "Smart Home",
    "icon": "./icon/common.png",
    "li_attr": {
      "id": "root"
    },
    "a_attr": {
      "href": "#",
      "id": "root_anchor"
    },
    "state": {
      "loaded": true,
      "opened": true,
      "selected": false,
      "disabled": false
    },
    "data": {
      "feat_comp_array": [
        {
          "name": "securityAlertHandler",
          "type": "message_broker_type",
          "variability_type": "kernel",
          "platform_name": "",
          "platform_specific_component_name": "",
          "is_group": "Any",
          "inputs": [
            {
              "name": "subscribe",
              "payloadlist": [
                {
                  "id": "",
```

```
▼ array [1]
  ▼ 0 {8}
      id   : root
      text : Smart Home
      icon : ./icon/common.png
    ▶ li_attr {1}
    ▶ a_attr {2}
    ▶ state {4}
    ▼ data {5}
      ▼ feat_comp_array [2]
        ▶ 0  {8}
        ▶ 1  {8}
      ▶ feat_connect [0]
      ▶ feat_properties [0]
        feat_type : common
        feat_innerhtml : \n            <div class=\"jtk-
                         demo-canvas canvas-wide
                         drag-drop-demo jtk-surface
                         jtk-surface-nopan\" id=\"canvas\">\n
                                   <div style=\"top: 159px;
                         left: 139px;\" class=\"window
                         jsplumb-endpoint-anchor
    ▼ children [6]
      ▼ 0  {8}
          id   : j1_2
          text : Phone Alert
```

Figure 6.14 Sample PIPL JSON Representation

6.5.4 PIPL to TeC PSPL Processing

The PIPLtoPSPLProcessor generates, from the EU SPL visual representation (PIPL), the TeC PSPL by following the PIPL to TeC PSPL mappings described in Chapter 5. The TeC PSPL created by the PIPLtoPSPLProcessor is distilled from visual elements and is exclusively used to describe the end user product line for the TeC environment. The separation of PIPL and TeC PSPL representations in the EUSPLP is used to decouple the user interface from the core product line logic of storing /retrieving and deriving applications form the TeC product line. This allows any updates to the user interface not to affect the core product line logic and vice versa. Figure 6.15 shows the main methods of the PIPLtoPSPLProcessor. In detail, the createPSPLfromPIPL method of the PIPLtoPSPLProcessor starts the PIPL to TeC PSPL conversion. The createPSPLfromPIPL method takes as input the JSON representation of the PIPL and returns the EUSPL object that represents the TeC PSPL. The createPSPLfromPIPL method makes calls the addFeaturetoPL method to extract the product line features from JSON and add it to the


<<Java Class>>
PIPLtoPSPLProcessor
edu.gmu.eud.pl

```
PIPLtoPSPLProcessor()
createPSPLfromPIPL(String):EUSPL
getFeatureConnectors(Feature,JsonObject):List<PL_Activity_Sheet_Connector>
getFeatureComponents(JsonObject):List
getInputs(JsonArray,String):List<PL_Input>
getOutputs(JsonArray,String):List<PL_Output>
getFeatureParameters(JsonObject):List
addFeaturetoPL(JsonObject,String,EUSPL,Feature):void
addChildFeatures(String,JsonElement,String,EUSPL,Feature):void
setVariability(Feature,String):void
setComponentVariability(PL_Activity_Sheet,String):void
setFeatureGroup(Feature,String):void
```

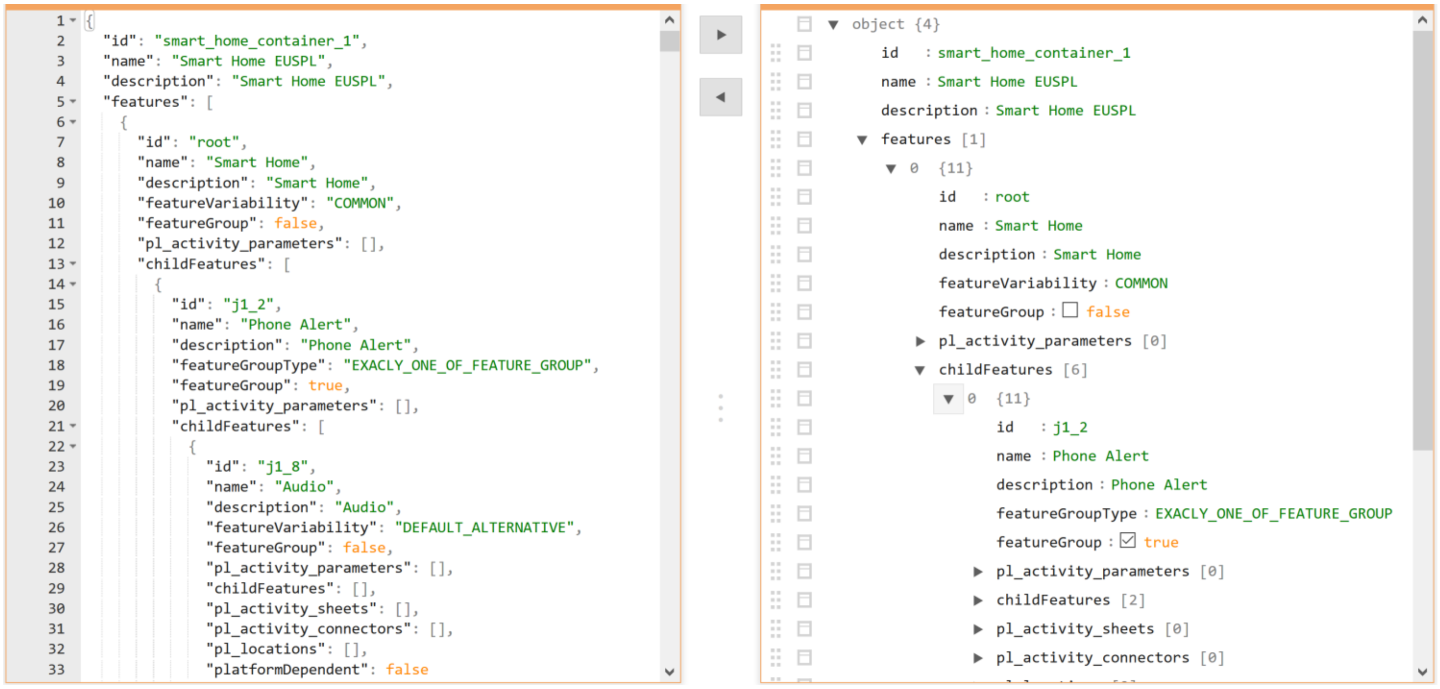
Figure 6.15 Methods of the PIPLtoPSPLProcessor Class

product line. The addFeaturetoPL method makes calls to: (1) the getFeatureComponents method to extract from the PIPL model the components that realize each feature (2) the getFeatureConnectors method to extract the component architecture of each feature (3) the getFeatureParameters method to extract the parameters of each feature (4) the setVariability method to extract the variability type and set it on each feature and (5) the addChildFeatures method that processes the child features. For each child feature the addChildFeatures calls recursively the addFeaturetoPL method. In addition to the methods above, there are also a set of utility methods defined to further extract feature and component meta-classes from the PIPL. In detail the utility method: (1) setFeatureGroup sets the group type of each feature, (2) getInputs extract the inputs of each component from the PIPL, (3) getInputs extract the outputs of each component from the PIPL, and (4) setComponentVariability sets the variability type of each component.

6.5.5 TeC PSPL JSON Representation

The output of the PIPLtoPSPLProcessor is the Java object representation of the TeC PSPL. The EU SPL Manager sends the TeC PSPL java representation to the EU SPL Storage class. The EU SPL Storage class converts the TeC PSPL java representation to JSON and stores it to the file system.

Figure 6.16 shows part of the TeC PSPL JSON representation created for the Smart Home EU SPL case study. The left side of Figure 6.16 displays the row JSON format of the Smart Home TeC PSPL as it is stored in the file system. The right side of Figure 6.16 displays the TeC PSPL JSON in a more readable format.



```
{
  "id": "smart_home_container_1",
  "name": "Smart Home EUSPL",
  "description": "Smart Home EUSPL",
  "features": [
    {
      "id": "root",
      "name": "Smart Home",
      "description": "Smart Home",
      "featureVariability": "COMMON",
      "featureGroup": false,
      "pl_activity_parameters": [],
      "childFeatures": [
        {
          "id": "j1_2",
          "name": "Phone Alert",
          "description": "Phone Alert",
          "featureGroupType": "EXACLY_ONE_OF_FEATURE_GROUP",
          "featureGroup": true,
          "pl_activity_parameters": [],
          "childFeatures": [
            {
              "id": "j1_8",
              "name": "Audio",
              "description": "Audio",
              "featureVariability": "DEFAULT_ALTERNATIVE",
              "featureGroup": false,
              "pl_activity_parameters": [],
              "childFeatures": [],
              "pl_activity_sheets": [],
              "pl_activity_connectors": [],
              "pl_locations": [],
              "platformDependent": false
```

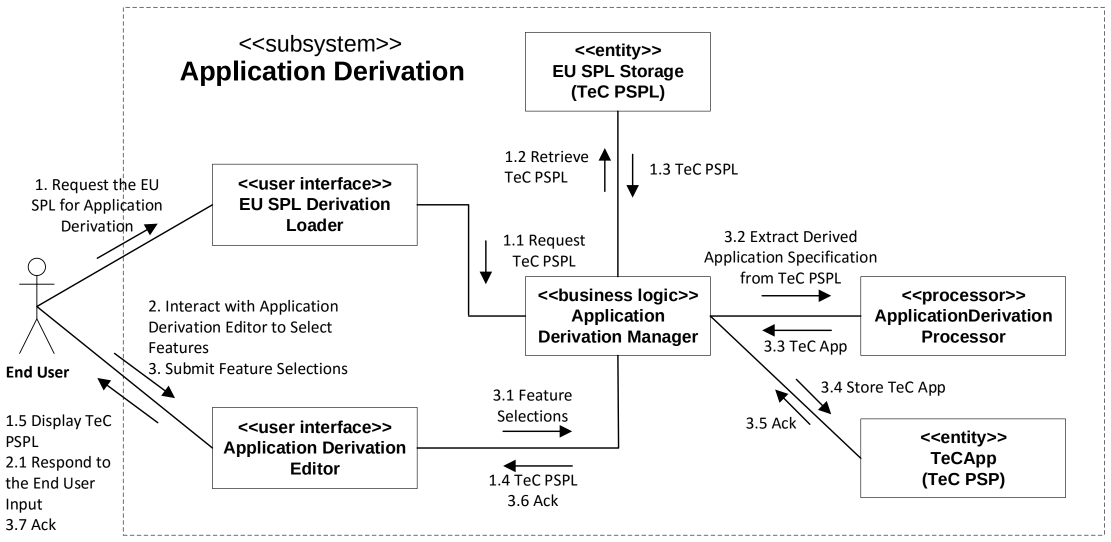
```
▼ object {4}
    id : smart_home_container_1
    name : Smart Home EUSPL
    description : Smart Home EUSPL
  ▼ features [1]
    ▼ 0 {11}
        id : root
        name : Smart Home
        description : Smart Home
        featureVariability : COMMON
        featureGroup : ☐ false
      ▶ pl_activity_parameters [0]
      ▼ childFeatures [6]
        ▼ 0 {11}
            id : j1_2
            name : Phone Alert
            description : Phone Alert
            featureGroupType : EXACLY_ONE_OF_FEATURE_GROUP
            featureGroup : ☑ true
          ▶ pl_activity_parameters [0]
          ▶ childFeatures [2]
          ▶ pl_activity_sheets [0]
          ▶ pl_activity_connectors [0]
```

Figure 6.16 Sample TeC PSPL JSON Representation

For instance, the right side of Figure 6.16 shows that the TeC PSPL is contained in the Smart Home EUSPL object. The Smart Home EUSPL has one common feature named Smart Home which is the root feature of the product line. The Smart Home feature is common and is not a feature group. The Smart Home feature contains six childFeatures. The six childFeatures are the Phone Alert, Net Notification, Home Security, Home Behavior, Water Detector and Smart Irrigation features which in return have their own features. Each feature on the EU SPL has the following properties: id, name, description, featureVariability, featureGroup, pl_activity_parameters, childFeatures, pl_activity_sheets, pl_activity_connectors, pl_locations, and platformDependent that capture the architecture of each feature. The JSON TeC PSPL is stored on the file system and gets accessed by the Application Derivation subsystem to derive the TeC Applications (PSP) based on the end user selections.

6.6 End User Application Derivation

Figure 6.17 shows the internal composition of the Application Derivation subsystem and the end user interactions needed to derive an application from the product line. The Application Derivation subsystem is composed of six components: (1) Application Derivation Editor, (2) EU SPL Derivation Loader, (3) Application Derivation Manager, (4) EU SPL Storage, (5) ApplicationDerivationProcessor, and (6) TeCApp. The Application Derivation Editor provides the user interface for deriving end user applications. The EU SPL Derivation Loader provides the user interface for selecting EU SPLs for application derivation. The EU SPL Manager provides the services and coordinates the interactions of components for creating and retrieving EU SPLs. The Application Derivation Manager provides services for retrieving the EU SPL and deriving/storing end user applications. The EU SPL Storage provides services for storing and retrieving the EU SPL. The ApplicationDerivationProcessor is used to derive



<<subsystem>>
Application Derivation
<<entity>> EU SPL Storage (TeC PSPL)
1.2 Retrieve TeC PSPL
1.3 TeC PSPL
1. Request the EU SPL for Application Derivation
<<user interface>> EU SPL Derivation Loader
1.1 Request TeC PSPL
3.2 Extract Derived Application Specification from TeC PSPL
<<business logic>> Application Derivation Manager
<<processor>> ApplicationDerivation Processor
3.3 TeC App
2. Interact with Application Derivation Editor to Select Features
3. Submit Feature Selections
End User
3.4 Store TeC App
3.5 Ack
3.1 Feature Selections
1.5 Display TeC PSPL
2.1 Respond to the End User Input
3.7 Ack
<<user interface>> Application Derivation Editor
<<entity>> TeCApp (TeC PSP)
1.4 TeC PSPL
3.6 Ack

Figure 6.17 Application Derivation Subsystem and Component Interactions

applications from the EU SPL. The TeCApp is used to store the derived applications in the file system as JSON. The different components were created to organize the application derivation logic and obtain separation of concerns. Thus each component is responsible for specific functionality. The sections below describe the interactions of the end user with the Application Derivation subsystem in detail.

The application derivation process starts with the End User that requests from the EU SPL Derivation Loader the EU SPL to derive applications shown in step "1. Request the EU SPL for Application Derivation." The Application Derivation subsystem retrieves the EU SPL (shown in steps "1.2 Retrieve TeC PSPL" through "1.4 TeC PSPL" in Figure 6.17) and populates the Application Derivation Editor user interface with the TeC PSPL shown in step "1.5 Display TeC PSPL."

The End User interacts with the Application Derivation Editor to select the features needed for his/her smart space shown in step "2. Interact with the Application Derivation Editor to Select Features" in Figure 6.17. The Application Derivation Editor responds to the End User inputs shown in step "2.1 Respond to End User Input" with additional configuration details for selected features

The End User submits his/her feature selections to the Application Derivation Editor shown in step "3. Submit Feature Selections" in Figure 6.17 to derive an application for his/her smart space. The Application Derivation subsystem derives the application and stores it in the on the file system in JSON format (as shown in steps "3.1 Feature Selections" through "3.7 Ack" in Figure 6.17).

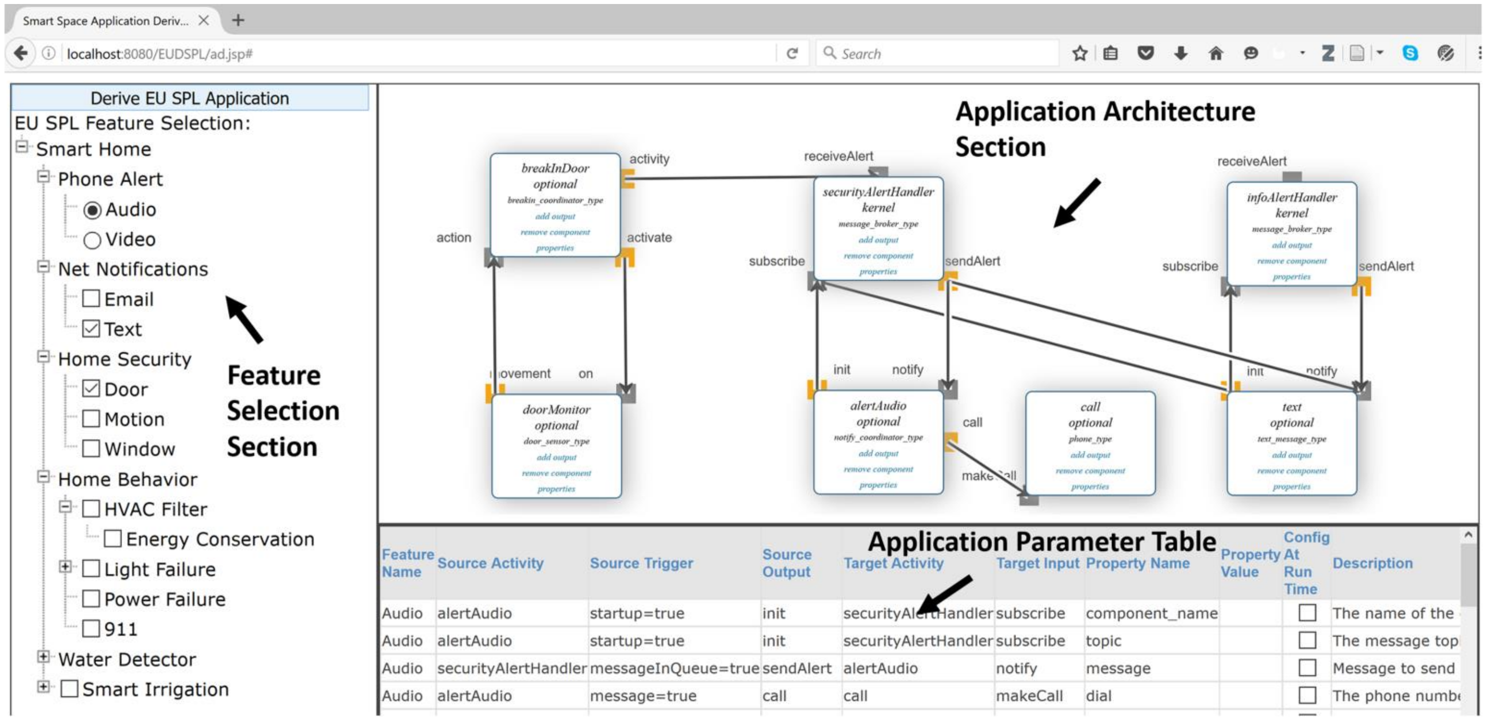
6.6.1 Application Derivation Editor

Figure 6.18 shows the user interface of the Application Derivation Editor. The user interface is divided in three main sections: (1) The Feature Selection, (2) The Application Architecture, and (3) The Application Parameter table.

6.6.1.1 Feature Selection Section

The Feature Selection Section displays the end user view of the EU SPL feature model called feature selection model. During application derivations the icon representation used during product line creation is transformed to actionable checkboxes and radio buttons that end users can use to select features for their smart spaces. The feature selection model is similar to the feature model on the EU SPL Editor and is represented as a tree data structure. The feature selection model was implemented in JavaScript by customizing and extending the TreeView (Livingston, 2002) JavaScript library. The JsTree library was also evaluated since it was used for the Feature Model Selection section of the EU SPL Editor but does not support combinations of HTML checkboxes ☑ /☐ and radio buttons ⦿/○.

In detail, the nodes of the feature selection model represent features and feature groups. Common features are not selectable and only their name is displayed on the node. The Smart Home feature shown on the Feature Selection Section in Figure 6.18 is an example of a common feature.


Smart Space Application Deriv...
localhost:8080/EUDSPL/ad.jsp#
Search
Application Architecture Section
Feature Selection Section
breakInDoor optional breakin_coordinator_type add output remove component properties
doorMonitor optional door_sensor_type add output remove component properties
securityAlertHandler kernel message_broker_type add output remove component properties
alertAudio optional notify_coordinator_type add output remove component properties
call optional phone_type add output remove component properties
infoAlertHandler kernel message_broker_type add output remove component properties
text optional text_message_type add output remove component properties
activity
action
activate
movement
on
receiveAlert
subscribe
sendAlert
init
notify
call
makeCall

Derive EU SPL Application

EU SPL Feature Selection:

- ⊟ Smart Home
  - ⊟ Phone Alert
    - ● Audio
    - ○ Video
  - ⊟ Net Notifications
    - ☐ Email
    - ☑ Text
  - ⊟ Home Security
    - ☑ Door
    - ☐ Motion
    - ☐ Window
  - ⊟ Home Behavior
    - ⊟ ☐ HVAC Filter
      - ☐ Energy Conservation
    - ⊞ ☐ Light Failure
    - ☐ Power Failure
    - ☐ 911
  - ⊞ Water Detector
  - ⊞ ☐ Smart Irrigation

Application Parameter Table

Feature Name	Source Activity	Source Trigger	Source Output	Target Activity	Target Input	Property Name	Property Value	Config At Run Time	Description
Audio	alertAudio	startup=true	init	securityAlertHandler	subscribe	component_name		☐	The name of the
Audio	alertAudio	startup=true	init	securityAlertHandler	subscribe	topic		☐	The message top
Audio	securityAlertHandler	messageInQueue=true	sendAlert	alertAudio	notify	message		☐	Message to send
Audio	alertAudio	message=true	call	call	makeCall	dial		☐	The phone numb

Figure 6.18 Application Derivation Editor User Interface

Optional default and platform dependent optional default features are displayed as checked checkboxes on the Feature Selection Section. The Text and Door features shown on the Feature Selection Section in Figure 6.18 are examples of optional default features displayed as checked checkboxes. Similar optional features and platform dependent optional features are displayed as non-checked checkboxes. The Email and Motion features shown in Figure 6.18 are examples of optional features displayed as non-checked checkboxes.

Alternative default features and platform dependent alternative default features are displayed as selected radio buttons. The Audio feature shown in Figure 6.18 is an example of an alternative default feature. Alternative features and platform dependent alternative features are displayed as non-selected radio buttons. The Video feature shown on Figure 6.18 is an example of a platform dependent alternative feature.

Feature groups appear as non-selectable and are used for grouping a set of features. The Phone Alert, Net Notifications, Home Security, Home Behavior and Water Detector feature groups shown on Figure 6.18 are examples of how features groups are displayed on the feature selection model. End users can change the default options and select the feature combinations needed for their spaces.

6.6.1.2 Application Architecture Section

The Application Architecture section is used to display the cumulative component/connector architecture for all features selected by the end user. This section utilizes the same interface as the one used on the Feature Architecture Section of the EU SPL editor. As end user select features in the feature selection section of the Application

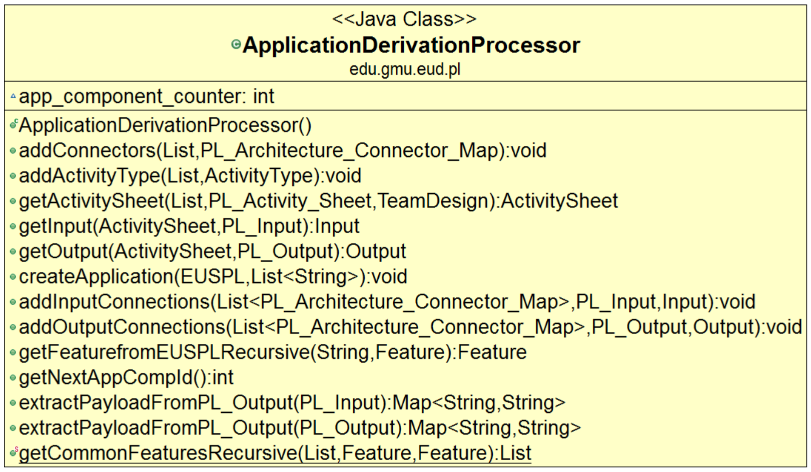
Derivation Editor the application architecture is shown in the Application Architecture section shown in Figure 6.18 In detail, the EUSPLP environment derives the component/connector architecture for the selected features and sends them as JSON objects to the Application Derivation editor. The editor draws the components and connectors on the Application Architecture section using the jsPlumb JavaScript framework.

6.6.1.3 Application Parameter Table

The Application Parameter Table section specifies all the derived application parameters that need to be configured by end users. Similar as the parameter table in the EU SPL Editor the application parameter user interface is created by extending the editablegrid JavaScript libraries. The Application Parameter Table displays all component connector properties applicable to the selected features in the feature selection model. The parameter table gets auto-populated as end users select features in the Feature Selection section.

6.6.2 Application Derivation Processor

The purpose of the ApplicationDerivationProcessor is to compose the Java object representation of the TeC application architecture based on features selected by end users. This section describes the approach followed to compose the TeC application architecture. In detail, the Application Derivation Manager sends the EU SPL and feature name selections to the ApplicationDerivationProcessor class to extract the TeC Application model. Figure 6.19 shows the main methods of the ApplicationDerivationProcessor. In detail, the createApplication method starts the TeC Application extraction. The



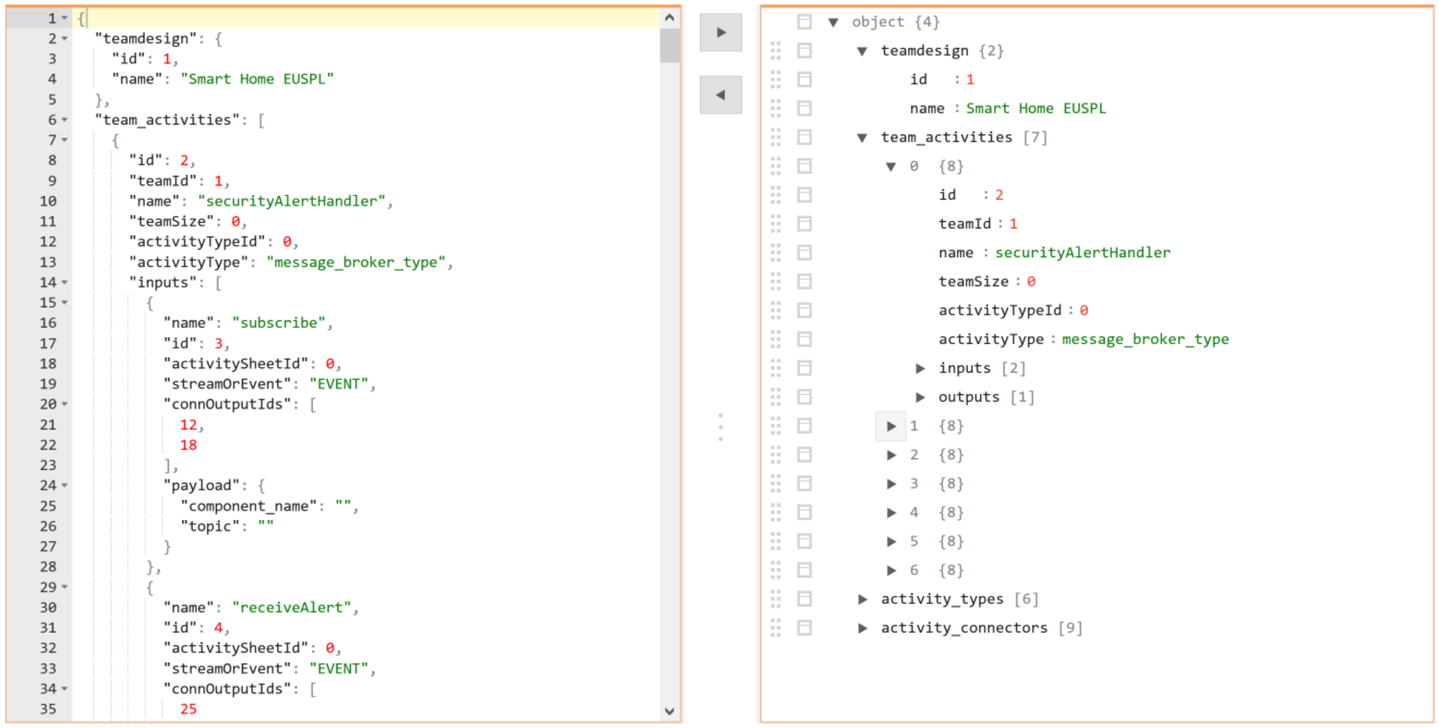
```
<<Java Class>>
ApplicationDerivationProcessor
edu.gmu.eud.pl
app_component_counter: int
ApplicationDerivationProcessor()
addConnectors(List,PL_Architecture_Connector_Map):void
addActivityType(List,ActivityType):void
getActivitySheet(List,PL_Activity_Sheet,TeamDesign):ActivitySheet
getInput(ActivitySheet,PL_Input):Input
getOutput(ActivitySheet,PL_Output):Output
createApplication(EUSPL,List<String>):void
addInputConnections(List<PL_Architecture_Connector_Map>,PL_Input,Input):void
addOutputConnections(List<PL_Architecture_Connector_Map>,PL_Output,Output):void
getFeaturefromEUSPLRecursive(String,Feature):Feature
getNextAppCompId():int
extractPayloadFromPL_Output(PL_Input):Map<String,String>
extractPayloadFromPL_Output(PL_Output):Map<String,String>
getCommonFeaturesRecursive(List,Feature,Feature):List
```

Figure 6.19 Methods of the ApplicationDerivationProcessor Class

createApplication calls the getCommonFeaturesRecursive method to get the Java object representation of the product line common features. After the common features are retrieved the getFeaturefromEUSPLRecursive method is called to get the Java object representation of the selected features. For each feature the activity sheet is extracted from the PL_Activity_Sheet through the addActivitySheet method. For each activity sheet the activity type is extracted through the addActivityType method. In addition for each activity sheet inputs and outputs are extracted through the getInput and getOutput methods. The payloads for inputs and outputs are extracted through the extractPayloadFromPL_Output and extractPayloadFromPL_Input methods. The addInputConnections and the addOutputConnections methods add other activity sheets connecting output and input ids respectively. The addConnectors method adds all activity sheet connectors to the TeC application. The getNextAppCompId method generates temporary IDs for components/inputs/outputs and payloads needed to link them together.

6.6.3 TeC Application JSON Representation

The ApplicationDerivationProcessor sends the derived Java object representation of the TeC application to the Application Derivation Manager which then get Serialized as JSON in the file system. Figure 6.20 shows part of the TeC Application JSON representation created by the feature selections shown in Figure 6.18. The left side of Figure 6.20 displays the row JSON format of the TeC Application as it is stored in the file system. The right side of Figure 6.20 displays the TeC Application in a more readable format. The main properties of the TeC application JSON shown in Figure 6.18 are: teamdesign, team_activities, activity_types and activity_connectors. The team design captures the ID and the name of the TeC application. As shown in Figure 6.20, the name of the team is Smart Home EUSPL. The team_activities property is an array that contains activity sheets. Activity sheets are TeC components. The team_activities array contains seven activity sheets: securityAlertHandler, infoAlertHandler, alertAudio, call, text, doorMonitor, breakInDoor shown on the Application Architecture section in Figure 6.18. The activity_types array captures the types of the activity sheets. Figure 6.20 shows that there are six activity types in the activity_types. The message-broker activity type is being used by the securityAlertHandler and the infoAlertHandler activity sheets. The activity_connectors array capture the input/output connectivity information between activity sheets. Figure 6.20 shows ten activity connectors which are consistent with the connectors shown on in Figure 6.18. The JSON TeC Application representation is stored on the file system and gets accessed by the Application Distribution subsystem to distribute the TeC Application to the end user TeC platform.



```
{
  "teamdesign": {
    "id": 1,
    "name": "Smart Home EUSPL"
  },
  "team_activities": [
    {
      "id": 2,
      "teamId": 1,
      "name": "securityAlertHandler",
      "teamSize": 0,
      "activityTypeId": 0,
      "activityType": "message_broker_type",
      "inputs": [
        {
          "name": "subscribe",
          "id": 3,
          "activitySheetId": 0,
          "streamOrEvent": "EVENT",
          "connOutputIds": [
            12,
            18
          ],
          "payload": {
            "component_name": "",
            "topic": ""
          }
        },
        {
          "name": "receiveAlert",
          "id": 4,
          "activitySheetId": 0,
          "streamOrEvent": "EVENT",
          "connOutputIds": [
            25
```

```
▼ object {4}
    ▼ teamdesign {2}
        id   : 1
        name : Smart Home EUSPL
    ▼ team_activities [7]
        ▼ 0  {8}
            id   : 2
            teamId : 1
            name : securityAlertHandler
            teamSize : 0
            activityTypeId : 0
            activityType : message_broker_type
          ▶ inputs [2]
          ▶ outputs [1]
        ▶ 1  {8}
        ▶ 2  {8}
        ▶ 3  {8}
        ▶ 4  {8}
        ▶ 5  {8}
        ▶ 6  {8}
    ▶ activity_types [6]
    ▶ activity_connectors [9]
```

Figure 6.20 Sample TeC PSP JSON Representation

6.7 End User Application Deployment

During application deployment, end users deploy the derived application from the EUSPLP to their TeC environment. Figure 6.21 shows the physical deployment of the different systems used in this prototype and the event sequence between the different subsystems to deploy an end user application. As shown on Figure 6.21, the EUSPLP is deployed on the Tomcat JEE container. Tomcat is deployed in a Windows environment. The Application Distributor subsystem handles requests to distribute the derived application specification through REST services. The Application Distributor subsystem is composed of two components, ApplicationPublisher and TeCApp. The ApplicationPublisher provides services for sending the TeC application (PSP) to an external system. The TeCApp is used to retrieve a derived application from the file system.

The TeC EUSPLP Adaptor and the TeC simulator are deployed to the end user environment on an Android platform. The TeC EUSPLP Adaptor subsystem is designed to be an extension to TeC environments. The purpose of the TeC EUSPLP Adaptor is to retrieve, configure and store the TeC applications (PSPs) derived from the EUSPLP environment to the TeC simulator. The EUSPLP adaptor subsystem is composed of two components, EUSPLP Manager and TeCAppImporter. EUSPLP Manager provides the user interface to end users to import derived applications form the EUSPLP. The TeCAppImporter provides the services for communication with the EUSPLP to retrieve the EU SPL and the TeC environment to store the derived application.

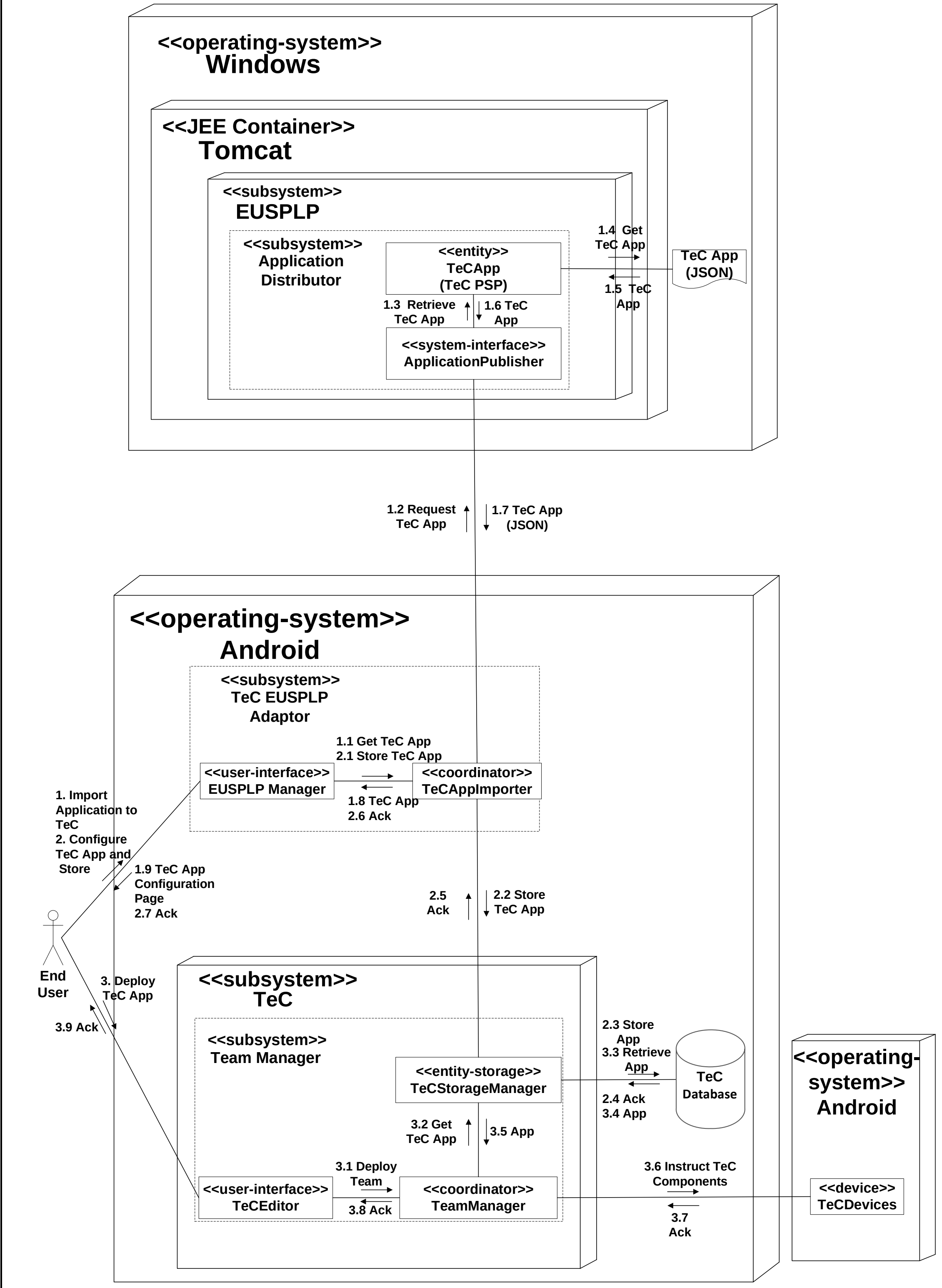


Figure 6.21 Application Deployment Diagram

The TeC simulator (Shen, 2014) used in this research simulates TeCDevices running as different Android instances. The TeC database used by the simulator is also running in Android. For application deployment there are three components used in the TeC simulator: (1) TeCEditor, (2) TeCStorageManager, and (3) TeamManager. The TeCEditor provides the user interface for designing and deploying TeC applications. The TeCStorageManager is used for the storage and retrieval of TeC applications. The TeamManager is responsible for deploying TeC devices deployed in a smart space with application instructions. There are several reasons for separating the Application Distributor and TeC EUSPLP Adaptor subsystems. One of the main reasons is the separation of concerns between retrieving the derived application and configuring/storing it to the target system. By separating the two subsystems the Application Distributor does not need to have information about how to store derived applications to different TeC environments. Another reason is that the EUSPLP Adaptor can be specific to an operating system, hardware etc. For example consider an EUSPLP Adaptor for a TeC system deployed in Windows versus Android. Finally the EUSPLP Adaptor could be extended to map TeC applications to other EUD environments for smart spaces similar to Jigsaw. The sections below discuss in detail the application deployment process.

Application deployment starts with end users that interact with the TeC EUSPLP Adaptor to import an application from EUSPLP to TeC as shown in steps “1. Import Application to TeC” through “1.9 TeC App Configuration Page” in Figure 6.21 End Users configure the derived application and submit their selections to the TeC EUSPLP Adaptor to store the application to the TeC environment shown in step “2. Configure TeC App and

Store” through “2.7 Ack” in Figure 6.21. End Users interact with the TeC subsystem to deploy the derived application to the TeC environment as shown in steps “3. Deploy TeC App” through “3.9 Ack” in Figure 6.21.

6.8 Summary

This chapter has described the EUSPLP development environment that was created as part of this research and described how it can be used to support the development of EU SPLs, application derivation and application deployment for end user smart spaces. In summary, the chapter described the use cases that EUSPLP implements. The overall EUSPLP subsystem architecture was presented to show the interactions between different subsystems that implement the use cases. The EUSPLP and TeC physical data model sections described the meta-classes and their relationships used by the prototype to capture end user product lines and derived applications. The EUSPLP EU SPL Development section described the processes, user interface and artifacts used by EU SPL designers to create or edit EU SPLs. The End User Application Derivation section describes the processes, user interface and artifacts used by End Users to derive applications from EU SPLs. Finally the End User Application Deployment section described the deployment of derived applications to the TeC environment.

7 RESEARCH VALIDATION

7.1 Introduction

This chapter describes the validation approach used in this research. The Smart Home EU SPL case study was used in the validation of: the End User Software Product Line (EU SPL) Process and the EUSPLP development environment. The EUSPLP environment was used to validate the EU SPL process and meta-model by enabling the creation of the EU SPL, from which EU applications were derived.

As part of this research an EU SPL Testing Approach was defined with corresponding tool support to test the TeC EUD platform specific SPL and TeC EUD platform specific applications. The testing approach consists of: (a) EU SPL Testing to test the TeC SPL, (b) EU Application Testing to test the derived TeC application, and (c) EU Application Deployment Testing to test the deployment of the TeC application. To perform EU SPL Testing and EU Application Testing the following tools were developed by this research: (a) ConsistencyRuleChecker, (b) FeatureBasedTestDriver, and (c) TeC interpreter. Finally, as part of EU Application Deployment Testing, the TeC Android simulator (Sousa et al., 2012) was used to test the distributed deployment and execution of derived applications in the TeC platform.

The chapter is organized as follows: section 7.2 describes the overall validation approach as it relates to the research problem. Section 7.3 describes the testing framework developed by this research to test EU SPLs and derived applications. Section 7.4 describes the overall EU SPL testing approach used in this research. Section 7.5 describes the testing process for testing the EU SPL created by using the EUSPLP environment. Section 7.6 describes the testing process for testing end user applications derived using the EUSPLP environment. Section 7.7 describes the deployment, execution and testing of derived end user applications by the TeC Android simulator. Finally section 7.8 provides a summary of this chapter.

7.2 Research Validation Approach

This research is validated through the implementation and testing of the Smart Home EU SPL case study described in Appendix A. The case study was designed using the EU SPL process described in Chapter 4 and was implemented using the EUSPLP environment described in Chapter 6. The remainder of this section describes the validation process:

1. Designed the Smart Home EU SPL case study using the End User Product Line Engineering (EUPLE) process described in Chapter 4. The design included:
    - Feature Modeling – A feature model was created for the Smart Home EU SPL case study. (Section 4.3.2.4 - Chapter (4).
    - Static Modeling – A static model was created with all components that realize the Smart Home EU SPL. (Section 4.3.3.1 - Chapter (4).
    - Dynamic Modeling – Sequence diagrams and a Feature / Component relationship table was developed for each feature defined in the Smart Home EU. (Sections 4.3.3.2/4.3.3.3 - Chapter (4).
    - Component Modeling – Component diagrams were developed for all features of the Smart Home EU SPL. A Component Input / Output table

was created to capture the input / output parameters and triggering conditions of each component. (Section 4.3.4.2 in Chapter (4).

- Inter-feature Component Communication Modeling – A component association table was created to capture components of the Smart Home case study that use the subscription/notification design pattern to communicate with components that realize other features. (Section 4.3.4.1 in Chapter (4).
- Platform Specific Feature/Component Modeling – A Feature / Component association table was created that captures platform specific component information for platform specific features in the Smart Home EU SPL. (Section 4.3.4.3 in Chapter (4)

2. Derived two end user TeC applications from the Smart Home EU SPL case study developed in the previous step. For the first end user application the application models (PSPs) were created for both the TeC and Jigsaw end user platforms (Section 4.4 in Chapter (4). The second end user application was platform specific and the application model was developed for the TeC platform (Sections A.5.4 and A.5.5 in Appendix-A).

3. Developed the EUSPLP development environment, which supports the development of EU SPLs, and application derivation. The EUSPLP was created based on the EU SPL process and meta-models described in Chapters 4 and 5.

The EUSPLP environment was used as follows:

- To implement several experimental EU SPLs, including the Smart Home EU SPL case study, using the EU SPL Editor subsystem of the EUSPLP environment. The EUSPLP environment produces Platform Independent Product Line (PIPL) and TeC Platform Specific Product Line (PSPL) specifications to store the EU SPLs created by the EU SPL Editor.
- To derive applications from several EU SPLs including the Smart Home EU SPL case study using the Application Derivation Editor subsystem of the EUSPLP environment. During application derivation, the EUSPLP environment produces the TeC Platform Specific Product (PSP) specification.

4. Developed a testing approach to test TeC PSPLs and TeC PSPs created by the EUSPLP environment. The testing approach is used to perform EU SPL Testing, EU Application Testing and EU Application Deployment Testing. During EU SPL Testing, EU SPL Feature-based Consistency and Feature-based Integration test cases are used to test the TeC SPL. During EU Application testing, EU Application Feature-based Consistency and Feature-based Integration test cases derived from the EU SPL are used to test the TeC PSP. During EU Application Deployment Testing Feature-based Integration tests are executed to deployed application. The TeC PSPL created using the EUSPLP environment to represent the Smart Home EU SPL was tested using EU SPL Testing. Two end user applications (TeC PSPs) derived from the Smart Home EU SPL were tested using EU Application Testing.

In addition a third end user application was derived from the Smart Home EU SPL that was tested using EU Application deployment testing. The remainder of this chapter describes the testing framework in detail.

5. Deployed several experimental applications (TeC PSPs) to the TeC Android simulator. In addition a TeC PSP derived from the Smart Home EU SPL case study was also deployed to the TeC Android simulator.

7.3 EU SPL Testing Framework

To validate that the EUSPLP development environment produces valid EU SPL specifications (PSPLs) and derives applications (PSPs) that can be executed by a TeC platform, a testing framework was created. The testing framework is composed of a set of tools to assist with test automation. The tool set can be divided into two categories: (1) Consistency rule checking and (2) Feature-based integration testing.

Consistency rule checking is used to ensure that the structure of the EU SPL is compliant with the product line consistency rules described in detail in section in 7.5.1 and that features selected from the EU SPL during application derivation are compliant with the feature set consistency rules described in section 7.6.1. As part of this research, the ConsistencyRuleChecker Java program was created to execute consistency rule checks on the EU SPL and features selected. To perform consistency rule checking on the EU SPL, the ConsistencyRuleChecker program takes as input the EU SPL JSON representation and executes the product line consistency rules. To perform consistency rule checking for an EU SPL feature selection, the ConsistencyRuleChecker program takes as input: (a) an array containing the names of the selected features and (b) the EU SPL JSON representation. The ConsistencyRuleChecker program checks that the selected features do not violate any of the feature dependency and feature group relationships in the EU SPL.

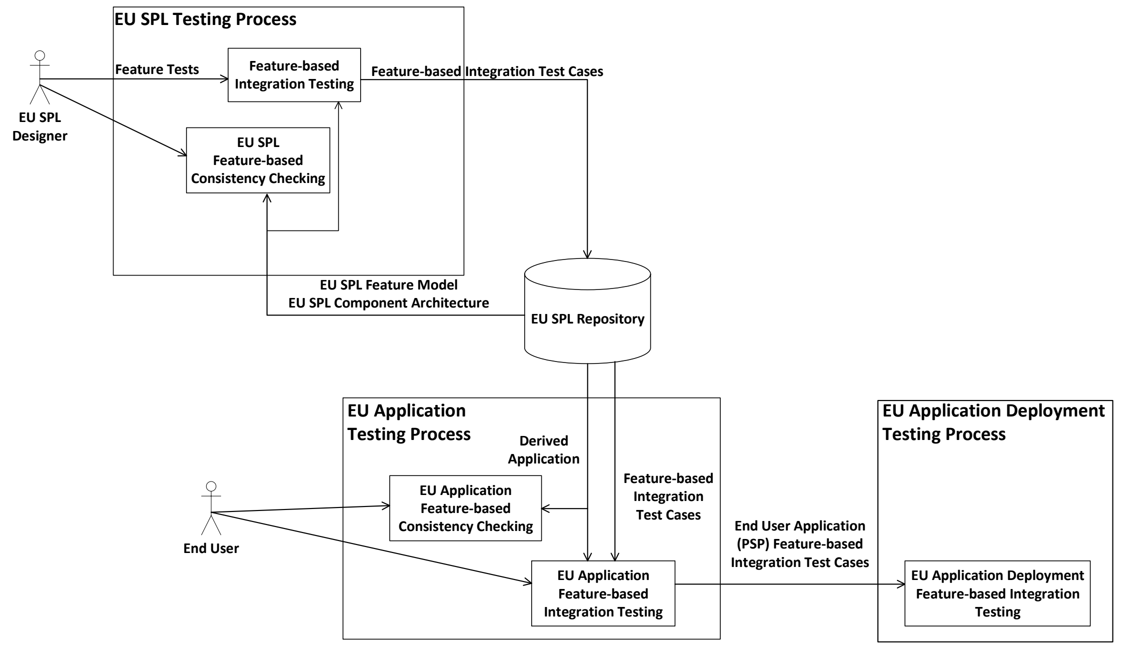
Feature-based integration testing is used to test the implementation of the component architecture of: (a) EU SPL features and feature combinations, (b) applications derived from the EU SPL, and (c) application deployment. As part of this research, two tools were developed to support the automation of Feature-based integration testing for EU SPL features and derived applications: (1) FeatureBasedTestDriver and (2) TeC interpreter. The FeatureBasedTestDriver is a Java program developed in Windows that reads feature-based test cases from the file system, instantiates the corresponding component architecture in the TeC interpreter, executes the test cases in the TeC interpreter, and evaluates the test results. The TeC interpreter is a Java program that instantiates and executes the component implementation of EU SPL features and derived applications. The FeatureBasedTestDriver is used to execute feature-based test cases by simulating external events input to the TeC interpreter. The TeC interpreter, based on each event, executes the appropriate components and component connectors.

To perform feature-based integration testing on a distributed platform, the TeC Android simulator (Sousa et al., 2012) was used. Tzeremes developed the user interface (TeCEditor) and TeC meta-model to develop end user applications in the TeC Android simulator. In particular, the TeCEditor provides user interfaces to create, display and edit available TeC applications in the simulator and their component architecture. Applications derived from the EU SPL and imported in the TeC Android simulator appear in the TeCEditor. During application deployment testing, the TeCEditor was used to ensure that derived applications were imported correctly into the TeC Android simulator. Shen and Hodum developed the TeC application execution. Shen developed a testing interface in the Android simulator to simulate external events. Shen’s user interface was used to manually execute feature-based integration tests in the TeC Android platform.

7.4 EU SPL Testing Approach

As part of this research an overall testing approach was defined to test EU SPLs and derived applications. The EU SPL Testing Approach is a hybrid approach that builds on the testing methods described in the theses of (Abu-Matar, Mohammad Ahmad, 2011) and (Olimpiew, 2008). Abu-Matar used static defined SPL consistency test cases to test SPLs and derived applications created in his research (Abu-Matar and Gomaa, 2013). Olimpiew described an approach for defining test cases for each feature that can be retrieved and executed during application derivation (Olimpiew and Gomaa, 2009). Similarly, the test cases created in this research consist of: consistency test cases for testing the EU SPL and the derived applications; and test cases for each feature that can be executed during product line creation, application derivation and application deployment.

Figure 7.1 shows the overall EU SPL Testing Approach used to test EU SPLs and derived applications. The testing approach is composed of: (a) the EU SPL Testing, (b) the EU Application Testing, and (c) the EU Application Deployment Testing processes. The EU SPL Testing process is responsible for testing the product line. The EU SPL Testing process performs EU SPL Feature-based Consistency Checking and Feature-based Integration Testing. EU SPL Feature-based Consistency Checking executes static test cases



EU SPL Testing Process
EU SPL Designer
Feature Tests
Feature-based Integration Testing
Feature-based Integration Test Cases
EU SPL Feature-based Consistency Checking
EU SPL Feature Model
EU SPL Component Architecture
EU SPL Repository
EU Application Testing Process
Derived Application
EU Application Feature-based Consistency Checking
Feature-based Integration Test Cases
End User
EU Application Feature-based Integration Testing
End User Application (PSP) Feature-based Integration Test Cases
EU Application Deployment Testing Process
EU Application Deployment Feature-based Integration Testing

Figure 7.1 Overall EU SPL Testing Approach

to verify feature and feature group dependencies. Feature-based Integration consists of integration test cases defined by EU SPL designers to test the EU SPL. In particular, integration test cases are developed for every feature and feature combination in the EU SPL to test the component interconnections. As shown in Figure 7.1 Feature-based Integration test cases are stored in the EU SPL Repository for later usage during application derivation.

The EU Application Testing Process is responsible for testing the applications derived from the EU SPL based on feature selected from the product line. The EU Application Testing consists of EU Application Feature-based Consistency Checking and EU Application Feature-based Integration Testing. EU Application Feature-based Consistency Checking contains static test cases used to verify the compatibility of features that comprise the derived application. EU Application Feature-based Integration involves executing integration test cases to test the component architecture and implementation of the derived application. The integration test cases are a subset of the EU SPL integration test cases that are based on the selected features that comprise the derived application. As shown in Figure 7.1, Feature-based Integration test cases to test the derived application are selected from the EU SPL Repository corresponding to the features selected by the end user.

The EU Application Deployment Testing Process shown in Figure 7.1, is responsible for testing the distributed deployment and execution of the TeC derived application. In detail, during the deployment testing process, EU Application Deployment Feature-based Integration Testing involves executing integration test cases to test the deployment and execution of components and their interconnections in the environment. The integration test cases are the same ones used during EU Application Feature-based Integration Testing. The integration test cases are reused to test the deployment of the derived application.

The Feature-based integration test cases provide test coverage of each feature and component during EU SPL Testing, EU Application Testing and EU Application Deployment Testing. In particular test cases are developed to: (a) test each component (b) test each feature by testing the components and connectors that realize the feature (c) If a feature depends on other features, test the feature in combination with the features it depends on. Detailed examples of the execution of feature-based integration test cases and test criteria are described in sections 7.5.2, 7.6.2, 7.6.3 and 7.6.4

7.5 End User Software Product Line (EUSPL) Testing Process

There were two types of tests performed on the EU SPL (PSPL) produced by the EUSPLP environment: (a) EU SPL Feature-based Consistency Checking, and (b) Feature-based Integration Testing. EU SPL Feature-based Consistency Checking ensures that the EU SPL is a valid product line. For instance these types of tests validate: (a) the consistency between the product line features and the components that realize them, (b) the consistency between feature groups and the features they contain, and (c) the consistency between features and features they depend on. Feature-based Integration Testing ensures that: (a) the visual EU SPL representation in the prototype is consistent with the TeC SPL model produced by the EUSPLP environment, and (b) the component architecture functions as the EU SPL designer intended.

7.5.1 EU SPL Feature-based Consistency Checking

To perform the EU SPL Feature-based Consistency Checking, three types of EU SPL Feature-based Consistency Test Cases were developed: (1) Feature to Component Consistency tests, (2) Feature Group to Feature Consistency tests, and (3) Feature Dependency Consistency tests. All test cases execute independently of each other. Table 7.1 to Table 7.3 show the test cases in detail. Each of the tables has 3 columns: (1) Test Case, (2) Expected Result, and (3) Test Result. The Test Case column shows the test case. The Expected Result column shows the expected result of the test case after it executes. The Test Result column shows the result found when the test case was executed.

Feature to Component Consistency tests verify that the feature type variability is consistent with the component types that realize each feature. For instance, an optional feature should not contain kernel components. Feature Group to Feature Consistency tests verify that the feature group type is consistent with each feature variability type contained in that group. For instance none of the following feature groups: At-least-one-of, Exactly-one-of, Zero-or-more-of and Zero-or-one-of should contain common features. Table 7.2 show all the feature group to feature consistency tests Feature Dependency Consistency tests verify that each feature depends on a feature with compatible feature type. For instance it is not valid to have a common feature depend on an optional feature. Table 7.3 shows all the feature dependency consistency tests.

To test that the EU SPL Feature-based Consistency Test Cases themselves execute correctly, a "Valid EU SPL" and an ""Invalid EU SPL" were defined. The "Valid EU SPL" contained valid feature to component dependencies, valid features under feature groups and valid feature to feature dependencies. The purpose of the "Valid EU SPL" was to evaluate that all positive tests defined in Table 7.1 to Table 7.3 were executed correctly.

Table 7.1 Feature to Component Consistency Tests

Test Case	**Expected Result**	**Test Result**
Common Feature contains Kernel Component	Pass	Pass
Common Feature contains Optional Component	Fail	Fail
Common Feature contains Variant Component	Fail	Fail
Optional Feature contains Kernel Component	Fail	Fail
Optional Feature contains Optional Component	Pass	Pass
Optional Feature contains Variant Component	Fail	Fail
Alternative Feature contains Kernel Component	Fail	Fail
Alternative Feature contains Optional Component or Variant Component	Pass	Pass

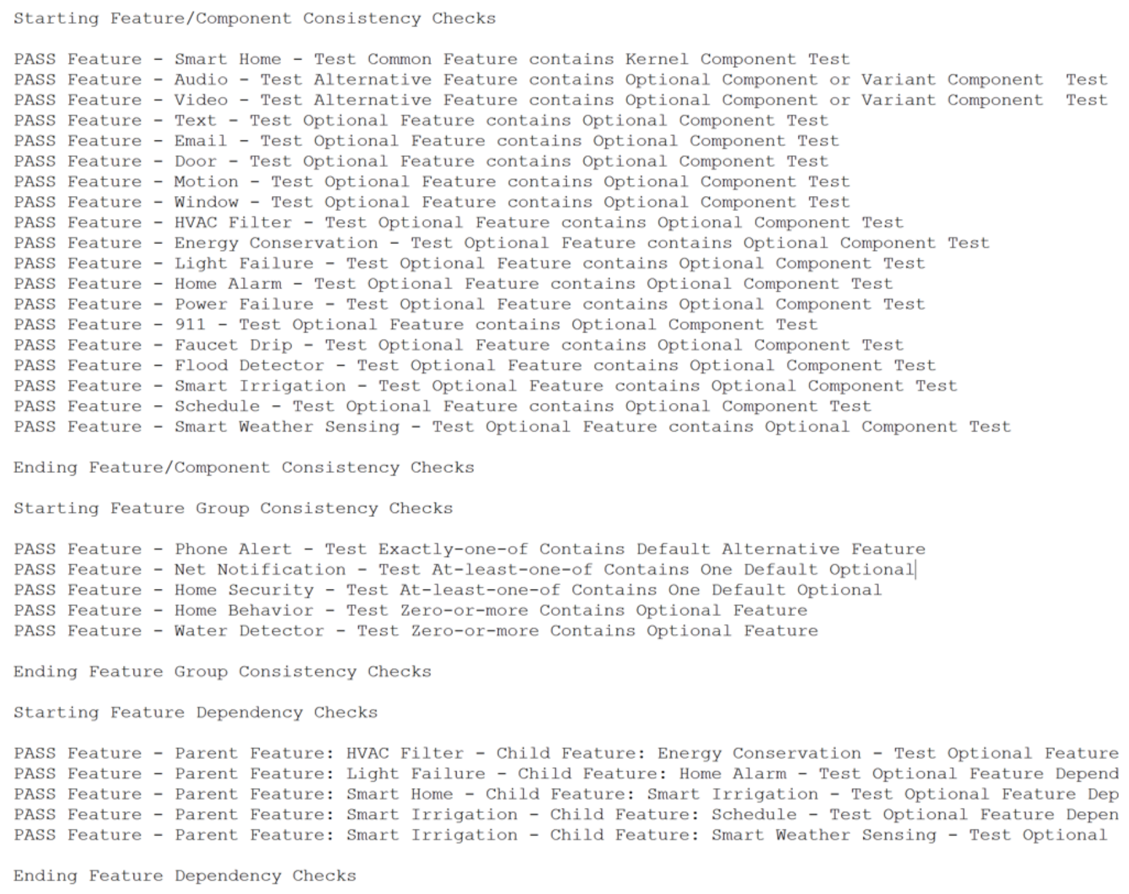
Table 7.2 Feature Group to Feature Consistency Tests

Test Case	Expected Result	Test Result
At-least-one-of Feature Group contains one Default Optional Feature	Pass	Pass
At-least-one-of Feature Group contains zero or more than one Default Optional Feature	Fail	Fail
At-least-one-of Feature Group contains Common, Default Alternative or Alternative Features	Fail	Fail
At-least-one-of Feature Group contains Common, Default Alternative or Alternative Features	Fail	Fail
Zero-or-more-of Feature Group contains Common Feature	Fail	Fail
Zero-or-more-of Feature Group contains Optional Feature	Pass	Pass
Zero-or-more-of Feature Group does not contain Optional Feature	Fail	Fail
Zero-or-more-of Feature Group contains Common, Default Optional, Default Alternative or Alternative Features	Fail	Fail
Zero-or-one-of Feature Group contains Alternative Feature	Pass	Pass
Zero-or-one-of Feature Group does not contain Alternative Feature	Fail	Fail
Zero-or-one-of Feature Group contains Default Optional, Optional or Alternative Feature	Fail	Fail
Exactly-one-of Feature Group contains Default Alternative Feature	Pass	Pass
Exactly-one-of Feature Group does not contain Default Alternative Feature	Fail	Fail
Exactly-one-of Feature Group contains Common, Default Optional or Optional Features	Fail	Fail
Exactly-one-of Feature Group contains zero or more than one Default Alternative Feature	Fail	Fail

Table 7.3 Feature Dependency Consistency Tests

Test Case	Expected Result	Test Result
Common Feature depends on Common Feature	Pass	Pass
Optional Feature depends on Common Feature	Pass	Pass
Alternative Feature depends on Common Feature	Pass	Pass
Common feature depends on Optional feature	False	False
Optional Feature depends on Optional Feature	Pass	Pass
Alternative Feature depends on Optional Feature	Pass	Pass
Common Feature depends on Alternative Feature	False	False
Optional Feature depends on Alternative Feature	Pass	Pass
Alternative Feature depends on Alternative Feature	Pass	Pass

The “Invalid EU SPL” contained invalid feature to component dependencies, invalid features under feature groups and invalid feature dependencies. The purpose of the “Invalid EU SPL” was to evaluate that all negative tests defined in Table 7.1 to Table 7.3 were executed correctly. After each test case was verified against the “Valid EU SPL” and the “Invalid EU SPL,” EU SPL Feature-based Consistency Checking was performed against the EU SPL produced by the EUSPLP environment for the Smart Home Case study. All positive tests defined on Table 7.1 to Table 7.3 were executed correctly. Figure 7.2 shows the output of the ConsistencyRuleChecker executing EU SPL Feature-based Consistency Checking test cases to the Smart Home EU SPL.



```
Starting Feature/Component Consistency Checks

PASS Feature - Smart Home - Test Common Feature contains Kernel Component Test
PASS Feature - Audio - Test Alternative Feature contains Optional Component or Variant Component  Test
PASS Feature - Video - Test Alternative Feature contains Optional Component or Variant Component  Test
PASS Feature - Text - Test Optional Feature contains Optional Component Test
PASS Feature - Email - Test Optional Feature contains Optional Component Test
PASS Feature - Door - Test Optional Feature contains Optional Component Test
PASS Feature - Motion - Test Optional Feature contains Optional Component Test
PASS Feature - Window - Test Optional Feature contains Optional Component Test
PASS Feature - HVAC Filter - Test Optional Feature contains Optional Component Test
PASS Feature - Energy Conservation - Test Optional Feature contains Optional Component Test
PASS Feature - Light Failure - Test Optional Feature contains Optional Component Test
PASS Feature - Home Alarm - Test Optional Feature contains Optional Component Test
PASS Feature - Power Failure - Test Optional Feature contains Optional Component Test
PASS Feature - 911 - Test Optional Feature contains Optional Component Test
PASS Feature - Faucet Drip - Test Optional Feature contains Optional Component Test
PASS Feature - Flood Detector - Test Optional Feature contains Optional Component Test
PASS Feature - Smart Irrigation - Test Optional Feature contains Optional Component Test
PASS Feature - Schedule - Test Optional Feature contains Optional Component Test
PASS Feature - Smart Weather Sensing - Test Optional Feature contains Optional Component Test

Ending Feature/Component Consistency Checks

Starting Feature Group Consistency Checks

PASS Feature - Phone Alert - Test Exactly-one-of Contains Default Alternative Feature
PASS Feature - Net Notification - Test At-least-one-of Contains One Default Optional
PASS Feature - Home Security - Test At-least-one-of Contains One Default Optional
PASS Feature - Home Behavior - Test Zero-or-more Contains Optional Feature
PASS Feature - Water Detector - Test Zero-or-more Contains Optional Feature

Ending Feature Group Consistency Checks

Starting Feature Dependency Checks

PASS Feature - Parent Feature: HVAC Filter - Child Feature: Energy Conservation - Test Optional Feature
PASS Feature - Parent Feature: Light Failure - Child Feature: Home Alarm - Test Optional Feature Depend
PASS Feature - Parent Feature: Smart Home - Child Feature: Smart Irrigation - Test Optional Feature Dep
PASS Feature - Parent Feature: Smart Irrigation - Child Feature: Schedule - Test Optional Feature Depen
PASS Feature - Parent Feature: Smart Irrigation - Child Feature: Smart Weather Sensing - Test Optional

Ending Feature Dependency Checks
```

Figure 7.2 ConsistencyRuleChecker Output of executing EU SPL Consistency Test Cases to the Smart Home EU SPL

7.5.2 Feature-based Integration Testing

Feature-based Integration Testing is used to test the component architecture of each feature. Feature-based Integration Test Cases defined for features and feature combinations were used to perform Feature-based Integration Testing. Table 7.4 shows the attributes of each Feature-based Integration Test Case. Feature-based Integration Test Cases were defined for each connector available in each feature. The connector tests ensure that the output / input interfaces between components are consistent, and the triggering conditions are executing correctly. In addition, Feature-based Integration Test Cases were defined to test the interaction sequence of multiple components.

Table 7.4 Feature-Based Integration Testing – Test Case Attributes

Test Case Element	Description
Test Case	The test case id
Feature Name	The name of the feature that the test applies
Feature Type	The expected feature variability type
Source Component	The name of the component that initiates the component communication by sending a message when the output triggering condition is true
Source Output	The name of the source component output sending the message
Source Output Parameters	The output message parameters
Source Trigger	The trigger that activates the output on the source component
Target Component	The name of the component that receives the message
Target Input	The name of the target component input receiving the message
Test Case Result	The expected test result

For example consider the “Audio” feature defined in the Smart Home EU SPL case study. As shown in Figure 7.3 the “Audio” feature has three components: “alertAudio”, “phone” and “securityAlertHandler.” The “Audio” feature contains three connectors: the “audioAlert” to “securityAlertHandler” connector, (2) the “securityAlertHandler” to “audioAlert” connector, and (3) the “audioAlert” to “phone” connector. The first three rows of Figure 7.4 shows the Feature-based Integration Test Cases defined for each connector. For instance, the first row tests the “audioAlert” to “securityAlertHandler” connector. When the source trigger “startup=true” is true in the “alertAudio” component, the source output “init” is executed that sends parameters “component_name=alertAudio, topic=security” to the input “subscribe” of the target component “securityAlertHandler.” If the test case is executed correctly the “subscribe” input of the “securityAlertHandler” component should receive a message with parameters “component_name=alertAudio, topic=security.”

The integration test case shown in the fourth row of Figure 7.4 tests a sequence of component connectors triggered by an external event. The source component of the test case is the “securityAlertHandler” and the target component is the “phone” component. This test case tests two component connectors: (1) the “securityAlertHandler” to “audioAlert” connector, and (2) the “audioAlert” to “call” connector shown in Figure 7.3. The purpose of this test case is to test that when there is a security alert, a call is made to the house residents. This test case exercises a set of inputs, outputs, and triggering conditions in all participating components for the test case to complete successfully. For

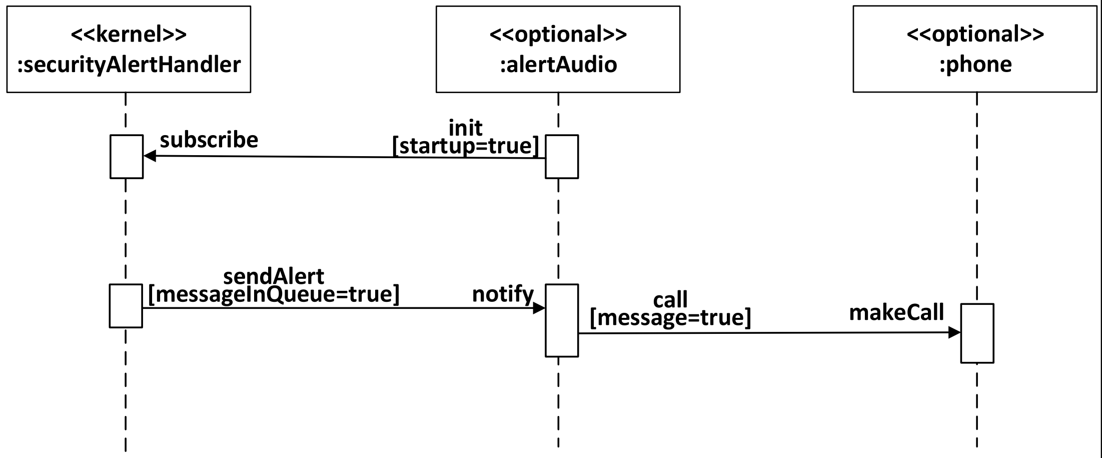
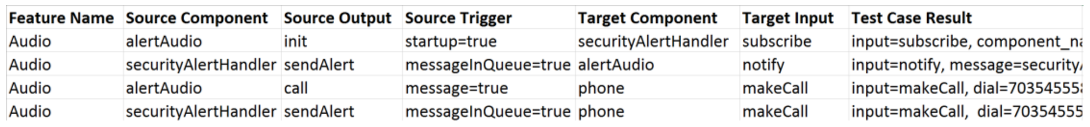


Figure 7.3 Smart Home EU SPL: Audio Feature



Feature Name	Source Component	Source Output	Source Trigger	Target Component	Target Input	Test Case Result
Audio	alertAudio	init	startup=true	securityAlertHandler	subscribe	input=subscribe, component_n
Audio	securityAlertHandler	sendAlert	messageInQueue=true	alertAudio	notify	input=notify, message=security
Audio	alertAudio	call	message=true	phone	makeCall	input=makeCall, dial=70354555
Audio	securityAlertHandler	sendAlert	messageInQueue=true	phone	makeCall	input=makeCall, dial=70354555

Figure 7.4 Audio Feature Test Cases

instance as shown in Figure 7.3, the “securityAlertHandler” needs to send a message to the “alertAudio” component. The “alertAudio” component evaluates the message and sends a message to the “phone” component.

As part of this research, a FeatureBasedTestDriver and a TeC interpreter were developed to perform feature-based testing. The FeatureBasedTestDriver is used to execute the integration test cases. The TeC interpreter is used to execute the component implementations of features and feature combinations. For example, to test the “Audio” feature shown in Figure 7.3 three testing components “securityAlertHandler,” “alertAudio”, and “phone” were implemented and executed by the TeC interpreter. Each component implementation contains: (a) methods that simulate the component inputs, (b) a method “evaluateTrigeringConditions” that executes the component triggering conditions, and (c) a “testResult” variable that captures the parameters passed in each component input. For example, for the “securityAlertHandler” component shown in Figure 7.3, an input “subscribe” was created and a method “evaluateTrigeringConditions” that executes the “messageInQueue=true” triggering condition. The “subscribe” input when called populates the “testResult” variable with the parameters that were passed to the input.

The FeatureBasedTestDriver for each integration test case extracts the component implementations of the corresponding feature(s) from the TeC PSPL. It then interfaces with the TeC interpreter to provide the test components with outputs, triggering conditions and component connectors that realize each feature.

The FeatureBasedTestDriver executes the triggering condition defined in the test case by calling the “evaluateTriggeringConditions” method on the source component in the TeC interpreter. The “evaluateTriggeringConditions” method will evaluate the triggering conditions of each output and if the condition is true it will execute the output. After the TeC interpreter executes the triggering condition, the FeatureBasedTestDriver will query the “testResult” of the target component in TeC interpreter. The FeatureBasedTestDriver compares the “testResult” variable with the expected test case result defined in the test case to verify that the parameters passed to the target object are what were expected.

For example, for the FeatureBasedTestDriver to execute the first test case shown in Figure 7.3, the three “securityAlertHandler”, “alertAudio”, and “phone” test components need to be instantiated and executed by the TeC interpreter. The FeatureBasedTestDriver interfaces with the TeC interpreter to execute the “message=true” triggering condition on the “alertAudio” component and retrieve the “testResult” variable on the “phone” component that contains the test results. The FeatureBasedTestDriver compares the “testResult” variable with the test case expected result to ensure that the correct input was called and the correct parameters were passed to it.

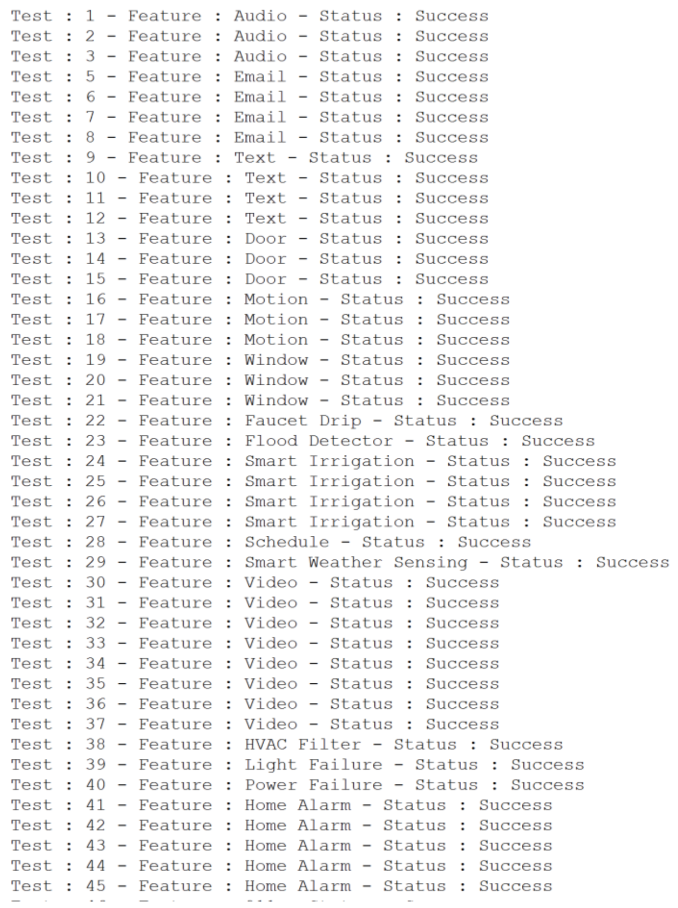
The execution of the feature-based test cases ensures that: (a) the visual representation of the component designs that realize each feature in the EU SPL is consistent with the TeC PSPL specification produced by the EUSPLP environment, (b) the component architecture is communicating as expected, and (c) the component implementations in the TeC interpreter are consistent with the component interfaces in the EUSPLP environment. To evaluate the execution of the FeatureBasedTestDriver and the TeC interpreter, valid and invalid test cases were developed. Valid test cases contained features, components, inputs, outputs and triggering conditions consistent with the TeC PSPL. Invalid test cases contained features that did not exist, components with incorrect inputs, outputs and triggering conditions. All valid test cases executed correctly and invalid test cases failed as expected. The FeatureBasedTestDriver and TeC interpreter helped to identify issues with (a) missing inputs from the component implementations, (b) triggering conditions not implemented correctly, and (c) PSPL specifications that were invalid (such as missing PL_Activity_Sheets, component connectors, inputs, outputs, output parameters, invalid JSON etc.).

To validate the Smart Home EU SPL case study, Feature-based Integration Test Case Test Cases were defined for (a) testing all connectors on all features defined in the

Smart Home EU SPL, (b) testing multi-component interactions of dependent features, and (c) testing multi-component interactions of features that are not dependent in the feature model level but an event on one feature affects the other. The Feature-based Integration Test Case Test Cases were executed using the FeatureBasedTestDriver and the TeC interpreter. All test cases were executed successfully. Figure 7.5 shows part of the FeatureBasedTestDriver output of the Smart Home EU SPL case study.

7.6 End User Application Testing Process

There are two types of tests performed to applications derived from the EUSPLP environment: (a) EU Application Feature-based Consistency Checking, and (b) EU Application Feature-based Testing. EU Application Feature-based Consistency Checking ensures that the feature selection is valid and the features selected are compatible with each other. For example, a feature selection that contains two mutually exclusive features is not valid. EU Application Testing ensures that: (a) the derived application component architecture adheres to the selected feature component architectures, and (b) the application component architecture functions correctly. To validate the application derivation process of the EUSPLP environment, EU Application Testing was performed on the “Smart Home Example 1 for TeC” and “Smart Home Example 2 for TeC” end user applications derived from the Smart Home EU SPL. The end user applications are described in detail in Appendix A.



```
Test : 1 - Feature : Audio - Status : Success
Test : 2 - Feature : Audio - Status : Success
Test : 3 - Feature : Audio - Status : Success
Test : 5 - Feature : Email - Status : Success
Test : 6 - Feature : Email - Status : Success
Test : 7 - Feature : Email - Status : Success
Test : 8 - Feature : Email - Status : Success
Test : 9 - Feature : Text - Status : Success
Test : 10 - Feature : Text - Status : Success
Test : 11 - Feature : Text - Status : Success
Test : 12 - Feature : Text - Status : Success
Test : 13 - Feature : Door - Status : Success
Test : 14 - Feature : Door - Status : Success
Test : 15 - Feature : Door - Status : Success
Test : 16 - Feature : Motion - Status : Success
Test : 17 - Feature : Motion - Status : Success
Test : 18 - Feature : Motion - Status : Success
Test : 19 - Feature : Window - Status : Success
Test : 20 - Feature : Window - Status : Success
Test : 21 - Feature : Window - Status : Success
Test : 22 - Feature : Faucet Drip - Status : Success
Test : 23 - Feature : Flood Detector - Status : Success
Test : 24 - Feature : Smart Irrigation - Status : Success
Test : 25 - Feature : Smart Irrigation - Status : Success
Test : 26 - Feature : Smart Irrigation - Status : Success
Test : 27 - Feature : Smart Irrigation - Status : Success
Test : 28 - Feature : Schedule - Status : Success
Test : 29 - Feature : Smart Weather Sensing - Status : Success
Test : 30 - Feature : Video - Status : Success
Test : 31 - Feature : Video - Status : Success
Test : 32 - Feature : Video - Status : Success
Test : 33 - Feature : Video - Status : Success
Test : 34 - Feature : Video - Status : Success
Test : 35 - Feature : Video - Status : Success
Test : 36 - Feature : Video - Status : Success
Test : 37 - Feature : Video - Status : Success
Test : 38 - Feature : HVAC Filter - Status : Success
Test : 39 - Feature : Light Failure - Status : Success
Test : 40 - Feature : Power Failure - Status : Success
Test : 41 - Feature : Home Alarm - Status : Success
Test : 42 - Feature : Home Alarm - Status : Success
Test : 43 - Feature : Home Alarm - Status : Success
Test : 44 - Feature : Home Alarm - Status : Success
Test : 45 - Feature : Home Alarm - Status : Success
```

Figure 7.5 Output of the FeatureBasedTestDriver for the Smart Home EU SPL

7.6.1 EU Application Feature-based Consistency Checking

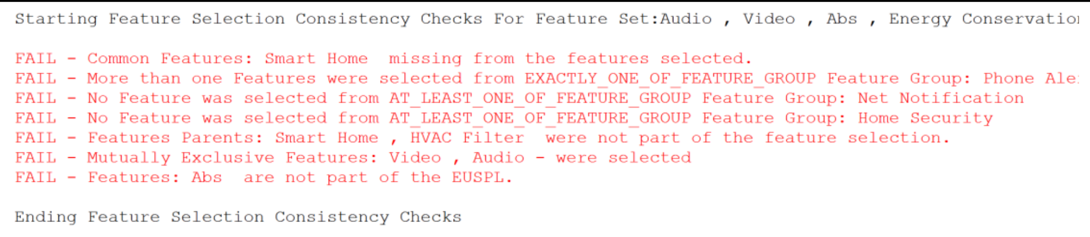
Table 7.5 shows the EU Application Feature-based Consistency test cases for validating the compatibility between features that comprise the derived application. To ensure that the consistency checking process used to execute the consistency test cases functions correctly, valid and invalid feature selection sets were evaluated. The feature sets were derived from the Smart Home EU SPL case study. The valid feature set contained

Table 7.5 EU Application Feature-Based Consistency Tests

Test Case	Expected Result	Test Result
All Common Features were selected	Pass	Pass
Not all Common Features were selected	Fail	Fail
More than one Feature was selected form Exactly-one-of Feature Group	Fail	Fail
Zero Features were selected form Exactly-one-of Feature Group	Fail	Fail
One Feature was selected from Exactly-one-of Feature Group	Pass	Pass
More than one Feature was selected from Zero-or-one-of Feature Group	Fail	Fail
Zero or one Feature was selected from Zero-or-one-of Feature Group	Pass	Pass
Zero or more Features were selected from Zero-or-more-of Feature Group	Pass	Pass
Zero Features were selected form At-least-one-of Feature Group	Fail	Fail
One or more Features were selected from At-least-one Feature Group	Pass	Pass
For each Feature selected the entire parent Feature hierarchy was selected	Pass	Pass
For each Feature selected the parent Feature hierarchy were not selected	Fail	Fail
Mutually Exclusive Alternative features were selected	Fail	Fail

features that are compatible with each other. The invalid feature set contained features that cannot exist together in a derived application.

Figure 7.6 shows the output the output of the ConsistencyRuleChecker executing consistency test cases on the invalid feature set. The invalid feature set contains features: Audio, Video, Abs and Energy conservation from the Smart Home EU SPL. As shown in the output of Figure 7.6 there are several issues with the invalid feature set for instance: the Smart Home common feature is not available, there are required features missing from the Net Notifications and Home Security feature groups, there are mutually exclusive

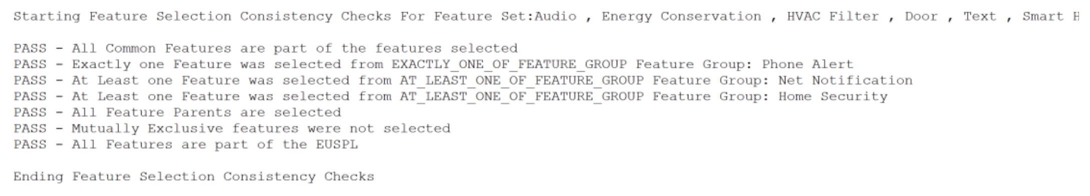


```
Starting Feature Selection Consistency Checks For Feature Set:Audio , Video , Abs , Energy Conservatio

FAIL - Common Features: Smart Home  missing from the features selected.
FAIL - More than one Features were selected from EXACTLY_ONE_OF_FEATURE_GROUP Feature Group: Phone Ale
FAIL - No Feature was selected from AT_LEAST_ONE_OF_FEATURE_GROUP Feature Group: Net Notification
FAIL - No Feature was selected from AT_LEAST_ONE_OF_FEATURE_GROUP Feature Group: Home Security
FAIL - Features Parents: Smart Home , HVAC Filter  were not part of the feature selection.
FAIL - Mutually Exclusive Features: Video , Audio - were selected
FAIL - Features: Abs  are not part of the EUSPL.

Ending Feature Selection Consistency Checks
```

Figure 7.6 ConsistencyRuleChecker Output of executing EU Application Feature-Based Consistency Tests on an invalid Feature Set from the Smart Home EU SPL



```
Starting Feature Selection Consistency Checks For Feature Set:Audio , Energy Conservation , HVAC Filter , Door , Text , Smart H

PASS - All Common Features are part of the features selected
PASS - Exactly one Feature was selected from EXACTLY_ONE_OF_FEATURE_GROUP Feature Group: Phone Alert
PASS - At Least one Feature was selected from AT_LEAST_ONE_OF_FEATURE_GROUP Feature Group: Net Notification
PASS - At Least one Feature was selected from AT_LEAST_ONE_OF_FEATURE_GROUP Feature Group: Home Security
PASS - All Feature Parents are selected
PASS - Mutually Exclusive features were not selected
PASS - All Features are part of the EUSPL

Ending Feature Selection Consistency Checks
```

Figure 7.7 ConsistencyRuleChecker Output of executing EU Application Feature-Based Consistency Tests on a valid Feature Set from the Smart Home EU SPL

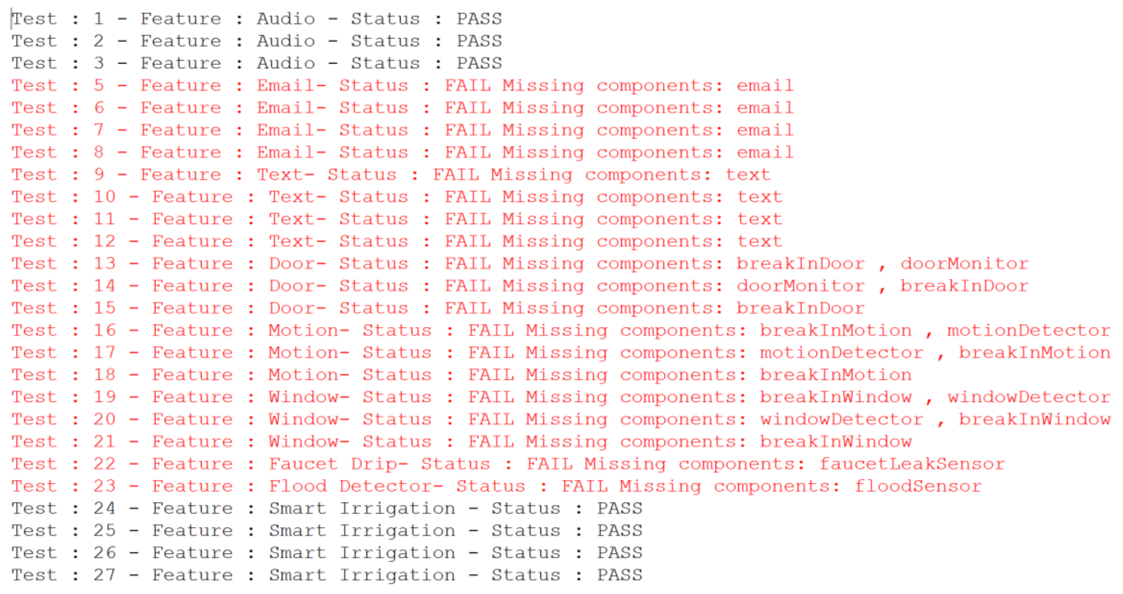
features present in the set, there are features missing that features in the set depend on and there is an Abs feature available in the set that is not available in the Smart Home EU SPL. Similar, Figure 7.7 shows the output of the ConsistencyRuleChecker testing the valid feature selection set. The valid feature set contains features: Audio, Energy Conservation, HVAC Filter, Door, Text and Smart Home from the Smart Home EU SPL. As shown in Figure 7.7 this feature set is valid. It contains all product line common features and required feature dependencies. The consistency checking process evaluated all test cases successfully for both feature selection sets.

7.6.2 EU Application Feature-based Testing

EU Application Feature-based Testing is used to test the component architecture and implementation of the end user derived application. In detail, for each feature that is part of the feature selection the corresponding integration test cases are selected from the EU SPL Repository. The integration test cases test the component architecture and implementation of the derived application. The test cases are executed using the FeatureBasedTestDriver and a TeC interpreter tools.

To execute an integration test, the TeC interpreter reads the derived application specification (TeC PSP) that was created by the EUSPLP environment and instantiates the TeC component implementations of the derived application. Each TeC component in the interpreter is assigned with TeC application instructions based on the derived application. The FeatureBasedTestDriver executes the triggering condition in the source component defined in the test case and evaluates the target component "testResult" variable with the expected result defined in the test case. The execution of the feature-based test cases ensures that the application component architecture derived by the EUSPLP environment is consistent with the component architecture of each feature that comprises the application.

Figure 7.8 shows an example of the FeatureBasedTestDriver output of executing feature-based tests to a derived application that contained the "Audio" and "Smart Irrigation" features. The tests executed by the FeatureBasedTestDriver included: (a) The test cases defined for the "Audio" and "Smart Irrigation" features, and (b) The test cases defined for the "Email", "Text", "Door", "Motion", "Window", "Faucet Drip" and "Flood Detector" features that did not apply to the derived application. As expected the test cases of the "Audio" and "Smart Irrigation" features executed correctly. The test cases of the additional feature test cases failed as expected since components and connectors of these



```
Test : 1 - Feature : Audio - Status : PASS
Test : 2 - Feature : Audio - Status : PASS
Test : 3 - Feature : Audio - Status : PASS
Test : 5 - Feature : Email- Status : FAIL Missing components: email
Test : 6 - Feature : Email- Status : FAIL Missing components: email
Test : 7 - Feature : Email- Status : FAIL Missing components: email
Test : 8 - Feature : Email- Status : FAIL Missing components: email
Test : 9 - Feature : Text- Status : FAIL Missing components: text
Test : 10 - Feature : Text- Status : FAIL Missing components: text
Test : 11 - Feature : Text- Status : FAIL Missing components: text
Test : 12 - Feature : Text- Status : FAIL Missing components: text
Test : 13 - Feature : Door- Status : FAIL Missing components: breakInDoor , doorMonitor
Test : 14 - Feature : Door- Status : FAIL Missing components: doorMonitor , breakInDoor
Test : 15 - Feature : Door- Status : FAIL Missing components: breakInDoor
Test : 16 - Feature : Motion- Status : FAIL Missing components: breakInMotion , motionDetector
Test : 17 - Feature : Motion- Status : FAIL Missing components: motionDetector , breakInMotion
Test : 18 - Feature : Motion- Status : FAIL Missing components: breakInMotion
Test : 19 - Feature : Window- Status : FAIL Missing components: breakInWindow , windowDetector
Test : 20 - Feature : Window- Status : FAIL Missing components: windowDetector , breakInWindow
Test : 21 - Feature : Window- Status : FAIL Missing components: breakInWindow
Test : 22 - Feature : Faucet Drip- Status : FAIL Missing components: faucetLeakSensor
Test : 23 - Feature : Flood Detector- Status : FAIL Missing components: floodSensor
Test : 24 - Feature : Smart Irrigation - Status : PASS
Test : 25 - Feature : Smart Irrigation - Status : PASS
Test : 26 - Feature : Smart Irrigation - Status : PASS
Test : 27 - Feature : Smart Irrigation - Status : PASS
```

Figure 7.8 FeatureBasedTestDriver Output executing Feature-Based Integration Test Cases to a Derived Application that contains the Audio and Smart Irrigation Features

features were not available in the derived application. Feature-based testing using the FeatureBasedTestDriver and the TeC interpreter helped to identify issues with (a) the component implementation (b) thread issues between components and (c) PSP specifications that were invalid (missing Activity_Sheets, component connectors, inputs, outputs, output parameters, invalid JSON etc.).

7.6.3 EU Application Testing for Smart Home End User Application 1

This section describes the EU Application Testing process applied to the "Smart Home Example 1" application described in Appendix A. The application was derived from the Smart Home EU SPL case study using the application derivation process of the EUSPLP development environment. Figure 7.9 shows the Feature Model of the derived

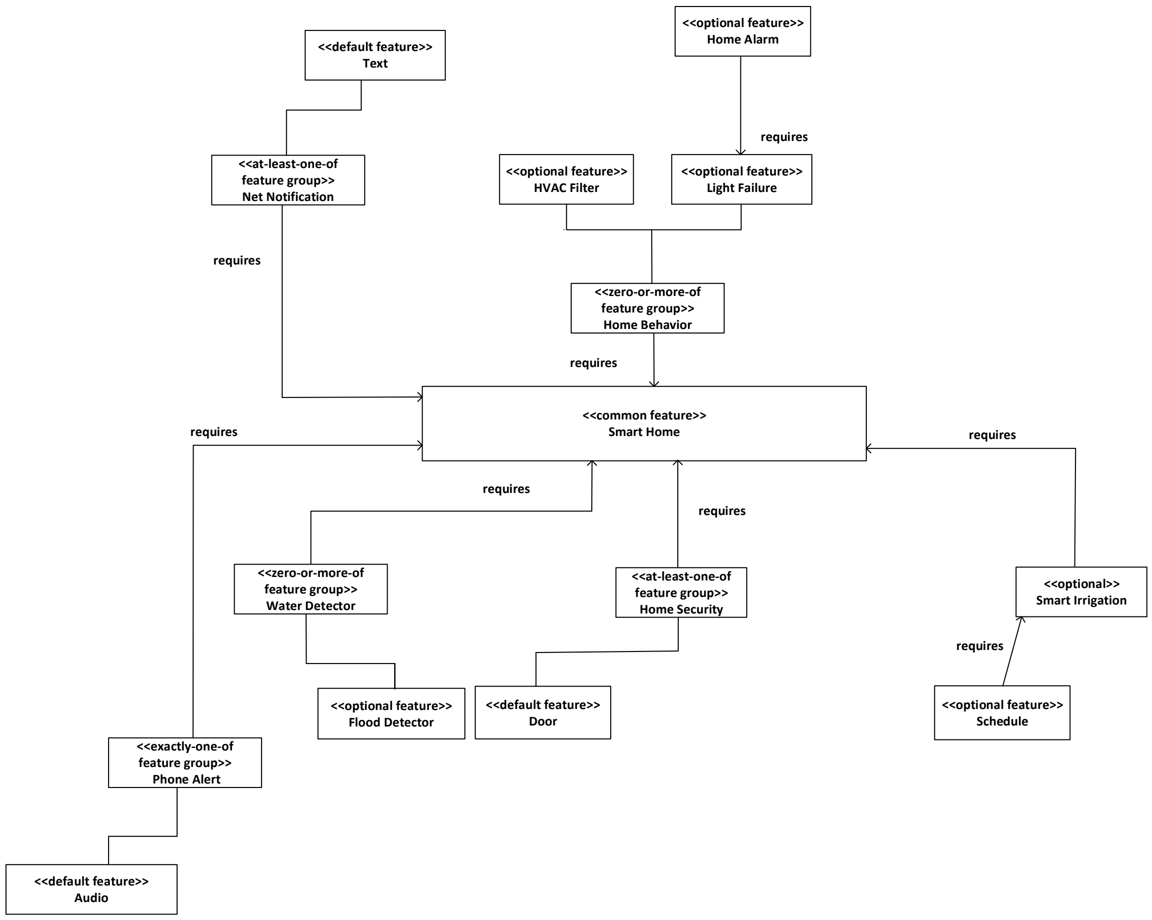
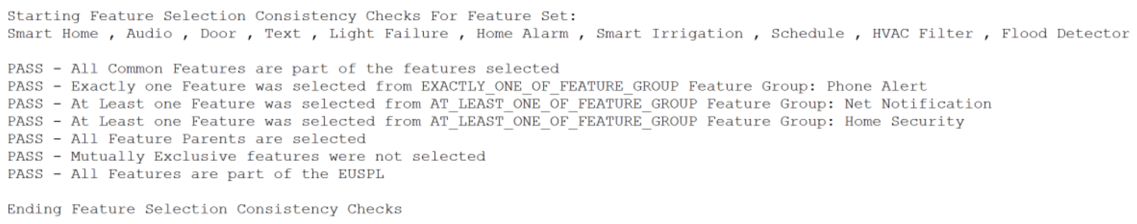


Figure 7.9 Smart Home Example 1 Application – Feature Model



```
Starting Feature Selection Consistency Checks For Feature Set:
Smart Home , Audio , Door , Text , Light Failure , Home Alarm , Smart Irrigation , Schedule , HVAC Filter , Flood Detector

PASS - All Common Features are part of the features selected
PASS - Exactly one Feature was selected from EXACTLY_ONE_OF_FEATURE_GROUP Feature Group: Phone Alert
PASS - At Least one Feature was selected from AT_LEAST_ONE_OF_FEATURE_GROUP Feature Group: Net Notification
PASS - At Least one Feature was selected from AT_LEAST_ONE_OF_FEATURE_GROUP Feature Group: Home Security
PASS - All Feature Parents are selected
PASS - Mutually Exclusive features were not selected
PASS - All Features are part of the EUSPL

Ending Feature Selection Consistency Checks
```

Figure 7.10 ConsistencyRuleChecker Output of executing EU Application Consistency Tests to the Features selected for the Smart Home Example 1 Application

application. The derived application consists of the following features: “Smart Home”, “Audio”, “Door”, “Text”, “Flood Detector”, “Smart Irrigation”, “Schedule”, “HVAC

Filter", "Home Alarm" and "Light Failure." Figure 7.10 shows the output of executing EU Application Feature-based Consistency Test Cases to the features that comprise the "Smart Home Example 1" application. The selected features for the "Smart Home Example 1" passed all the EU Application Feature-based Consistency tests.

Figure 7.11 shows the application architecture of the "Smart Home Example 1" application. Figure 7.12 shows the Feature-based Integration Test Cases derived for the "Smart Home Example 1" application to support EU Application Testing. To perform EU Application Testing of the "Smart Home Example 1" application, three types of Feature-based Integration Test Cases were executed: (1) component interface test cases defined for every connector in the derived application, (2) multi-component interaction sequence test cases of dependent features, and (3) multi-component interaction sequence test cases for features that don't explicitly depend on each other in the feature model but an event in one feature affects the other. Below are examples of each test case type. Test case 2, defined for the "Audio" feature shown in Figure 7.12, is an example of component interface testing. This test case tests the connector of the "sendAlert" output of the "securityAlertHandler" component to the "notify" input of the "alertAudio" component. There are two components tested, the "securityAlertHandler" component of the "Smart Home" feature and the "alertAudio" component of the "Audio" feature. The scenario that this test case evaluates is that when a security alert is available in the "securityAlertHandler" component queue, a message is send to the "notify" input of the "alertAudio" component to notify the house residents.

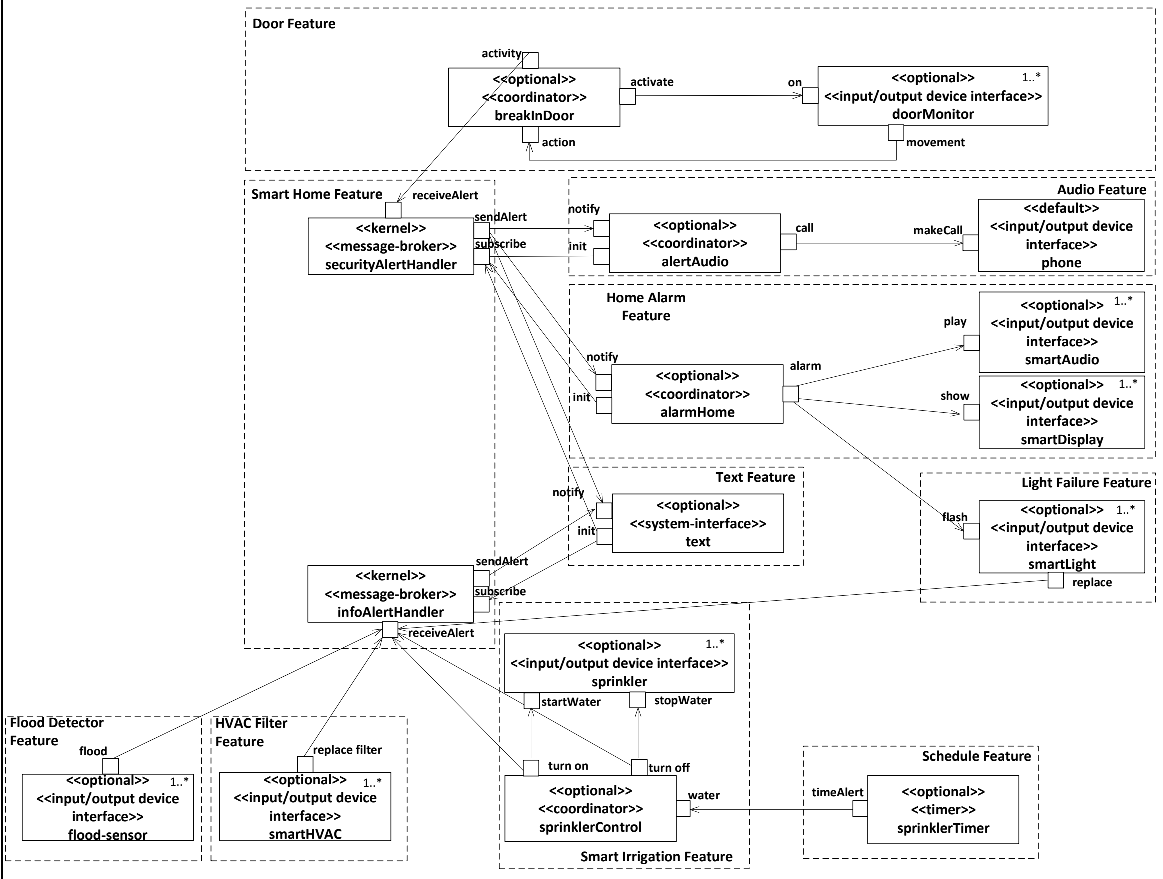


Figure 7.11 Smart Home Example 1 - Application Architecture for TeC

The test case 16 shown in Figure 7.12 is an example of multi-component interaction sequence test case. The test case source component is the “doorMonitor” of the “Door” feature and the target component is the “securityAlertHandler” of the “Smart Home” feature. The scenario evaluated is that when a door break-in is detected a message will be send to the “securityAlertHandler.” For the test to be successful the “breakInDoor” component of the “Door” feature shown Figure 7.11 needs to send a message to the “receiveAlert” input of the “securityAlertHandler” with the parameters shown in the test case.

TestCase	Feature Name	Feature Type	Source Component	Source Output	Source Output Params	Source Trigger	Target Component	Target Input	Test Case Result
1	Audio	default alternative	alertAudio	call	dial=703545558,msg=alert	message=true	call	makeCall	input=makeCall, dial=703545558,msg=alert
2	Audio	default alternative	securityAlertHandler	sendAlert	message=securityAlert	messageInQueue=true	alertAudio	notify	input=notify, message=securityAlert
3	Audio	default alternative	alertAudio	init	component_name=alertAu	startup=true	securityAlertHandler	subscribe	input=subscribe, component_name=alertAudio,topic=security
4	Audio	default alternative	securityAlertHandler	sendAlert	message=securityAlert, dia	messageInQueue=true	call	makeCall	input=makeCall, dial=703545558,msg=alert
9	Text	default optional	text	init	component_name=text,to	startup=true	securityAlertHandler	subscribe	input=subscribe, component_name=text,topic=security
10	Text	default optional	text	init	component_name=text,to	startup=true	infoAlertHandler	subscribe	input=subscribe, component_name=text,topic=info
11	Text	default optional	securityAlertHandler	sendAlert	msg=securityAlert,phone_	messageInQueue=true	text	notify	input=notify, msg=securityAlert,phone_number=703545558
12	Text	default optional	infoAlertHandler	sendAlert	msg=infoAlert,phone_num	messageInQueue=true	text	notify	input=notify, msg=infoAlert,phone_number=703545558
13	Door	default optional	breakInDoor	activate	turn on=true	startup=true	doorMonitor	on	input=on, turn on=true
14	Door	default optional	doorMonitor	movement	device_id=door1,device_ty	move=true	breakInDoor	action	input=action, device_id=door1,device_type=sensor
15	Door	default optional	breakInDoor	activity	component_name=breakIr	motion=true	securityAlertHandler	receiveAlert	input=receiveAlert, component_name=breakInDoor,topic=security
16	Door	default optional	doorMonitor	movement	device_id=door1,device_ty	move=true	securityAlertHandler	receiveAlert	input=receiveAlert, component_name=breakInDoor,topic=security
26	Flood Detector	optional	flood-sensor	flood	component_name=flood-s	moisture=true	infoAlertHandler	receiveAlert	input=receiveAlert, component_name=flood-sensor,topic=info
27	Smart Irrigation	optional	sprinklerControl	turn on	component_name=sprinkl	on=true	infoAlertHandler	receiveAlert	input=receiveAlert, component_name=sprinklerControl,topic=info
28	Smart Irrigation	optional	sprinklerControl	turn off	component_name=sprinkl	off=true	infoAlertHandler	receiveAlert	input=receiveAlert, component_name=sprinklerControl,topic=info
29	Smart Irrigation	optional	sprinklerControl	turn on	start=true	on=true	sprinkler	startWater	input=startWater, start=true
30	Smart Irrigation	optional	sprinklerControl	turn off	stop=true	off=true	sprinkler	stopWater	input=stopWater, stop=true
31	Schedule	optional	schedule	timeAlert	on=true	alarm=true	sprinklerControl	water	input=water, on=true
42	HVAC Filter	optional	smartHVAC	replace	component_name=smartH	replaceFilter=true	infoAlertHandler	receiveAlert	input=receiveAlert, component_name=smartHVAC,topic=info
43	Light Failure	optional	smartLight	replace	component_name=smartL	light=out	infoAlertHandler	receiveAlert	input=receiveAlert, component_name=smartLight,topic=info
45	Home Alarm	optional	alarmHome	init	component_name=alertAu	startup=true	securityAlertHandler	subscribe	input=subscribe, component_name=alertAudio,topic=security
46	Home Alarm	optional	securityAlertHandler	sendAlert	message=securityAlert	messageInQueue=true	alarmHome	notify	input=notify, message=securityAlert
47	Home Alarm	optional	alarmHome	alarm	flash=true	message=true	smartLight	flash	input=flash,flash=true
48	Home Alarm	optional	alarmHome	alarm	msg=alert	message=true	smartDisplay	play	input=play, msg=alert
49	Home Alarm	optional	alarmHome	alarm	msg=alert	message=true	smartAudio	play	input=play, msg=alert
50	Home Alarm	optional	securityAlertHandler	sendAlert	message=securityAlert, ms	messageInQueue=true	smartAudio	play	input=play, msg=alert
51	Home Alarm	optional	securityAlertHandler	sendAlert	message=securityAlert, fla	messageInQueue=true	smartLight	flash	input=flash,flash=true
52	Home Alarm	optional	securityAlertHandler	sendAlert	message=securityAlert, ms	messageInQueue=true	smartDisplay	play	input=play, msg=alert
53	Schedule / Smart Irrigation	n/a	schedule	timeAlert	on=true,start=true	alarm=true	sprinkler	startWater	input=startWater, start=true
54	Door / Audio	n/a	doorMonitor	movement	device_id=door1,device_ty	move=true	call	makeCall	input=makeCall, dial=703545558,msg=alert
55	Door / Text	n/a	doorMonitor	movement	device_id=door1,device_ty	move=true	text	notify	input=notify, msg=securityAlert,phone_number=703545558
56	Flood Detector / Text	n/a	flood-sensor	flood	component_name=flood-s	moisture=true	text	notify	input=notify, msg=infoAlert,phone_number=703545558
57	Schedule / Smart Irrigation /	n/a	schedule	timeAlert	on=true,start=true,msg=in	alarm=true	text	notify	input=notify, msg=infoAlert,phone_number=703545558
58	HVAC Filter / Text	n/a	smartHVAC	replace	component_name=smartH	replaceFilter=true	text	notify	input=notify, msg=infoAlert,phone_number=703545558
59	Light Failure / Text	n/a	smartLight	replace	component_name=smartL	light=out	text	notify	input=notify, msg=infoAlert,phone_number=703545558
60	Door / Home Alarm	n/a	doorMonitor	movement	device_id=door1,device_ty	move=true	smartDisplay	play	input=play, msg=alert

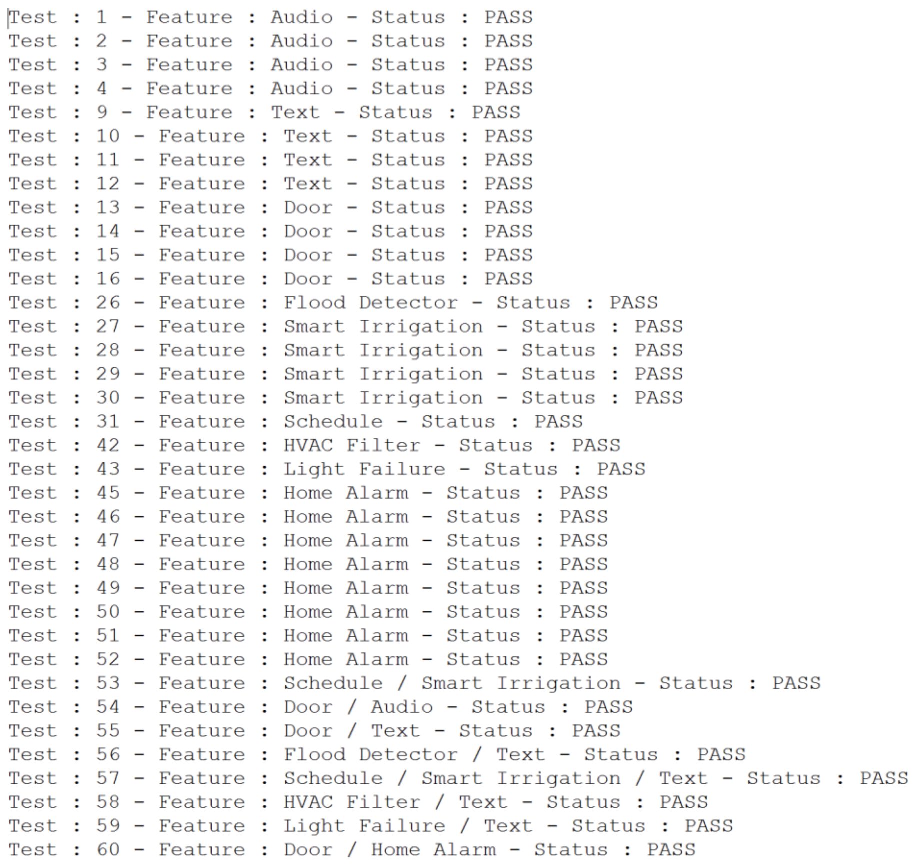
Figure 7.12 Featured-Based Integration Test Cases for the Smart Home Example 1 EU Application

This test case tests the interaction of the “doorMonitor”, “breakInDoor” and “securityAlertHandler” components.

Test case 54 defined for the “Door” and “Audio” features shown in Figure 7.12 is an example of multi-component interaction sequence test case across features that are not dependent in the feature model but an event on one affects the other. The scenario that this test case evaluates is that when there is a door break-in, a security message notification is send to the resident’s phone. Although the components of the “Door” are not communicating with the “Audio” feature directly, they communicate through the “Smart Home” feature. For instance, when a door break-in is detected, the “securityAlertHandler” receives a security alert. The “securityAlertHandler” sends the security alert to the “alertAudio” component that is subscribed to receive messages. As shown in Figure 7.12 when the “alertAudio” component receives the security alert message, it will evaluate the corresponding triggering condition and send an alert message to the “makeCall” input of the “phone” component to contact the house resident. This test case tests the interaction sequence of the “doorMonitor”, “breakInDoor”, “securityAlertHandler”, “alertAudio” and “phone” components.

All test cases have the same format shown in Table 7.4. Triggering conditions were used to simulate external events. The “Source Trigger” column in Figure 7.12 shows all the triggers executed in the derived application. The test case execution starts with a triggering condition that evaluates to true. Triggering conditions are evaluated to true when an external event occurs. For example, when there is a break-in detected, the triggering condition “move=true” of the “doorMonitor” component evaluates to true, which causes the “movement” output to get executed. To ensure that the test cases executed correctly, the “testResult” attribute of the test case target component was compared with the expected results of the test case shown in the “Test Case Result” column. The “testResult” attribute contains: (a) the component input that was called and (b) the parameters that were passed to the target component. Separate test cases were created to test a triggering condition that causes inputs on different components to get triggered. For example, as shown in Figure 7.11 the “alertHome” component sends three independent messages to the “smartLight”, “smartDisplay” and “smartAudio” components when it receives a message from the “securityAlertHandler” component. To test this scenario, three test cases were created test cases 50, 51and 52 shown in Figure 7.12. System traces were also used to verify that all three events executed when the “alertHome” component received a message from the “securityAlertHandler” component.

EU Application Testing validated (a) that all application components were derived from the features selected and (b) the connectivity between components worked as designed in the EU SPL. Figure 7.13 shows the output of executing the Featured-Based Integration Test Cases against the derived application. The output shows that all tests executed successfully, which indicates that the expected test result in the test case is consisted with the “testResult” attribute values found in the target component. In addition to the test cases that relate to the features selected, all Feature-based Integration Test Cases defined for the Smart Home EU SPL were executed to verify that no additional components or component connectors were introduced. All test cases defined for features that were not part of the “Smart Home Example1” application failed as expected.



```
Test : 1 - Feature : Audio - Status : PASS
Test : 2 - Feature : Audio - Status : PASS
Test : 3 - Feature : Audio - Status : PASS
Test : 4 - Feature : Audio - Status : PASS
Test : 9 - Feature : Text - Status : PASS
Test : 10 - Feature : Text - Status : PASS
Test : 11 - Feature : Text - Status : PASS
Test : 12 - Feature : Text - Status : PASS
Test : 13 - Feature : Door - Status : PASS
Test : 14 - Feature : Door - Status : PASS
Test : 15 - Feature : Door - Status : PASS
Test : 16 - Feature : Door - Status : PASS
Test : 26 - Feature : Flood Detector - Status : PASS
Test : 27 - Feature : Smart Irrigation - Status : PASS
Test : 28 - Feature : Smart Irrigation - Status : PASS
Test : 29 - Feature : Smart Irrigation - Status : PASS
Test : 30 - Feature : Smart Irrigation - Status : PASS
Test : 31 - Feature : Schedule - Status : PASS
Test : 42 - Feature : HVAC Filter - Status : PASS
Test : 43 - Feature : Light Failure - Status : PASS
Test : 45 - Feature : Home Alarm - Status : PASS
Test : 46 - Feature : Home Alarm - Status : PASS
Test : 47 - Feature : Home Alarm - Status : PASS
Test : 48 - Feature : Home Alarm - Status : PASS
Test : 49 - Feature : Home Alarm - Status : PASS
Test : 50 - Feature : Home Alarm - Status : PASS
Test : 51 - Feature : Home Alarm - Status : PASS
Test : 52 - Feature : Home Alarm - Status : PASS
Test : 53 - Feature : Schedule / Smart Irrigation - Status : PASS
Test : 54 - Feature : Door / Audio - Status : PASS
Test : 55 - Feature : Door / Text - Status : PASS
Test : 56 - Feature : Flood Detector / Text - Status : PASS
Test : 57 - Feature : Schedule / Smart Irrigation / Text - Status : PASS
Test : 58 - Feature : HVAC Filter / Text - Status : PASS
Test : 59 - Feature : Light Failure / Text - Status : PASS
Test : 60 - Feature : Door / Home Alarm - Status : PASS
```

Figure 7.13 FeatureBasedTestDriver Output of executing the Featured-Based Integration Test Cases to the Smart Home Example 1 EU Application

7.6.4 EU Application Testing for Smart Home End User Application 2

This section presents the EU Application Testing process applied to the "Smart Home Example 2" application described in Appendix A. The application was derived from the Smart Home EU SPL case study using the application derivation process of the EUSPL development environment. Figure 7.14 shows the Feature Model for the derived application. The derived application consists of the following features: "Smart Home", "Video", "Door", "Motion", "Window", "Email", "HVAC Filter", "Energy Conservation"

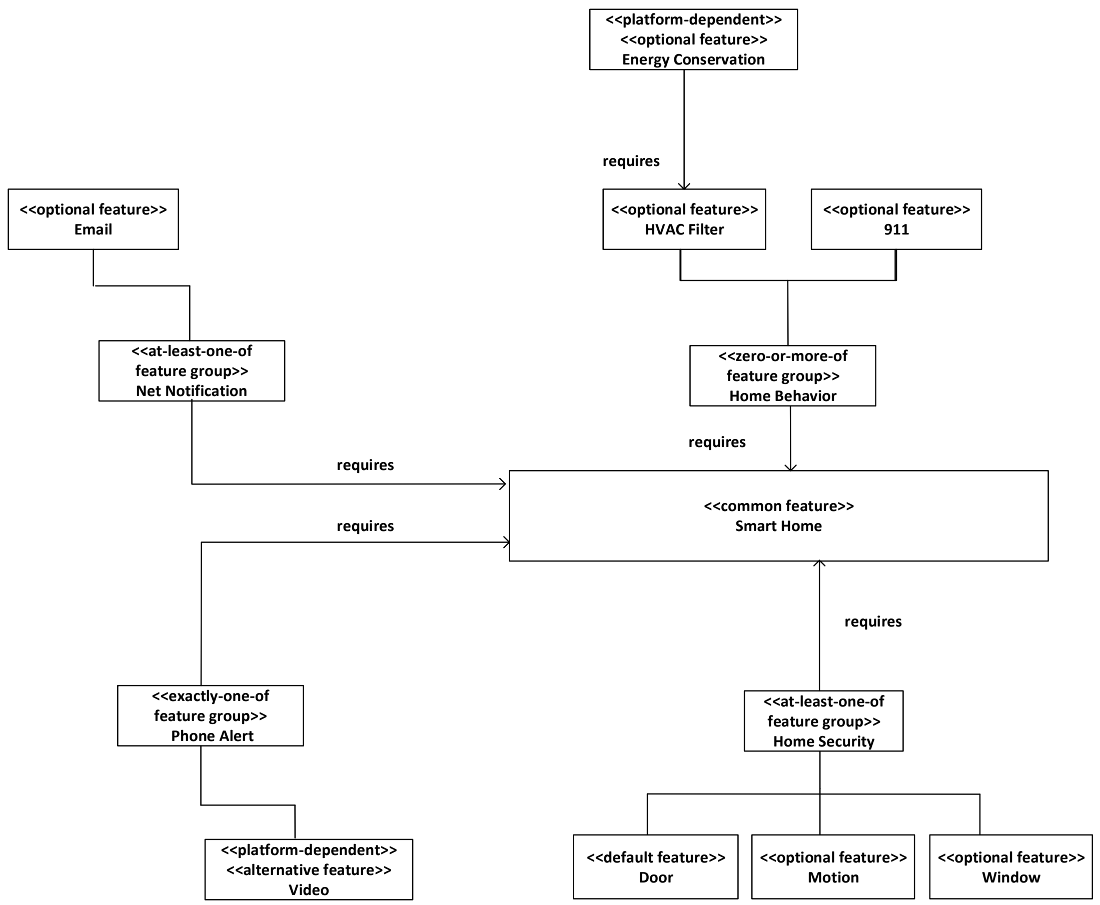
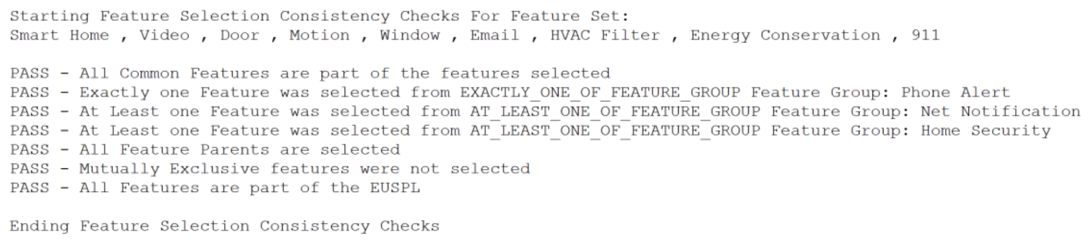


Figure 7.14 Smart Home Example 2 Application – Feature Model



```
Starting Feature Selection Consistency Checks For Feature Set:
Smart Home , Video , Door , Motion , Window , Email , HVAC Filter , Energy Conservation , 911

PASS - All Common Features are part of the features selected
PASS - Exactly one Feature was selected from EXACTLY_ONE_OF_FEATURE_GROUP Feature Group: Phone Alert
PASS - At Least one Feature was selected from AT_LEAST_ONE_OF_FEATURE_GROUP Feature Group: Net Notification
PASS - At Least one Feature was selected from AT_LEAST_ONE_OF_FEATURE_GROUP Feature Group: Home Security
PASS - All Feature Parents are selected
PASS - Mutually Exclusive features were not selected
PASS - All Features are part of the EUSPL

Ending Feature Selection Consistency Checks
```

Figure 7.15 ConsistencyRuleChecker Output of executing EU Application Consistency Tests to the Features selected for the Smart Home Example 2 Application

and "911." Figure 7.15 shows the output of executing EU Application Feature-based Consistency Tests to the features that comprise the "Smart Home Example 2" application to test if the features are compatible. As shown in Figure 7.15 all EU Application Feature-based Consistency Tests executed successfully.

Figure 7.16 shows the application architecture of the “Smart Home Example 2.” Figure 7.17 shows the Feature-based Integration Test Cases derived for the “Smart Home Example 2” application to support EU Application Testing. To perform EU Application Testing to the “Smart Home Example 2” three types of Feature-based Integration Test Cases were executed: (1) component interface test cases defined for every connector in the derived application, (2) multi-component interaction sequence test cases of depend features, and (3) multi-component interaction sequence test cases of independent features that an event on one feature affects the other. Below are examples of each test case type. Test case 8 defined for the “Email” feature shown in is an example of component interface testing. This test case tests the connector of the “sendAlert” output of the “infoAlertHandler” component to the “notify” input of the “email” component. The scenario that this test case evaluates is that when an informational alert is available in the “infoAlertHandler” component queue, a message is sent to the “email” component to notify the house residents.

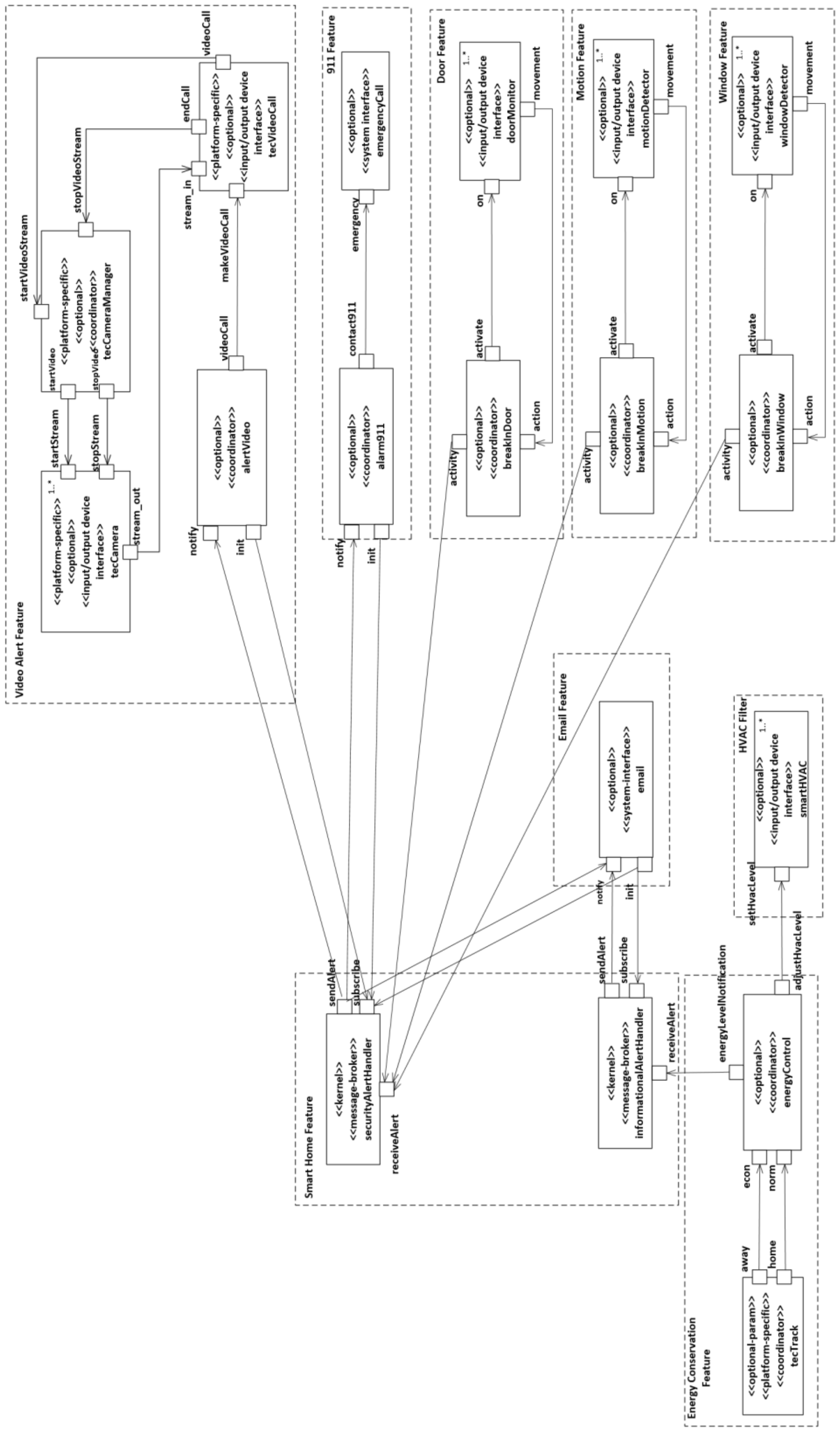
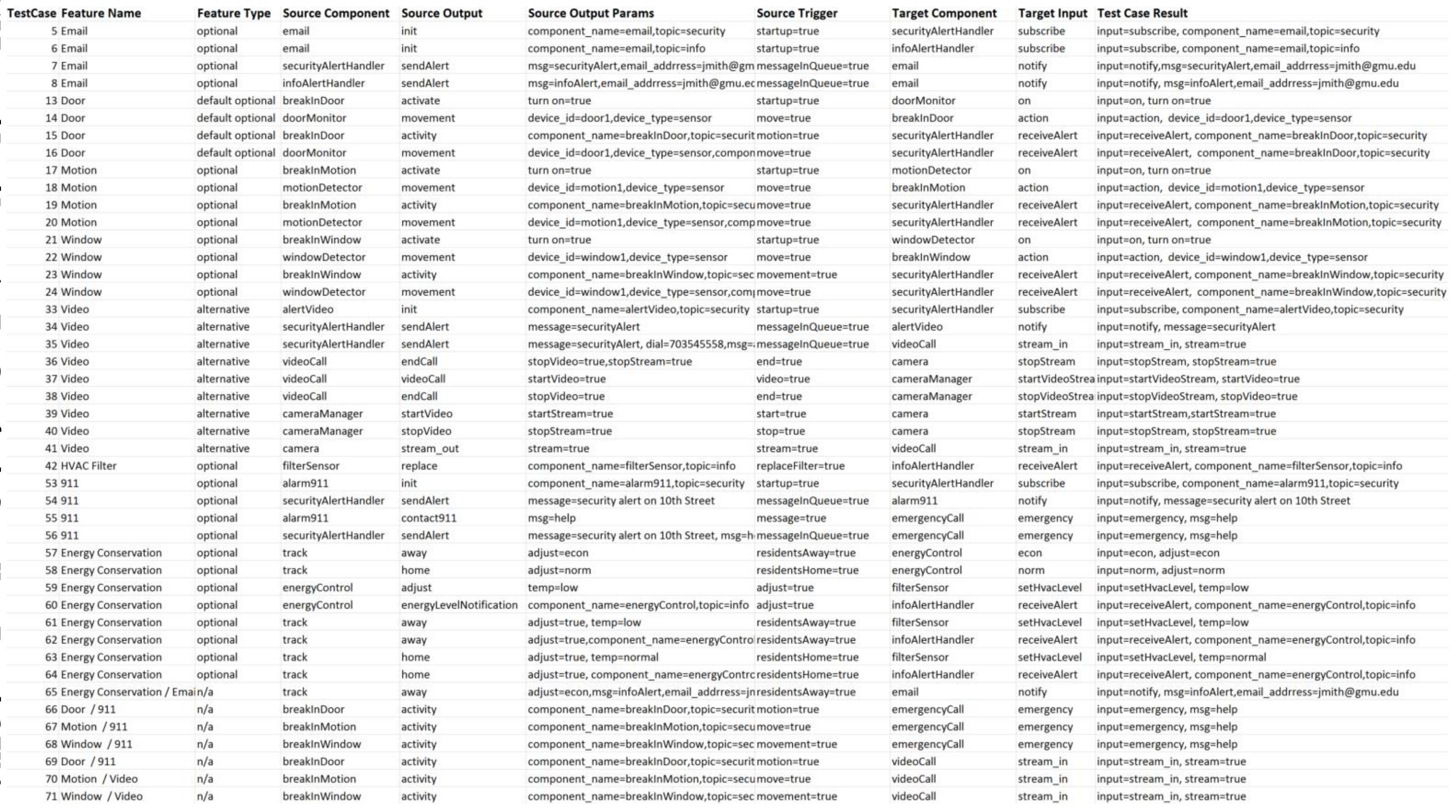

Video Alert Feature
<<platform-specific>> <<optional>> <<input/output device interface>> tecCamera
<<platform-specific>> <<optional>> <<coordinator>> tecCameraManager
<<platform-specific>> <<optional>> <<input/output device interface>> tecVideoCall
<<optional>> <<coordinator>> alertVideo
startVideoStream
stopVideoStream
startVideo
stopVideo
startStream
stopStream
stream_out
stream_in
endCall
videoCall
makeVideoCall
notify
init
911 Feature
<<optional>> <<system interface>> emergencyCall
<<optional>> <<coordinator>> alarm911
emergency
contact911
Door Feature
<<optional>> <<input/output device interface>> doorMonitor
<<optional>> <<coordinator>> breakInDoor
Motion Feature
<<optional>> <<input/output device interface>> motionDetector
<<optional>> <<coordinator>> breakInMotion
Window Feature
<<optional>> <<input/output device interface>> windowDetector
<<optional>> <<coordinator>> breakInWindow
on
movement
activate
action
activity
1..*
Email Feature
<<optional>> <<system-interface>> email
HVAC Filter
<<optional>> <<input/output device interface>> smartHVAC
setHvacLevel
Smart Home Feature
<<kernel>> <<message-broker>> securityAlertHandler
<<kernel>> <<message-broker>> informationalAlertHandler
sendAlert
subscribe
receiveAlert
Energy Conservation Feature
<<optional>> <<coordinator>> energyControl
<<optional-param>> <<platform-specific>> <<coordinator>> tecTrack
energyLevelNotification
adjustHvacLevel
econ
norm
away
home

Figure 7.16 Smart Home Example 2 - Application Architecture for TeC



TestCase	Feature Name	Feature Type	Source Component	Source Output	Source Output Params	Source Trigger	Target Component	Target Input	Test Case Result
5	Email	optional	email	init	component_name=email,topic=security	startup=true	securityAlertHandler	subscribe	input=subscribe, component_name=email,topic=security
6	Email	optional	email	init	component_name=email,topic=info	startup=true	infoAlertHandler	subscribe	input=subscribe, component_name=email,topic=info
7	Email	optional	securityAlertHandler	sendAlert	msg=securityAlert,email_addrress=jmith@gm	messageInQueue=true	email	notify	input=notify,msg=securityAlert,email_addrress=jmith@gmu.edu
8	Email	optional	infoAlertHandler	sendAlert	msg=infoAlert,email_addrress=jmith@gmu.ec	messageInQueue=true	email	notify	input=notify, msg=infoAlert,email_addrress=jmith@gmu.edu
13	Door	default optional	breakInDoor	activate	turn on=true	startup=true	doorMonitor	on	input=on, turn on=true
14	Door	default optional	doorMonitor	movement	device_id=door1,device_type=sensor	move=true	breakInDoor	action	input=action, device_id=door1,device_type=sensor
15	Door	default optional	breakInDoor	activity	component_name=breakInDoor,topic=securit	motion=true	securityAlertHandler	receiveAlert	input=receiveAlert, component_name=breakInDoor,topic=security
16	Door	default optional	doorMonitor	movement	device_id=door1,device_type=sensor,compon	move=true	securityAlertHandler	receiveAlert	input=receiveAlert, component_name=breakInDoor,topic=security
17	Motion	optional	breakInMotion	activate	turn on=true	startup=true	motionDetector	on	input=on, turn on=true
18	Motion	optional	motionDetector	movement	device_id=motion1,device_type=sensor	move=true	breakInMotion	action	input=action, device_id=motion1,device_type=sensor
19	Motion	optional	breakInMotion	activity	component_name=breakInMotion,topic=secu	move=true	securityAlertHandler	receiveAlert	input=receiveAlert, component_name=breakInMotion,topic=security
20	Motion	optional	motionDetector	movement	device_id=motion1,device_type=sensor,comp	move=true	securityAlertHandler	receiveAlert	input=receiveAlert, component_name=breakInMotion,topic=security
21	Window	optional	breakInWindow	activate	turn on=true	startup=true	windowDetector	on	input=on, turn on=true
22	Window	optional	windowDetector	movement	device_id=window1,device_type=sensor	move=true	breakInWindow	action	input=action, device_id=window1,device_type=sensor
23	Window	optional	breakInWindow	activity	component_name=breakInWindow,topic=sec	movement=true	securityAlertHandler	receiveAlert	input=receiveAlert, component_name=breakInWindow,topic=security
24	Window	optional	windowDetector	movement	device_id=window1,device_type=sensor,com	move=true	securityAlertHandler	receiveAlert	input=receiveAlert, component_name=breakInWindow,topic=security
33	Video	alternative	alertVideo	init	component_name=alertVideo,topic=security	startup=true	securityAlertHandler	subscribe	input=subscribe, component_name=alertVideo,topic=security
34	Video	alternative	securityAlertHandler	sendAlert	message=securityAlert	messageInQueue=true	alertVideo	notify	input=notify, message=securityAlert
35	Video	alternative	securityAlertHandler	sendAlert	message=securityAlert, dial=703545558,msg=	messageInQueue=true	videoCall	stream_in	input=stream_in, stream=true
36	Video	alternative	videoCall	endCall	stopVideo=true,stopStream=true	end=true	camera	stopStream	input=stopStream, stopStream=true
37	Video	alternative	videoCall	videoCall	startVideo=true	video=true	cameraManager	startVideoStrea	input=startVideoStream, startVideo=true
38	Video	alternative	videoCall	endCall	stopVideo=true	end=true	cameraManager	stopVideoStrea	input=stopVideoStream, stopVideo=true
39	Video	alternative	cameraManager	startVideo	startStream=true	start=true	camera	startStream	input=startStream,startStream=true
40	Video	alternative	cameraManager	stopVideo	stopStream=true	stop=true	camera	stopStream	input=stopStream, stopStream=true
41	Video	alternative	camera	stream_out	stream=true	stream=true	videoCall	stream_in	input=stream_in, stream=true
42	HVAC Filter	optional	filterSensor	replace	component_name=filterSensor,topic=info	replaceFilter=true	infoAlertHandler	receiveAlert	input=receiveAlert, component_name=filterSensor,topic=info
53	911	optional	alarm911	init	component_name=alarm911,topic=security	startup=true	securityAlertHandler	subscribe	input=subscribe, component_name=alarm911,topic=security
54	911	optional	securityAlertHandler	sendAlert	message=security alert on 10th Street	messageInQueue=true	alarm911	notify	input=notify, message=security alert on 10th Street
55	911	optional	alarm911	contact911	msg=help	message=true	emergencyCall	emergency	input=emergency, msg=help
56	911	optional	securityAlertHandler	sendAlert	message=security alert on 10th Street, msg=h	messageInQueue=true	emergencyCall	emergency	input=emergency, msg=help
57	Energy Conservation	optional	track	away	adjust=econ	residentsAway=true	energyControl	econ	input=econ, adjust=econ
58	Energy Conservation	optional	track	home	adjust=norm	residentsHome=true	energyControl	norm	input=norm, adjust=norm
59	Energy Conservation	optional	energyControl	adjust	temp=low	adjust=true	filterSensor	setHvacLevel	input=setHvacLevel, temp=low
60	Energy Conservation	optional	energyControl	energyLevelNotification	component_name=energyControl,topic=info	adjust=true	infoAlertHandler	receiveAlert	input=receiveAlert, component_name=energyControl,topic=info
61	Energy Conservation	optional	track	away	adjust=true, temp=low	residentsAway=true	filterSensor	setHvacLevel	input=setHvacLevel, temp=low
62	Energy Conservation	optional	track	away	adjust=true,component_name=energyContro	residentsAway=true	infoAlertHandler	receiveAlert	input=receiveAlert, component_name=energyControl,topic=info
63	Energy Conservation	optional	track	home	adjust=true, temp=normal	residentsHome=true	filterSensor	setHvacLevel	input=setHvacLevel, temp=normal
64	Energy Conservation	optional	track	home	adjust=true, component_name=energyContrc	residentsHome=true	infoAlertHandler	receiveAlert	input=receiveAlert, component_name=energyControl,topic=info
65	Energy Conservation / Emai	n/a	track	away	adjust=econ,msg=infoAlert,email_addrress=jn	residentsAway=true	email	notify	input=notify, msg=infoAlert,email_addrress=jmith@gmu.edu
66	Door / 911	n/a	breakInDoor	activity	component_name=breakInDoor,topic=securit	motion=true	emergencyCall	emergency	input=emergency, msg=help
67	Motion / 911	n/a	breakInMotion	activity	component_name=breakInMotion,topic=secu	move=true	emergencyCall	emergency	input=emergency, msg=help
68	Window / 911	n/a	breakInWindow	activity	component_name=breakInWindow,topic=sec	movement=true	emergencyCall	emergency	input=emergency, msg=help
69	Door / 911	n/a	breakInDoor	activity	component_name=breakInDoor,topic=securit	motion=true	videoCall	stream_in	input=stream_in, stream=true
70	Motion / Video	n/a	breakInMotion	activity	component_name=breakInMotion,topic=secu	move=true	videoCall	stream_in	input=stream_in, stream=true
71	Window / Video	n/a	breakInWindow	activity	component_name=breakInWindow,topic=sec	movement=true	videoCall	stream_in	input=stream_in, stream=true

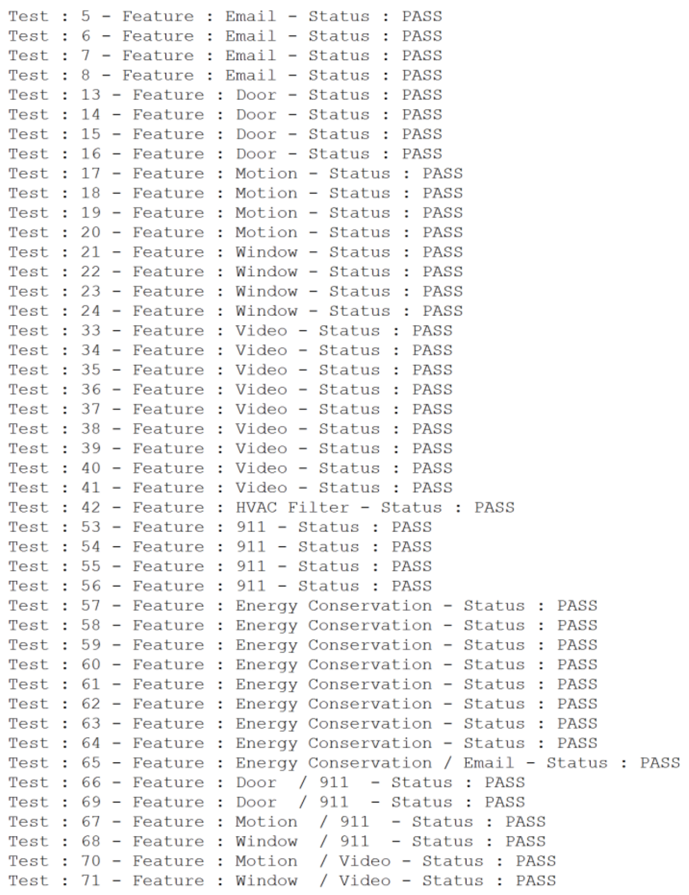
Figure 7.17 Featured-Based Integration Test Cases for the Smart Home Example 2 EU Application

Test case 35, defined for the “Video” feature shown in Figure 7.17, is an example of multi-component interaction sequence test case. The test case source component is the “securityAlertHandler” component of the “Smart Home” feature and the target component is the “videoCall” component of the “Video” feature. The scenario tested is that when a security alert is detected, a video call is placed and the resident gets a live video feed of the events in the house. This test case tests the connectors between components: “securityAlertHandler”, “alertVideo”, “videoCall”, “cameraManager” and “camera” required to complete the scenario.

Test case 65 defined for the “Energy Conservation” and the “Email” features shown in Figure 7.17 is an example of multi-component interaction sequence test case across features that are not dependent. The scenario tested is that when the residents are away, the house energy consumption gets adjusted and an informational email is send to the house resident. Although, the components of the “Energy Conservation” are not communicating directly with the components of the “Email” feature, they communicate through the components of the “Smart Home” feature. For instance, when the “away” output of the “tecTrack” component gets triggered, the “energyControl” component will send a notification to the “infoAlertHandler” component. The “infoAlertHandler” component will send a notification to the “email” component to notify the house residents. This test case tests the interaction sequence of the following components: “tecTrack”, “energyControl”, “infoAlertHandler”, and “email.”

The “Source Trigger” column of the test cases in Figure 7.17 shows all the triggers executed in the “Smart Home Example 2” application. Triggers are used to simulate external events in the smart space. For example as shown on test case 55 in Figure 7.17 when the "messageInQueue=true" triggering condition is true, the "sendAlert" output gets executed and through a sequence of component interactions, the "emergency" input is executed on the "emergencyCall" component. To verify that the test case executed successfully, the testResult attribute of the "emergencyCall" component was compared to the expected test result of the test case. For this test case it was found that (a) the input captured in the testResult attribute was emergency and (b) the parameter passed to the "emergency" input was "msg=help." Separate test cases were created to test triggering conditions that send messages to multiple inputs on different components. For example as shown in Figure 7.16 the "energyControl" component sends two independent messages when it receives a message from the "tecTrack" component. One message is to the "infoAlertHandler" component and another message is to the "smartHVAC." To test this scenario, two test cases were created: test case 61 and test case 62 shown in Figure 7.17 EU Application logging messages were also used to confirm that both events occurred when the "energyControl" component received a message from the "tecTrack" component.

EU Application Testing validated that all application components were derived for the features that comprise the "Smart Home Example 2" application, and the connectivity between components worked as were designed in the EU SPL development environment. Figure 7.18 shows the EU Application Testing output that executed the Derived Feature-based Test Cases against the component architecture of the "Smart Home Example 2" application. All Feature-based Integration Test Cases were executed successfully.



```
Test : 5 - Feature : Email - Status : PASS
Test : 6 - Feature : Email - Status : PASS
Test : 7 - Feature : Email - Status : PASS
Test : 8 - Feature : Email - Status : PASS
Test : 13 - Feature : Door - Status : PASS
Test : 14 - Feature : Door - Status : PASS
Test : 15 - Feature : Door - Status : PASS
Test : 16 - Feature : Door - Status : PASS
Test : 17 - Feature : Motion - Status : PASS
Test : 18 - Feature : Motion - Status : PASS
Test : 19 - Feature : Motion - Status : PASS
Test : 20 - Feature : Motion - Status : PASS
Test : 21 - Feature : Window - Status : PASS
Test : 22 - Feature : Window - Status : PASS
Test : 23 - Feature : Window - Status : PASS
Test : 24 - Feature : Window - Status : PASS
Test : 33 - Feature : Video - Status : PASS
Test : 34 - Feature : Video - Status : PASS
Test : 35 - Feature : Video - Status : PASS
Test : 36 - Feature : Video - Status : PASS
Test : 37 - Feature : Video - Status : PASS
Test : 38 - Feature : Video - Status : PASS
Test : 39 - Feature : Video - Status : PASS
Test : 40 - Feature : Video - Status : PASS
Test : 41 - Feature : Video - Status : PASS
Test : 42 - Feature : HVAC Filter - Status : PASS
Test : 53 - Feature : 911 - Status : PASS
Test : 54 - Feature : 911 - Status : PASS
Test : 55 - Feature : 911 - Status : PASS
Test : 56 - Feature : 911 - Status : PASS
Test : 57 - Feature : Energy Conservation - Status : PASS
Test : 58 - Feature : Energy Conservation - Status : PASS
Test : 59 - Feature : Energy Conservation - Status : PASS
Test : 60 - Feature : Energy Conservation - Status : PASS
Test : 61 - Feature : Energy Conservation - Status : PASS
Test : 62 - Feature : Energy Conservation - Status : PASS
Test : 63 - Feature : Energy Conservation - Status : PASS
Test : 64 - Feature : Energy Conservation - Status : PASS
Test : 65 - Feature : Energy Conservation / Email - Status : PASS
Test : 66 - Feature : Door  / 911  - Status : PASS
Test : 69 - Feature : Door  / 911  - Status : PASS
Test : 67 - Feature : Motion  / 911  - Status : PASS
Test : 68 - Feature : Window  / 911  - Status : PASS
Test : 70 - Feature : Motion  / Video - Status : PASS
Test : 71 - Feature : Window  / Video - Status : PASS
```

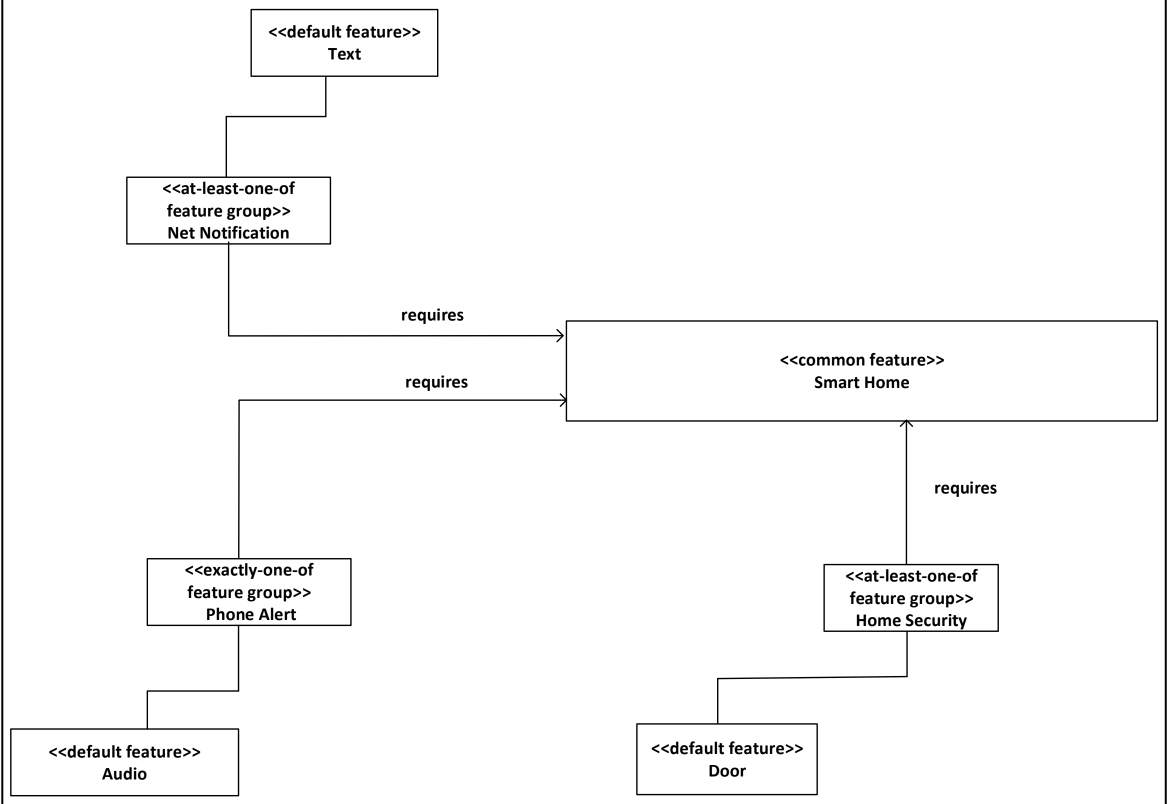
Figure 7.18 FeatureBasedTestDriver Output of executing the Featured-Based Integration Test Cases to the Smart Home Example 2 EU Application

7.7 Application Deployment Testing Process

The deployment of a derived application from the EUSPLP environment to the TeC EUD platform is a multi-step process. As described in Chapter 5, the first step of the deployment process is for the TeC EUSPLP Adaptor deployed in the TeC EUD environment to retrieve the TeC PSP for the derived application from the Application Distributor subsystem of the EUSPLP. The second step of the process is for the TeC EUSPLP Adaptor to store the derived application to the TeC platform. Finally the TeC environment deploys the derived application to the TeC devices of the smart space. To test each step of the application deployment process, a third application, "Smart Home Example 3" was derived from the Smart Home EU SPL that was deployed to the TeC Android simulator. To support the deployment and execution of "Smart Home Example 3", this research extended the TeC Android simulator with additional TeC devices from the Smart Home domain.

Figure 7.19 shows the Feature Model for the "Smart Home Example 3" derived application. As shown in Figure 7.19, the "Smart Home Example 3" application consists of the following features: "Smart Home", "Audio", "Text", and "Door." Figure 7.20 shows the application architecture of the derived application. EU Application Feature-based Consistency Checking and EU Application Testing were performed on the "Smart Home Example 3" application. Figure 7.21 shows the Feature-based Integration Test Cases related to the derived application. All test cases performed on the "Smart Home Example 3" application executed successfully.

The "Smart Home Example 3" derived application was imported successfully by the TeC EUSPLP Adaptor to the TeC simulator. Figure 7.22 shows three Android windows related to the application importing process. The left Android window shows the TeC EUSPLP Adaptor Android device retrieving the "Smart Home Example 3" derived application (PSP) from the Application Distributor subsystem. The two Android windows on the right show the imported application as it appears in the TeCEditor. The TeCEditor



<<default feature>> Text
<<at-least-one-of feature group>> Net Notification
requires
<<common feature>> Smart Home
requires
requires
<<exactly-one-of feature group>> Phone Alert
<<at-least-one-of feature group>> Home Security
<<default feature>> Audio
<<default feature>> Door

Figure 7.19 Smart Home Example 3 Application – Feature Model Deployed to the TeC Android Simulator

is used to develop TeC applications for the TeC Android platform. The TeCEditor was created as part of this research. In detail, the middle Android window of Figure 7.22 shows the ActivitySheet objects that the TeC EUSPLP Adaptor stored in TeC. The last Android window in Figure 7.22 shows all the ActivityConnector objects that were stored in TeC. In addition to verifying the TeCEditor, the TeC database entries were also verified to confirm that the Smart Home derived application was stored correctly. Finally to verify that the derived application functions as intended, the application was deployed to the TeC Devices that are part of the TeC Android simulator to simulate the execution of ActivitySheet objects. The TeC Device simulators provide a testing user interface for executing triggering

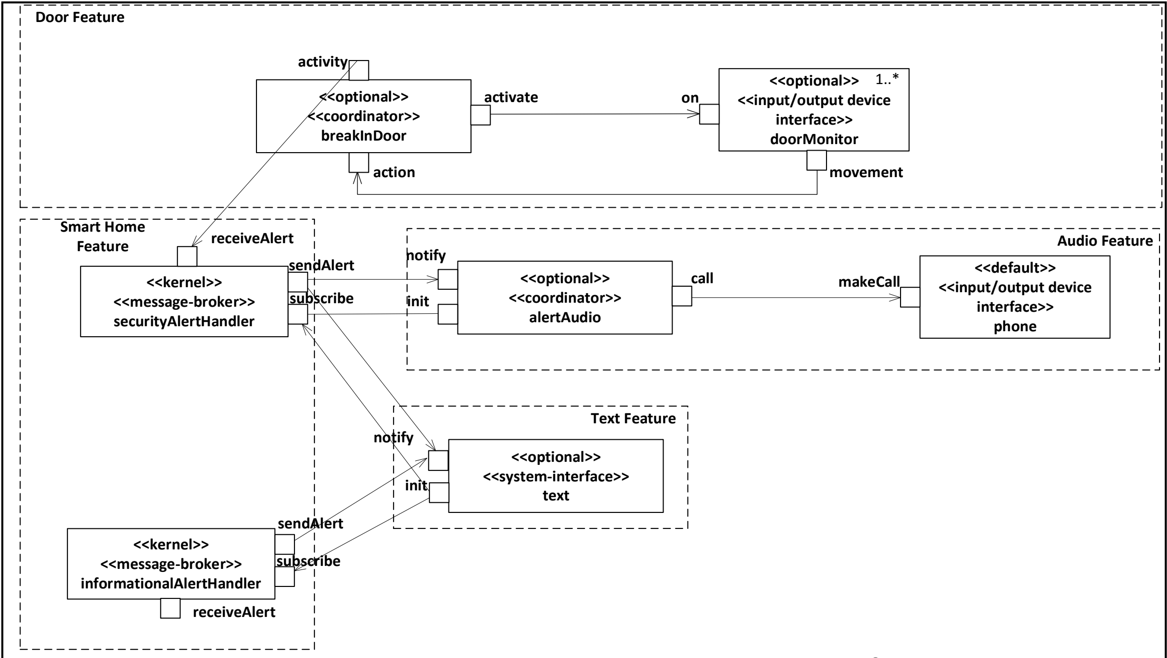
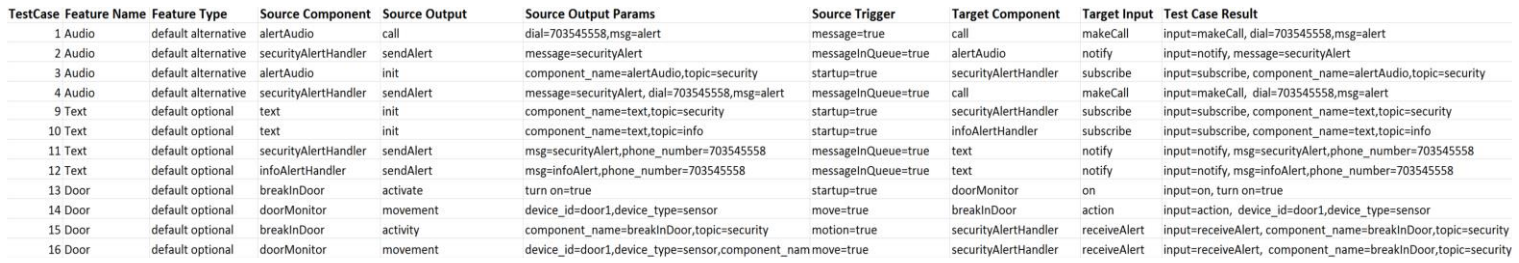


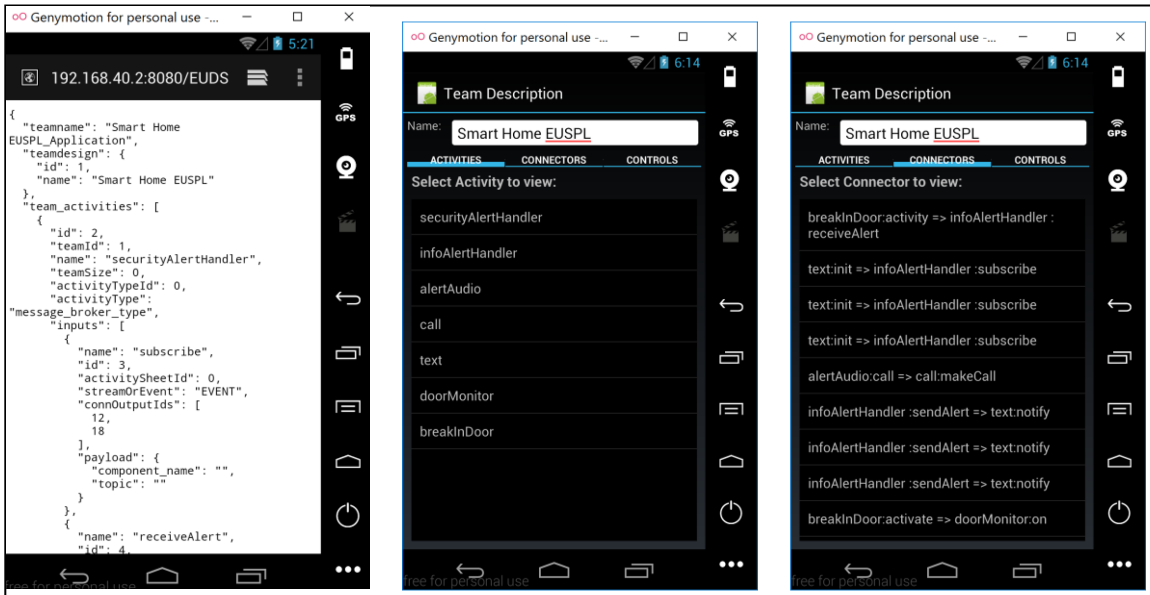
Figure 7.20 Smart Home Example 3 - Application Architecture for TeC

conditions for the ActivitySheet deployed in the devices. The testing interface of the TeC Device simulators was used to execute the test cases shown in Figure 7.21. All test cases were executed successfully. Figure 7.23 shows an example of executing the first test case in Figure 7.21 between two TeC Devices, “Coordinator” and “Notify”, that simulate the “alertAudio” and “phone” components respectively. When the “message=true” trigger executes in the “alertAudio” component, the “call” output executes, which causes the "Dial # is: 703545558 and Message: securityAlert” message to be displayed by the “phone” component. Similarly, all the other test cases shown on Figure 7.21 were executed successfully.



TestCase	Feature Name	Feature Type	Source Component	Source Output	Source Output Params	Source Trigger	Target Component	Target Input	Test Case Result
1	Audio	default alternative	alertAudio	call	dial=703545558,msg=alert	message=true	call	makeCall	input=makeCall, dial=703545558,msg=alert
2	Audio	default alternative	securityAlertHandler	sendAlert	message=securityAlert	messageInQueue=true	alertAudio	notify	input=notify, message=securityAlert
3	Audio	default alternative	alertAudio	init	component_name=alertAudio,topic=security	startup=true	securityAlertHandler	subscribe	input=subscribe, component_name=alertAudio,topic=security
4	Audio	default alternative	securityAlertHandler	sendAlert	message=securityAlert, dial=703545558,msg=alert	messageInQueue=true	call	makeCall	input=makeCall, dial=703545558,msg=alert
9	Text	default optional	text	init	component_name=text,topic=security	startup=true	securityAlertHandler	subscribe	input=subscribe, component_name=text,topic=security
10	Text	default optional	text	init	component_name=text,topic=info	startup=true	infoAlertHandler	subscribe	input=subscribe, component_name=text,topic=info
11	Text	default optional	securityAlertHandler	sendAlert	msg=securityAlert,phone_number=703545558	messageInQueue=true	text	notify	input=notify, msg=securityAlert,phone_number=703545558
12	Text	default optional	infoAlertHandler	sendAlert	msg=infoAlert,phone_number=703545558	messageInQueue=true	text	notify	input=notify, msg=infoAlert,phone_number=703545558
13	Door	default optional	breakInDoor	activate	turn on=true	startup=true	doorMonitor	on	input=on, turn on=true
14	Door	default optional	doorMonitor	movement	device_id=door1,device_type=sensor	move=true	breakInDoor	action	input=action, device_id=door1,device_type=sensor
15	Door	default optional	breakInDoor	activity	component_name=breakInDoor,topic=security	motion=true	securityAlertHandler	receiveAlert	input=receiveAlert, component_name=breakInDoor,topic=security
16	Door	default optional	doorMonitor	movement	device_id=door1,device_type=sensor,component_nam	move=true	securityAlertHandler	receiveAlert	input=receiveAlert, component_name=breakInDoor,topic=security

Figure 7.21 Derived Featured-Based Integration Test Cases for the Smart Home Example 3 EU Application



```
{
  "teamname": "Smart Home
EUSPL_Application",
  "teamdesign": {
    "id": 1,
    "name": "Smart Home EUSPL"
  },
  "team_activities": [
    {
      "id": 2,
      "teamId": 1,
      "name": "securityAlertHandler",
      "teamSize": 0,
      "activityTypeId": 0,
      "activityType":
"message_broker_type",
      "inputs": [
        {
          "name": "subscribe",
          "id": 3,
          "activitySheetId": 0,
          "streamOrEvent": "EVENT",
          "connOutputIds": [
            12,
            18
          ],
          "payload": {
            "component_name": "",
            "topic": ""
          }
        },
        {
          "name": "receiveAlert",
          "id": 4,
```

Team Description

Name: Smart Home EUSPL

ACTIVITIES | CONNECTORS | CONTROLS

Select Activity to view:

- securityAlertHandler
- infoAlertHandler
- alertAudio
- call
- text
- doorMonitor
- breakInDoor

Team Description

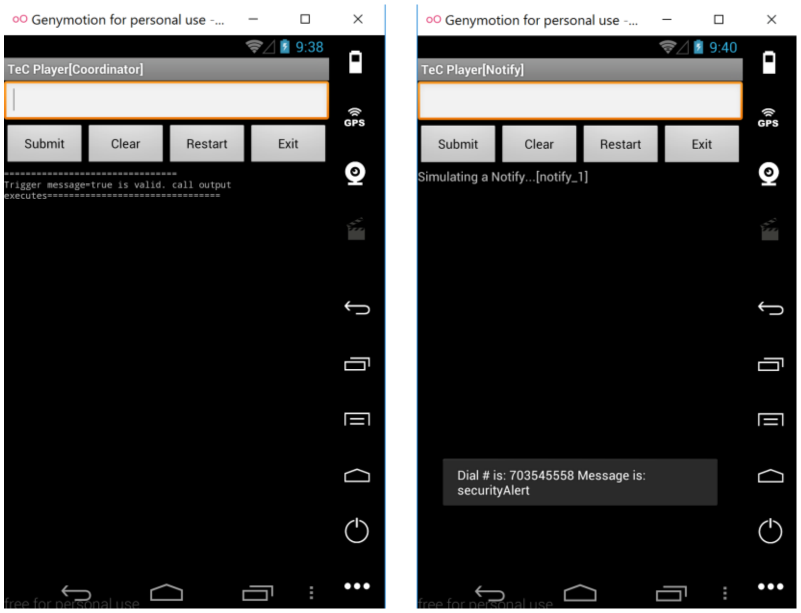
Name: Smart Home EUSPL

ACTIVITIES | CONNECTORS | CONTROLS

Select Connector to view:

- breakInDoor:activity => infoAlertHandler : receiveAlert
- text:init => infoAlertHandler :subscribe
- text:init => infoAlertHandler :subscribe
- text:init => infoAlertHandler :subscribe
- alertAudio:call => call:makeCall
- infoAlertHandler :sendAlert => text:notify
- infoAlertHandler :sendAlert => text:notify
- infoAlertHandler :sendAlert => text:notify
- breakInDoor:activate => doorMonitor:on

Figure 7.22 Smart Home Example 3 Derived Application Stored in TeC Android



TeC Player[Coordinator]

```
================================
Trigger message=true is valid. call output
executes================================
```

TeC Player[Notify]

Simulating a Notify...[notify_1]

Dial # is: 703545558 Message is: securityAlert

Figure 7.23 Executing a Test Case Example in TeC Device Simulators

7.8 Summary

This chapter has provided an overview of the validation process for this research. In summary, the Smart Home EU SPL case study was developed to validate: (a) the proposed design method for creating end user product lines, (b) the end user application derivation process, (c) the EUSPLP development environment for the product line creation process, (d) the EUSPLP environment for the application derivation process, and (e) the application deployment process. The EUSPLP environment was used to validate the EU SPL process and meta-model. To validate the TeC PSPLs produced by the EUSPLP environment, EU SPL Testing was performed. EU SPL Testing consisted of executing EU SPL Feature-based Consistency and Feature-based Integration Test Cases for the Smart Home EU SPL case study. All EU SPL Testing test cases executed successfully. In addition, to verify the TeC applications derived by the EUSPLP environment, EU Application Testing was performed. EU Application Testing consisted of executing EU Application Feature-based Consistency and Feature-based Test Cases to test two applications derived from the Smart Home EU SPL case study. All EU Application Testing test cases executed successfully in both derived applications. Finally to test the application deployment process a third application was derived from the Smart Home EU SPL that was deployed to and executed by the TeC Android simulator. The author of this dissertation developed and executed all test cases described in this chapter.

8 CONTRIBUTIONS AND FUTURE RESEARCH

8.1 Introduction

This dissertation has described a systematic approach and development environment for designing, developing and testing End User Software Product Lines (EU SPL) that end users can use to derive applications for their smart spaces. This research investigated the EU SPL process for technical end users and domain experts to create EU SPLs, which provides a step by step process for designing, developing and testing EU SPLs. The EU SPL process has extended existing product line approaches to end user development and smart spaces, as well as for deriving EU applications from the EU SPL. The EU SPL meta-model was designed to capture the underlying representation of end user product lines in terms of meta-classes and their relationships. The EUSPLP development environment was developed to enable the implementation of EU SPLs and application derivation for smart spaces. Finally a testing framework was developed to test the EU SPL and application models created using the EUSPLP development environment.

The remaining sections of this chapter describe the contributions of this research and future work. Section 8.2 describes the contributions of this research. Section 8.3 discusses areas where this research could be extended. Finally, section 8.4 provides a summary of this chapter.

8.2 Research Contributions

This section discusses the contributions of this research as they relate to the research goals described in Chapter 1. The overall contributions of this research are: (a) the End

User Product Line Engineering (EUPLE) process, (b) the End User Application Engineering (EUAE) process, (c) the EU SPL meta-model, (d) the EUSPLP development environment, and (e) the EU SPL Testing framework. The following subsections briefly detail the contributions of this research.

8.2.1 End User Product Line Engineering (EUPLE) Process

The End User Product Line Engineering (EUPLE) process for designing, developing and testing EU SPLs for smart spaces, is one of the contributions of this research. The EUPLE process is part of the EU SPL process. In particular, the EUPLE process provides EU SPL designers with a systematic approach for designing and developing EU SPLs. The EUPLE process extended conventional Product Line Engineering (PLE) approaches (Gomaa, 2005a) to account for EUD development and smart spaces. SPL design artifacts were extended by the EUPLE process to capture platform and component / connector architecture information available in smart spaces. The EUPLE process provides a lightweight product line approach for technical end users and domain experts to design and develop EU SPLs that can be used to derive applications for different EUD environments.

8.2.2 End User Application Engineering (EUAE) Process

The End User Application Engineering (EUAE) process for deriving end user applications from the EU SPL is another contribution of this research. The EUAE process is part of the EU SPL process. In particular, the EUAE process enables end users to derive software applications for their smart spaces. The EUAE process extended conventional

Application Engineering approaches (Gomaa, 2005a) to account for end users and smart spaces. In conventional Application Engineering, application engineers and application test engineers work with end users to derive and install applications from the product line. The EUAE process is executed by end users. EUAE provides sub-processes for collecting end user requirements for smart spaces, deriving the EU application architecture, testing the application, and deploying the application to the smart space. The EUPLE process provides a lightweight approach for end users to derive applications from the EU SPLs for their spaces.

8.2.3 EU SPL Meta-model

The EU SPL meta-model is another contribution of this research. The EU SPL meta-model is used to capture the underlying representation of EU SPLs and derived applications artifacts in terms of meta-classes and relationships. The EU SPL meta-model extended conventional SPL meta-models with support for EUD environments. In addition, the EU SPL meta-model contains platform independent and platform specific meta-models. Platform independent meta-models are used to capture the underlying representation of end user product lines and applications in terms of meta-classes and relationships independent of the EUD environment. Platform independent product lines are beneficial because they can be reused to derive applications for different EUD environments. Platform specific meta-models are applicable to specific EUD environments. Platform specific meta-models are beneficial when designing an end user product line that uses exclusive functions of a specific EUD environment.

8.2.4 EUSPLP Development Environment

The End User Software Product Line Prototype (EUSPLP) development environment used to validate this research is another contribution. This development environment enables: (a) EU SPL designers to develop end user product lines, and (b) End users to derive and deploy applications for their smart spaces. The EUSPLP environment is different from conventional SPL environments as it is based on the EU SPL process and targets end users. The EUSPLP provides different user interfaces for supporting EU SPL development and application derivation. EU SPL designers use the EU SPL development user interface to design and implement end user product lines. End users use the application derivation user interface to derive applications for their spaces. The EUSPLP is integrated with the TeC Android environment for application deployment. The EUSPLP design supports the deployment of derived applications to additional EUD environments by developing EUSPLP adaptors for each different end user development environment. The EUSPLP environment was implemented using open source technologies and is web-based.

As part of the EUSPLP environment, an end user oriented visual language was defined to support the development of EU SPLs and application derivation. In particular, during EU SPL design, the feature model is represented as a tree structure to capture feature and feature group dependencies. During application derivation, end users are presented with a different view of the feature model applicable for feature selection. The visual language is beneficial for developing end user product lines since it uses simple visual representations and symbols to capture complex product line terminology.

8.2.5 EU SPL Testing Approach

The EU SPL Testing Approach is another contribution of this research. The testing approach extended conventional SPL testing approaches for end user product lines and derived applications. In particular, the testing approach consists of three sub-processes: (1) EU SPL Testing, (2) EU Application Testing, and (3) EU Application Deployment Testing. The EU SPL testing process executes at the product line level, and tests feature dependencies and component interconnections of the EU SPL. The EU Application Testing process executes applications derived from the EU SPL, and tests the validity of each feature combination that composes the derived application in addition to the application component interconnections. The EU Application Deployment Testing process executes during the application deployment to the smart space and tests that the application has been deployed successfully and executes correctly. The EU SPL Testing framework is beneficial since it provides testing throughout the EU SPL process.

8.3 Future Research

This section discusses possible future research for extending this work. The proposed future work in this section can further promote the adoption of end user software product lines for end user development of smart spaces.

8.3.1 Smart Space Security models for End User Software Product Lines

There are several security challenges in multi-user smart spaces. Some of the issues involve authentication, access control, privacy and confidentiality of communication (Jani Suomalainen and Pasi Hyttinen, 2011). Each EUD environment has its own mechanisms for addressing these challenges. Additional research can be conducted to create a security meta-model that addresses the authentication, access control, privacy and confidentiality security attributes of smart spaces, which can be used in the design, implementation and testing of EU SPLs. The security meta-model could be mapped to security models of different EUD environments. In addition, different design artifacts that address each of the security attributes could be used to expand the EU SPL process.

8.3.2 End User Visual Languages for End User Software Product Lines

A visual language was developed as part of the EUSPLP development environment to enable technical end users and domain experts to create EU SPLs and end users to derive applications for their environments. This research performed a preliminary user study (Tzeremes and Gomaa, 2016b) to investigate (a) different visual symbols for representing feature types, and (b) user interfaces for creating EU SPLs and deriving applications for smart spaces. An extension of the original user study could be conducted to ensure that the visual language and user interface created in the EUSPLP is sufficient for (a) technical end users and domain experts to create EU SPLs, and (b) end users to derive applications.

8.3.3 Enhancements to the EUSPLP Development Environment

The EUSPLP development environment provides functions for creating EU SPLs and deriving applications for the TeC EUD environment. The prototype can be extended to support additional EUD environments for smart spaces using the meta-models described in Chapter 5. A conversion mechanism could be investigated to convert EU SPLs created by the EUSPLP to different EUD smart spaces. Additional research can be performed in the area of addressing conflicts between the smart space security policy and the EU SPL features. Finally, additional research can be conducted in extending this prototype to other domains of end user development and product line development.

8.3.4 Testing of End User Software Product Lines

This research developed a testing approach and framework for testing end user product lines for smart spaces. The testing framework could be enhanced by investigating approaches to automatically generate test cases based on feature dependencies and component relationships, in addition to test cases provided by EU SPL designers. Another area that needs additional research is automated methods for testing mobile systems (Canfora et al., 2013) that can be integrated with the EU SPL process. For instance the TeC Android simulator (Shen, 2014) could be extended with an automated method for software testing. Furthermore additional research is needed in incorporating usability testing (Brinkman et al., 2008) in the EU SPL process. Usability testing can assist EU SPL designers to ensure that feature designs are easy to use and increase the satisfaction of end users.

8.3.5 Evolution of End User Product Lines for Smart Spaces

As part of this research a manual process was created for EU SPL designers to communicate with end users to address the evolution of EU SPL. New requirements are identified by end users, defects are addressed, and new features are added and other features are retired. An automated process could be investigated that (a) informs end users about updates in features that are part of derived applications deployed in their spaces, (b) informs end users about new features that are applicable to their spaces, (c) tests and deploys enhancements to derived applications, and (d) reports defects back to EU SPL designers.

8.4 Summary

This dissertation has described an approach for designing, developing and testing end user product lines for smart spaces. This research investigated an EU SPL process for creating EU SPLs and deriving applications for smart spaces. This research also defined a meta-model that captures the underlying representation of the commonality and variability of EUD smart spaces and product lines. A prototype was created to validate the approach and to enable EU SPL development and application derivation. The Smart Home EU SPL was created as a case study to validate the different parts of this research. A testing approach and supporting testing framework was developed to test end user product lines and derived applications. Security for smart spaces, visual languages for EU SPLs, EUSPLP enhancements, extensions to the testing framework and EU SPL evolution are some areas that could further enhance this research.

A APPENDIX: SMART HOME EU SPL CASE STUDY

A.1 Introduction

The Smart Home EU SPL case study presented in this appendix was developed in this research following the EU SPL Process described in Chapter 4 and was used to validate this research. Smart homes are physical environments equipped with sensors, actuators, appliances and devices that can react proactively or reactively to environment changes. End User Development (EUD) environments for smart homes integrate sensors, actuators, appliances and devices and provide end user friendly interfaces to allow ordinary end users to create applications for their environments. As smart homes evolve and get additional instrumentation they become complex and it can be difficult for ordinary end users to create software applications using EUD environments. By adopting the EU SPL process described in this research advanced end users and domain experts can develop end user product lines for smart spaces. Ordinary end users can use end user product lines to select features, derive and deploy applications for their homes.

The Smart Home EU SPL case study presents an end user product line created for a complex smart home. The case study includes features from the domains of home automation, home security, home notifications, home maintenance, resident comfort and energy conservation. The case study was developed following the EU SPL Process. In particular, the End User Product Line Engineering (EUPLE) process was used to design and develop the case study and the End User Application Engineering process was used to derive applications.

The appendix is organized as follows. Section A.2 describes the EUPLE process (requirements elicitation, feature modeling, analysis modeling and design modeling) used to create the Smart Home EU SPL. Section A.3 describes how the EUAE process was used to derive end user applications from the Smart Home EU SPL for the TeC and Jigsaw EUD environment. Finally, section A.4 summarizes this chapter.

A.2 End User Product Line Engineering (EUPLE)

End User Product Line Engineering (EUPLE) is the process that EU SPL designers (technical end users and domain experts) follow to develop EU SPLs. This section describes the EU SPL Requirements Elicitation, EU SPL Analysis modeling and EU SPL Design modeling as related to the Smart Home EU SPL case study.

A.2.1 EU SPL Requirements Elicitation

EU SPL requirements elicitation involves a set of activities to help define the overall scope of the product line. EU SPL designers with domain expertise define the overall road map for the EU SPL. Then EU SPL designers work with end users to collect and document requirements. Based on product line scoping and requirements, the product line feature model is defined. This section describes the end user requirement elicitation process and provides examples for a smart home case study. In detail section A.2.1.1 describes the Smart Home EU SPL features. Section A.2.1.2 presents the Smart Home EU SPL feature model. Section A.2.1.3 shows the product line features groups and their features in a tabular view.

A.2.1.1 Smart Home EU SPL Feature Description

Table A.1 provides a summary of the features that comprise the Smart Home EU SPL case study.

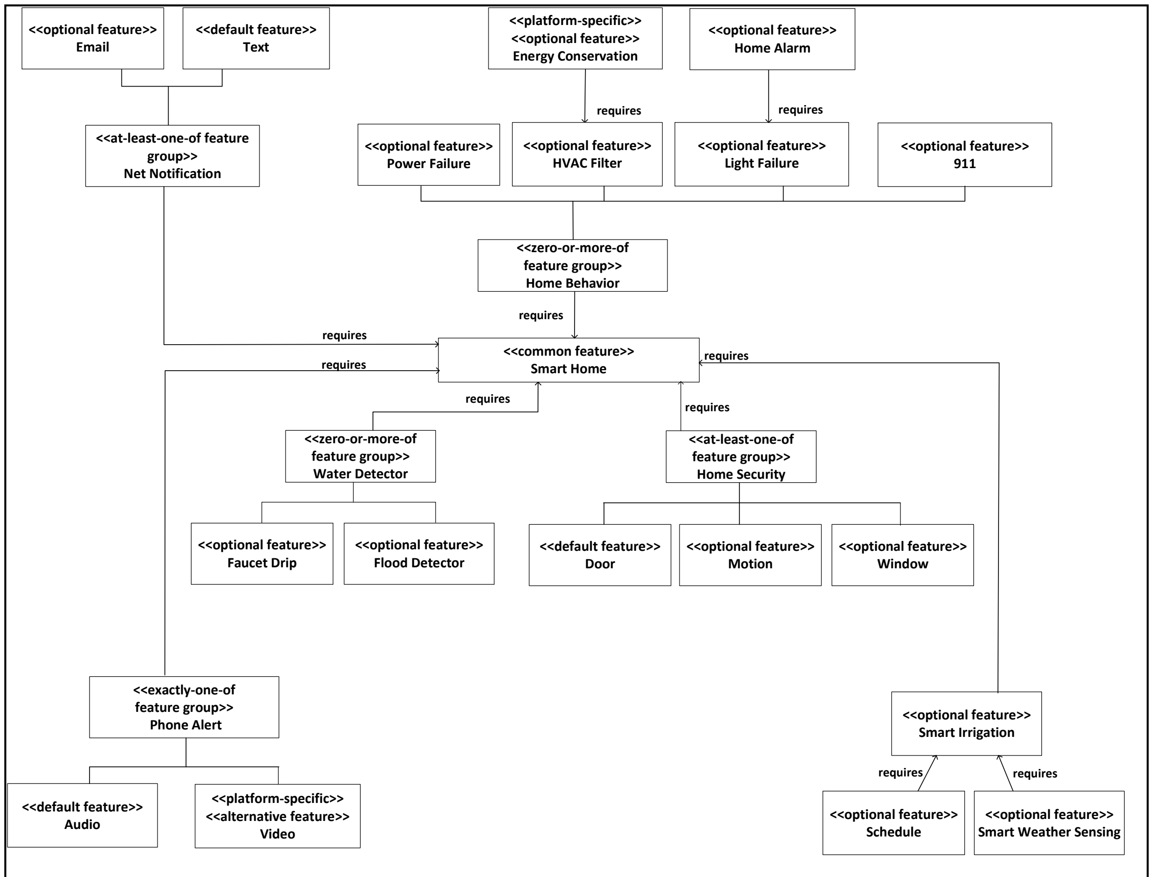
Table A.1 Smart Home EU SPL Feature Description

Feature Name	**Feature Description**
Smart Home	Provides common mechanisms for informational and security notifications
Audio	Provides audio notifications to the home residents phone when there are security alerts
Video	Provides video notifications to the home residents smart phone when there are security alerts
Home Alarm	The siren, flashing Lights and smart displays get activated when a security bridge is detected
911	The police is notified when a security bridge is detected
Door	Door sensors send security notifications that the doors have been bridged
Motion	Motion sensors send security notifications that the doors have been bridged
Window	Window sensors send security notifications that the doors have been bridged
Smart Irrigation	Controls the sprinkler system
Schedule	Starts the sprinkler system based on a schedule
Smart Weather Sensing	Starts the sprinkler system based on the soil moisture
Email	Provides email notifications to the home residents phone when there are informational or security alerts
Text	Provides text notifications to the home residents phone when there are informational or security alerts
Light Failure	Light sensors send informational notifications when a light bulb need to be changed
HVAC Filter	HVAC filter quality sensors send informational notifications when the filter needs to be changed
Power Failure	Power Failure sensors send informational notifications when a device has no power
Energy Conservation	When the home residents are away the home adjusts the home appliances to lower energy consumption. The home adjust to normal energy levels when the home residents are back in the house
Flood Detector	Moisture sensors send informational notifications when a flood is detected
Faucet Drip	Faucet sensors send informational notifications when a faucet keeps dripping

A.2.1.2 Smart Home EU SPL Feature Model

Feature modeling is used to capture feature commonality/variability and feature dependencies within the EU SPL. In a feature model, features can be organized (a) as common or variable, (b) in feature groups, and (c) as parameterized features. Figure A.1 shows the feature model for the Smart Home EU SPL case study. As shown in Figure A.1 the smart home feature model has one common feature called Smart Home that all other features and feature groups depend on. There is one optional feature Smart Irrigation that depends on the Smart Home feature. The Schedule and Smart Weather Sensing features are also optional and depend on the Smart Irrigation feature. There is one exactly-one-of feature group called Phone Alert that depends on the Smart Home feature. The Phone Alert feature group has two mutually exclusive features Audio and Video. The Audio feature is the default feature and Video is the alternative feature. Default features are selected by default if no other feature in the feature group is selected. The Video feature is platform specific.

The feature model also contains two at-least-one-of feature groups: Net Notification and Home Security. Both of the feature groups depend on the Smart Home common feature. The Net Notification feature group contains two optional features Email and Text. Text is the default feature. The Home Security feature group contains three optional features: Door, Motion and Window. Door is the default option of the feature group. The Smart Home feature model also contains two zero or more feature groups: Water Detector and Home Behavior. The Water Detector feature group contains two optional features Faucet Drip and Flood Detector. The Home Behavior feature group



<<optional feature>> Email
<<default feature>> Text
<<platform-specific>> <<optional feature>> Energy Conservation
<<optional feature>> Home Alarm
requires
requires
<<at-least-one-of feature group>> Net Notification
<<optional feature>> Power Failure
<<optional feature>> HVAC Filter
<<optional feature>> Light Failure
<<optional feature>> 911
<<zero-or-more-of feature group>> Home Behavior
requires
requires
requires
<<common feature>> Smart Home
requires
requires
requires
<<zero-or-more-of feature group>> Water Detector
<<at-least-one-of feature group>> Home Security
<<optional feature>> Faucet Drip
<<optional feature>> Flood Detector
<<default feature>> Door
<<optional feature>> Motion
<<optional feature>> Window
<<exactly-one-of feature group>> Phone Alert
<<optional feature>> Smart Irrigation
requires
requires
<<default feature>> Audio
<<platform-specific>> <<alternative feature>> Video
<<optional feature>> Schedule
<<optional feature>> Smart Weather Sensing

Figure A.1 Smart Home EU SPL Feature Model

contains four optional features: Power Failure, HVAC Filter, Light Failure and 911. In addition the Home Alarm optional feature depends on the Light Failure feature. Furthermore the Energy Conservation optional feature depends on the HVAC Filter. The Energy Conservation feature also is platform specific.

A.2.1.3 Smart Home EU SPL Feature Group / Feature Dependency Table

The Feature group / Feature dependency table is another view that captures the relationship between product line features and feature groups. The Feature group / Feature dependency table assists EU SPL designers to ensure consistency between features and feature groups.

Table A.2 shows the Feature Group / Feature dependency table for the Smart Home case study. The table captures the Smart Home EU SPL feature groups with features dependencies. The purpose of this table is to ensure consistency between each feature group and the features it contains. Table A.2 the table has four columns: (a) Feature Group Name, (b) Feature Group Category, (c) Feature Name, and (d) Feature Category. The Feature Group Category and Feature Category need to be compatible for example exactly-one-of feature group needs to have a set of alternative features since only one can be selected. For example as shown in Table A.2 the Phone Alert exactly-one-of feature group has two alternative features Audio and Video with the Audio feature being the default option.

Table A.2 Smart Home EU SPL Feature Group / Feature Dependency Table

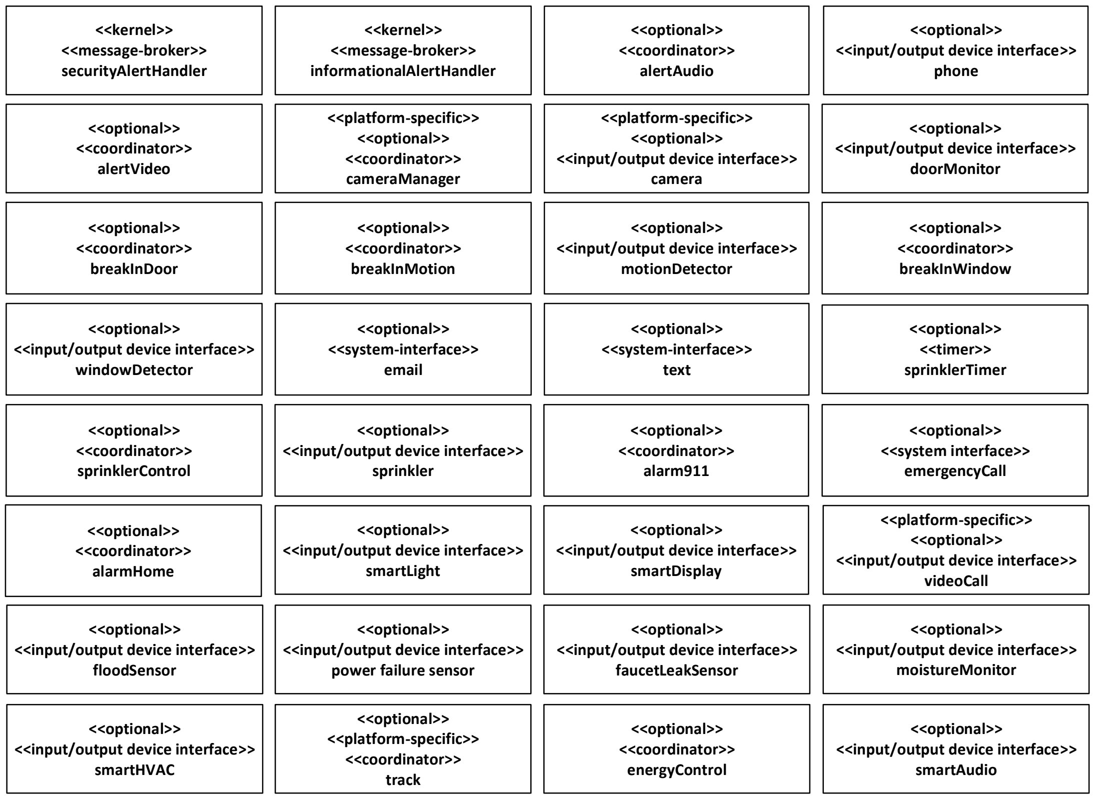
Feature Group Name	**Feature Group Category**	**Features in Feature Group**	**Feature Category**
Phone Alert	exactly-one-of	Audio Video	default alternative
Home Security	at-least-one-of	Door Motion Window	default optional optional
Water Detector	zero-or-more-of	Flood Detector Faucet Drip	optional optional
Home Behavior	zero-or-more-of	Light Failure HVAC Filter Power Failure 911	optional optional optional optional
Net Notification	at-least-one-of	Text Email	default optional

A.2.2 EU SPL Analysis Modeling

EU SPL Analysis modeling consists of static modeling and component structuring, dynamic modeling and feature / component modeling. Section A.2.2.1 describes the Smart Home EU SPL static model and component structuring. Section A.2.2.2 captures the Smart Home EU SPL dynamic modeling in the form of sequence diagrams. Section A.2.2.3 provides details about the feature/component dependencies.

A.2.2.1 Smart Home EU SPL Static Model and Component Structuring

Figure A.2 shows the static model and the component structuring for the components used in the Smart Home case study. The Smart Home EU SPL static model is composed of the platform specific feature / component table and the components diagram. The components diagram shown in Figure A.2 captures all the components used on the Smart Home EU SPL annotated with the reuse, role and platform dependency UML stereotypes. For example as shown on the securityAlertHandler component is annotated with the <<kernel>> stereotype to capture reuse category and the <<message-broker>> stereotype to capture the component role category. Similar the component videoCall is annotated with the <<optional>> stereotype to capture the reuse category, the <<input / output device interface>> stereotype to capture the role category and the <<platform-specific>> stereotype to indicate that this component only applies to specific platforms.


<<kernel>> <<message-broker>> securityAlertHandler
<<kernel>> <<message-broker>> informationalAlertHandler
<<optional>> <<coordinator>> alertAudio
<<optional>> <<input/output device interface>> phone
<<optional>> <<coordinator>> alertVideo
<<platform-specific>> <<optional>> <<coordinator>> cameraManager
<<platform-specific>> <<optional>> <<input/output device interface>> camera
<<optional>> <<input/output device interface>> doorMonitor
<<optional>> <<coordinator>> breakInDoor
<<optional>> <<coordinator>> breakInMotion
<<optional>> <<input/output device interface>> motionDetector
<<optional>> <<coordinator>> breakInWindow
<<optional>> <<input/output device interface>> windowDetector
<<optional>> <<system-interface>> email
<<optional>> <<system-interface>> text
<<optional>> <<timer>> sprinklerTimer
<<optional>> <<coordinator>> sprinklerControl
<<optional>> <<input/output device interface>> sprinkler
<<optional>> <<coordinator>> alarm911
<<optional>> <<system interface>> emergencyCall
<<optional>> <<coordinator>> alarmHome
<<optional>> <<input/output device interface>> smartLight
<<optional>> <<input/output device interface>> smartDisplay
<<platform-specific>> <<optional>> <<input/output device interface>> videoCall
<<optional>> <<input/output device interface>> floodSensor
<<optional>> <<input/output device interface>> power failure sensor
<<optional>> <<input/output device interface>> faucetLeakSensor
<<optional>> <<input/output device interface>> moistureMonitor
<<optional>> <<input/output device interface>> smartHVAC
<<optional>> <<platform-specific>> <<coordinator>> track
<<optional>> <<coordinator>> energyControl
<<optional>> <<input/output device interface>> smartAudio
Figure A.2 Smart Home Case Study Static Model

The Platform Specific Feature / Component relationship table captures the relationship between platform specific features and platform specific components. As shown in Table A.3 the platform specific feature / component relationship table has 4 columns: (a) Feature Name, (b) Platform Name, (c) Component Name, and (d) Platform Specific Name. The Feature Name column captures the name of the feature. The Platform Name column captures the end user platform(s) that the feature applies. The Component Name column captures the component name as it appears on the static model. The Platform Specific Name column captures the actual component name in the specific platform. For example the Energy Conservation feature applies only to the TeC platform.

Table A.3 Platform Specific Feature / Component relationship table

Feature Name	Platform Name	Component Name	Platform Specific Name
Energy Conservation	Team Computing	track	tecTrack
Video	Team Computing	videoCall cameraManager camera	tecVideoCall tecCameraManager tecCamera

The track component of the Energy Conservation feature would have to be mapped to the tecTrack component of Team computing during the end user application deployment process.

A.2.2.2 Smart Home EU SPL Dynamic Modeling

EU SPL designers use dynamic modeling to capture the object interactions needed to satisfy EU SPL features. This research used UML sequence diagrams to model object interactions. Sequence diagrams model the message interaction of objects based on a time sequence (Rumbaugh et al., 2004). Figure 4.3 to Figure A.20 show the sequence diagrams developed for each feature defined in the feature model. The components securityAlertHandler and informationalAlertHandler are kernel components and support the Smart Home Feature. The kernel components support all sequence diagrams.

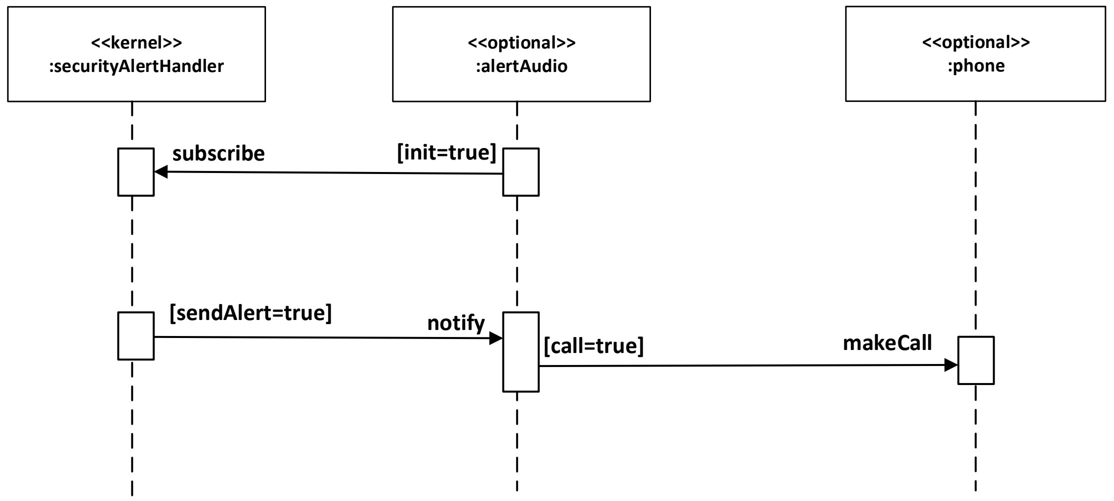


Figure A.3 Sequence Diagram for the Smart Home EU SPL Audio Feature

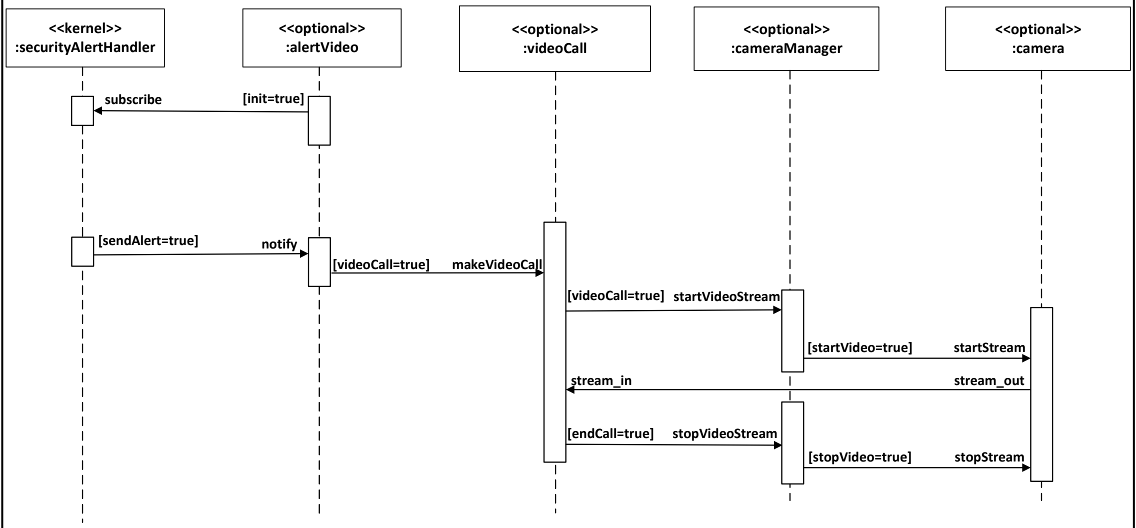


Figure A.4 Sequence Diagram for the Smart Home EU SPL Video Feature

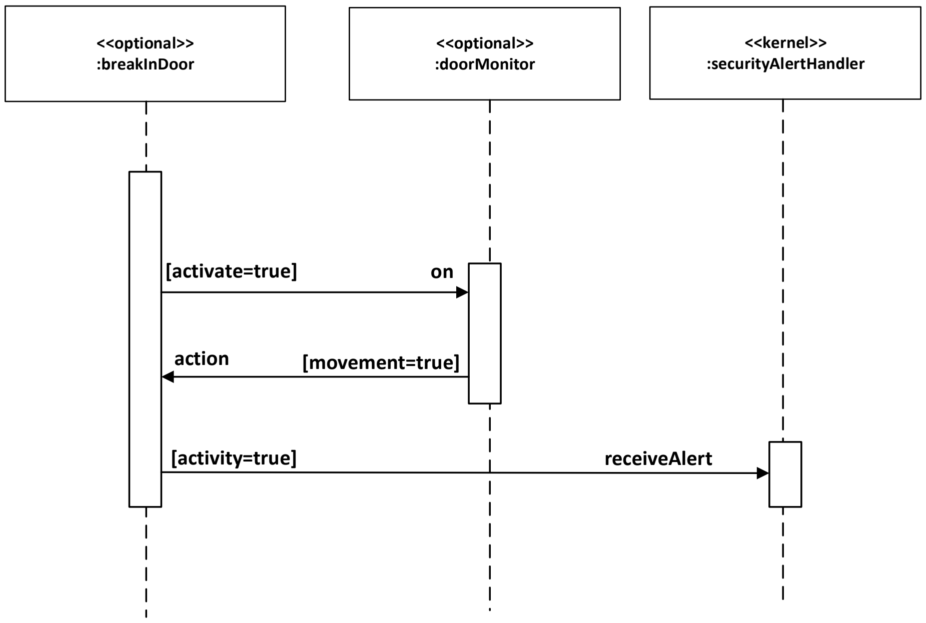


Figure A.5 Sequence Diagram for the Smart Home EU SPL Door Feature

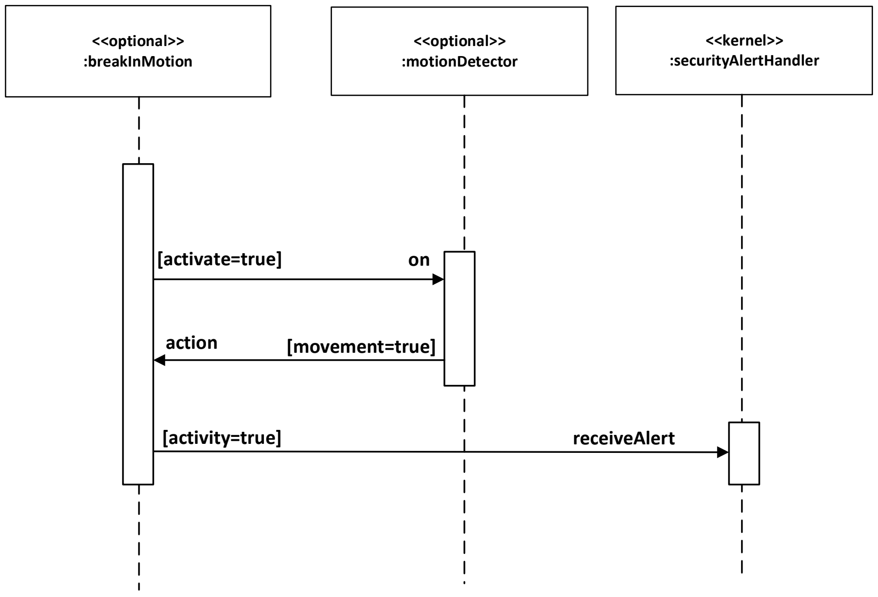


Figure A.6 Sequence Diagram for the Smart Home EU SPL Motion Feature

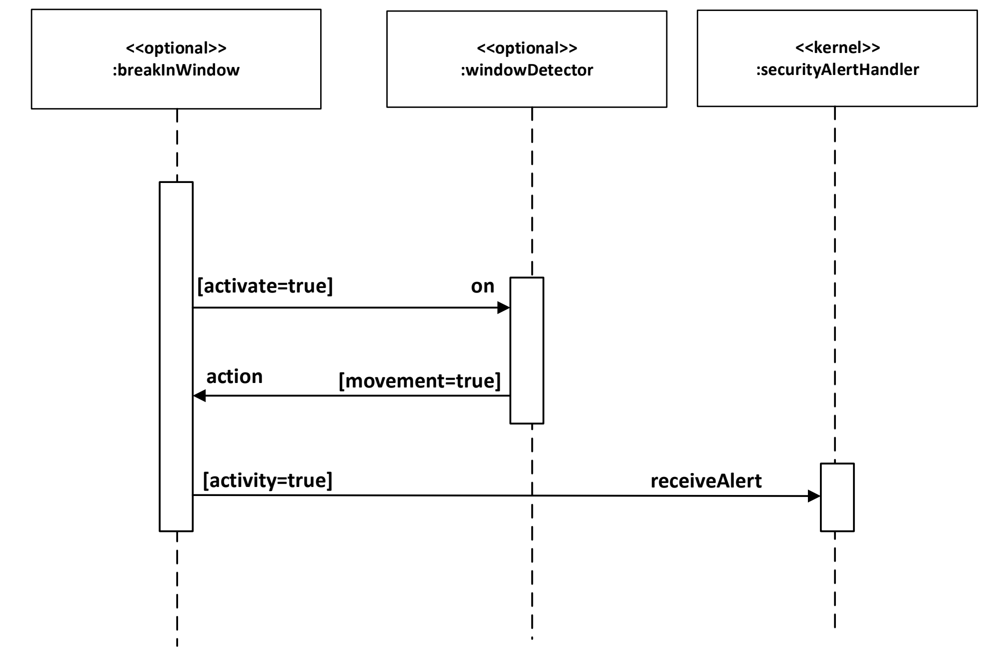


Figure A.7 Sequence Diagram for the Smart Home EU SPL Window Feature

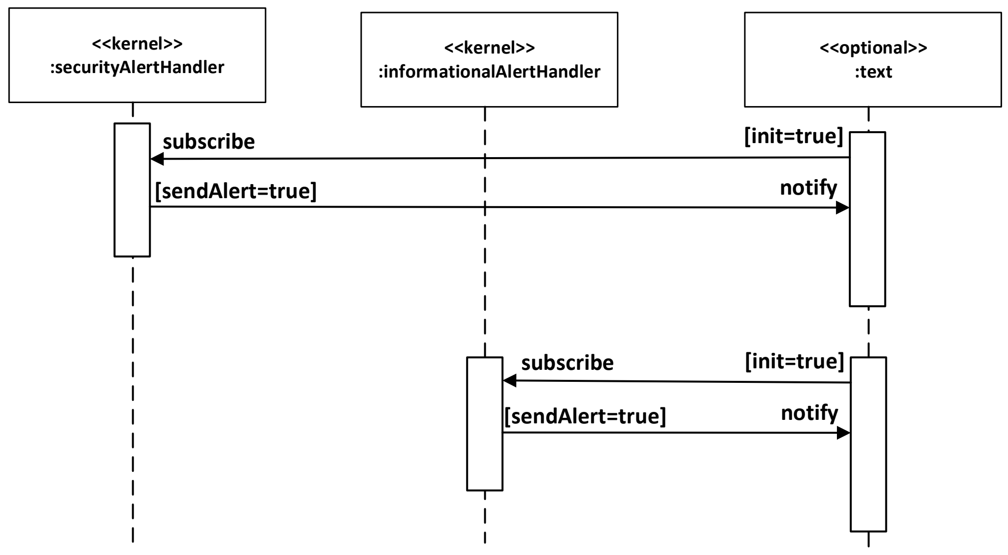


Figure A.8 Sequence Diagram for the Smart Home EU SPL Text Feature

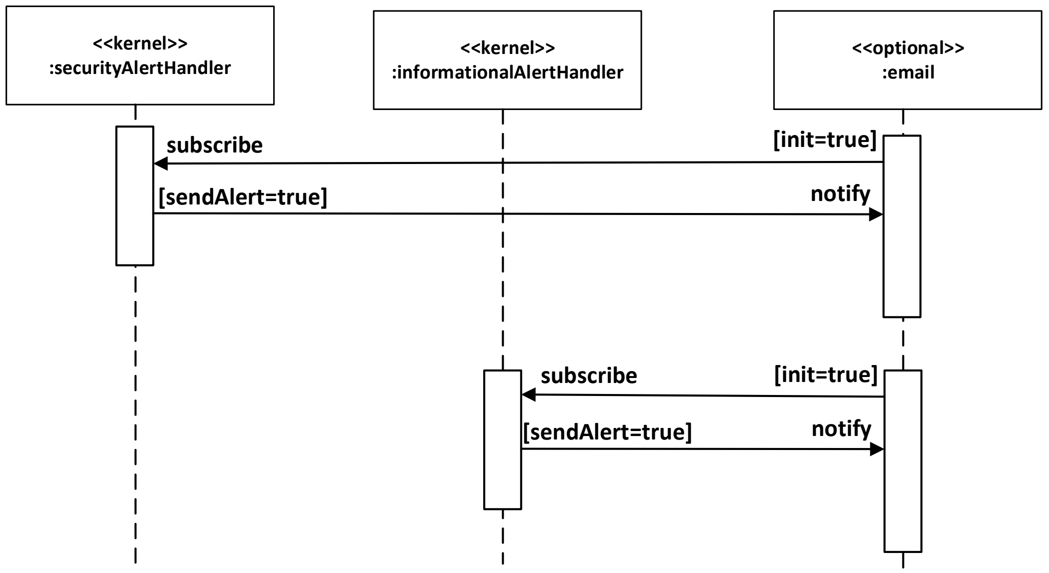


Figure A.9 Sequence Diagram for the Smart Home EU SPL Email Feature

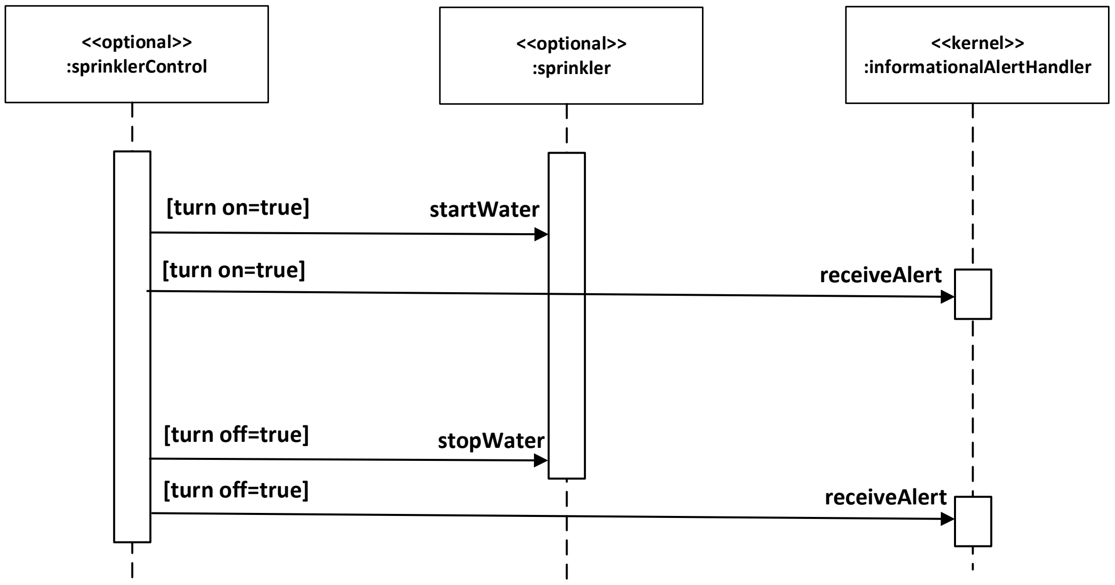


Figure A.10 Sequence Diagram for the Smart Home EU SPL Smart Irrigation Feature

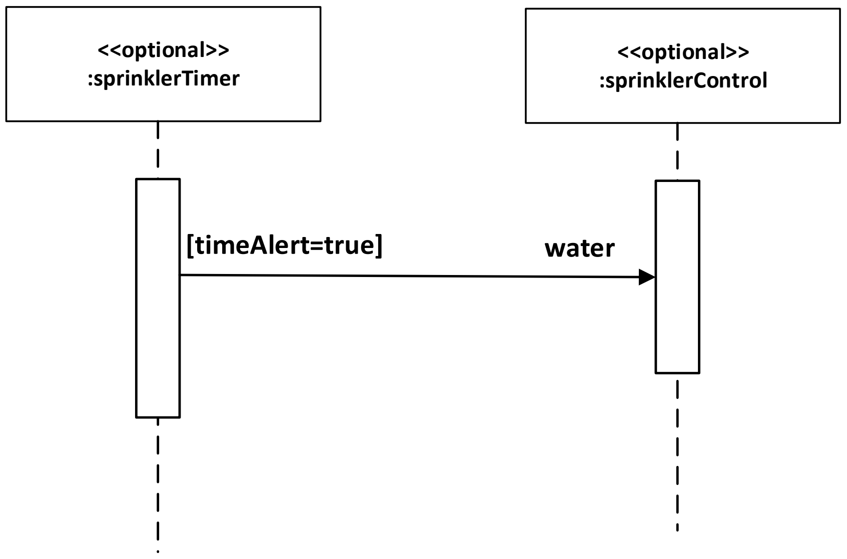


Figure A.11 Sequence Diagram for the Smart Home EU SPL Schedule Feature

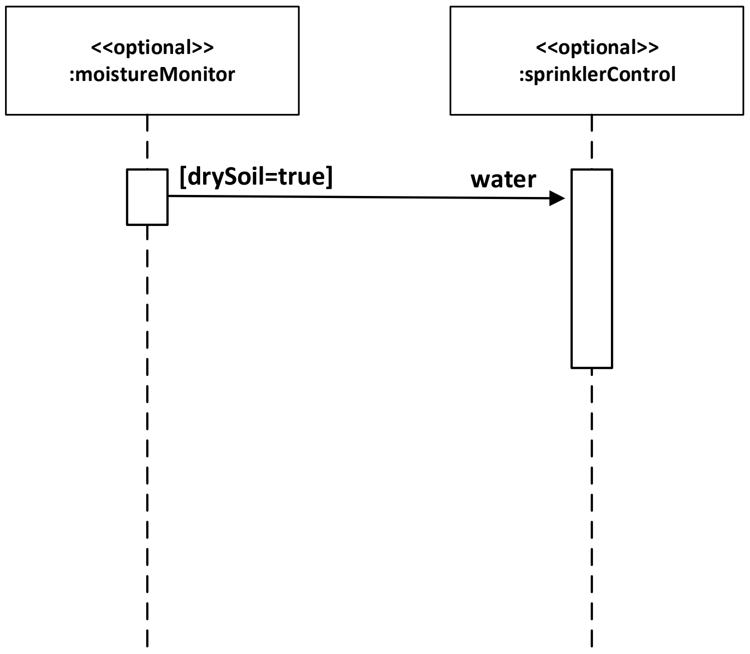


Figure A.12 Sequence Diagram for the Smart Home EU SPL Smart Weather Sensing Feature

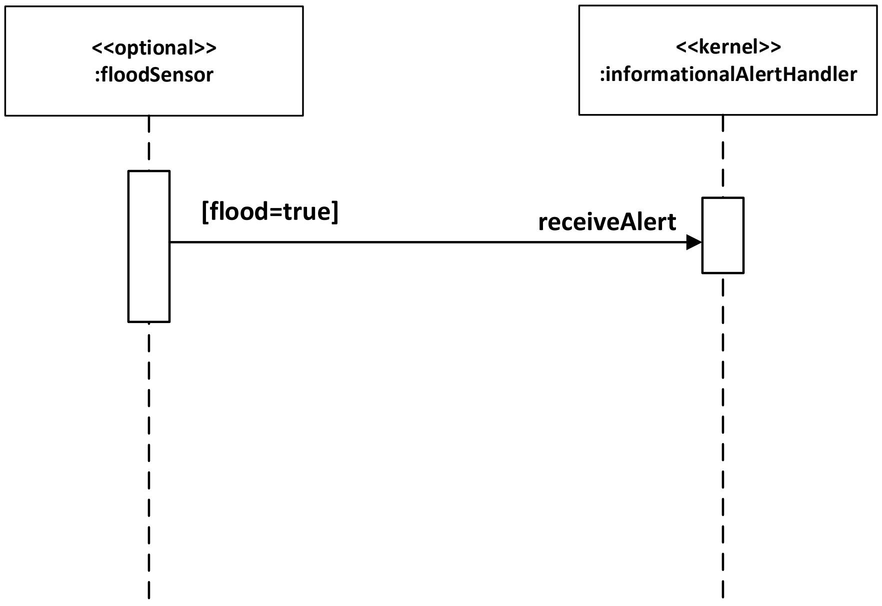


Figure A.13 Sequence Diagram for the Smart Home EU SPL Flood Detector Feature

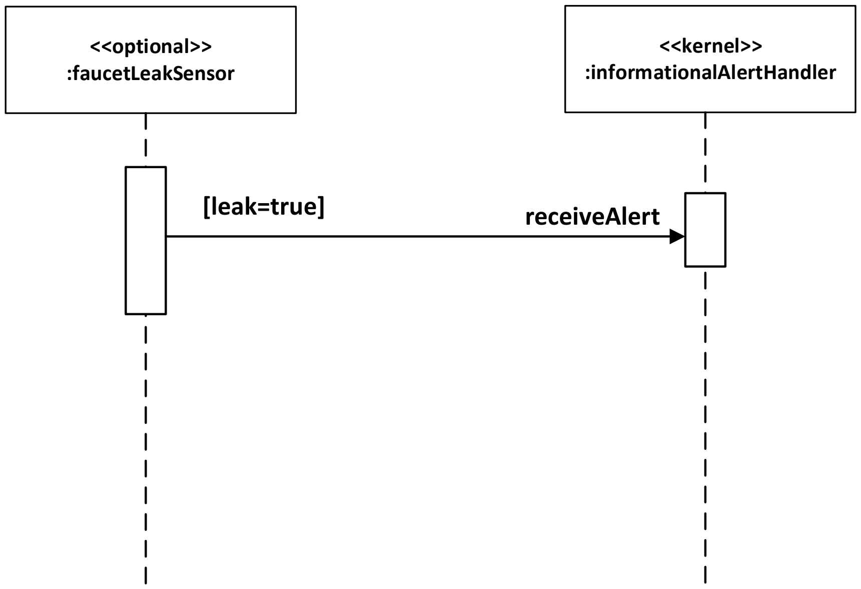


Figure A.14 Sequence Diagram for the Smart Home EU SPL Faucet Drip Feature

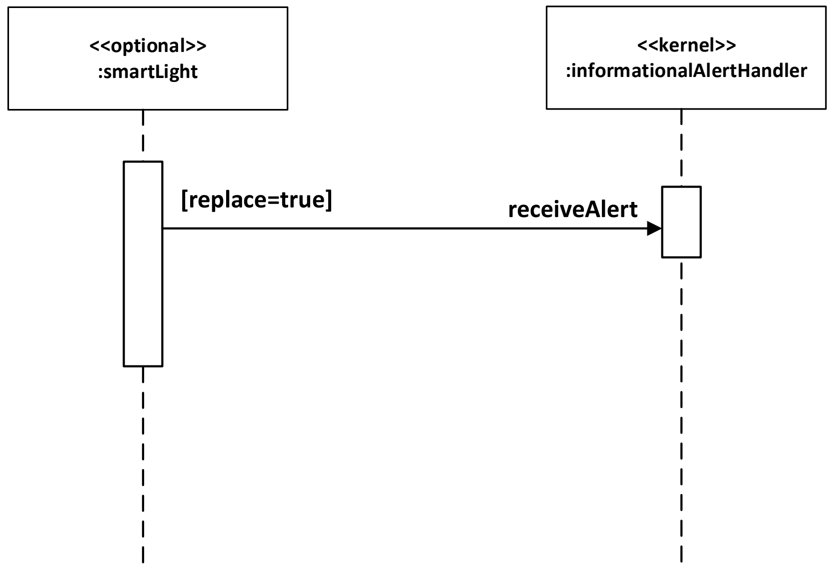


Figure A.15 Sequence Diagram for the Smart Home EU SPL Light Failure Feature

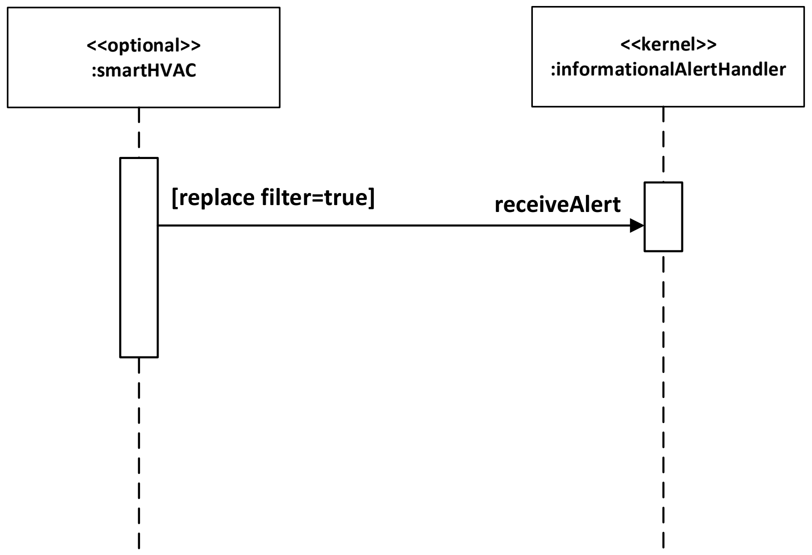


Figure A.16 Sequence Diagram for the Smart Home EU SPL HVAC Filter Feature

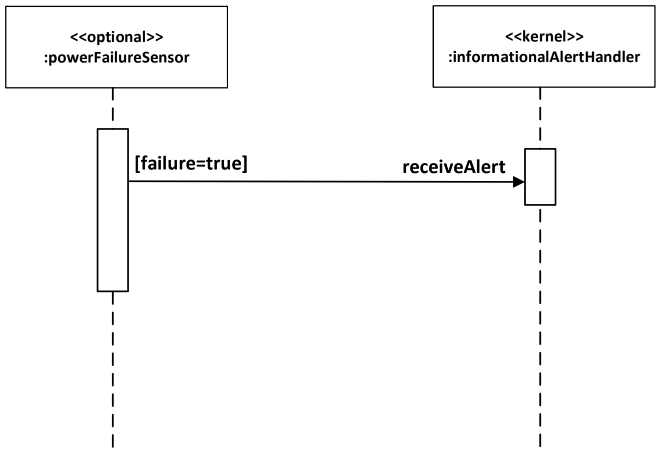


Figure A.17 Sequence Diagram for the Smart Home EU SPL Power Failure Feature

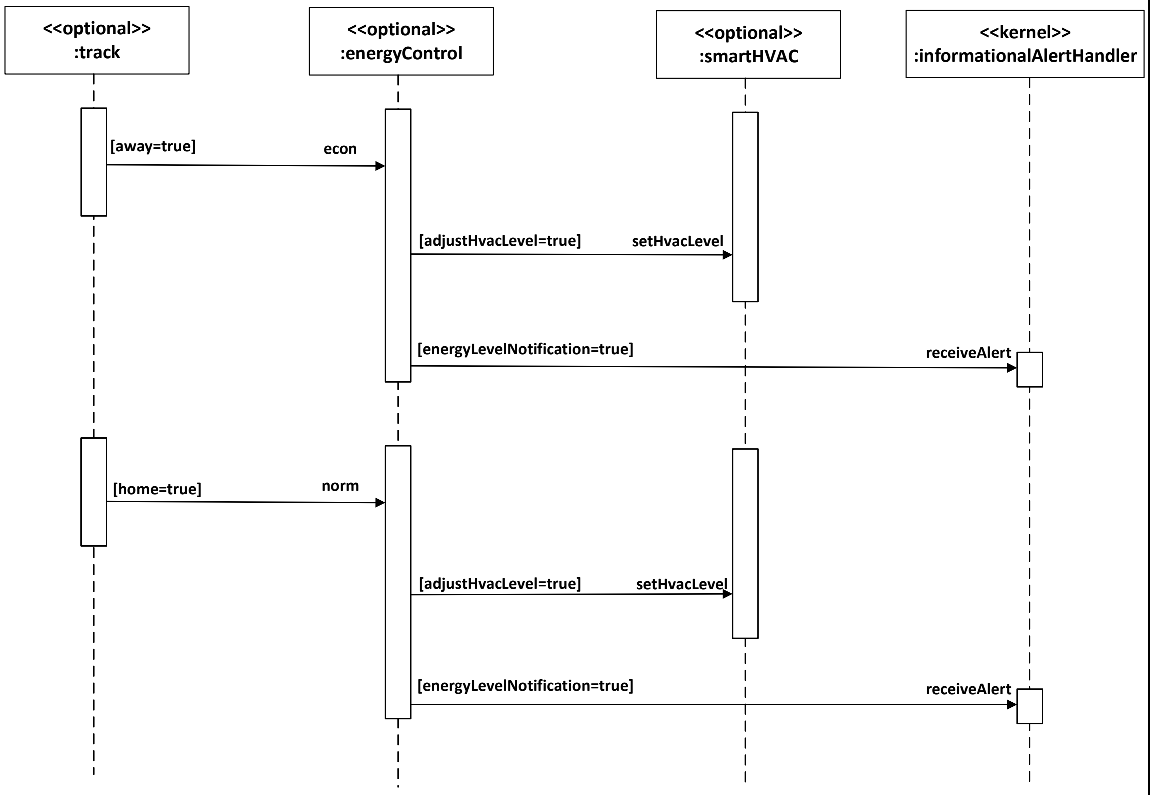


Figure A.18 Sequence Diagram for the Smart Home EU SPL Energy Conservation Feature

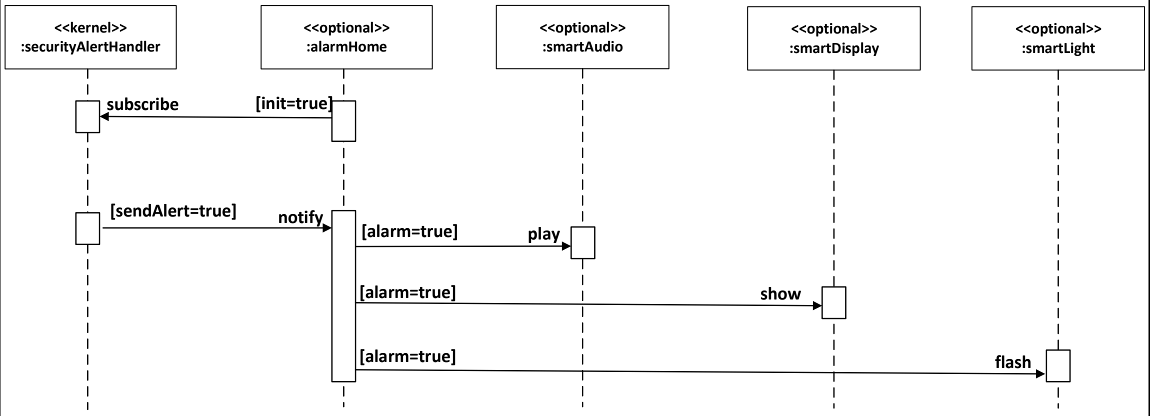


Figure A.19 Sequence Diagram for the Smart Home EU SPL Home Alarm Feature

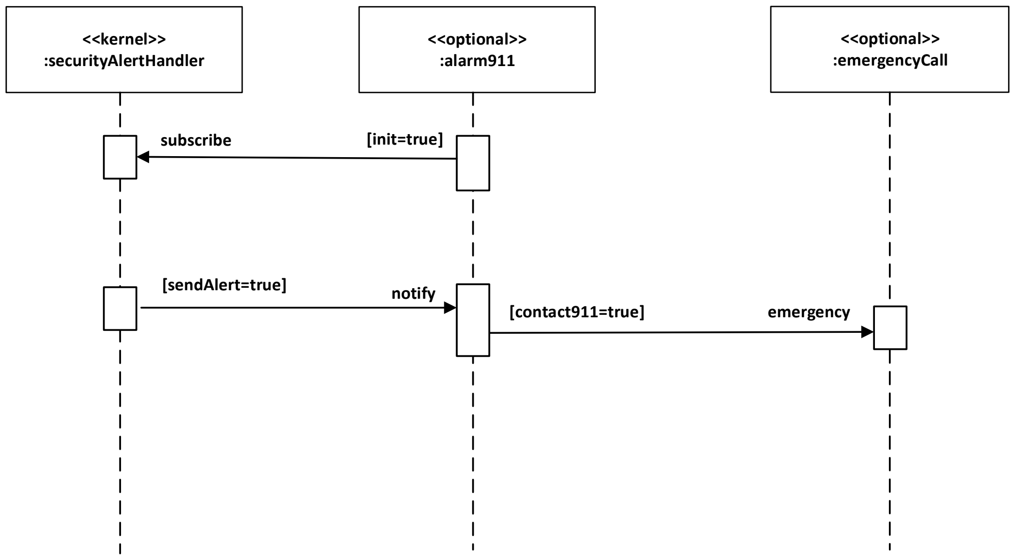


Figure A.20 Sequence Diagram for the Smart Home EU SPL 911 Feature

A.2.2.3 Smart Home EU SPL Feature/Component Dependency Table

The Feature / Component table describes in detail the EU SPL features and the components needed to support the implementation of each of the features. The purpose of the table is for EU SPL designers to ensure consistency between features and the components that support them.

Table A.4 shows the Feature / Component Dependency Table that was developed for the Smart Home EU SPL Case Study used in this research. For example the common feature Smart Home is implemented by the securityAlertHandler and the informationalAlertHandler component that are kernel. Similarly the alternative Video feature is implemented by the alertVideo, videoCall, cameraManager and camera optional components. Since the Video feature depends on the Smart Home feature, the Video feature will also be supported by the securityAlertHandler and informationalAlertHandler kernel components. Finally, the optional Energy Conservation feature is implemented by the optional track and energyControl components. The component parameter residentIDs of the track component indicate the smart home residents that need to be tracked by the component.

Table A.4 Smart Home EU SPL Feature/Component Dependency Table

Feature Name	Feature Group Name	Feature Category	Component Name	Component Reuse Category	Component Parameter
Smart Home		common	securityAlertHandler informationalAlertHandler	kernel kernel	
Audio	Phone Alert	default	alertAudio phone	optional optional	
Video	Phone Alert	alternative	alertVideo videoCall cameraManager camera	optional optional optional optional	
Door	Home Security	default	breakInDoor doorMonitor	optional optional	
Motion	Home Security	optional	breakInMotion motionDetector	optional optional	
Window	Home Security	optional	breakInWindow windowDetector	optional optional	
Smart Irrigation		optional	sprinkler sprinklerControl	optional optional	
Schedule		optional	sprinklerTimer	optional	timetorun : String
Smart Weather Sensing		optional	moistureMonitor	optional	
Email	Net Notification	optional	email	optional	
Text	Net Notification	default	text	optional	
Flood Detector	Water Detector	optional	floodSensor	optional	
Faucet Drip	Water Detector	optional	faucetLeakSensor	optional	
Home Alarm	Home Behavior	optional	alarmHome smartAudio smartDisplay	optional optional optional	
911	Home Behavior	optional	alarm911 emergencyCall	optional optional	
Light Failure	Home Behavior	optional	smartLight	optional	
HVAC Filter	Home Behavior	optional	smartHVAC	optional	
Power Failure	Home Behavior	optional	powerFailureSensor	optional	
Energy Conservation	Home Behavior	optional	track energyControl	optional optional	residentIDs: List<String>

A.2.3 EU SPL Design Modeling

EU SPL Design modeling maps the EU SPL Analysis model to the solution domain (Gomaa, 2016). During EU SPL Design modeling the component inter-feature communication, component relationships and component interface models are designed. Section A.2.3.1 describes the inter-feature component communication table. Section A.2.3.2 presents the component relationships and component interfaces in the form of component diagrams. Section A.2.3.3 provides additional details about the component inputs/outputs and component output triggering conditions that initiate an event.

A.2.3.1 Smart Home EU SPL Inter-Feature Component Communication Table

The inter-feature component communication table captures all product line components that send and receive messages through message broker components. Table A.5 shows the inter-feature component communication table that was created to support the Smart Home case study.

Table A.5 Inter-Feature Component Communication Table

Message Broker	**Subscribed Components**	**Message Producer Components**
securityAlertHandler	alertAudio alertVideo alarmHome alarm911 email text	breakInDoor breakInMotion breakInWindow
informationalAlertHandler	email text	schedule sprinklerControl smartLight smartHVAC powerFailureSensor energyControl floodSensor faucetLeakSensor

A.2.3.2 Smart Home EU SPL Component Diagrams

UML component diagrams can be used by EU SPL designers to capture (a) components available in a smart home, (b) component relationships, and (c) provided and required interfaces needed for components to communicate. The components diagrams are developed based on the sequence diagrams shown in Figure A.21 to Figure A.38 during EU SPL Analysis phase. Figure A.21 to Figure A.38 show the component diagrams developed for the Smart Home EU SPL case study.

Figure A.37 shows the component diagram of the Home Alarm Feature. The component diagram is composed of the securityAlertHandler, alarmHome, smartAudio, smartDisplay and the smartLight components. The components are decorated with UML stereotypes to indicate whether a component is kernel, optional, or variant. For example the securityAlertHandler is a <<kernel>> component while alarmHome, smartAudio, smartDisplay and smartLight are <<optional>> components. Furthermore additional stereotypes are used to capture the role of each component. For example securityAlertHandler is a <<message-broker>> component. Components may also have a multiplicity indicator to indicate the number of component instances in a smart space. For example the smartAudio, component has 1…* multiplicity that indicates that there are one or more smartAudio in the smart space. The connections between components also indicate the required and provided interfaces between components.

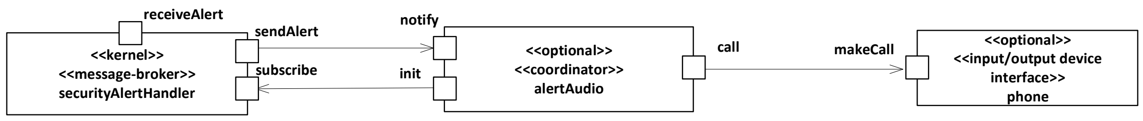


Figure A.21 Component Diagram for the Audio Feature

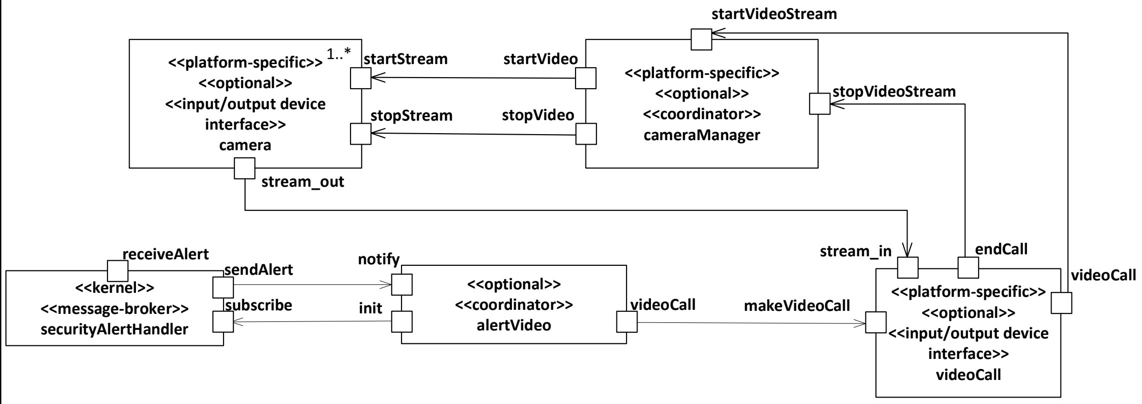


Figure A.22 Component Diagram for the Video Feature

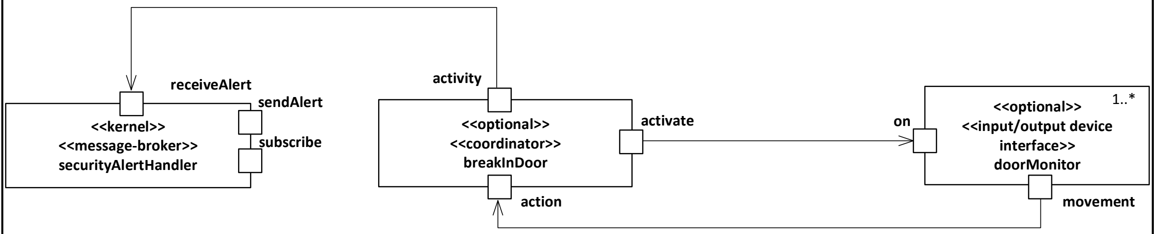


Figure A.23 Component Diagram for the Door Feature

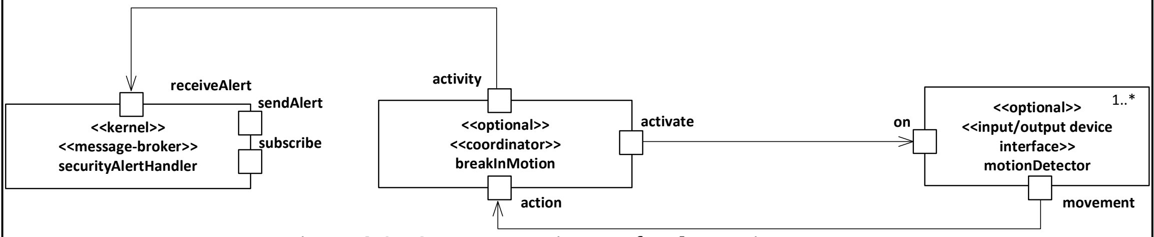


Figure A.24 Component Diagram for the Motion Feature

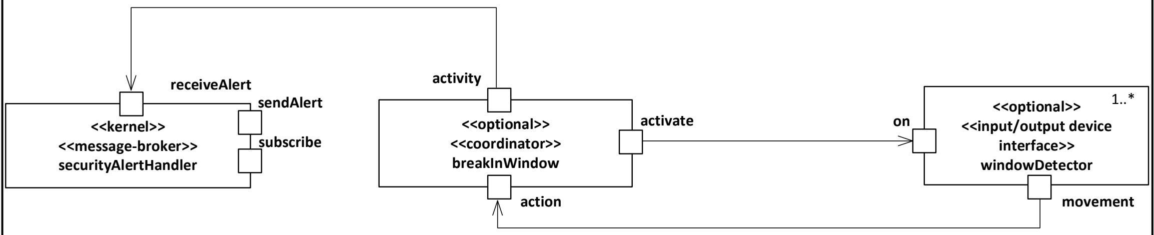


Figure A.25 Component Diagram for the Window Feature

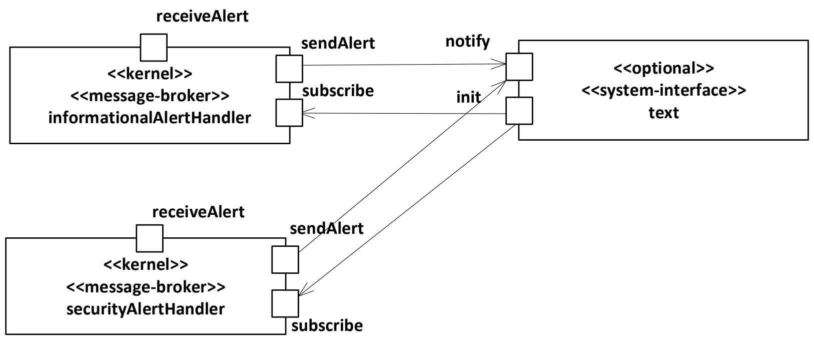


Figure A.26 Component Diagram for the Text Feature

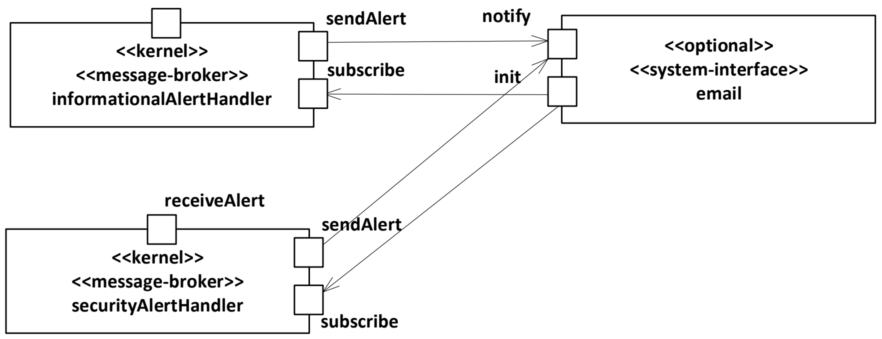


Figure A.27 Component Diagram for the Email Feature

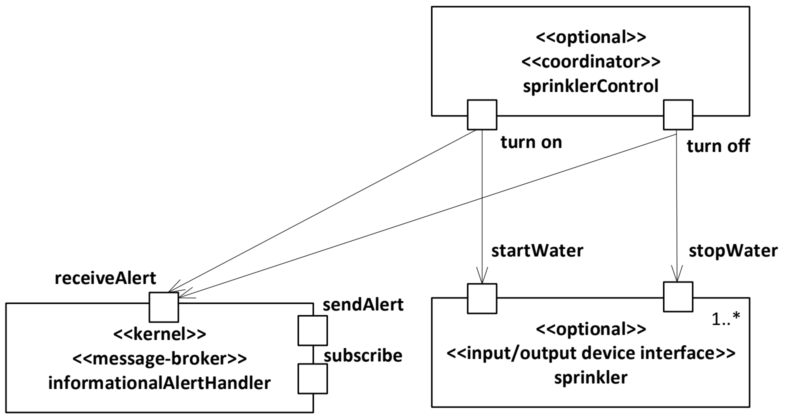


Figure A.28 Component Diagram for the Smart Irrigation Feature

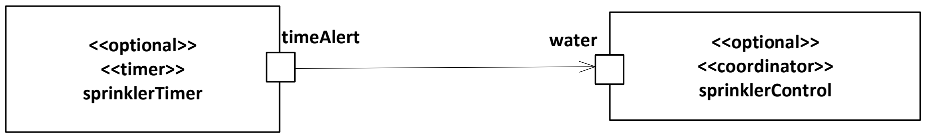


Figure A.29 Component Diagram for the Schedule Feature

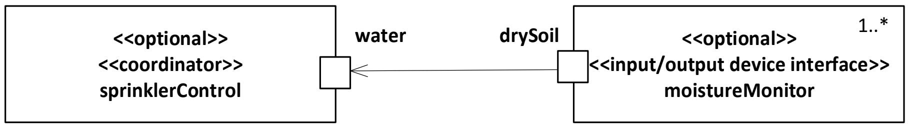


Figure A.30 Component Diagram for the Smart Weather Sensing Feature

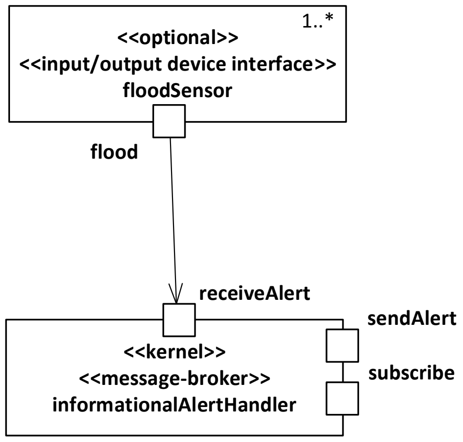


Figure A.31 Component Diagram for the Flood Detector Feature

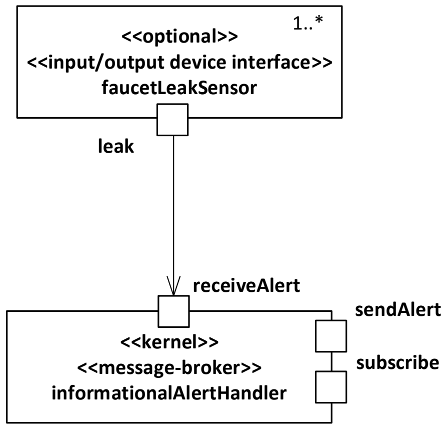


Figure A.32 Component Diagram for the Faucet Drip Feature

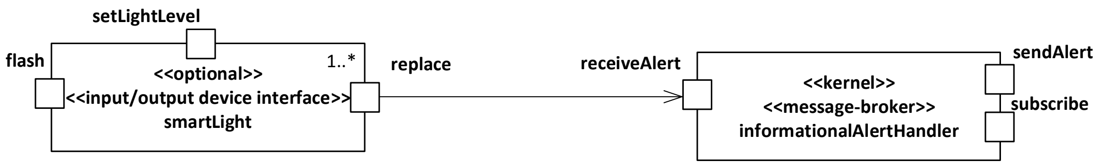


Figure A.33 Component Diagram for the Light Feature

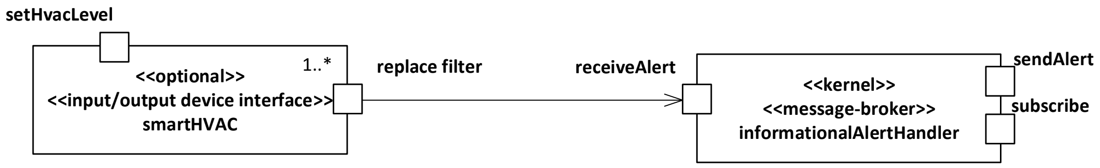


Figure A.34 Component Diagram for the HVAC Filter Feature

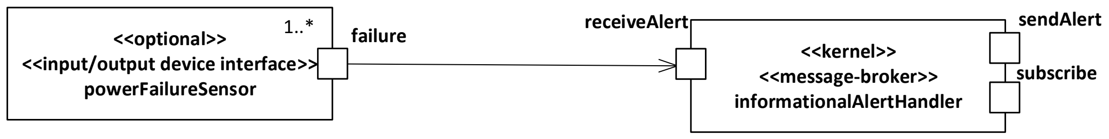


Figure A.35 Component Diagram for the Power Failure Feature

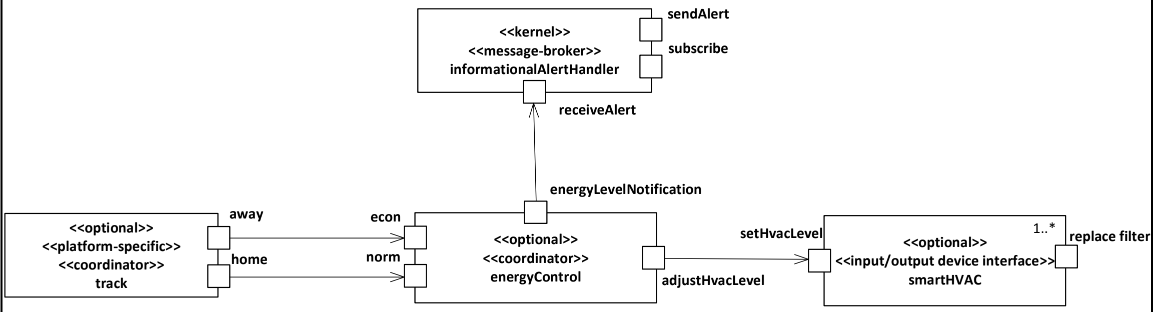


Figure A.36 Component Diagram for the Energy Conservation Feature

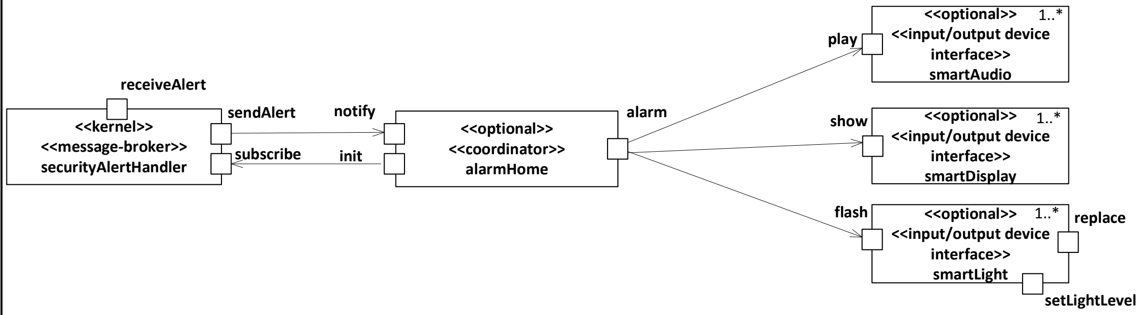


Figure A.37 Component Diagram for the Home Alarm Feature

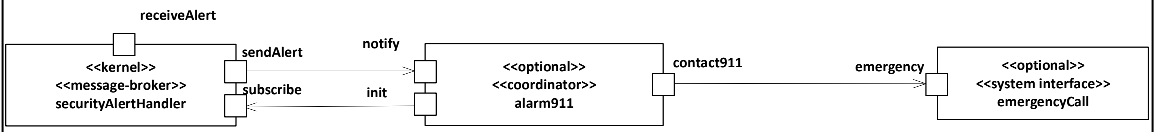


Figure A.38 Component Diagram for the 911 Feature

A.2.3.3 Smart Home EU SPL Component Input / Output Table

Table A.6 and Table A.7 shows component input / output table developed for the **S**mart Home EU SPL. The component input / output table describes all the inputs, outputs and triggering conditions of each component in order to support the features described in the product line. The input / output table has four columns: (1) Component Name, (2) Component Input, (3) Component Input, and (4) Component Output triggering condition. For example the alarm911 has one input called notify that takes as a parameter a message. The alarm911 has two outputs: (1) init, and (2) contact911. The init output sends the component clientID when the triggering condition "startup=true." This indicates that this output executes during initialization. The contact911 output sends a message out when the "message=true" condition is true. The Component Input / Output Table gets mapped to specific platform during application derivation.

Table A.6 Smart Home EU SPL Component Input / Output Table

Component Name	**Component Input**	**Component Output**	**Component Output Triggering Condition**

securityAlertHandler	receiveAlert(in message)
	subscribe(in clientID)
		sendAlert(out message)	messageInQueue=true

informationalAlertHandler	receiveAlert(in message)
	subscribe(in clientID)
		sendAlert(out message)	messageInQueue=true

alertAudio	notify(in message)	init(out clientID)
		call(out phone_number, out message)
			startup=true
			message=true

phone	makeCall(in phone_number, in message)		

alertVideo	notify(in message)	init(out clientID)
		videocall(out phone_number, out message)
			startup=true
			message=true

videoCall	makeVideoCall(in phone_number, in message)
	stream_in(in video_stream)
		videoCall(out client_IP_address)
		endCall(out client_IP_address)
			videoInit=true
			pressedEndCallButton=true

cameraManager	startVideoStream(in client_IP_address)
	stopVideoStream(in client_IP_address)
		startVideo(out client_IP_address)
		stopVideo(out client_IP_address)	

camera	startStream(out client_IP_address)
	stopStream(out client_IP_address)
		stream_out(out video_stream)	

alarmHome	notify(in message)	init(out clientID)
		alarm(out message)
			startup=true
			message=true

smartAudio	play(in message)		
smartDisplay	show(in message)		
smartLight	flash setLightLevel(in lightLevel)	replace(out lightID)	light=out

alarm911	notify(in message)	init(out clientID)
		contact911 (out message)
			startup=true
			message=true

emergencyCall	emergency(in message)		

breakInDoor	action(in deviceID, in deviceType)	activate
		activity(out message)
			startup=true
			motion=true

doorMonitor	on	movement(out deviceID , out deviceType)	move=true

breakInMotion	action(in deviceID, in deviceType)	activate
		activity(out message)
			startup=true
			motion=true

motionDetector	on	movement(out deviceID , out deviceType)	move=true

breakInWindow	action(in deviceID, in deviceType)	activate
		activity(out message)
			startup=true
			motion=true

windowDetector	on	movement(out deviceID , out deviceType)	move=true

sprinkler	startWater(out message)
	stopWater(out message)		

sprinklerTimer		turn on (out message)
		turn off (out message)
			scheduledStartTime=true
			scheduledEndTime=true

sprinklerControl	water	turn on (out message) turn off (out message)	
moistureMonitor		drySoil	dry=true

Table A.7 Component Input/Output Table (Continuation)

Component Name	Component Input	Component Output	Component Output Triggering Condition
email	notify(in message)	init(out clientID)	startup=true
text	notify(in message)	init(out clientID)	startup=true
smartHVAC	setHvacLevel(in temp)	replace filter(out hvacID)	replaceFilter=true
powerFailureSensor		failure(out deviceID)	powerFailure=true

track		away
		home
			residentsAway=true
			residentsHome=true

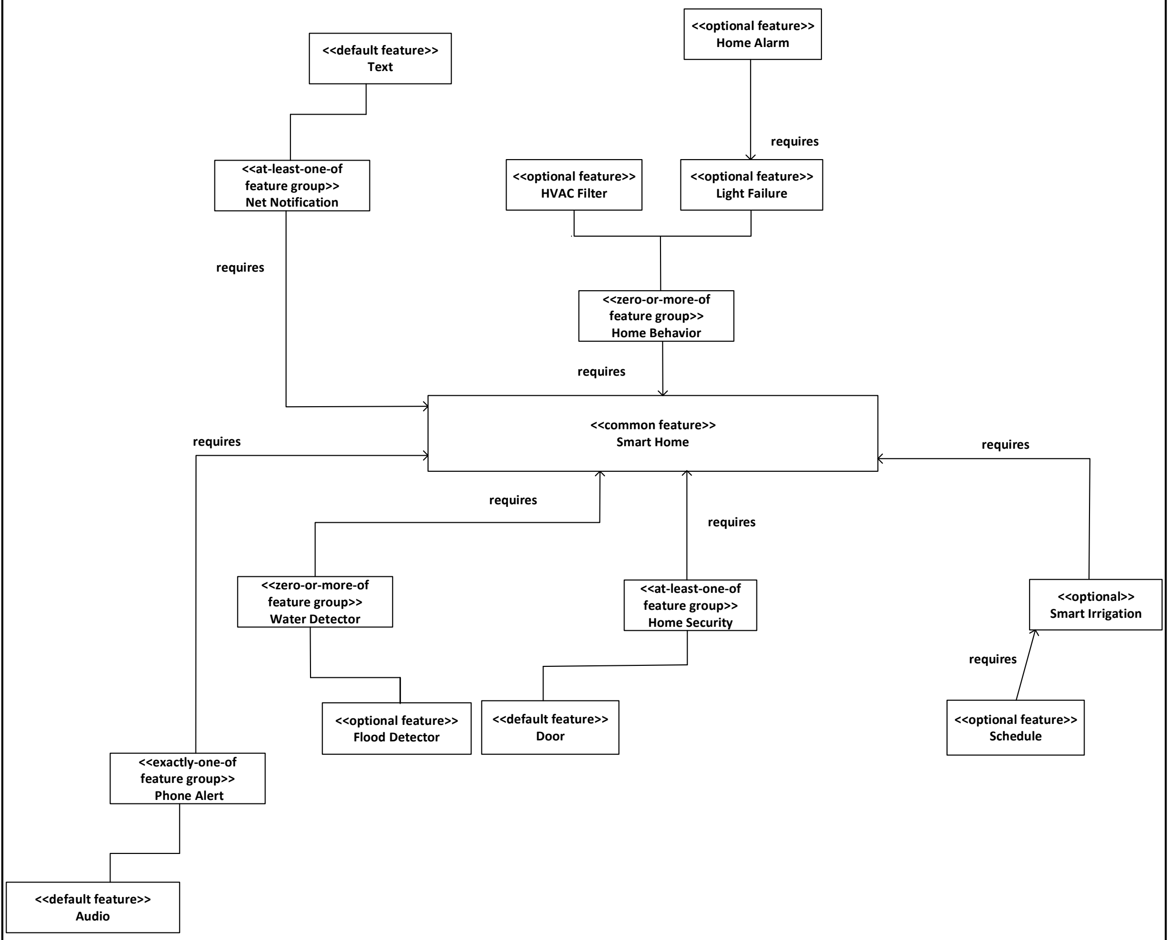
energyControl	econ norm	energyLevelNotification (out message) adjustLightLevel (out lightLevel) adjustHvacLevel (out temp)	econLvl=true OR normLvl=true econLvl=true OR normLvl=true econLvl=true OR normLvl=true
floodSensor		flood(out location)	moisture=true
faucetLeakSensor		leak(out deviceID)	leakDetected=true

A.3 End User Application Engineering

End User Application Engineering (EUAE) is the process to derive end user applications from the End User SPL and deploy them to end user smart spaces. This section, describes two application derivation examples, “Smart Home Example 1” and “Smart Home Example 2” from the Smart Home EU SPL. In particular, the feature selection for the “Smart Home Example 1” does not contain any platform specific features and application derivation examples are given for both the Jigsaw and TeC EUD environment. The “Smart Home Example 2” is platform specific in which the application is derived for the TeC EUD framework. The remainder of this section describes the EUAE process for both examples. Section A.3.1 describes the “Smart Home Example 1” application and section A.3.2 describes the “Smart Home Example 2” application.

A.3.1 Smart Home Example 1 - End User Application Engineering

The “Smart Home Example 1” is an example of an application derived from the Smart Home EU SPL based on the end user requirements. Figure A.39 shows the Feature Model of the derived application. The derived application consists of the following features: “Smart Home”, “Audio”, “Door”, “Text”, “Flood Detector”, “Smart Irrigation”, “Schedule”, “HVAC Filter”, “Home Alarm” and “Light Failure.” The feature model follows the feature and feature group consistency rules. For example there is only one feature selected form the “Phone Alert” exactly-one-of feature group, there is one feature selected from the “Home Security” and “Net Notification” at-least-one-of feature groups. Also all parent features that other features depend on are also available. Some examples of parent features are the “Smart Home” common feature that all other features depend on,



<<optional feature>> Home Alarm
<<default feature>> Text
requires
<<at-least-one-of feature group>> Net Notification
<<optional feature>> HVAC Filter
<<optional feature>> Light Failure
requires
<<zero-or-more-of feature group>> Home Behavior
requires
<<common feature>> Smart Home
requires
requires
requires
requires
<<zero-or-more-of feature group>> Water Detector
<<at-least-one-of feature group>> Home Security
<<optional>> Smart Irrigation
requires
<<optional feature>> Flood Detector
<<default feature>> Door
<<optional feature>> Schedule
<<exactly-one-of feature group>> Phone Alert
<<default feature>> Audio

Figure A.39 Smart Home Example 1 – Feature Model

the “Light Failure” feature that the “Home Alarm” depends on and the “Smart Irrigation” feature that the “Schedule” feature depends on. As shown in Figure A.39 there are not any platform specific features selected, thus the derived application can be deployed to either the Jigsaw or TeC EUD environment. Table A.8 and Figure A.40 describe the deployment of the derived application to the Jigsaw EUD environment. Table A.8 shows the mapping of the “Smart Home Example 1” features to the Jigsaw architecture. In detail Table A.8 has six columns: (1) Feature Name, (2) Feature Group Name, (3) Jigsaw

Component Name, (4) Jigsaw Component Input, (5) Jigsaw Component Output, and (6) Jigsaw Component Output Triggering Condition. For example, the first row of shows that the feature “Smart Home” is implemented by one component the “securityAlertHandler.” The “securityAlertHandler” component contains two inputs: (1) “receiveAlert (in message)”, and (2) “subscribe (in clientID).” The “receiveAlert” input is used to receive security alerts from other components and expects a parameter called “message.” The “subscribe” input is used for other components to register to the “securityAlertHandler.” The “subscribe” input and expects a parameter called “clientID” with the identification name of the component that needs to be registered. The “securityAlertHandler” component contains one output called “sendAlert (out message).” The output send alerts to registered components. The output sends one parameter to registered components called “message” that contain the alert details. The “sendAlert (out message)” output of the “securityAlertHandler” component is executed when the “messageInQueue=true” triggering condition evaluates to true. Figure A.40 visualizes the derived end user application architecture as it would be displayed to the Jigsaw editor. As shown in Figure A.40 components are represented as Jigsaw pieces put together to form application logic. Similarly, Table shows the application mapping for Smart Home derived application to the Team Computing EUD environment based on the feature selections shown in Figure A.41 visualizes the derived application architecture as it would be displayed to the Team Computing application editor.

Table A.8 Smart Home Example 1 – EU Derived Application Mapped to Jigsaw

Feature Name	**Feature Group Name**	**JIGSAW PSP Component Name**	**JIGSAW PSP Component Input**	**JIGSAW PSP Component Output**	**JIGSAW PSP Component Output Triggering Condition**
Smart Home		securityAlertHandler	receiveAlert(in message) subscribe(in clientID)	sendAlert(out message)	messageInQueue=true
Smart Home		informationalAlertHandler	receiveAlert(in message) subscribe(in clientID)	sendAlert(out message)	messageInQueue=true
Light Failure	Home Behavior	smartLight	flash	replace(out lightID)	light=false
HVAC Filter	Home Behavior	smartHVAC		replace filter(out hvacID)	replaceFilter=true
Home Alarm	Home Behavior	alarmHome	notify(in message)	init(out clientID) alarm(out message)	startup=true message=true
Home Alarm	Home Behavior	smartAudio	play(in message)		
Home Alarm	Home Behavior	smartDisplay	show(in message)		
Audio	Phone Alert	alertAudio	notify(in message)	init(out clientID) call(out phone_number, out message)	startup=true message=true
Audio	Phone Alert	phone	makeCall(in phone_number, in message)		
Door	Home Security	breakInDoor	action(in deviceID, in deviceType)	activate activity(out message)	startup=true motion=true
Door	Home Security	doorMonitor	on	movement(out deviceID , out deviceType)	move=true
Flood Detector	Water Detector	floodSensor		flood(out location)	moisture=true
Smart Irrigation		Sprinkler	startWater stopWater		
Smart Irrigation		sprinklerControl	water	turn on (out message) turn off (out message)	
Schedule		sprinklerTimer	turn on (out message) turn off (out message)	scheduledStartTime=true scheduledEndTime=true	
Text	Net Notification	text	notify(in message)	init(out clientID)	startup=true

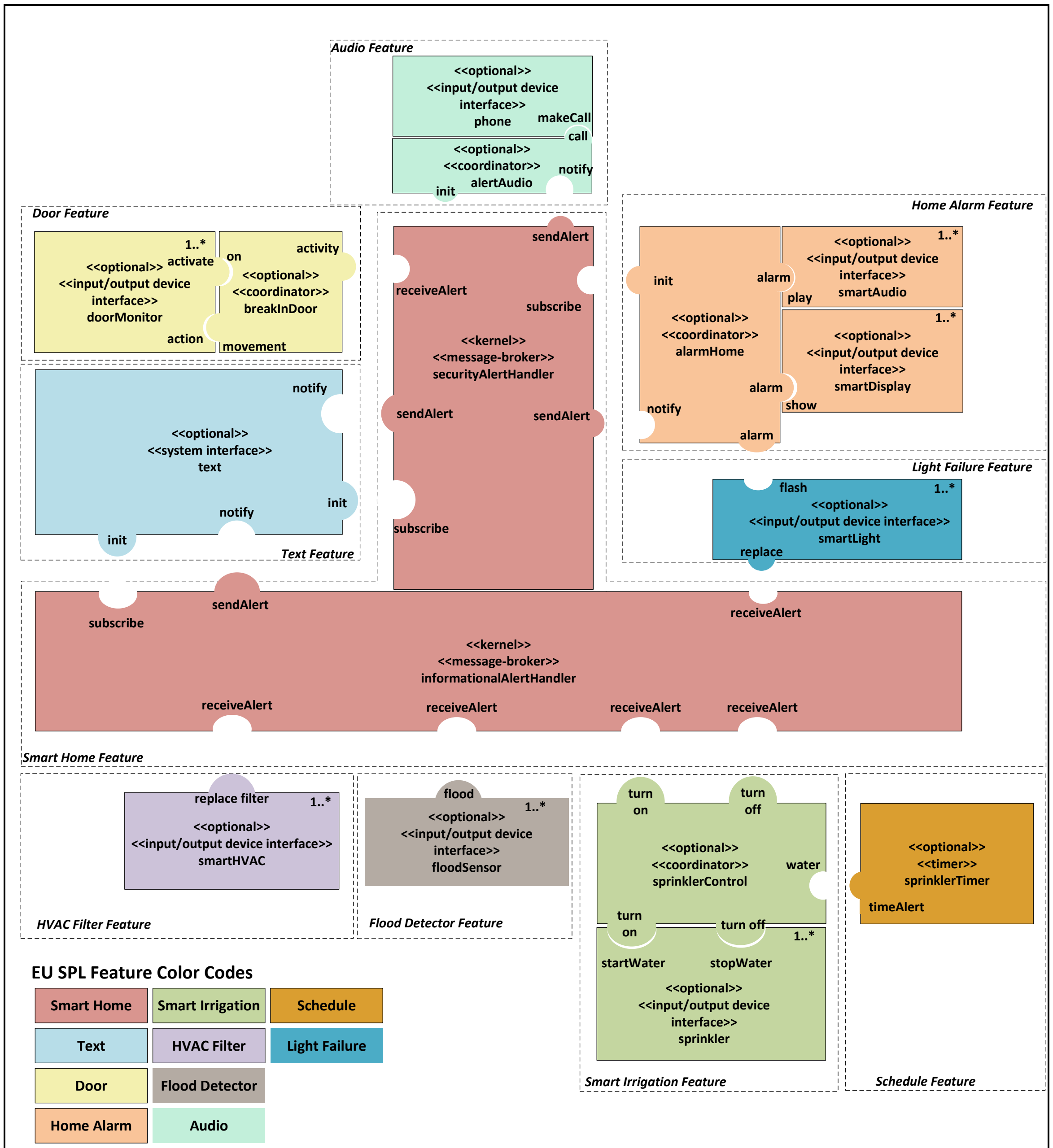

Audio Feature
<<optional>> <<input/output device interface>> phone
makeCall
call
<<optional>> <<coordinator>> alertAudio
notify
init
Door Feature
1..*
<<optional>> <<input/output device interface>> doorMonitor
activate
action
on
activity
<<optional>> <<coordinator>> breakInDoor
movement
Text Feature
notify
<<optional>> <<system interface>> text
notify
init
init
sendAlert
receiveAlert
subscribe
<<kernel>> <<message-broker>> securityAlertHandler
sendAlert
sendAlert
subscribe
Home Alarm Feature
init
alarm
<<optional>> <<coordinator>> alarmHome
alarm
notify
alarm
1..*
<<optional>> <<input/output device interface>> smartAudio
play
1..*
<<optional>> <<input/output device interface>> smartDisplay
show
Light Failure Feature
flash
1..*
<<optional>> <<input/output device interface>> smartLight
replace
sendAlert
subscribe
receiveAlert
<<kernel>> <<message-broker>> informationalAlertHandler
receiveAlert
receiveAlert
receiveAlert
receiveAlert
Smart Home Feature
replace filter
1..*
<<optional>> <<input/output device interface>> smartHVAC
HVAC Filter Feature
flood
1..*
<<optional>> <<input/output device interface>> floodSensor
Flood Detector Feature
turn on
turn off
<<optional>> <<coordinator>> sprinklerControl
water
turn on
turn off
1..*
startWater
stopWater
<<optional>> <<input/output device interface>> sprinkler
Smart Irrigation Feature
<<optional>> <<timer>> sprinklerTimer
timeAlert
Schedule Feature
EU SPL Feature Color Codes
Smart Home
Smart Irrigation
Schedule
Text
HVAC Filter
Light Failure
Door
Flood Detector
Home Alarm
Audio

Figure A.40 Smart Home Example 1 - EU Application Architecture for Jigsaw

Table A.9 Smart Home Example 1 – EU Derived Application Mapped to TeC

Feature Name	**Feature Group Name**	**TeC PSP Component Name**	**TeC PSP Component Input**	**TeC PSP Component Output**	**TeC PSP Component Output Triggering Condition**
Smart Home		securityAlertHandler	receiveAlert(in message) subscribe(in clientID)	sendAlert(out message)	messageInQueue=true
Smart Home		informationalAlertHandler	receiveAlert(in message) subscribe(in clientID)	sendAlert(out message)	messageInQueue=true
Light Failure	Home Behavior	smartLight	flash	replace(out lightID)	light=false
HVAC Filter	Home Behavior	smartHVAC		replace filter(out hvacID)	replaceFilter=true
Home Alarm	Home Behavior	alarmHome	notify(in message)	init(out clientID) alarm(out message)	startup=true message=true
Home Alarm	Home Behavior	smartAudio	play(in message)		
Home Alarm	Home Behavior	smartDisplay	show(in message)		
Audio	Phone Alert	alertAudio	notify(in message)	init(out clientID) call(out phone_number, out message)	startup=true message=true
Audio	Phone Alert	phone	makeCall(in phone_number, in message)		
Door	Home Security	breakInDoor	action(in deviceID, in deviceType)	activate activity(out message)	startup=true motion=true
Door	Home Security	doorMonitor	on	movement(out deviceID , out deviceType)	move=true
Flood Detector	Water Detector	floodSensor		flood(out location)	moisture=true
Smart Irrigation		sprinkler	startWater stopWater		
Smart Irrigation		sprinklerControl	water	turn on (out message) turn off (out message)	
Schedule		sprinklerTimer	turn on (out message) turn off (out message)	scheduledStartTime=true scheduledEndTime=true	
Text	Net Notification	text	notify(in message)	init(out clientID)	startup=true

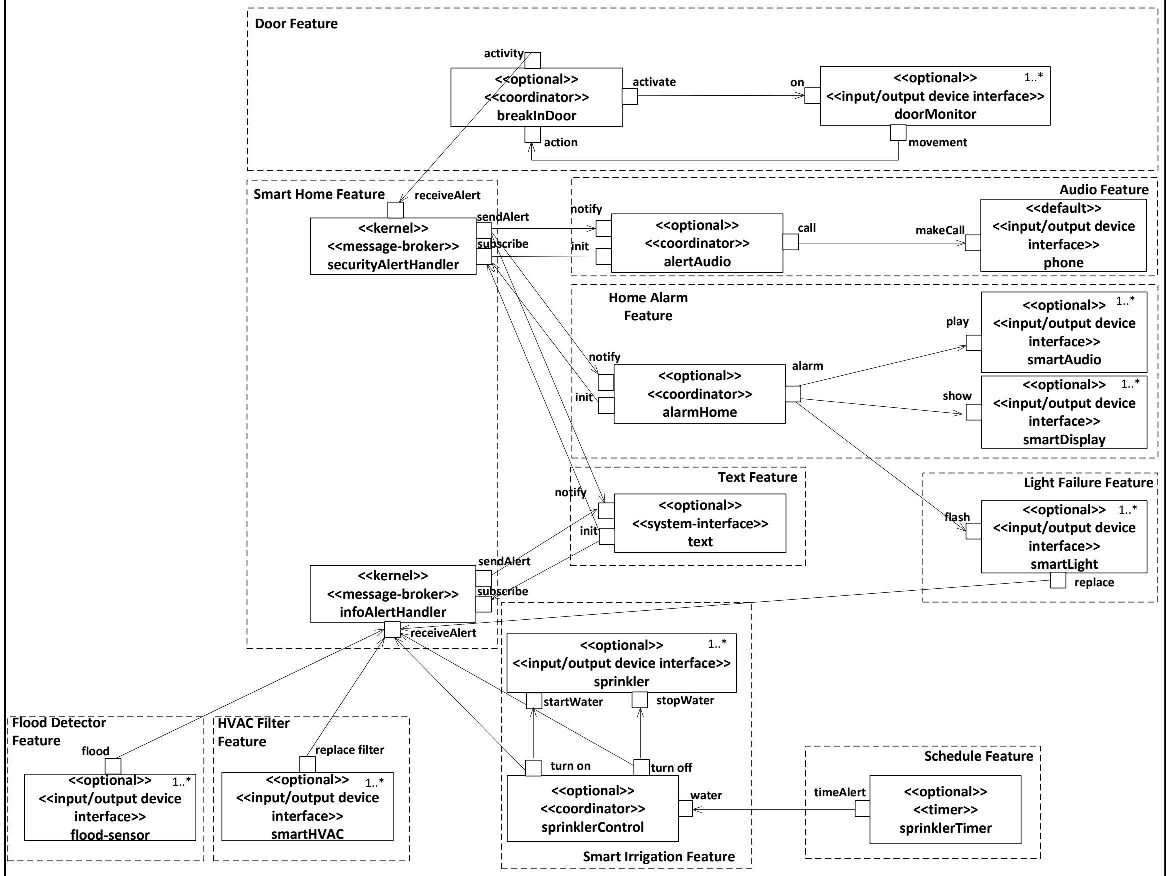


Figure A.41 Example 1 - Smart Home Example 1 - EU Application Architecture for TeC

A.3.2 Smart Home Example 2 - End User Application Engineering

The "Smart Home Example 2" is an example of an application derived from the Smart Home EU SPL based on the end user requirements. Figure A.42 shows the Feature Model of the derived application. The derived application consists of the following features: "Smart Home", "Video", "Door", "Motion", "Window", "Email", "HVAC Filter", "911"and "Energy Conservation." The feature model follows the feature and feature group consistency rules. For example there is only one feature selected form the

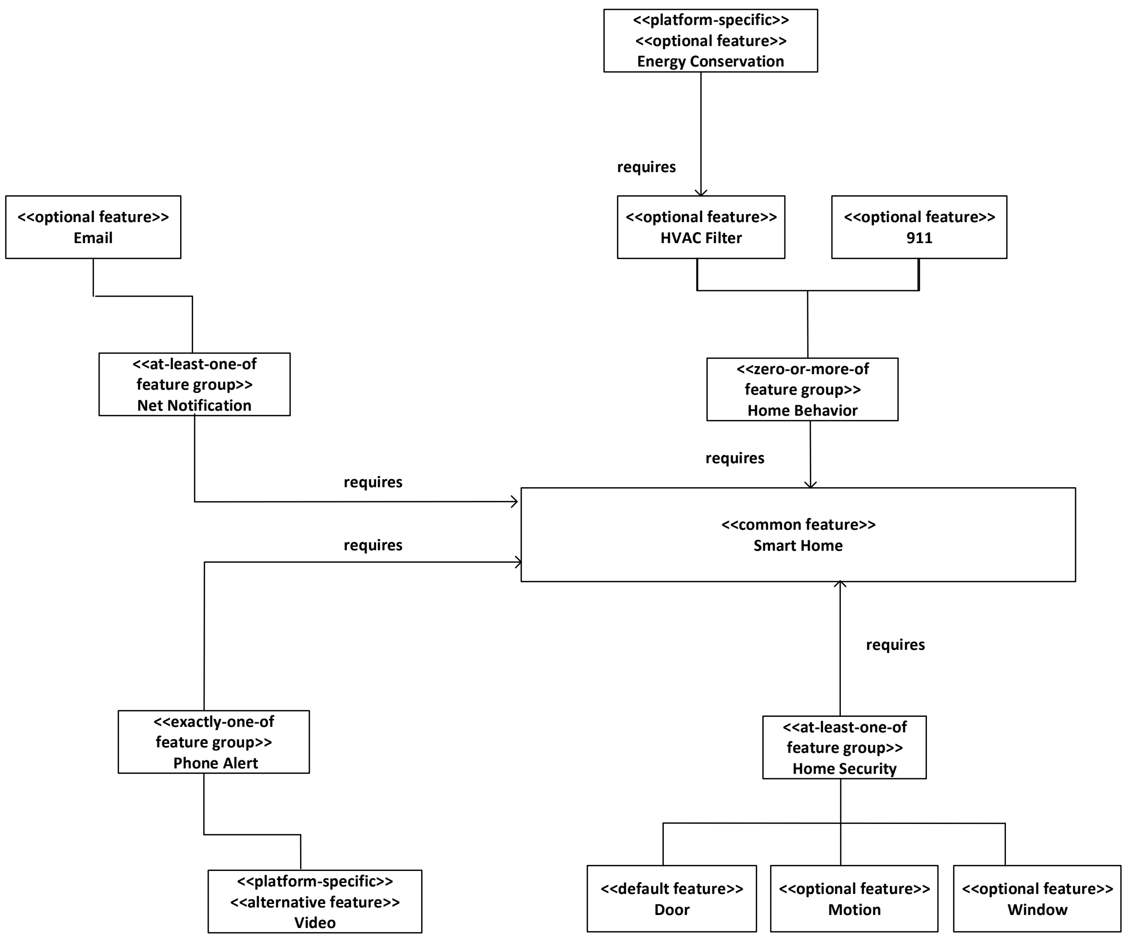


Figure A.42 Smart Home Example 2 – Platform Specific Feature Model

"Phone Alert" exactly-one-of feature group, there is one feature selected from the "Home Security" and "Net Notification" at-least-one-of feature groups. Also all parent features that other features depend on are also available. This feature model contains two platform specific features: (1) "Video", and (2) "Energy Conservation." The features are platform specific to the TeC platform. This means that the features are realized by TeC components. During the application derivation the EU SPL Platform Specific Feature / Component

Table is consulted to get the platform specific components for the platform specific features.

Table A.10 shows the application mapping for Smart Home Example 2 derived application to the Team Computing EUD environment. The components: tecTrack, tecCamera, tecCameraManager, tecVideoCall in Table A.10 are specific only to TeC. Figure A.43 visualizes the derived application architecture as it would be displayed to the Team Computing application editor.

Table A.10 Smart Home Example 2 – Platform Spesific EU Application Mapped to TeC

Feature Name	Feature Group Name	TEC PSP Component Name	TEC PSP Component Input	TEC PSP Component Output	TEC PSP Component Output Triggering Condition

Smart Home		securityAlertHandler	receiveAlert(in message)
			subscribe(in clientID)
				sendAlert(out message)	messageInQueue=tr
					ue

Smart Home		informationalAlertHand
		ler
			receiveAlert(in message)
			subscribe(in clientID)
				sendAlert(out message)	messageInQueue=tr
					ue

Door	Home Security	breakInDoor	action(in deviceID, in deviceType)	activate activity(out message)	startup=true motion=true
Door	Home Security	doorMonitor	on	movement(out deviceID , out deviceType)	move=true
Motion	Home Security	breakInMotion	action(in deviceID, in deviceType)	activate activity(out message)	startup=true motion=true
Motion	Home Security	motionDetector	on	movement(out deviceID , out deviceType)	move=true
Window	Home Security	breakInWindow	action(in deviceID, in deviceType)	activate activity(out message)	startup=true motion=true
Window	Home Security	windowDetector	on	movement(out deviceID , out deviceType)	move=true
911	Home Behavior	alarm_911	notify(in message)	init(out clientID) contact911 (out message)	startup=true message=true
911	Home Behavior	emergency_call	emergency(in message)		

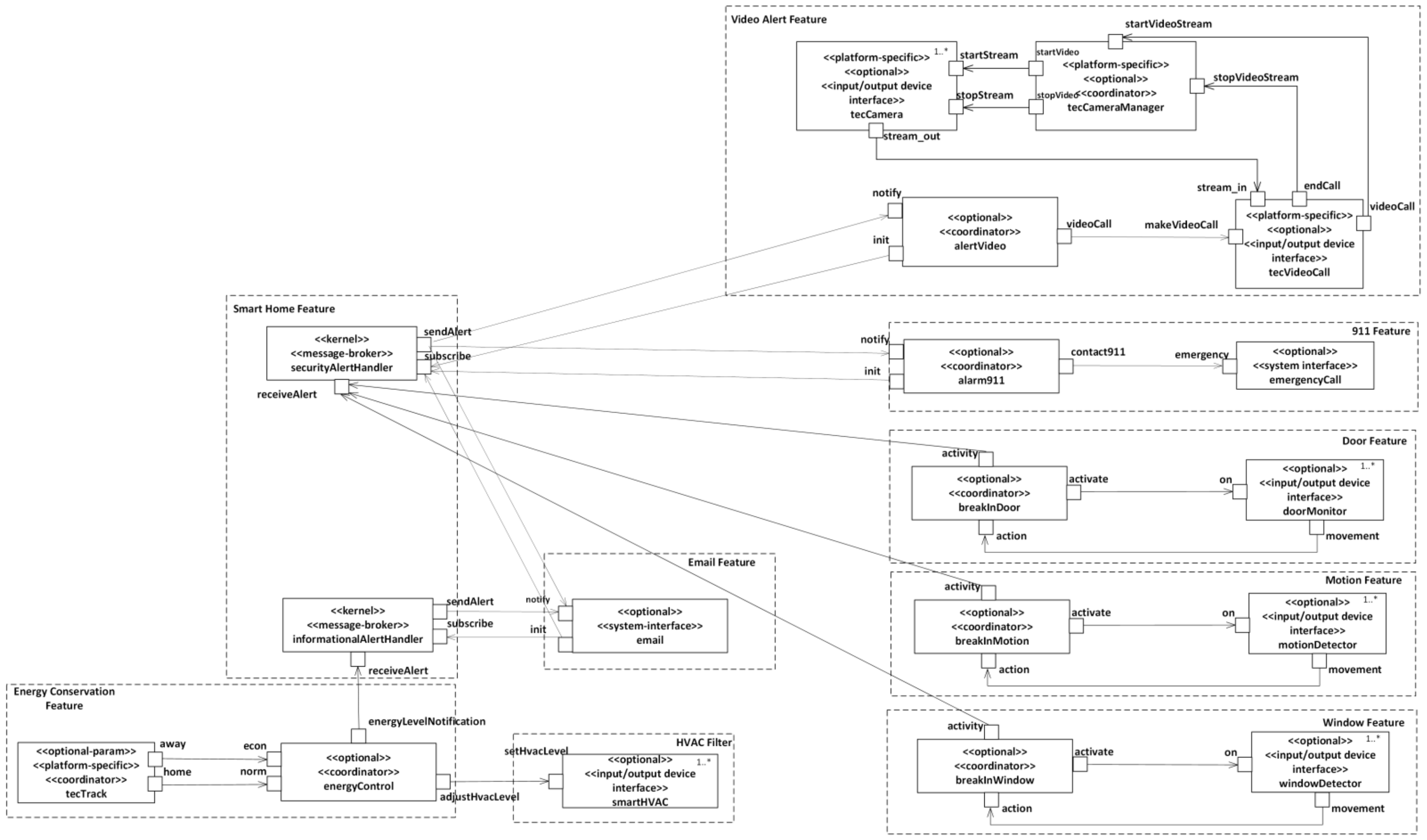
Video	Phone Alert	alertVideo	notify(in message)	init(out clientID)
				videocall(out phone_number, out message)
					startup=true
					message=true

Video	Phone Alert	tecVideoCall	makeVideoCall(in phone_number, in message)
			stream_in(in video_stream)
				videoCall(out client_IP_address)
				endCall(out client_IP_address)
					videoInit=true
					pressedEndCallButt
					on=true

Video	Phone Alert	tecCameraManager	startVideoStream(in client_IP_address)
			stopVideoStream(in client_IP_address)
				startVideo(out client_IP_address)
				stopVideo(out client_IP_address)	

Video	Phone Alert	tecCamera	startStream(out client_IP_address)
			stopStream(out client_IP_address)
				stream_out(out video_stream)	

HVAC Filter	Home Behavior	smartHVAC	setHvacLevel(in temp)	replace filter(out hvacID)	replaceFilter=true
Energy Conserv ation	Home Behavior	tecTrack		away home	residentsAway=true residentsHome=true
Energy Conserv ation	Home Behavior	energyControl	econ norm	energyLevelNotification (out message) adjustLightLevel (out lightLevel) adjustHvacLevel (out temp)	econLvl=true OR normLvl=true econLvl=true OR normLvl=true econLvl=true OR normLvl=true
Email	Net Notification	email	notify(in message)	init(out clientID)	startup=true



Video Alert Feature
<<platform-specific>> <<optional>> <<input/output device interface>> tecCamera
1..*
startStream
stopStream
stream_out
<<platform-specific>> <<optional>> <<coordinator>> tecCameraManager
startVideo
stopVideo
startVideoStream
stopVideoStream
<<optional>> <<coordinator>> alertVideo
notify
init
videoCall
makeVideoCall
<<platform-specific>> <<optional>> <<input/output device interface>> tecVideoCall
stream_in
endCall
videoCall
Smart Home Feature
<<kernel>> <<message-broker>> securityAlertHandler
sendAlert
subscribe
receiveAlert
<<kernel>> <<message-broker>> informationalAlertHandler
sendAlert
subscribe
receiveAlert
911 Feature
<<optional>> <<coordinator>> alarm911
notify
init
contact911
emergency
<<optional>> <<system interface>> emergencyCall
Door Feature
<<optional>> <<coordinator>> breakInDoor
activity
activate
action
on
<<optional>> <<input/output device interface>> doorMonitor
1..*
movement
Motion Feature
<<optional>> <<coordinator>> breakInMotion
activity
activate
action
on
<<optional>> <<input/output device interface>> motionDetector
1..*
movement
Window Feature
<<optional>> <<coordinator>> breakInWindow
activity
activate
action
on
<<optional>> <<input/output device interface>> windowDetector
1..*
movement
Email Feature
<<optional>> <<system-interface>> email
notify
init
Energy Conservation Feature
<<optional-param>> <<platform-specific>> <<coordinator>> tecTrack
away
home
econ
norm
<<optional>> <<coordinator>> energyControl
energyLevelNotification
adjustHvacLevel
HVAC Filter
setHvacLevel
<<optional>> <<input/output device interface>> smartHVAC
1..*

Figure A.43 Smart Home Example 2 – Platform Specific EU Application Architecture for TeC

A.4 Summary

This appendix has described the analysis and design of the Smart Home EU SPL case study that was used in this research. In detail, the chapter described (a) the Smart Home EU SPL requirements that included the Smart Home EU SPL features, feature model and feature group / feature dependencies, (b) the Smart Home EU SPL analysis model that included the EU SPL static model, component structuring, platform dependent component analysis, dynamic modeling through the use of sequence diagrams and features to component relationships, and (c) the Smart Home EU SPL design model that included the EU SPL inter-feature component communication analysis, component relationships, component interfaces and component input / output details. Finally the appendix provided two application derivation examples from the Smart Home EU SPL. The first example of the derivation process was for the Jigsaw and TeC EUD environments. The second example was platform specific and the application derivation process was for the TeC EUD environments.

REFERENCES

Abu-Matar, M. and Gomaa, H. (2013), "An Automated Framework for Variability Management of Service-Oriented Software Product Lines", *Proceedings of the 2013 IEEE 7th International Symposium on Service Oriented System Engineering (SOSE)*, San Francisco Bay, USA, pp. 260–267.

Abu-Matar, M. and Gomaa, H. (2012), "Feature-based Variability Meta-Modeling for Service-Oriented Product Lines", Proceedings of the 2011 Models in Software Engineering, Workshops and Symposia (MoDELS), Springer LNCS 7167, pp. 68-82, 2012

Abu-Matar, Mohammad Ahmad. (2011), "Variability Modeling and Meta-Modeling for Service-Oriented Architectures", *Doctoral Dissertation*, George Mason University.

America, P., Obbink, H., Muller, J. and Ommering, R.V (2000), "COPA: A Component-Oriented Platform Architecting Method for Families of Software Intensive Electronic Products", *Proceedings of the First Conference on Software Product Line Engineering*, Denver, Colorado.

Appert, C., Chapuis, O., Pietriga, E. and Lobo, M.-J. (2015), "Reciprocal Drag-and-Drop", *ACM Transactions on Computer-Human Interaction (TOCHI)*, Vol. 22 No. 6, p. 29:1–29:36.

Atkinson, C. and Muthig, D. (2002), "Component-Based Product-Line Engineering with the UML", *Software Reuse: Methods, Techniques, and Tools*, Vol. 2319, Springer Berlin / Heidelberg, pp. 155–182.

Bardram, J.E. (2005), "The Java Context Awareness Framework (JCAF) – a Service Infrastructure and Programming Framework for Context-aware Applications", *Proceedings of the 3rd International Conference on Pervasive Computing*, Springer-Verlag, Berlin, Heidelberg, pp. 98–115.

Barker, R. (1990), *Case Method: Entity Relationship Modelling*, 1st ed., Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Beckwith, L. and Burnett, M. (2004), "Gender: An important factor in end-user programming environments?", *Proceeding of 2004 IEEE Symposium on Visual Languages - Human Centric Computing*, Rome, Italy, pp. 107–114.

Bendraou, L., Gervals, M., and Blanc X. (2005), "UML4SPM: A UML2.0-Based Metamodel for Software Process Modeling," *Proceedings of the ACM/IEEE 8th*

International Conference on Model Driven Engineering Languages and Systems, Montego Bay, Jamaica, pp. 17–38.

Blackwell, A.F. and Hague, R. (2001), "AutoHAN: An architecture for programming the home", *Proceedings of the 2001 IEEE Symposia on Human-Centric Computing Languages and Environments,* Stresa, Italy, pp. 150–157.

Blanc, X., Ramalho, F. and Robin, J. (2005), "Metamodel Reuse with MOF," *Proceedings of the ACM/IEEE 8th International Conference on Model Driven Engineering Languages and Systems*, Montego Bay, Jamaica, pp. 17–38.

Brinkman, W.P., Haakma, R. and Bouwhuis, D.G. (2008), "Component-Specific Usability Testing", *IEEE Transactions on Systems, Man, and Cybernetics - Part A: Systems and Humans,* Vol. 38 No. 5, pp. 1143–1155.

Brock, J., Gupta, A. and Wielenga, G. (2014), *Java EE and HTML5 Enterprise Application Development*, 1st ed., McGraw-Hill Education Group.

Burnett, M. (2009), "What Is End-User Software Engineering and Why Does It Matter?" *Proceedings of the 2nd International Symposium on End-User Development (IS-EUD)*, Siegen, Germany, pp. 15–28.

Burnett, M. and Scaffidi, C. (2014), *End-User Development. In "The Encyclopedia of Human-Computer Interaction, 2nd Ed." Aarhus, Denmark: The Interaction Design Foundation. Available Online at https://www.interaction-Design.org/Encyclopedia/End-User_development.html*.

Burnett, M.M. and Myers, B.A. (2014), "Future of End-user Software Engineering: Beyond the Silos", *Proceedings of the 2014 of the Future of Software Engineering (FOSE)*, Hyderabad, India, pp. 201–211.

Canfora, G., Mercaldo, F., Visaggio, C.A., DAngelo, M., Furno, A. and Manganelli, C. (2013), "A Case Study of Automating User Experience-Oriented Performance Testing on Smartphones", *Proceedings of the 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation (ICST),* Luxembourg, Luxembourg, pp. 66–69.

Chin, J., Callaghan, V. and Clarke, G. (2010), "End-user Customization of Intelligent Environments", *Handbook of Ambient Intelligence and Smart Environments*, Springer US, Boston, MA, pp. 371–407.

Clements, P. and Northrop, L.M. (2002), *Software Product Lines: Practices and Patterns*, Addison-Wesley.

Danado, J. and Paternò, F. (2012), "Puzzle: a visual-based environment for end user development in touch-based mobile phones", *Human-Centered Software Engineering,* Springer, pp. 199–216.

Dautriche, R., Lenoir, C., Demeure, A. and Coutaz, J. (2013), "End-User-Development for Smart Homes: Relevance and Challenges", *Proceedings of the 2013 Workshop "EUD for Supporting Sustainability in Maker Communities", 4th International Symposium on End-user Development (IS-EUD),* Eindhoven, Nederland, p. 6.

Debnath, N., Leonardi, M.C., Mauco, M.V., Montejano, G. and Riesco, D. (2008), "Improving Model Driven Architecture with Requirements Models", *Proceedings of the 5th International Conference on Information Technology: New Generations (ITNG),* Las Vegas, Nevada, USA, pp. 21–26.

Dey, A.K., Hamid, R., Beckmann, C., Li, I. and Hsu, D. (2004), "a CAPpella: programming by demonstration of context-aware applications", *Proceedings of the 2004 Special Interest Group on Computer-Human Interaction Conference on Human Factors in Computing Systems,* Vienna, Austria, pp. 33–40.

Dey, A.K., Sohn, T., Streng, S. and Kodama, J. (2006), "iCAP: Interactive prototyping of context-aware applications", *Pervasive Computing,* Springer, pp. 254–271.

Dimitris Kalofonos and Franklin Reynolds. (2006), "Task-Driven End-User Programming of Smart Spaces Using Mobile Devices", *NRC-TR-2006-001,* Technical Report, Nokia.

Duckett, J. (2014), *JavaScript and JQuery: Interactive Front-End Web Development,* 1st ed., Wiley Publishing.

Ebling, M.R. (2016), "Pervasive Computing and the Internet of Things", *IEEE Pervasive Computing,* Vol. 15 No. 1, pp. 2–4.

Fortino, G., Guerrieri, A., Lacopo, M., Lucia, M. and Russo, W. (2013), "An Agent-Based Middleware for Cooperating Smart Objects", *Proceedings of the 2013 Highlights on Practical Applications of Agents and Multi-Agent Systems: International Workshops of PAAMS, Salamanca, Spain,* pp. 387–398.

Fortino, G. and Trunfio, P. (2014), *Internet of Things Based on Smart Objects: Technology, Middleware and Applications,* Springer.

Gomaa, H. (2000), *Designing Concurrent, Distributed, and Real-Time Applications with UML,* Addison-Wesley Professional.

Gomaa, H. (2005a), *Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures*, Addison-Wesley Professional.

Gomaa, H. (2005b), "Software Product Line Engineering for Web Services and UML", *The 3rd ACS/IEEE International Conference on Computer Systems and Applications*, Cairo, Egypt, pp. 110–114.

Gomaa, H. (2016), *Real-Time Software Design for Embedded Systems*, Cambridge.

Gomaa, H. and Shin, M.E. (2008), "Multiple-View Modeling and Meta-Modeling of Software Product Lines", *Journal of IET Software, Volume 2, Issue 2,* Pages 94-122.

Goumopoulos, C. and Kameas, A. (2009), "Smart objects as components of ubiquitous computing applications", *International Journal of Multimedia and Ubiquitous Engineering, Special Issue on Smart Object Systems*, Vol. 4(3), SERSC Press, pp. 1–20.

Harsu, M. (2002), "FAST product-line architecture process", *Technical Report*, Institute of Software Systems, Tampere University of Technology.

Haugen, Ø., Wąsowski, A. and Czarnecki, K. (2013), "CVL: Common Variability Language", *Proceedings of the 17th International Software Product Line Conference*, ACM, New York, NY, USA, pp. 277–277.

Henricksen, K., Indulska, J. and Rakotonirainy, A. (2001), "Infrastructure for Pervasive Computing: Challenges", *Proceedings of the 2001 Workshop on Pervasive Computing Informatik,* Vienna, Austria, pp. 214–222.

Humble, J., Crabtree, A., Hemmings, T., Akesson, K.P., Koleva, B., Rodden, T. and Hansson, P. (2003), "Playing with the Bits User-Configuration of Ubiquitous Domestic Environments", *Proceedings of the 5th International Conference in Ubiquitous Computing*, Springer LNCS, Seattle, WA, USA, pp. 256–263.

ISO/IEC 18092. (2013), *Information Technology – Telecommunications and Information Exchange between Systems – Near Field Communication–Interface and Protocol (NFCIP-1)*, Standard No. ISO/IEC 18092:2013, International Organization for Standardization, Geneva, CH.

ISO/IEC 26550:2016. (2016), *Software and Systems Engineering – Reference Model for Product Line Engineering and Management*, ISO No. ISO/IEC 26550, International Organization for Standardization, Geneva, Switzerland.

Jani Suomalainen and Pasi Hyttinen. (2011), “Security Solutions for Smart Spaces”, *Proceedings of the 11th IEEE/IPSJ International Symposium on Applications and the Internet (SAINT),* Munich, Germany, pp. 297–302.

Ji, Y. and Xia, L. (2016), “Improved Chameleon: A Lightweight Method for Identity Verification in Near Field Communication”, *Proceedings of the International Symposium on Computer, Consumer and Control (IS3C),* Xi’an, China, pp. 387–392.

Kakola, T. and Leitner, A. (2014), “Introduction to Software Product Lines: Engineering, Services, and Management Minitrack”, *Proceedings of 47th Hawaii International Conference on System Sciences (HICSS),* Hawaii, USA, pp. 5048–5048.

Kang, K.C., Lee, J. and Donohoe, P. (2002), “Feature-oriented product line engineering”, *Software, IEEE,* Vol. 19 No. 4, pp. 58–65.

Kawsar, F., Nakajima, T. and Fujinami, K. (2008), “Deploy Spontaneously: Supporting End-Users in Building and Enhancing a Smart Home”, *Proceedings of the 10th International Conference in Ubiquitous Computing*, Seoul, South Korea, pp. 282–291.

Kleppe, A. (2008), *Software Language Engineering: Creating Domain-Specific Languages Using Metamodels*, 1st ed., Addison-Wesley Professional.

Ko, A.J., Myers, B., Rosson, M.B., Rothermel, G., Shaw, M., Wiedenbeck, S., Abraham, R., et al. (2011), “The State of the Art in End-User Software Engineering”, *ACM Computing Surveys*, Vol. 43 No. 3, pp. 1–44.

Kopetz, H. (2011), *Real-Time Systems: Design Principles for Distributed Embedded Applications*, 2nd ed., Springer Publishing Company, Incorporated.

Lee, J., Garduño, L., Walker, E. and Burleson, W. (2013), “A Tangible Programming Tool for Creation of Context-Aware Applications”, *Proceedings of the 2013 International Joint Conference on Pervasive and Ubiquitous Computing,* Zurich, Switzerland, ACM Press, p. 391.

Lieberman, H., Paternò, F., Klann, M. and Wulf, V. (2006), “End-User Development: An Emerging Paradigm”, in Lieberman, H., Paternò, F. and Wulf, V. (Eds.), *End User Development*, Vol. 9, Springer Netherlands, pp. 1–8.

Livingston, D. (2002), *Advanced Javascript,* Prentice Hall PTR, Upper Saddle River, NJ, USA.

Máca, P. (2016), “Editablegrid”, *JavaScript Library*, available at: http://www.editablegrid.net/en/.

Mahmoud, R., Yousuf, T., Aloul, F. and Zualkernan, I. (2015), “Internet of things (IoT) security: Current status, challenges and prospective measures”, *Proceedings of the 10th International Conference for Internet Technology and Secured Transactions (ICITST)*, London, UK, pp. 336–341.

Malaer, A. and Lampe, M. (2008), “SimPL: A Simple Software Production Line for End User Development”, *Proceedings of the 15th Asia-Pacific Software Engineering Conference*, Beijing, China, pp. 179–186.

Marinho, F.G., Andrade, R.M.C., Werner, C., Viana, W., Maia, M.E.F., Rocha, L.S., Teixeira, E., Filho, J.B., Dantas, V., Lima, F., Aguiar, S., (2013), “MobiLine: A Nested Software Product Line for the Domain of Mobile and Context-aware Applications”, Science of Computer Programming, Vol. 78 No. 12, pp. 2381–2398.

Mavrommati, I., Kameas, A. and Markopoulos, P. (2004), “An Editing Tool That Manages Device Associations in an In-home Environment”, *Personal Ubiquitous Computing*, Vol. 8 No. 3–4, pp. 255–263.

Messer, A., Kunjithapatham, A., Sheshagiri, M., Song, H., Kumar, P., Nguyen, P. and Yi, K. (2006), “InterPlay: A Middleware for Seamless Device Integration and Task Orchestration in a Networked Home”, *Proceedings of the 4rth Annual IEEE International Conference on Pervasive Computing and Communications (PERCOM)*, Pisa, Italy, pp. 298–307.

Mühlhäuser, M. (2008), “Smart Products: An Introduction”, *Constructing Ambient Intelligence: Workshops Darmstadt, Germany (AMI), Revised Papers*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 158–164.

Myers, B.A. (1990a), “Creating User Interfaces Using Programming by Example, Visual Programming, and Constraints”, *ACM Transactions on Programming Languages and Systems*, Vol. 12 No. 2, pp. 143–177.

Myers, B.A. (1990b), “Taxonomies of Visual Programming and Program Visualization”, *Journal of Visual Languages and Computing*, Vol. 1 No. 1, pp. 97–123.

Olimpiew, E.M. (2008), “*Modeling-Based Testing For Software Product Lines*”, *Doctoral Dissertation,* George Mason University.

Olimpiew, E.M. and Gomaa, H. (2009), “Reusable Model-Based Testing”, *Formal Foundations of Reuse and Domain Engineering: Proceedings of the 11th*

International Conference on Software Reuse (ICSR), Falls Church, VA, USA, pp. 76–85.

OMG. (2003), *MDA Guide Version 1.0.1*, edited by Miller, J. and Mukerji, J., available at: http://www.omg.org/cgi-bin/doc?omg/03-06-01.pdf.

Pereira, D. and Loyola, L. (2012), “Inferring User Context from Spatio-Temporal Pattern Mining for Mobile Application Services”, Proceedings of the 2012 *IEEE/WIC/ACM International Conferences on Web Intelligence and Intelligent Agent Technology (WI-IAT),* Macau, China, Vol. 2, pp. 450–457.

Pérez, F., Cetina, C., Valderas, P. and Fons, J. (2009), “Towards End-User Development of Smart Homes by means of Variability Engineering”, *Proceedings of the 3rd International Workshop on Variability Modelling of Software-Intensive Systems*, Seville, Spain, pp. 103–110.

Perez, F. and Valderas, P. (2009), “Allowing End-Users to Actively Participate within the Elicitation of Pervasive System Requirements through Immediate Visualization”, *Proceedings of the 4th International Workshop on Requirements Engineering Visualization*, Atlanta, Georgia, USA, pp. 31–40.

Pérez, F. and Valderas, P. (2009), “A Tool-supported Natural Requirements Elicitation Technique for Pervasive Systems centered on End-users”, *Proceedings of the 14th Jornadas de Ingeniería del Software y Bases de Datos (JISBD),* San Sebastián, Spain, pp. 115–120.

Pilgrim, M. (2010), *HTML5: Up and Running*, 1st ed., O’Reilly Media, Inc.

Prähofer, H., Hurnaus, D., Schatz, R. and Wirth, C. (2008), “Software support for building end-user programming environments in the automation domain”, *Proceedings of the 4th International Workshop on End-User Software Engineering*, Leipzig, Germany, pp. 76–80.

Quigley, M., Conley, K., Gerkey, B.P., Faust, J., Foote, T., Leibs, J., Wheeler, R., et al. (2009), “ROS: an open-source Robot Operating System”, *Proceedings of the 2009 International Conference in Robotics and Automation (ICRA) Workshop on Open Source Software*, Kobe, Japan

Rashidi, P. and Cook, D.J. (2009), “Keeping the Resident in the Loop: Adapting the Smart Home to the User”, *Journal of IEEE Transactions on Systems, Man, and Cybernetics Part A,* Vol. 39 No. 5, pp. 949–959.

Reinhartz-Berger, I., Figl, K. and Haugen, Ø. (2014), “Comprehending feature *models* expressed in CVL”, *Proceedings of the 17th International Conference Model-*

Driven Engineering Languages and Systems (MODELS), Valencia, Spain, pp. 501–517.

Richardson, L. and Ruby, S. (2007), *Restful Web Services, 1st Edition,* O'Reilly.

Rumbaugh, J., Jacobson, I. and Booch, G. (2004), *The Unified Modeling Language Reference Manual, 2nd Edition,* Pearson Higher Education.

Saha, D. and Mukherjee, A. (2003), "Pervasive computing: a paradigm for the 21st century", *Computer,* Vol. 36 No. 3, pp. 25–31.

Satyanarayanan, M. (2001), "Pervasive computing: Vision and challenges", *IEEE Personal Communications,* Vol. 8 No. 4, pp. 10–17.

Schneiderman, R. (2015), "Internet of Things/M2M - (Standards) Work in Progress", *Modern Standardization: Case Studies at the Crossroads of Technology, Economics, and Politics,* Wiley-IEEE Standards Association.

Shen, X. (2014), "A Team Computing Implementation on the Android Platform", *Engineering Thesis,* George Mason University.

Singh, R., Bhargava, P. and Kain, S. (2006), "State of the art smart spaces: application models and software infrastructure", *ACM Ubiquity*, September, Vol. 2006 No. September, p. 7:2–7:9.

Singh, Y. and Sood, M. (2009), "Model Driven Architecture: A Perspective", *Proceedings of the 2009 IEEE International Advance Computing Conference, (IACC)* Patiala, India, pp. 1644–1652.

Sousa, J.P. (2010), "Foundations of Team Computing: Enabling End Users to Assemble Software for Ubiquitous Computing", *Proceedings of the* 2010 *International Conference on Complex, Intelligent and Software Intensive Systems* (CISIS), Krakow, Poland, pp. 9–16.

Sousa, J.P. and Garlan, D. (2002), "Aura: an architectural framework for user mobility in ubiquitous computing environments", *Software Architecture: System Design, Development and Maintenance: IFIP 17th World Computer Congress--TC2 Stream/3rd Working IEEE/IFIP Conference on Software Architecture (WICSA3), Montréal, Québec, Canada,* Kluwer Academic Publishers, p. 29.

Sousa, J.P., Shen, X., Tzeremes, V. and Hodum, F. (2012), "TeC apps for smart spaces: simple, decentralized, resilient, and self-healing", *Proceedings of the 2012 ACM Conference on Ubiquitous Computing, Pittsburgh, PA, USA,* pp. 637–638.

Sousa, J.P., Tzeremes, V. and El Masri, A. (2010), “Space-aware TeC: End-user Development of Safety and Control Systems for Smart Spaces”, *Proceedings of 2010 Systems Man and Cybernetics, IEEE International Conference on*, Istanbul, Turkey, pp. 2914–2921.

Taylor, M. (2014), *Introduction to JavaScript Object Notation: A To-the-Point Guide to JSON*, CreateSpace Independent Publishing Platform, USA.

Totty, B., Gourley, D., Sayer, M., Aggarwal, A. and Reddy, S. (2002), *HTTP: The Definitive Guide*, O’Reilly & Associates, Inc., Sebastopol, CA, USA.

Truong, K.N., Huang, E.M. and Abowd, G.D. (2004), “CAMP: A Magnetic Poetry Interface for End-User Programming of Capture Applications for the Home”, *Proceedings of the 6th International Conference in Ubiquitous Computing*, Nottingham, UK, pp. 143–160.

Tzeremes V (2015), “End User Software Product Line Support for Smart Spaces” *In: Doctoral Symposium, International Conference on Software Reuse (ICSR)*, Miami, USA

Tzeremes, V. and Gomaa, H. (2015), “A Software Product Line Approach for End User Development of Smart Spaces”, *Proceedings of the 5th International Workshop on Product LinE Approaches in Software Engineering (PLEASE)*, IEEE Press, Piscataway, NJ, USA, pp. 23–26.

Tzeremes, V. and Gomaa, H. (2016a), “A Multi-platform End User Software Product Line Meta-model for Smart Environments”, *Proceedings of the 11th International Joint Conference on Software Technologies (ICSOFT) - Volume 1: ICSOFT-EA, Lisbon, Portugal*, pp. 290–297.

Tzeremes, V. and Gomaa, H. (2016b), “XANA: An End User Software Product Line Framework for Smart Spaces”, *Proceedings of 49th Hawaii International Conference on System Sciences (HICSS)*, Hawaii, USA, pp. 5831–5840.

Want, R., Hopper, A., Falcao, V. and Gibbons, J. (1992), “The Active Badge Location System”, *ACM Transactions on Information Systems*, Vol. 10 No. 1, pp. 91–102.

Weiser, M. (1991), “The Computer for the 21st Century”, *Scientific American (Special Issue: Communications, Computers, and Networks)*, Vol. 265 No. 3, pp. 94–104.

Whitmore, A., Agarwal, A. and Xu, L. (2015), “The Internet of Things–A Survey of Topics and Trends”, *Information Systems Frontiers*, Vol. 17 No. 2, pp. 261–274.

Yun, J., Choi, S.-C., Sung, N.-M. and Kim, J. (2015), “Demo: Towards Global Interworking of IoT Systems – oneM2M Interworking Proxy Entities”, *Proceedings of the 13th ACM Conference on Embedded Networked Sensor Systems*, ACM, New York, NY, USA, pp. 473–474.

BIOGRAPHY

Vasilios Tzeremes is a senior software engineer with over 17 years of technical experience and founding partner of a software consulting company operating in Northern Virginia. Throughout his career, Vasilios has developed numerous software solutions for private and government organizations. His area of expertise is in enterprise software design and development. Vasilios has an undergraduate degree in Business from the Technological Educational Institute of Athens and an M.S. degree in Information Systems from American University. He continues to learn and develop in his field by completing a PhD in Information Technology with concentration in Software Engineering from George Mason University. Vasilios research interests include software design and development, software product lines, end user development, software and enterprise design patterns, distributed systems, human computer interaction and enterprise systems.