Part I

FRAMING THE PROBLEM AND CURRENT STATE-OF-THE-ART

You can put some informational part preamble text here. Illo principalmente su nos. Non message *occidental* anglo-romanic da. Debitas effortio simplificate sia se, auxiliar summarios da que, se avantiate publicationes via. Pan in terra summarios, capital interlingua se que. Al via multo esser specimen, campo responder que da. Le usate medical addresses pro, europa origine sanctificate nos se.

Part II

DESIGN ITERATIONS AND CONSTRUCTING A THEORY

You can put some informational part preamble text here. Illo principalmente su nos. Non message *occidental* anglo-romanic da. Debitas effortio simplificate sia se, auxiliar summarios da que, se avantiate publicationes via. Pan in terra summarios, capital interlingua se que. Al via multo esser specimen, campo responder que da. Le usate medical addresses pro, europa origine sanctificate nos se.

3

DESIGN ITERATION I

I think the only way forward is going from applying algorithms to individual transactions, to first placing information in context — pixels to pictures — and only applying algorithms after one sees how the transaction relates to the other data. It's the only way that I can see that it's going to close this sense-making gap.

— Jeff Jonas [120]

Parts of this chapter appear in [73] and [109].

An iterative development process was followed for the work described in this thesis. In the following chapters three iterations, each consisting of a requirements and planning phase, analysis and design phase, implementation phase and evaluation phase, is described in more detail. Iterative processes are essential to modern-day software and hardware development methodologies, exemplified by the various agile development frameworks [61].

3.1 REQUIREMENTS

Scenarios are commonly used in software engineering and interaction design to help discover and analyse requirements. The following scenario was presented at the start of the project to guide the design process:

Mark is a 12-year-old boy and he is at home receiving his friend Dries from school. Dries arrives with a portable music player loaded with his favourite songs. He wants to play some recent collections for Mark. Mark's home is equipped with a sophisticated surround sound system, and they have recently installed an ambient lighting system that is connected to the sound system and renders the mood of the music by dynamic colour lighting in the room. They decide to use both to enjoy the music. Dries starts streaming his music to the environment.

An object (or several objects) shows possible input and output ports for streaming music in the environment. By interaction with the object/objects, Mark connects the output from Dries' music stream to the input of the sound system. Now the room is full with Dries' music and the colourful lighting effects. Mark's mom, Sofia, now comes back from work. She starts preparing dinner for the family. Mark and Dries don't want to bother her with their loud music. They again use the object(s) to re-arrange the music stream. Now the music is streamed to Mark's portable music player while playing back at Dries'. It is also connected to the ambient lighting system directly, bypassing the sound system. They both are enjoying the same music using

their own favourite earphones, and the colourful lighting effects, but without loud music in the environment.

The object(s) shows the connection possibilities with a high level of semantic abstraction, hiding the complexity of wired or wireless networks. By interacting with the object(s), semantic connections can be built, redirected, cut or bypassed.

The first takeaway from this scenario is that the focus is on the connections between the devices, instead of on the devices themselves. This brings us to the first design decision: *Semantic connections* are introduced as a means for users to indicate their intentions concerning the information exchange between smart objects in a smart environment.

SEMANTIC CONNECTION A semantic connection is a relationship between two entities in a smart environment for which we focus on the semantics—or meaning—of the connections between these entities.

The term semantic connections is used to refer to meaningful connections and relationships between entities in a smart environment. These connections are both real "physical" connections (e.g. wired or wireless connections that exist in the real world) and "mental" conceptual connections that seem to be there from the user's perspective. The context of the connections, for example the objects that they connect, provide meaning to the connections. The term "semantics" refers to the meaningfulness of the connections. The type of connection, which often has the emphasis now (e.g. WiFi, Bluetooth or USB) is not considered to be the most relevant, but what the connection can do for someone — its functionality — even more.

The following requirements were defined during this phase:

- Semantic connections exist in both the physical and the digital world. We need ways to visualise these invisible connections and to control them.
- Devices need to be able to share their capabilities and content with the other devices in their environment.

A number of different approaches to visualising and controlling semantic connections were explored in the first iteration, and these are described in Section 3.3. We also need a way to model the devices, their capabilities and the connections themselves. This is the subject of the next section.

3.2 ONTOLOGY DESIGN

Web Ontology Language (OWL) 2, the ontology language used to build ontologies for the Semantic Web, was used to create the ontologies in this thesis. OWL 2 has been a W3C Recommendation since

October 2009, and adds new capabilities like property chains to the original OWL standard.

Ontologies and ontology engineering are described in more detail in Section 9.

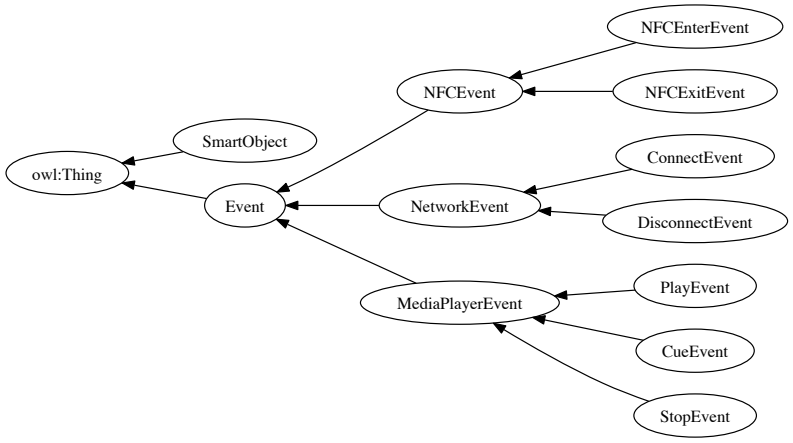

Figure 3: Ontology indicating subclass relationships

A first attempt at modelling the various entities in an ontology is shown in Figure 3. A bottoms-up approach to modelling was used, where we attempted to model each entity using the least number of statements. These entities were later aligned with foundational ontologies – the approach that was followed is discussed further in Chapter 9. Each entity is modelled as an `owl:Class`, where all classes are subclassed from the root class, `owl:Thing`. Each edge in the graph above is a `rdf:type` relationship, and the direction of the arrow indicates the direction of the subclass relationship.

The `owl:` prefix is used to denote the OWL 2 Namespace Document located at `http://www.w3.org/2002/07/owl`.

During the initial development stages, we realised that the most promising way of describing low-level interactions seemed to be to describe them in terms of *interaction events*, that are defined as follows:

Interaction events are discussed in more detail in Chapter 8.

INTERACTION EVENT An interaction event is defined as an event that occurs at a certain time instant and was generated by a specific smart object. It reports either the intent of a user's action directly, or a perceivable change in a smart object's state.

An interaction event in the smart space consists of an event ID, timestamp and other related information, like the smart object that generated that event. For the scenario, three types of interaction events were defined:

- Network events: A `ConnectEvent` indicates that a device is entering the smart space, while a `DisconnectEvent` means that the device is exiting the smart space.

- Near Field Communication (NFC) events: An `NFCEnterEvent` signifies that an NFC tag has entered the `RFID` field, and a `NFCExitEvent` is generated when it leaves the field.
- Media player events: When the user presses the Play button on the media player, a `PlayEvent` is generated. When the music is stopped, or at the end of the song, a `StopEvent` is generated. Pressing the Forward button forwards the song by 5 seconds. This time period is attached to a `CueEvent` using an `atTime` relationship.

All OWL code listings in this thesis are written using Turtle[1] syntax. Turtle is a human-friendly alternative to Extensible Markup Language (XML) based syntaxes.

The following properties were defined:

```
:connectedTo
    a owl:ObjectProperty;
    a owl:IrreflexiveProperty;
    a owl:SymmetricProperty ;
    rdfs:domain :SmartObject ;
    rdfs:range :SmartObject .

:atTime
    a owl:DatatypeProperty ;
    rdfs:comment "At a specific time (in milliseconds)" ;
    rdfs:range xsd:integer .

:generatedBy
    a owl:ObjectProperty ;
    rdfs:domain :Event ;
    rdfs:range :SmartObject .

:hasPosition
    a owl:DatatypeProperty ;
    rdfs:range xsd:integer .

:hasRFIDTag
    a owl:DatatypeProperty ;
    rdfs:range xsd:string .

:inXSDDateTime
    a owl:DatatypeProperty ;
    rdfs:range xsd:dateTime .
```

The `connectedTo` object property is both *symmetric* and *irreflexive*. Irreflexive properties are a new feature in OWL 2. A symmetric property is its own inverse, which means that if we indicate a `connectedTo` relationship from device A to device B, device B will also have a `connectedTo` relationship to device A. Another way to think of symmetric properties is that they are bidirectional relationships.

An irreflexive property is a property that never relates an individual to itself [49]. This allows us to restrict our model by not allowing a `connectedTo` relationship from a device to itself.

An example with individuals, also called instances, that make use of the ontology is shown in Figure 4. In the figure, classes are denoted with ellipses, individuals with boxes and datatypes as plain text. Class membership is denoted with dotted lines and relationships are denoted with solid lines. It shows a Nokia N900 and N95 smartphone instantiated as `SmartObjects` with their associated Radio Frequency Identification (RFID) tags.

Why an RFID tag? in Section 6.2.1 we argue that that each smart object must be uniquely identifiable in the physical world by digital devices.

An instantiated `NFCExitEvent`, called `event-1cecdba5`, is also shown. When an event is generated a Universally unique identifier (UUID) is assigned to it, to enable the event to be uniquely identified in the smart space. It is also associated with a smart object using the `generatedBy` property. The `hasPosition` relationship provides additional metadata required by the interaction tile, which is described in the next section.

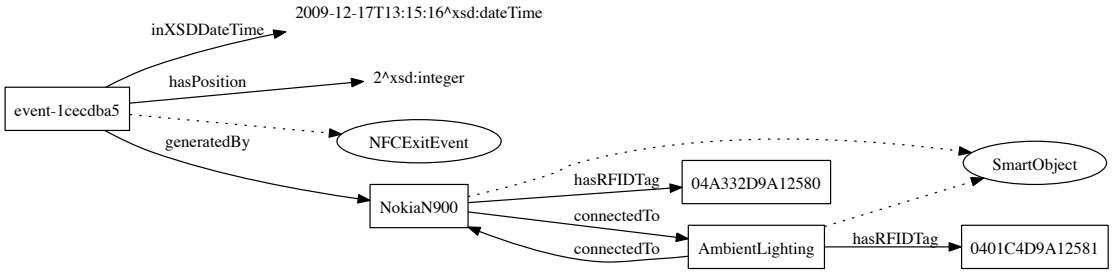

Figure 4: Individuals that were instantiated based on the ontology

SPARQL Protocol and RDF Query Language (SPARQL)[3] form the query language for the Semantic Web. Along with OWL, it is one of the core technologies of the Semantic Web, having been a W3C Recommendation since January 2008. SPARQL queries are based on the idea of graph pattern matching [95], where data that is returned from the query is set to match the pattern.

We also make use of SPARQL to define rules, which is described in Section 9.2.

To determine which other smart objects a specific device, for example a mobile phone, is connected to, a simple SPARQL query suffices:

```
SELECT DISTINCT ?object WHERE{
:phone1 :connectedTo ?object .
}
```

A *triple store* is used to store both the instances and the ontology. A triple store is a purpose-built database for storing and retrieving

3 http://www.w3.org/TR/rdf-sparql-query/

triples, in the format subject-predicate-object. In the above example *phone1* would be the subject, *connectedTo* the predicate and *?object* the object. There are a number of commercial and open-source triple store implementations. The Jena[4] framework is a Java Application Programming Interface (API) that enables access to many triple store implementations, supports SPARQL and also has its own persistent triple store. It was used in this first implementation and was also later adopted by the Smart Objects For Intelligent Applications (SOFIA) project.

An advantage of using SPARQL and a triple store is that it is easy to add additional constraints and/or specifics to the query, compared to a traditional Structured Query Language (SQL) database, where unions between columns and tables can get quite complicated very quickly.

To get the last event that was generated by a specific device, the SPARQL query is a little bit more complex, but still surprisingly manageable:

```
SELECT ?event ?eventType WHERE{
:deviceID :hasRFIDTag ?tag .
?event :hasRFIDTag ?tag .
?event a ?eventType .
?event :inXSDDateTime ?time .
FILTER (?eventType = :NFCEnterEvent  || ?eventType = :NFCExitEvent)
}
ORDER BY DESC(?time)
```

How do we model the semantic connections between devices? Since semantic modelling is property-oriented instead of object-oriented, we started by focusing on the possible predicates that can be used to describe connections. We need a way to model whether a connection is possible — this can be done with a `canConnectTo` property. We also need to know if a device is currently connected to another device — `connectedTo`. Then we need a way to model the capabilities that each device provides. In this first iteration, we defined two properties called `consumes` and `provides`. They are used as follows:

```
NokiaN900 provides AudioCapability .
NokiaN95 consumes AudioCapability .
```

During the later design iterations we decided to model capabilities as functionalities of a device instead, and make the name of the property clearer to indicate whether it is a functionality of a source or a sink. The property `provides` was changed to `functionalitySource`, and `consumes` was changed to `functionalitySink`. These early properties are mentioned here for the sake of completeness, and to show how aspects of the ontology have changed between iterations.

4 http://jena.apache.org/

3.3 DEVICE DESIGN

Based on the scenario, a number of smart objects had to be constructed or repurposed, and the necessary software had to be developed.

To explore the different possibilities of visualising and manipulating connections between devices, a number of different prototypes were constructed. The first of these is called the *interaction tile*.

3.3.1 *Interaction Tile*

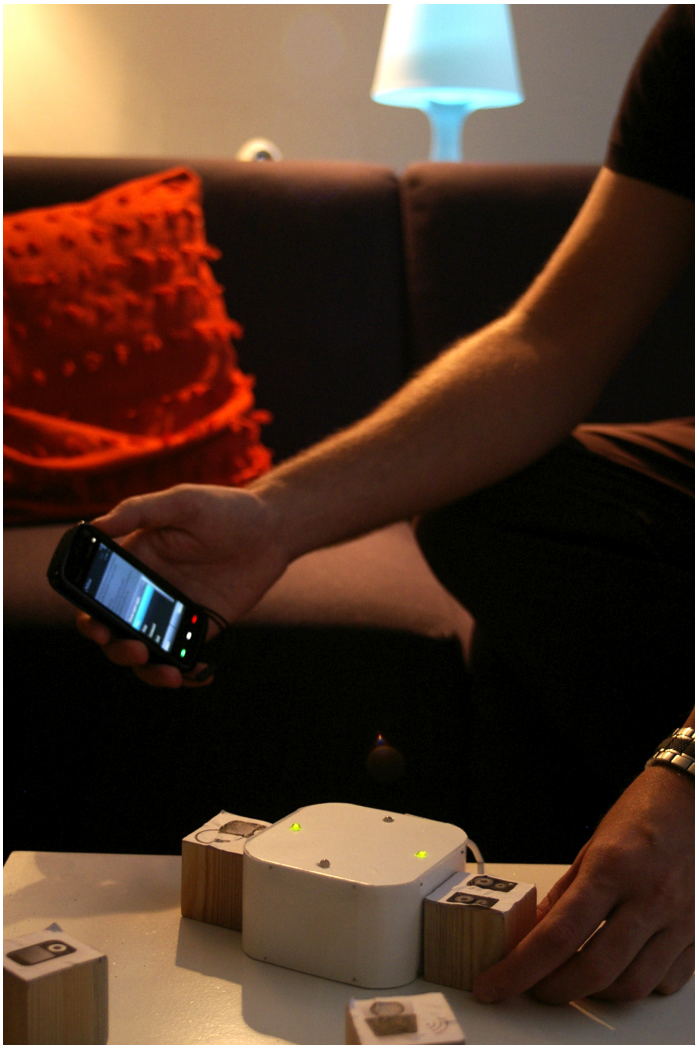

Figure 5: The interaction tile and mobile phone

The interaction tile, shown in Figure 5, was inspired by Kalanithi and Merrill's "Siftables" cubes [67]. It was designed to explore and manipulate connections through direct manipulation – by making simple spatial arrangements. Each device in the smart environment is represented by a cube containing an RFID tag and a small magnet,

with an icon on the top of the cube to signify the device being represented. When a cube is placed next to one of the four sides of the tile, an LED on the tile lights up to indicate that it has been recognised. When a second cube is placed next to the tile, the following LED visualisations are used:

- Pulsating green light - a connection is possible
- Constant green light - a connection exists
- Red light - no connection is possible

The interaction tile visualises the various connections by allowing a user to explore which objects are currently connected, and what connections are possible. By means of putting a cube representing a device close to one of the four sides of the tile, a user can check if there is a connection, and if not, whether a connection is possible. By shaking the tile it is possible to create a connection between two devices, or where there is an existing connection, to break the connection. The interaction tile consists of the following components:

- Arduino Duemilanove with Atmel ATMega328 microcontroller
- ACR122/Touchatag 13.56MHz RFID reader
- RF Solutions ANT-1356M 13.56MHz RFID Antenna Coil
- Multi-colour LEDs
- Accelerometer
- Vibration motor
- Piezoelectric speaker
- Magnetic switches

The RFID reader component has been tested under Windows, Linux and Mac OS X.

The Arduino communicates with a PC via a serial interface over USB, while the RFID reader uses Personal Computer/Smart Card (PC/SC) drivers over USB. The accelerometer is used to measure when the user is shaking the tile, while the vibration motor and speaker provide haptic and auditory feedback. The magnetic switches are used to determine which side of the tile a cube has been placed. The final laser-cut version of the interaction tile prototype is shown in Figure 6.

Two alternative designs are presented in Van der Vlist's thesis [108]. A more detailed discussion of the interaction tile and how its design is informed by product semantics is available in [109].

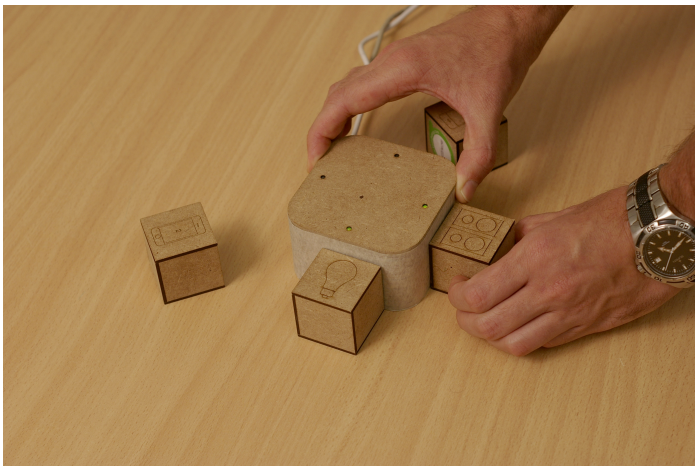

Figure 6: A laser-cut version of the interaction tile prototype

3.3.2 *Lamp*

To create the ambient lighting system, we replaced the internals of a table lamp with an RGB LED array and an Arduino[5]. A Bluetooth module was connected to the Arduino to facilitate communication with a computer, the final result of which can be seen in Figure 7.

The Ikea Lampan lamp that was used for the prototype currently retails for around €3.

The coloured lighting can be changed by sending three RGB values (in the range 0-255) to the lamp via the serial-over-Bluetooth interface.

3.3.3 *Mobile phones*

For the first iteration, a Nokia N95 and Nokia 5800 XpressMusic phone (shown in Figure 8) were used. The two phones use the Symbian S60 operating system, and Python for S60 was used to write software for the mobile phones.

Python for S60 is Nokia's port of the Python programming language for Symbian devices.

3.3.4 *RFID reader used in interaction tile*

Most of the RFID readers and tags targeted at the amateur and hobbyist markets, like the PhidgetRFID and Innovations ID-12 modules, operate in the 125KHz range. While they are relatively cheap and readily available, the 125KHz readers cannot read multiple tags within range of the reader at the same time. For this a 13.56MHz reader is required. The most widely used RFID tags at the moment, the MiFare range owned by NXP, operate at 13.56MHz. These tags are used in most public transport payment systems, including the London Oyster Card and the Dutch OV-Chipkaart system.

A relatively cheap 13.56MHz RFID reader system, the ACR122, is developed by Hong Kong-based Advanced Card Systems (ACS). It

5 http://www.arkadian.eu/pages/219/arduino-controlled-ikea-lamp

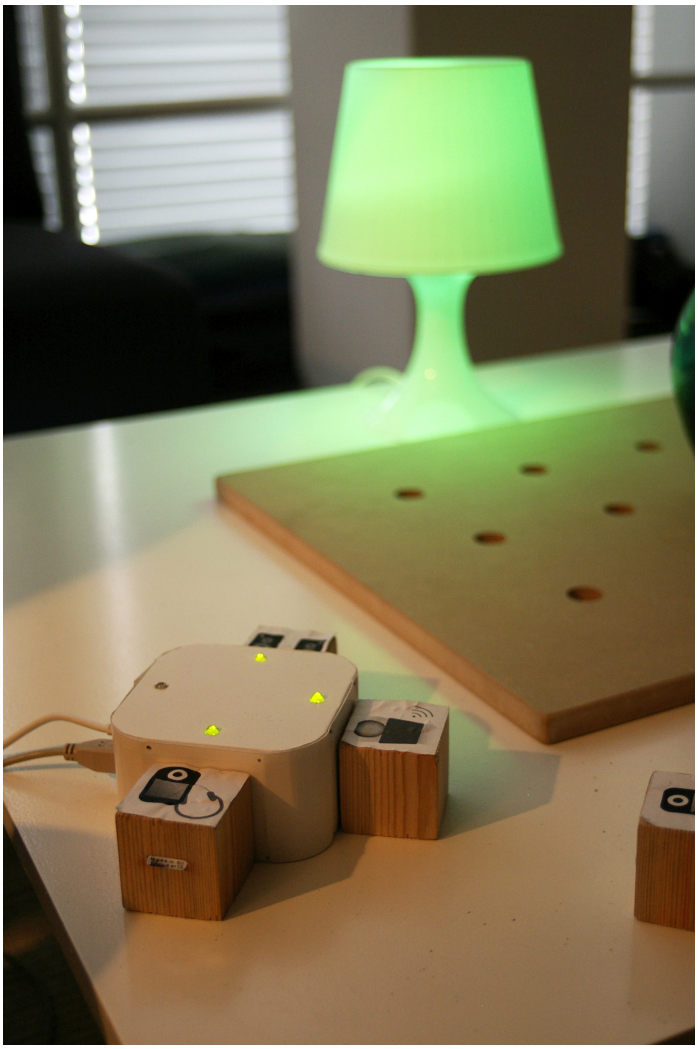

Figure 7: The interaction and cubes, with the lamp in the background

A rebranded version of the ACR122, called the Touchatag[6], is currently sold with 10 tags for around €30.

uses the NXP PN532 chip to read RFID tags. The reader has an on-board PCB antenna – to extend the range of the unit we removed two capacitors on the PCB and soldered in an external ANT-1356M coil antenna from RF Solutions.

3.4 IMPLEMENTATION

Following the design and development of the ontology and required devices, a demonstrator that implements the the scenario was created. A visual overview of the demonstrator can be seen in Figure 9. A video of the scenario is available[8].

Each device in the demonstrator is represented by a Knowledge Processor (KP) software module. KPs communicate via the Semantic Information Broker (SIB), as shown in Figure 10. As discussed in Sec-

8 https://vimeo.com/15594590

Figure 8: The Nokia 5800 XpressMusic mobile phone with the lamp and some cubes

tion 1.1.1, the SIB acts as an information broker, distributing messages between devices. This was an early design decision to reduce coupling, by minimising direct communication between devices, with all messages relayed via the SIB. This philosophy of having a blackboard architectural model, where devices can write to and read from, was followed through all subsequent design iterations. The technical implementation of the various KPs are described in the following subsections.

The system architecture model is described in more detail in Chapter 10.

3.4.1 *Interaction Tile KP*

The interaction tile KP was written in Python and tested on Ubuntu Linux 10.04. On startup, the KP connects to the Arduino inside the interaction tile via the serial-over-USB interface. It establishes a connection with the SIB, after which it connects to the RFID reader inside the tile.

The open-source rfidiot.org library was used to communicate with the RFID reader.

The KP then enters an event loop, waiting until a cube is placed next to the tile. When this happens, the Arduino sends the position of the cube next to the tile to the KP via the serial interface. The RFID tag is read, and a `NFCEnterEvent` is generated. After the RFID tag is read it is temporarily disabled, to ensure that the tag will only be read again after being removed from the field and coming into range again.

If the tile is shaken and a connection is possible, the KP updates the SIB by inserting `connectedTo` relationships between the devices, represented by the cubes next to the tile. If there are existing connections, the `connectedTo` relationships are removed instead. When a cube is removed from the tile, the Arduino again sends the position of the cube via the serial interface to notify the KP. The `python-pyscard`,

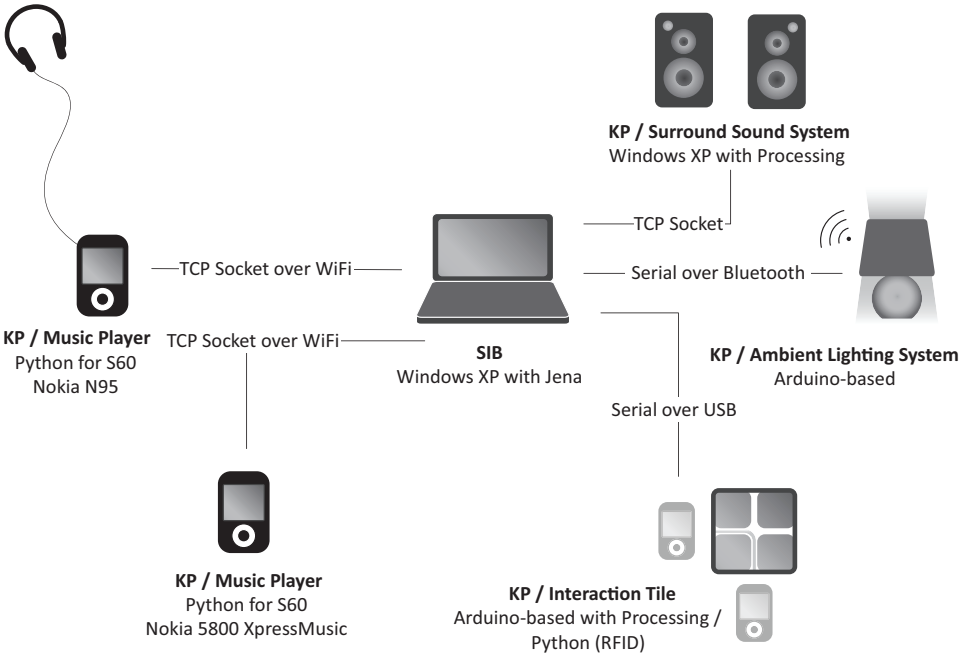

Figure 9: An overview of the demonstrator

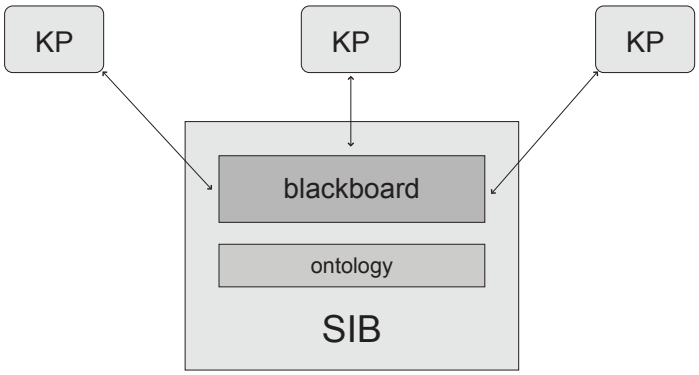

Figure 10: System architecture of demonstrator

`pcsc-tools` and `pcscd` PC/SC libraries are required on Ubuntu Linux to communicate with the RFID reader.

3.4.2 *Music Player KP*

This Python-based KP runs on Symbian S60. It has been tested on a Nokia N95 and Nokia 5800 XpressMusic phone. When the KP starts up, it connects to the SIB, generates a `ConnectEvent` and subscribes to new `PlayEvents`, `StopEvents` and `CueEvents`. It then enters an event loop. Pressing the play/stop/forward buttons on the phone's Graphical User Interface (GUI) will generate the corresponding event, and the KP will also respond to events generated by other devices that it is connected to via the `connectedTo` relationship.

Another version of the music player KP was developed for a Nokia N900 smartphone that runs on Maemo 5 Linux. This KP was also written in Python and makes use of the PyQt4 library. This KP is functionally equivalent to the Symbian S60 version, apart from running on the Maemo platform and using the Qt4 Phonon framework to provide music play/stop/forward capabilities.

3.4.3 *Light KP*

This KP was written in Java and makes use of the Minim audio library[9] for beat detection, in order to generate meaningful lighting patterns that can be sent to the table lamp.

The Minim audio library is part of the Processing software development environment, used for interaction design prototyping.

The KP listens for media player events from connected devices, and generates RGB values based on the rhythm of the music. These RGB values are then sent to the Arduino in the table lamp via the serial-over-Bluetooth interface. On Ubuntu Linux the `librxtx-java` package is required for serial communication when using Java.

Part of the event handler that handles subscriptions from the SIB is shown in the following code fragment:

```
@Override
public void kpic_SIBEventHandler(String xml) {
String subject = null;
String object = null;
String predicate = null;

println( "Subscription notification!" );
//Get triples that were added or updated in the SIB
Vector<Vector<String>> triples = xmlTools.getNewResultEventTriple(xml);

if(triples!=null){
        for(int i=0; i<triples.size() ; i++ ){
                Vector<String> t=triples.get(i);
                subject=xmlTools.triple_getSubject(t);
                predicate=xmlTools.triple_getPredicate(t);
                object=xmlTools.triple_getObject(t);

                //when we have a new connectedTo relationship to the LightKP
                if(predicate.contains( "connectedTo" ) && object.contains(deviceID)){

                        //subscribe to source events
                        subscribeToSourceEvents(subject);
                }
   ...
```

9 http://code.compartmental.net/tools/minim/

When the Light KP is connected to another device KP using a `connectedTo` relationship, we subscribe to the interaction events generated by that device using the `subscribeToSourceEvents()` function. This code fragment is shown below as an example of how subscriptions are created using the Java KP interface:

```
void subscribeToSourceEvents(String source) {

println( "Subscribing to source events from "  + source);
xml=kp.subscribeRDF( null , sofia +  "launchedBy"  , source, URI);

if(xml==null || xml.length()==0){
print( "Subscription message NOT valid!\n" ); return;}
print( "Subscribe confirmed:" +
(this.xmlTools.isSubscriptionConfirmed(xml)? "YES" : "NO" )+ "\n" );

if(!this.xmlTools.isSubscriptionConfirmed(xml)){return;}
String sub_id = this.xmlTools.getSubscriptionID(xml);
println( "Subscription ID: "  +sub_id);
subscriptions.put(source,sub_id);
}
```

3.4.4 *SIB*

The first SIB implementation used in the SOFIA project is called Smart-M3, developed by Nokia, and an open source implementation is available online[10]. The SIB is written in C and uses Nokia's Piglet triple store as a database backend. It is only available on Linux as it makes of the D-Bus message bus system. Other dependencies include the Avahi service discovery framework and Expat XML parser. The SIB consists of a daemon called `sibd`, which communicates with KPs over TCP/IP using a `sib-tcp` connector module.

3.5 DISCUSSION & CONCLUSION

This first iteration constructed a number of devices that could be reused in future iterations, and explored approaches to creating connections between devices. These approaches were focused at proximal interactions with tangible interfaces instead of the usual GUI-based solutions. Let us look at some issues that were uncovered during the implementation, followed by a conclusion.

This iteration details the first use of Smart-M3, where KPs communicate with a SIB using Smart Space Access Protocol (SSAP)[54]. SSAP consists of a number of operations to insert, update and subscribe to information in the SIB. These operations are encoded using XML. A

10 http://sourceforge.net/projects/smart-m3/

triple-format query from a KP is sent, and the response from the SIB is in triple-format as well.

We attempt to solve the interoperability problem by following a blackboard-based approach. Some of the problems associated with current blackboard-based platforms are scalability and access rights. While the goals of this thesis do not involve solving these problems, they should be considered as possible constraints. In Chapter 11 we will look in more detail at the performance-related issues of the system architecture.

The evaluation of this iteration, where the various alternative tangible approaches are compared in a usability study, is discussed in more detail in [60]. This study was performed in the Context Lab at the Eindhoven University of Technology, and made use of the Teach-Back protocol [107] and Norman's Action Cycle Diagram [76].

From this first iteration we learned that using interaction events to model device and user interaction works well. Yet the different types of interaction events need to be generalised, so that they can be reused in other scenarios and environments. One difficulty we encountered was how to model the capabilities of devices in more detail so that they can be shared with other devices. In the next chapter we extend the scenario to include devices from our various partners in the SOFIA project, and model the media capabilities of devices in order to perform semantic matching of different media types.

Part III

CONTRIBUTIONS AND EVALUATION

In this part of the thesis, the more general concepts and techniques that can be applied to ubiquitous computing are described. These concepts and techniques were extracted from work done during the three design iterations.

Part IV

APPENDIX