LUND UNIVERSITY

PalCom Meets the End-User: Enabling Interaction with PalCom-based Systems

Johnsson, Björn A

2014

Link to publication

Citation for published version (APA):
Johnsson, B. A. (2014). *PalCom Meets the End-User: Enabling Interaction with PalCom-based Systems*. [Licentiate Thesis, Department of Computer Science]. Department of Computer Science, Lund University.

Total number of authors:
1

General rights
Unless other specific re-use rights are stated the following general rights apply:
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.
• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
• You may freely distribute the URL identifying the publication in the public portal

Read more about Creative commons licenses: https://creativecommons.org/licenses/

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

LUND UNIVERSITY

PO Box 117
221 00 Lund
+46 46-222 00 00

Download date: 04. May. 2026

PalCom Meets the End-User:

Enabling Interaction with PalCom-based Systems

Björn A. Johnsson

Licentiate Thesis, 2014

Department of Computer Science
Lund University

ISSN 1652-4691
Licentiate Thesis 2, 2014
LU-CS-DISS: 2014-2

Department of Computer Science
Lund University
Box 118
SE-221 00 Lund
Sweden

Email: bjorn_a.johnsson@cs.lth.se
WWW: http://www.cs.lth.se/bjorn_a_johnsson

Typeset using LaTeX
Illustrations created with Google Drawings
Printed in Sweden by Tryckeriet i E-huset, Lund, 2014

© *2014 Björn A. Johnsson*

Abstract

When developing applications for end-user nodes in distributed systems, it is not unusual to find that the business logic part of the solution is relatively inexpensive to develop. The resource intense parts of the solution are more likely to be those of presenting the functionality provided by the node to the end-user, and of connecting the node with the rest of the system. By considering systems that connect nodes using the PalCom middleware, this thesis presents work on a new technology to create graphical solutions for the end-user nodes of such systems. The criteria for the technology were formulated as a long-term aspiration of introducing a graphical editor for a PalCom specific User Interface Description Language (UIDL). Technically inexperienced PalCom users were introduced as the main target audience for the new technology, meaning no programming should be required for creating graphical solutions. To allow for this, a novel approach of turning the conventional way of specifying functionality $180°$ was introduced. The primary contribution of this work, serving as a first step towards the long-term aspirations, is the PalCom specific UIDL entitled PML – PalCom User Interface Markup Language. This language contains constructs that realize the ambition of inverted functionality specification, and will lie as a basis for the graphical editor that is to be developed in future work. The language was evaluated in a small scale user case study. The results were indicative of the high efficiency of PML, showing shorter developments time than the alternatives by a factor of 10. It was also evaluated in the real-world context of the itACiH project, showing the scalability potential of the language and its usability in the real world. For PalCom-based systems, PML has already been proved a competent option for creating graphical solutions, and will only grow more efficient with future research.

Contents

Preface v

Acknowledgements vii

I The Premise 1

1 Problem Description . . . 1
2 IT support for Advanced Care in the Home . . . 3
2.1 Introduction . . . 3
2.2 Challenges . . . 4
2.3 System Overview . . . 5
2.4 Development . . . 7
2.5 Project Status . . . 7
2.6 Conclusions . . . 8
3 PalCom . . . 8
3.1 Introduction . . . 8
3.2 Overview . . . 9
3.3 Concepts . . . 9
3.4 Tools . . . 13
3.5 Conclusions . . . 14
4 Previous Work . . . 14
4.1 Graphical Editor . . . 15
4.2 User Interface Description Language . . . 16
4.3 Conclusions . . . 17

II The Prototype 19

1 Aspirations . . . 19
2 PalCom User Interface Markup Language . . . 22
2.1 Motivation . . . 22
2.2 Language Overview . . . 22
2.3 Description Overview . . . 24
2.4 Description Details . . . 26
2.5 Description Interpreters . . . 33
3 New Tools . . . 35
3.1 AndroidPUIDI . . . 35
3.2 TheAndroidThing . . . 36

III The Evaluation **39**
1 Efficiency Comparison . . . 39
1.1 Application Solutions . . . 40
1.2 Development Time . . . 43
1.3 Lines of Code . . . 45
1.4 Discussion . . . 46
2 Scalability in itACiH . . . 47
2.1 Methodology . . . 48
2.2 System Overview . . . 48
2.3 Application Overview . . . 49
2.4 Login Screen . . . 50
2.5 Main Screen . . . 53
2.6 Locked Screen . . . 63
2.7 Discussion . . . 64
3 Practical Findings . . . 65
3.1 Link Vagueness . . . 65
3.2 Unpredictable Needs . . . 66
3.3 Text Formatting . . . 66
3.4 Command Filtering . . . 67
3.5 Dynamic Content . . . 68

IV The Finale **71**
1 Future Work . . . 71
1.1 Graphical Editor . . . 71
1.2 Language Augmentations . . . 72
1.3 Integration with PalCom . . . 73
1.4 User Case Study . . . 75
2 Summary . . . 75
3 Conclusions . . . 78

A The Listings **81**
1 Echo Application . . . 81
2 Heart Rate Application . . . 83

Bibliography **87**

Preface

The work presented in this thesis was carried out during the first three (out of five) years of the author's Ph.D. candidacy. Although research has been done in other related areas (see list below), this work constitutes the primary research effort for the aforementioned period of time.

Contribution Statement

The author of this thesis, Björn A. Johnsson, is the main contributor of the work presented herein. He is the main designer and implementor, and also carried out the work on evaluating the thesis results.

List of Related Publications

Below follows a list of publications that are related to this dissertation, and which have contributions by its author.

I **Developing Mobile Systems using PalCom: A Data Collection Example**
Björn A. Johnsson, Boris Magnusson
The 10th International Conference on Pervasive Computing (Pervasive 2012), 18-22 June 2012, Newcastle, UK

II **Some Like it Hot: Automating an Electric Kettle Using PalCom**
Boris Magnusson, Björn A. Johnsson
In Proceedings of the 2013 ACM conference on Pervasive and Ubiquitous Computing adjunct publication, pp. 63-66. ACM, 2013.

III **An On-Demand WebRTC and IoT Device Tunneling Service for Hospitals**
Thomas Sandholm, Boris Magnusson, Björn A. Johnsson
The 2nd International Conference on Future Internet of Things and Cloud (FiCloud 2014), 27–29 August 2014, Barcelona, Spain

Acknowledgements

First of all, I would like to express my gratitude to my supervisor, professor Boris Magnusson, for the opportunity to work on this project. His invaluable support and advice has been essential to the success of my research. I would also like to thank my co-supervisor Görel Hedin.

In addition, I thank my coworkers from the itACiH project, both past and present, for providing useful feedback that helped shape the course of this work. Furthermore, the medical staff working with the project deserve thanks for providing the kind of insight into my research that only could have come from using its results in a practical setting.

Finally, I thank my wife Emma Johnsson for her patience and support throughout the entire study period, and especially during the time it took to write this thesis. Without her encouragement and understanding during all those nights of working late none of this would have happened.

This thesis is dedicated to my firstborn, Frida Johnsson.

Björn A. Johnsson
Lund, October 2014

This work was funded by The Swedish Governmental Agency for Innovation Systems (VINNOVA) under a grant to Lund University for itACiH – IT support for Advanced Care in the Home.

Chapter I

The Premise

This chapter will introduce the work that is to be presented throughout this thesis, and will provide the background material needed to understand the same. In section 1, the underlying problem which motivates this work is discussed, i.e. the creation of graphical solutions for end-user nodes in distributed systems. The itACiH project is presented in section 2. The project both manifests the problem description and provides a practical evaluation environment for the results of this work. The results are based on PalCom, a middleware that is introduced in section 3. The chapter is rounded off in section 4 by the presentation of previous work on the subject matter.

1 Problem Description

Distributed Computing [22] is a well-established field of research. In a *Distributed System* components are distributed across multiple networked computers – nodes – that communicate by sending messages. Through such messages the components can coordinate and interact with each other in order to accomplish a common, often large scale objective. Distributing a system across a network comes with some drawbacks, e.g. the lack of concurrency between components. However, such trade-offs are usually out-weighed by the possibilities of a distributed system. As an example, outsourcing resource-heavy computations to a capable machine could make an otherwise unmanageable feature possible on a device with limited resources.

An application for an end-user node in a distributed system is generally composed of three primary parts: *network communication*, *business logic* and *graphical user interface* as illustrated in fig. 1.1. The **network communication** (NC) part of the application is made up of all functionality needed to include the node in a given distributed system: establishing and maintaining connections between the distributed nodes, handle the sending/packaging and receiving/parsing of messages, etc. The **business logic** (BL) part of the application includes all functionality needed for the node to accomplish what it was actually designed to accomplish. This could involve simple functionality like collecting data measurements using

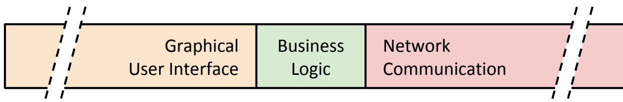


Figure 1.1: The three primary parts of an application for a general end-user node in a distributed system

a sensor connected to the node, e.g. a thermometer, or more complex orchestrations. The final part, **graphical user interface** (GUI), consists of the functionality needed to present the node's features to the end-user. The purpose of the GUI is to enable the possibly inexperienced end-user to access the functionality from the business logic part of the application, as supported by the network communication part of the application.

In fig. 1.1, BL is depicted as well-defined and small relative to the other two parts. This is because in many cases the BL of an end-user node application is not the primary drain on development resources; making the node accomplish its design goals can be relatively easy. The resource intense tasks are usually to present the solution to the end-user (GUI) and to the rest of the system (NC). In fig. 1.1 this is depicted as the size of GUI and NC being an unbound magnitude greater than that of BL. This allocation of resources is sub-optimal; resources are wasted on ambient tasks that are certainly necessary for a fully working application, but that don't really contribute to the solution of the problem at hand. Minimizing this waste would free up resource that could be better directed elsewhere.

When examining the overhead to the BL of an end-user node application, this section has distinguished between GUI and NC. The problematic overhead caused by the later is a well-covered area of research. Simplifying this part of the application could be achieved by using a *middleware*, e.g. *SpeakEasy* [18], *Jini* [14] or *PalCom* (section 3). If used correctly, this should make NC smaller in relation to BL.

Remaining is the overhead caused by the GUI part of the application. This overhead is caused by many factors, e.g:

- Special skills might be needed due to varying platform conditions. Depending on the platform (Android, PC, Mac, etc.) the GUI is targeted for, the user may need to master a new programming language and/or set of tools.

- Graphical solutions are a hotbed for *boilerplate code*, i.e. code that is similar between different solutions, but that still has to be created time and time again. Underlying infrastructure, e.g. screen allocation or startup/shutdown procedures, is one source of this. Another one is *glue code*, i.e. code that simply connects (glues) the GUI to different parts of the application.

- The mere requirement of writing programming code can severely undermine the ability to create the graphical solution for less technically savvy users. The lack in competence for writing code may even completely prevent the user from doing so.

This list is by no means exhaustive, but serves to highlight some of the problems associated with GUI development.

Assuming a middleware solution to the network communication problem, it is the aim of this thesis to streamline the process of creating the graphical solutions for end-user nodes in distributed systems.

2 IT support for Advanced Care in the Home

The contributions presented in this work have primarily been evaluated in the context of the *itACiH* project. A basic understanding of the project and its domain should therefore contribute to a more contextually sound interpretation of the work.

2.1 Introduction

The number of patients diagnosed with cancer in Sweden are expected to double over the period 2013–2030 [15]. To counter this development, a new regional health organization has been established with the purpose of streamlining the treatment of these patients. Many patients appreciate the possibility of being treated at home, given that they can feel safe. This is most welcome, since the expectation is that an increased rate of treatment in the home will be most cost-efficient in the long run. To make the patients feel safe at home, innovation is required in many areas, e.g. communication with both hospital and mobile staff, remote surveying patient status, remote adjusting medical equipment, etc. Work on systems that support such features is critical to increase the number of patients that can be treated at home.

In the *IT support for Advanced Care in the Home* (itACiH) project [6] a new open platform for supporting home-based care is being developed. Features include remote interaction with medical equipment at home, tablets for mobile consumption and production of patient data, and stationary units for wards that provide overview, control and video interaction. The current focus is on home-based care for terminally ill cancer patients, but the long-term goal is to provide additional treatment forms, as well as support home care for patients with diseases other than cancer.

The project is a cooperation between academia and the corporate world, including Lund University, Lund University Hospital, 6 inner circle companies, and a multitude of outer circle companies. The founding is coming from *VINNOVA*, a government research funding agency, as part of their ”Challenge Driven Innovation” program. In the project, academic research in Information and Communications

Technology (ICT) is combined with compentence from cross-discipline companies, e.g. medical, telecom, and security companies. The project plan is agile and the work iterative, giving the end users, such as caregivers and patients, great influence over how the project is to evolve.

2.2 Challenges

Most patients prefer not to spend more time away from home than is absolutely necessary. This is particularly true for patients in the final stages of life. Depending on where in Sweden the patient is living, the possibility to receive home care may vary. Advanced home care is made possible by specialized facilities, where patients are enrolled at a hospital ward, but treated at home. The availability of such facilities is supposed to increase in all regions during the coming years.

In the specific case of palliative care for cancer patients, the treatment focus is mostly on relieving pain and maximizing quality of life. This kind of treatment involves medical equipment in the home, and typically weekly visits from the medical staff. Visits may be required more frequently, but if they become too frequent, the patient is moved to the ward instead. In addition to planned visits, every reading or adjustment of the equipment located in the patient's home requires a visit from the staff.

2.2.1 Patient Record Entries

Prior to the itACiH project, there was no system or infrastructure for supporting home care of patients. Outside the project, the usual way of working is that the medical staff do essentially all their work on paper notes. These notes can be regarding who to visit, what to do during the visit, what observations were made, which forms were filled in, equipment readings, etc. After these notes are brought back to the ward they have to be manually entered into a patient record system. Handwritten notes have to be retyped, forms and checklists have to be scanned, etc. This not only means delay in registration and duplication of work for the staff, but also affects the quality of the digital records. Typed notes and scanned documents cannot be searched or used for statistical evaluations, nor can they be graphically rendered to observe trends. Having both the equipment and the staff enter data directly in digital form opens up a lot of possibilities in terms of efficiency and quality.

2.2.2 Equipment Measurements

For equipment in the patient's home, the staff has to take sample measurements manually, e.g. by reading the display on the physical device. Aside from the additional travel time, this is a problem because the data cannot be analyzed before the sample is brought back to the clinic. This is not ideal since the staff usually

visits multiple patients during the same trip, and may not return to the clinic until later in the day. If tests indicate the need for a change in treatment, this treatment is also delayed. Having equipment report measurements remotely would alleviate the problem of delays, and increase accuracy in the readings.

2.2.3 Equipment Control

Adjustment and general control of the equipment in the patient's home is another challenge. The equipment is designed for hospital use, and is typically operated by buttons on the device itself. In the case of cancer patients, pain relief is often provided using infusion pumps delivering anesthesia. The dosage often has to be adjusted, which involves a lot of adjust-observe-repeat to get the dosage right. Every adjustment and observation requires a visit. Hence, making the home-based equipment available for remote monitoring and control would not only save much travel time, but would also mean less interference for the patients.

2.2.4 Piece of Mind

It is important that the patients feel safe at home, otherwise the quality of life benefits of being at home would be nullified. The patients need to know that someone is keeping an eye on them. Someone who knows their situation and condition, and someone that they can easily get in touch with. When treated at the palliative centers, the patient can press a button to summon a nurse at any time. This sense of security must be supported by the infrastructure, only substituting the physical nurse with a video conference nurse. If the situation calls for it, a staff member can be dispatched to the home just like today. However, if the situation can be managed remotely, travel time is saved.

2.3 System Overview

The primary focus of the itACiH project is to develop a working prototype for a system that supports palliative home-based care. The system is being built using a framework called PalCom (see section 3), which supports flexible communication and combination of devices over heterogeneous networks. Figure 1.2 illustrates the system being developed in the project. A few key nodes have been identified: *server*, *home*, *mobile team*, *ward station*, *patient*, and *relative*.

The **server** is the primary junction point of the system. All other entities connect to the server in some way, and either read or write (or both) data. Patient data is decoupled from patient information through the use of pseudo anonymous patient identifiers.

Support for communication with equipment based in the patient's **home** is supported through a communication hub to which equipment can be connected. To be practical, some equipment needs to be mobile. For such cases, wireless

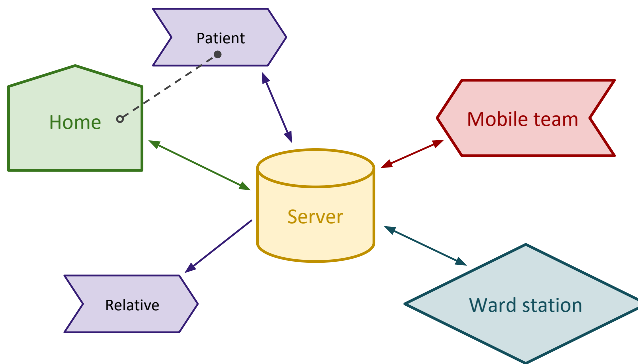


Figure 1.2: Simplistic overview of itACiH system

solutions are being explored. Technologies for integrated video communications solution are also being explored.

The nurses that visit the patients in their homes, the **mobile team**, are equipped with tablets, through which they can communicate information in real-time. They can for example view patient information, or look up which tasks are to be perform during a visit and check them off as they are completed. They can also fill out electronic counterparts of forms and enter information about the patient's condition, making it instantly available throughout the system.

The **ward station** is used by the staff for interaction with home-based equipment and the viewing of patient data. This station integrates the information from all the various sources in the system, for example: the medical equipment at home; assessments, checklists, pictures and other data entered by the mobile team; legacy patient record systems; etc. The presented information can be expanded as the needs of the staff are identified, and may in the future integrate different kinds of useful information about the patients.

For individual members of staff that are unable to access the main station at the ward and that are not equipped with a tablet, there is another option. Select features from the main station are available using any ordinary web browser, given the right credentials. What features are available depend on the professional role of the person who logged in, e.g. nurse, or physician. For both technical and security reasons, the functionality of the web-based interface is somewhat more limited than the primary station at the ward.

For **patients**, there are a number of interesting scenarios to support, e.g. self administered condition assessments, or registering that a scheduled drug has been consumed. So far, this entity has not been explored in depth. The idea is to use a scaled down version of the tablets used by the mobile team. The device could be used in the home, but there may also be features that are useful outside the home, on the go. Another alternative is to use the web solution with patient credentials.

Similar to patients, **relatives** or friends involved in the treatment of the patient may be interested in monitoring the condition of the patient. This could be done with a further scaled down version of the mobile team tablet, or using the web solution.

2.4 Development

The itACiH project is being developed using participatory design [21]. Prospective users, such as physicians and nurses, are part of the development process, and actively participate in the design of the system. The system is being built iteratively, meaning that it is build and delivered in increments to the end-user, where it is used and hence evaluated by professionals in a real-life, practical environment. To safeguard against failures due to the prototype nature of the system, the system is used alongside the normal, mostly paper-based routine. Tools supporting the nurses in the field are used and evaluated in the field. Functionality for home-based equipment, however, is evaluated in a special room at the ward, a ”simulated home”. This is to ensure patient security; should anything go wrong, e.g. when remote managing equipment, the staff is right outside the door instead of a 30 minute drive away.

The iterative design loop is turned around on a weekly basis, and the system is updated to a new, improved version every month. The development team is situated in close approximation to the palliative care unit, enabling good communication between the developers and the end-users. Comments can hence relatively quickly become improvements to the system, and suggestions for new functionality can easily be collected and feed into the design process. This gives the end-users a sense that they can make a difference, further encouraging communication. The staff, alongside the development team, is also involved in prioritizing new functionality, putting focus on both what is needed in the field as well as on what is technically viable.

Medical systems need to fulfill certain requirements for active use. The certification process is also done using an incremental model. The model is supported by the component-based architecture of the infrastructure which structures the system as a collection of several smaller, separate subsystems. This allows for partial review after changes and additions to the system.

2.5 Project Status

The first stage (two years) of the project ended in November of 2013. The success of it was apparent, and the project received another two years of funding, thus entering its second stage. During the first stage, much of the basic functionality was developed. It is now in place, and is actively being used by professionals at a single location in Lund, Sweden. The second stage of the project will focus on

refining and adding new functionality to support the staff, spreading to multiple locations, and adding support for areas other than palliative care.

2.6 Conclusions

The many end-user nodes in the distributed system being developed in the itACiH project, e.g. the patient or the mobile team, serve as a great motivator for the work in this thesis. Many (if not all) of the identified nodes require a graphical solution for the end-user; creating a new, ad hoc solution for every node is not sustainable. In this way, the project manifests the need of a uniform technology for creating graphical solutions for distributed systems in general. Furthermore, the project provides a natural, real-world context in which to evaluate such a technology.

3 PalCom

PalCom is a middleware upon which the system in the itACiH project is being built. It is alo the primary focus of the graphical solutions presented in this work. Hence, a fundamental understanding of what PalCom is and aspires to achieve is necessary for the scope of this work.

3.1 Introduction

Ubiquitous Computing [28] refers to a type of computing in which computers in different forms completely penetrate every aspect of the users' life. Unlike traditional computing devices, e.g. desktop computers or laptops, ubiquitous devices are everywhere, and typically do their work without the user ever noticing; they ”vanish into the background” [27]. They're usually built to solve a limited set of problems, and therefore only offer one or a few specific services. A ubiquitous device could for example count the number of people entering and exiting a room, and offer the current number in the form of a service. The limited functionality of these devices introduces the issue that to fully take advantage of their power, communication amongst one or more devices is often necessary – ”the whole is greater than the sum of the parts”. Since most devices offer only one or a few predefined methods of communication, interconnecting devices is guarantied to be a challenging and time consuming task.

The problem of having different devices communicate with each other was (among other things) addressed in the *PalCom* (*Palpable Computing*) project [8], which ran between 2004 and 2007. The project aimed to solve this problem by introducing the concept of *palpability* to the emerging computing devices. The term ”palpable” refers to systems that can be both discovered and logically understood. In practice, this means that palpable systems should support the user in understanding and controlling the services provided by their devices, by offering a

coherent interface. The project extended to how services on any type of device may be logically interlinked with each other, providing not only the means to control individual devices, but actually combining several devices to provide entirely new functionality [13].

PalCom was originally developed as part of an *EU-FP6 Information and Society Technology* [5] program on *Future and Emerging Technologies*, and has since been further developed by researchers at Lund University. It has been used in healthcare systems; in combination with *1-Wire* and *Tellstick* to enable flexible combinations of wired and wireless actuators and sensors; in combination with *AXIS* cameras and remote sensory products; and much more. Current development includes applications for real-time control of mobile robots in the *ENGROSS* project [4], and advanced health care in the home in the itACiH project.

3.2 Overview

PalCom is a middleware framework [25, 26] used to combine the services offered by devices in an easy and flexible manner [20]. The middleware enables devices to be combined across heterogeneous networks, and the services they offer to communicate even if they were not designed to work together. By doing so, new functionality can be created by coordinating already existing services in new formations. PalCom supports this by offering mechanisms for discovery and routing between different network technologies, a standardized way to exchange descriptions of services (rather than standardize the services themselves), and a combination (assembly) mechanism based on configurations and coordination scripts. PalCom uses a homogeneous approach to representing system components which hides the complexity of systems that connect across heterogeneous networks. Once created, PalCom components can be used and reused in any number of solutions.

The ability to offer transparent combination of devices across heterogeneous networks is enabled by a network independent addressing mechanism. This means that devices can move between networks, yet remain attached to their system(s). Assemblies, which store bindings to devices, can be moved between different devices and still connect to the correct targets. This creates a unique flexibility, allowing devices and functionality to be moved between networks without the need for reconfiguration.

PalCom is implemented in Java and runs on most computers including Mac OS-X, Windows, Linux/Unix, and Android. A partial implementation in C makes it possible to include also smaller devices, although with limited functionality.

3.3 Concepts

An illustrative overview of the main concepts of PalCom are depicted in fig. 1.3. A PalCom *Device* ($D1$, $D2$, $D3$) represents a physical (or simulated) device in

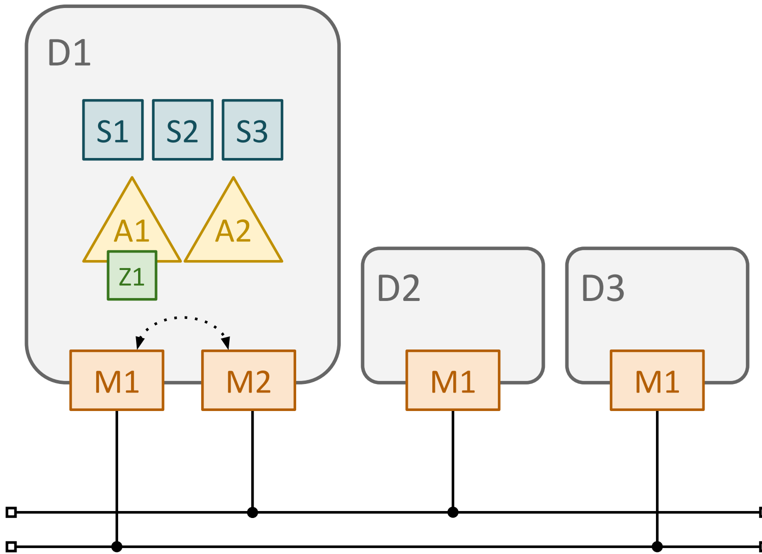


Figure 1.3: Main concepts in PalCom: Device (D), Media Abstraction Object (M), Service (S), Assembly (A), and Synthesized Service (Z)

the PalCom *Universe*[1]. Devices provide functionality in the form of one or more PalCom *Services* ($S1$, $S2$, $S3$). Services are coordinated by PalCom *Assemblies* ($A1$, $A2$), which can be "silent drivers" or provide functionality of their own in the form of *Synthesized Services* ($Z1$). In PalCom, devices connect to the universe through a number of *Media Abstraction Objects* ($M1$, $M2$), which represent some kind of communication media, e.g. UDP or Bluetooth. Devices can communicate with other devices that are connected to a corresponding media abstraction object. Furthermore, the media abstraction objects of a device can route traffic between each other, enabling a device on one network to find a device on a second network.

The concepts introduced above will be touched upon in more detail in the following sections.

3.3.1 Devices and Services

A PalCom **device** can represent any kind of device. A device is typically a piece of hardware, like a GPS tracker or a digital camera, but it could also be a virtual device simulated in software. Ideally, the PalCom device should run on the physical device's hardware, accessing the functionality of the device with no middleman and communicating directly through its available communication medias. However,

[1] Colorful name for the conceptual collective that all PalCom devices belong to

this is not always possible, e.g. due to limited computing or memory resources or simply due to the lack of access to the hardware's inner workings. In such situations it can be useful to run the PalCom device as a simulated device. This is usually done be connecting the hardware to a computer on which the simulated device is run. A *bridge service* that communicates with the hardware through given drivers can then serve as a layer between the PalCom environment and the system of the physical device.

A PalCom **service** is hosted on a PalCom device and typically represent some kind of computation or action that can be performed by the device. The service can have a direct correlation to something in the physical world, such as displaying a text the screen of a device or moving one of its actuators, but may also be purely software oriented, e.g. with the purpose of updating an internal counter. A service is defined by its *service description*, which defines the interface towards which future users will operate. By having services provide their description on request, they are said to be *self-describing*. This is in clear contrast to the traditional approach in service-based architectures, where services have to adhere to some sort of application standard. One strength of PalCom then becomes that independently developed services that weren't designed to cooperate, and that traditionally would be incompatible, can still be combined due to their self-describing nature.

The service description specifies a list of *commands* that the service provides for the user; it is through sending (and receiving) commands that the user manipulates a given service. A PalCom **command** is a message that can be sent to, or received from, a service. They are identified with a (service level) unique ID, specify a direction that indicates whether the service should receive or send the command, and optionally list none or more *parameters*. A PalCom **parameter** is a construct that holds the data being passed to or from a service; a command with no parameters can be sent or received, but won't contain any actual data. Parameters are identified by a (command level) unique ID, and may contain different kinds of data, such as plain text, images, etc.

3.3.2 Communication

As illustrated in fig. 1.3, a device can have multiple Media Abstraction Objects (MAO). Each MAO represents some media over which the device can communicate, e.g. UDP or Bluetooth. When a service on a device wants to communicate with a service on a second device, it delegates the command to be sent to the sender device's *communication layer*. The communication layer then chooses any appropriate MAO on which the receiver device is reachable, and sends the command. Notice how PalCom in this way becomes *network transparent*: service developers don't have to think about how the command is going to reach the target device, only if it actually did or not, hence practically hiding the network complexity from the developers perspective. This is possible since devices are

addressed by a PalCom level unique ID, not on a MAO specific level e.g. by an IP address. This enables devices to seamlessly ”move” between networks.

To maintain this flexibility, PalCom uses a proprietary *discovery protocol* to enable devices to be aware of each other. In short, this is achieved using *heartbeats*. At given frequencies, all MAOs of a device (optionally) send a broadcast message that tells other devices reachable through that MAO that the sending device is still ”alive”. The heartbeat also urges other devices to in turn identify themselves as alive. If no heartbeat for a given device has been received for some predefined amount of time, the device is considered unreachable on that MAO. For example, if a device migrates from one network to another, it would eventually become unreachable for all devices on the old network, since the heartbeats from the device would stop arriving. However, for devices connected to both the old and the new network the device would immediately reappear. Since the device is always addressed by its PalCom ID, the switch between networks would go unnoticed; such is the power of network transparency.

On devices with more than one MAO it can be meaningful to enable *routing*. Routing enables messages (discovery heartbeats, service commands, etc.) sent from devices on one MAO to be resent to devices on all other MAOs of the receiving device, enabling devices to expand their reach beyond their own communication capabilities. In fig. 1.3, $D1$ has routing enabled as illustrated by the dashed line between the device's two MAOs, $M1$ and $M2$. Because of this, $D2$ will be able to communicate with $D3$ and vice versa, even thought there is no direct line of communication between the two. For example, a Bluetooth gadget device might be limited to Bluetooth (MAO) for communication. To enable non-Bluetooth devices to interact with the gadget, a Bluetooth MAO could be enabled on a smartphone which has several other MAOs and routing enabled. Other devices could then reach the gadget through the smartphone.

3.3.3 Configuration and Coordination

In PalCom the *functionality* of services is strictly separated from *configuration* and *coordination*. Typically, services don't establish connections to other services; they provide functionality as described in the service description, and don't need to know how or from where commands are arriving.

> **Functionality** *Services should provide functionality, and only functionality*

To create connections to one or more services, a PalCom **assembly** is used. The *configuration* of services is the first of the two major parts of an assembly. In fig. 1.4 the assembly (A) is configured to create connections to two services: $S1$ and $S2$.

> **Configuration** *Assemblies specify which services to establish connections towards*

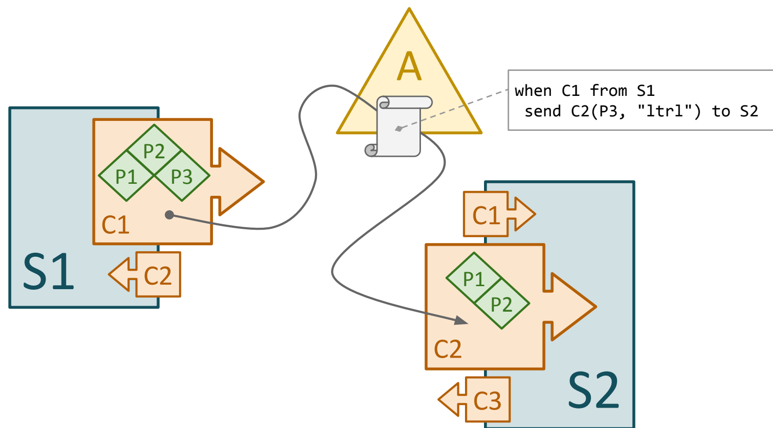


Figure 1.4: Assembly (A) *configuration* of services (S) and command (C) *coordination* with selective parameter (P) passing

The other major part of an assembly is the *coordination* of services. Coordination is specified in the form of a simple event-based scripting language. A script is made up of a set of *event–action* pairs. There are a few different events to choose from, the most common being a command arriving from a service. For each event, one or more actions are specified. Actions are also available in a variety of different shapes, e.g. sending a command to a service or setting the value of a variable. In the script excerpt of A in fig. 1.4, the arrival of $S1$'s first command ($C1$) is listed as an event. When $C1$ arrives, the second command ($C2$) of $S2$ will be sent to $S2$. Notice how only one of $C1$'s parameters ($P3$) is passed to $C2$, and how the second and last parameter of $C2$ will get the value of a string literal (*"ltrl"*).

Coordination *Assemblies script the interaction behavior of services through a set of event–action pairs*

In most cases, assemblies are completely autonomous and silently drive its part of a system. Another possible use of assemblies is to aggregate functionality into new, combined functionality and provide that in the form of a *synthesized service*. Synthesized services are specified as part of the assembly, and can then included in scripts (its own or others') like any other service.

3.4 Tools

To enable and sometimes just to simplify the use of PalCom, a number of helpful resources have been produced. To mention a few of the major, the PalCom toolset consists of:

PalcomBrowser

The PalcomBrowser is used for browsing the PalCom universe. It is itself a PalCom device with selectable Media Abstraction Objects (MAO). The tool list devices reachable by the browser's device. Devices can be examined, and services on those devices can be interacted with by means of primitive, but automatically generated GUIs. The PalcomBrowser also enables the creation of assemblies through the use of a graphical editor.

TheThing

TheThing is an all purpose PalCom device which acts as a virtual machine for services and assemblies. Primarily, TheThing is used to host assemblies on a device that can reach all of the configured services. Hosting assemblies is typically not something that special purpose devices do. Secondarily, TheThing can also host services. This functionality can be used during development to test services, for services that don't naturally belong to the domain of any special purpose device (e.g. text formatting, or unit conversions), or for bridge services. Lastly, TheThing can also be used to define cross service/assembly parameters, i.e. values that should be defined once for multiple uses.

Library

Although technically not a tool, the PalCom library (*lib-palcom*) provides a collection of Java classes used during the development of new services, MAOs, etc. Most commonly, the generic service class `AbstractSimpleService` is inherited and extended in order to create new, specialized services.

3.5 Conclusions

It stands clear that the many constructs of PalCom makes it well-suited to handle the network communication part (and more) of any distributed system; the capacity to distribute a system of considerable size has been established in (among others) the itACiH project. However, it is also clear that PalCom currently lacks the constructs and tools necessary to produce graphical solutions for such systems. At present, third-party development tools is the only option. This absence motivates the creation of a uniform technology for creating graphical solutions, specifically for PalCom systems.

4 Previous Work

When tackling the challenge of creating graphical solutions (in general), two primary technologies have been identified: *Graphical Editors* and *User Inter-*

face Description Languages. Both have strengths and weaknesses that must be considered from the perspective of this work.

4.1 Graphical Editor

An *Integrated Development Environment* (IDE) is an application that integrates features needed for developing, compiling and debugging new applications, all in the same environment. IDEs are in many cases very efficient at shortening development times for applications in general. To shorten development times specifically for the *Graphical User Interfaces* (GUI) of new programs, many IDEs also include a graphical editor, e.g. *Swing GUI Builder* (formerly known as *Project Matisse*) [7] in *NetBeans*, *WindowBuilder* [3] in *Eclipse*, or *Interface Builder* [9] in *Xcode*.

Graphical editors typically present the user with a blank canvas on which to compose the GUI. The user selects graphical components, e.g. buttons, text boxes or images, from a palette of components as provided by some underlying framework. The components are positioned on the canvas by dragging them from the palette (using the pointing device, e.g. mouse) and dropping them onto the canvas. When a component is selected, a list of its properties will typically be displayed in the editor, allowing the user to customize the look and, to some degree, behavior of the component simply by entering new values into the editor. This method of working simplifies the process of creating GUIs by requiring no (or little) programming skills, and by constantly providing the user with visual feedback, confirming what the final GUI will look like.

After partially or completely finishing the graphical layout of the GUI, the user is expected to define the behavior of the GUI, typically by *linking graphical components to functionality*. This is done by implementing *callback* methods, i.e. methods that are called by the graphical components when certain events occur. A button might for example call a method `onClicked` every time it is clicked in the GUI by the end-user. In contrast to the first phase of GUI production using a graphical editor, this seconds phase requires at least some programming skills from the user. Depending on the complexity of the application, this requirement might be small or large.

Graphical editors are generally tightly bound to a certain programming language and/or runtime platform. Both Swing GUI Builder and WindowBuilder are used to develop GUIs for Java applications, and the resulting products run mainly on desktop computers. As a counterexample, the *Graphical Layout Editor* [1] of *Android* is also used for Java applications, but the resulting product only runs on devices powered by the Android OS (Operating System). This amounts to a situation where the user might be required to have expertise regarding multiple languages/platforms in order to create the desired graphical solutions.

4.2 User Interface Description Language

A *User Interface Description Language* (UIDL) allows the user to describe Graphical User Interfaces (GUI) that are *platform independent*, i.e. GUIs that are not restricted to a single, specific platform, but rather can be loaded on multiple different platforms. Some examples of such languages that have been considered for the context of this work are *UsiXML* [19], *XIML* [24], *TeresaXML* [23], and *UIML* [10–12].

When creating a GUI using a UIDL, the user does so by writing code according to the given language specification. This specification specifies which graphical components are available (buttons, text boxes, etc.), how GUIs should be structured (layout, spacing, etc.), how to define component behavior, and more. Different platforms have different visual styles and capabilities. Since the language is platform independent, its graphical components are typically derived by generalizing platform specific GUI components into general purpose abstractions that are applicable on any platform. Using the language, the user defines what components should be included in the GUI. For each component, the user then writes code that sets properties specifying how the component should be positioned in the GUI, what it should look like, etc. This method of working admittedly requires programming skills from the side of the user, which might be a complication. However, it also simplifies the process of creating GUIs by only requiring the user to learn the one language, which can then be used for any platform as necessary.

Since a UIDL described GUI is created by writing programming code, during the development process the user gets no visual confirmation regarding if the resulting product looks as desired. Some work has been done in providing graphical editors for purpose specific UIDLs, e.g. *SMUIML* [17], hence alleviating this issue for those specific purposes. However, no general purpose UIDL with a graphical editor has been identified in this work. Instead, to test a GUI its description is loaded into a platform specific application that interprets the description based on the language specification. The result is an application with a GUI that works as specified in the description. While the UIDL provides a platform independent way of describing GUIs, a platform specific interpreter application has to be developed for each platform that is actually to be supported. Such applications can be expected to be provided to the user, meaning no additional development investment.

The functionality of components in a UIML described GUI is primarily (and from a simplistic perspective) specified in one of two ways: as *internal actions*, or as *external method calls*. Internal actions allow the user to specify behavior that directly affect other part of the GUI. What actions are available depend on the language, but generally these types of actions only allow for simple behavior to be specified. As an example, a button component could be instructed to change the text of some other component in the GUI whenever it is pressed by the end-user. For more advanced behavior, such as heavy computations or network communica-

tion, components can be instructed to perform method calls on external modules. External modules are not subject to the platform independence criteria of UIDLs, and can as such be programmed using any programming language. Limitations on available languages may be imposed by the interpreter application of a given platforms, where external modules must be runnable. To allow for this type of method calls and still remain platform independent, a UIDL will typically provide a generic way to invoke methods. This way, given the external modules, the user only needs to be concerned with the one language – the UIDL – regardless of the language(s) used when developing the modules.

4.3 Conclusions

A typical graphical editor offerings the user a mostly graphical approach to creating graphical solutions. However, select parts of a solution must be written in programming code, which requires at least some programming skills from the user. Furthermore, graphical editors are typically language/platform dependent. UIDLs are on the other hand platform independent by definition. However, in this work no general purpose UIDL has been identified to have a graphical editor to support the user. Because of this, all graphical solutions must be created entirely by writing code when employing a UIDL, which again requires the user be have a certain level of technical knowledge.

Summary

The discussion around the problem description in section 1 motivates the need for a uniform technology to create graphical solutions for end-user nodes in distributed systems. The many end-user nodes in the distributed system being developed in the itACiH project, as presented in section 2, manifest this general need from a practical perspective: all of the nodes in the system require a graphical solution of some sort and creating a new, ad hoc solution for every node is not sustainable. Furthermore, it stands clear from the introduction of PalCom in section 3 that the middleware currently lacks the constructs and tools necessary to produce graphical solutions for systems built upon its framework. At present, the user must resort to third-party development tools when such solution are necessary. This lack of support for graphical solution in PalCom, coupled with the many nodes in need of such solutions in the itACiH project motivates the creation of a uniform technology for creating graphical solutions, specifically for PalCom systems. The technologies of previous work, as presented in section 4, do not meet the last criteria; a new technology is needed.

Chapter II

The Prototype

This chapter will present the prototype solution to the problem posed for this thesis, as presented in chapter I. Section 1 bridges the two chapters by presenting the aspirations, both long-term and short-term, for such a solution. The scope of the work is also narrow here by the introduction of a target audience. In section 2, the developed prototype for a PalCom specific UIDL is discussed, both on a superficial level and in greater detail. Finally, section 3 closes the chapter by introducing two new PalCom tools necessary for the language presented in section 2 to be useable in the context of the itACiH project and beyond.

1 Aspirations

As concluded in chapter I, the lack of support for graphical solution in PalCom and the vast need of just such solutions in the itACiH project motivates the creation of a uniform technology for creating graphical solutions, specifically for PalCom systems. A new such technology needs to adhere to the already established *user levels* of PalCom. Table 2.1 show the primary roles that PalCom user can currently take on. The distinction in user level indicates that the roles require different levels of technical expertise from the user. Level 0 users create assemblies using the PalcomBrowser – a standard PalCom tool. No other software, and hence no other technical skills, are required. The user experience it totally graphical, demanding no coding knowledge from the user. This non-requirement is intrinsic of the level 0 user. Level 1 users develop new PalCom services using, for example, a third-party IDE. At level 1, users are expected to have the technical expertise required to

Level	Role	Creates	Uses
1	Developer	Service	3rd-party IDE
0	Assembler	Assembly	PalcomBrowser
	End-user		PalCom systems

Table 2.1: User levels in PalCom

install required software (compilers, libraries, etc.) and to program in a given programming language, at the moment Java. This is a considerable step-up in technical skill requirements compared to that of level 0 users, and the distinction is key for any new addition to PalCom.

To solve the problem at hand, a new technology that provides support for graphical solution in PalCom is to be introduced for level 0 users. The choice of introducing the technology at this level has the following implications:

1. The technology must be platform independent. PalCom runs on multiple platforms, and any graphical solutions must do so as well. However, users at level 0 cannot be expected to have the necessary knowledge to make a solution work on multiple platforms. Instead, the new technology must allow the user to create the graphical solution in such a manner that it becomes platform independent, allowing for it to be loaded on not just one predetermined platform.

2. The technology must allow for *codeless functionality specification*. In PalCom, functionality is provided through services which are produced by level 1 users. When assembling PalCom systems, level 0 users can utilise these services by incorporating their functionality into the system. This allows level 0 users to define the functionality of an entire PalCom system, all without having to write any programming code. The same concept must also hold true when creating the graphical solutions for such systems.

Considering requirements 1 and 2 above, it becomes apparent why the previous approaches to creating graphical solutions, as identified in section 4 (I), don't suffice as a solution to the stated problem. A typical graphical editor accommodates level 0 users by offering a (partially) graphical approach to creating graphical solutions. However, some parts of the solutions requires code to be written which violates req. 2. Furthermore, graphical editor are typically language/platform dependent, which makes them incompatible with req. 1. User Interface Description Languages (UIDL), on the other hand, are by definition platform independent, which complies with req. 1 above. However, as graphical solutions are created entirely by writing code when employing a UIDL, this approach is not suitable for level 0 users (req. 2).

As a long-term aspiration for supporting graphical solutions in PalCom, a hybrid of the two previous approaches is proposed here. By harnessing the platform independent nature of a UIDL and combining that with the user friendliness of a graphical editor, a solution that satisfies both requirements 1 and 2 above can be realized. The idea is to introduce a graphical editor for a PalCom specific UIDL, targeted at level 0 users, i.e. the same type of users that create assemblies today. To avoid violating req. 2 above in the same way that traditional graphical editors do, the way in which functionality is specified will be turned 180°; instead of *linking graphical components to functionality*, the user will be *linking functionality to graphical components*. The distinction is subtle, but key to satisfying req. 2.

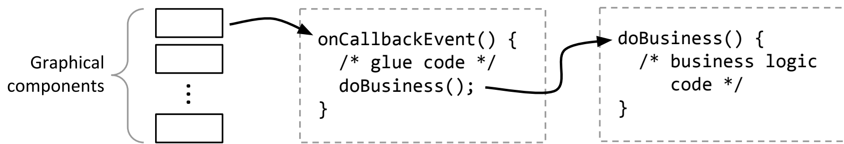


Figure 2.1: Traditional – *linking graphical components to functionality*

Figure 2.1 shows the traditional method of linking a graphical component to its intended functionality. Note that this is (at least) a two step process, including time consuming work on implementing tedious glue code. More importantly, the user is required to write both the glue code and the business logic code, which is clearly not in line with req. 2. In contrast, fig. 2.2 illustrates the inverted method where functionality – in the form of PalCom components – is linked directly to graphical components. In practice, PalCom components could for example be dragged and dropped onto the editor's canvas to produce appropriate graphical components to represent them.

The platform independence that results from building the would-be editor on top a UIDL comes at a price: the graphical parts of the language must be general enough to work on any given platform. This results in a *greatest common divisor* situation, where more graphically expressive platforms must conform to the limitations of less expressive platforms. To some degree, this limits the user's ability to control the appearance of the graphical solution on a fine-grained level; to what degree depends on how expressive the UIDL is made, and hence, how complicated it is for the user to employ it. The balance between language expressiveness and simplicity must be carefully considered. As a final note on platform independence, it is not the aspiration that one and the same graphical solution must look identical on all platform; there may be minor or major variations in how graphical component look and feel, how they are structured and arranged, etc. The aspiration is however, as stated earlier, that the user can employ the same technology for any (supported) platform.

As a first step towards the long-term goal presented here, this work will focus on the PalCom specific UIDL. The graphical editor for the language is outside the scope of this work, but will be explored in future work (section 1.1 [IV]).

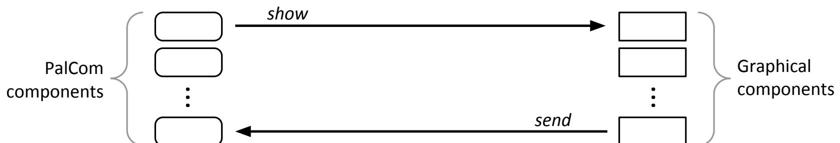


Figure 2.2: 180°– *linking functionality to graphical components*

2 PalCom User Interface Markup Language

The PalCom specific UIDL that is proposed as a solution to the thesis problem statement will be presented in this section. The language, together with two developed Android applications that accompany it serve as the primary contributions of this work.

2.1 Motivation

As discussed in section 1, existing UIDLs do not meet the criteria of a PalCom specific language for specifying graphical solutions in a platform independent manner. The basic idea for such a language is to allow for the user to specify the functionality of an application, all without having to write any programming code. To allow for this, the already established constructs of PalCom must be deeply integrated into the language, which of course is not the case for any existing UIDL. Furthermore, the language is to be developed prior to the graphical editor that will present the user with a completely graphical experience when creating graphical solution. Because of this, the language is to be initially kept simple, and is to be expanded upon as needed thereafter. This will keep the threshold for getting started using the language low. With the future introduction of the graphical editor for the language, this threshold will no longer be a problem and the expressiveness of the language can increase unhindered.

While not perfect matches for the purposes of this work, existing UIDLs still contributed to the problem solution. The PalCom specific UIDL developed in this work is entitled *PalCom User Interface Markup Language*, or *PML* for short, and is heavily inspired by UIML [10–12]. In addition to not meeting the PalCom specific criteria, UIML is far too expressive, both in terms of graphical and behavioral possibilities. However, by learning from what other languages got right and adapting that for the specific problem of this work, a first draft of what would become PML could be created.

2.2 Language Overview

PML is used to create *PalCom User Interface Descriptions*, also simply referred to as descriptions. Descriptions are made up of PML code that describe the application and GUI (Graphical User Interface) that will be created from it. This section will introduce the core concepts of PML on a superficial level. Sections 2.3 and 2.4 will then expand upon the practical details of these concepts.

Similar to PalCom assemblies, every PML description must specify which PalCom devices, services, commands and parameters are to be used by the application. This is done by defining PML components that through it's class (device, service, etc.) and properties uniquely identify a PalCom component. In doing

so, the PML component can then be assigned to handle all communication the application is to have with that specific PalCom component.

The graphical content of the application's GUI is specified by selecting from a by the language predefined suite of classes for graphical components. The selection is performed by specifying a number of PML components that belong to one of the classes. The resulting components represent generic graphical components, i.e. text boxes and buttons, in a platform independent manner. Graphical details can be adjusted by setting properties of the PML component. The structure of the GUI is defined by nestling compatible components within each other, e.g. a button inside a window. As mentioned in section 2.1 the complexity of PML is initially to be kept low. Regarding the discussion of expressiveness vs. simplicity of UIDLs in section 1, this means that PML will focus on simplicity, by toning down the graphical expressiveness of the language. The limited suite of available classes and properties for graphical components is a consequence of this. However, as will be shown in chapter III, the language is more than capable of handling a wide array of needs as is, and will only grow more competent with time.

As discussed in section 1, it is the ambition of the planned graphical editor for PML to turn the conventional way of defining application functionality 180°, by linking functionality to graphical components instead of the other way around. To support this novel approach of specifying functionality, PML defines behavior by assigning values to a number of *behavior properties* for PML components. These properties *link* one component of the description to another, giving the link a specific *role*. Which components can be linked, as well as which roles are acceptable for the link depends on the classes of the linked components. The four roles for links are *invoker*, *reactor*, *viewer* and *provider*

The role of **invoker** is used when one PML component (source) is to trigger, or *invoke*, an *action* at another component (target). The role is paired with a value – the *qualifier* – that indicates which *event* at the source component starts the process; which action is triggered at the target component depends on the target component's class specification. In fig. 2.3, the PML component for a PalCom command will cause the command to be sent to it's PalCom service when invoked. In the example, this will happen when a button in the GUI, as represented by a PML component, is clicked by the end-user. The role of **reactor** is the inverse to that of the invoker role.

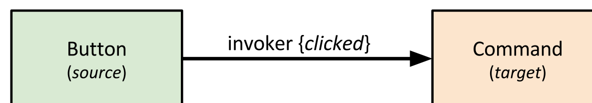


Figure 2.3: Example of PML components linked with the role of *invoker*

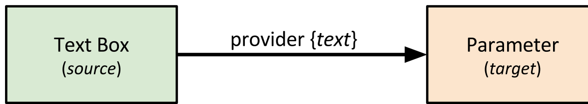


Figure 2.4: Example of PML components linked with the role of *provider*

The role of **provider** is used when a property value of one PML component (source) is to be assigned as the new value for a property of another component (target). The qualifier of the role indicates which property of the source is to be used for the assignment. When the assignment should happen, i.e. when the link is activated, depends on the source components's class specification. Similarly, the property value that should be set for the target component depends on it's class specification. In fig. 2.4, the PML component for a PalCom parameter will set the value of the parameter to the text of a text box, as entered by the end-user in the GUI. In this example, this will happen every time the end-user edits the text of the text box. The role of **viewer** is the inverse to that of the provider role.

These simple, yet powerful behavior properties allow the user to specify the functionality of applications almost exclusively in terms of PalCom communication.

2.3 Description Overview

The code in PML descriptions is formatted according to standard XML (eXtensible Markup Language) specification. The concepts of PML, as introduced in section 2.2, are realized in the language by dividing the structure of all descriptions into six separate blocks: *universe*, *discovery*, *structure*, *style*, *logic* and *behavior*. These blocks, as well as other structural basics of PML will be introduced here. Section 2.4 will dive into more language details from the perspective of the six blocks.

The *units* of a description are defined in the **universe** block. Units represent the PalCom components (devices, services, etc.) that can be used in the description. The universe block is supplemented by the discovery and behavior blocks. The **discovery** block contains *properties* that identify a unit's PalCom component. The **behavior** block contains properties that specify the behavior of a unit.

In the **structure** block, the *parts* of a description are defined. Parts represent the graphical components that make up the GUI that will be presented to the end-user. The block is supplemented by the style and behavior block. The **style** block contains properties that affect the visual characteristics of parts. Like for units, the behavior block contains properties that specify the behavior of parts.

Facts are defined in the **logic** block of the description. Facts represent logical components (constants, variables) that can be used in the description. The logic

```
<puiml>
  <universe>
    <unit/>
  </universe>
  <discovery>
    <property/>
  </discovery>

  <structure>
    <part/>
  </structure>
  <style>
    <property/>
  </style>

  <logic>
    <fact/>
  </logic>

  <behavior>
    <property/>
  </behavior>
</puiml>
```

Listing 2.1: Overview of PML description, showing the six blocks

block is supplemented by the behavior block. Same as for units and parts, behavior properties can also apply to facts.

Listing 2.1 illustrates the structure of a PML description, including the six blocks with one component/property each. Note that this is not proper PML code, and only serve to illustrate the structure.

In PML, all components are represented by an XML *element* with the same tag name: ”unit”, ”part” or ”fact”. These elements have two XML *attributes*: ”id”, a string that uniquely identifies the component in the context of the description, and ”class”, which specifies what type of component is being represented. Depending on the class of the component, its element may contain other *nested* components. These concepts are illustrated in listing 2.2.

Discovery, style and behavior properties are represented by XML elements with the tag name ”property”. All property elements must have the ”name” attribute set to specify which property is being set for the PML component in question. A value is assigned to the selected property by editing the *inner text* of the property element. In order to successfully link to property to an existing component, one of

```
<puiml>
  <universe>
    <unit id="device1" class="P:Device">
      <unit id="service1" class="P:service"/>
    </unit>
  </universe>
  <!-- omissions -->
</puiml>
```

Listing 2.2: PML component elements – attributes and component *nesting*

```xml
<puiml>
  <universe>
    <unit id="device1" class="P:Device">
      <discovery>
        <property name="p:id">C:30e8f8a2</property>
      </discovery>
    </unit>
    <unit id="device2" class="P:Device"/>
  </universe>
  <discovery>
    <property unit-name="device2" name="p:id">C:30e8f8a2</property>
  </discovery>
  <!-- omissions -->
</puiml>
```

Listing 2.3: PML property elements – *local* (line 5) and *global* (line 11)

the attributes ”unit-name”, ”part-name” or ”fact-name” must also be set, depending on what sort of component the property should apply to. The attribute value should match that of a previously defined PML component. Listing 2.3 illustrates this by setting the ”p:id” property of a unit ”device2” to ”C:30e8f8a2” on line 11.

As an alternative way of structuring PML descriptions, one can choose to specify a component’s properties directly in its definition, rather than in one of the *global* properties blocks. To do this, a *local* properties block is created inside the component’s element. In listing 2.3, this concept of local properties is illustrated by setting the ”p:id” property for unit ”device1” on line 5. Notice how contrary to for global properties, no unit/part/fact-name attribute has to be specified for local properties; the property element is nested inside the component element, implicitly specifying which component the property belongs to. Both ways of structuring descriptions yield the same end result; it’s a matter personal preference which should be used. Defining properties locally can be useful for smaller descriptions in order to have all information pertaining to a PML component in one place. For larger descriptions, defining properties globally will allow for a quicker overview of the description’s structure, while keeping the details provided by the properties in one collective place.

2.4 Description Details

This section will go into greater detail regarding how PML descriptions are created, and expands upon the purpose of each block of a description.

2.4.1 Example Application

The PML described application shown in fig. 2.5 will be used as a continuous example for the upcoming sections. To demonstrate what each PML block might look like for a given application, the block of code that makes up the application in fig. 2.5 will be presented in its respective section; the complete, unabbreviated code can be found in appendix 1 (A).

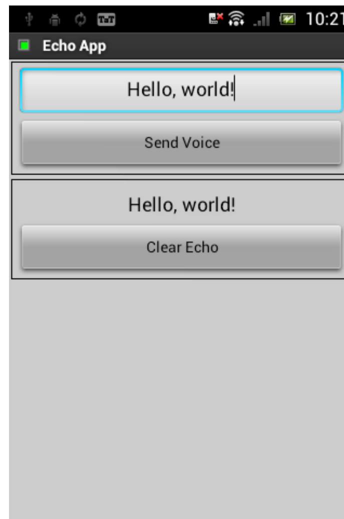


Figure 2.5: Demonstration example of PML described application

The application in fig. 2.5 is a mere toy example, serving only to demonstrate PML. It connects to a PalCom service – `SimpleEchoService` – that echoes all messages sent to it back to the sender. The service has one input command, VOICE, and one output command ECHO. Both carry a parameter containing the echo message. In the application, the user can type freely in the uppermost text box. By pressing the button labeled ”Send Voice”, VOICE with the content of the text box as its message is sent to the echo service. When received, `SimpleEchoService` will respond by sending ECHO containing the same message back to the application. Upon reception, the application display the content of the message parameter (”Hello, world!”). Pressing the button labeled ”Clear Echo” clears this message, resetting it to empty.

2.4.2 Universe Block

In the universe block, the units of the PML description are declared. Units represent PalCom components, hence there is only four classes that units can belong to; table 2.2 outlines theses classes.

P:Device	PalCom Device
P:Service	PalCom Service
P:Command	PalCom Command
P:Param	PalCom Parameter

Table 2.2: Valid classes for PML units

```
1  <universe>
2    <unit id="devHost" class="P:Device">
3      <unit id="svcEcho" class="P:Service">
4        <unit id="cmdVoice" class="P:Command">
5          <unit id="prmVoiceMsg" class="P:Param"/>
6        </unit>
7        <unit id="cmdEcho" class="P:Command">
8          <unit id="prmEchoMsg" class="P:Param"/>
9        </unit>
10     </unit>
11   </unit>
12 </universe>
```

Listing 2.4: Universe block of example application (fig. 2.5)

Unit elements can contain other unit elements – units can be nested. How units can be nested depends on the units class. The universe block can contain an unbound number of units, but only units of class ”P:Device”. Units of class ”P:Device” can in turn only contain units of class ”P:Service”, since a device cannot contain e.g. a command without it belonging to a service first. For analogous reasons, ”P:Service” can only nest ”P:Command”, which in turn only nests ”P:Param”.

It is important to note that how units are nested affects how they will be located in the PalCom universe. For example, when trying to locate a PalCom service, the discovery properties of the unit describing said service will be used. However, these alone are not enough to identify the service. The discovery properties of the parent unit describing the PalCom device the service is hosted on must be used as well. This means that the same unit could be used to describe different services just by being nested in a different (device) unit.

Listing 2.4 shows the universe block of the PML description that describes the application in fig. 2.5. Notice that this code gives no details as to the identity of the PalCom components to be used, only the relation between those component: one device hosting one service, which has two commands with one parameter each.

2.4.3 Discovery Block

In the discovery block, the discovery properties of the PML description are declared. Discovery properties apply to the units declared in the universe block, and are mainly used to identify the units' corresponding PalCom component, i.e. how they can be located in the PalCom universe. The class of the unit determines which discovery properties are available to set. Some of these properties are required and must be specified, while others are optional and may be left out for an implicit default value.

The discovery block of the PML description that describes the application in fig. 2.5 is presented in listing 2.5. Line 2 identifies the device hosting `SimpleEchoService`, which is in turn identified by the discovery properties on lines 3–8. VOICE, ECHO and their respective message parameter are identified on lines 9–14.

```
1  <discovery>
2    <property unit-name="devHost" name="p:id">C:30e8f8a2-2ae5-455b-abe3-287
         d3a2015cd</property>
3    <property unit-name="svcEcho" name="p:required">true</property>
4    <property unit-name="svcEcho" name="p:instance">1</property>
5    <property unit-name="svcEcho" name="p:cdid">X:SimpleEchoService</property
         >
6    <property unit-name="svcEcho" name="p:cn">BAJ1</property>
7    <property unit-name="svcEcho" name="p:udid">X:SimpleEchoService</property
         >
8    <property unit-name="svcEcho" name="p:un">BAJ1</property>
9    <property unit-name="cmdVoice" name="p:id">Voice</property>
10   <property unit-name="cmdVoice" name="p:direction">in</property>
11   <property unit-name="prmVoiceMsg" name="p:id">message</property>
12   <property unit-name="cmdEcho" name="p:id">Echo</property>
13   <property unit-name="cmdEcho" name="p:direction">out</property>
14   <property unit-name="prmEchoMsg" name="p:id">message</property>
15 </discovery>
```

Listing 2.5: Discovery block of example application (fig. 2.5)

2.4.4 Structure Block

In the structure block, the parts of the PML description are declared. Parts represent the graphical components of the GUI that will be presented to the end-user. Parts can belong to one of several classes; table 2.3 outlines these classes.

The structure element can and must hold one, and only one, part element of class ”G:Application”. This class represent the resulting application inside the description, and must be the first part defined in any description. Similar to units, parts can be nested, i.e. part elements can contain other part elements. The nesting of parts is used to logically structure the application GUI; the way in which they may be nested depends on the class of the part. In principle, only parts of container classes (containers) can contain other parts. As an example, a button (part of class ”G:Button”) can logically reside within a window (part of container class ”G:Window”), but a button can not logically reside within another button.

It is through the logical nesting of parts, in conjunction with the layout related style properties provided for container parts, that the application GUI gets its structure. Layout properties define how graphical components should be laid out within the container component, e.g. from left to right, or from top to bottom. To properly structure the graphical components of a PML application GUI, there are three things to consider:

1. The nesting of parts decides in which container component the resulting component should be placed.
2. The order in which the parts are declared decides the order in which the resulting component will be laid out within the container component.
3. The specified layout related style properties of the container part decide the formation in which the resulting components should be laid out.

Class		Description
G:Application	○	Logical component that represents the application itself internally
G:Window	●	Top-level, self-contained, structural component. May contain other components
G:Area	●	Structural component. Primary purpose is to contain and layout other components
G:Tabbed	●	Structural component. Contained components are represented as tabs, and displayed after selection
G:RadioGroup	○	Logical component that groups radio buttons
G:RadioButton		Clickable, two-state component. One instance per group can be checked
G:Label		Component for displaying simple text
G:Button		Clickable component
G:CheckBox		Clickable, two-state component. Multiple instances can be checked
G:TextField		Component for text input
G:Image		Component for displaying images
G:TextArea		Component for displaying text (advanced)
G:NumberSlider		Component with horizontally slidable handle. Selects numeric value in given range
G:DropDownList		Compact (expandable) component for item selection from a list
G:SystemNotification		System wide notification. Visible even when application is not
G:YesNoDialog		Dialog box that blocks the rest of the application until confirmed/dismissed
G:QuickNote		Temporarily visible text message

Table 2.3: Valid classes for PML parts

○ = *limited* container, ● = *full* container

```
1  <structure>
2    <part id="appDemo" class="G:Application">
3      <part id="winMain" class="G:Window">
4        <part id="panTop" class="G:Area">
5          <part id="txtVoice" class="G:TextField"/>
6          <part id="btnSend" class="G:Button"/>
7        </part>
8        <part id="panBottom" class="G:Area">
9          <part id="lblEcho"class="G:Label"/>
10         <part id="btnClear" class="G:Button"/>
11       </part>
12     </part>
13   </part>
14 </structure>
```

Listing 2.6: Structure block of example application (fig. 2.5)

Listing 2.6 shows the structure block of the PML description that describes the application in fig. 2.5. Notice that this code only specifies two out of three items from the list above; item 3 is specified in the style block. However, the structure seen in fig. 2.5 is observable: an application with a window (lines 2–3) divided into two separate sections, one containing a text box and a button (lines 4–7), and another containing a label and a button (lines 8–11).

2.4.5 The Style Block

In the style block, the style properties of the PML description are declared. Style properties apply to the parts declared in the structure block. They are used to specify what a part's corresponding graphical component should look like in the resulting application GUI; what properties are available to set depends on the class of the part. For example, container parts provide several layout related properties, such as component arrangement and padding, while text input/output parts offer several font related properties, such as font size and color. Some style properties are compulsory and must be given a value, while others are optional and may be omitted for an implicit default value.

```
1  <style>
2    <property part-name="winMain" name="g:title">Echo App</property>
3    <property part-name="winMain" name="g:layout">linear</property>
4    <property part-name="winMain" name="g:layout-orientation">vertical</
         property>
5    <property part-name="panTop" name="g:border">line</property>
6    <property part-name="panTop" name="g:layout-orientation">vertical</
         property>
7    <property part-name="txtVoice" name="g:align-h">center</property>
8    <property part-name="btnSend" name="g:text">Send Voice</property>
9    <property part-name="panBottom" name="g:border">line</property>
10   <property part-name="panBottom" name="g:layout-orientation">vertical</
         property>
11   <property part-name="lblEcho" name="g:align-h">center</property>
12   <property part-name="lblEcho" name="g:align-v">center</property>
13   <property part-name="lblEcho" name="g:font-size">18</property>
14   <property part-name="lblEcho" name="g:size">-1,75</property>
15   <property part-name="btnClear" name="g:text">Clear Echo</property>
16 </style>
```

Listing 2.7: Style block of example application (fig. 2.5)

The style block of the PML description that describes the application in fig. 2.5 is presented in listing 2.7. Notice how the layout formation of sub-parts (item 3, section 2.4.4) is specified for container parts. On lines 3–4, for example, the main window of the application (*winMain*) is set to arrange sub-parts in a *linear*, *horizontal* fashion, i.e. one after another from top to bottom.

2.4.6 The Logic Block

In the logic block, the facts of the PML description are declared. Facts represent logical components. There are at present only two classes for facts, as outlined in table 2.4. Future updates will likely include more classes for simple logical constructs, e.g. conditional expressions (*if–then–else*).

G:Constant	Holds a single, constant value
G:Variable	Holds a single, changeable value

Table 2.4: Valid classes for PML facts

Listing 2.8 shows the logic block of the PML description that describes the application in fig. 2.5.

```
1 <logic>
2   <fact id="cstClear" class="G:Constant"/>
3 </logic>
```

Listing 2.8: Logic block of example application (fig. 2.5)

2.4.7 The Behavior Block

In the behavior block, the behavior properties of the PML description are declared. Behavior properties apply to the units, parts and facts declared in the universe, structure and logic block. They are used to specify the behavior of the application. As mentioned in section 2.2, this is done by creating links between two PML components, and assigning a role to that link. This is mainly how behavior properties are used, but there are exceptions, e.g. properties to specify how incoming data should be parsed. Similar to discovery and style properties, what behavior properties are available to set depends on the class of the component. Some behavior properties are required while others are optional.

The behavior block of the PML description that describes the application in fig. 2.5 is presented in listing 2.9. On line 2, the application is instructed to open the main window (*winMain*) when it starts. On line 3, the text box (*txtVoice*) is set to provide its text value to the message parameter of the outbound echo command. This way, when the command is sent, its parameter will already be set the latest value entered by the end-user in the text box. The send button is linked as invoker to the command (line 4); whenever the button is clicked, VOICE is sent to the echo

```
1  <behavior>
2    <property part-name="appDemo" name="p:invoker" event="loaded">winMain</
         property>
3    <property part-name="txtVoice" name="p:provider" get="text">prmVoiceMsg</
         property>
4    <property part-name="btnSend" name="p:invoker" event="clicked">cmdVoice</
         property>
5    <property part-name="lblEcho" name="p:viewer" set="text" order="0">
         prmEchoMsg</property>
6    <property part-name="lblEcho" name="p:viewer" set="text" order="1">
         cstClear</property>
7    <property part-name="btnClear" name="p:invoker" event="clicked">cstClear<
         /property>
8    <property fact-name="cstClear" name="p:value"></property>
9  </behavior>
```

Listing 2.9: Behavior block of example application (fig. 2.5)

service. To capture the response from the service, the label (*lblEcho*) is linked as viewer to the message parameter of the inbound echo command (line 5). Whenever the command is received, the value of its parameter will be displayed in the label. *lblEcho* is also set as viewer for the the constant *cstClear* (line 6), the value of which is set to empty on line 8. The clear button (*btnClear*) is linked as invoker to the constant (line 7). This means that whenever the clear button is clicked, the value of *cstClear* is propagated to its viewers, hence clearing the echo message label.

2.5 Description Interpreters

A *PalCom User Interface Description Interpreter*, also simply referred to as an interpreter, is used to produce a fully workable application from a PML description. Interpreters are wrapper applications that interpret PML descriptions, and thereafter host the resulting application GUI (Graphical User Interface). As input, interpreters are fed a description, and as output they provide the end-user with the GUI of the application as specified in the provided description. Interpreters are made up of two main parts: a **front end** and a **back end**.

A simplified overview of how the process of generating an application from a PML description works is illustrated in fig. 2.6. A description is passed to the interpreter, which as mentioned is the wrapper application that will host the generated product. In the interpreter, the input file containing the description is processed by a *front end*. The front end parses the file content and evaluates if it constitutes as valid PML code. Upon success, this results is an internal *model* detailing the structure of the to-be application, as specified by the input description. This model is made up of objects that represent units, parts and facts. It is the internal representation of the provided description, and contains all information regarding what the application should look like and how it should act. The front end and the description model are platform independent, and can hence be reused for implementations of interpreters for different platforms.

After successfully being produced by the front end, the model is passed on to the interpreter's *back end*. The back end analyses the objects of the model that represent parts and interpretes them as graphical components, e.g. text boxes or buttons. The end result is the GUI that will be presented to the end-user. The back end creates the GUI through the use of a number of callback methods that all back ends must implement. The back end is platform dependent, and is the part of the interpreter that decides on the look and feel of all GUI:s that it generates. Hence, it will not be reusable to any great extent between implementations, and constitutes the major workload when creating a new interpreter.

The model is also passed on to the interpreter's *connector*. In the connector, the information held by the objects of the model that represent units are used to establish the necessary PalCom connections for the application. These connections are established once upon starting the interpreter, and are then maintained by the connector. The result of running the model through the connector is that the unit objects get connected and can effectively communicate through PalCom; the model transforms into a *connected model*. The connector is an extension of the front end, and can as such be reused between interpreter implementations.

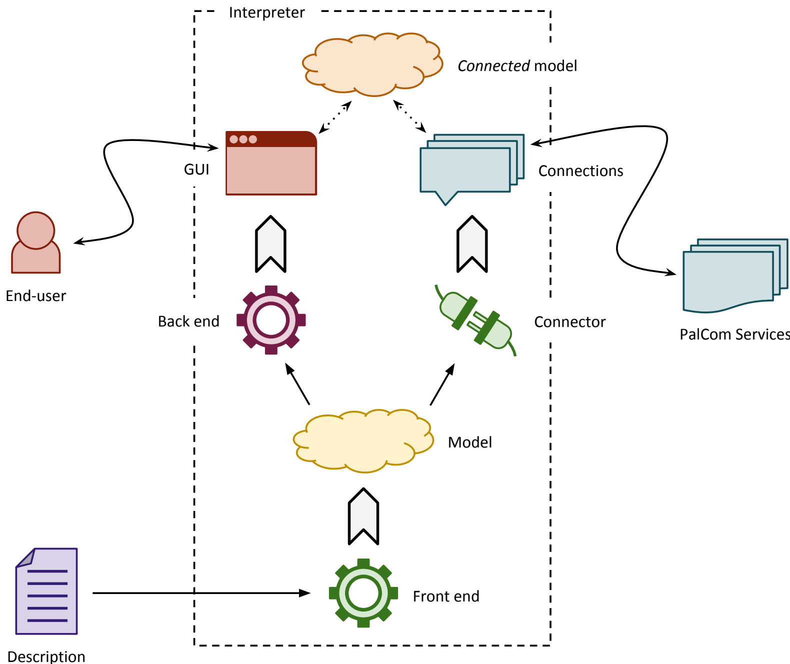


Figure 2.6: PML description interpretation process

After providing the model to both the back end and the connector, the interpreter has completed the process of generating an application from the given PML description. At this stage, the GUI as output by the back end is presented to the end-user. The end-user interacts with it like he/she would with any other application. Through the now connected model, the GUI can communicate with the specified PalCom services. The part objects of the model react to input from the end-user through the GUI. Since these objects are linked to other objects in the model, as specified in the PML description, the expected behavior can then be exercised. By invoking unit objects, the actual communication with the PalCom services is performed through the connections established by the connector.

PML provides a platform independent manner of describing graphical solutions for PalCom systems. Interpreters are used to transform PML descriptions into fully workable applications that can be operated by the end-user. While some parts of the interpreters are platform independent and can be reused in all implementations, ultimately a platform specific interpreter has to be implemented for each platform that PML is to be applicable for.

3 New Tools

The many end-user nodes of the PalCom-based itACiH system was one of the main motivator for creating the PalCom specific UIDL that is PML. To enable PML to actually be used in the itACiH project, a few new tools have been created as part of the prototype solution. These tools were also essential to the evaluation process of this work. Included in the new toolset is a reference implementation of a PML interpreter for Android OS, and a mobile adaption of TheThing for the same.

3.1 AndroidPUIDI

The Android application *AndroidPUIDI*, or *APu* for short, is one of two PML interpreter reference implementations that was created as part of this work. The other is a desktop version based on *Swing* (Java). In the itACiH project, the mobile team of nurses that visit the homes of patients was selected as in introductory point for PML. Their mobile method of working required a solution that supports mobile devices, in this case Android tablets. Because of this, heavy emphasis was put on the refinement of APu while the Swing-based interpreter, which will not be presented here, remains a mere reference implementation.

Upon startup, APu searches a description folder on the Android device's filesystem for available PML descriptions; the specific folder location can be changed by the user in the application's preference screen. If no descriptions are installed (present in the folder), the user is prompted to make an installation. Since APu is represented by a PalCom device that hosts a service for description installation, the user can do so remotely simply by interacting with this service,

e.g. using the PalcomBrowser. If multiple descriptions are installed, the user is prompted to select one, after which it is interpreted and the resulting GUI is presented to the user. To hide complexity and hence minimize the effort on the part of the user, a description is automatically loaded upon startup if it is the only one installed.

APu is built according to and follows the process laid out in section 2.5. It uses the common, platform independent front end to parse the selected description file, and implements the back end callback methods to create the Android specific graphical components that make up the GUI present to the user. The connector extension to the front end uses the same PalCom device that hosts the description installation service in order to establish and communicate with the PalCom components specified in the description.

3.2 TheAndroidThing

One of the main principles of PML is that no business logic should be directly specified in the descriptions created by the user. Such logic should instead be relegated to suitable PalCom services, which can then be accessed through the constructs of PML. During the development phase of the Android-based PML interpreter (APu), there was no way of hosting services on the Android device itself. Instead, the services needed for an application where deployed in a TheThing running on a centralized computer (server). The application could then access the services from anywhere by *tunneling* the PalCom device of APu to the device of TheThing on the server, using a specialized MAO (Media Abstraction Object). Since accessing the server was only possible when the Android device was online (WiFi, 3G, etc.), there was no way to provide offline functionality. This, of course, severely limited the usefulness of APu.

In order to remedy this issue, a mobile adaptation of TheThing, entitled *TheAndroidThing*, was developed as part of this work. TheAndroidThing works much like its desktop counterpart. Upon startup, the application gets representation in the PalCom universe in the form of a PalCom device. The device connects to other devices though one or more MAOs. In the application, the user can deploy custom service and assemblies, either by loading from files on the Android device's filesystem, or remotely though installation services hosted on the device of TheAndroidThing.

The introduction of TheAndroidThing made it possible to have services running locally on the Android device even when it was offline, enabling e.g. offline sensory data collecting. However, APu could still only access the services of TheAndroidThing by means of both applications tunneling to TheThing on the centralized server, where routing had to be enabled. This not only resulted in a hugely unnecessary communication overhead, but also did not enable offline functionality in the desired manner. To address this final problem a new, Android specific *Inter Process Communication* (IPC) MAO was developed. When instated

for the PalCom devices of both APu and TheAndroidThing, this MAO enables the two applications to communicate without the need of the centralized server.

By enabling PalCom services to be deployed locally on Android devices and providing a method for communicating to happen internally on those devices, TheAndroidThing is an essential complement to APu. In conjunction with it, the PML described applications created by APu can have access to the services needed for the application's functionality even when offline, thus fully realizing the potential of APu.

Summary

The processing of the problem description in section 1 concluded in a long-term aspiration of introducing a graphical editor for a PalCom specific UIDL, targeted at level 0 PalCom users, i.e. the same type of users that create PalCom assemblies today. PalCom runs on multiple platforms, and any graphical solutions must do so as well, hence the need for a platform independent UIDL as a basis. The introduction of level 0 users as the main target audience for the solution implies that no programming can be required when creating graphical solutions. In the long-term, it is the ambition of the planned graphical editor to enable this by turning the conventional way of defining application functionality 180°, linking functionality to graphical components instead of the other way around.

As a first step towards the long-term goal, the PalCom specific UIDL PML was introduced in section 2. While still requiring the user the write code when creation PML application descriptions, PML contains the necessary constructs to realize the ambition of inverted functionality specification. In the long-term upon the introduction of the graphical editor for PML, this will ensure that the user can specify an entire application, including its functionality, all without having to write any programming code.

The two new PalCom tools introduced in section 3 provide a practically sound manner of deploying applications described using PML on devices powered by the Android OS. These tools were essential to the introduction of a PML-based solution for the nurses of the itACiH project's mobile team. The same tools were also necessary for the upcoming evaluation of PML.

Chapter III

The Evaluation

This chapter will report on the findings of evaluating PML. In section 1, the efficiency of PML is evaluated for small scale solutions by comparing three different solutions (of which one is based on PML) to the same problem. The scalability of the language is evaluated in section 2 by presenting the PML described application developed for the mobile nurses of the itACiH project; the application is currently in active use in the project. This real-world usage resulted in some practical findings that are reported on in section 3.

1 Efficiency Comparison

It is not an easy task to determine how efficient a new programming language or style is. One approach is to do a user case study, where one group of test subjects solve a given problem using different but comparable alternatives, and measure the time it took them to solve the problem. Another group could then solve the same problem using the new technology, in this case PML. However, while this approach may be indicative of how efficient the new technology is, it is not perfect. What's being tested is more likely than not the individual level of skill of the subjects, rather than the efficiency of the new technology. There is no way to know how fast they would have solved the task, given the other technology.

Another approach would be to compare the amount of code needed to solve the same problem, using different technologies. This approach is less susceptible to variations in quality caused by the skill level of the producer of code, since the code can be studied and adjusted/optimized as needed. However, the validity of the comparison can be questioned. How good a measurement for technology efficiency is the number of *Lines of Code* (LOC)? This approach can be indicative of efficiency – a technology solving a problem in 100 LOC is probably more efficient than a technology requiring 10000 LOC. Still, the comparison is not perfect since a line can be long or short or even empty, it can be auto-generated, etc.

With the considerations above in mind, this section will make comparisons of development time and LOC for three solutions (using different techniques) to the

same problem. It is not a perfect comparison, but it should serve as an indicator of the efficiency of PML.

1.1 Application Solutions

The example problem that will be used in this comparison is based on a simple client-server network scenario: a client application connects to a server application, after which the client can send request to the server, and receive answers to those requests. The server never takes initiative to send messages.

The problem to be solved is kept minimalistic to keep development time down, and to avoid further complicating a situation where resources and code interweave between solutions. In the example, a medical professional, e.g. a nurse, is to be allowed to identify him/herself using a username and a pertaining password. Upon successful authentication, the user is to be presented with a list of available patients in his/her care. Selecting one such patient should show some piece(s) of live information about that patient, e.g. heart rate.

To solve this problem, two solutions to the server side of the problem were developed:

Solution S_A Based on an ad hoc network protocol, built specifically for this problem situation. As such, the solution is rigid and possibly extensive coding is necessary to adapt it to new situations. The server listens to network (UDP) trafic using two threads: one for discovery, where it simply replies to requests for it's IP address and port number, and one for receiving and answering command request from clients.

Solution S_P Based on PalCom. The server is implemented completely as a PalCom service, which is loaded into the TheThing on the server. As PalCom handles all network aspects of the solution, the server (service) is independent of the network situation, making it easily adaptable.

The commands accepted by the server solutions include: LOGIN, GETPATIENTLIST and GETDATA. The server replies with one of the following commands: LOGINSUCCEEDED, LOGINFAILED, PATIENTLIST or DATA. Since business logic is not the point of this example, the authentication process is faked in both solutions: any username is accepted, and so is any password. The only way that login fails if is no username is provided. For the same reason, the data (heart rate) values provided by the server are not real either; they're simply randomized values. Also, the patient list returned by the server is static, and the same for all users.

On the client side, three solutions were created:

Solution S_{AC} Based on the same ad hoc network protocol as S_A, used from within an custom Android application. Rigid solution, hard to adapt. Finds a server by broadcasting a specific UDP request.

Solution S_{PC} Based on PalCom. A custom Android application creates a PalCom device representing the application. A connection (like the ones created by PalCom assemblies) between the device and the server's PalCom service is then established, enabling the application to communicate with the server. The solution is easily adaptable to new network situations.

Solution S_{PP} Based on PalCom. Implemented as a PML description. Uses the PML interpreter application for Android (AndroidPUIDI) too automatically generate the solution's GUI (Graphical User Interface) and all necessary PalCom "paraphernalia". This generation of content makes it possible to dynamically change any part of the solution on the fly, with no need for a programming environment or re-compilation.

Table 3.1 summarizes the client solutions listed above, and fig. 3.1 (page 42) shows screenshots of the same: figs. 3.1a to 3.1c for S_{PP}, and figs. 3.1d to 3.1f for S_{AC} and S_{PC}. S_{AC} and S_{PC} share screenshots, since the graphical part of the two solution is the same. Notice how the GUIs are almost indistinguishable; the white background of the GUI in figs. 3.1d to 3.1f is merely to more easily distinguish the two visually.

When the application (all solutions) starts, it displays a loading screen while it attempts to find the server. If no server is found within a specific time limit, an error message is displayed. Otherwise, the login screen is displayed (a, d). There, the user may enter his/her credentials. Text boxes and buttons were selected for this example to demonstrate *simple* graphical components. The credentials are sent to the server by the push of a button and, if accepted, the data view screen (c, f) is displayed. Otherwise, a popup dialog displaying an error (b, e) is shown. The popup dialog was selected for this example to demonstrate *non-persistent* graphical components. In the data view, the user may select a patient from a list (as received from the server) and a (heart rate) value is displayed for that patient. The drop-down list was selected for this example to demonstrate a *complex* graphical component.

All solutions except for S_{PP} were developed using the Eclipse IDE (Interactive Development Environment), and are written in Java. The GUIs for the Android applications in S_{AC} and S_{PC} were created using Android's Graphical Layout Editor (GLE) in Eclipse. There is currently no editor tools available for PML, so S_{PP} was developed entirely from a text editor. No special care was given to optimize the code in any of the solutions. The solutions are well structured and

Solution	S_{AC}	S_{PC}	S_{PP}
Communication	Ad hoc UDP	PalCom	PalCom
Application (GUI)	Custom Android	Custom Android	PML

Table 3.1: Summary of client solutions

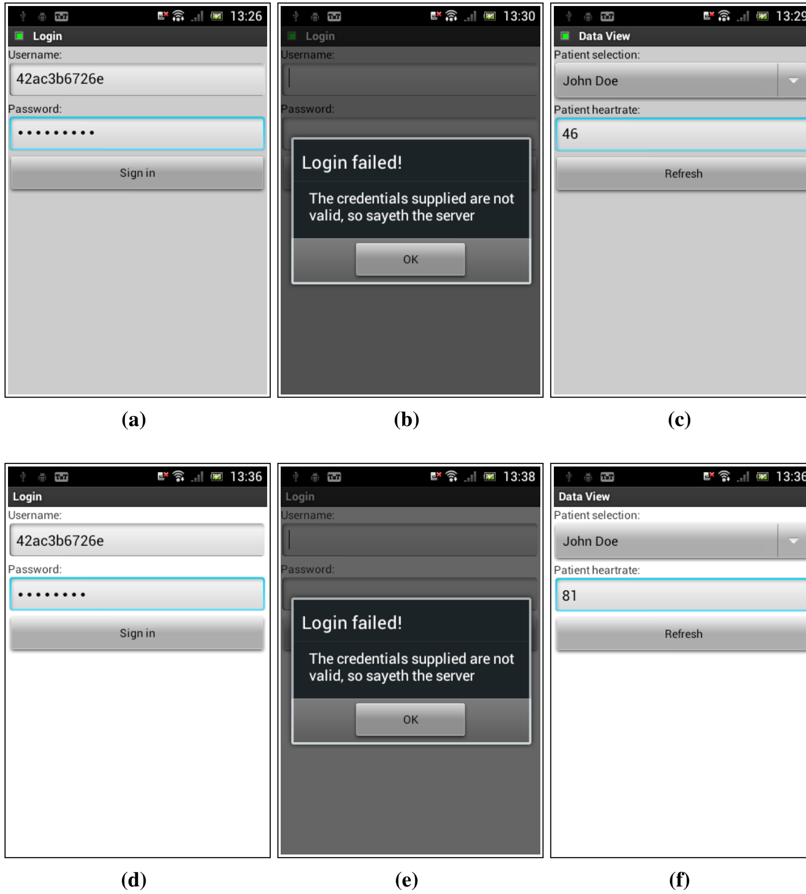


Figure 3.1: Comparison of screenshots for PML solution S_{PP} (a–c) and custom Android solutions S_{AC} and S_{PC} (d–f)

readable, but both the number of lines of code and performance could surely be optimized at the expense of more development time.

As a final note on development, the solutions were developed in the following order: S_P, S_{PP}, S_{PC}, S_A, S_{AC}. If anything, this affected the solutions later in the development process in a positive manner. With so many similarities in functionality, code and structure between the solutions, the experience with the previously developed client(s)/server can have contributed with faster/better development of the solutions later in the process.

1.2 Development Time

During the development of solutions S_{AC}, S_{PC} and S_{PP} notes where kept regarding development time (DT). Start and end times of development session where recorded with a resolution of 1 minute. Note that in the context of software development this fine granularity might have little meaning, as minutes can easily be lost due to e.g. distractions in the environment. Another point to consider is that of re-usage of code. S_{AC} and S_{PC} share common code, and since S_{PC} was developed before S_{AC}, the DT of the common code must be added to the total DT of the later. However, during the development of S_{PC} it wasn't clear exactly what code would ultimately be shared between solutions, making the time records lacking in some parts of this area. In such cases, qualified estimations have been used instead.

Table 3.2 reports on the findings of the development time study for the client side of the solutions. The total DT has been broken down into three sub-categories: *Graphics* refers the DT devoted to purely graphical aspects of the solution, e.g. the structure of GUI components and their properties; *Functionality* refers to the DT of all non-network related functionality of the solution, e.g. application initialization, GUI component action handlers, etc; *Network* refers to the DT of the communicative part of the solutions, i.e. the part that handles sending and receiving data over the network. Note that for S_{PP} all three columns have one and the same value, since PML handles all three as one.

Comparing the total development times, the PML solution (S_{PP}) was developed noticeably faster than the other two solution, requiring a factor of 10 less time to complete. This is remarkable, especially considering that the other two

Solution	**Graphics (min.)**	**Functionality (min.)**	**Network (min.)**	**Total (min.)**
S_{AC} (Ad hoc, Custom)	45	259	136	440
S_{PC} (PalCom, Custom)			33	337
S_{PP} (PalCom, PML)	44			44

Table 3.2: Development time for client solutions

solutions where developed using an IDE while S_{PP} was created in a mere text editor. The planned graphical editor for PML can therefore be expected to further lower DT of PML solutions.

Comparing the total DT of the two custom Android applications, S_{PC} using PalCom and S_{AC} using a naive, ad hoc protocol for network communication, the difference is not insignificant: about 31% more DT for S_{AC}. However, ignoring the common DT of column ”Graphics” and ”Functionality” in table 3.2 and focusing instead on what differentiates the two solution, i.e. PalCom, the difference is much more striking: using PalCom to handle network communication is roughly 4 times faster than starting from scratch, even for a trivial solution such as this.

Looking at table 3.3 similar numbers can be seen. Table 3.3 reports on the development time of the two server solutions: one for S_{PP} and S_{PC} using PalCom – S_P – and one for S_{AC} using the ad hoc network protocol – S_A. The columns are the same as in table 3.2, with the exception of the ”Graphics” column which has been dropped since the servers operate without visual representation. As with the case in table 3.2, the difference in total DT is observable between the PalCom and the non-PalCom solutions, but becomes most impressive when focusing on what differentiates the two solution. Comparing the values of the ”Network” column, it’s again shown that using PalCom to handle network communication is roughly 4 times faster than starting from scratch.

Even though these are good numbers, this simplistic client–server example doesn’t do PalCom justice. S_{AC} and S_A don’t handle nearly as many situations (e.g. device/network migration, automatic reconnection, device discovery, etc.) as their PalCom counterparts. Extending the ad hoc solution to handle all such situations would undoubtably increase the time difference between the solutions further. The same can be said for the network scenario. Had the scenario been more complex than a simple client–server setup the DT for the ad hoc solution would surely drift further away from the PalCom solutions.

As shown above, by using PalCom, development time can be significantly lowered in relation to a ”from scratch” ad hoc solution. Furthermore, when PalCom is supplemented with PML the development time is expected to be drastically lowered yet again.

Solution	**Functionality** (min.)	**Network** (min.)	**Sum** (min.)
S_A (Ad hoc)	60	127	187
S_P (PalCom)		30	90

Table 3.3: Development time for server solutions

1.3 Lines of Code

Lines of Code (LOC) will be used as a second metric to illustrate the complexity of the three solution S_{AC}, S_{PC} and S_{PP}, and hence the relative efficiency of PML. At the end of development, the LOC count for each solution was summarized. For consistency, *CLOC* [2] was used to count lines. CLOC is a tool designed to count LOC for a wide array of programming languages, including XML and Java as needed here. When counting lines CLOC excludes, inter alia, all empty lines and lines containing only comments. This makes for a more dependable comparison, as the developer's preferences and coding style will have less of an impact on LOC.

Table 3.4 presents the results of the LOC summation for the client side of the solutions. The total LOC count has been broken down in the same manner as table 3.2 was: "Graphics", "Functionality", and "Network". For more on the individual meaning of the subcategories, please revisit section 1.2.

Comparing the total LOC count, the PML solution (S_{PP}) is noticeably more lightweight than the other two solution. At 158 LOC, that's less than $\frac{1}{3}$ the amount of LOC needed in comparison to the alternatives. Only 74 lines ("Graphics") where automatically generated by Android's GLE; the remaining lines of S_{AC} and S_{PC} had to be produced manually by the developer. In contrast, all lines of S_{PP}, though fewer in numbers, hade to be produced manually. However, with the planned advent of a graphical editor for PML, all lines would instead be produced through the editor, with no need to enter a programming environment.

Comparing the total LOC count of the two custom solutions S_{PC} (PalCom) and S_{AC} (ad hoc network protocol), the difference is negligible: about 6% more lines for S_{AC}. However, in alignment with the results seen in table 3.2 the effect becomes more significant if the comparison between S_{AC} and S_{PC} is focused on what actually differentiates the two, namely PalCom. Comparing the entries of column "Network" in table 3.4, the non-PalCom solution is made up of roughly 36% more lines than the PalCom solution. This may not be a massive gain for PalCom, but considering the simplicity of the network scenario of the problem, and the fact that S_{AC} is not nearly as adaptable/robust as S_{PC}, the numbers are prominent.

Table 3.5 presents the results of the LOC summation for the two server solutions, S_P (PalCom) and S_A (ad hoc network protocol). The columns are the

<table>
<tr><th>Solution</th><th>Graphics (LOC)</th><th>Functionality (LOC)</th><th>Network (LOC)</th><th>Total (LOC)</th></tr>
<tr><td>S_{AC} (Ad hoc, Custom)</td><td rowspan="2">74</td><td rowspan="2">383</td><td>113</td><td>570</td></tr>
<tr><td>S_{PC} (PalCom, Custom)</td><td>83</td><td>540</td></tr>
<tr><td>S_{PP} (PalCom, PML)</td><td colspan="3">158</td><td>158</td></tr>
</table>

Table 3.4: Number of lines of code (LOC) for client solutions

Solution	Functionality (LOC)	Network (LOC)	Sum (LOC)
S_A (Ad hoc)	35	239	274
S_P (PalCom)		124	159

Table 3.5: Number of lines of code (LOC) for server solutions

same as in table 3.4 except for column "Graphics" which is not present in table 3.5 since the server solutions do not provide a GUI. On the server side, the difference between the PalCom and non-PalCom solution are more observable than for the client side, even in total LOC count: about 72% more lines when not using PalCom. Like before, focusing on the "Network" column magnifies this effect: almost half the amount of lines were needed when using PalCom to handle network communication.

As seen above, the solution using PML is made up of considerably fewer lines of code than the other two. PalCom in general reduces the LOC count of a networked solution, even though the effect on LOC count is not as noticeable as it is on development time.

1.4 Discussion

The data reported in tables 3.2 to 3.5, as discussed in sections 1.2 and 1.3, point to the fact that PML is relatively efficient at what it does, i.e. creating applications with GUIs for PalCom systems.

The data for number of lines of code (LOC) and the development time (DT) data must be considered in alignment with each other. On the client side (tables 3.2 and 3.4), the numbers for the PML solution (S_{PP}) are considerably lower than for the other two solution. Between the two non-PML solutions, it is clear when focusing on the differentiating part (network) that using PalCom results in a lower metric (DT/LOC). Because of the high overhead of a custom application, the total metric of the two solutions are however similar. In a more complex network scenario, where the network part of the solutions would require more attention, the difference in total metric can be expected to increase in favor of the PalCom solution. On the server side (tables 3.3 and 3.5), the results are similar. Comparing the network specific metrics shows that the PalCom solution is favorable, even in a simple client-server example such as this.

An interesting note is that even thought the data for the two different metrics are in alignment across all tables, the net gain is more noticeable when studying the development times of the different solutions and its parts. It would seem as though LOC and DT don't relate to each other in a completely linear fashion. It is outside the scope of this work, but it could be of interest to see if this trend scales.

As the study reported on here only involved a single developer[1] (author), one could argue about the results. However, a factor of 10 less DT for the PML solution (S_{PP}) is suggestive of the efficiency of PML. Even compensating for the developer's experience with PML, and the fact that a more routined network developer could decrease the total DT of the non-PalCom custom solution (S_{AC}) would not change the situation significantly. Since the network part of S_{AC} only makes up roughly 31% of the total DT, a more efficient network programmer could only decrease it by so much. Furthermore, S_{PP} was developed in a simple text editor; using the future graphical editor of PML would decrease the DT of such solutions further.

Contemplating the network DT of the two non-PML solutions, one could make a similar argument: the developer's experience with PalCom result in lower DT for the PalCom solution (S_{PC}) in this study, while a more skilled network programmer could lower the DT of the non-PalCom solution (S_{AC}), thus bridging the gap. However this argument is countered by the fact that, as mentioned before, the network scenario of the toy example presented in section 1.1 doesn't do PalCom justice. A more complex network scenario than the mere client-server scenario presented here would do little to complicate S_{PC} because of how PalCom handles the distribution of systems. Without PalCom this is would however require more complex constructs, and hence more DT. As a final note, S_{PC} handles far more network situations (such as network migration) than the relatively barebones S_{AC}, making the direct comparison of network DT made in section 1.2 if anything modest.

On the subject of application generators other than AndroidPUIDI, a more complete survey of available alternatives is warranted. Such tools could indeed lower both graphics and functionality DT (table 3.2), and hence total DT. However, since there is currently no such tool dedicated to PalCom, they would do little to address the network DT. That would still be subject to a PalCom vs. non-PalCom solution, as discussed above.

2 Scalability in itACiH

In section 1, PML was evaluated from a practical perspective and in small scale through a direct metrical comparison of independent but equivalent solutions to the same problem. In this section the focus will be on the scalability of PML. To evaluate if the technology is usable for solution of larger scale than e.g. those seen in section 1, PML was used to create the application for the nurses of the mobile teams in the itACiH project. The resulting application will be presented and commented here to showcase how well PML scales. Direct practical findings from the evaluation are presented in section 3.

[1] A bigger study is part of future work; see section 1.4 (IV) for further details

2.1 Methodology

The scalability evaluation of PML was performed in an empirical manner in the context of the itACiH project. The *itACiH app*, a special build of the PML interpreter for Android – AndroidPUIDI– was deployed on the tablets to be used by the medical staff. As the system grew, new functionality was naturally added to the system. This functionality was created in the form of PalCom services deployed on both the TheAndroidThing on the tablet itself and on the TheThing on the server. Using the itACiH app to present the functionality of the system created a situation where every new piece of functionality spawned new requirements to be evaluated from the perspective of PML. The new requirements were also provided with a natural, empirical test scenario for evaluation purposes.

2.2 System Overview

The mobile team of the itACiH project consists primarily of nurses, but to some degree also of other health professionals. Their job is to treat the patients in their homes; to check up on them, follow up on previous issues, deliver medication and other consumables, take measurements, etc. Through the project, this job has been facilitated by providing the nurses with Android tablets, and introducing the itACiH app on said tablets.

From the perspective of the mobile team, the itACiH system is divided into two primary sites: the Android *tablets* the nurses bring into the home of the patients, and the *server* where collected data is permanently stored and retrieved from. Figure 3.2 illustrated this simplified view of the system. On the tablets two collaborating Android applications are installed. One is the itACiH app itself. This is what the nurses explicitly start when they want to interact with the system. The other is TheAndroidThing, which is started automatically upon device boot up and runs in the background with no need for user interaction. Both applications are represented in the PalCom universe as their own PalCom device ("TheAndroidThing"/"itACiH", fig. 3.2), even though they're actually running on the same *physical* device ("Tablet", fig. 3.2). As the two devices are running on the same tablet, "itACiH" always has access to "TheAndroidThing". The same can not be said for the device on the server ("TheThing", fig. 3.2). Because access to "TheThing" depends on an active Internet connection, which may not always be available[2], "itACiH" must be able to operate in its absence, i.e. in *offline mode*.

A main principle of PML is that the business logic and the visual aspects of an application should be kept separate (section 2.2 [II]). Hence, no business logic is embedded in the itACiH app; it only displays information provided by the services of the system. The app connects to these services using a special connector service ("Connector", fig. 3.2) hosted on its device. For functionality that does not have to be available to the team in offline mode, the corresponding

[2] It is not uncommon for patients to live in rural parts of the country

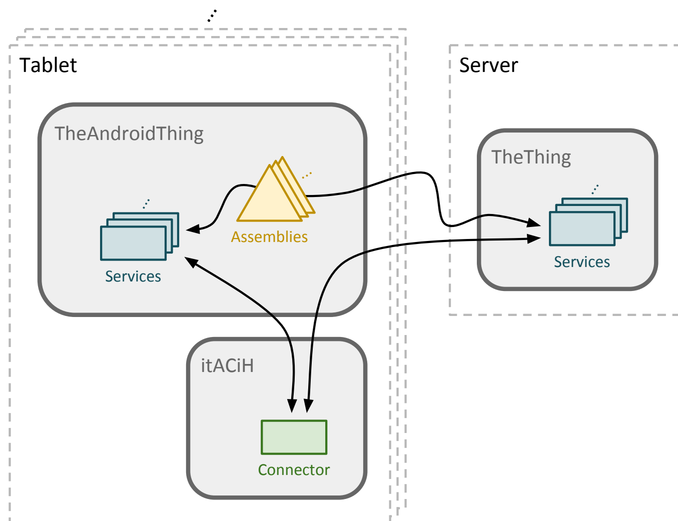


Figure 3.2: Simplified overview of the itACiH system from the perspective of the mobile team

services are deployed on the server. The app connects *directly* to such services. For functionality that must be available offline the corresponding services are instead deployed locally on the tablet's TheAndroidThing. To keep synchronized, such services are connected to services on the server's TheThing by assemblies, resulting in an *indirect* connection between the itACiH app the server's services. Examples of both offline-only and online/offline functionality and the services involved will be presented in the upcoming sections.

Finally, please note that the system overview in fig. 3.2 is a coarse-grained simplification of the reality. TheThing on the server appears to work in isolation, when in fact it connects its services (using assemblies) to services on other devices, establishes and maintains database connections, etc. In practice the mobile team's tablets don't only communicate with one server, but several. Also, fig. 3.2 only shows the system from the perspective of the mobile team; other sites that push/pull data to/from the server(s), e.g. the ward station (section 2.3 [I]), are not presented.

2.3 Application Overview

The itACiH application is divided into three screens: the *login screen* (section 2.4), the *main screen* (section 2.5), and the *locked screen* (section 2.6). The three screens and the transitional actions that take the user from one to another are illustrated in fig. 3.3.

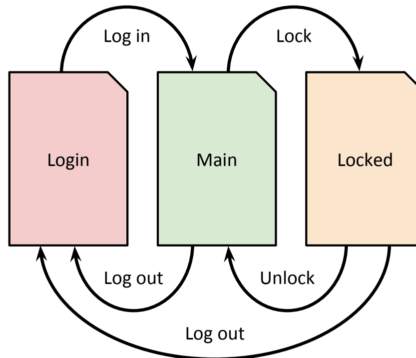


Figure 3.3: Overview of the screens in the itACiH app

This part of the solution illustrates PML's multi-screen capabilities. Allowing for multiple screens potentially reduces the amount of information necessary per screen, and hence greatly increases scalability. The screen transition feature is a fairly new addition to PML, and was added as a result of the requirements put on PML by the development of itACiH. This is a great example of how the needs of the area of application spawned new requirements. Thanks to the iterative development process, this lead to new constructs which could then themselves be evaluated. Being a new feature, screen transitions have yet to be applied to all suitable parts of the solution.

The current state of the tablet is managed by the locally deployed service `LoginService`. Since an attempt to log in can only succeed under certain circumstances (tablet at the ward) a session specific PIN is assigned when logging in. If the tablet is powered off or the itACiH app is otherwise closed, the user can resume the session by providing the correct PIN. This also provides a way to authenticate the use of a tablet in offline mode.

The screen that is shown to the user upon starting the itACiH app depends on the state of the tablet. If in state *locked* the locked screen should be displayed, and if in state *logged out* the login screen should be displayed. To handle situation like these PML allows for *actions* to be taken upon given *triggers*. In this case, a state query command is sent (action) to `LoginService` upon application startup (trigger). Using the same mechanism, screens are opened (action) upon reception (trigger) of a corresponding state command from `LoginService`. This is also how the transition between screens is handled in general.

2.4 Login Screen

Figure 3.4 shows the screen the user is presented with the first time he/she opens the itACiH app: the login screen. The user is prompted to enter his/her username

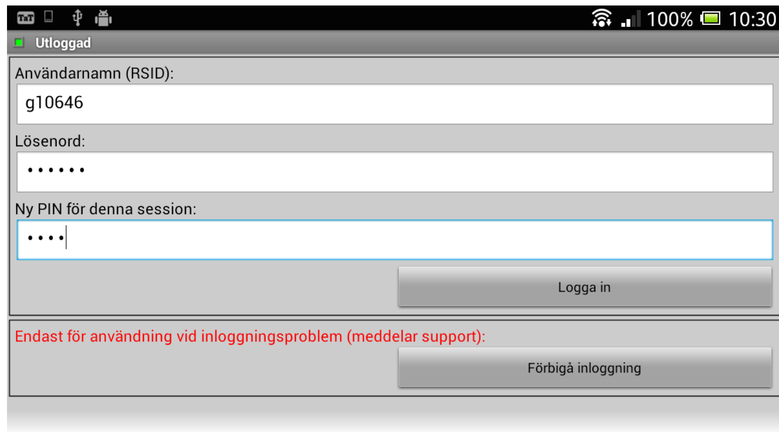


Figure 3.4: Login screen of the itACiH app

(*Användarnamn*), password (*Lösenord*), and a session specific PIN code (*Ny PIN*). Upon pressing the log in button (*Logga in*), the user's credentials are validated, and if valid the main screen (section 2.5) is presented. Otherwise, a dialog box (fig. 3.5) pops up and explains why the login failed (bad credentials, unacceptable PIN, etc.). The user may then adjust the input, and attempt to login again.

During the introduction phase of the login system, an option to bypass the login procedure (*Förbigå inloggning*) was added. This feature was added to prevent user from being locked out from the system due to (at the time) possible technical errors. It directly takes the user to the main screen without the need to enter credentials. To prevent misuse, an automatically generated text message (SMS) is sent to inform technical support of the transgression.

To validate login credentials, the server is indirectly queried through the locally deployed service `LoginService`. No access results in immediate failure. Text message are sent by using the local service `SmsService`.

Figure 3.5: Login failure notification dialog

Figure 3.6: Screenshot of the main screen of the itACiH app

From a graphical standpoint this screen is crude, but is serves to illustrate a few basic parts of PML:

- Labels that show static, plain text. Font, text color, etc. can be adjusted.
- Text boxes with optional hints that allow for user input. Font, sensitivity (asterisks, fig. 3.4), etc. can be adjusted. Provides data for command parameters.
- Buttons that trigger the sending of commands.
- Dialog boxes shown upon reception of given commands.

2.5 Main Screen

The main screen is (as hinted to by its name) the primary screen of the itACiH app, and where the nurses of the project actually contribute to and make use of patient information, pain assessments, etc. A full view of the first ”spread” of the main view is shown in fig. 3.6 (page 52). The fictive patient *Elizabeth Kammare* and her pretend data will be used throughout this section of the chapter to demonstrate a selection of the available features on the main screen.

2.5.1 Session Status

At the top of the main screen is the session status bar (fig. 3.7). It displays who is currently logged in (*Inloggad som*) and provides two buttons to navigate between the applications states. The lock button (*Lås*) locks the app for temporary inactivity. The log out button (*Logga ut*) terminates the session, requiring a new login. As mentioned in section 2.3, screen transition in PML is triggered by the reception of commands, in this case from `LoginService`.

From a PML perspective the text ”Inloggad som Staffan Lundborg” (swe. *Logged in as Staffan Lundborg*) is noteworthy. The text is actually made up of two text labels: one with the static text ”Inloggad␣som␣”, and one viewer label that displays one of the parameters of the login-accepted command from `LoginService`, in this case the fictive nurse ”Staffan␣Lundborg”. Here, this approach works well but the bigger problem of text formatting in PML is discussed in section 3.3.

Figure 3.7: Session status bar; logged in as Staffan Lundborg

2.5.2 Patient Selection

Figure 3.8 shows the main screen prior to any patient being selected. To select a patient, the user is presented with two *drop-down lists*: one to select a unit of care (*Välj avdelningsenhet*), and one to select a patient (*Välj patient*) at that unit. The lists are populated with values from a parameter of a command received from `LocalHomeIdService`. Since patients must be selectable in offline mode, the service is deployed locally on the tablet and is synchronized with the server whenever possible. The PML drop-down list is an example of a complex graphical part. The *display value* (patient name) of a selected item may differ from the *send value* (pseudo-anonymous ID). When an item (patient) is selected, a command can be sent containing the data of the send value, without the user ever seing it.

Comparing fig. 3.8 to fig. 3.6 demonstrates another feature of PML parts: the ability to *disable*. Since no patient is selected, interaction with the screen should not be allowed. Notice how the function-selection tabs are disabled (user can't switch tab), how the buttons are "greyed out" (can't be pressed), and how the text box does not accept input from the user. PML parts can be disabled e.g. by the reception of a specific parameter value, in this case from `LocalHomeIdService`. In situations like these, where a PML part has multiple properties that can be set (text, visibility, etc.), confusion around behavior link vagueness may occur. The subject is discussed in section 3.1.

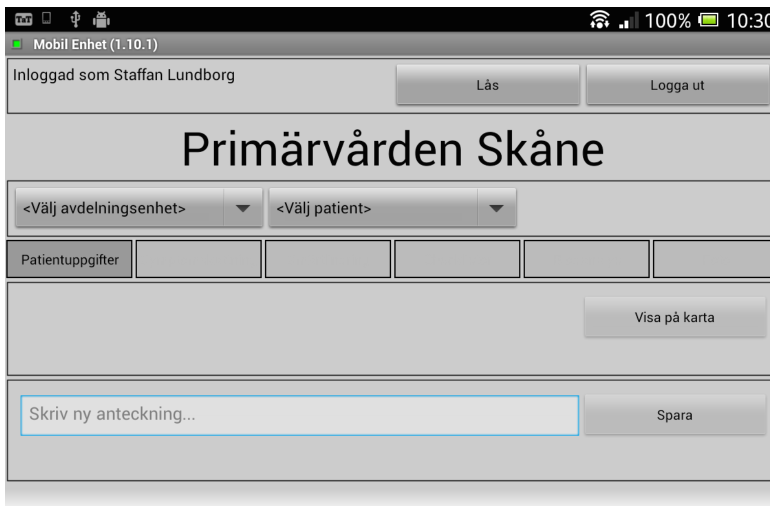


Figure 3.8: No patient selected; application "greyed out" (disabled)

2.5.3 Offline Mode

Since the itACiH app can function both on- and offline, it is important for the users to know which is the case at all times. To achieve this, a highly visible red text appears under the patient selection area, stating that the application is in offline mode (*Offline-läge*) when such is the case (fig. 3.9). This informs the user that their changes will not propagate to the rest of the system immediately, but rather as soon as contact with the server can be re-established.

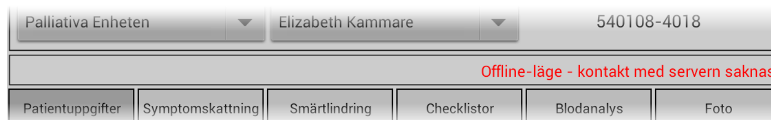


Figure 3.9: Server unavailable; application offline warning

Much like disabling parts, changing the *visibility* of parts is a highly useful feature of many PML parts. This functionality is used thoroughly throughout the application, mostly to hide functionality not available when offline, but also to avoid confusion in multi-step processes. The visibility of PML parts can be set from a any given parameter value. `LocalMobileTeamConnectorService` is a locally deployed service that, among other things, keeps an application level eye on the availability of the server. When the server appears/disappears a command with appropriate parameter values is sent to the itACiH app, either hiding or showing the offline warning text.

2.5.4 Patient Information

The first thing the user sees after selecting a patient is the patient information view (*Patientuppgifter*). Figure 3.10 shows the upper half of this view. Aligned to the left is information about the patient, e.g. personal identity number, address, phone numbers, etc. The data is received from the same service that populates the patient selection lists, `LocalHomeIdService`. To the right is a button for directions to the patients home (*Visa på karta*). Pressing it sends a command to `AddressIntentService` which, using the data from `LocalHomeIdService`, opens up the *Google Maps* app with navigation destination set to the patient's home address.

This view illustrates the use of a PML *text areas*. Text areas are similar to labels but e.g. allow for multiple rows of text. This opens up for some degree of content dynamicity: the first five entries are static for all patients, but the last entry (*Portkod*) is dynamic. For a different patient, there could be nine dynamic entries, or none. These entries are added from the ward station. The dynamic content is however limited to plain text. The problem with dynamic content in PML is discussed in section 3.5.

Figure 3.10: Information about the selected patient

2.5.5 Patient Notes

The bottom half of the patient information view houses the patient notes functionality (fig. 3.11). The user can write a note (*Skriv ny anteckning*) about the selected patient and save it (*Spara*). The new entry will instantly appear at the top of notes for the patient. Because the notes functionality is available offline, the new entry may appear in the list before it is stored on the server. To avoid confusion, such entries are clearly marked and are distinguishable from entries that can currently be seen on other tablets or the ward station. The locally deployed service `LocalChatService` buffers patient notes synchronized from the server (when available) for offline availability, and handles new notes.

Similar to the patient information, the dynamic text feature of the PML text area is used to display the note entries. Though not shown in either fig. 3.10 nor 3.11, the PML *layout* that houses the content of the patient information view is

Figure 3.11: Notes about the selected patient, entered by staff

marked as *scrollable*. If the content of a patient overfills the screen, the user scroll it by swiping with his/her finger.

2.5.6 ESAS

ESAS [16], or *Edmonton Symptom Assessment System*, is the first of three available assessment methods available in the assessment view (*Symptomskattning*). The creation part (*Skapa ny*) of the ESAS view can be seen in fig. 3.12. The patient, possibly with the help of the nurse, estimates their overall wellbeing by answering ten question with a floating number on a scale from 0 to 10. This view illustrates a new PML part, the *numeric slider*. The user slides the handle horizontally across the slider, filling it from left to right. The part's output is a value in a given interval to any given parameter. The slider was selected for this view because patients are expected to use this part of the app. The sliders are intuitive; filling up a bar to mean more of something by simply dragging on the screen should be understandable by most patients.

New assessments are sent to the locally deployed general purpose buffer service `LocalMobileTeamConnectorService`. Here, assessments (of varying type) are temporarily buffered until they can be sent to the corresponding service on the server, `MobileTeamConnectorService`. This is a multi-purpose service used to connect the mobile team to the rest of the itACiH system. In this case, ESAS assessments are converted into a database friendly format and stored permanently on the server. If in online mode, this happens in moments, and the user is notified through a PML *quick-note*, a self-dismissing message strip (Android *toast*). However, if in offline mode the assessment retains its place in the queue of outbound assessments, and is sent as soon as possible. To alert the user that the

Figure 3.12: Creation of a new *ESAS* assessment

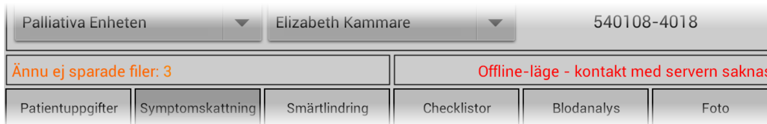


Figure 3.13: Buffering message; three items buffered

new assessment is not yet publicly available, buffered objects are notified through dialog boxes. To provide continues feedback while in offline mode, a buffering message appears next to the offline message (fig. 3.13), stating how many items are currently in the outbound queue (*Ännu ej sparade filer*).

There is a corresponding view for viewing the most recent ESAS, located under the neighboring view *Se senaste*. New ESAS assessments from any source are automatically pushed to and displayed in this view. Data is fetched directly from the server (`MobileTeamConnectorService`); when in offline mode, the view is empty.

2.5.7 HAD

HAD [29], or *Hospital Anxiety and Depression Scale*, is a self-assessment scale to be used by patients to assess their level of depression. Figure 3.14 shows part of the HAD creation view (*Skapa ny*). The patient answers 14 multiple-choice questions (4 alternatives) by pressing the *radio buttons*. Once saved on the server, the latest assessment can by viewed under *Se senaste*, where all answer are listed.

Figure 3.14: Creation of a new *HAD* assessment

The depression and anxiety scores that are added up from the value of the answered questions are also viewable here.

Other than the new PML part presented here, HAD assessments work much like ESAS assessments: outbound assessments are temporarily stored in `LocalMobileTeamConnectorService` before being sent to the server, and the latest HAD is (automatically) fetched directly from the server. User feedback is also as described in section 2.5.6.

2.5.8 Pain Analysis

An in-depth pain analysis is the final assessment methods available in the assessment view (*Symptomskattning*). See fig. 3.15. The users start by specifying which of several pains to work with, either by selecting an existing one (*Välj smärta*) or by defining a new one (*Lägg till ny smärta*). Pressing the new-pain button triggers the add-new-pain *input dialog box* to be shown (fig. 3.16). This is a more complex variant of the simple ”OK” dialog box seen previously. The user enters the name of the new pain (*Ryggsmärta*), which sets a parameter value of the add-new-pain command to be sent to a locally deployed service `LocalPainNameService`. Not all pain analysis functionality is available offline, but the pain name drop-down list must always be populated to allow for pain selection. The local service enables

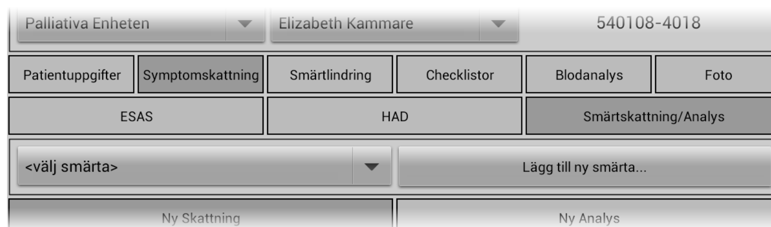


Figure 3.15: Selection of pain for analysis

Figure 3.16: Defining a new pain name in a *input dialog box*

this feature by buffering pain names for offline usage. Adding a new pain (*Lägg till*) automatically selects it in the drop-down list.

Creating a new pain analysis is a four step process. Figure 3.17 shows the second (*2*) of those four steps. The user is to draw with their finger on a figure to illustrate where on the body they are in pain. The drawing feature is of interest here because it was created as the need for it arose by altering the already existing PML part *image*. The alteration allows for optional drawing on images. The resulting line data can be used as value for any given parameter. The image can also present line data from the parameter data of a received command, enabling the loading of previous analyses. This part of the analysis serves to exemplify a reoccurring problem in PML, namely that of unpredictable future needs for parts. This problem is further discussed in section 3.2.

Like ESAS and HAD, pain analyses interact with the server indirectly through `LocalMobileTeamConnectorService` (buffering) and directly through `MobileTeamConnectorService` (fetch latest analysis). User feedback is also the same. However, unlike ESAS and HAD, their is no dedicated view for the latest analysis. Instead, the most recent analysis for a given pain name is optionally loaded and edited instead of creating a new analysis from scratch.

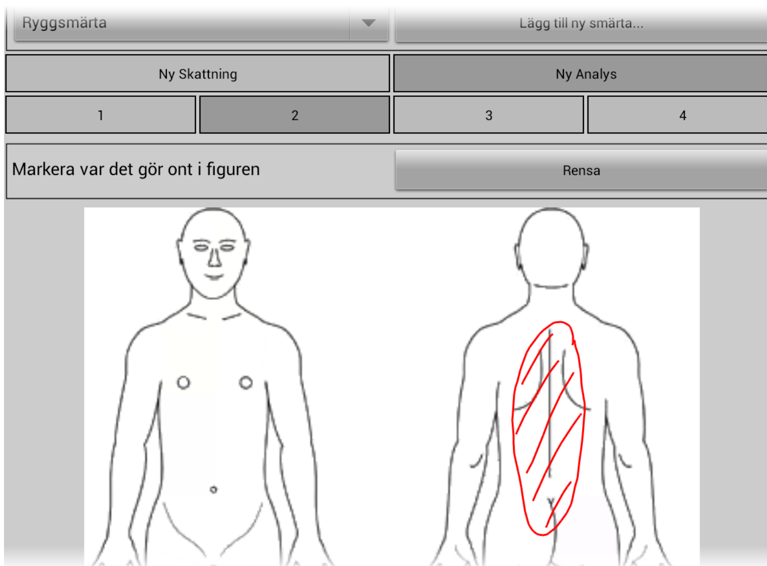


Figure 3.17: Pain analysis, step two; pain localization through drawing

2.5.9 Checklists

Remembering to do the right things in a given situation and keeping track of what has already been done is an integral part of the nurses everyday job. As a digital adaption to the previously used paper checklists, the itACiH app presents the user with five different checklists. Figure 3.18 shows part of one of them, used during weekly visits (*Veckobesök*) in the patients home.

Figure 3.18: *Checklist* for weekly visits; two items checked

To change a checklist, the user simply touches an empty *checkbox* to check it, or a checked box to un-check it. Contrary to e.g. ESAS assessments where the user presses a save button to store their changes, changing a checklist automatically saves it to the server. This works because the PML part for checkboxes can both set a parameter value and send a command when un/checked. In this case, parameters for a store-checklist command are set, and the command is then sent directly to the server (`MobileTeamConnectorService`). Due to the direct interaction with the server, checklists are not available in offline mode.

The checklists of the itACiH app are statically defined in the description file, one checkbox at a time. This situation is not ideal when changes, additions or retractions to the checklists are needed, as all description files on all tablets have to be exchanged. Instead, the checklist content should dynamically be pushed from the server to the tablets automatically. This yet again shows the need for dynamic content in PML (see section 3.5).

2.5.10 Photo

The photography feature of the itACiH app was a feature specifically request by the staff. Thanks to the iterative work process, and the close communication with the end-users, a first iteration of the refined solution shown in fig. 3.19 (page 62) was quickly put in the hands of the nurses. To take a photo, the user presses the

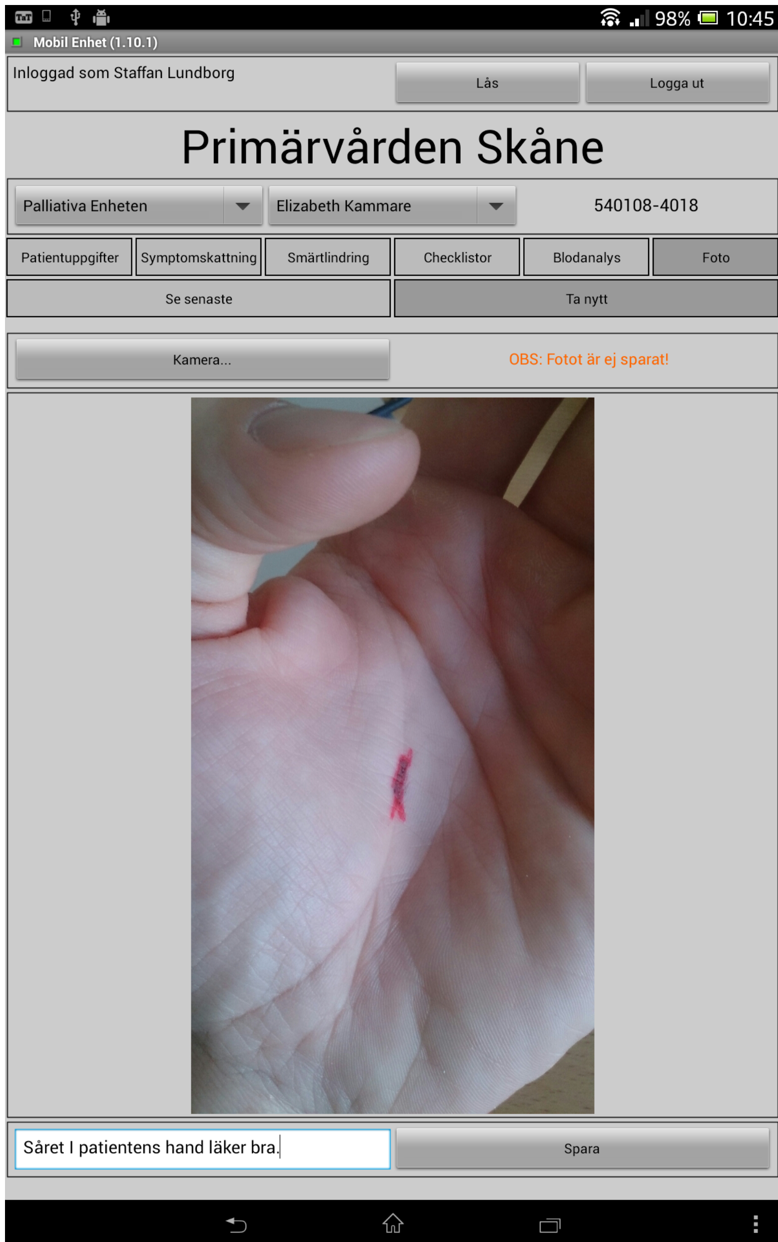


Figure 3.19: New photograph taken, but not yet saved

camera button (*Kamera*), which brings upp the (Android) device specific camera app. After snapping the picture, the user is returned to the itACiH app, where the photo is displayed for review. In this state, the user can enter a comment to the photo and press the save button (*Spara*) to save the photo. Prior to this state both those options where disabled, since no picture had been taken yet. To avoid confusion, an orange colored label warns the user that the photo is not saved (*Fotot är ej sparat*) directly upon returning from the camera app.

The PML point-of-interest in this example is the camera app button. Going against the PML design philosophy of model–view separation, the opening of the camera app is not packaged as a service but as a feature of the PML button class. This is an old exception in PML that originated in technical limitations, and will likely be removed eventually. Other parts from the project, e.g. the map feature in section 2.5.4, show how this feature should be designed.

The photo feature works in offline mode. Similarly to other solutions that do (e.g. ESAS), interaction with the server is indirect through `LocalMobileTeamConnectorService` for buffering outgoing photos, and direct through `MobileTeamConnectorService` to fetch photos. User feedback for buffering and storing items is also the same.

To save on bandwidth, the latest photo for a patient is not automatically pushed to all tablets; they must be explicitly requested by the user by pressing a refresh button. A problem that was discovered because of the relatively high transmission time of photos is that of receiving irrelevant commands out of place. If not handled properly on the service level, a photo could be requested for one patient and arrive only after the user switched to another patient. The problem is further discussed in section 3.4.

2.6 Locked Screen

Figure 3.20 shows the screen that is presented to the user when the system is locked. The screen clearly states who is currently logged in (*Inloggad som*). If there is a missmatch in users, e.g. if the previous user forgot to log out, the user may opt to log out (*Logga ut*) returning him/her to the login screen (section 2.4). Otherwise, the user is prompted to enter his/her session specific PIN code. Empty or incorrect PIN code entry will be reported in a dialog box upon pressing the unlock button (*Lås upp*). Else, the user is taken back to the main screen (section 2.5). The system gets locked in one of two ways:

1. The user presses the lock button (*Lås*) on the main screen (fig. 3.6)
2. The tablet's screen goes to into sleep mode, either automatically after a set amount of time or explicitly by the user pressing the physical power botton on the device

As mentioned in section 2.3, the locally deployed state service `LoginService` handles application state. Item 1 above is handled by simply having the lock

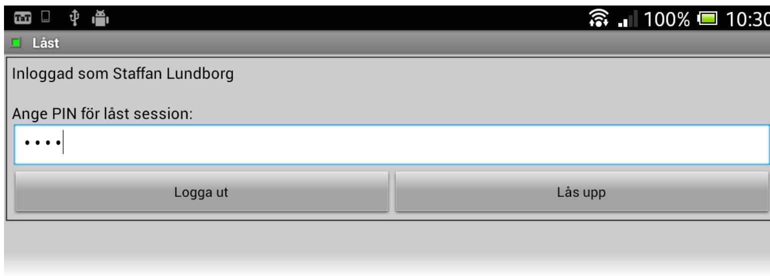


Figure 3.20: Locked screen of the itACiH app

button send a lock command to the state service. Item 2 makes use of another locally deployed service `ScreenStateService` that registers whenever the screen goes in to or out of sleep mode, and sends out corresponding on/off commands. An assembly connects `LoginService` and `ScreenStateService`, making the locking process automatic from the perspective of the user.

2.7 Discussion

The PML described itACiH application is at present being actively used by the nurses of the mobile teams in the project. This section has highlighted a few of the major features of the application, with the purpose of evaluating the scalability potential of PML. One could presume that due to the many self-imposed restrictions in graphical and behavioral expressiveness for PML, the language would only be useable for small scale applications, like the one seen in section 1. However, the conclusion from what has been shown here is that such is not the case. The itACiH application is by no means of small scale, and its wide range of partially advanced features indicate that PML is indeed a language that scales well.

Another presumption would be that even if PML could be scaled up for larger use cases, the produced final applications would be of substandard quality. From the experiences of using a PML described application in the itACiH project and from the oral feedback provided by the medical professionals using said application, it would appear that such is not the case either. At worst, the application used by the nurses has been described as ”*colorless*”; as can be seen in the screen shots, the application is quite grey. Others have called it ”*bulky*”; the feature navigation tabs allocate a large part of the screen real estate. However, these are superficial issues that could easily be addressed. More importantly, the application has never been described as anything other than highly functional by its users.

Even though PML already scales well, the evaluation has revealed some language design flaws that have practical implications. These shortcomings will be discussed next.

3 Practical Findings

As part of the evaluation process of PML, practical shortcomings of the language and its use have been collected throughout the evolution period. The findings range from addressable usability hurdles, to new must-have features, to critical scalability bottlenecks. This section will present a selection of such findings, in no particular order. The findings could act as the primary input for a possible redesign process of PML.

3.1 Link Vagueness

The concept of behavior properties was introduced in section 2.2 (II). These properties are used to specify behavior in PML descriptions, and do so by linking one component (source) to another (target) with a given role. The role can be paired with a qualifier value which, depending on the role, can have different effects on the link. As an example, the qualifier for the invoker role specifies the event at the source component that will activate the link. The fact that only one qualifier can be specified per link results in a certain degree of *link vagueness* – the user can not fully specify the effects of a link.

Figure 3.21 illustrates the problem. In the example, a parameter of a PalCom command is specified as the source of the link, and a text box is selected as its target. The role of the link is provider, which means that the text box will be provided with the value of one parameter property when the link activates. The qualifier specifies which property is to be used, in this case *value*. The link in the example is vague since the text box property that will be set cannot be specified; it is subject to the target component's specification. The text box will in this case default to set the *text* property when the link is activated, giving the designer no say in the matter. Furthermore, the event of the parameter that activates the link can not be specified. Parameters trigger every time their value changes (command received), but one could imagine other events for the designer to choose from.

In the itACiH project, the problem of link vagueness was experienced on several occasions. Fortunately, it could be circumvented by accessing the source code of the employed PML description interpreter. However, outside the scope of the project that is not an option, which is why this problem has to be addressed as part of the language.

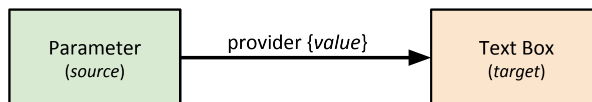


Figure 3.21: Illustration of the *link vagueness* problem in PML

3.2 Unpredictable Needs

One major, reoccurring problem in PML is the unpredictability of the future needs of PML application designers. Currently, PML does not have support for users to add new graphical parts as the need arises. The available classes of parts are static, defined by the language and can not be complemented by the user in any practical way.

In the itACiH project, this meant that every time a new graphical component was needed, the language specification had to be updated, the interpreter modified and recompiled, and the resulting application redistributed to affected devices. This was both impractical and time consuming, but more importantly, not an option for possible future users outside of the PalCom/itACiH project. Such external users would not have access to alter the source code of PML and its interpreters. Even if they were granted such privileges, they are unlikely to have the right level of competence or the resources needed to add new parts to both the language and all affected interpreters. The same type of users would also probably not have time to wait for an update with requested features, even if a system to request features were in place.

As an analogy, only some special case services are built into PalCom, e.g. the manager services of TheThing. Since all thinkable services could not possibly be conceived and included at compile-time, a system to dynamically load services was introduced. The same principle should apply to PML; a handful of common, highly usable parts should be included at compile-time as part of the language. For more specialized purposes, support for dynamically loaded parts should be added.

3.3 Text Formatting

Since the main paradigm of PML is to dislodge the business logic of an application from the graphical part of the application, the possibilities to process the parameter data of incoming commands are nearly non-existant. An area where this has proven especially apparent is in that of text formatting. Something as trivial as displaying a predefined, static message concatenated with a text received from the parameter data of a command requires the use of multiple graphical components.

As an example, contemplate that the current temperature is to be shown in an application, e.g. formatted as ”Temperature: 24 C”. The service providing the temperature outputs 24 as parameter value in a command sent as response to a temperature-request command. Creating an application that displays the temperature text as described here is definitely doable using PML, but at the moment

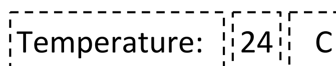


Figure 3.22: ”Text formatting” in PML, using three text labels

it is needlessly complicated. It would typically be done by aligning three labels as illustrated in fig. 3.22: a first one displaying the static text ”Temperature:␣”, a second one displaying the parameter value of incoming temperature commands from the temperature service, and a third one displaying the static text ”␣C”.

For more complex formatting, similar workarounds might quickly become unmanageable; the need for a unified text formatting mechanism in PML is apparent.

3.4 Command Filtering

Due to the asynchronous nature of PalCom (and distributed computing in general), it is not easy to provide hard guarantees regarding message delivery/reception order. Even in situations where such guarantees can be made, variations in message delivery and process times make it difficult for the designer to reason about message ordering. In safety critical applications, this could be potentially hazardous.

To illustrate the problem, consider an application that is to display medical data critical to a patient's health, e.g. what kind of medication he/she is supposed the take. As depicted in fig. 3.23, a user of such an application could request this information first for patient *A*, and rapidly thereafter for patient *B*. If ordering is not guaranteed, or if the designer made a mistake when reasoning about ordering, *info(B)* could arrive before *info(A)*. When *info(A)* eventually arrives, it would be displayed without question as the medical information of patient *B*.

In PML, business logic should be contained in PalCom services. While this is overall a sound principle, there is clearly a need for some sort of high level construct to prevent situations similar to that of the one described here.

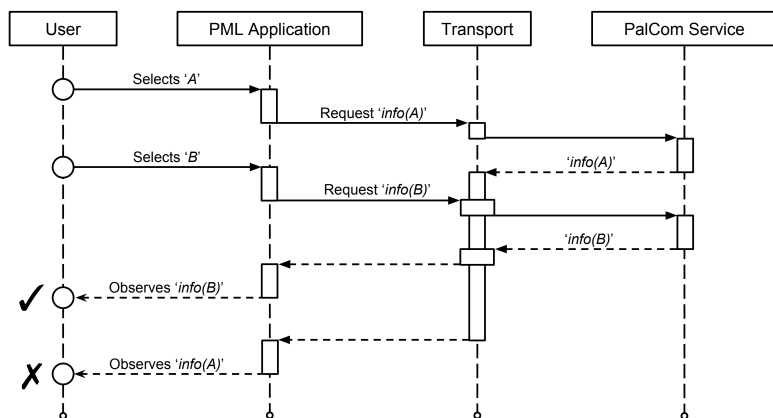


Figure 3.23: User observing wrong patient data due to varying delivery times of patient information

3.5 Dynamic Content

The script-like approach taken in PML to generating applications dynamically from description files provides flexible, adjustable applications. Last-minute changes or additions to description files are done with ease, without the need for heavy development environments and without having to recompile the application. This stands in contrast to the traditional approach of pre-compiling applications, with the primary benefit being the ability to alter the application without the need of an IDE and/or a specific compiler.

While the PML described applications are dynamic, the content of those applications is mostly static. There are some dynamic elements to the language that support changes to the application during run-time. At the most basic level, a text label displaying the content of a parameter from an incoming PalCom command is to some degree contributing to a dynamically changing application. More notably, changing a graphical parts' visibility, or enabling/disable it, can change the flow of an application quite drastically. However, this dynamicity manifests itself on a sub-part level (dynamicity as part of a PML part) rather than on a supra-part level (a PML part as part of dynamicity): a label can dynamically display incoming parameter values, but a dynamic number of labels can currently not be created from the same.

The need to display dynamic content has become apparent during the itACiH project. A general construct is needed in PML to dynamically spawn parts of any class from a non-static source, e.g. parameter values.

Summary

From the results presented in section 1 it appears that for applications on the lower side of the complexity scale (e.g. application prototypes), PML is highly efficient. For the reported example scenario, the development time when using PML was a factor of 10 lower than for a solution that used neither PML nor PalCom. Even considering possible beneficial factors, this notable difference in development time must be considered at least suggestive of the efficiency of PML for low-complexity solutions.

The PML description file that describes the application used by the nurses in the itACiH project, as presented and evaluated in section 2, was at the time of writing this made up of 8405 lines of code. Considering only the findings in section 1, it is hard to draw any real conclusions on whether an ad hoc solution would have been more efficient in the context of the project. However, the PML described application definitely shows the scalability potential of the language. It appears that PML is not only usable for low-complexity applications, but also for solutions that aspire to do more for the end-user.

Evaluating PML in a real-world project resulted in several real-world practical findings. When addressed, these findings (as presented in section 3) should further increase the scalability potential of the language and its usability in the real world.

Chapter IV

The Finale

This final chapter will conclude the work that has been presented throughout this thesis. Section 1 will present the future work that is planned to be carried out in order to make PML adoptable on a wider front. In section 2, a summary of the work that has been presented in chapters I to IV will be given; section 3 will discuss the conclusions of the same.

1 Future Work

PML as presented in this work is only the first step on the road towards the long-term aspirations for supporting graphical solutions in PalCom. This section will present the future work that is planned in order to realize these aspirations.

1.1 Graphical Editor

The example PML described applications from section 2 (II) and section 1 (III), as well as the PML described application used by the mobile team in the itACiH project, as presented in section 2 (III), were created "by hand" using a common text editor. For the smaller example solutions, this approach works well. However, for the larger scale application developed as part of the itACiH project, the PML description became hard to overview and costly to navigate. It also became increasingly harder to keep track of the internal reference IDs of the PML components. These and similar usability issues can all be traced back to the fact that PML descriptions were not meant to be created manually.

As mentioned in section 1 (II), the long-term aspirations for supporting graphical solutions in PalCom is to introduce a graphical editor for a PalCom specific UIDL. As a first step toward that goal, PML was created in this work. As seen above, the need for a graphical editor has already been practically established. This insight and the long-term aspirations of this work point to an obvious piece of future work: the development of a graphical editor for PML. The details for the graphical PML editor are not yet fully formed, however the concept of code-less functionality specification in PML gives room for informed speculation. To

reiterate, functionality in PML is specified by linking functionality to graphical components, instead of the other way around.

In a traditional graphical editor (section 4.1 [I]) the user would typically be presented with a blank canvas on which to compose a GUI (Graphical User Interface) and a palette of graphical component to choose from. Components would then be added to and positioned in the GUI by dragging them from the palette and dropping them onto the canvas. A list of available properties would typically be displayed to the user when selecting the resulting component. The values of such properties could then be edited from within the editor, allowing for an easy way to customize the component. However, the added component would have to be linked to functionality by writing programming code.

The hypothetical PML editor would tentatively turn this concept $180°$. The user would still be presented with a blank canvas, but instead of a palette of components the user would instead be presented with a list of PalCom devices reachable from the editor. Similar to in the PalcomBrowser, this list could then be expanded selectively to reveal the services of a device, the commands of service, etc. The user would then drag-and-drop available PalCom components of any level onto the canvas to add a graphical component for it, hence realizing the vision of linking functionality to graphical components. In ambiguous situations, e.g. if multiple graphical component are viable, the user could be prompted to resolve the ambiguity by selecting from a list of available options. For added graphical components, properties could be edited in the same manner as in traditional editors.

As a visionary example, consider a user that is to create a PML described application for a service that has one input command which in turn has one (text) parameter. Dragging the parameter from the list in the editor and dropping it onto the canvas would result in a text box, rather than a text label, being added. The disambiguity is derived from the fact that the parameter belongs to an input command and is hence to be set, rather than to be displayed. When the command is dropped onto the canvas, one outcome could be a button that invokes it upon being clicked.

Throughout the work on this thesis, several ways in which PML could be extended in order to include more use cases have been identified. A graphical editor would enable the necessary growth without complicating the experience of the user, as language specific details would be hidden from the user under the visual veil of the editor.

1.2 Language Augmentations

The primary reason that the balance in PML was tilted towards simplicity, rather than expressiveness, was the understanding that initially PML descriptions would have to be created manually; the threshold for getting started using the language had to be kept low. When the planned work on the graphical editor for PML has caught up to the current level of the language, that threshold will no longer matter,

and the expressiveness of the language can be allowed to increase well beyond the current self-imposed restrictions.

The most clear-cut way of augmenting PML is to complement its predefined suite of graphical components. By providing greater component variety through a larger total number of components, the need to create custom components will get less likely. This, in turn, will increase the probability that even technically inexperienced users (level 0, table 4.1) will be able to create the applications they envision more accurately. With the visual aid of the graphical editor, the number of properties per component that can be set by the user can also be increased with minimal detrimental effect on user friendliness.

From the evaluation of PML in the itACiH project came a number of practical findings (section 3 [III]). While the language augmentations mentioned above are easily deduced, the findings of the evaluation are more intricate and could only have come from using the language in a real-world scenario. Examples of such findings include the need to support more specific links, in-language text formatting, command filtering, and dynamic content.

Even though both types of augmentations are needed for the future welfare of PML, it is likely the latter kind that will enhance the language in the most significant manner. From a research point of view, they constitute an important piece of future work.

1.3 Integration with PalCom

As one outcome of the evaluation of PML in the itACiH project, the unpredictability of the future needs of PML application designers is established as a reoccurring problem of the language (section 3.2 [III]). No matter how complete the suite of available PML component is made, there will always be future needs that could not be predicted in advance. It is suggested that similar to how all thinkable PalCom services are not included at compile-time, only a handful of common, highly usable PML components should be included at compile-time as part of the language. To ensure that all future needs are covered, a system to dynamically load user created PML components at run-time must be introduced. Such a system should be reminiscent of similar systems already in place in PalCom, e.g. loading services in TheThing, and will be vital in making PML more widely adoptable.

The above is just one example that motivates the future work that is to be carried out in order to more seamlessly integrate PML into the existing PalCom ecosystem. The details of exactly how this integration is to be structured are not yet fully formed. However, the following points are being considered:

- The functionality currently offered by PML interpreters could be integrated into TheThing. In the tool, PML descriptions could then be added in the same way that services and assemblies are added today. This integration would allow the user to use one and the same tool for the deployment of all

PalCom content, instead of being required to use different tools for different purposes.

- The planned graphical editor for PML, as discussed in section 1.1, could possibly be integrated into the PalcomBrowser. Similar to how new assemblies are added and administered, the tool could also handle new PML application descriptions. Again, this type of integration would provide the user with a one-stop application to be used when creating new (high-level) content.

When integrating PML with the rest of the PalCom ecosystem, the idea is to continue to build upon the concept of PalCom's levels of users. This concept was introduced in section 1 (II). In short, the level indicates the level of technical expertise that can be expected of the user. Level 0 users cannot be expected to write programming code and instead heavily rely on graphical experiences. Users at level 1 must have the technical expertise required to install required software (compilers, libraries, etc.) and to program in a given programming language. Table 4.1 shows what the user roles could look like after integrating PML, given that the work presented in this section is pursued as specified.

With the advent of the graphical PML editor, the creator of PML descriptions (designer) no longer needs to write PML code manually. All work is done graphically from within the editor, putting the role of ”designer” on par with that of ”assembler” at level 0. Note that even though the standard library of PML components will be limited, the ambition is for the library to be complete enough for most user needs. This means that most user will not have to create custom components which makes the level of 0 reasonable for designers. This is comparable to how most assemblers should not have to create their own services, but rather reuse existing ones in new constellations. The creation of custom PML components, similar to PalCom services, requires more technical expertise from the side of the user. The task is hence assigned to developers at user level 1.

The planned future work of integrating PML into the PalCom ecosystem is expected to make for a more streamlined user experience, allowing for more users to create great content.

<table>
<tr><th>Level</th><th>Role</th><th>Creates</th><th>Uses</th></tr>
<tr><td rowspan="2">1</td><td rowspan="2">Developer</td><td>Service</td><td rowspan="2">3rd-party IDE</td></tr>
<tr><td>PML Component</td></tr>
<tr><td rowspan="2">0</td><td>Assembler</td><td>Assembly</td><td rowspan="2">PalcomBrowser</td></tr>
<tr><td>Designer</td><td>PML Description</td></tr>
</table>

Table 4.1: User levels in PalCom, post PML integration

1.4 User Case Study

The comparative case study that was reported on in section 1 (III) only involved a single developer. Because of this, one could argue about the value of the study. However, the one-man case study was never meant as a fully fleshed study that should be interpreted as absolute proof, but was rather to function as a first indication of the efficiency of PML. As such, the study was successful, indicating towards roughly 10 times shorter development times when using PML, as opposed to creating an custom graphical solution from scratch.

The main reason for the small study was the immaturity of the developed technology, PML. The large (relative to e.g. creating assemblies) threshold to getting started using PML, combined with the lack of an up-to-date language documentation amounted to a situation that was not optimal for a large scale user case study. Such a study has therefore been delayed until the technology is matured, and is part of the future work to be done. Preliminarily, when the graphical PML editor gets in place, members of this work's target audience of technically inexperienced users can be selected for the case study. Although details are not finalized, the study is likely to resemble the one reported on previously. There will be one or a few given scenarios for which to create a graphical solution. Depending on the size of the pool of selected test subjects, they could be asked to create only a PML solution, only a custom solution, a PML solution followed by an equivalent custom solution, or a custom solution followed by an equivalent PML solution. Development time and/or the lines of codes of each solution will be recorded and possibly followed up by a semi-structured interview with the test subject. To evaluate the intuitiveness of the technology, some test subject may be given instructions on how to use the editor beforehand, while others would create their solutions without such instructions.

The planned large scale user case study is expected to showcase the efficiency of PML when used with a proper editor, and hopefully provide valuable insight into how the technology can be further improved.

2 Summary

A general end-user node in a distributed system can simplistically be divided into three parts: the *Graphical User Interface*, the *Business Logic*, and the *Network Communication*. When considering the division of development resources between these parts, it is not unusual to find that the business logic part of the node is relatively small. The resource intense parts are instead to present the functionality provided by the node to the end-user, and to connect the node with the rest of the system. The solution to the latter is a well-covered area of research, and the problem could be alleviated by building the system on-top a middleware of choice. Remaining was the issue of presenting the node to the end-user in a cost efficient manner. Assuming distributed systems built upon the PalCom middleware, this

work has explored the possibilities of streamlining the process of creating graphical solutions for the end-user nodes of such systems.

The itACiH project has served as the primary motivator for this work. The project is a collaboration between academia, healthcare, and industry companies with funding coming from the swedish research funding agency VINNOVA. It seeks to support care of terminally ill cancer patients in the final stages of live by allowing them to be treated at home, rather than at a cancer ward. To allow for this, a distributed system based on PalCom has been created that includes system nodes for:

- the patients' homes, which connects medical equipment and enables video communication
- the portable devices that allow the nurses of the mobile team to enter and read patient data on the move
- the monitoring station at the ward, which displays patient data, controls equipment in patients' homes and handles video communication

These are just a few examples. The many end-user nodes in the system manifested, from a practical perspective, a general need for a solution to the problem of creating graphical solutions: most of the nodes required a graphical solution of some sort and creating a new, ad hoc solution for every node was not a sustainable approach. The identified need, coupled with the lack of support for such solutions in PalCom motivated what would become the primary contribution of this work: a uniform technology for creating graphical solutions, specifically for PalCom systems.

Researching previous work in the form of technologies for creating graphical solutions, namely *Graphical Editors* and *User Interface Description Languages* (UIDL), provided inspiration for what would ultimately become the solution to the stated problem. As they were, however, the technologies did not meet the solution criteria. These criteria were formulated as a long-term aspiration of introducing a graphical editor for a PalCom specific UIDL, targeted at level 0 PalCom users, i.e. technically inexperienced users that heavily rely on graphical experiences. Building on PalCom, which runs on multiple platforms, any graphical solutions would have to do so as well, hence the need for a platform independent UIDL as a basis. The introduction of level 0 users as the main target audience for the solution radically affected the direction of both this and future work, in that no programming can be required of the user when creating graphical solutions. To allow for this, the technology was to support a novel approach of specifying functionality by turning the conventional approach 180°, linking functionality to graphical components instead of the other way around.

The work presented in this thesis constitutes a first step towards the long-term goal, taken in the form of the PalCom specific UIDL entitled PML. As a first part to the grander aspiration, this technology still requires the user to write code when creating PML described applications. However, the language contains the

necessary constructs to realize the ambition of inverted functionality specification. When the planned future work of creating a graphical editor for PML has been carried out, the users will be able to specify an entire application, including its functionality, all without having to write any programming code.

For now though, the user has to write PML code when creating applications. Because of this, the balance between language expressiveness and simplicity was tilted towards simplicity, keeping the number of available components and pertaining properties in the language's suite low. From working under such restrictions, it can be concluded that it is no easy task to provide the user with a technology for creating graphical solutions that is: generic enough to be applicable on multiple platforms; simple enough for technically inexperienced user to understand; expressive enough for advanced user to create the applications they desire. However, a suite of basic componentes for standard user, complemented by future work on allowing advanced users to specify their own components should allow for both advanced content creation and ease of use.

As concrete, practically usable results of this work, two applications were developed for devices powered by the Android OS:

1. AndroidPUIDI – a PML interpreter used to generate applications for end-users from PML descriptions

2. TheAndroidThing – a mobile adaption of the multi-functional PalCom tool TheThing

Both these tools were essential to the introduction of a PML described application for the nurses of the itACiH project's mobile team, which carry Android tablets when visiting the patients in their homes. The development of the mobile application provided a scenario with real-world requirements in which to evaluate the results of this work on a large scale. It is hard to draw any real conclusions regarding the efficiency of PML from the developed application alone, as no non-PML alternative was developed in parallell. However, the mobile application for the nurses definitely shows the scalability potential of PML. From this part of the evaluation, it can be concluded that the language is not only suitable for low-complexity applications like e.g. early prototypes, but is in fact also applicable for solutions of considerable size and complexity. As a beneficial side-effect, evaluating PML in a real-world project resulted in several real-world practical findings. By addressing these findings in future work, the scalability potential of the language and its usability in the real world is only expected to increase even further.

The other part of the evaluation of PML sought to evaluate the efficiency of a solution created using the language, relative to non-PML solutions. It took on the form of a single test subject user case study. Three solutions to the same small scale problem where developed: one using PML and PalCom, a second using a custom graphical solution and PalCom, and a third using a custom graphical

solution and an ad hoc network protocol. Values for development time and lines of code where recorded and compared for the three solutions. The results for the two metrics were in alignment, although more pronounced for development time, and indicated that PML was highly efficient compared to the two non-PML-based approaches. For development time, the results showed that the PML solution was a factor of 10 faster to develop, which is a notable speed up even considering possible beneficial factors. The conclusion of the results is that PML is an efficient alternative when developing graphical solutions for PalCom-based systems, at the very least for low-complexity solutions. In this context it should also be noted that the PML solution was developed without the planned graphical editor, which can only be expected to further decrease development time. Once this editor is in place, a follow-up user case study of greater magnitud is planned to confirm this statement.

The two-part evaluation shows that this work has been successful in enabling end-user interaction with PalCom-based systems. The primary contributions can be summarized as:

1. PML – a PalCom specific, platform independent language for describing graphical solutions
2. The implementation of framework and tool support for PML, e.g. AndroidPUIDI and TheAndroidThing
3. A large scale, real-world use case for PML, i.e. supporting the mobile team of nurses in the itACiH project

As shown here and throughout this thesis, PML has proved itself capable. It has been shown to be efficient, scalable and useable in real-world scenarios, and has great growing potential in the form of identified future work.

3 Conclusions

Creating graphical solutions for end-user nodes in a distributed system can prove problematic and resource heavy. By considering systems that connect using the PalCom middleware, this work has provided a solution to the problem in the form of a PalCom specific UIDL – PML.

From the evaluations performed, it is concluded that PML is efficient at enabling the user to create graphical solutions for PalCom-based systems; the results showed roughly 10 times shorter development times for small scale applications. Considering this substantial reduction in development time, combined with the self-imposed graphical limitations of PML, one could presume that the produced final application would be of substandard quality. However, the PML described application used by the medical professionals in the itACiH project has always

been described as highly functional by the staff. From this, it is concluded that PML can indeed produce applications with a professional level of quality.

From a language perspective, two aspirational requirements were stated for what would ultimately become PML: *platform independence* and *codeless functionality specification*. It must be concluded that PML satisfies the former of these requirements completely, and the latter conceptually. Two PML description interpreter applications were developed as part of this work: one for Android devices, and another based on Swing for desktop computers. This exemplifies the platform independence of PML. The absence of a graphical PML editor forces users to write descriptions manually, which violates the second requirement. However, all necessary constructs needed to provide codeless functionality specification is present in the language. Hence, the requirement is conceptually satisfied.

As a final reflection, it is from the promising evaluation results of PML and the identified future work concluded that even though the created technology is already efficient, it can only grow more competent as it matures.

Summary

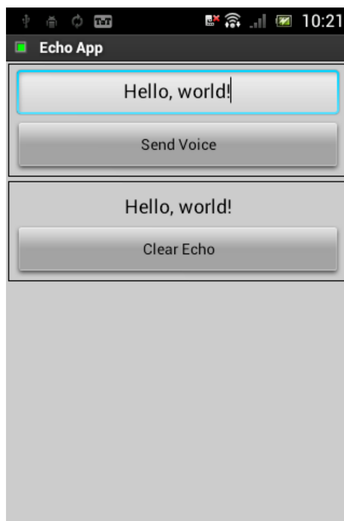
In this chapter, the future needs of PML have been established in the form of a graphical editor, language augmentations, deeper integration with PalCom, and a large scale user case study. In a final contemplation, the chapter was closed by summarizing and concluding the work that has been presented throughout this thesis.

Appendix A

The Listings

In this appendix, the full code for the two PML described applications that have been presented in the thesis are listed. In appendix 1, the code for the application that was used as a continuous example during the introduction of PML (section 2.4 [II]) is listed. The code for the application that was used as part of the efficiency comparison evaluation of PML (section 1 [III]) is listed in appendix 2.

1 Echo Application

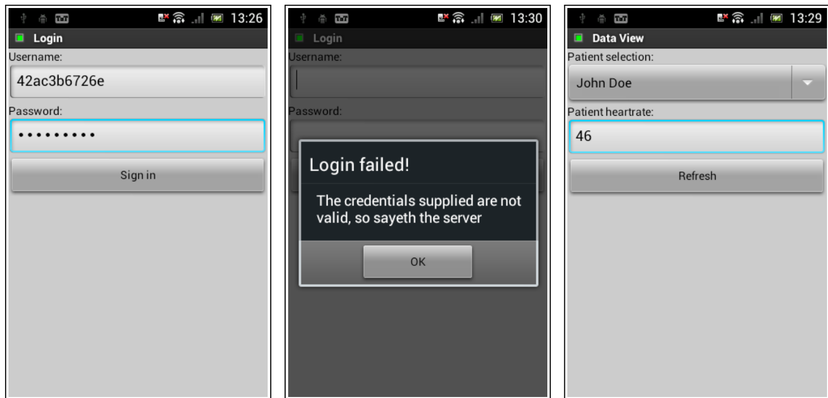



```
1 <?xml version="1.0" encoding="UTF-8"?>
2
3 <puiml>
4   <universe>
5     <unit id="devHost" class="P:Device">
6       <unit id="svcEcho" class="P:Service">
```

```
7         <unit id="cmdVoice" class="P:Command">
8           <unit id="prmVoiceMsg" class="P:Param"/>
9         </unit>
10        <unit id="cmdEcho" class="P:Command">
11          <unit id="prmEchoMsg" class="P:Param"/>
12        </unit>
13      </unit>
14    </unit>
15  </universe>
16
17  <discovery>
18    <property unit-name="devHost" name="p:id">C:30e8f8a2-2ae5-455b-abe3-287
        d3a2015cd</property>
19    <property unit-name="svcEcho" name="p:required">true</property>
20    <property unit-name="svcEcho" name="p:instance">1</property>
21    <property unit-name="svcEcho" name="p:cdid">X:SimpleEchoService</
        property>
22    <property unit-name="svcEcho" name="p:cn">BAJ1</property>
23    <property unit-name="svcEcho" name="p:udid">X:SimpleEchoService</
        property>
24    <property unit-name="svcEcho" name="p:un">BAJ1</property>
25    <property unit-name="cmdVoice" name="p:id">Voice</property>
26    <property unit-name="cmdVoice" name="p:direction">in</property>
27    <property unit-name="prmVoiceMsg" name="p:id">message</property>
28    <property unit-name="cmdEcho" name="p:id">Echo</property>
29    <property unit-name="cmdEcho" name="p:direction">out</property>
30    <property unit-name="prmEchoMsg" name="p:id">message</property>
31  </discovery>
32
33  <structure>
34    <part id="appDemo" class="G:Application">
35      <part id="winMain" class="G:Window">
36        <part id="panTop" class="G:Area">
37          <part id="txtVoice" class="G:TextField"/>
38          <part id="btnSend" class="G:Button"/>
39        </part>
40        <part id="panBottom" class="G:Area">
41          <part id="lblEcho"class="G:Label"/>
42          <part id="btnClear" class="G:Button"/>
43        </part>
44      </part>
45    </part>
46  </structure>
47
48  <style>
49    <property part-name="winMain" name="g:title">Echo App</property>
50    <property part-name="winMain" name="g:layout">linear</property>
51    <property part-name="winMain" name="g:layout-orientation">vertical</
        property>
52    <property part-name="panTop" name="g:border">line</property>
53    <property part-name="panTop" name="g:layout-orientation">vertical</
        property>
54    <property part-name="txtVoice" name="g:align-h">center</property>
55    <property part-name="btnSend" name="g:text">Send Voice</property>
56    <property part-name="panBottom" name="g:border">line</property>
57    <property part-name="panBottom" name="g:layout-orientation">vertical</
        property>
58    <property part-name="lblEcho" name="g:align-h">center</property>
59    <property part-name="lblEcho" name="g:align-v">center</property>
60    <property part-name="lblEcho" name="g:font-size">18</property>
61    <property part-name="lblEcho" name="g:size">-1,75</property>
62    <property part-name="btnClear" name="g:text">Clear Echo</property>
63  </style>
64
```

```
65   <logic>
66     <fact id="cstClear" class="G:Constant"/>
67   </logic>
68
69   <behavior>
70     <property part-name="appDemo" name="p:invoker" event="loaded">winMain</
         property>
71     <property part-name="txtVoice" name="p:provider" get="text">prmVoiceMsg<
         /property>
72     <property part-name="btnSend" name="p:invoker" event="clicked">cmdVoice<
         /property>
73     <property part-name="lblEcho" name="p:viewer" set="text" order="0">
         prmEchoMsg</property>
74     <property part-name="lblEcho" name="p:viewer" set="text" order="1">
         cstClear</property>
75     <property part-name="btnClear" name="p:invoker" event="clicked">cstClear
         </property>
76     <property fact-name="cstClear" name="p:value"></property>
77   </behavior>
78 </puiml>
```

2 Heart Rate Application




```
1  <?xml version="1.0" encoding="UTF-8"?>
2
3  <puiml>
4    <universe>
5      <unit id="1" class="P:Device">
6        <unit id="1.1" class="P:Service">
7          <unit id="1.1.1" class="P:Command">
8            <unit id="1.1.1.1" class="P:Param"/>
9            <unit id="1.1.1.2" class="P:Param"/>
10         </unit>
11         <unit id="1.1.2" class="P:Command"/>
12         <unit id="1.1.3" class="P:Command"/>
13         <unit id="1.1.4" class="P:Command"/>
14         <unit id="1.1.5" class="P:Command">
```

```
15          <unit id="1.1.5.1" class="P:Param"/>
16        </unit>
17        <unit id="1.1.6" class="P:Command">
18          <unit id="1.1.6.1" class="P:Param"/>
19        </unit>
20        <unit id="1.1.7" class="P:Command">
21          <unit id="1.1.7.1" class="P:Param"/>
22        </unit>
23      </unit>
24    </unit>
25  </universe>
26
27  <discovery>
28    <property unit-name="1" name="p:id">C:30e8f8a2-2ae5-455b-abe3-287
          d3a2015cd</property>
29    <property unit-name="1.1" name="p:required">true</property>
30    <property unit-name="1.1" name="p:instance">1</property>
31    <property unit-name="1.1" name="p:cdid">X:PatientDataService</property>
32    <property unit-name="1.1" name="p:cn">BAJ1</property>
33    <property unit-name="1.1" name="p:udid">X:PatientDataService</property>
34    <property unit-name="1.1" name="p:un">BAJ1</property>
35    <property unit-name="1.1.1" name="p:id">Login</property>
36    <property unit-name="1.1.1" name="p:direction">in</property>
37    <property unit-name="1.1.1.1" name="p:id">username</property>
38    <property unit-name="1.1.1.2" name="p:id">password</property>
39    <property unit-name="1.1.2" name="p:id">LoginSucceeded</property>
40    <property unit-name="1.1.2" name="p:direction">out</property>
41    <property unit-name="1.1.3" name="p:id">LoginFailed</property>
42    <property unit-name="1.1.3" name="p:direction">out</property>
43    <property unit-name="1.1.4" name="p:id">GetPatientList</property>
44    <property unit-name="1.1.4" name="p:direction">in</property>
45    <property unit-name="1.1.5" name="p:id">PatientList</property>
46    <property unit-name="1.1.5" name="p:direction">out</property>
47    <property unit-name="1.1.5.1" name="p:id">content</property>
48    <property unit-name="1.1.6" name="p:id">GetData</property>
49    <property unit-name="1.1.6" name="p:direction">in</property>
50    <property unit-name="1.1.6.1" name="p:id">patientId</property>
51    <property unit-name="1.1.7" name="p:id">Data</property>
52    <property unit-name="1.1.7" name="p:direction">out</property>
53    <property unit-name="1.1.7.1" name="p:id">content</property>
54  </discovery>
55
56  <structure>
57    <part class="G:Application">
58      <behavior>
59        <property name="p:invoker">wLogin</property>
60      </behavior>
61      <part class="G:Window" id="wLogin">
62        <style>
63          <property name="g:title">Login</property>
64          <property name="g:layout">linear</property>
65          <property name="g:layout-orientation">vertical</property>
66        </style>
67        <part class="G:Label">
68          <style>
69            <property name="g:text">Username:</property>
70          </style>
71        </part>
72        <part class="G:TextField">
73          <behavior>
74            <property name="p:provider">1.1.1.1</property>
75          </behavior>
76        </part>
77        <part class="G:Label">
```

```
78            <style>
79              <property name="g:text">Password:</property>
80            </style>
81          </part>
82          <part class="G:TextField">
83            <style>
84              <property name="g:sensitive">true</property>
85            </style>
86            <behavior>
87              <property name="p:provider">1.1.1.2</property>
88            </behavior>
89          </part>
90          <part class="G:Button">
91            <style>
92              <property name="g:text">Sign in</property>
93            </style>
94            <behavior>
95              <property name="p:invoker">1.1.1</property>
96            </behavior>
97          </part>
98        </part>
99        <part class="G:Window" id="wData">
100         <style>
101           <property name="g:title">Data View</property>
102           <property name="g:layout">linear</property>
103           <property name="g:layout-orientation">vertical</property>
104         </style>
105         <part class="G:Label">
106           <style>
107             <property name="g:text">Patient selection:</property>
108           </style>
109         </part>
110         <part class="G:DropDownList">
111           <style>
112             <property name="g:content">;&lt;No selection&gt;;</property>
113           </style>
114           <behavior>
115             <property name="p:viewer">1.1.5.1</property>
116             <property name="p:invoker">1.1.6</property>
117             <property name="p:provider">1.1.6.1</property>
118             <property name="p:delimiter">;</property>
119             <property name="p:row-length">2</property>
120             <property name="p:value-index">1</property>
121             <property name="p:option-index">2</property>
122           </behavior>
123         </part>
124         <part class="G:Label">
125           <style>
126             <property name="g:text">Patient heartrate:</property>
127           </style>
128         </part>
129         <part class="G:TextField">
130           <style/>
131           <behavior>
132             <property name="p:editable">false</property>
133             <property name="p:viewer">1.1.7.1</property>
134           </behavior>
135         </part>
136         <part class="G:Button">
137           <style>
138             <property name="g:text">Refresh</property>
139           </style>
140           <behavior>
141             <property name="p:invoker">1.1.6</property>
```

```
142             </behavior>
143           </part>
144         </part>
145         <part class="G:YesNoDialog" id="dLoginFailed">
146           <style>
147             <property name="g:title">Login failed!</property>
148             <property name="g:text">The credentials supplied are not valid, so
                     sayeth the server</property>
149             <property name="g:text-positive">OK</property>
150           </style>
151         </part>
152       </part>
153     </structure>
154 
155     <style/>
156 
157     <behavior>
158       <property unit-name="1.1.2" name="p:invoker" event="received" order="0">
              wData</property>
159       <property unit-name="1.1.2" name="p:invoker" event="received" order="1">
              1.1.4</property>
160       <property unit-name="1.1.3" name="p:invoker" event="received">
              dLoginFailed</property>
161     </behavior>
162 
163     <logic/>
164   </puiml>
```

Bibliography

[1] Android Developer Tools: Graphical Layout Editor. http://developer.android.com/tools/help/adt.html#graphical-editor.

[2] CLOC: Count Lines of Code. http://cloc.sourceforge.net.

[3] Eclipse WindowBuilder: Project proposal. http://www.eclipse.org/proposals/tools.windowbuilder.

[4] ENGROSS: ENabling GROwing Software Systems. Swedish Foundation for Strategic Research (SSF) project RIT08-0075.

[5] EU-FP6 Information and Society Technology. http://cordis.europa.eu/fp6/ist.htm.

[6] IT-stöd för Avancerad Cancervård i Hemmet. VINNOVA 2011-02796. http://itacih.cs.lth.se/itACiH/itACiH.html.

[7] NetBeans IDE features: Swing GUI builder. https://netbeans.org/features/java/swing.html.

[8] Palcom: Palpable computing. IST 002057. http://www.ist-palcom.org.

[9] Xcode Interface Builder: Interface Builder Built-In. https://developer.apple.com/xcode/interface-builder.

[10] M. Abrams and J. Helms. User interface markup language (UIML) specification. 2004.

[11] M. F. Ali and M. Abrams. Simplifying construction of multi-platform user interfaces using UIML. In *UIML Europe 2001 Conference*, 2001.

[12] M. F. Ali, M. A. Perez-Quinones, M. Abrams, and E. Shell. Building multi-platform user interfaces with UIML. *Computer-Aided Design of User Interfaces III*, pages 255–266, 2002.

[13] P. Andersen and S. B. Larsen. PalCom external report 70: Developer's companion. March 2009.

[14] K. Arnold. The Jini architecture: Dynamic services in a flexible network. In *Proceedings of the 36th annual ACM/IEEE design automation conference*, pages 157–162. ACM, 1999.

[15] O. Bergman, L. Jacobson, and M. Jaresand. Cancerfondsrapporten 2013, 2013.

[16] E. Bruera, N. Kuehn, M. J. Miller, P. Selmser, and K. Macmillan. The edmonton symptom assessment system (ESAS): a simple method for the assessment of palliative care patients. *Journal of palliative care*, 1991.

[17] B. Dumas, B. Signer, and D. Lalanne. A graphical UIDL editor for multimodal interaction design based on SMUIML. *Proceedings of the workshop on software support for user interface description language*, 2011.

[18] W. K. Edwards, M. W. Newman, J. Z. Sedivy, T. F. Smith, D. Balfanz, D. K. Smetters, H. Wong, and S. Izadi. Using Speakeasy for ad hoc peer-to-peer collaboration. In *Proceedings of the 2002 ACM conference on Computer supported cooperative work*, pages 256–265. ACM, 2002.

[19] Q. Limbourg, J. Vanderdonckt, B. Michotte, L. Bouillon, and V. López-Jaquero. USIXML: A language supporting multi-path development of user interfaces. *Engineering human computer interaction and interactive systems*, pages 200–220, 2005.

[20] B. Magnusson. Supporting intelligibility in a healthcare scenario using Palcom. In *Workshop on Intelligibility and Control in Pervasive Computing, PERINT2012, at Pervasive*, 2012.

[21] L. Mathiassen. *Computers and Design in context*. MIT press, 1997.

[22] P. L. McEntire, J. G. O'Reilly, and R. Laison. *Distributed Computing: Concepts and Implementations*. IEEE Press, 1984.

[23] G. Mori, F. Paternò, and C. Santoro. Tool support for designing nomadic applications. In *Proceedings of the 8th international conference on Intelligent user interfaces*, pages 141–148. ACM, 2003.

[24] A. Puerta and J. Eisenstein. XIML: A universal language for user interfaces. *White paper*, 2001.

[25] D. Svensson, G. Hedin, and B. Magnusson. Pervasive applications through scripted assemblies of services. In *IEEE International Conference on Pervasive Services*, pages 301–307, 2007.

[26] D. Svensson Fors, B. Magnusson, S. Gestegård Robertz, G. Hedin, and E. Nilsson-Nyman. Ad-hoc composition of pervasive services in the PalCom architecture. In *Proceedings of the 2009 international conference on Pervasive services*, pages 83–92. ACM, 2009.

[27] M. Weiser. The computer for the 21st century. *Scientific american*, 265(3):94–104, 1991.

[28] R. J. Weiss and J. P. Craiger. Ubiquitous computing. *The Industrial-Organizational Psychologist*, 39(4):44–52, 2002.

[29] A. S. Zigmond and R. P. Snaith. The hospital anxiety and depression scale. *Acta psychiatrica scandinavica*, 67(6):361–370, 1983.