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* angloromanic 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* angloromanic 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 III

# GENERALISED MODELS, SOFTWARE ARCHITECTURE 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.

# 11 EVALUATION

*Statistics are a little like anarchists:*
*if you force them to stay in line, you're begging for trouble.*

— Sarah Slobin, Graphics Editor, The Wall Street Journal

*Parts of this chapter have previously appeared in [73].*

In this chapter, two evaluations are described. First we will look at an evaluation of the system performance of the software architecture. This evaluation was performed during the smart home pilot of the second design iteration described in Chapter 4. Secondly, we will look at a method the author developed to evaluate ontologies based on the Cognitive Dimensions (CD) framework, as well as an evaluation of the ontology described in this thesis using the method.

*The CD framework was first mentioned in Section 2.1.4.*

## 11.1 EVALUATING THE SYSTEM PERFORMANCE

### 11.1.1 *Introduction*

To evaluate the software architecture described in Chapter 10, we compared it against a previous evaluation of the two M3-based smart space implementations, Smart-M3 and RDF Information Base System (RIBS), described in Section 10.4. These implementations were evaluated by Eteläperä et al [34]. They performed both a qualitative evaluation and quantitative measurements. The performance measurements were made on a Intel Atom 1.6GHz laptop connected via a 100Mbps Ethernet router to a Intel Pentium M 1.7GHz laptop. The qualitative evaluation focused on documentation, installation process and portability as well as run-time usability. According to [34] RIBS is up to 237 times faster than Smart-M3 in certain instances, but it is reported that its memory model limits the number of use cases it can be applied to. RIBS uses static memory allocation with no disk storage and a bitcube triple store, which means that the maximum number of triples has to be known a priori.

Query time measurements for Smart-M3 indicated a query time of 4.4ms for one triple and 8.6ms for 10 triples. For RIBS a query time of 0.65ms was measured for one triple. RIBS did not support querying 10 triples at the time the evaluation was performed. Subscription time measurements indicated a subscription indication time of 140ms for Smart-M3, while RIBS measured 0.75ms.

Bhardwaj et al. [13] compared Smart-M3 against their Open Source Architecture for Sensors (OSAS) framework. They did a performance analysis based on end-to-end delay measurements between the smart objects in smart spaces. The analysis shows that the end-to-end delays are mostly dominated by KP-to-SIB updates, rather than the processing delays on Knowledge Processors (KPs) or on the Semantic Information Broker (SIB).

Luukkala et al. [62] used Smart-M3 with Answer Set Programming (ASP) techniques to handle resource allocation and conflict resolution. They used the Service Platform for Innovative Communication Environment (SPICE) Mobile Ontology[1] to describe device capabilities and ASP as a rule-based approach to reasoning. The SPICE ontology allows for the definition of device capabilities in a sub-ontology called Distributed Communication Sphere (DCS) [108].

*The SPICE DCS ontology was first mentioned in Section 7.2.3.*

*The smart home pilot scenario was first described in Section 4.1.*

### 11.1.2 *Experimental setup*

In the smart home pilot, media content is shared among several devices in a smart home setting. Music is shared between a mobile device, a stereo speaker set and a lighting device that renders the mood of the music with coloured lighting. The music experience, consisting of both light and music information, is also shared remotely between friends living in separate homes through the lighting device. Other lighting sources, like the smart functional lighting and the smart wall wash lights are sensitive to user presence and the use of other lighting sources in the environment.

The performance measurements were made in an environment that approximates a real-world home environment for these kinds of devices. Two wireless routers were placed in two different locations, bridged with an ethernet network cable. One router was configured to act as a DHCP server, while the other acted as a network bridge. The Connector KP, Music Player KP and SIB were connected to the router in location A, while the Sound/Light Transformer (SLT) KP was connected to the router in location B. All components were connected to the network via the 802.11g wireless protocol. The system specifi-

1 http://ontology.ist-spice.org/

| COMPONENT | CPU | OS | MEMORY | LANGUAGE |
|---|---|---|---|---|
| SIB | Core 2 Duo 2.8GHz | Ubuntu 10.04 | 4GB | Java |
| SLT KP | Core 2 Duo 2.2GHz | Ubuntu 11.04 | 2GB | Java |
| Connector KP | Core 2 Duo 2.6GHz | OS X 10.6.8 | 4GB | Python |
| Music Player KP | ARM Cortex-A8 | Maemo 5 | 256MB | Python |
| Presence KP | Pentium M | Ubuntu 10.04 | 512MB | Python |
| Lamp KP | Pentium M | Ubuntu 10.04 | 512MB | Python |

Table 7: System specifications of components used in evaluation

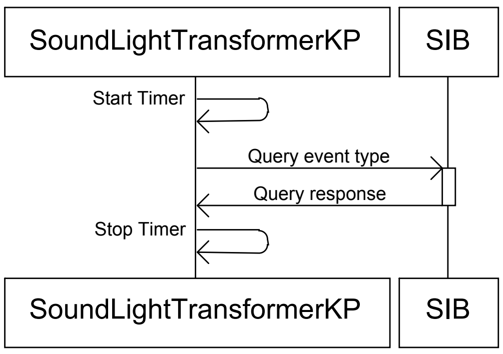


Figure 54: Sequence diagram of Sound/Light Transformer KP query measurement

cations of each component used in the performance evaluation are shown in Table 7.

*An overview of the smart home pilot is shown in Figure 19 on page 65, while a diagram showing the technical details is shown in Figure 20 on page 67.*

Figure 54 and Figure 55 show the sequence diagrams of the measurements made for the SLT KP and the Connector KP respectively. During the pilot, 86 measurements were made by the SLT KP – each time an event was received. 961 measurements were made by the Connector KP – each time a user scans a tag. Note that the query name in 55 could differ between subsequent queries, as the user could be scanning a different tag every tames.

For the music player KP, we measured the time between inserting a new event, and receiving an update from the SIB indicating that the specific event had occurred. First a subscription is made to the `PlayEvent` type, as shown in Figure 56. A new `PlayEvent` is generated by the KP, and when the KP is notified of this event by the SIB, the KP queries the SIB to determine if the notification is indeed for the event that it generated itself.

The Lamp-KP was connected to the decorative wall-wash lights (four LED lamps), creating coloured illumination on the

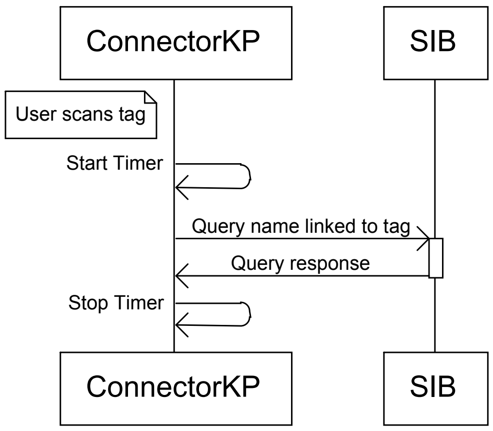


Figure 55: Sequence diagram of Connector KP query measurement

wall of the room. The lamps are shown in Figure 58, including a description of its components. The Presence-KP determines the presence of a user in an activity area of a room and sends the presence information to the SIB. The Lamp-KP is subscribed to this presence information, and gets updated whenever the presence is updated by the Presence-KP to the SIB. There are two states to be updated by the Presence-KP on the SIB: `Away` and `Present`. Based on these states, the Lamp-KP turns the lamps on or off. For example, when the `Present` state is specified by the Presence-KP, the Lamp-KP sends the `ON` command to all lamps, and the `OFF` command when the `Away` state is specified. The Lamp-KP is also subscribed to the states of the SLT KP. The sequence diagram for the Presence-KP, SLT KP, Lamp-KP and SIB is shown in Figure 57.

### *Reasoning setup*

For the pilot, constraint violation checking was disabled, as this introduced quite a large delay ($> 1000\text{ms}$), and was not necessary for the purposes of the pilot. Constraint checking ensures that instances in the triple store meet the constraints attached to classes and properties in an ontology. Constraint violation checks are computationally expensive and cannot be performed for each add, remove and update operation. One possible solution is to perform constraint violation checks at regular intervals and then remove the offending triples.

We made use of OWL 2 RL/RDF Rules in the smart home pilot, which is a *semantic* subset of OWL 2 *Full*. This should not

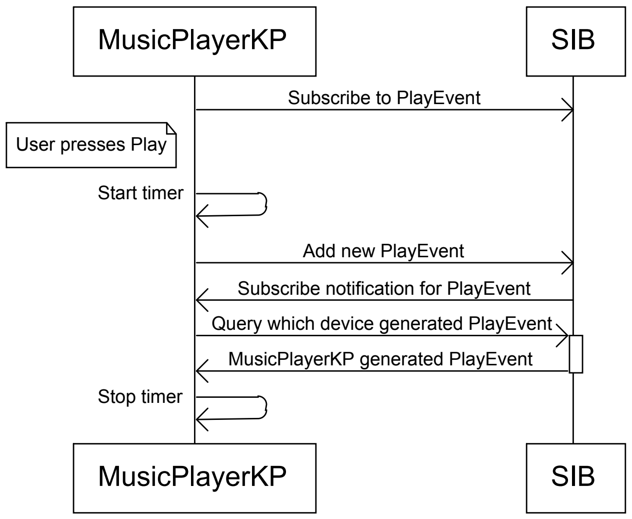


Figure 56: Sequence diagram of Music Player KP subscription measurement

be confused with the first part of the OWL 2 RL Profile[2], which is a *syntactic* subset of OWL 2 *DL*, and restricted in the type of inferences that can be performed. In practice, most OWL 2 reasoners implement OWL 2 RL/RDF Rules (from here on known as OWL 2 RL). OWL 2 RL addresses a significant subset of OWL 2, including property chains and transitive properties. It is fully specified as a set of rules - in our case, as a set of SPARQL Inferencing Notation (SPIN) rules. This means that it is even possible to select only the parts of OWL 2 that are required for a specific ontology, to allow for scalable reasoning.

2 http://www.w3.org/TR/owl-profiles/

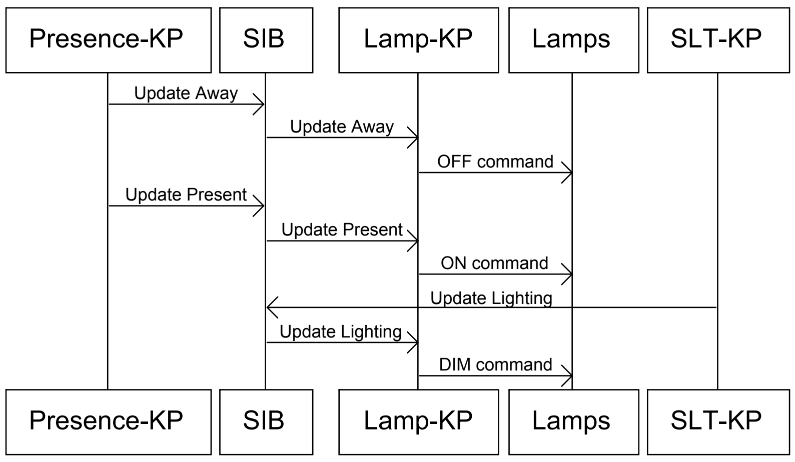


Figure 57: Sequence diagram of Presence-KP and Lamp-KP

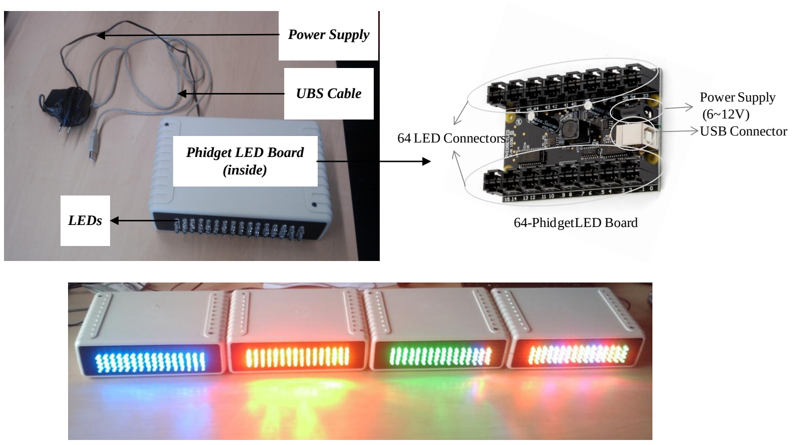


Figure 58: Lamp-KP

During the smart home pilot, all Smart Space Access Protocol (SSAP) messages received by the SIB were logged for further analysis.

*SSAP is described in Section 10.3.*

### 11.1.3 *Experimental Results*

After every reasoning cycle both the asserted and inferred models were written to disk, generating a total of 8306 models during the pilot. Reasoning was performed once, after all ontologies were loaded, and then for every add, remove and update operation. This resulted in a total of 5158 measurements of model size and reasoning time during the pilot. No reasoning was performed during queries.

During the pilot, 70655 total queries were performed by devices connected to the SIB. The time to perform each query was recorded on the SIB, and is shown in Figure 59. The histogram with bin size 25 is plotted on a logarithmic scale. Around 70000 queries take 2ms or less to complete, accounting for more than 99% of the queries. Of all the queries, only 3 queries took 30ms or longer to complete, with all queries completing in less than 60ms. Keep in mind that these measurements were performed on the SIB, hence the measurements do not take network latency into account.

Figure 60 shows the histograms, Gaussian Kernel Density Estimates (KDEs) and Cumulative Distribution Functions (CDFs) of the Connector KP and SLT KP *query time measurements*. A bin size of 20 and a bandwidth of 0.5 was used to plot the figures. It shows that the typical query time for the Connector KP is very short, with a few outliers that took a very long time to com-

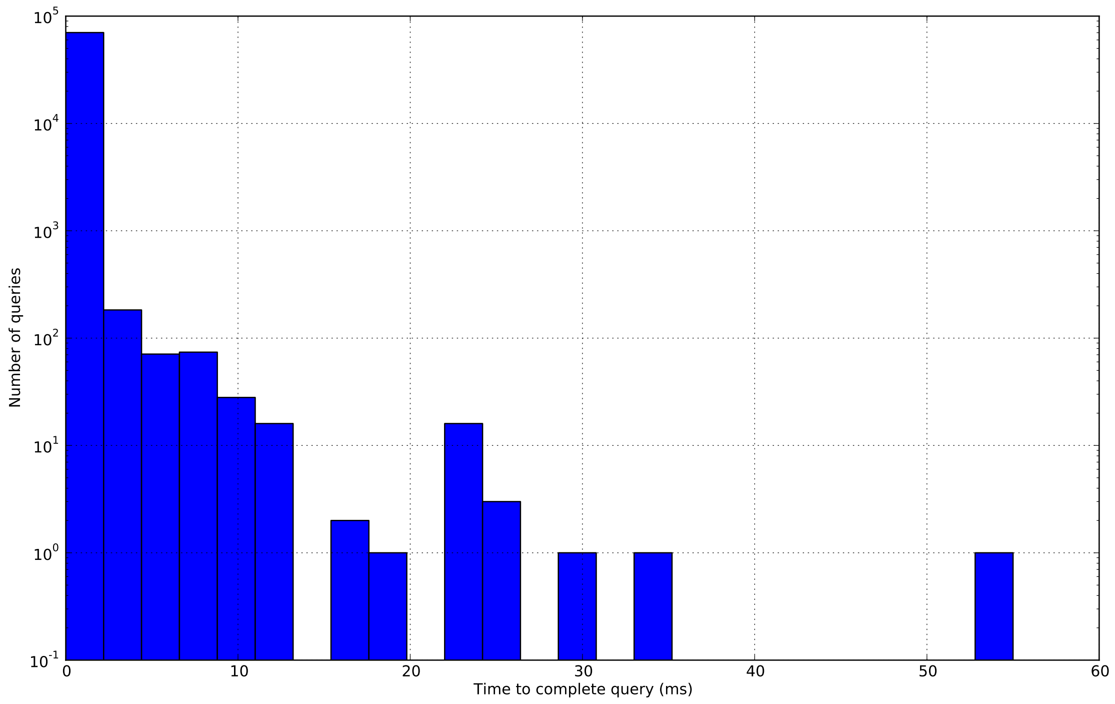


Figure 59: Query time measurements on SIB

plete (35.2s). For the SLT KP, the case is similar, but there are no extreme outliers, with the longest query taking only 587ms to complete. Note that the KDE provides similar information to the histogram, but handles outliers more gracefully by not using binning, and also results in a smoother graph.

The CDF of the Connector KP indicates that queries taking more than two seconds to complete are very rare. The queries that do take longer than two seconds, take an unusually long time to complete. We believe that it could be related to problems in the wireless network, or related to the Python implementation of the Knowledge Processor Interface (KPI), as the problem did not present itself when using other KPI implementations.

For the SLT KP, most queries completed within 100ms, with very few queries taking longer than 500ms to complete.

For the music player KP, most subscription notifications completed in an average of 0.86s, as shown in Figure 61. Keep in mind that after the new PlayEvent is added, inferencing is performed on the triple store before the subscribe notification is generated. Summary statistics of Music Player KP, Connector

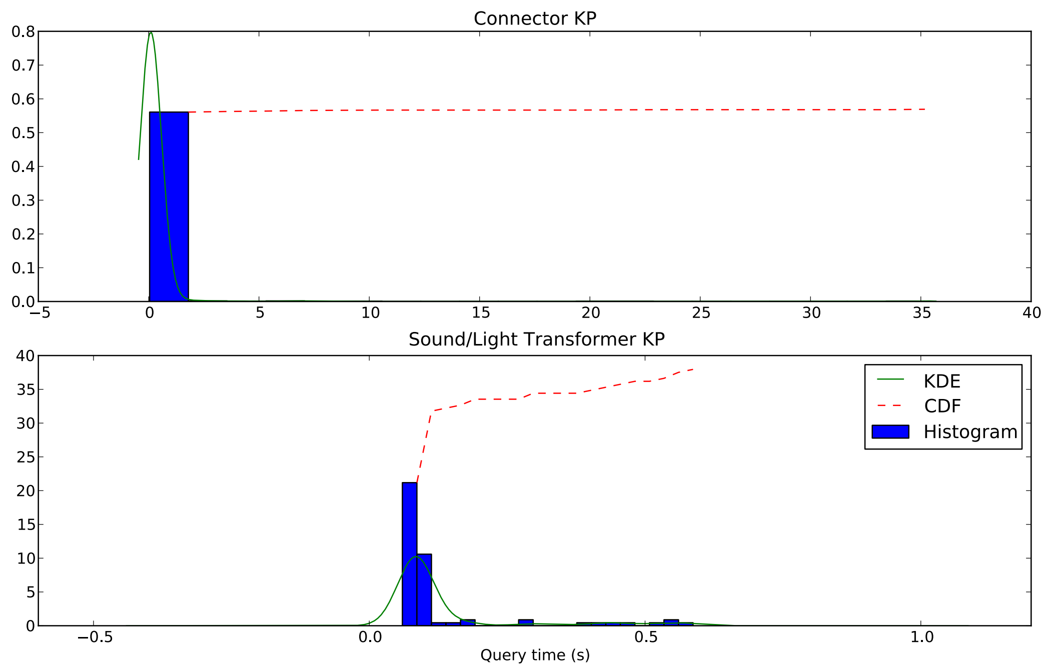


Figure 60: Histograms, kernel density estimates and cumulative distribution functions of Connector KP and Sound/Light Transformer KP measurements

KP and Sound/Light Transformer KP measurements are shown in Table 8.

In Figure 62, the following is shown:

- Model size: Number of triples asserted by ontology or connected KPs
- Inferred model size: Number of triples inferred by reasoning engine
- Inferencing duration: Time (in ms) to complete one reasoning cycle

The sharp peaks indicate the times that the SIB was restarted. The first reasoning cycle after a restart takes about 3 seconds, with subsequent cycles taking on average 275ms (as seen in Table 9).

There is a slow but steady increase in the number of triples as new events get added to the triple store. After each restart these events are cleared and the base assertions loaded from the ontology. These assertions include the OWL 2 RL specification, stored as SPIN rules, which account for the large number of triples.

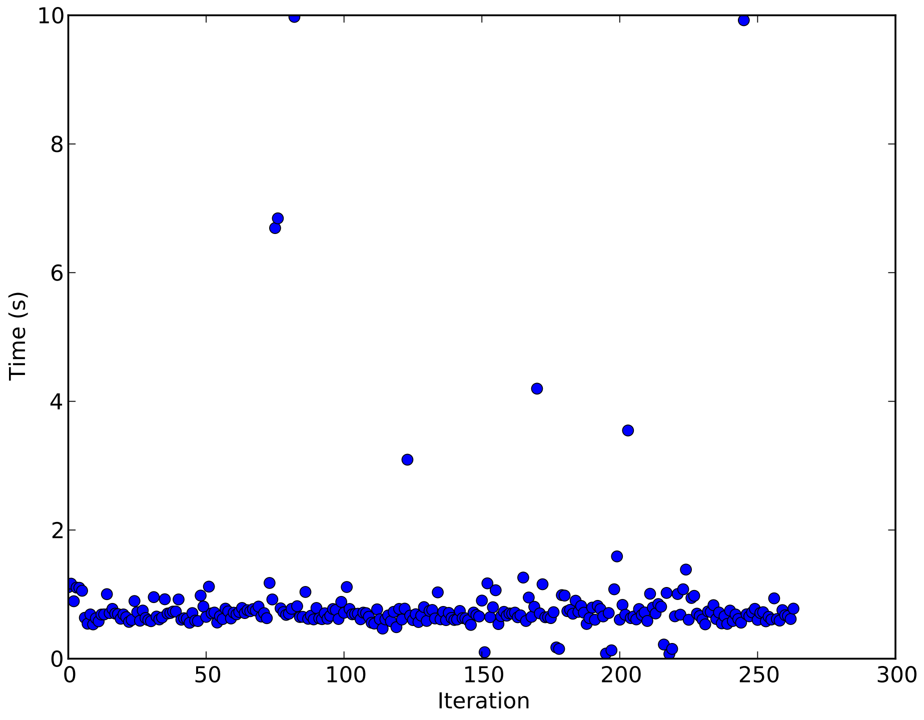


Figure 61: Subscription measurements of Music Player KP

| COMPONENT | NR. OF OBS. | MIN. | MAX. | MEAN | STD. DEV. |
|---|---|---|---|---|---|
| Music Player KP | 264 | 0.074 | 9.975 | 0.861 | 1.017 |
| Connector KP | 961 | 0.044 | 35.184 | 0.275 | 1.942 |
| Sound/Light KP | 86 | 0.06 | 0.587 | 0.131 | 0.122 |
| Lamp-KP | 98 | 0.012 | 0.049 | 0.03 | 0.006 |
| Presence-KP | 172 | 0.145 | 0.244 | 0.176 | 0.018 |

Table 8: Summary statistics of Music Player KP, Connector KP and Sound/Light Transformer KP measurements

| COMPONENT | NR. OF OBS. | MIN. | MAX. | MEAN | STD. DEV. |
|---|---|---|---|---|---|
| Model size | 5158 | 1346 | 3396 | 2916.7 | 201.07 |
| Inferred model size | 5158 | 1369 | 1819 | 1501.8 | 107.6 |
| Reasoning time | 5158 | 181 | 2912 | 274.99 | 152.96 |

Table 9: Summary statistics for asserted and inferred model sizes and reasoning time

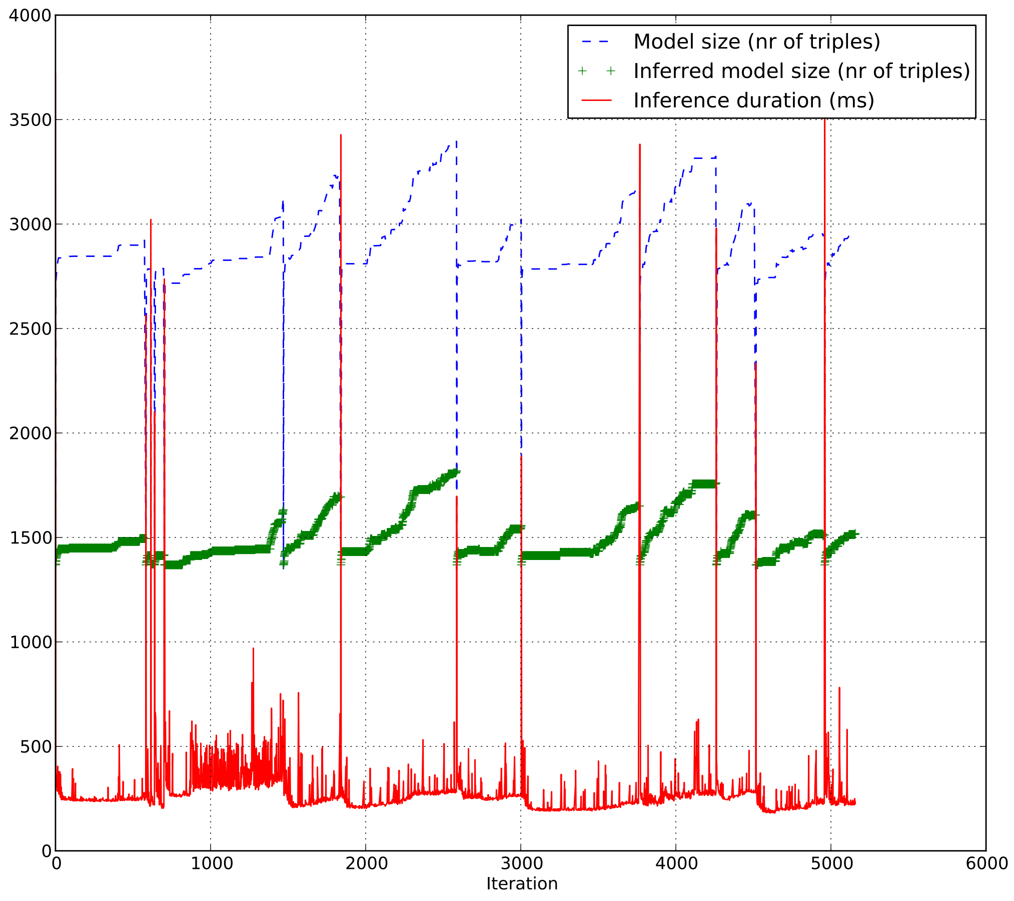


Figure 62: Size of asserted and inferred models for each iteration, including reasoning time

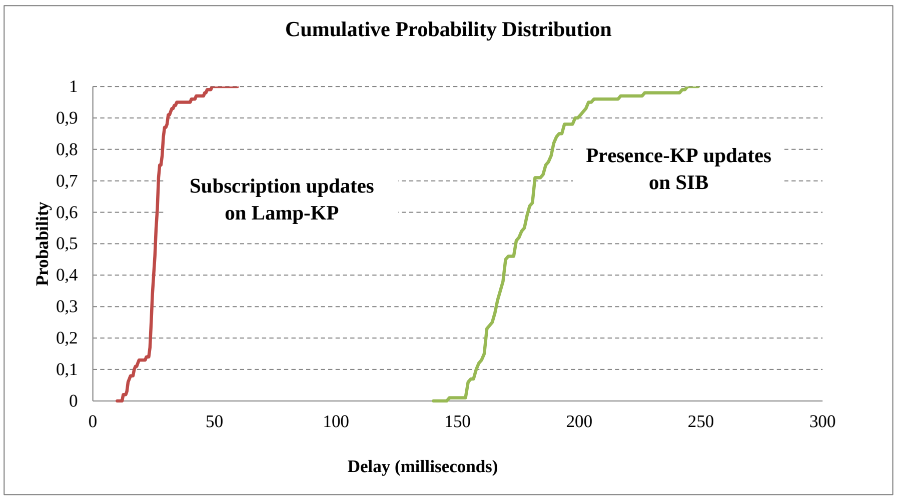


Figure 63: Cumulative probability distribution of delays between Presence-KP and SIB, as well as SIB and Lamp-KP

In Figure 63, the measurements show that the delay between the Presence-KP and the SIB is rather large with a considerable variance. The communication from the Presence-KP to the SIB consists of an update request and the related confirmation response. The average delay of 176.71 milliseconds is the largest component of the total end-to-end delay between the links. The communication from the SIB to the Lamp-KP consists of an indication event from the SIB due to a subscription and results in a query request from the Lamp-KP, terminated by a confirmation response by the SIB. There is some variance in the communication delay and the average delay of 25.87 milliseconds might become problematic when the SIB has to inform and handle multiple subscribers.

### 11.1.4 *Discussion*

During the pilot, most problems could be attributed to problems with the wireless network. A number of devices from different manufacturers experienced intermittent problems while connected to the SIB. For the purposes of the pilot, these devices were then connected to the SIB via ethernet. This does, however, demonstrate some of the problems with existing wireless networking technologies. It cannot be expected that a device with a wireless connection will always stay connected to the smart space, even when it is within range of the wireless router.

If we compare our query time measurements to the ones performed on Smart-M3 (4.4ms) and RIBS (0.65ms), we can see that the KPs were substantially slower: The Connector KP at 44ms, the Music Player KP at 74ms and the Sound/Light KP at 60ms. This can be attributed to additional network latency in the field study that approximated a real-world environment. Query measurements that were performed on the ADK-SIB directly (0.445ms) shows that it performs even better than RIBS, but this is not directly comparable as the measurements do not include network latency time. If network latency is taken into account, measured at 0.43ms per packet round-trip by [34], RIBS is still faster.

The subscription indication measurements of our setup are also significantly slower. While the Smart-M3 measurement was only 140ms, the Music Player KP measures around 860ms. This is mostly due to the additional time required for reasoning, which on average takes about 275ms. Reasoning improves the flexibility and capabilities of the SIB to such an extent that it is worth the hit in performance. Other contributing factors include the number of devices used, the number of triples and the network environment. While the Smart-M3 and RIBS measurements were made in a lab environment, our ADK-SIB measurements were made in conditions that approximate a real-world environment, including having a larger number devices active at the same time.

In the 1960s there already existed some controversy over the maximum allowable response times in human-computer interfaces [66]. It was shown that different human actions will have different acceptable response times. While the delay between pressing a key and visual feedback should be no more than 0.1-0.3 seconds, the response to a inquiry request may take up to 2 seconds. The performance measurements are still well within this two-second limit, indicating that from a user's point of view, when performing routine tasks, the system is responsive enough. The user study and interviews performed during the pilot seem to confirm this, as no participant indicated any issues with regards to the responsiveness of the system.

The scalability of the system could still be evaluated with larger triple sizes, but we do not foresee any scalability issues, due to the platform and software architecture used.

Not only the software architecture described in this thesis should be evaluated, but also the ontology that was developed during the three design iterations. This is the focus of the next section.

## 11.2 EVALUATING THE ONTOLOGY

### 11.2.1 *Introduction*

In the book Beautiful Data [92], the notion of beauty is described as "a simple and elegant solution to some kind of problem". In a paper on the notion of beauty when building and and evaluating ontologies, D'Aquin and Gangemi [25] argue that the GoodRelations[3] e-commerce ontology could be considered an example of a beautiful ontology. GoodRelations is an Web Ontology Language (OWL) 1 DL ontology that is used by stores to describe products and their prices and features. Companies using the ontology include, Google, BestBuy, Sears, K-Mart and Yahoo. It addresses a complex domain and covers many of the complex situations that can occur in the domain. It is well designed, ontologically consistent, lightweight and used extensively by practitioners in the domain.

Hepp [50], creator of the GoodRelations ontology, describes a number of characteristics that can be used to evaluate an ontology:

- High expressiveness: An ontology of higher expressiveness would be richly axiomatised in higher order logic, while a simple vocabulary would be of low expressiveness.
- Large size of community: As ontologies are considered community contracts, an ontology targeted towards a large community should be easy to understand, well documented and of a reasonable size.

*For more on community contracts, see Section 9.*

- Small number of conceptual elements: A larger ontology is more difficult to visualise and review. A reasoner could also take a long time to converge when the ontology is very large.
- Low degree of subjectivity: This is very much related to the domain of the ontology, where something like religion would be more subjectively judged than engineering.
- Average size of specification per element: The number of axioms or attributes used to describe each concept influences the ontological commitment that must be made before adopting the ontology.

3 http://www.heppnetz.de/projects/goodrelations/

We consider our ontology to be of higher expressiveness compared to other ubiquitous computing ontologies. Most of the existing technologies try to describe a large number of concepts, while the number of actual axioms used to describe these concepts are rather low. The size of the community is small at the moment, as only project partners in the Smart Objects For Intelligent Applications (SOFIA) project and students have been using the ontology up to now. Although the final version of our ontology contains 1192 asserted triples, the number of conceptual elements are low. There are only 34 OWL classes,29 object properties, 7 datatype properties and one magic property, making the ontology easy to visualise and review. Based on our experiments as described in the previous, the ontology also converges in a reasonable amount of time.

We now look at a method we developed to evaluate ontologies using the CD framework, a framework that has been used to evaluate notational systems and programming environments [46], and has also been used to evaluate two of the related projects described in Section 2.1: AutoHAN [15] and e-Gadgets [64].

### 11.2.2 *Validating the work using Cognitive Dimensions*

The CD framework is a broad-brush approach to evaluating the usability of interactive devices and non-interactive notations, e.g. programming languages and Application Programming Interfaces (APIs). It establishes a vocabulary of terms to describe the structure of an artefact and shows how these terms can be traded off against each other. These terms are, at least in principle, mutually orthogonal.

Traditional HCI evaluation techniques focus on 'simple tasks' like deleting a word in a text editor, or trying to determine the time required to perform a certain task. They are not well suited to evaluating programming environments or notational issues. The CD framework has been used to perform usability analyses of visual programming environments [46] as well as APIs [23]. Mavrommati et al. [64] used the framework to evaluate the usability of an editing tool that is used to manage device associations in a home environment.

Microsoft [23] used the CD framework to evaluate API usability, as part of a user-centred design approach to API design. Every API has a set of actions that it performs. However, developers browsing the API might not comprehend all the possible actions that the API offers. In a usability study they asked a group of developers to use an API to perform a set of tasks, and then asked a set of questions for each dimension. For example, for role expressiveness(see Section 11.2.2), the question was posed that when reading code that uses the API, if it was easy to tell what each section of the code does and why.

For our evaluation, we focused on a subset of cognitive dimensions that are related to ubiquitous computing ontologies and systems. What follows is a list of these dimensions, including a short description where necessary, as well as an example question.

LEVELS OF ABSTRACTION An abstraction is a grouping of elements that is treated as one entity, either for convenience or to change the conceptual structure [46]. *What are the minimum and maximum levels of abstractions?*

CLOSENESS OF MAPPING *How clearly did the available components map onto the problem domain?*

CONSISTENCY *When some part of the ontology has been learnt, how much of the rest can be inferred by the developer? Where there were concepts in the ontology that mean similar things, is the similarity clear from the way they appear?*

VISCOSITY To solve problems of viscosity, usually more abstractions (see earlier definition) are introduced in order to handle a number of components as one group, for example in object-oriented programming [46]. An example of viscosity is where it is necessary to make a global change by hand because the environment used does not have a global update tool. *How much effort was required to make a change to the environment?*

ROLE EXPRESSIVENESS The dimension of role expressiveness is intended to describe how easy it is determine what a certain part is for. *Are there parts of the ontology that are particularly difficult to interpret?*

HARD MENTAL OPERATIONS *Are there places where the developer needs to resort to fingers or pencilled annotation to keep track of what is happening? What kind of things required the most mental effort with this system and ontology?*

ERROR-PRONENESS *Did some kinds of mistakes seem particularly common or easy to make?*

HIDDEN DEPENDENCIES *If some parts were closely related to other parts, and changes to one may affect the other, are those dependencies visible?*

### 11.2.3 *Method*

The original CD questionnaire [14] was adapted for use with ubiquitous computing ontologies. Non-relevant questions were eliminated and some wording and questions were adjusted to the subject matter, without changing the fundamental meaning of the questions themselves.

*As mentioned in Section 6.2.2, our academic partners in Bologna published a journal paper [9] based on an independent implementation of our work.*

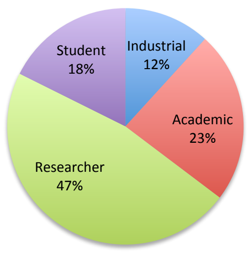
Developers of the SOFIA smart home pilot completed the questionnaire, as well as students and developers affiliated to the University of Bologna. The SOFIA developers used the ontology and software architecture for a couple of weeks in order to construct the smart home pilot. The students in Bologna took part in a course based on technology developed within the SOFIA project, where they also made use of the ontology and software architecture.

### 11.2.4 *Results*

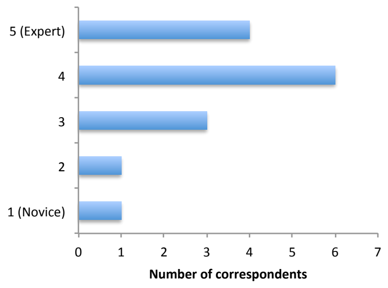
#### *Levels of abstraction*

> *Were you able to define your own concepts and terms using the system and ontology? Did you make use of different levels of abstraction?* An abstraction is a grouping of elements to be treated as one entity. In the ontology, these are defined as superclasses and subclasses, e.g. `PlayEvent` is a subclass of `MediaPlayerEvent`. Please indicate to what extent you made use of different levels of abstraction. If you did not use it, please indicate why.

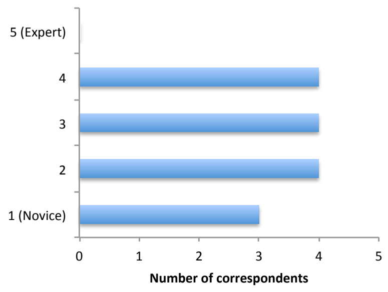
Most developers were able to make use of the existing concepts as defined, where the definition included different abstraction levels. Where necessary, developers were able to define their own concepts using different abstraction levels. Some of the developers used a simple ontology that did not require different levels of abstraction. As the level of knowledge about ontologies differed between different parties working on the same project, this necessitated the simplification of the ontology to a schema without semantics in some cases. Others avoided



(a) Background of correspondents



(b) Correspondents' experience with software engineering and programming



(c) Correspondents' experience with ontologies and semantic technologies

Figure 64: Correspondent demographics

ontological reasoning altogether by embedding the logic in the KP itself.

*Closeness of mapping*

> *How clearly did the available components map to the problem domain?* Did you have to define any of your own components, or were any special tricks needed to accomplish specific functionality?

Most of the developers experienced a clear, consistent mapping, with the domain mapped to already available components. In a few cases, developers developed their own components.

While it was easy to achieve the required functionality, it remains difficult to achieve component re-use. This becomes problematic for achieving emergent intelligent behaviour. It was also stated that more detailed descriptions of device capabilities, for example the coverage area of a presence sensor, are required.

*Consistency*

> *Where there were concepts in the ontology that mean similar things, is the similarity clear from the way they appear?* Are there places where some things ought to be similar, but the ontology defines them differently? Please give examples.

Most developers thought that similar entities in the ontology were subclassed correctly. It was indicated that `owl:sameAs` may be useful to indicate that different terms with the same meaning are in fact the same thing.

Developers not well acquainted with ontologies found it difficult to understand the difference between declaring entities using `rdfs:subClassOf` and declaring them as individuals or instances, as well as how to model an entity that contains another entity.

Similar entities were not always instantiated in the same way, for example no state information was available for some smart objects. Where multiple domain ontologies with similar concepts were used, these concepts were not aligned - most developers expected some upper (or core) ontology to align and unify main concepts. Concepts need a clear textual description and usage examples to make them easier to understand.

*Viscosity*

> *How much effort was required to make a change to the environment? Why?* How difficult is it to make changes to your program, the ontology or the system? For example, was it necessary to make a global change by hand because no global update tools were available?

Although different domains used different terms to define ontological concepts, most developers found it quite easy to make changes for ontological agreement. However, changes to the ontology sometimes necessitated changes at code level. In most cases, it was easier to adapt to changes on a semantic level, as the KP domain boundaries were well defined.

Using ontologies made it easier to allow for definition changes at run-time. Depending on the inferencing method used, changes to the ontology could require some existing inferences to be removed.

Embedded system developers found changes to be more difficult to implement, as it still required rebuilding images and downloading them to embedded boards for each modification. Some found it difficult to view changes made to the environment, due to a lack of tools to explore the contents of the SIB.

*Role expressiveness*

> *Are there parts of the ontology that are particularly difficult to interpret?* How easy is it to answer the question: 'What is this bit for?' Which parts are difficult to interpret?

Most responses indicated that the ontology was easy to understand. More clarifying comments inside the ontology could be useful - this can be implemented using the `rdfs:comment` field. Some developers indicated that application ontologies (ontologies that are device-specific) were still hard to interpret.

Some concepts might be instinctively interpreted differently, but the defined meaning became clear when viewed in context with the rest of the ontology. The ontology provides the structure that is necessary to make sense of the concepts.

*Hard mental operations*

> *What kind of things required the most mental effort with this system and ontology?* Did some things seem especially complex or difficult to work out in your head (e.g. when combining several things)? What are they?

Multiple developers indicated that ontologies are not an easy concept to grasp and that common practice is not always clear. Once familiar with ontologies, and understanding the specific ontologies involved, developers thought they were easy to use.

There is still effort required to make the system adaptive. This issue is also mentioned in Section 11.2.4.

*Error-proneness*

> *Did some kinds of mistakes seem particularly common or easy to make? Which ones?* Did you often find yourself making small slips that irritate you or make you feel stupid? What are some examples?

Some developers found it quite easy to mistype string literals, indicating that it would be better to define entities to represent strings that occur more than once. Mistyping URIs was another common error, as well as mixing up namespaces. Developers not familiar with ontologies also mixed up or misunderstood the differences between URIs and literals.

*Hidden dependencies*

> *If some parts were closely related to other parts, and changes to one may affect the other, are those dependencies visible?* What kinds of dependencies were hidden? How difficult was it to test the implemented system? Were there hidden faults that were difficult to find?

Multiple developers noted that using ontologies made the relationships among entities/parts more visible.

Changes to the ontology may affect others, therefore versioning and update notification are important. In the system used no errors were raised when components adhered to different versions of the ontology. Broken dependencies were only visible when the overall system failed.

One developer indicated that the publish/subscribe approach followed by the system architecture allowed for minimal dependencies due to loose coupling.

Some developers noted that it was difficult to determine when something did not work. This was especially noticeable in the case of subscriptions, were it becomes really difficult to understand why a certain subscription-based notification was not received.

One developer suggested that an ontology viewer could be used to make dependencies more visible, and that the Protégé ontology editor is too complex for most users.

*The Protégé ontology editor wa first mentioned in Section 4.6.*

### 11.2.5 *Non-CD related questions*

Some questions in the questionnaire were not directly related to the cognitive dimensions, but were meant to elicit more general responses to the usability of the ontologies and system.

**Question 1.** *What obstacles made it difficult to use the system?*

This question was based on a survey done by [88] to determine aspects that make APIs hard to learn. The question was phrased as follows:

> *What obstacles made it difficult for you to use the system?* Obstacles can have to do with the system itself, with your background, with learning resources etc. List the three most important obstacles, in order of importance (1 being the biggest obstacle). Please be more specific than the general categories mentioned here.

Responses included:

- Lack of background in ontologies
- Poor documentation
- Insufficient code examples
- SSAP poorly documented
- Difficult setup and installation procedure
- Lack of proper tools for viewing and exploring contents of SIB
- Reliability of subscription mechanism, especially on wireless networks
- QNames not supported on all platforms, requiring full Uniform Resource Identifier (URI) to be specified
- Ontology agreement
- Stability, performance, network issues

*SSAP is described in more detail in Section 10.3.*

*Names enable full URIs to be ubstituted by short prefixes.*

**Question 2.** *What did you appreciate most about the system and ontology?*

Responses included:

- Decoupling of interaction between components (i.e. all communication through broker, but could be single point of failure)

- Semantic interoperability between different devices, manufacturers and architectures
- Easy and quick to define new applications based on ontologies (after training)
- Using semantic connections to connect devices
- Ontology usable in different domains and more complex scenarios
- Ability to react to context changes through subscriptions

One respondent had the following insight: “Once you have agreed on the ontologies and the KP’s functionalities, you can focus on handling the various subscriptions and inserting the necessary triples. One does not have to focus on the communication protocols used or on the communication with the other components themselves.”

**Question 3.** *Can you think of ways the design of the system and ontology can be improved?*

Responses included:

- SIB discovery
- Agreement on ontological concepts to be used by a technical group
- Tools for ontology design (currently external tools are required)
- Better documentation
- Self-description of UI concepts and component functionality
- Hierarchic Smart Spaces
- Locality/routing/separation of message buses
- Authentication/security/locking
- Forcing a programming paradigm like Object-Oriented Programming (OOP) influences semantic treatment
- How to handle faulty smart objects

*These results were communicated by the author to the other project partners in the SOFIA project at a review meeting.*

### 11.2.6 *Discussion*

Ontologies allow developers to create additional levels of abstraction when the existing abstractions are not sufficient. The bigger issue seems to be unfamiliarity with ontologies, with some developers going so far as to embed all logic in the code itself in order to avoid using ontologies.

For the ontologies we defined, there seems to be clear mapping between objects in the domain and the ontological entities that they are mapped to. Adding additional components where necessary did not present any problems. One area that needs more attention is the extending the level of detail for device capability descriptions. Keep in mind, however, that too many low-level primitives create a cognitive barrier to programming [46]. It is not easy to deal with entities in the program domain that do not have corresponding entities in the problem domain. For example, having many ways to describe a presence sensor, when only one or two of these are relevant to the problem domain, makes it more difficult for the developer to comprehend.

When creating an ontology, it is important to provide clear textual descriptions, clarifying comments and usage examples for concepts to make them easier to understand. The GoodRelations e-commerce ontology is a good example of how this can be achieved.

*GoodRelations was first discussed in Section 11.2.1.*

Tools to explore the contents of the triple store more efficiently could decrease viscosity, as it would simplify viewing changes made to the environment and make dependencies more visible. Existing ontology viewers are still considered complex to use, usually only by ontology experts. Tools to automatically detect namespaces and prevent mistyping of URIs and strings used in the ontology would also be very useful.

In addition to the strengths and limitations of the software architecture and the ontology described in this chapter, we discuss more general conclusions and achievements in the next chapter.

Part IV

# APPENDIX