

An Ontology-based Programming Platform for Smart Artifact Systems

A dissertation submitted to the
Keio University

for the degree of
Doctor of Engineering

presented by

Bin Guo

Graduate School of Science and Technology
Keio University
2009

Abstract

Computing is moving towards everyday artifacts to make them “smart” and “intelligent”. By making use of the perceived contexts, smart artifacts can support a variety of human-centric applications. Due to the reasons such as privacy, personality and creativity, end users should be empowered to exert control over these applications, or even create new applications if they find existing ones cannot meet their particular needs. However, because existing smart artifact systems mainly rely on ad-hoc definitions of contexts, they don’t provide any reusable components that can facilitate other developers’ effort. Moreover, diversity of user ability and interest is not considered in existing user-oriented smart home tools.

To avoid having to start from scratch when building new human-artifact interaction systems, we proposed an ontology-based knowledge infrastructure called *Sixth-Sense*, which enables rapid prototyping of artifact-related applications. Unlike previous systems, *Sixth-Sense* builds upon the Semantic Web technologies, which includes a normalized ontology (called *SS-ONT*) definition to reflect vital aspects of human-artifact interactions. *Sixth-Sense* also supports semantically querying of collected contexts and includes a generic reasoning engine to derive higher level contexts from raw sensor data. In a word, our infrastructure paves the path for rapid prototyping of smart artifact applications.

Based on this context infrastructure, we developed an ontology-based programming model, called *Open-Programming*. Different from previous systems, this model is designed to meet both the “simplicity”, “high intelligence” requirement from novice users and the “functionality”, “in-depth” requirement from advanced users. For example, it enables an advanced user from *family-A* to create a rule-based game, and allows parents from *family-B* to reprogram it for their children to play through a simple front-end (according to their domestic settings and imagination). Because “error checking” is an important function as a programming tool, we also integrated several mechanisms to debug the programmed applications, including an object-relation-based reasoning method to deal with hardware errors in smart artifact applications.

A wide variety of applications are enabled by making use of our system, such as the real-world search service (e.g., searching lost objects) and artifact-based pervasive games. We also did a series of experiments to evaluate the performance of our infrastructure as well as the feasibility of the *Open-Programming* model. The results indicate that *Sixth-Sense* is a promising tool for users with different abilities to control and program smart artifacts in future homes.

Contents

Chapter 1. Introduction	1
1.1 Motivation.....	1
1.1.1 A Unified Knowledge Infrastructure	1
1.1.2 A Programming Toolkit.....	2
1.2 Contributions.....	3
1.2.1 Sixth-Sense Knowledge Infrastructure	3
1.2.2 The Open-Programming Model.....	4
1.2.3 Error-Checking Mechanisms	5
1.3 Organization.....	5
Chapter 2. Background and Related Work	7
2.1 Aspects of Context-Aware Computing.....	7
2.1.1 Application-Specific Context-Aware Systems	7
2.1.2 Smart Artifact Systems	7
2.1.3 Context-Aware Entertainments.....	8
2.2 Knowledge Representation and Context Modeling	9
2.2.1 Context Modeling Methods	9
2.2.2 Ontology-based Studies	11
2.3 End-User Programming at Home.....	13
2.3.1 Programming at Home, Why?	13
2.3.2 Programming at Home, How?	14
2.3.3 End-User Programming Tools	15
2.4 Robustness of Smart Artifact Systems.....	16
2.5 Summary and Conclusions	17
Chapter 3. A Knowledge Infrastructure for Smart Artifacts.....	18
3.1 An Overview of Sixth-Sense	18
3.2 An Ontology-based Context Model.....	20
3.2.1 SS-ONT Design	20
3.2.2 SS-ONT Domain Specific Ontology	21
3.2.3 User-Oriented Ontology Customization	27

3.2.4 SS-ONT Version Updating	29
3.3 Context Querying Engine	29
3.3.1 Two Query Methods	29
3.3.2 SPARQL-based Query Mechanism	31
3.4 Context Reasoning Engine.....	32
3.4.1 Ontology Reasoning.....	32
3.4.2 User-Defined Rule-based Reasoning.....	33
3.5 Run-Time Process of Sixth-Sense	37
Chapter 4. Open-Programming Model	39
4.1 Open-Programming Platform: an Overview	39
4.1.1 Combination of a Smart-Artifact Application	39
4.1.2 Open-Programming Model and Users	40
4.2 Establishing a Smart Artifact Environment	42
4.2.1 Indoor Sensing Techniques.....	42
4.2.2 Augmented Reality Techniques.....	44
4.2.3 Action Control Mechanism.....	45
4.3 Designing Customizable Services.....	47
4.3.1 Creating a New Service	47
4.3.2 Default Action Settings.....	49
4.3.3 Designing a Configuration Front-End	50
4.4 Customizing Services.....	53
4.5 Enabled Applications	54
4.5.1 Treasure Game	55
4.5.2 Real-World Search.....	58
Chapter 5. Error Checking Mechanism	60
5.1 Causes of Errors in a Smart Artifact Program	60
5.2 Software Error Checking	61
5.3 Hardware Error Checking.....	62
5.3.1 Physical Relations among Objects.....	62
5.3.2 Hidden Object Detection.....	64
5.3.3 Hidden Object Identification.....	69
5.4 Other Approaches to Deal with Uncertainties	71
Chapter 6. Evaluation Study	73

6.1 Evaluation Tasks	73
6.2 Performance Study	74
6.2.1 Evaluation of Runtime Performance	74
6.2.2 Evaluation of Effectiveness	77
6.3 Ontology Customization	80
6.4 Open-Programming Model	82
6.4.1 Evaluation of Rule-based Programming	83
6.4.2 Evaluation of Customization Mode	86
6.5 User Experiences	88
6.5.1 User Experience of the Real-World Search Service	88
6.5.2 User Experience of the Treasure Game	89
6.6 Interviews and Findings	91
Chapter 7. Discussions	93
7.1 Comparison with Related Studies	93
7.2 Open-Programming and OWL-S	95
7.3 Evolution of Sixth-Sense	97
7.4 Potential Improvement Areas	98
7.4.1 User Ontology Customization	98
7.4.2 Artifact-based Human Behavior Recognition	99
7.4.3 A Third-Programming Mode	99
Chapter 8. Conclusions and Future Work	101
8.1 Summary	101
8.2 Future Work	102
Acknowledgements	104
References	106

List of Figures

Figure 1.1: A scenario of Sixth-Sense	5
Figure 3.1: Overview of Sixth-Sense system.....	18
Figure 3.2: Three levels' design of SS-ONT	20
Figure 3.3: SS-ONT domain ontology model (partial).....	22
Figure 3.4: A screenshot of the Protégé-OWL editor	27
Figure 3.5: A screenshot of the “hasSensor” property-setting dialog.....	28
Figure 3.6: A partial graph model of SS-ONT	30
Figure 3.7: Different reasoning levels	34
Figure 3.8: Rule-based reasoning process	37
Figure 3.9: Sixth-Sense working process.....	37
Figure 4.1: Combination of a smart-artifact application.....	39
Figure 4.2: User operations in the Open-Programming model.....	41
Figure 4.3: U3D tags (top left), Mote sensors (top right) and prototypical smart artifacts (bottom).....	43
Figure 4.4: Ultrasonic receivers on the ceiling	43
Figure 4.5: A working example of the Kinotex pressure sensor.....	44
Figure 4.6: The Prot action device (left) and a working example (right)	44
Figure 4.7: The control structure of Prot	46
Figure 4.8: The Open-Programming model interfaces	47
Figure 4.9: Default action setting interfaces	50
Figure 4.10: A screenshot of the service-browsing page	53
Figure 4.11: Programming the Treasure game in the Open-Programming model	57
Figure 4.12: Real-world search interfaces	59
Figure 5.1: A hidden object scenario	61
Figure 5.2: Simulation checking process	62
Figure 5.3: A diagram of our error checking mechanism.....	63
Figure 5.4: Physical relations among objects	63
Figure 5.5: Spatial relations between smart objects.....	64
Figure 5.6: Hidden object detection scenarios	67

Figure 5.7: A screenshot of the extended real-world search service	69
Figure 5.8: One example for attribute matching.....	70
Figure 5.9: Logical relations among objects.....	70
Figure 6.1: Evaluation tasks.....	73
Figure 6.2: The test environment for Sixth-Sense	74
Figure 6.3: Performance of reasoning with the changes of ontology scale (left) and CPU speed (right)	75
Figure 6.4: Performance of querying	77
Figure 6.5: An exception case for R5.1	79
Figure 6.6: Evaluation results for ontology customization.....	81
Figure 6.7: Fragments of the questionnaire for rule-based mode testing	84
Figure 6.8: Testing results for rule R3.1 (left) and R3.2 (right)	84
Figure 6.9: Testing result for rule creation	85
Figure 6.10: User feelings about rule-based programming	85
Figure 6.11: Users in the gaming experiment (left) and the reprogramming experiment (right)	86
Figure 6.12: Testing results for the front-end based reprogramming	87
Figure 6.13: User feedback about real-world search service	89
Figure 6.14: Testing results for user experience of a game	90
Figure 7.1: Service construction in Open-Programming and OWL-S.....	96
Figure 7.2: A scenario of the third programming mode	100

List of Tables

Table 2.1: Database vs. Ontology	11
Table 2.2: Different user types and supporting systems	14
Table 3.1: Definitions about the components of a game in SS-ONT	26
Table 3.2: Concepts needed to be configured by users	28
Table 3.3: A sample query using three different query methods	30
Table 3.4: Instances of OWL ontology reasoning rules	33
Table 3.5: Instances of inference rules for human-artifact interaction	35
Table 4.1: Different action elements	46
Table 4.2: Basic rules used in the WeatherReporter service	49
Table 4.3: Rule-interface settings for the WeatherReporter service	51
Table 4.4: Front-end design and background information	52
Table 4.5: Front-end settings for the WeatherReporter service	52
Table 4.6: The Treasure game rules	56
Table 4.7: Front-end settings for the Treasure game	56
Table 6.1: Anticipated maximum reasoning time in different scales of smart spaces	76
Table 6.2: Experiment results of effectiveness	78
Table 6.3: Tasks for ontology customization	80
Table 7.1: Comparison with other related systems	93
Table 7.2: Open-Programming and OWL-S	96

Chapter 1. Introduction

1.1 Motivation

The emergence of new types of simple, cheap, interconnected sensors and enabling technologies for ubiquitous computing are driving the extension of the computing domain from general computers or computer-augmented appliances (e.g., household appliances) to other facets of everyday life. Smart artifacts, which are mundane, everyday objects (e.g., cups) equipped with computing capabilities, will become part of everyday life in the near future. By making use of information perceived from attached sensors, smart artifacts can cooperate to support a range of human-centric services, such as acting on a person's behalf, anticipating a person's activities or needs, and delivering alerts or other assistance in an "anywhere, anytime" fashion.

There have been recently many smart artifact systems developed by researchers. However, because not all of the domestic settings and user considerations could be accurately known by system developers, for example, facts like "*this diary is privately owned by Bob*", and particular needs like "*reminding me to take that 'black wallet' before I leave home*", there also emerges a trend to empower end users who have intimate knowledge about their living environments to program at home. Our research is mainly focused on how to facilitate both experts and end users to program smart artifacts in future homes.

A programming platform for smart artifact systems implies that we want to build a bridge between raw sensor data and various applications. There are several requirements for building such a bridge, as we mentioned below.

1.1.1 A Unified Knowledge Infrastructure

Designing and developing smart-artifact applications have been drawing much attention from researchers in recent years. However, smart-artifact services have never been widely available to everyday use. Recent research results show that building and maintaining smart-artifact systems is still a complex and time-consuming task due to the lack of an adequate infrastructure support. We believe such an infrastructure requires the following supports:

1.1 Motivation

(1) *A formal context model that can facilitate context sharing and reuse among different smart artifacts and context-aware services.* Raw context data obtained from various sources comes in heterogeneous formats, and applications that do not have prior knowledge of the context representation cannot use the data directly. Therefore, a single method for explicitly representing context semantics is crucial for sharing common understandings among independently developed smart-artifact systems. A unified knowledge-representation scheme also makes it easy to implement knowledge-reuse operations, such as extending or merging existing context definitions. Without such a common context model, most smart-artifact systems have to be written from scratch at a high cost.

(2) *It can easily integrate with generic reasoning and querying engines.* Context reasoning is a key aspect of context-aware systems because high-level contexts (e.g., *is there anyone near the book? what is the person doing?*) cannot be directly provided by low-level sensor contexts (e.g., *the lighting level of a room, the 3D coordinate data of a book*). On the contrary, they have to be derived by reasoning. A context querying engine is also important for context-aware systems, which allows applications to selectively access the contexts they concern by writing expressive queries.

As a platform that is intended to work for users from different families, the main challenge to realize this requirement is how to make the knowledge infrastructure “sharable” and “easily-customizable”, because different families usually have different domestic settings, daily routines, and user considerations.

1.1.2 A Programming Toolkit

As a programming platform, another requirement is obvious. That is, we should provide an integrated toolkit that can facilitate users’ effort to program smart artifacts. Because users are different from each other on their ability and interest, one design challenge to build such a toolkit is that there needs a trade-off between “*simplicity*” and “*functionality*”. For novice users, simplicity seems to be the most important thing. However, a toolkit that is too simple to use often implies that its functionality is limited, which inhibits the design of high-quality applications. On the other hand, for experienced users, a system that allows them to exert in-depth control over their smart homes is more acceptable. Therefore, measures should be taken to meet different user requirements.

One of the significant issues relevant to ubiquitous computing is the problem of dealing with uncertainty and failure in sensing, wireless communication as well as reasoning. Smart artifact systems are faced with the same problems. In general, there might be two kinds of errors to a programmed smart artifact application, they are, *software errors* (e.g., a wrong human input) and *hardware errors* (e.g., a broken sensor embedded in a smart artifact, which caused the object a “hidden” one). Therefore, to ensure developed applications work in the right way, an error-checking mechanism that can deal with programming errors and fallible sensors is also important for our programming toolkit.

1.2 Contributions

This thesis investigates the challenges and solutions surrounding building a programming platform for smart artifacts in future homes, which seeks to cope with some of the issues presented in last section, for example, *how to build a sharable knowledge infrastructure for smart artifacts*, *how to answer diverse user requirements about the programming toolkit*, *how to deal with errors in a programmed smart artifact application*.

1.2.1 Sixth-Sense Knowledge Infrastructure

To facilitate rapid prototyping of smart artifact applications, we proposed a new system infrastructure called *Sixth-Sense*. Unlike similar studies that mainly use ad-hoc data structures to represent contexts, *Sixth-Sense* explores the Semantic Web technology [Berners-Lee *et al.* 01] to define a common ontology that can assist the development of human-artifact interaction systems. The *Sixth-Sense Ontology (SS-ONT)*, expressed by the Semantic Web language OWL (Web Ontology Language) [Smith *et al.* 03], reflects portion of contexts that typically exist in human-artifact interactions:

- Artifact properties (static information like “*the size or shape of an object*”) and status (e.g., dynamic information like “*this object is tilted*”), which reflect information obtained from heterogeneous data sources. For example, with the temperature data from a cup, we can ascertain if the teacup is filled with hot water.
- Physical relations among objects (e.g., the spatial relation involved in an object being *on* or *near* another one) and the relationships between humans and objects (e.g., the artifact a human is interacting with). With these contexts, it is possible to recognize

1.2 Contributions

the behavior of a human drinking.

- Logical relations among objects (e.g., the functional relation between a toothbrush and a glass), which are derived from common sense knowledge. Compared to physical relations, logical relations among objects can sometimes provide more precise and direct information (e.g. based on the known functional relationship, we can guess that the object placed in a glass is a toothbrush and not a pen).

Benefiting from the hierarchical definition structure and semantic sharing natures of the ontology, *SS-ONT* enables home-knowledge sharing and customization among different families. By exploring this formal context modeling method, *Sixth-Sense* also integrates several standardized approaches and tools that support expressive querying and reasoning of defined facts and contexts. In a word, the *Sixth-Sense* infrastructure builds a good foundation to support the development of a wide variety of smart artifact applications. A series of experiments have been conducted to evaluate the performance as well as the effectiveness of our knowledge infrastructure.

1.2.2 The Open-Programming Model

To address the diversity of user abilities and interests, we proposed a novel approach to support end-user programming in smart environments. The ontology-based programming model, called *Open-Programming*, is based on the two design principles, “*simplicity*” and “*functionality*”. For experienced users, like expert users and advanced users, they can create high quality, flexible services through a generic rule-language and publish them as shared services to a Web server. For the majority average home users who have no ability or willing to create new services, they can search among the shared services and, according to their domestic settings and preferences, reprogram them through a simple front-end.

Various human-centric applications can be developed based on the *Open-Programming* model, such as context-aware services and artifact-based games. One scenario that describes the working of this model is shown in Fig. 1.1. In this scenario, an artifact-based game application created by an advanced user from *family-A* can be published to a Web server and shared by other families. For example, parents from *family-B* can search and reprogram this shared game according to their home resources (e.g., smart artifacts, user-generated contents like photos and video clips) and imagination.

Their children can enjoy different game plays when their parents assign different settings in the “reprogramming” or “customizing” process. The results from a user study indicate that our *Open-Programming* model provides a promising way for users with different abilities to program smart artifacts at home.

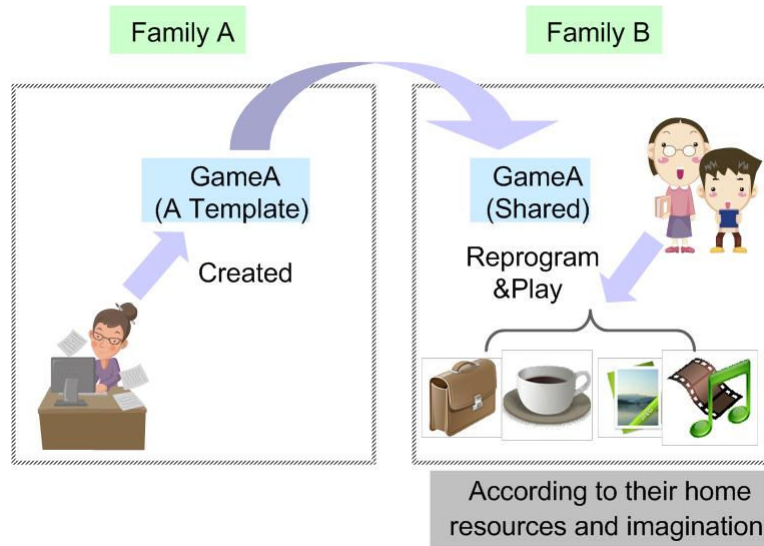


Figure 1.1: A scenario of Sixth-Sense

1.2.3 Error-Checking Mechanisms

Robustness is significant for smart artifact systems. In this work, we integrated a simulation mode for users to detect possible programming errors in their applications. Moreover, a novel mechanism to handle fallible sensors in smart artifact systems is also proposed. Different from other studies, hidden objects in our system can be detected using a set of user-defined rules. The rules are mainly abstracted from the knowledge of physics and various physical relations (e.g., spatial relations, motion relations) among objects. The evaluation results indicate that this approach can, to a certain extent, improve the robustness of smart artifact systems.

1.3 Organization

The organization of the rest of this thesis is as follows:

Chapter 2 reviews previous research into smart artifact systems. This encompasses context-aware systems, context modeling methods, ontology-based studies, programming tools for end users, and methods to deal with uncertainty and failure in context-aware systems.

1.3 Organization

Chapter 3 presents the foundation of our work, *Sixth-Sense infrastructure*, including an ontology-based knowledge definition for smart artifacts and several normalized reasoning and querying mechanisms based on this definition.

Chapter 4 presents the *Open-Programming* model, an ontology-based programming model that is designed to address the diversity of user abilities and interest. A detailed description of its principles, implementation and some enabled applications will be described in this chapter.

The implementation of the error-checking module of our system will be presented in Chapter 5.

In Chapter 6, we present the evaluations of our system, which consists of several aspects, including the performance and effectiveness measurement of our knowledge infrastructure, the usability validation of the *Open-Programming* model, as well as user studies on ontology customization and their feedback about the enabled smart artifact applications. Some lessons and findings learned from the evaluation process are also reported here.

A comparison of our work with other related studies and discussions of a few potential areas to improve our system are presented in Chapter 7.

Finally, Chapter 8 presents a summary of this research and indicates areas of possible future work.

Chapter 2. Background and Related Work

Research in context-aware computing is very diverse as the field itself has not yet been clearly defined. Researchers from different communities make efforts to understand and improve concepts, technologies as well as applications for computing beyond desktop computers, and undertake research in ubiquitous computing. This chapter presents background information on smart everyday artifacts and context-aware computing, and reviews related work carried out by other researchers. Finally, we will give a summary of the remaining issues to be addressed in our work.

2.1 Aspects of Context-Aware Computing

Since Weiser's vision of ubiquitous computing [Weiser 99] more than a decade ago, many groups have explored the domain of context-aware applications.

2.1.1 Application-Specific Context-Aware Systems

In the earlier stage of this research, most studies were built to demonstrate the usefulness of context-aware computing in smart environments and they mainly focused on specific applications. AT&T Lab's Active Badge project provides a means of locating persons within a building by determining the location of their active badge [Want *et al.* 92]. Georgia Tech's Aware Home project focuses on building a sensor-rich living environment that is aware of its occupants' activities [Kidd *et al.* 99]. Microsoft's EasyLiving team developed a camera-based person detection and tracking system that can be used to detect users' presence and adjust environment settings to suit their needs [Brumitt *et al.* 00]. The HP's Cooltown project provides physical entities (e.g., people and objects) with "Web presence" and lets users navigate from the physical world to the Web by picking up links to Web resources using many sensing techniques [Kindberg *et al.* 02]. These projects contribute much to context-aware computing research for they explored different ubiquitous computing features.

2.1.2 Smart Artifact Systems

In recent years, context-aware computing is extending from computers and mobile devices to everyday artifacts. Numerous reports of prototyping research on designing and

2.1 Aspects of Context-Aware Computing

developing smart-artifact systems have been reported [Beigl *et al.* 01, Beigl *et al.* 03, Lampe *et al.* 03, Philipose *et al.* 04, Siio *et al.* 03, Yap *et al.* 05]. Enabled human-centric applications, such as helping people quickly locate indoor objects [Yap *et al.* 05], inferring human activities or behaviors from their interactions with everyday objects [Beigl *et al.* 01, Beigl *et al.* 03, Philipose *et al.* 04], or assisting humans to manage or organize their belongings [Lampe *et al.* 03, Siio *et al.* 03], have been reported in these studies. These studies demonstrate the usefulness of smart artifacts in future homes. However, the problem to these systems is that they are typically proprietary and due to the ‘ad-hoc’ approach they deployed to obtain and process context information, they do not provide basic structures or reusable components to ease the creation of context-aware systems.

2.1.3 Context-Aware Entertainments

Ubiquitous entertainments have been another research trend in context-aware computing field. For example, *pervasive gaming*, a new kind of entertainment that aims at combining the properties and advantages of both the physical world and the virtual world, has become an interest of many researchers. To put it simple, pervasive games are “*computer-augmented games to be played in physical environments, stressing the physical and social nature of the game*” [Magerkurth *et al.* 05].

The development of pervasive games is still in its early stage. Most projects in this field were built to mainly demonstrate the concepts or the usefulness of this new technology. Existing pervasive gaming studies are diverse in the technology they used and the form they presented. One approach is to augment traditional, real-world games with computing functionality, enabled games include smart toys like StoryToy [Fontijn *et al.* 05], storytelling games like StoryRoom [Montemayor *et al.* 04, Alborzi *et al.* 00] and KidsRoom [Bobick *et al.* 99], location-aware games like CYSMN [Benford *et al.* 06], and augmented tabletop games like STARS [Magerkurth *et al.* 04]. Another approach, on the contrary, attempts to map computer games onto real-world settings utilizing augmented-reality techniques, as demonstrated by Touch Space [Cheok *et al.* 02] and Human Pacman [Cheok *et al.* 04]. More information about pervasive games can be found in Magerkurth *et al.* [05] and Hinske *et al.* [07]. The approach to increase physical interaction in computer games has also been followed recently by a few commercial

systems, such as Dance Dance Revolution [Hoysniemi 06], a dancing game that requires players to step on pressure sensitive tiles in time with music, and Wii Sports [Yim *et al.* 07], a video game that allows players to physically mimic the actions required to play sports using a sensor-augmented console.

Existing systems have greatly contributed to context-aware entertainment research by emphasizing the natures of ubiquitous interaction. The major problem to them is that they mainly explored the resources that were specially designed for a certain gaming experience, there lacks a general platform which can make use of easily-found resources (e.g., everyday artifacts, user generated contents—like photos and video clips) in a smart environment. For example, physical props and playing fields are two key resources of pervasive games [Hinske *et al.* 07]. However, as envisioned in previous studies, there has been a high cost to arrange props, for they either offer proprietary props [Fontijn *et al.* 05, Montemayor *et al.* 04, Magerkurth *et al.* 04, Cheok *et al.* 02, Cheok *et al.* 04] or ask users to make props manually using some materials (e.g., cardboard boxes [Alborzi *et al.* 00]). The playing field in existing systems ranges from a table [Magerkurth *et al.* 04] to a city [Benford *et al.* 06]. Though it demonstrates well the ubiquitous feature of pervasive games, some issues are also raised. For example, some designated gaming spaces are not commonly found in our life (e.g., specially-designed storytelling corners in a room [Montemayor *et al.* 04, Bobick *et al.* 99]) and some are even unsafe for players (e.g., playing on the street [Benford *et al.* 06]).

2.2 Knowledge Representation and Context Modeling

According to Dey’s definition [Dey *et al.* 00], context is “*any information that can be used to characterize the situation of an entity*”. An entity is a person, or object that is considered relevant to the interaction between a user and an application, including the user and application themselves. In his definition, the knowledge definition of a particular domain is also regarded as contexts (i.e., static contexts). There have been previous many methods on how to represent contexts and knowledge in context-aware systems.

2.2.1 Context Modeling Methods

Numerous approaches have been explored for context modeling. Most studies are based on informal context models, such as using simple data structures (e.g., key-value pairs)

2.2 Knowledge Representation and Context Modeling

[Beigl *et al.* 01, Philipose *et al.* 04] and data formats (e.g., pictures) [Siio *et al.* 03], relational database schemes [Yap *et al.* 05], XML [Lampe *et al.* 03], and programming objects (e.g., Java classes). As these systems rely on ad hoc representations of contexts, independently developed applications cannot interoperate based on a common understanding of context semantics. Furthermore, because the expressive power involved in using such ad hoc schemes is quite low, it is difficult for them to perform context reasoning by introducing generic inference engines. As a result, the reasoning tasks of these systems are mostly implemented as programming procedures, which makes the overall systems inflexible and difficult to maintain.

Recent research work has focused on providing toolkit or infrastructure support to tackle the above issues. The pioneering work of Context Toolkit provided an object-oriented architecture for rapid prototyping of context-aware applications [Salber *et al.* 99]. It gives developers a set of programming abstractions which separate context acquisition from actual context usage and reuse sensing and processing functionality. The Smart-Its project proposed a generic layered architecture for sensor-based context computation, including separate layers for feature extraction and abstract context derivation [Beigl *et al.* 03]. Henricksen *et al.* [02] model context using both ER and UML models, where contexts can be easily managed with relational databases. A logic-programming based platform for context-aware applications is described in LogicCAP [Loke 04].

To serve as the context modeling method of a user-oriented platform for smart homes (like our system), it must meet several requirements: First, it can easily represent the heterogeneous, semantically-rich relationships among physical (like smart artifacts and humans) and logical entities (like software agents); Second, it should be “sharable” among different families and be “easily-extendable” to meet the evolving environment (e.g., a new kind of sensor is deployed or a new relationship between two object classes is needed to be added); Third, it can work easily with an inference engine to interpret collected contexts; Finally, it should provide an “easy-to-use” tool for users to customize the knowledge definition of their home. However, the context management studies mentioned above cannot fulfill these requirements: (1) the expressive power of the methods they used are still low (e.g., semantics like a symmetry property “*isNear*” and taxonomy information cannot be easily represented by them); (2) they don’t provide

adequate support for organizing contexts in formal structure format, which makes it difficult to realize knowledge sharing and reuse among different families; (3) they provide no generic mechanism for context querying and reasoning. Though LogicCAP introduced a formal logic-based model for context reasoning, it didn't support knowledge sharing and was difficult to be maintained by home users; (4) existing methods are mainly designed for experts, and they don't provide any easy-to-use interface for end users. To address these issues, recently there emerges another way — an ontology-based approach, to model contexts, as surveyed in the next section.

2.2.2 Ontology-based Studies

The term “ontology” has a long history in philosophy, in which it refers to the subject of existence. In the context of knowledge management, ontology is referred as the shared understanding of some domains, which is often conceived as a set of entities, relations, axioms, facts and instances [Uschold *et al.* 96].

Table 2.1: Database vs. Ontology

<i>Axis of Comparison</i>	<i>Database Schemes</i>	<i>Ontology Schemes</i>
Modeling perspective	Intended to model data being used by one or more applications	Intended to model a domain
Structure vs. Semantics	Emphasis while modeling is on structure of the tables	Emphasis while modeling is on the semantics of the domain – emphasis on relationships, also facts/knowledge.
What concerns	Efficiency and effectiveness on data processing	Expressiveness, semantic sharing, logical reasoning

Table 2.1 gives a comparison of database schemes and ontology schemes. In contrast to database-based knowledge infrastructures, context modeling based on ontology is more normalized and promising [Strang *et al.* 04], which can at least offer the following four advantages. First, ontology is based on a set of normalized standards and it can explicitly represent the knowledge of a domain [Uschold *et al.* 96]. Second, an ontology provides a shared vocabulary base, which enables the sharing of context semantics, and its hierarchical structure facilitates developers to reuse and extend domain ontologies. Third, using ontology as a knowledge infrastructure allows the underlying system implementations to be better integrated with various existing logical reasoning mechanisms. Finally, there are free, mature and user-friendly ontology editors, like Protégé-OWL editor (available at: “<http://protege.stanford.edu/>”). These four characters

2.2 Knowledge Representation and Context Modeling

just meet the requirements mentioned in last section, so our system explored the ontology-based method to build our context model.

Semantic Web [Berners-Lee *et al.* 01] is an effort that has been going on in the W3C to provide rich and explicit descriptions of Web resources. The essence of it is a set of standards for exchanging machine-understandable information. Among these standards, Resource Description Framework (RDF) provides data model specifications [Brickley *et al.* 00], and Web Ontology Language (OWL) enables the definition and sharing of domain ontologies [Smith *et al.* 03]. Using OWL, one can (1) formalize a domain by defining classes and the properties of those classes, (2) define individuals and assert their properties, and (3) reason about these classes and individuals. From a formal point of view, OWL is rooted in description logic (DL), which allows OWL to exploit DL reasoning tasks such as class consistency and consumption [Smith *et al.* 03].

Web ontology and other Semantic Web technologies have been recently employed in modeling and reasoning contexts in different ubiquitous computing domains. Ranganathan *et al.* [03] developed a middleware for context awareness, where they represented context ontology written in DAML+OIL. Chen *et al.* [04a] proposed an agent-oriented Context Broker Architecture (CoBrA) infrastructure for semantic context representation and privacy control. Semantic Space is a pervasive computing infrastructure that exploits Semantic Web technologies to support explicit representation and flexible reasoning of contexts in smart spaces [Wang *et al.* 04]. Yamada *et al.* [05] explores ontology to describe the knowledge of digital information appliances in home networks. Ontology has also been employed in sensor network systems to deal with some particular issues, such as relation-based data search [Lewis *et al.* 06] and the adaptive problem of sensor networks [Acanha *et al.* 04].

The above studies explored ontology to cope with different problems in different domains, such as context modeling in context-aware systems and problem description in sensor networks. However, none of them address ontology-based programming, for example, the concept that developing a programming toolkit that explores the knowledge sharing nature of an ontology to support cooperative development among users is not presented in them. Moreover, though ontology-based method has been employed into many domains for context modeling, there lacks an ontology definition for human-artifact

interaction systems. Because so many interactions in daily life occur between people and everyday artifacts (such as books, toothbrushes, etc.), ignoring such rich and valuable contexts will limit the reasoning tasks that a knowledge infrastructure can support. For example, inferring whether a teacup is filled with tea, or recognizing an artifact-related behavior such as a person drinks, cannot be implemented by existing studies. Finally, all ontologies in above systems are defined by experts, and there lacks a study to investigate whether end users can exert control over ontology definitions. Because home users have the most intimate knowledge about their living environments, they should be allowed to manage the ontology definition of their home.

2.3 End-User Programming at Home

For users of all ages, learning to program can be a very difficult task. Since early 1960's, there have been a number of programming languages and environments with the intention of making programming accessible to a larger number of users, as surveyed by Kelleher *et al.* [05].

2.3.1 Programming at Home, Why?

In recent years, with the development of ubiquitous computing technology, there also emerges a trend that aims at empowering end users to create context-aware applications in smart homes. There are several reasons for learning to program at home: First, in order for smart homes to achieve their promise of significantly improving the lives of families through socially appropriate and timely assistance, they will need to sense, anticipate and respond to activities in the home. Interestingly, expanding system capabilities can easily overstep some invisible boundary, making families feel at the mercy of, instead of in control of that technology [Davidoff *et al.* 2006]. Therefore, seeking to be more sensitive to users becomes one major reason for smart home programming. Second, our homes are changing rapidly with the introduction of sensors into many everyday artifacts. Allowing users to program their home can increase their knowledge and acceptance of new smart objects. Third, not all domestic settings and user considerations can be accurately known by application developers, so home users who have more intimate knowledge about their living environments and daily routines than “hired programmers” should be allowed to exert control over these services in terms of their demands, as reported in Dey *et al.* [06].

2.3 End-User Programming at Home

Finally, there are also funny and education considerations, such as allowing users to enjoy self-designed entertainments at home and drawing some users into programming activities at an early age [Mattila *et al.* 06].

2.3.2 Programming at Home, How?

In light of the reasons mentioned in the last subsection, there is no doubt that it is vital to import end-user programming at home, but another problem emerges, that is, *how much control is appropriate for end users?* In current desktop computing environments, most home computer users find themselves becoming system administrators, for they have to concern themselves with chores that would seem familiar to a mainframe system operator: updating hardware, performing software installation and removal, learning how to use a software, and so on [Edwards *et al.* 01]. Within a coming smart home era, our homes will be filled with much more smart objects and context-aware services, the operations under which will be more complex and time-consuming than in current desktop computing era. Lacking of ability or interest to deal with such chores will be a big design challenge to smart home developers.

Table 2.2: Different user types and supporting systems

<i>User Type</i>	<i>Technical Level</i>	<i>Capability</i>	<i>Supporting Systems</i>
Expert users (Programmers)	High	Creating services using APIs, toolkits, and libraries	CoBrA, Semantic Space
Advanced users	Middle	With certain technical skills, willing to experience various new techniques	Alfred, Jigsaw, CAMP, iCAP
Average users (Novices)	Low	With little or no technical experience, do not like to exert complex control	–

As illustrated in Table 2.2, according to their different technical levels, end users can be broadly classified into three types: *expert users*, *advanced users*, and *average users*. Average users (or novices) have little technical experience and they may lose interest in controlling a smart home if the operations are complex. In contrast, advanced users and expert users are experienced computer users and they are willing to experience various new techniques. They prefer to exert in-depth control over their smart living environments. The existence of diverse user types yields a third problem: *how to meet their different requirements?* Before answering this question, we will first give a survey of previous end-user programming methods.

2.3.3 End-User Programming Tools

There have been recently a few studies that focus on empowering users to program at home. Some of them seek to create a tool that can help users to easily control their smart homes. For example, the Alfred project uses simple speech-based commands to support end user programming in intelligent environments [Gajos *et al.* 02]. The Jigsaw project uses a jigsaw-puzzle metaphor based interface to allow end-user configuration of smart devices [Humble *et al.* 03]. A magnetic-poetry metaphor based system to enable end users to create context-aware applications is described in CAMP [Truong *et al.* 04]. AutoHAN is a software architecture that enables user-programmable specification of the interaction between appliances in a domestic house [Blackwell *et al.* 01]. iCAP allows users to exert control over a sensing environment through a visual, rule-based programming tool [Dey *et al.* 06]. There are also some projects that concern game creation in specially designed playing fields. StoryRoom proposes a physical programming approach that allows children to control sensors and actuators in a playing field [Montemayor *et al.* 04]. A visual programming platform that empowers children to create games in interactive playground environments is reported in UbiPlay [Mattila *et al.* 06]. According to the approach they used, previous end-user programming systems broadly fall into two groups, namely *visual programming* and *physical programming*.

(1) *Visual programming*: Many novices are struggled with the syntax problems (e.g., remembering the order of parameters) when programming using texts [Kelleher *et al.* 05]. Visual programming tools attempt to provide alternative input mechanisms to bypass these problems. Two major ways are available to build a visual programming environment. Some systems use graphical objects to represent elements of a program such as action commands and control structures. These objects can be moved around and combined in different ways to form programs, as demonstrated in iCAP [Dey *et al.* 06] and UbiPlay [Mattila *et al.* 06]. Utilizing explicit, easy-to-understand metaphors is another way to teach beginners how to program, which has also been exploited in many previous studies, as reported in Jigsaw [Humble *et al.* 03] and CAMP [Truong *et al.* 04].

(2) *Physical programming*: Unlike studies in the prior group that ask users to work with programming techniques on the screen, systems in this group attempt to create an entirely physical or tangible programming environment. As defined in Montemayor *et al.*

2.4 Robustness of Smart Artifact Systems

[04], physical programming is “*the creation of computer programs by physically manipulating computationally augmented objects in a smart environment*”. StoryRoom [Montemayor *et al.* 04] and AutoHan [Blackwell *et al.* 01] are two good examples that exploit the physical programming method to assist end-user developers.

These systems indeed lower barriers to allow users who don't have special technical skills to program. However, the problem to them is whether users have ability or interest to utilize these programming tools. For example, to create a simple application in iCAP, though no coding is needed, users still have to perform a series of “chore-like” operations, such as drawing icons, creating elements, specifying parameters and dealing with ambiguity, which are time-consuming and complex for average home users. Therefore, it's an impractical hypothesis that end-users would be interested in creating applications by using these visual toolkits. One study that supports our view was reported in Barkhuus *et al.* [03], which investigated that most people were willing to give up partial control over context-aware applications if the reward in usefulness was great enough. Another similar viewpoint came from Edwards *et al.* [01], where they suggested that in a utility model, most of the “intelligence” should be resided in the system itself, while only the most simple and minimal “front-end” functionality was needed to be administrated by end users. As shown in Table 2.2, visual-based programming systems are probably useful for part of advanced users. However, the constrained functionality and limited expression ability of these systems often inhibit them from building more interesting applications. Therefore, for a significant portion of advanced users, existing systems can not meet their requirement on “high-quality” and “functionality”.

2.4 Robustness of Smart Artifact Systems

Keeping the robustness of ubiquitous computing applications is important for end users to accept them and use them. There are several reasons that may influence the robustness of smart-artifact systems, such as fallible sensors and inconsistent contexts (or uncertainty). There have been previous some studies on how to handle uncertainty in context-aware systems. Chen *et al.* [04a] presented an assumption based reasoning method for resolving information inconsistency. Gaia project allowed context-aware services to reason about uncertainty using Bayesian networks [Ranganathan *et al.* 04]. CYSMN surveyed the causes of uncertainty in a location game and presented several strategies to deal with

them [Benford *et al.* 03]. These systems exploited different reasons that may cause uncertainty to context-aware systems, and explored nonmonotonic techniques to deal with inconsistent contexts. However, none of them concern the healthy of embedded sensors in smart homes. Benford *et al.* [03] had once suggested that for the technical problems that were difficult to solve, we could reveal them to users for their attention. But they didn't provide any ways to detect the problems caused by sensors.

2.5 Summary and Conclusions

A range of characters, challenges and approaches for context-aware systems and user-oriented programming tools has been surveyed and analysed in this chapter. The research surveyed also lead to the following observations. Based on these observations issues to take on in the course of this research were identified:

- As existing smart artifact studies use “ad-hoc” approach to define and interpret contexts, there lacks a unified knowledge infrastructure to support rapid prototyping of smart artifact applications. To address this, we proposed a new system infrastructure, named *Sixth-Sense*, which explores the Semantic Web technologies to define a common ontology to assist the development of human-artifact interaction systems. The *Sixth-Sense Ontology (SS-ONT)*, expressed by OWL, provides standardized approaches and support tools for knowledge sharing and logical reasoning. We present this infrastructure in Chapter 3.
- As seen from previous programming toolkits for home users, diversity of user ability and interest is not considered in them. To deal with this issue, we have developed a new programming model, *Open-Programming*, which is based on two design principles, “*functionality*” and “*simplicity*”, and enables users with different abilities to exert control over smart home services. We will describe this model as well as some applications based on it in Chapter 4.
- Existing smart home programming tools do not include an error-checking mechanism to deal with possible hardware and software errors in the programmed applications. In our work, we have proposed a rule-based mechanism to deal with fallible sensors in smart artifact systems, which explores physical and logical relations among objects, as will be described in Chapter 5.

Chapter 3. A Knowledge Infrastructure for Smart Artifacts

To avoid having to start from scratch when building new human-artifact interaction systems, we developed an ontology-based knowledge infrastructure, which supports explicit representation of contexts and can easily work with generic querying and reasoning tools. In this chapter, we will first describe the structure of the whole system and then present the implementation of its knowledge infrastructure.

3.1 An Overview of Sixth-Sense

This section will give an overview of our system, *Sixth-Sense*, which is an ontology-based programming platform for smart artifact systems. We call it *Sixth-Sense* because all applications created in our system follow a rule-based paradigm, and this word depicts well the intuition beyond rules, that is, the new facts derived from these rules. As shown in Fig. 3.1, *Sixth-Sense* consists of three different layers and an error-checking mechanism.

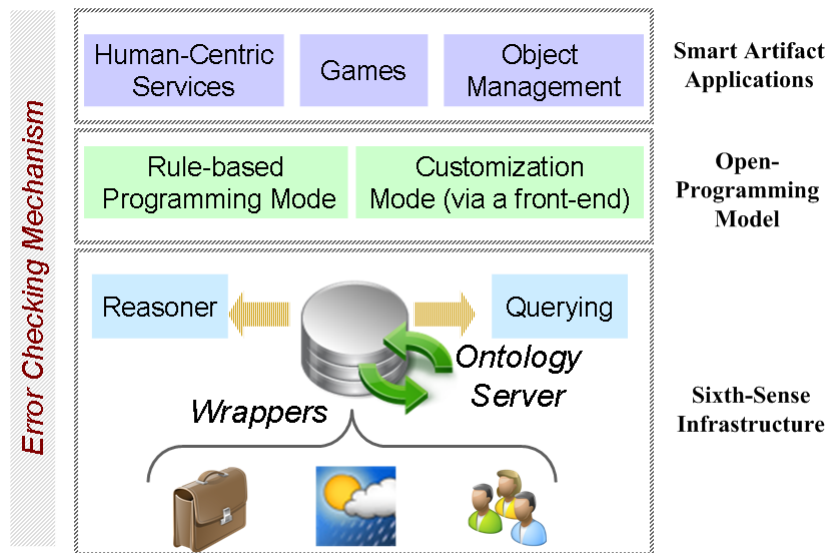


Figure 3.1: Overview of Sixth-Sense system

(1) *Sixth-Sense Infrastructure*. This is the foundation of our system, which can collect contexts from heterogeneous sources and interpret higher-level contexts from raw sensor data. It consists of several components.

- *Ontology Server*: The core component of our infrastructure is an ontology server, which stores the ontology definition of smart artifact systems. It serves as a shared knowledge base from which all other layers can retrieve information. Our ontology definition, *SS-ONT*, is described in Section 3.2.
- *Wrappers*: Our infrastructure includes many interfaces to acquire low-level context information from various context sources, such as smart artifacts, Web services, such as *Yahoo Weather* (see “<http://developer.yahoo.com/weather/>” for its introduction), and humans. With these wrappers, raw sensor data collected will be wrapped in a unique form, i.e., represented as OWL statements. Each wrapper-derived context reflects a feature or attribute of a related physical entity (e.g., a cup) or an abstract concept (like weather).
- *Querying*: The query engine enables applications to selectively access contexts defined in our ontology server. Different from traditional SQL queries, by making use of Semantic Web techniques, our system supports semantic querying of the defined ontology terms. We further clarify this in Section 3.3.
- *Reasoners*: Ontology-based context reasoning is implemented at two levels. First, since OWL is rooted in description logic (DL), well-known DL reasoners such as Racer [Haarslev *et al.* 01], and also a set of predefined OWL-axiom-based rules can be used for ontology inference in our system. Reasoning tasks such as class subsumption and class consistency can be implemented at this level. Second, to derive information that cannot be directly inferred from ontology axioms (e.g., *what the human is doing*), users can define a set of inference rules and use an inference engine to deduce higher-level contexts. Both reasoning results are asserted into the ontology server. We describe our reasoning mechanisms in detail in Section 3.4.

(2) *Open-Programming Model*. This layer includes two programming modes for end users, they are, *rule-based programming mode* and *customization mode*. The prior mode allows advanced users to create rule-based, high-quality smart artifact applications, and the latter one empowers average home users to reprogram the shared applications in a simple fashion (according to their preferences). We will describe them in Chapter 4.

3.2 An Ontology-based Context Model

(3) *Application Layer*. This layer includes various applications that can be developed using our system, including human-centric services and artifact-based game applications. Some examples are presented in Chapter 4.5.

(4) *Error-Checking Mechanism*. As a programming platform, an error-checking mechanism for users is also important. In *Sixth-Sense*, we also deployed several mechanisms to deal with software and hardware errors in smart artifact applications, as we present in Chapter 5.

3.2 An Ontology-based Context Model

A unified method for explicitly representing context semantics is crucial for sharing common understandings among independently developed smart-artifact systems. Our *Sixth-Sense* infrastructure explores the Semantic Web technology to model contexts. The ontology, called *SS-ONT*, is described using the OWL language.

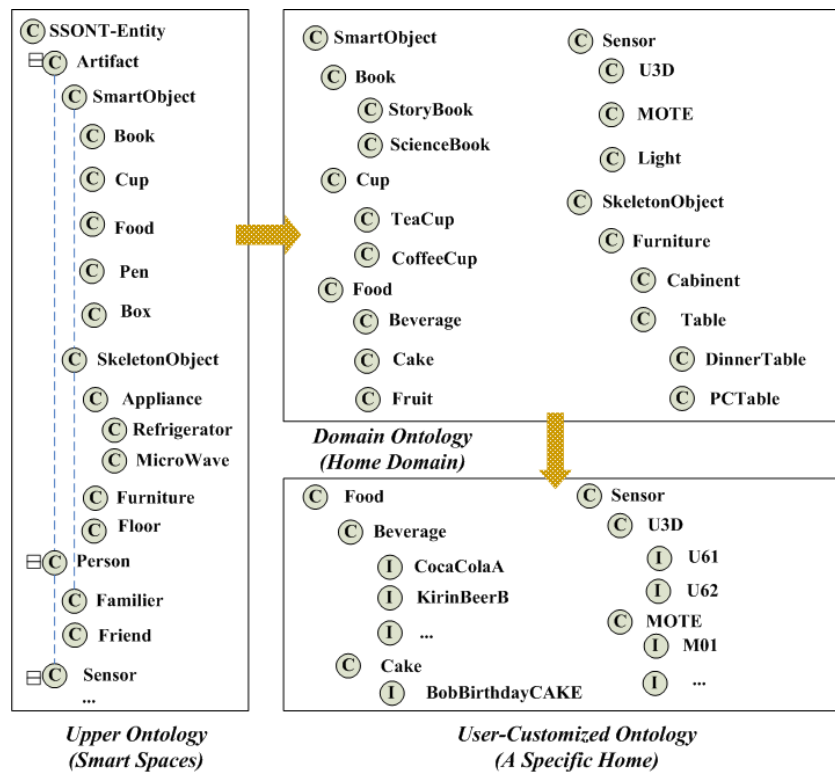


Figure 3.2: Three levels' design of SS-ONT

3.2.1 SS-ONT Design

As shown in Fig. 3.2, we define *SS-ONT* at three levels. Definition of the first two levels, namely *upper ontology* and *domain ontology*, follows the general ontology design

strategy as reported by Wang *et al.* [04]. The upper ontology is a high-level ontology that provides a set of basic concepts common to different smart spaces (e.g., homes and offices). The domain-specific ontology is an extension of the upper ontology created by adding detailed concepts and their features in different application domains (e.g., “*dinner table*” class and “*cabinet*” class in home domain, as illustrated in Fig. 3.2). These two levels make *SS-ONT* more extensible, and enable developers to customize particular domain ontologies by inheriting the classes defined in the upper ontology. Different from previous systems, as a user-oriented system, we also introduce a third level, *instance definition level* or *user-customization level*, which allows home users to insert particular instances that exist in their home (e.g., *a birthday cake* and *a bottle of beer*, as shown in Fig. 3.2) to a shared domain ontology. In this way, each family can share a common knowledge structure and only adding the instances of their home, which helps to realize application sharing and customization.

There have been many attempts in the ontology research field that try to bridge the gap between real world and computational world, such as SOUPA [Chen *et al.* 04b] and ULCO [Wang *et al.* 04]. In light of the advantage of knowledge reuse provided by using an ontology, we referenced several such consensus ontologies and standard specifications about home devices, such as ECHONET (please refer to “<http://www.echonet.gr.jp/>” for its information), borrowing some terms from them but not importing them directly. Different from previous ontology studies, our *SS-ONT* ontology is an attempt to model portion of knowledge in the domain of human-artifact interaction. From the various contexts, we selected 14 top-level classes, and a set of sub-classes, to form the skeleton of *SS-ONT*, i.e., the *SS-ONT* upper ontology (see Fig. 3.2). In next subsection, we will describe an extension of this upper ontology to the smart home domain.

3.2.2 SS-ONT Domain Specific Ontology

Part of an extension of the *SS-ONT* upper ontology for the smart-home domain is shown in Fig. 3.3. This specific ontology defines a number of concrete subclasses, properties, and individuals in addition to the upper classes. The current version of the *SS-ONT* (ver-1.5) domain ontology consists of 108 classes and 114 properties, the whole definition of which is available at: “<http://www.ayu.ics.eio.ac.jp/members/bingo/SS-ONT-v1.5.owl>”). It models a portion of the general relationships and properties associated with artifacts,

3.2 An Ontology-based Context Model

people, sensors, behaviors as well as games in smart environments. In the following subsections we explain how to achieve this using OWL.

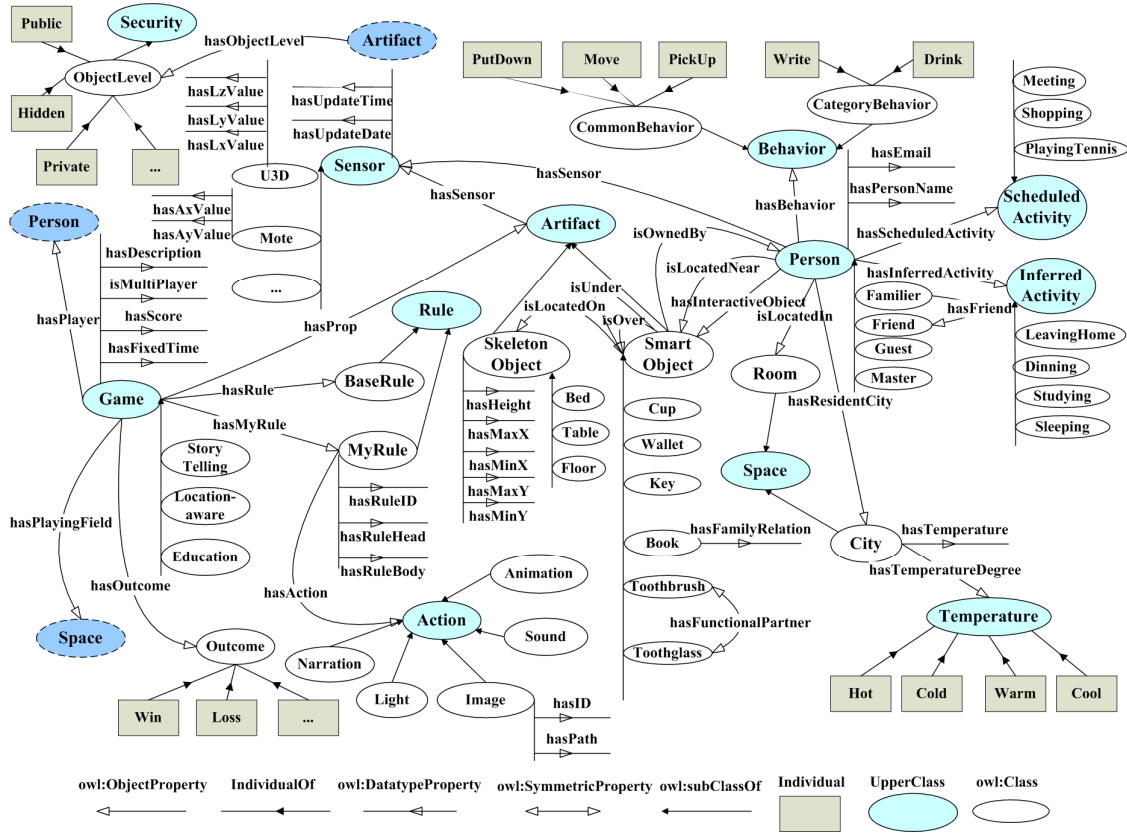


Figure 3.3: SS-ONT domain ontology model (partial)

(1) Person

The “*Person*” class defines typical vocabularies for describing a person. Besides the profile description, there are specific predicates for representing human locations, human behaviors, and activities.

Location: The location of a human is modeled in several ways in response to varied references. For example, the predicate “*isLocatedIn*” denotes the spatial relation between a person and a room, the range of “*isLocatedAt*” is a skeleton object (such as a bed, explained later), and the predicate “*isLocatedNear*” represents a person’s location by nearby *SmartObject* individuals. Different representation formats deliver different contexts, which makes the reasoning task more effective.

Human Behavior: In the course of human-artifact interactions, ongoing human behaviors are reflected by changes in smart-artifact states. As shown in Fig. 3.3, the class

“*Behavior*” has two subclasses, “*CommonBehavior*” and “*CategoryBehavior*”. The former denotes behavior types that exist among all object categories, for instance, *pick up*, *put down*, and *move*, while the latter indicates category-specific behaviors, such as the “*drink*” behavior to the cup category. The relation between persons and their behaviors is expressed by a “*hasBehavior*” property. In human-artifact interactions, each human behavior also implies that a potential object is being interacted with, and we use “*hasInteractiveObject*” to reflect this information.

Human Activity: The OWL class “*Activity*” also has two subclasses: “*InferredActivity*” and “*ScheduledActivity*”. The former denotes activities derived from context reasoning, while the latter refers to activities retrieved from software services, such as *Google Calendar API* (available at: “<http://code.google.com/apis/calendar/>”). Two properties are available to distinguish them: “*hasInferredActivity*” and “*hasScheduledActivity*”.

(2) *Artifacts*

Explicit definitions of physical artifacts are important for smart-artifact applications. Defined artifact properties can be classified into four types: object profiles and status information, location description, relationships with humans and with other objects.

Profile and Status: These describe, respectively, the static (e.g., *hasName*, *hasShape*) and dynamic properties (e.g., *hasTiltAngle*, *isMoving*) of an artifact.

Location: We use a symbolic expression method to represent the location of objects. For example, the location of a key can be represented as “on (key, table)”, which means the key is on a table. Obviously, a symbolic expression provides an efficient way for humans to find things than using an absolute-location expression (i.e. raw coordinate data). To achieve this, we borrowed two subclasses from the *Artifact* class: *SkeletonObject* and *SmartObject* (see Fig. 3.3). Several large and mostly static artifacts (e.g., furniture) are selected to serve as skeleton objects, while other small, easily movable objects (e.g., keys, books) are categorized into the *SmartObject* class. In *SS-ONT*, the property “*isLocatedOn*” is defined to represent the spatial relation between smart object *A* and skeleton object *B* (meaning *A* is located on *B*). There are also two OWL inverse properties terms, *isOver* and *isUnder*, defined to reflect the spatial relations between two *SmartObject* instances (e.g., a pen is placed on a book).

3.2 An Ontology-based Context Model

For the *SkeletonObject* instances, because they may act as reference objects for other smart objects, particular properties are needed to specify their coverage area. For example, the coverage area of a skeleton object with a rectangular surface can be expressed by four OWL data-type properties, *hasMaxX*, *hasMinX*, *hasMaxY*, *hasMinY*, which involve the maximum and minimum horizontal coordinate values. There is also another datatype property “*hasHeight*” to denote the height of this skeleton object, and a *Boolean* property “*hasLoadedObject*” to express whether there is any object located on this skeleton object. An OWL object property “*isLocatedIn*” is defined to specify in which room the skeleton object is placed.

Relationships with humans: This mainly reflects the relations between artifacts and humans. *isOwnedBy* and *isInteractedBy* are two such properties.

Relationships with objects: In addition to physical relations, there are also logical relations among artifacts (e.g., the functional relation between a toothbrush and a glass). Compared to physical relations, logical relations among objects can sometimes provide more precise and direct information (e.g. based on the known functional relationship, we can guess that the object placed in a glass is a toothbrush and not a pen). To the best of our knowledge, the logical relations between physical objects have not been discussed in previous smart-space studies. The current *SS-ONT* defines four types of logical relations.

Definition 3.1 (Combinational Relation): Some objects act as components of, or accessories for, other objects, e.g., the relation between a mouse and a computer (reflected by the predicate “*hasComponent*” in *SS-ONT*).

Definition 3.2 (Partner Relation): Several objects can cooperate as partners to perform human-centric tasks and are always arranged in the same way. Possible instances include tables and chairs, and foreign language books and electronic dictionaries. The partner relation between objects is symmetric, and a symmetric property “*hasWorkPartner*” is used to represent this relation.

Definition 3.3 (Family Relation): Objects that belong to the same category sometimes have this relationship because, in daily life, similar operational modes exist when humans deal with objects in the same category (e.g., people deposit all their books on the bookshelf, or place all their umbrellas in the umbrella stand near the door). We use a *Boolean* property “*hasFamilyRelation*” to denote if the object class has such a relation.

Definition 3.4 (Functional Relation): Some objects are designed for a specific functional purpose, which is to serve the work of another object (e.g. a toothbrush and its supporting glass). We use “*hasFunctionalPartner*” to express this relation.

(3) *Security Filter*

As reported in Meyer *et al.* [03], if people prefer to live in a context-aware home for extra convenience, every measure should be taken to prevent information about their private life being accessed by anybody else. Otherwise, they will not find life in a smart house pleasant. In an artifact-rich environment, residents should have the authority to decide which artifacts can be publicly viewed and which cannot, and also the authority to allow access to different users, such as friends.

To this end, we classify the “*Person*” class into four subclasses: *Master*, *Familier* (denotes a family member), *Friend*, and *Guest*. In addition, artifacts are classified into five types: *Public* (only available to guests, e.g., magazines), *ProtectedFamily* (objects shared by the whole family, such as a shoe cabinet), *ProtectedFriend* (an object that is privately owned but set by its owner to friend level), *Private* (personally owned objects, such as diaries), and *Hidden* (objects whose sensors do not work well, see Chapter 5). A *Familier* user can manage and view the private objects that belong to him (defined by the “*isOwnedBy*” property) and all the other three object types. As a specific *Familier* user, a *Master* user has intimate knowledge about home computers and is responsible for maintaining *ProtectedFamily* objects (e.g., insert an item about a new chair, or modify information about an existing object). If a *Familier* user *A* has a friend *B*, and *A* sets the level of a private object *O* to *ProtectedFriend*, then *O* will be accessible to *B*. In addition, for the sake of security, objects at the *Hidden* level can only be tracked by *Master* users. With this protection policy, if *B* wants to view an object owned by *Familier* user *C*, his access will be denied.

(4) *Sensor*

Our system uses various sensors, including U3D (ultrasonic 3D location) sensors, MICA2 mote sensors (see “<http://www.xbow.com/>”) and Kinotex pressure sensors (see “<http://www.tactex.com/>”). Such sensors are either attached to smart artifacts or placed on skeleton objects. The technical details of these sensors will be described in Section 4.2. To reflect the various sensor types, we borrowed several subclasses from the top

3.2 An Ontology-based Context Model

class “*Sensor*”, like *U3D* and *Mote* (see Fig. 3.3). A number of properties that represent sensor values are also defined. For instance, “*hasLxValue*”, “*hasLyValue*” and “*hasLzValue*” denote the real-time coordinate values derived from U3D, whereas “*hasOldLxValue*”, “*hasOldLyValue*”, and “*hasOldLzValue*” represent the sensor data for the last update time (i.e., a historical data record). “*hasAxValue*” and “*hasAyValue*” denote the two axis acceleration sensor values.

(5) *Category Standard*

It is known that people always use the concept of category to distinguish between objects. Each category should have one or more particular attributes that distinguish it from other object categories. A definition of this artifact category principle is given here.

Definition 3.5 (Category Standard): Each artifact category has its product or design standards, such as shape, size, color or weight, which are all endowed with specific or unified definitions (e.g., the size of a sheet of paper can be A4, B5; while the shape of the bottom of a cup is usually circular). We call this category standard.

Use of category standard helps recognize objects (e.g., identity of a detected hidden object, see Chapter 5) and *SS-ONT* has a top class “*Standard*” to reflect this concept. The class *Standard* has a number of individuals (e.g., *Cup-Standard*, *Key-Standard*) that denote different category standards. A set of properties is defined to describe the standards. OWL datatype properties “*hasMaxLength*”, “*hasMinLength*”, “*hasMaxWidth*”, “*hasMinWidth*”, “*hasMaxHeight*”, and “*hasMinHeight*” are used to express the general dimension range of this object category. Properties such as “*hasWeight*” and “*hasShape*” are used to describe the common attributes of the same kind of object as well.

Table 3.1: Definitions about the components of a game in *SS-ONT*

<i>Elements of a Pervasive Game</i>	<i>Related Class</i>	<i>Relation with Games</i>
Goals/Description	Data type: String	hasDescription
Rules	Rule (BaseRule, MyRule)	hasRule
Playing field	Space	hasPlayingField
Outcome	Data type: Int Outcome (Win, Loss, et al.)	hasScore, hasOutcome
Players/Actors	Person	hasPlayer
Props/Resources	Artifact	hasProp
Feedback/Virtual Elements	Action (Animation, Sound, et al.)	hasAction

(6) Games

Pervasive gaming is an emerging and promising context-aware application area, and it is defined by several new elements. Table 3.1 lists the key elements of a pervasive game (referring to the definitions about games in Hinske *et al.* [07]) and their relevant representations in *SS-ONT*. To distinguish *pattern rules* (i.e., rules created by rule designers) and *user-customized rules* (users can reprogram a pattern rule to produce new rules), we defined two subclasses of class “*Rule*”, namely “*BaseRule*” and “*MyRule*”. Definitions in this way enable users to experience different gaming experiences by configuring the elements of an originally created game (or a pattern game). Outcome is another essential element of a game. In our system, both quantitative (e.g., scoring) and qualitative (e.g., *win* or *loss*) outcomes are available for games. Finally, the OWL object-property “*hasAction*” describes the relation between a rule and the actions (e.g., playing a video clip) should be performed in response to events provided that stated conditions of the rule hold.

3.2.3 User-Oriented Ontology Customization

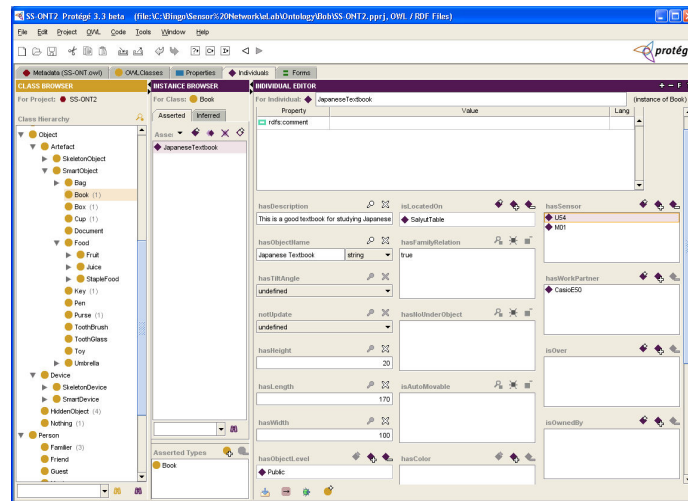


Figure 3.4: A screenshot of the Protégé-OWL editor

As mentioned in Section 3.2.1, the third level’s definition of *SS-ONT* allows users who have intimate knowledge about their homes and daily routines to customize an ontology for their home. As shown in Fig. 3.4, the tool we chose to support this user-oriented task is the Protégé-OWL editor, because it provides well-designed documents, user-friendly interfaces and error-checking mechanisms, which make it a good tool for home computer users. User-oriented ontology customization in our system is mainly about individuals,

3.2 An Ontology-based Context Model

i.e., the ABox definition of *SS-ONT*, which reflects things that are different among different homes. Following is the description of two such tasks.

(1) *Creating individuals*. It is not possible for an ontology developer to predefine the instances (e.g., *a story book about Harry Potter*) of the defined classes within an individual home. Instead, this should be defined by end users. It is not difficult to achieve using Protégé. For example, to create an instance for a certain class, a user can first choose the targeted class (e.g., *Book*) from the ‘*classes tree*’ shown in the left frame of Fig. 3.4, and then press the ‘*create instance*’ button in the ‘*instance browser*’ frame (middle part of Fig. 3.4) to create a new instance (with a desired name) for the selected class. Users can repeat the above steps to create more individuals that exist in their home. Table 3.2 summarizes the types of individuals needed to be created by users.

Table 3.2: Concepts needed to be configured by users

<i>Tasks</i>	<i>Concepts to be Configured</i>
Creating individuals	Person, Artifact (Smart artifact, Skeleton artifact), Device, Sensor, Space (City, Room)
Asserting properties	Human or object profiles (e.g., <i>hasName</i> , <i>hasSize</i> , <i>hasResidentCity</i>)
	Sensor deployment information (e.g., <i>hasSensor</i>)
	Static human-artifact relations (e.g., <i>isOwnedBy</i>)
	Static artifact-artifact relations (e.g., <i>hasFunctionalPartner</i> , <i>hasComponent</i>)
	Static human-human relations (e.g., <i>hasParent</i> , <i>hasFriend</i>)
	Space settings (e.g., <i>hasPart</i> , <i>isPartOf</i>)

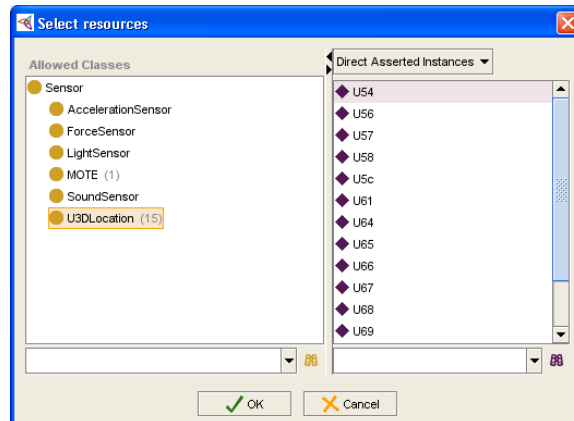


Figure 3.5: A screenshot of the “hasSensor” property-setting dialog

(2) *Setting properties*. Another task for ontology customization is to assign static-property values (or *static contexts*) for new created individuals. As shown in Fig. 3.4, when an individual is created, all its properties will be listed in the ‘*individual editor*’

frame (right part of Fig. 3.4). For an OWL datatype property like *hasEmail*, users can directly fill its value in the relevant form, while for an OWL object property like *hasSensor*, users should press the diamond-shaped buttons on the ‘*resource widget*’ of this property, and specify its value in the pop-upped resource dialog, where all available individuals within the range of this property are listed (see Fig. 3.5). A set of user-controlled properties are also listed in Table 3.2.

3.2.4 SS-ONT Version Updating

Current *SS-ONT* version reflects a range of common context categories and their properties in a smart home. However, as a unified context model, it should meet the dynamic changes of smart home techniques (e.g., a new kind of sensor is deployed) and user needs (e.g., one user requires a new property to represent that the yogurt is rotten). Benefitting from OWL ontology’s hierarchical structure, it is easy to extend *SS-ONT* to meet these new requirements. When a new *SS-ONT* version is published, our system can, at the same time, automatically update user-customized ontologies by inserting new defined terms and updating the changed ones. This is achieved by using the Protégé-OWL API [Knublauch *et al.* 05], a programming-level API for parsing and updating OWL statements. In a word, when new *SS-ONT* versions are published, users can maintain the newest ontology definition with little manual effort.

3.3 Context Querying Engine

The query engine provides an abstract interface that enables applications or agents to extract desired context information from *SS-ONT*. We provide two query mechanisms, *Protégé-OWL API based query* and *SPARQL based query*.

3.3.1 Two Query Methods

For experts and programmers, a simple way to query *SS-ONT* is to use the Protégé-OWL API, which allows them to query the content of user-defined classes, properties or individuals from an OWL file by writing Java codes. However, the efficiency and expressive power of this kind of query is not high. To support advanced queries, we also adopted the SPARQL language [Prud’hommeaux *et al.* 06], the Semantic Web query language developed by W3C, as the context query language. A SPARQL query statement consists of a series of triple patterns (i.e., $\langle \textit{subject}, \textit{predicate}, \textit{object} \rangle$) and modifiers.

3.3 Context Querying Engine

Different from traditional SQL queries, as the query language for the Semantic Web, SPARQL-based query approach allows users to perform “semantic querying” of defined concepts. Here we give an example to manifest the differences of different query methods.

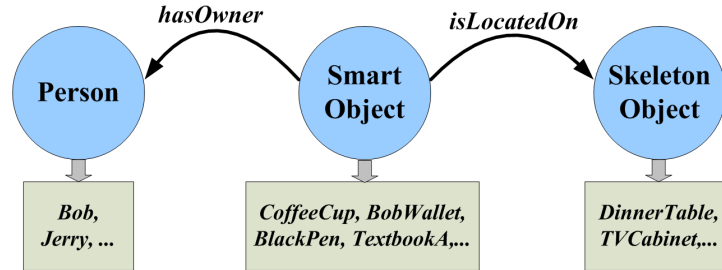


Figure 3.6: A partial graph model of SS-ONT

Table 3.3: A sample query using three different query methods

Query Method	Query Expressions
Protégé-OWL API Query	<pre> JenaOWLModel owlModel = ProtegeOWL.createJenaOWLModelFromInputStream(SS-ONT_Path); OWLNamedClass cls = owlModel.getOWLNamedClass("SmartObject"); Collection instances = cls.getInstance(true); //retrieve all instances of the SmartObject class for (Iterator jt = instances.iterator(); jt.hasNext();) { OWLIndividual object = (OWLIndividual) jt.next(); RDFProperty isLocatedOn = owlModel.getRDFProperty("isLocatedOn"); RDFIndividual ske_Obj = (RDFIndividual) individual.getPropertyValue(isLocatedOn); if(ske_Obj.getBrowerText() == "DinnerTable") System.out.println(object.getBrowerText()); } </pre>
SPARQL Query	<pre> PREFIX ssont: <http://~/SSONT.owl#> SELECT ?object WHERE{?object ssont: isLocatedOn ssont: DinnerTable .} </pre>
SQL Query	<pre> Select Object_Name From SmartObject, LocationRelation Where SmartObject.ObjectID=LocationRelation.EntityID and LocationRelation.Place=DinnerTable </pre>

Figure 3.6 shows a simple model that expresses the relations between several classes (extracted from *SS-ONT*). A simple query over this model, says, “*List all the objects that are placed on the DinnerTable*”, is listed in Table 3.3. This example clearly indicates that writing SPARQL query statements is easier and more intuitive than writing Java codes (i.e., API-based method), especially for non-programmers. But for expert users, the Protégé-OWL API based query method is sometimes more flexible because it can be seamlessly integrated into their Java-based projects. From this example we can also find

that SQL-based query is mainly based on integer/string comparisons, which ignores any relationships between physical entities (i.e., objects, sensors, humans). In contrast, SPARQL-based query is founded on semantic mapping, and its WHERE clause consists of several “triple-formatted” semantic descriptions. For example, the triple pattern in this example (i.e., `<?object ssont: isLocatedOn ssont: DinnerTable>`) denotes “*what are placed on the dinner table*”. It’s obvious that, for smart artifact systems, semantic querying is more intuitive and expressiveness, and it may traverse some hops in SQL querying.

3.3.2 SPARQL-based Query Mechanism

Here we explain in detail how SPARQL-based query engine works. As mentioned previously, the WHERE clause in a SPARQL query consists of several triple patterns. These triples together comprise what is known as a *graph pattern*, which can be used to identify the shape of the graph that we want to match against. That’s to say, a query in our system attempts to match the triples of the graph pattern to the *SS-ONT* context model (as shown in Fig. 3.3). Each matching binding of the graph pattern’s variables to the model’s nodes becomes a query solution, and the values of the variables named in the SELECT clause become part of the query results. The example mentioned previously used one triple pattern to match the two nodes (i.e., *SmartObject* and *SkeletonObject*) of the sample model given in Fig. 3.6. For a relatively complex query, says, “*Whose objects are placed on the DinnerTable*”, which relates to all three nodes in the sample model, we need to use a two-triple-formed graph pattern for its matching, as shown in Eq. (3.1). Therefore, for users who want to perform SPARQL-based query in our system, they have to firstly examine the graphical context model to identify which nodes (or what shape) are to be covered in the target query, and then they can combine the triples related to these nodes and create the graph pattern for this query. In our system, users are recommended to use the Protégé-OWL query panel to perform SPARQL-based queries over the *SS-ONT* context model.

```

PREFIX ssont :<http://~/SSONT.owl#>
SELECT ?person
WHERE{?object ssont:isLocatedOn ssont:DinnerTable .
      ?object ssont:hasOwner ?person .}

```

(3.1)

3.4 Context Reasoning Engine

As discussed previously, *Sixth-Sense* infrastructure supports two reasoning levels: ontology reasoning based on OWL axioms and logic inference via user-defined rules. Below, we will explain these two reasoning levels in detail.

3.4.1 Ontology Reasoning

Since OWL is rooted in description logic (DL), in the design stage of *SS-ONT*, we used a well-known DL reasoner *Racer* [Haarslev *et al.* 01] to execute class subsumption (i.e., to compute the inferred class hierarchy) and consistency checking. *Racer* is capable of detecting inconsistencies as demonstrated in the following example: if a property like “*isLocatedIn*” is defined as an OWL functional property, and in the definition of the “*Person*” class, a minimum cardinality restriction “2” is added to this property, there will be a conflict between these two definitions according to the principles of OWL language. If we run *Racer*, it detects this inconsistency. In the course of *SS-ONT*'s extension, users are also suggested to use *Racer* to keep their ontology's consistency.

In current implementation of *Sixth-Sense*, we also provide a mechanism to perform rule-based ontology reasoning among the defined ontology concepts (i.e., classes, properties, and instances). It is implemented by combining the Protégé-OWL API and Jess inference engine [Friedman-Hill 07]. Our ontology reasoning mechanism supports both RDF Scheme (RDF-S) and OWL Lite (please refer to Smith *et al.* [03] for more details). The RDF-S reasoner supports all the RDF-S specifications (e.g., sub-class, sub-property), while the OWL reasoner supports simple constraints on classes and properties (e.g., symmetry, cardinality and transitivity). All these rules are needed to be predefined, and some examples are listed in Table 3.4.

Ontology reasoning is useful for context-aware systems, by which some implicit relationships among defined ontology concepts can be derived. For example, the *Transitivity* rule can be used in spatial reasoning. In *SS-ONT*, we use an OWL transitive property “*isPartOf*” to define relations between different “*Place*” instances. If two “*isPartOf*” relations between “*Room412*” and building “*25#*”, and between “*25#*” and “*Yagami_Campus*” are known, an implicit context that “*Room412*” is part of the “*Yagami_Campus*” can then be deduced. This is because the spatial relation “*isPartOf*” is transitive. Another example comes from two other properties called *isOver* and *isUnder*,

which are defined as OWL inverse properties. If we know that “*BookA*” is placed on “*BookB*”, an implied context that “*BookB*” is under “*BookA*” can be derived according to the *InverseOf* rule listed in Table 3.4.

Table 3.4: Instances of OWL ontology reasoning rules

<i>Name</i>	<i>Rule</i>
SubClassOf	$(?a \text{ rdfs:subClassOf } ?b) \wedge (?b \text{ rdfs:subClassOf } ?c) \Rightarrow (?a \text{ rdfs:subClassOf } ?c)$
SubPropertyOf	$(?a \text{ rdfs:subPropertyOf } ?b) \wedge (?b \text{ rdfs:subPropertyOf } ?c) \Rightarrow (?a \text{ rdfs:subPropertyOf } ?c)$
Transitivity	$(?p \text{ rdf:type owl:TransitivityProperty}) \wedge (?a ?p ?b) \wedge (?b ?p ?c) \Rightarrow (?a ?p ?c)$
InverseOf	$(?p1 \text{ owl:inverseOf } ?p2) \wedge (?a ?p1 ?b) \Rightarrow (?b ?p2 ?a)$
DisjointWith	$(?c1 \text{ owl:disjointWith } ?c2) \wedge (?a \text{ rdf:type } ?c1) \wedge (?b \text{ rdf:type } ?c2) \Rightarrow (?a \text{ owl:differentFrom } ?b)$
Functional Property	$(?p \text{ rdf:type owl:FunctionalProperty}) \wedge (?a ?p ?m) \wedge (?a ?p ?n) \Rightarrow (?m \text{ owl:sameAs } ?n)$
Symmetry	$(?p \text{ rdf:type owl:SymmetricProperty}) \wedge (?a ?p ?b) \Rightarrow (?b ?p ?a)$

3.4.2 User-Defined Rule-based Reasoning

In the logical reasoning level, the creation of user-defined inference rules is allowed, which makes the reasoning more flexible. A wide range of high-level contextual information, such as “*what is the user going to do*”, “*how should the agent react to the current situation*”, can be deduced at this level.

Currently, *SS-ONT* supports rules in the form of the *Semantic Web Rule Language* (SWRL) [Horrocks *et al.* 04] and Jess [Friedman-Hill 07]. SWRL is based on a combination of the OWL DL and OWL Lite sublanguages, and it enables users to write Horn-like rules to reason about OWL individuals and to infer new knowledge about these individuals. We chose to use SWRL for several reasons: (1) SWRL is a standardized language to realize rule interoperability on the Web, that’s to say, to share rules among different rule-based systems, and it is designed to be the rule language of the Semantic Web. (2) The Protégé team from Stanford University has developed a full-featured editor for SWRL [O’Connor *et al.* 05]. The SWRL editor tightly integrates with Protégé-OWL (the editor we used to define *SS-ONT*), and also supports inference with SWRL rules using the Jess rule engine. The highly-interactive interfaces, well-designed documentation and error-checking support make it an ideal tool for experienced users to create, edit and test their rules. (3) One of the most powerful features of SWRL is its

3.4 Context Reasoning Engine

ability to support a range of built-ins. A built-in is a predicate that takes one or more arguments and evaluates them as true if the arguments satisfy the predicate. For example, an “*equal*” built-in is defined to accept two arguments and return true if the arguments are the same. Using the built-ins, users can create more flexible rules and more interesting applications. As an emerging rule language, there does not exist a standard rule engine that can directly execute SWRL rules. Therefore, we integrated a famous rule engine, Jess, into our infrastructure. We chose Jess as the rule engine, because it works seamlessly with Java (note that all our work builds upon the Java platform), has an extensive user base, is well documented, and is very easy to use and configure, as reported in O'Connor *et al.* [05]. Jess has its own rule language, so current implementation of *Sixth-Sense* also supports rules written in the Jess rule language.

1. Defining Human-Artifact Interaction Rules

Note that inference rules are used to deduce high-level, implicit contextual knowledge from low-level raw data, and that the premises of some inference rules might involve facts that are newly derived from other inference rules. That is, inference rules are designed and implemented at different levels, as illustrated in Fig. 3.7.

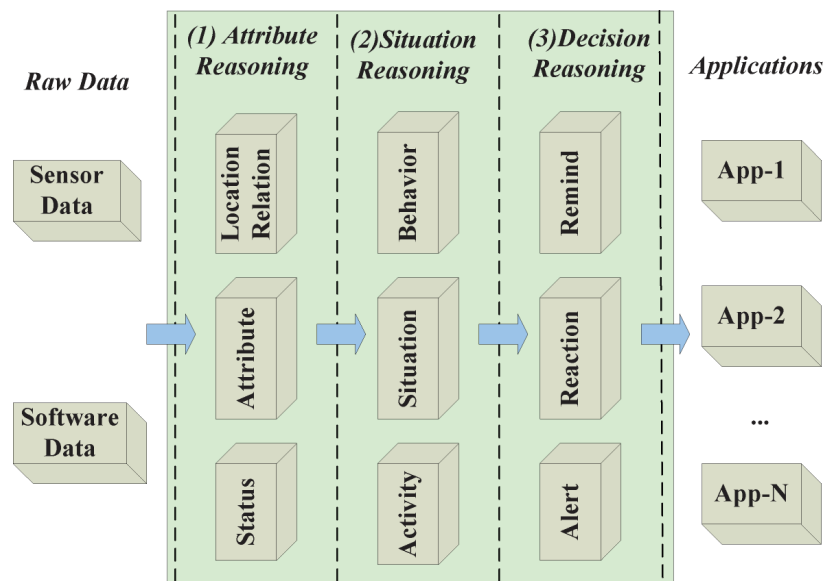


Figure 3.7: Different reasoning levels

- *Basic Attribute Reasoning*: The lowest reasoning level, the task of which is to read in the updated sensory data (or software data) to derive some state, attribute, or location information about the relevant entity (the entity might be an artifact, a person, or an

abstract concept like the weather). Inference rules at this level usually follow the form given in Eq. (3.2):

$$\begin{aligned}
 & \langle \text{entity}(?x) \rangle \wedge [\text{existed_objectproperty}(?x, ?y)] \wedge \\
 & \langle \text{datatypeproperty}(?x, ?num1) \{>, <, =, >=, <= \} num2 \rangle \\
 & \rightarrow \langle \text{inferred_objectproperty}(?x, ?value1) \rangle
 \end{aligned} \tag{3.2}$$

Where: $\langle x_1, x_2, \dots, x_n \rangle$ denotes that the formula can involve “1” or *several* such terms defined in this set; $[x_1, x_2, \dots, x_n]$ denotes that “0” or *several* such terms defined in this set can be used in this formula; and $\{x_1, x_2, \dots, x_n\}$ denotes that the formula can only select one term from the set. Two instances of rules (R3.1 and R3.2) at this level are given in Table 3.5. Note that some SWRL built-ins are used in these rules, preceded by the namespace qualifier: *swrlb*).

Table 3.5: Instances of inference rules for human-artifact interaction

Rule Type Name(Level)	Inference rules represented by SWRL
Weather R3.1 (L-1)	$City(?x) \wedge hasTemperature(?x, ?y) \wedge swrlb : greaterThanOrEqual(?y, 25) \wedge$ $\rightarrow hasTemperatureDegree(?x, Hot)$
Location Near R3.2 (L-1)	$Person(?m) \wedge hasSensor(?m, ?s1) \wedge U3D(?s1) \wedge SmartObject(?n) \wedge$ $hasSensor(?n, ?s2) \wedge U3D(?s2) \wedge hasLxValue(?s1, ?x1) \wedge$ $hasLxValue(?s2, ?x2) \wedge swrlb : subtract(?x3, ?x1, ?x2) \wedge$ $swrlb : abs(?x4, ?x3) \wedge swrlb : lessThanOrEqual(?x4, 500) \wedge$ $hasLyValue(?s1, ?y1) \wedge hasLyValue(?s2, ?y2) \wedge swrlb : subtract(?y3, ?y1, ?y2) \wedge$ $swrlb : abs(?y4, ?y3) \wedge swrlb : lessThanOrEqual(?y4, 500) \rightarrow isLocatedNear(?m, ?n)$
Behavior Pick-Up R3.3 (L-2)	$SmartObject(?A) \wedge hasSensor(?A, ?s) \wedge U3D(?s) \wedge Person(?B) \wedge$ $hasLzValue(?s, ?z1) \wedge hasOldLzValue(?s, ?z2) \wedge$ $swrlb : greaterThan(?z1, ?z2) \wedge swrlb : subtract(?z3, ?z1, ?z2) \wedge$ $swrlb : greaterThan(?z3, 120) \wedge isLocatedNear(?B, ?A)$ $\rightarrow hasBehavior(?B, PickUp) \wedge hasInteractiveObject(?B, ?A)$

- *Situation Reasoning*: By reflecting changes in information relating to the state or location of smart artifacts, some higher level contexts, such as a human’s behavior or current activity in a room, can be deduced. A general form for this kind of rule is given in Eq. (3.3), and an instance (R3.3) is listed in Table 3.5.

3.4 Context Reasoning Engine

$$\begin{aligned}
& human(?x) \wedge \langle artifact(?y) \rangle \wedge \langle artifact_property(?y,?value1) \rangle \wedge \\
& \langle human_artifact_location_property(?x,?y) \rangle \wedge \\
& [existed_human_behavior_property(?x,?value2)] \\
& \rightarrow \{human_behavior_property(?x,?b) \wedge behavior_metadata(?b,?value3), \\
& human_activity_property(?x,?a) \wedge activity_metadata(?a,?value4)\} \\
& \wedge [other_human_artifact_property(?x,?y)]
\end{aligned} \tag{3.3}$$

- *Decision Making*: How the application or agent reacts to the current situation or environment change is dealt with at this reasoning level. For instance, when a robot detects the resident is going to leave home, and also finds that his room-key is still on the table, the robot can generate an alert to the human, reminding him to take the key. A general form for this kind of rule is given in Eq. (3.4).

$$\begin{aligned}
& human(?x) \wedge \langle artifact(?y) \rangle \wedge \langle artifact_property(?y,?value1) \rangle \wedge \\
& \langle human_behavior_property(?x,?b), human_activity_property(?x,?a) \rangle \wedge \\
& [behavior_property(?b,?value2), activity_property(?a,?value3)] \\
& \rightarrow reaction_property(?x,?value4)
\end{aligned} \tag{3.4}$$

2. Integration with an Inference Engine

We used the SWRL factory mechanism to integrate the Jess rule engine with user-defined SWRL rules. Jess is a forward-chaining rule engine, with which users can run SWRL rules interactively to create new OWL concepts and then insert them into the OWL knowledge base. In our system, the interaction between SWRL rules and the Jess rule engine is implemented through the SWRL-Jess Bridge API [O'Connor *et al.* 05].

To allow Jess to integrate with SWRL rules, our *Sixth-Sense* infrastructure performs the following four steps: (1) represents relevant OWL knowledge defined in *SS-ONT* as Jess facts; (2) represents SWRL rules as Jess rules; (3) performs inference using those rules; and (4) reflects the resulting Jess facts back to the *SS-ONT* knowledge base. The interaction between Jess and SWRL is data-driven in our system. That is, when the system receives updated data from a sensor node, the *four-step* interaction process will be triggered to see if some new facts can be deduced. An example of the overall inference process is shown in Fig. 3.8. In this example, if a smart book, called *BookA*, was previously on the table and, in the next instant, picked up by a nearby person (called *Bob*), the sensed location value (read from U67) for *BookA* will change. Rule *R3.3* (see Table 3.5) will be triggered then and a new fact – *Bob* picks up *BookA* – will be generated. A

detailed description about the cooperation between the reasoner and other components (e.g., context acquisition module, ontology server) is clarified in the next section.

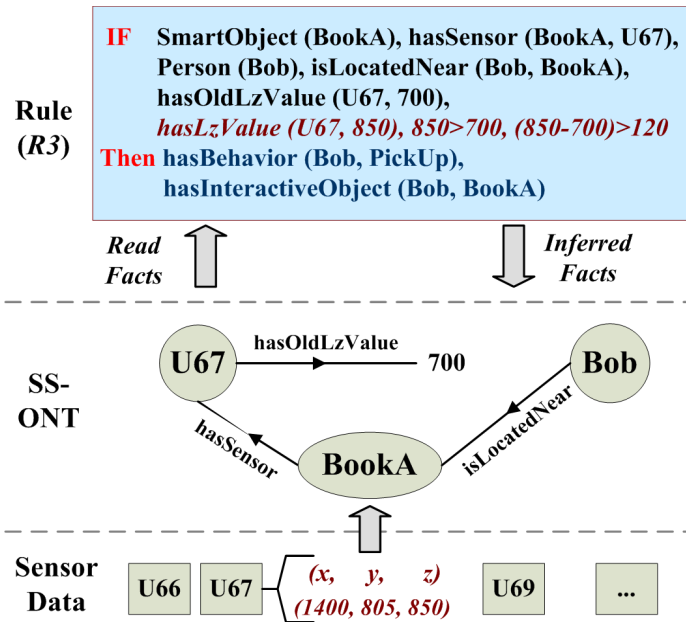


Figure 3.8: Rule-based reasoning process

3.5 Run-Time Process of Sixth-Sense

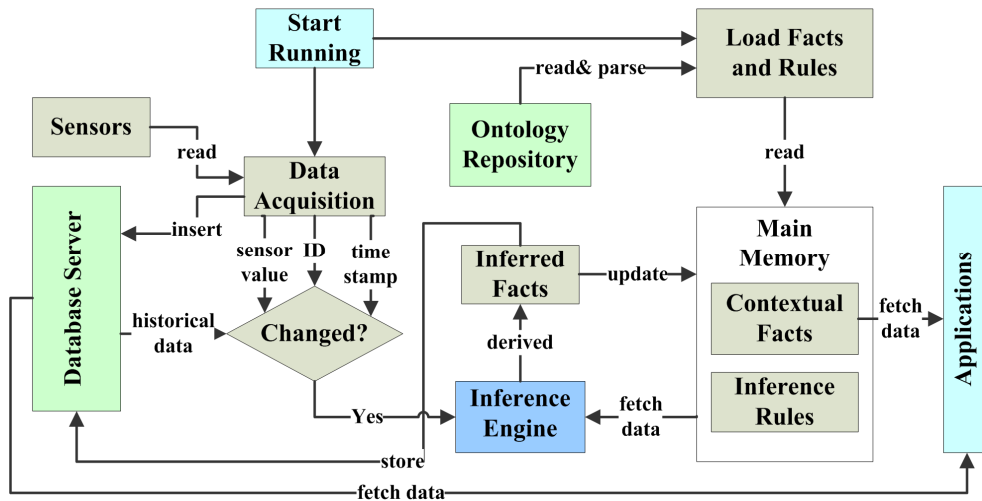


Figure 3.9: Sixth-Sense working process

The process used by our system is partially illustrated in Fig. 3.9. Because reasoning and querying operations are performed in the memory, when *Sixth-Sense* starts running, it first loads the ontology model and SWRL rules (from OWL files) into the main memory. Meanwhile, all the sensors start working and their data is acquired by the data acquisition

3.5 Run-Time Process of Sixth-Sense

component of our system. The acquired data is stored in a MySQL database, thus providing a record of data history and enabling us to compare newly acquired sensor data with the last update time data recorded in the database to see whether its value has changed. As there is some sensor noise, we use predefined thresholds to measure these changes. For example, for U3D (ultrasonic 3D location) sensors, the threshold value is set to 10; that is, if the differences in the 3D coordinate values between two U3D sensor updates are all below 10, i.e., $(|x_1 - x_2| < 10) \& \& (|y_1 - y_2| < 10) \& \& (|z_1 - z_2| < 10)$, we consider that its value has not changed. The inference engine is activated once our system detects a change in a sensor value. This decision policy helps improve the performance of our system because we do not have to trigger the rule-reevaluation process if all the contexts in the physical world remain stable.

The newly inferred facts are asserted into the context model residing in the memory. All running applications can retrieve the up-to-date contextual data from the memory and react to the changing world. It should be noted that the ontology repository (stored in OWL files) is not updated with the newly inferred facts in the meantime because these logically inferred facts (e.g., object locations, human activities) are not persistent contexts and thus change dynamically from one second to another. Incorporating backup operations like this would overburden our system and do nothing to help the future use of our ontology. However, these inferred facts are stored in our database server with their timestamps. They thus act as historical contexts and allow other applications to query past information.

Chapter 4. Open-Programming Model

For the reasons such as privacy, controllability and personality, end users should be empowered to exert control over the enabled smart home services, or to create some new ones if they find that current applications can not meet their needs. To address diverse user requirements and interest, we developed a new programming model based on our *Sixth-Sense* infrastructure. The ontology-based programming approach, called *Open-Programming*, is a well-working blend of the two design goals, “*simplicity*” and “*functionality*”. In this chapter, we will present the design and implementation of our *Open-Programming* model. Some enabled applications will also be described.

4.1 Open-Programming Platform: an Overview

This section will give an overview of our *Open-Programming* model. Because it is a programming toolkit for smart artifact applications, we will first describe the combination of a typical smart artifact application in our system.

4.1.1 Combination of a Smart-Artifact Application

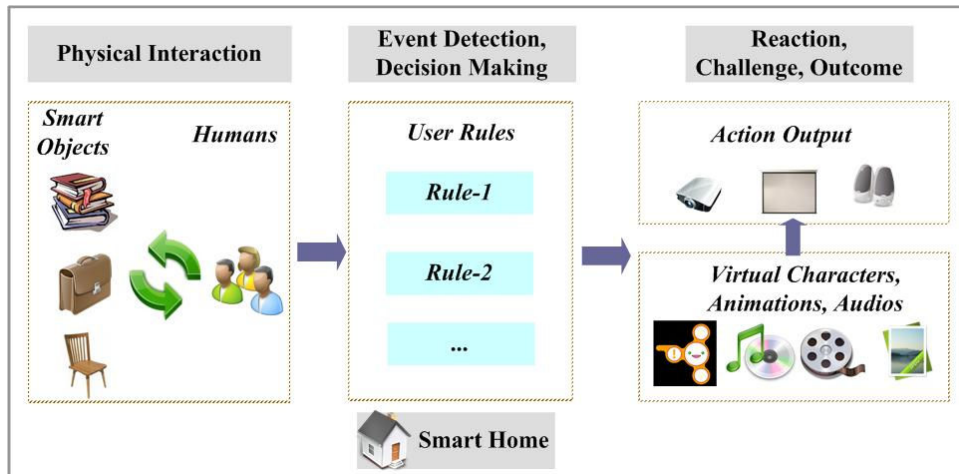


Figure 4.1: Combination of a smart-artifact application

Following the consensus from previous studies that most context-aware applications are built upon a rule-based paradigm [Dey *et al.* 06], our system allows users to create rule-based applications. In view of advanced users’ characters that they are willing to experience a relatively high technology if it can produce considerably high rewards to them, the rules in our system are expressed by a normalized, expressive rule language, Jess (one rule language that is supported by *Sixth-Sense*, see Section 3.4). That’s to say,

4.1 Open-Programming Platform: an Overview

comparing with visual programming toolkits, the *Open-Programming* model empowers advanced users to create more complex and flexible services.

As illustrated in Fig. 4.1, a smart artifact application consists of four main parts: a *working environment* (e.g., a smart home), *physical interaction* (between humans and everyday objects), *event detection/decision making* (based on a set of inference rules), and *reaction/outcome* (i.e., displaying animations to real-world surfaces or playing a music clip). A platform for programming smart artifacts should provide an integrated solution that can combine different elements (e.g., smart objects, action devices, rules, virtual characters) and make them work as a whole. In Section 4.2, we will describe the hardware configuration as well as the controlling mechanism of our prototypical smart home environment. The connection between “physical interaction” and “event detection” parts are handled by the *Sixth-Sense* infrastructure presented in last chapter. The *Open-Programming* model, which copes with creation of rules and the connection between rules and actions, will be described in Section 4.3 and 4.4.

4.1.2 Open-Programming Model and Users

The *Open-Programming* model is an ontology-based, user-oriented model, which imports the ideas like “*free*” and “*sharing*” from the open-source culture. It is aimed at providing a collaborative programming environment where users with different technical abilities can perform different-level’s programming work. For example, for novices, it seems to be difficult for them to create rules in a rule language. However, because in the *Open-Programming* model, all services created by advanced users can be shared on the Web, novice users can browse them and perform “second-development” upon these shared services. In total, there are four kinds of entities shared in this model, they are, *ontology*, *rules*, *context-aware services* and *actions*. Users with different abilities can perform different operations over these entities, as illustrated in Fig. 4.2.

(1) **Ontology:** To support sharing of context-aware rules and services, one key aspect is that every user should follow a unified method to represent contexts and facts. In Chapter 3, we have described *SS-ONT*, an ontology-based context model for smart environments, which defines a series of general relationships associated with artifacts, humans and sensors in a typical smart home. Before using our system, a user should first customize an ontology for their home, as we described in Section 3.2.3.

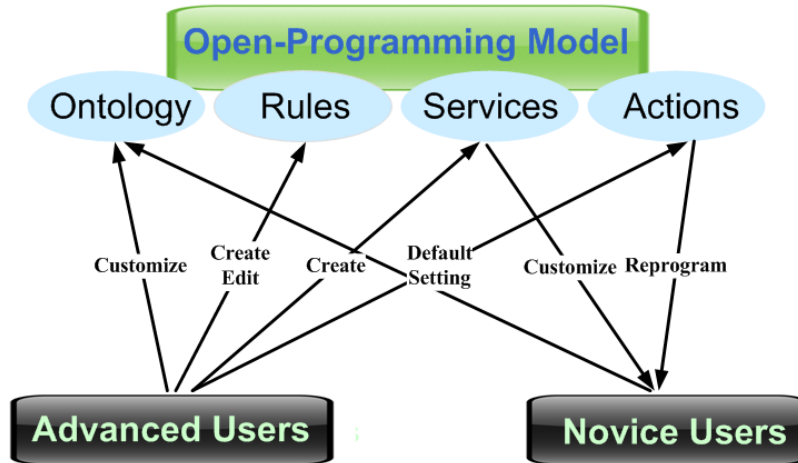


Figure 4.2: User operations in the Open-Programming model

- (2) **Rules:** All services in our system are rooted in rules. An environment that asks users to create rules using a specific rule-language implies a longer period of time to grasp it. However, as all user-created rules are shared through the *Open-Programming* model, the learning process could be greatly speeded up. First, our system provides a “*basic rule*” repository, which consists of rules at the *fundamental reasoning level*, or *perception level*, such as object or human localization (e.g., *Bob is near a cup*), basic human behavior perception (e.g., *Bob picks up something*), as well as object status measurement (e.g., *a cup is tilted*), etc. These rules are common and frequently used in all kinds of applications. For end users, the basic rules we offered can be directly integrated into their services or, alternatively, be good examples for them to learn how to create rules using the terms defined in *SS-ONT*. Second, various rules created by users are also shared through the Web server, which are good references for others as well. All these benefits ensure that, when creating a new service, a developer can utilize mostly existing resources and only create a few new rules to meet his particular needs.
- (3) **Services:** For users who have no ability or will to develop services, they can search the shared services developed by advanced users from the Web server and customize the ones that interest them. Considering that a service developed by one user may not meet others’ needs, we allow the service developer to design a simple “front-end” for other users to configure it. In this way, both advanced users and novice users are allowed to program smart artifacts in our system.

4.2 Establishing a Smart Artifact Environment

(4) Actions: Our system allows a user to define related actions (e.g., animations, audios, etc) for each inference rule defined in his application. Interactive devices, such as projectors and speakers (see Section 4.2), are utilized to make the digital content information available to home users. For advanced users, they can give a default action setting for the application they created; while for other users, they can either follow the default settings or change it to their preferences.

With the *Open-Programming* model we proposed, the two contradictory-seemingly design principles, “*functionality*” and “*simplicity*” (because a toolkit that allows users to create high quality, powerful applications usually indicates that it is not easy to master), is well balanced in our system.

4.2 Establishing a Smart Artifact Environment

Our system is an exploration of smart artifact applications within smart homes. Ideally, smart homes should be filled with small, cheap and interconnected sensors and interactive devices, as envisioned in Weiser [99]. To investigate the prospects of artifact-based applications in future homes, we established an experimental smart home environment in our lab using a combination of sensing and action techniques.

4.2.1 Indoor Sensing Techniques

We created numerous sensor-augmented smart artifacts in our prototypical experiment environment. Three types of sensors were employed for this, namely ultrasonic 3D tags (or U3D for short, see top left of Fig. 4.3), and MICA2 Mote sensors (top right of Fig. 4.3), and Kinotex pressure sensors (see Fig. 4.5).

U3D is one kind of location sensor, which consists of an ultrasonic transmitter, a wireless communication unit and a microcomputer. The ultrasonic pulses emitted from a U3D’s ultrasonic transmitter will be received by the ultrasonic receivers deployed in our experimental environment (see Fig. 4.4). Based on the time-of-flight (the travel time of the signal from transmission to reception) measuring results from more than three receivers, the position of this U3D tag can be calculated. Our experimental room is about 4.0×4.0 m in size, where we embedded 16 ultrasonic receivers on the ceiling. U3D tags were attached to various indoor objects, including a cup and a pen shown in bottom of Fig. 4.3. Both absolute location (i.e., raw coordinate values) and symbolic location (see

Section 3.2.2) of indoor objects are supported by our U3D positioning system. For more details about this positioning technique, please refer to Nishida *et al.* [03].

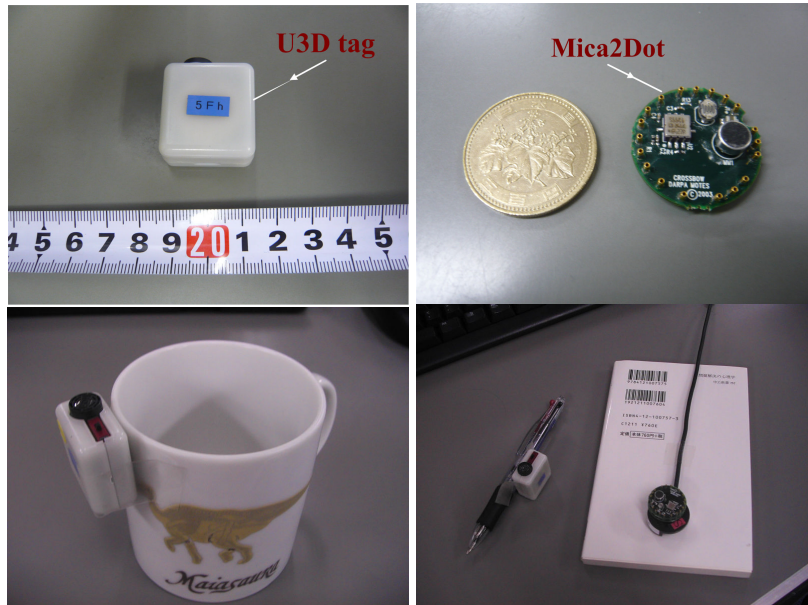


Figure 4.3: U3D tags (top left), Mote sensors (top right) and prototypical smart artifacts (bottom)

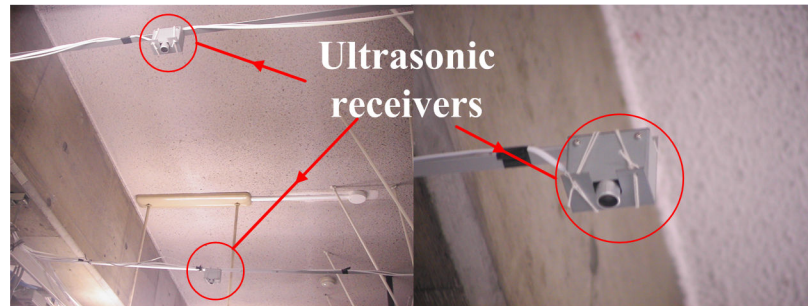


Figure 4.4: Ultrasonic receivers on the ceiling

MICA2 Mote sensor, a product of Crossbow, is a combination of a two-axis acceleration sensor, a light sensor, and a sound sensor. The Mote provides compact design, low power consumption, computational ability, and programmability, which makes it ideal for making smart artifacts. By interpreting the data acquired from the equipped Mote sensor, the status of a smart artifact (e.g., is tilt or not) as well as the change of this object’s surrounding environment (e.g., a human passes by) can be derived. An example of a Mote-equipped book is given in bottom right of Fig. 4.3.

From Fig. 4.5 we can see that Kinotex is formed by sixty ($10rows \times 6lines$) sensor cells (size: $2.4cm \times 3.2cm$). When an object *A* is placed on it, the corresponding cells under *A* will generate sensory value for the pressure, by which we can get some valuable

4.2 Establishing a Smart Artifact Environment

attributes of A , such as shape and size. Right of Fig. 4.5 shows the output of Kinotex (of the three loaded objects).

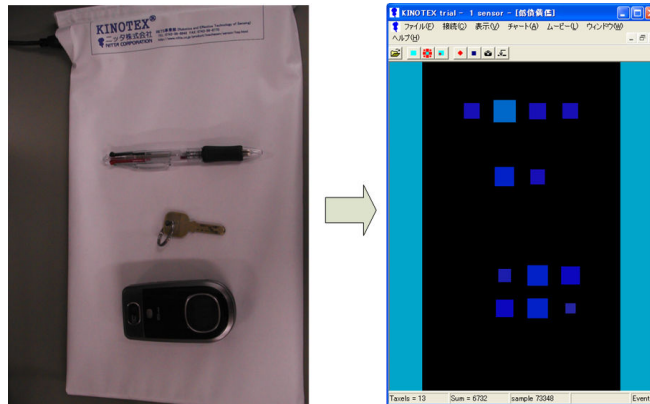


Figure 4.5: A working example of the Kinotex pressure sensor

4.2.2 Augmented Reality Techniques

The most technically advanced ubicomp applications use augmented reality techniques (e.g., handheld devices [Benford *et al.* 06] and head-mounted displays [Cheek *et al.* 02, 04]) as a basis. Augmented reality is also implemented as part of our experimental environment, which deals with the combination of real-world and computer-generated data. The action module of our system, called Prot, consists of a projector, a mirror, an ultrasonic directional speaker [Nakadai *et al.* 05], and a 3-DOFs rotating base, as shown in left of Fig. 4.6.

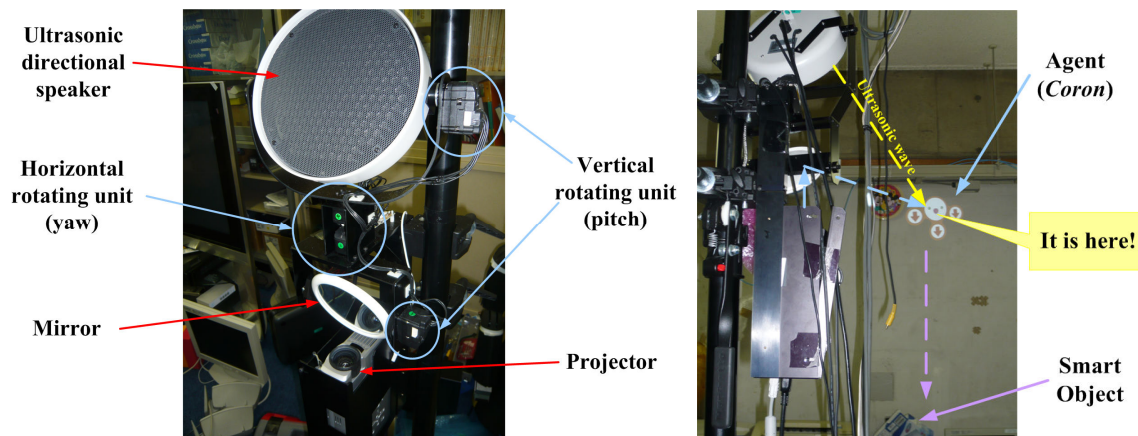


Figure 4.6: The Prot action device (left) and a working example (right)

- *Rotating base.* Over the three rotating units of the rotating base, two are used to define the vertical direction of the mirror and the ultrasonic directional speaker, and the third one determines the horizontal direction of these two action parts. With the

support of the rotating base, the direction of the ultrasonic directional speaker and the mirror can be changed to arbitrary directions.

- *Projector & mirror.* The mirror reflects the projected video signal from the projector to real-world surfaces, such as the floor, walls, and objects. In this way, virtual characters that typically reside in computers are available to users as they move into the physical world. An example is shown in right of Fig. 4.6, where an “on-wall” agent is prompting the user to pick up the object under it. Note that this technology enables users to view “real-world” agents without using any head-mounted devices, which prohibits the issues relevant to them, such as occlusion [Magerkurth *et al.* 05] and uncomfortable feelings from users [Cheok *et al.* 02].
- *Ultrasonic directional speaker.* As reported in Ishii *et al.* [07], ultrasonic wave transmitted by an ultrasonic directional speaker can go straight through the air, and be converted to audible sound when reflected by real-world objects. Because humans can not hear ultrasonic sound, they feel as if the sound is made by these physical objects. This property helps us to insert new interaction and entertainment elements when designing smart artifact applications, such as anthropomorphosis and interactivity. As shown in right of Fig. 4.6, the ultrasonic wave from the directional speaker is reflected by the wall and changes to audible sound of an “on-wall” agent.

4.2.3 Action Control Mechanism

The control structure of Prot is illustrated in Fig. 4.7. The control commands are generated after interpretation of the sensory data and then sent out to three different control modules.

(1) *Motor control.* This module handles exact motion of the three rotating units. There are two main control modes. The first one can adjust the rotating base to an optimal status where other action devices relied on it can perform their tasks well. (e.g., a predefined rotation angle which ensures the agent can be projected to the best place for viewing). In the second mode, the rotating base can change its pose according to the position of a target object. For example, in a treasure hunting game, the agent projected from Prot can appear at a nearby place of the treasure and navigate the player to it.

(2) *Projection control.* As summarized in Table 4.1, both animations and images are supported in our projection system. All these files are shared in a Web server and can be

4.2 Establishing a Smart Artifact Environment

viewed through a 3D Web browser — uBrowser (see “http://www.ubrowser.com” for its information). We chose uBrowser not only because it is open source, but also due to the fresh and cool 3D experience it can bring to users. We have published a series of animation files on the Web server, including 34 behaviors of the *Coron* agent (see right of Fig. 4.6) in the form of Flash files. User generated contents, such as recorded video clips, photos, self-made Flash files, are another important source of video materials.

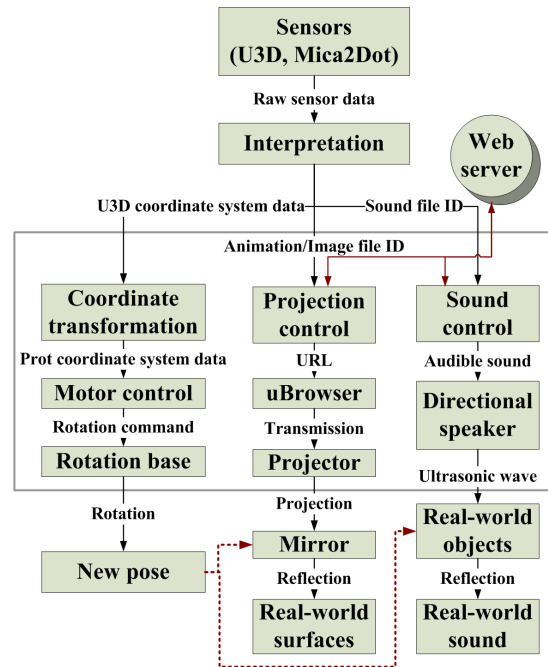


Figure 4.7: The control structure of Prot

Table 4.1: Different action elements

Type	Sources	
Video	Animation	Flash movie clips
		Captured videos
		Movie clips
		Coron agents (34 behaviors, provided by iFun)
	Image	Cartoon pictures
		Photos
Audio		Self recorded files (using a microphone)
		Using a TTS (Text-to-Speech) service
		Music files

(3) *Sound control*. As mentioned previously, utilizing the ultrasonic directional speaker, a sound clip played by a computer can finally be changed to the voices of a real-world object. User generated contents are also acceptable. As listed in Table 4.1, there are several ways to create sound files. First, users can record their own voices using a

microphone. Second, they can simply utilize a text-to-speech (TTS) system (e.g., the online AT&T Natural Voices system, which is available at: “http://www.naturalvoices.att.com”). The video or audio files supplied by users can be used to create or customize applications, as we depict later.

4.3 Designing Customizable Services

From this section, we will present the implementation of our *Open-Programming* model. As mentioned in Section 3.1, our programming model involves two programming modes, *rule-based programming mode* and *customization mode*. In this section, we firstly describe the prior one, i.e., the programming mode that allows advanced users to create high-quality, reconfigurable smart artifact applications.

4.3.1 Creating a New Service

The main interface (i.e., the service-center page) of our *Open-Programming* model is shown in Fig. 4.8 (c), which consists of four parts: the search area, where users can browse the shared services in the service repository and subscribe the ones that interest them; two management areas for published services and subscribed services, where users can manage the services they created or configure the services they subscribed; the service creation area, which is the starting point for new service creation.

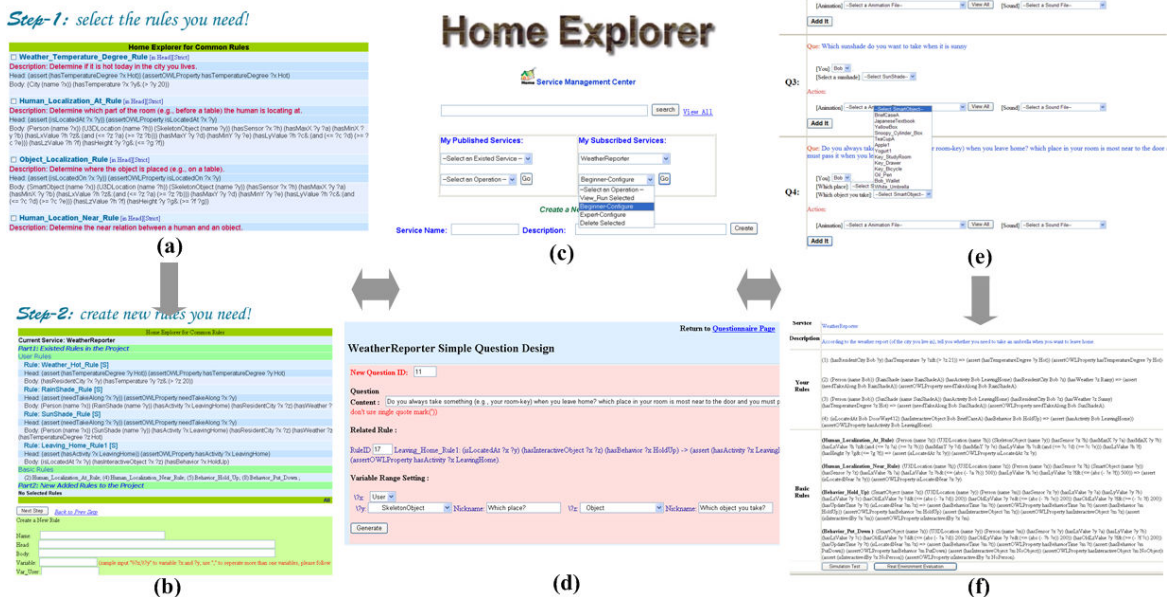


Figure 4.8: The Open-Programming model interfaces

4.3 Designing Customizable Services

To create a new service and publish it as a public service for sharing, there are four steps to perform: (1) create a new service; (2) select useful rules from the shared rule repository and, if necessary, create some new rules; (3) give a default action setting; (4) design a front-end that can help other users to configure this service.

Following we will use a simple example to manifest the above development process. The service, called *WeatherReporter*, can give suggestions to a resident according to the local weather information when it detects that he is going to leave home, for example, reminding him to take an umbrella when it rains. We can easily create a new service at the bottom area of the service-center page (see Fig. 4.8 (c)). When a new service is being created, a brief description about this service is also necessary, since it includes a summary introduction of this service that can help others to quickly understand the uses of the service.

Typically, to build a service, there are three rule sources available: *basic rules* from the “*basic rule*” repository, *shared user rules*, and *new-created rules*. As mentioned previously, basic rules are frequently used in various applications and are good references to service developers. There are preliminary 16 basic rules provided by our system, and we hope to keep on enlarging the scale of the rule repository. Rules created by other users may either be directly used or be good references (i.e., modifying some parts to meet a new need) to a service developer. However, since shared rules (*basic rules* and *shared user rules*), in most cases, can not meet all the requirements of a new service, the service developer often has to create some new rules to fulfill his particular needs.

WeatherReporter is a situation-triggered service. The specific situation it concerns is that the resident is going to leave home. Assume that in a developer’s perspective, most users usually take something (e.g., *a briefcase* or *a wallet*) when leaving home. He can then define a rule to predict the leaving-home activity using the following premise: *a person is situated at the doorway with the object he usually takes when leaving home*. Several contexts need to be derived before this situation-detection rule works, they are, *the person’s location in the room, his behavior relevant to the target object*. As these perceptions belong to the fundamental-reasoning level, we can find them from the “*basic rule*” repository (the rule-selection page is illustrated in Fig. 4.8 (a)) and directly import

them into the new service. Table 4.2 lists all the basic rules chosen for the *WeatherReporter* service, some of them have been presented in Chapter 3.

Table 4.2: Basic rules used in the WeatherReporter service

<i>Rule ID</i>	<i>Description</i>
Human_Localization_At Rule	It can report a human's position relative to the indoor skeleton objects (e.g., on the bed, at doorway).
Human_Location_Near Rule (R3.2)	It determines whether a human is near to a particular smart object.
Behavior_PickUp Rule (R3.3)	It detects if a human picks up a particular smart object.
Behavior_PutDown Rule	It detects if a human puts down a particular smart object.

With the four selected rules, we can infer whether a human is situated at the doorway with the object he usually takes. In detail, if a human picks up an object and does not put it down, we infer that he is carrying it (note that we assume that the human should be near to the object if he is performing an object-related behavior). Based on the four basic rules, we can create a new rule, *Leaving_Home_Rule*, to detect if a human is going to leave home. This rule, *Rule 4.1*, is formalized as a Jess rule in Eq. (4.1).

$$\begin{aligned} & (isLocatedAt \ ?x \ ?y) (hasInteractiveObject \ ?x \ ?z) (hasBehavior \ ?x \ PickUp) \\ \Rightarrow & (assert (hasActivity \ ?x \ LeavingHome)) \end{aligned} \quad (4.1)$$

Having prepared the rules for situation-detection, following we define the rules for decision-making. The decision process can be described as this: the service agent first sends a request to the *Yahoo Weather Service* for today's weather information; if the result predicts a rainy day, the agent will suggest that the person takes an umbrella. Another rule, *Umbrella_Rule* (*Rule 4.2*) is created for this, as formulated by Eq. (4.2).

$$\begin{aligned} & (Person (name \ ?x)) (Umbrella (name \ ?y)) (hasActivity \ ?x \ LeavingHome) \\ & (hasResidentCity \ ?x \ ?z) (hasWeather \ ?z \ Rainy) \Rightarrow (assert (needTakeAlong \ ?x \ ?y)) \end{aligned} \quad (4.2)$$

The above six rules, including four basic rules and two new created rules, jointly constitute the *WeatherReporter* service. The interface where users can create new rules and manage existing rules is shown in Fig. 4.8 (b).

4.3.2 Default Action Settings

A significant function of our *Open-Programming* model is that it allows users to define related actions (e.g., animations, audios, etc) for the inference rules they created. As mentioned previously, we exploit the Prot device to assert actions on the environment. That's to say, we allow users to build a connection between a rule-detected event and an action that Prot should perform. For example, a user can specify a video clip, an image,

4.3 Designing Customizable Services

an audio clip or a combination of these different forms to be played or displayed when a specific rule is triggered. For an application developer, he can assign a default value for the action to be taken when a rule is created. Other users may either follow the default setting or change its value in terms of their preferences. This task is performed in the action-setting page (see Fig. 4.9), where developers can browse all available action-resources, such as audio files shown in Fig. 4.9 (a), and animation and image files shown in Fig. 4.9 (b) (c). For *WeatherReporter* service, an “umbrella” image and an audio clip, “*You need to take an umbrella*”, are defined as the default actions for the *Umbrella_Rule*.

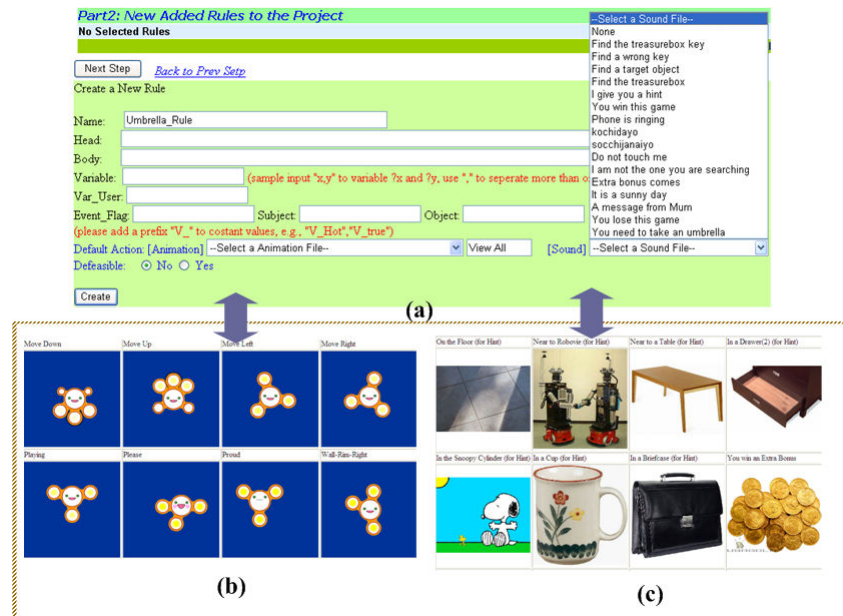


Figure 4.9: Default action setting interfaces

4.3.3 Designing a Configuration Front-End

One of the important aims of our system is to empower average users to exert control over smart home services in terms of their needs. That’s to say, we should support the development of user-configurable services. To achieve this, we have integrated a “*front-end-design*” widget in our system, which allows service developers to design a user-oriented front-end for others to perform “*second-development*” over the services they created. This is implemented by the following two steps.

(1) **Creating dynamic rules.** Since services are rooted in rules, controlling of a service should be mapped to the configuration of its rules. That’s to say, we should make the rules configurable. In our *Open-Programming* model, we call the rules created by service developers (e.g., *R4.1* and *R4.2*) *pattern rules*. A pattern rule involves several elements

whose value can be assigned by other users according to their preferences and domestic resources (e.g., assign a specific value of ‘*BlackBriefcase*’ to variable ‘?z’ in *R4.1*), and therefore addresses the diversity of user considerations. In this way, when a rule is created, its developer should meanwhile specify the user-configurable elements in it. A typical Jess rule consists of four types of elements, namely predicates (e.g., ‘*isLocatedAt*’ in *R4.1*), functions (e.g., ‘subtract’ in *R3.2*), constants (e.g., ‘25’ in *R3.1*) and variables. Since it may easily influence or damage the logic design when allowing users to change the values of predicates and functions in a rule, in most cases, only the latter two types of elements are allowed to be manipulated by users.

In our system, elements selected for end-user configuration are called “*rule-interfaces*”. For basic rules, their rule-interfaces are predefined by our system; but for user-created rules, they are specified by rule developers. To create more flexible rules, a service developer can change some static premises (e.g., preferences or habits of him) into “programmable” variables. For example, for the *Leaving_Home_Rule*, though the developer always takes his briefcase when leaving home, he didn’t define it as a fixed value in this rule; oppositely, he used a variable “?z” instead of a fixed setting, which enables other users to “configure” it according to their own habits.

Table 4.3: Rule-interface settings for the WeatherReporter service

<i>Rule name</i>	<i>User Variable</i>	<i>Variable1 (Range)</i>	<i>Variable1 Nickname</i>	<i>Variable2 (Range)</i>	<i>Variable2 Nickname</i>
Leaving_Home_Rule	?x	?y (Skeleton Object)	Which place	?z (Smart Object)	Which object you take
Umbrella_Rule	?x	?y (Umbrella)	Select an umbrella	–	–

When creating a new rule, the rule developer should decide how many rule-interfaces are to be defined, and specify them clearly in the “*Create New Rules*” area shown at the bottom of Fig. 4.9 (a). There are two items related to this setting: “*Variable*” and “*Var_User*”. The latter one is used for defining which variable represents a human in the rule, and all other programmable variables and constants can be specified in the “*Variable*” item. We separate the user-variable from other variables because the identity of a user can be previously learned by our system (because the user has to firstly login our platform). All rule interfaces designed for the *WeatherReporter* service are listed in Table 4.3.

4.3 Designing Customizable Services

Table 4.4: Front-end design and background information

	<i>Rule</i>	<i>Question</i>
Technical Terms	A→B	?x, 25
Simple Words	A configuration introduction	Nickname (Range setting)

Table 4.5: Front-end settings for the WeatherReporter service

<i>Rule name</i>	<i>Question</i>
Leaving_Home_Rule	Do you always take something (e.g., your room-key) when leaving home? Which place in your room is most near to the door and you must pass it when leaving home?
Umbrella_Rule	Which umbrella do you want to take when it rains?

(2) **Designing a configuration front-end.** Even though clearly indicating the rule-interfaces, it is still difficult for average users to configure a rule written in a rule language, partially due to the unfamiliar syntax or lacking of interest to read rules in this form. Therefore, there still needs a simple way for users to perform these settings.

Preference setting is a common user-oriented service supplied by most Web-based systems, such as language setting in *Google* and city selection in *Yahoo Weather*. Most of the setting-items in these systems consist of a simple question or introduction and a set of candidate answers for users to choose from, and average users have been familiar with such a simple way to customize their services. To enable novice users to configure rules in a simple fashion, service developers are also asked to design a configuration front-end for users. In this front-end, all technical terms in an application will be changed to simple words that can be easily understood by average users, as listed in Table 4.4. In this way, each configurable rule can be represented as a gap-filling question, which consists of a configuration-task introduction and sets of optional values for its rule-interfaces. All available values for a rule-interface are determined by its range setting from the rule-developer. Technically speaking, an interface-range is an OWL class (e.g., *Umbrella*) defined in *SS-ONT*, which restricts the acceptable values of this rule-interface. To help users easily understand and perform the value-setting task, a nickname, or a label for the rule-interface is also supposed to be provided. We list the front-end settings for the *WeatherReporter* service in Table 4.3 and Table 4.5.

The main interface for user-oriented front-end design (including configuration introduction, ranges and nicknames for rule-interfaces) is illustrated in Fig. 4.8 (d). We support two configuration-item types: *simple question* and *multiple choice question*. One

simple question corresponds to one rule in a service, while a multiple choice question consists of several simple questions that have a similar reasoning purpose. For example, there might be several rules that can detect a person is going to leave home (in different ways, such as the person puts on his coat, takes up his briefcase, or opens the door), and a multiple choice question allows a person to customize one or several such rules to deal with the same reasoning task according to his habits. The front-end design task is completed when each rule in the service has a related configuration-item created. The enabled configuration front-end, shown in Fig. 4.8 (e), can greatly ease novice users' effort to customize services, as we mentioned later.

4.4 Customizing Services

Having described the process for advanced users to create configurable services, we will present how our *Open-Programming* model allows novice users to reprogram the shared services in a simple fashion.

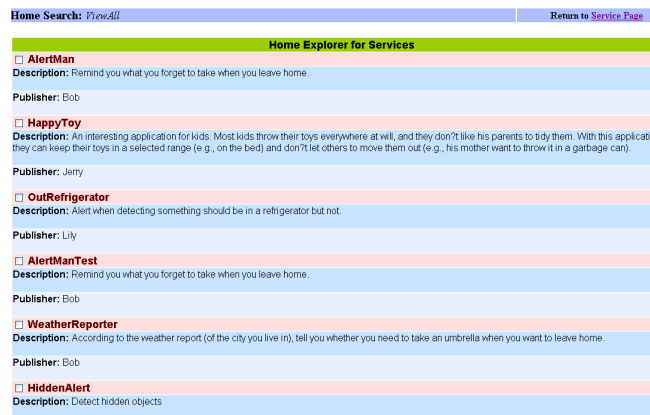


Figure 4.10: A screenshot of the service-browsing page

In the service-center page (Fig. 4.8 (c)), users can input keywords to search interested services or simply click the “View All” link to browse all shared services. Figure 4.10 shows the service-browsing page, which includes all services published by different users. Here we choose *WeatherReporter*, the service we created in Section 4.3, to manifest how a service can be customized. Figure 4.8 (e) illustrates the configuration front-end of the *WeatherReporter* service. It consists of several gap-filling questions, which have the similar form as the preference-setting page of *Google*. Each gap-filling question consists of an introduction and several choice-boxes, which match well the front-end settings mentioned in Section 4.3.3. It should be noted that because the OWL instances listed in

4.5 Enabled Applications

an answer-choice box are read from different user-customized ontologies (according to the identity of current user), there will be different candidate answers for different users. In other words, only the resources owned by the user will be listed here.

Our user-oriented front-end is natural, simple and easy to operate by average users. For example, for Q4 (*Leaving_Home_Rule*), he can easily choose one from all his smart objects that he usually takes when leaving home (e.g., his briefcase). After performing his choice, a user can press the “Add” button to submit his selection. This, at the background level, generates a new user-customized rule. In some cases, there may be more than one answer to a configuration item. For example, for Q4, a user may take both his briefcase and wallet when leaving home. A customizer can re-execute the adding-operations to submit all his answers. Hence, there will be several customized rules added for this question. There are also two configuration items about actions (see Fig. 4.8 (e)), by which a customizer can easily change the default action settings to his preferences. Figure 4.8 (f) shows the browsing page for a user-customized service, which consists of a set of user customized rules. Note that all rule-interfaces in these rules have been changed to specific OWL instances specified by its customizer.

Our front-end based user programming mechanism empowers end users to customize services according to their home resources and everyday habits. Furthermore, since our user-oriented front-end is easy to understand and operate, the increased control doesn't bring a heavy burden to end users. In fact, as its simplicity, an average user can generally carry out the reprogramming work in several minutes.

4.5 Enabled Applications

To demonstrate the feasibility of our programming platform, we prototyped a smart-home application platform called *Home-Explorer*, which involves various applications related to human-artifact interaction, such as *context-aware services* and *pervasive games*. Both *customizable services* (developed by end users) and *professional services* (developed by experts) are enabled using our programming platform. In this section, we will describe two such services: an artifact-based game and the real-world search service.

4.5.1 Treasure Game

Benefited from the *ubiquitous*, *intelligent*, and *tangible* natures, we believe that smart artifacts can become the most significant resources for designing pervasive games in future homes. However, there lacks a study on how to design entertainment applications by making use of indoor smart artifacts. In contrast, our *Open-Programming* platform enables users to design and play artifact-based games in future homes. In our system, smart artifacts are mainly used as interactive game props, for example, a pen can be used to act as a magic wand. Utilizing smart artifacts and other smart devices (e.g., speakers, projectors), users can create a wide variety of games in our system, such as *smart toys*, *storytelling games* and *location-aware games*. Following I will give an example to show how this is achieved using our programming platform.

This is a game about treasure hunting (called Treasure). In this game, a smart home is imagined to be an old castle and players hunt for the hidden treasure in it. Both a treasure box and the key for it have to be found to win this game. A few other objects can also be selected to give hints about the positions of target objects or be utilized to provide other funny or risky gaming experiences (e.g., when a drawer is opened, a monster who resides in it appears on the wall and shouts, “*Don’t disturb me, I am now sleeping*”).

Treasure is a multiplayer game, where one player hides the pre-specified objects and another one or two players act as treasure hunters. We explored a technical shortcoming of U3D tagging system to detect whether or not an object is found. As presented in Harter *et al.* [05], the ultrasonic signal emitted from ultrasonic tags can not be received by ultrasonic receivers subject to the “limited coverage” problem (e.g., when a U3D-tagged object is placed in a desk drawer). Therefore, in Treasure, all hidden objects can not be detected at the beginning of a game session until they are found by a player. Two basic rules, *R4.3* and *R4.4* (see Table 4.6), are used to measure whether a smart object is detected or not. The derived conclusions can be used by *R4.5* to infer which object is found by a player. The outcome of this game, i.e., the rule to determine if the player wins, is expressed by *R4.6*. The latter two rules (*R4.5* and *R4.6*) are particularly designed for this game, and they need to be defined by end-user developers.

4.5 Enabled Applications

Table 4.6: The Treasure game rules

Rule Type	Rule Name (Rule ID)	Rule Content
Basic Rules	Sensor_Not_Updated_Rule (R4.3)	<i>(SmartObject (name ?x)) (U3D (name ?y)) (hasSensor ?x ?y) (hasUpdateTime ?y ?t&:(intervalGreaterThan ?t Int :2)) ⇒ (assert (notUpdate ?x "true"))</i>
	Sensor_Updated_Rule (R4.4)	<i>(SmartObject (name ?x)) (U3D(name ?y)) (hasSensor ?x ?y) (hasUpdateTime ?y ?t&:(intervalLessThan ?t Int :1)) ⇒ (assert (notUpdate ?x "false"))</i>
New Rules	Object_Find_Rule (R4.5)	<i>(SmartObject (name ?x)) (notUpdate ?x "false") ⇒ (assert (find ?x "true"))</i>
	Game_Win_Rule (R4.6)	<i>(Key (name ?x)) (Box (name ?y)) (find ?x "true") (find ?y "true") (Game (name Treasure)) ⇒ (assert (isGameWin Treasure "true"))</i>








To allow novice users to reprogram the games they created, game developers should define a configuration “front-end” for others. The “front-end” settings for the Treasure game are illustrated in the upper part of Fig. 4.11 and Table 4.7. For each rule in the game, the developer can give a default action setting for it. For example, according to the settings in Fig. 4.11, once an object is found, a “smiling” agent will be displayed on the wall and tells the player “*You have found a target object*”.

Table 4.7: Front-end settings for the Treasure game

Rule name	Introduction
Object_Find_Rule	Which object(s) do you want to hide? (1) You must specify a box and a key to act as treasure-box and treasure-box-key. (2) You can additionally select some other objects for giving hints (e.g., speaking “ <i>the treasure-box is under a table</i> ”), or transmitting other funny or risky information (e.g., speaking “ <i>don’t touch me, I am sleeping now</i> ”) to game players.
Game_Win_Rule	The player wins if he finds the following objects that you chose to act as treasure box and treasure box key in Q1.

Novice users can reprogram the shared “Treasure” game through its configuration front-end. For example, the first configuration-item for Treasure (see Table 4.7) asks users to specify several objects to play the roles (e.g., a treasure box) of this game. A customizer can easily choose the ones he prefers (e.g., a used gift box) from the optional-answer list for this item (where all available smart artifacts in his house are listed). In the example of a customized Treasure game shown in the lower part of Fig. 4.11, there are four objects selected, where a room-key and a used keyboard-box were used to play the

roles of the treasure-box-key and the treasure-box, and two other objects, a bicycle-key and a wallet, were respectively used to confuse players (because it was not the real key for the “treasure box”) or to give hints to players, as mentioned later.

		Rules	Rule-Interfaces	Animation Setting	Sound Setting
Game Developer	Basic Rules	Sensor_Not_Updated	No	No	No
		Sensor_Updated Rule	No	No	No
	New Rules	Object_Find Rule	?x: Name of the object		“You have found a target object.”
		Game_Win Rule	?x: Name of the Target Key ?y: Name of the Treasure-Box		“Oh, my god. You win this game. Congratulations!”
Game Customizer	Customized Rules	Object_Find	A room-key (act as target-key)		“You have found the treasure-box-key.”
		Object_Find	A used keyboard-box (act as treasure box)		“You have found the treasure-box”
		Object_Find	A bicycle-key (to confuse the player)		“Do not touch me! I am sleeping now”
		Object_Find	A black-wallet (to transmit hints)		“Thanks for rescuing me, I will give you a hint about the key”
		Game_Win	The room-key The keyboard-box		“Oh, my god. You win this game. Congratulations!”

Partial

New

Default

Figure 4.11: Programming the Treasure game in the Open-Programming model

As illustrated in Fig. 4.11, a user can either follow the default actions defined by a game developer or change the settings (*partially* or *totally*) according to his imagination. For example, if a customizer imagines that a wallet can cue the player of treasure-box-key’s location when this wallet is found, he can set the video to be played to an image of a cardboard box (assume the treasure-box-key is placed in it) and the sound to be played to “*Thanks for saving me from the dark drawer. I will give you a hint about the key*”, as depicted in Fig. 4.11.

From the description of Treasure game, several features of our programming platform can also be derived, they are:

- Different families have different resources (e.g., everyday objects, user generated contents like photos), and they can give different settings to a shared game.
- Different user-settings lead to different gaming experiences, which indicates that our system also enables novices to author games in terms of their imagination.

4.5 Enabled Applications

- The same game can be reprogrammed many times to generate different gaming experiences.

4.5.2 Real-World Search

Our programming platform also facilitates experts to create professional smart artifact services, or public services that can work for different families (i.e., they don't have to be customized). One of them is the real-world service, which can help a user quickly locate his belongings in his house. We use the following scenario to describe the functionality of this service: *Assume it is 7:50 in the morning, and Bob is going to work and must catch the bus at 8:00. He always wears his watch to work, but has forgotten where he put it last night. As a result, he takes too much time to find it and misses the bus.*

Situations like this are common and cause us considerable inconvenience in our daily lives. However, a real-world search system will make it possible to solve this problem. Using the U3D sensor data from smart artifacts and the following rule *R4.7* (formulated in Eq. (4.3)), we developed an indoor-object search system.

$$\begin{aligned} & SmartObject(?A) \wedge SkeletonObject(?B) \wedge hasSensor(?A, ?h) \wedge U3D(?h) \wedge hasLxValue \\ & (?h, ?x) \wedge hasMaxX(?B, ?x1) \wedge hasMinX(?B, ?x2) \wedge swrlb : lessThanOrEqual(?x, ?x1) \wedge \\ & swrlb : greaterThanOrEqual(?x, ?x2) \wedge hasLyValue(?h, ?y) \wedge \\ & hasMaxY(?B, ?y1) \wedge hasMinY(?B, ?y2) \wedge swrlb : lessThanOrEqual(?y, ?y1) \wedge \\ & swrlb : greaterThanOrEqual(?y, ?y2) \wedge hasLzValue(?h, ?z) \wedge \\ & hasHeight(?B, ?z1) \wedge swrlb : greaterThanOrEqual(?z, ?z1) \rightarrow isLocatedOn(?A, ?B) \end{aligned} \quad (4.3)$$

This rule can be explained as this: For smart object *A* and skeleton object *B*, if the downward projection of *A* is within the range of *B*, we conclude that *A* is placed on *B*. The main interfaces of this service are given in Fig. 4.12. To ensure security, a user must first login to our system via an authorized account, including user type (e.g., *Master* or *Friend*), user name, and password (see Fig. 4.12 (a)). In terms of the security policy defined in our ontology, different user types can only search objects in the relevant level (e.g., a *Friend* user cannot search a *FamilyProtected* level object). This policy is reflected in the search results. The main search interface (see Fig. 4.12 (b)) is presented after the user logs in. The user can then input a certain keyword to search for objects. We provide three search modes: (1) *Search by object names*: an individual object (e.g., Bob's wallet) can be searched for using this mode; (2) *Search by category*: in some cases, people may want to find a series of objects that belong to the same category (e.g. Bob may want to

choose an interesting book to read from among his books). From the ontology's point of view, this mode can be interpreted as returning all the OWL individuals that belong to a targeted OWL class; and (3) *Search by particular locations*: this can list all the smart objects placed on a specific skeleton object (e.g. what objects are placed on the dinner table). Figure 4.12 (c) illustrates the search results for books obtained via the category search mode, where the relative locations of these searched books are listed.



Figure 4.12: Real-world search interfaces

In summary, an indoor-object search system can save us much time and effort in organizing and managing our physical belongings. The context used in this application, i.e., one object is placed on another object, fits in the category of artifact-artifact relationships. The relationships between artifacts are significant for most artifact-managing or home-monitoring systems. Our system can be used to implement other similar applications in these two fields. For example, Smart Toolbox [Lampe *et al.* 03], which sends alerts if an operator forgets to put the tools he used back in a toolbox; an application that can send a message to a mother that her children have not put their toys in order after playing with them; and a monitoring application that can generate a warning if it detects that a cup of tea is placed near a notebook PC or on a book. The logical relations between artifacts (defined in Section 3.2.2) are also crucial to smart-artifact systems, as we report in the next chapter.

Chapter 5. Error Checking Mechanism

As a programming platform, an error-checking mechanism is very important for users. In this chapter, we will survey the possible causes of errors of a smart artifact program and present some approaches to deal with them.

5.1 Causes of Errors in a Smart Artifact Program

According to our work on smart-artifact applications and games, we find that there are mainly five causes of weaknesses to a smart artifact application, which can be grouped into either *software errors* or *hardware errors*:

- *Software errors.* (1) user-supplied information is inaccurate subjected to human errors, for example, a wrong input in the rule, incomplete or incorrect logic design of a service, reprogramming errors and deployment errors (e.g., in a Treasure game play, an object is not placed to the pre-specified place by the game customizer). (2) contexts acquired from distinct sources may be inconsistent with each other (e.g., “*Bob is picking up a cup*” vs. “*Bob is sleeping*”, these two contexts are inferred from different rules).
- *Hardware errors.* (1) the raw data collected from sensors is inaccurate as a result of sensor noise, which may lead to incorrect inference results (e.g., a cup on the table is wrongly inferred to be on the floor); (2) sensor data is unavailable because of limited coverage (e.g., the “sensor blockage” problem mentioned in Section 4.5.1) and sensor failure (e.g., the sensor is broken or has a dead battery); (3) problems relevant to action devices, such as communication failure or a broken working part (e.g., a broken rotating unit of the Prot device).

Naturally, it can be very harmful to a smart artifact application if incongruous behaviors are performed or it fails to work sometimes. Although the sensor-enhanced smart artifact strategy has been the subject of many studies, little attention has been paid to the robustness of smart objects. To address this, we present some mechanisms to debug the programmed smart object applications. As mentioned previously, there have been many causes of errors, and our system mainly concerns two of them, they are, *programming errors* and *the hidden object problem*. The latter problem is probably caused by *sensor faults* and *sensor blockage*. As shown in Fig. 5.1, the smart book *Book-*

C is not detected by its ultrasonic sensor signal because it is blocked by *Book-A*. Objects in situations like this are called “*hidden objects*” and locating them is sometimes very important. For instance, for the real-world search service, the target object might be a hidden object and cannot be reflected in the search result. In this way, if the programmer tries to identify some errors in his programming codes, it would be a waste of time. Therefore, detecting hidden objects like *Book-C* is important to our system.

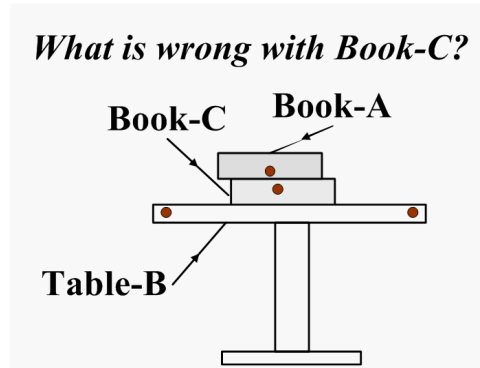


Figure 5.1: A hidden object scenario

5.2 Software Error Checking

Similar to the development of other computer applications, users, especially service developers, should test the service they created and remove any errors in it before it is published. As shown in top of Fig. 5.2, we provide two ways for service evaluation: First, since a real context sensing environment may not be available, we support pure simulation interface, which allows a user to pre-specify the sensing inputs in the ontology he customized through the Protégé-OWL editor (bottom left of Fig. 5.2); the second way is a real context sensing mode, which acquires inputs from the real world.

The simulation mode is mainly designed for service developers, through which a developer can quickly specify some simulation values and test whether the service he created works. For example, to test a customized *WeatherReporter* service, the following two facts should be true: *a user takes up his briefcase*, and *he is located at the doorway place*. The service developer can simulate these two facts by predefining the related sensor inputs (e.g., the coordinate values of the U3D sensor which is equipped to the briefcase, see bottom left of Fig. 5.2) or by simply defining the relevant dynamic properties (e.g., the “*isLocatedAt*” property of a user). After all the relevant context values are specified, the developer can press the “*Simulation Test*” button (see top of Fig.

5.3 Hardware Error Checking

5.2) to evaluate this service. One example of the evaluation result is shown in Fig. 5.2 (bottom right), which consists of the newly inferred facts and a reasoning result message. The developer can check the derived information then to see whether there is something wrong with his application.

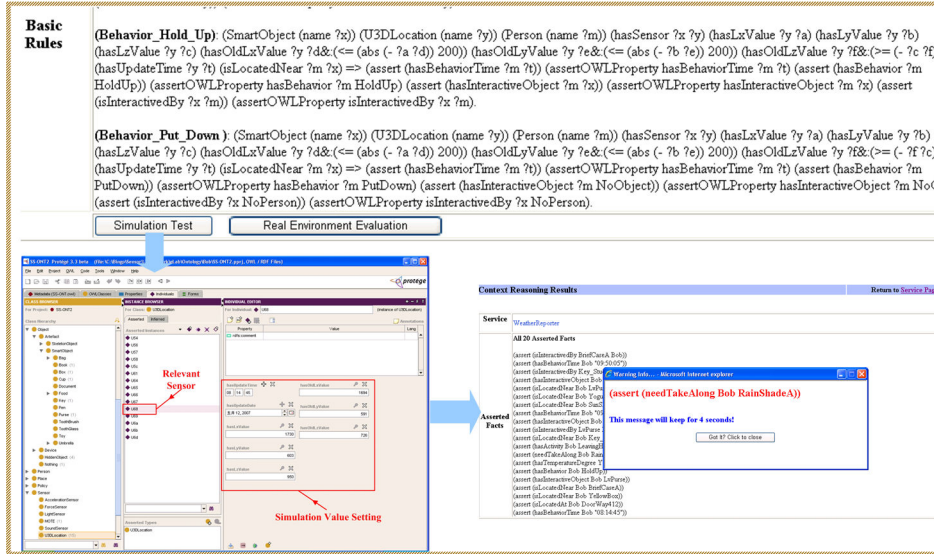


Figure 5.2: Simulation checking process

The simulation mode, compared to the real context sensing mode, is a much easier and quicker way for users to test and debug the newly created applications. After passing the simulation test, we can evaluate the application under a real sensing environment. Benefiting from our *Open-Programming* environment, if a customizer finds some errors or problems that he can not solve, he can seek help from the service developer or an online discussion forum for possible reasons and solutions.

5.3 Hardware Error Checking

We also developed a mechanism to deal with application errors caused by fallible sensors. As shown in Fig. 5.3, this rule-based mechanism can help users detect and locate hidden objects, and it also provides methods to identify the detected hidden objects.

5.3.1 Physical Relations among Objects

Similar to other smart artifact applications mentioned previously, detection of hidden objects is also based on a set of user-defined rules. The rules are mainly abstracted from various physical relations among objects.

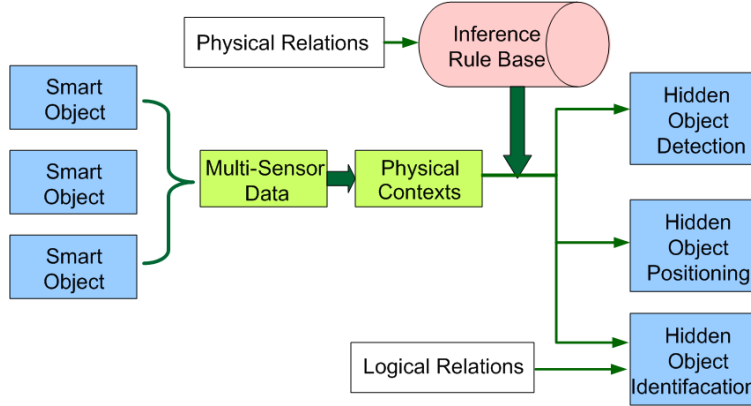


Figure 5.3: A diagram of our error-checking mechanism







Relation Type	Illustration	Relation Type	Illustration
Location Relations	 <i>On, Under</i>	Force Relations	 <i>Pressure</i>
	 <i>In, Out</i>	Motion Relations	 <i>Relative motion Relative rest</i>
	 <i>Near, Adjacent</i>	Sender & Receiver	 <i>Illumination Sound...</i>

Figure 5.4: Physical relations among objects

As shown in Fig. 5.4, there are different kinds of physical relations among objects, including location (or spatial) relations, force relations, etc. By making use of predefined rules and our context infrastructure mentioned in Chapter 3, these relations can be accurately derived. For example, in Section 4.5.2, we have described a rule that can determine the “*On*” relation between a smart object and a skeleton object (see *R4.7*). Following we will give another example, which can detect the “*Under*” relation between two smart objects.

Using U3D sensors to determine if one object is under another one is not an easy thing. As Fig. 5.5 shows, owing to the relatively fixed spatial relations between the object and its sensors, and different locations of two objects *X* and *Y*, the horizontal distance, (x_4, y_4) , i.e., $(|x_1 - x_2|, |y_1 - y_2|)$, between them will be different (because the data from their location sensors only represent two object locations in the real world but do not represent whole object spaces). For simplicity, we merely specify that all U3D sensors are attached to the top surface of smart objects. In Fig. 5.5, it can be seen that, although the value of the pair (x_4, y_4) is changing, it cannot exceed the range of (l, w) , where *l* and *w* separately denote the sums of object *X* and *Y*'s lengths and widths. This condition can, to a certain

5.3 Hardware Error Checking

extent (an exceptional case is discussed in Section 6.2.2), ensure that there is an overlap between X and Y . The second condition comes from the difference in height between these two U3D sensors, i.e., if the Z-coordinate value difference ($z1-z2$) is larger than the height of object X , we can conclude that Y is under X . This rule ($R5.1$), represented in the form of SWRL, is given in Eq. (5.1).

$$\begin{aligned}
 & SmartObject(?X) \wedge SmartObject(?Y) \wedge hasLength(?X, ?l1) \wedge \\
 & hasWidth(?X, ?w1) \wedge hasHeight(?X, ?h1) \wedge hasLength(?Y, ?l2) \wedge \\
 & hasWidth(?Y, ?w2) \wedge hasHeight(?Y, ?h2) \wedge hasSensor(?X, ?c) \wedge \\
 & hasSensor(?Y, ?d) \wedge U3D(?c) \wedge U3D(?d) \wedge hasLxValue(?c, ?x1) \wedge \\
 & hasLxValue(?d, ?x2) \wedge swrlb:subtract(?x3, ?x1, ?x2) \wedge swrlb:abs(?x4, ?x3) \wedge \\
 & swrlb:add(?l, ?l1, ?l2) \wedge swrlb:greaterThanOrEqual(?l, ?x4) \wedge \\
 & hasLyValue(?c, ?y1) \wedge hasLyValue(?d, ?y2) \wedge swrlb:subtract(?y3, ?y1, ?y2) \wedge \\
 & swrlb:abs(?y4, ?y3) \wedge swrlb:add(?w, ?w1, ?w2) \wedge swrlb: \\
 & greaterThanOrEqual(?w, ?y4) \wedge hasLzValue(?c, ?z1) \wedge hasLzValue(?d, ?z2) \wedge \\
 & swrlb:subtract(?z3, ?z1, ?z2) \wedge swrlb:greaterThanOrEqual(?z3, ?h1) \\
 & \rightarrow isUnder(?Y, ?X) \wedge hasNoUnderObject(?X, false)
 \end{aligned} \tag{5.1}$$

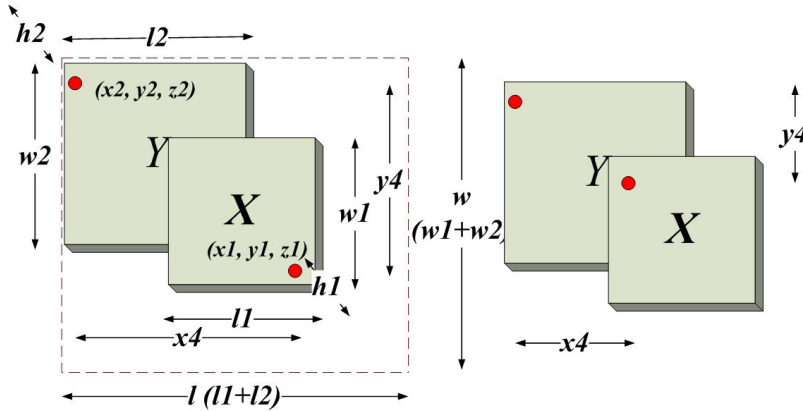


Figure 5.5: Spatial relations between smart objects

5.3.2 Hidden Object Detection

Look back to Fig. 5.1 to see how physical relations among objects can be used to detect hidden objects. By analyzing the physical relations between smart object *Book-A* and skeleton object *Table-B*, we can infer that *Book-A* is not placed directly on the desktop because there is a height difference between them, then the knowledge of physics indicates that there must be something between *Book-A* and the desktop; however, by executing $R5.1$ for smart objects, we find that none of them is located under *Book-A*, so we may conclude that there is a hidden object. Based on this analysis we can conclude two things about the hidden object: first, that it is present and second, its approximate

location. In other words, physical-relation based reasoning can effectively and simultaneously detect and locate hidden objects.

All the inference rules derived from physical relations among objects are represented as SWRL rules and stored in the *Inference Rule Base* (see Fig. 5.3). Based on the different types of relations the rules concern, we formulated four types, examples of which are presented below.

(1) *Spatial relation*

Rule 5.2: If smart object A is on skeleton object B and is laid flat (i.e., not tilted), there is no touch point between A and B , and there is no other smart object under A , then we infer that there is a hidden object C between them (as illustrated in Fig. 5.1). This is formulated in Eq. (5.2).

$$\begin{aligned}
 & isLocatedOn(?A, ?B) \wedge hasTiltAngle(?A, false) \wedge hasSensor(?A, ?a) \wedge \\
 & hasHeight(?A, ?b) \wedge hasHeight(?B, ?c) \wedge hasLzValue(?a, ?d) \wedge \\
 & swrlb:subtract(?e, ?d, ?c) \wedge swrlb:subtract(?f, ?e, ?b) \wedge \\
 & swrlb:greaterThanOrEqual(?f, 10) \wedge hasNoUnderObject(?A, true) \\
 & \wedge isDetected(?C, false) \rightarrow isLocatedOn(?C, ?B) \wedge isNear(?C, ?A) \wedge \\
 & isDetected(?C, true) \wedge hasHeight(?C, ?f)
 \end{aligned} \tag{5.2}$$

Several comments should be made about Eq. (5.2). First, the height (size) of A and B is separately represented as b and c , and A 's real-time z -coordinate value (data from its sensor) is represented as d , then the symbol f ($f = d - c - b$) denotes the height of a potential object C . If this height value is above a threshold M (in Eq. (5.2), $M = 10$ mm), then we conclude that hidden object C exists, otherwise not. Threshold M is used to reduce the influence of sensor noise, which causes fluctuations in sensor values even when the object does not move.

Second, the SWRL rules can not deduce that there is some new individual that has not been defined in the OWL ontology (though they can derive, assign, and update the property values of existing individuals), we must define an individual *HiddenObject* (e.g., *Hidden01*) in our ontology but set its *hasDetected* property to *false* before a search using the SWRL rules is attempted. When, by some inference rule such as *R5.2*, a new hidden object is detected, the present individual *HiddenObject*'s (*Hidden01*'s) *hasDetected* property will be set to *true*. Meanwhile, a new individual *HiddenObject* (like *Hidden02*)

5.3 Hardware Error Checking

will be created in our ontology. Of course, its *hasDetected* property will again be set to *false* when it is created.

Third, by using this rule, we not only detect the hidden object, but also get its relative location, which is reflected by *isLocatedOn* and *isNear* in Eq. (5.2).

Fourth, the *hasTiltAngle* property determines whether the relative object is laid flat or tilted, the value of which can be determined by the two axis acceleration values from the Mote sensor the object equipped. This determining process is also implemented according to several rules, and one rule that asserts that the object is laid flat is given in *R5.3* (see Eq. (5.3)).

$$\begin{aligned}
 & \text{SmartObject}(?x) \wedge \text{hasSensor}(?x, ?y) \wedge \text{Mote}(?y) \wedge \text{hasAxValue}(?y, ?x1) \\
 & \wedge \text{hasAyValue}(?y, ?y1) \wedge \text{swrlb} : \text{greaterThan}(?x1, -1.1) \wedge \text{swrlb} : \text{lessThan}(?x1, -0.7) \\
 & \wedge \text{swrlb} : \text{greaterThan}(?y1, 0.5) \wedge \text{swrlb} : \text{lessThan}(?y1, 0.75) \\
 & \rightarrow \text{hasTiltAngle}(?x, \text{false})
 \end{aligned} \tag{5.3}$$

The main principle is that different tilt angles can yield different acceleration values. First, we measured the acceleration value when the object was laid flat. Because there was noise, the measured results were bounded in two intervals (the *x*-axis in $[-1.1g, -0.7g]$, and the *y*-axis in $[0.5g, 0.75g]$ (the available range is $\pm 2g$ in the Mote sensor we used). This result is reflected in *R5.3*.

Another rule for detecting a hidden object using spatial relations is given below.

Rule 5.4: If smart object *A* is located on skeleton object *B*, *A* is tilted, there is no other smart object under *A*, and nobody is using it, then we infer that there is an object *C* between *A* and *B*, as illustrated in Fig. 5.6 (a). This is formulated in Eq. (5.4).

$$\begin{aligned}
 & \text{isLocatedOn}(?A, ?B) \wedge \text{hasTiltAngle}(?A, \text{true}) \wedge \text{isInteractedBy}(?A, \text{NoPerson}) \wedge \\
 & \text{hasNoUnderObject}(?A, \text{true}) \wedge \text{isDetected}(?C, \text{false}) \wedge \\
 & \rightarrow \text{isLocatedOn}(?C, ?B) \wedge \text{isNear}(?C, ?A) \wedge \text{isDetected}(?C, \text{true})
 \end{aligned} \tag{5.4}$$

It should be noted about this rule that, because some human action may have caused the object to tilt, we added the premise that nobody is using the object. The value of the property *isInteractedBy* will be assigned to a certain individual *Person* when human behavior is detected (*NoPerson* is a special individual *Person* defined in *SS-ONT* to indicate that nobody is present).

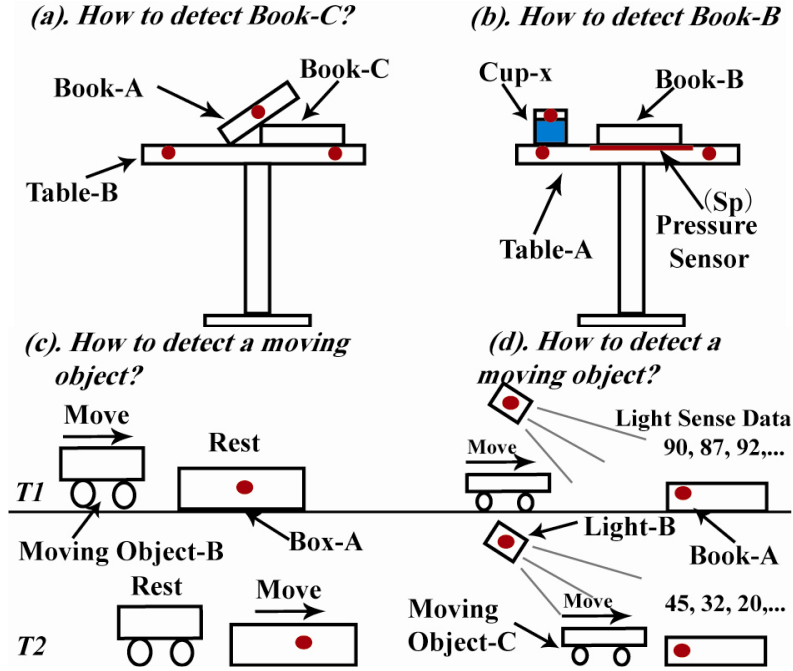


Figure 5.6: Hidden object detection scenarios

(2) Force relation

Rule 5.5: If skeleton object A is subject to downward pressure in scope Sp (action zone of the force) and there is no smart object x located in Sp , then there will be a hidden object B on A , as shown in Fig. 5.6 (b). This is formulated in Eq. (5.5).

$$\begin{aligned}
 & SkeletonObject(?A) \wedge PressureSensor(?p) \wedge hasSensor(?A, ?p) \wedge \\
 & hasActionZone(?p, ?Sp) \wedge SkeletonObject(?Sp) \wedge \\
 & hasLoadedObject(?Sp, false) \wedge isDetected(?B, false) \wedge \\
 & \rightarrow isLocatedOn(?B, ?A) \wedge isDetected(?B, true)
 \end{aligned} \tag{5.5}$$

In Eq. (5.5), the action zone of force Sp (derived from the pressure sensor values) is represented as an individual *SkeletonObject*. In this way, we can easily use the smart object localization rules, such as *R4.7*, to determine whether a smart object is placed on it, i.e., to derive the value of *hasLoadedObject* (see Eq. (5.5)). Using this rule, the system can both detect hidden objects and determine some of the properties of the object, such as its size (by *action zone*) and weight (by *pressure intensity*).

(3) Change of motion state

Rule 5.6: If smart object A is located on skeleton object B , A 's state changes during period P (from *rest* to *horizontal motion*), and there is no human interacting with A , then there will be a moving object C near A , as shown in Fig. 5.6 (c) (formulated by Eq. (5.6)).

5.3 Hardware Error Checking

$$\begin{aligned}
& \text{SmartObject}(?A) \wedge \text{SkeletonObject}(?B) \wedge \text{isLocatedOn}(?A, ?B) \wedge \\
& \text{hasSensor}(?A, ?s) \wedge \text{U3D}(?s) \wedge \text{isInteractedBy}(?A, \text{NoPerson}) \wedge \\
& \text{hasLxValue}(?s, ?x1) \wedge \text{hasLzValue}(?s, ?z1) \wedge \text{hasOldLxValue}(?s, ?x2) \wedge \\
& \text{hasOldLzValue}(?s, ?z2) \wedge \text{swrlb} : \text{subtract}(?a, ?x1, ?x2) \wedge \text{swrlb} : \text{abs}(?x, ?a) \wedge \quad (5.6) \\
& \text{swrlb} : \text{greaterThanOrEqual}(?x, 100) \wedge \text{swrlb} : \text{subtract}(?b, ?z1, ?z2) \wedge \\
& \text{swrlb} : \text{abs}(?z, ?b) \wedge \text{swrlb} : \text{lessThanOrEqual}(?z, 10) \wedge \text{isDetected}(?C, \text{false}) \\
& \rightarrow \text{isLocatedOn}(?C, ?B) \wedge \text{isNear}(?C, ?A) \wedge \text{isDetected}(?C, \text{true})
\end{aligned}$$

In Eq. (5.6), if the x -coordinate value difference between two U3D sensor updates ($|x1 - x2|$) is above some threshold M (in Eq. (5.6) it is assigned to 100 mm) and the z -coordinate value difference ($|z1 - z2|$) is below another threshold N (set to 10 mm in Eq. (5.6)), then we conclude that there is movement toward A . Thresholds M and N are also used to reduce the influence of sensor noise. However, Eq. (5.6) only makes an assertion based on the object's displacement relative to the x -coordinate, so a symmetric rule that relative to the y -coordinate is also required. As these two rules are very similar (they merely change the relative x variables in Eq. (5.6) to y variables), we don't list the second one in detail.

(3) Senders and Receivers

The term, sender-receiver pair, refers to several smart objects cooperating to monitor the environment. In such a pair, one object acts as the physical signal sender (e.g., a light source), and others act as signal receivers by using equipped sensors (e.g., a light sensor). *Rule 5.7*, given below, illustrates how to use smart object pairs to detect a hidden object.

Rule 5.7: Of smart objects A and B , B is a stable light source, and A can sense the light B transmits. If A 's light sensing value changes during period P (from bright to dark) and A is not acted upon by any person, then there is an object C between A and B , as shown in Fig. 5.6 (d). This is formulated in Eq. (5.7).

$$\begin{aligned}
& \text{SmartObject}(?A) \wedge \text{SmartObject}(?B) \wedge \text{isLuminous}(?B) \wedge \\
& \text{hasSensor}(?A, ?s1) \wedge \text{Mote}(?s1) \wedge \text{hasSensor}(?B, ?s2) \wedge \text{Mote}(?s2) \wedge \\
& \text{isInteractedBy}(?A, \text{NoPerson}) \wedge \text{hasLightValue}(?s1, ?x1) \wedge \\
& \text{hasOldLightValue}(?s1, ?x2) \wedge \text{swrlb} : \text{subtract}(?a, ?x2, ?x1) \wedge \quad (5.7) \\
& \text{swrlb} : \text{greaterThanOrEqual}(?a, 30) \wedge \text{hasLightValue}(?s2, ?y1) \wedge \\
& \text{hasOldLightValue}(?s2, ?y2) \wedge \text{swrlb} : \text{subtract}(?b, ?y1, ?y2) \wedge \\
& \text{swrlb} : \text{abs}(?c, ?b) \wedge \text{swrlb} : \text{lessThanOrEqual}(?c, 20) \wedge \text{isDetected}(?C, \text{false}) \\
& \rightarrow \text{isNear}(?C, ?A) \wedge \text{isNear}(?C, ?B) \wedge \text{isDetected}(?C, \text{true})
\end{aligned}$$

In Eq. (5.7), $x_2 - x_1$ and $|y_1 - y_2|$ separately denote the change in A and B 's light sensing value between two Mote sensor updates. $x_2 - x_1$ is above some threshold M (set to 30 mm in Eq. (5.7)), but $|y_1 - y_2|$ is less than another threshold N (set to 20 mm in Eq. (5.7)). Therefore, it can be concluded that B does not change but that A 's light sense value does change.

5.3.3 Hidden Object Identification

In last section, we have presented the mechanism to detect hidden objects. However, when hidden objects are detected, there still needs a way to inform their information and, if possible, their identities to programmers and users. To this end, we made an extension to our real-world search service (presented in Section 4.5.2) by adding several search modes for hidden objects, as shown in Fig. 5.7.

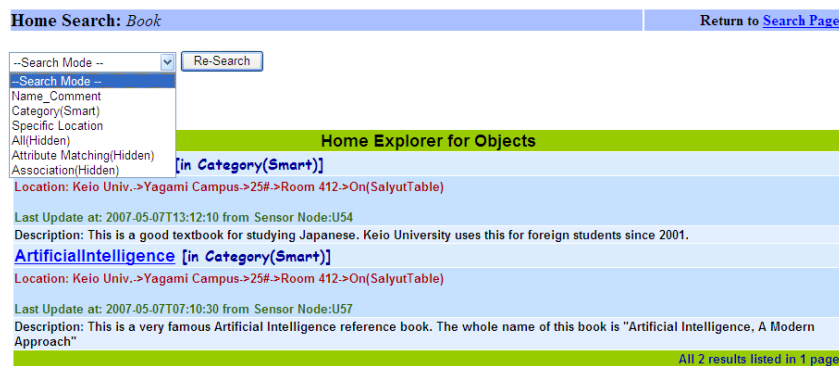


Figure 5.7: A screenshot of the extended real-world search service

Search All: This method lists all the detected hidden objects. In the ontology's view, this mode can be interpreted as returning all individuals that belong to the *HiddenObject* class. This search mode is useful for a programmer, who can get a global view of hidden objects and check if there are any hardware errors to his application.

Search by Attribute Matching: One general way to recognize an unknown object is by its attributes. The more attributes the system recognizes, the more easily it can determine its identity. Therefore, to the system should acquire as much of the hidden object's attributes as possible. In reality the system acquires attribute information during the hidden object detection process, such as the height value in *R5.2* and the weight and size attributes in *R5.5*. A novel way of recognizing hidden objects can be implemented based on the derived attributes and the *category standard* (see 3.2.2) defined in *SS-ONT*. Figure 5.8 shows two entities: *Key Category* (the key category standard) and *Hidden06* (a hidden object). *Key Category* consists of five specific attributes, while *Hidden06* has four

5.3 Hardware Error Checking

derived attributes. Note that *Hidden06*'s attributes all drop in the relative intervals or have the same value in comparison with those in *Key Category*. This means that we can assign a degree of probability or match rate, 0.8, to the proposition that *Hidden06* is a key.

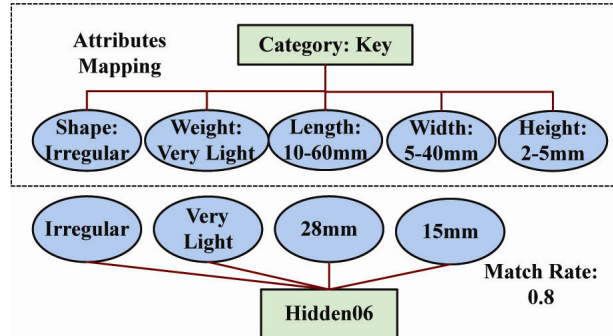


Figure 5.8: One example for attribute matching

The idea of this approach is similar to the topological or fingerprint mapping used for robot navigation and localization [Lamon *et al.* 03], which compares perceptual object-features with known database data (for place description) to estimate a robot's current position. It's also similar to the role-based classification approach mentioned by Beer *et al.* [Beer *et al.* 03], which classifies an unknown object by comparing its attribute sets with the known attribute templates. Approaches like these all work with uncertainty, mainly informing users of the probability of each hidden object being the target, leaving the truth to be determined by the user.

Search by Associated Rules: This approach is based on the logical relations among objects that we discussed in 3.2.2. A summary of these relations and their examples are illustrated in Fig. 5.9. A definition of how the logical relations work is given below.





Relation Type	Illustration	Examples
Combinational Relation		Display-Mouse-Keyboard, Projector-Projector Screen.
Family Relation		Book-Book CD-CD Umbrella-Umbrella
Partner Relation		Table-Chair Pen-Book Electronic Dictionary-English TextBook
Functional Relation		Toothbrush-Glass Bookshelf-Book Water Pipe-Sink

Figure 5.9: Logical relations among objects

Definition 5.1 (Associated Rules): there are various logical relations existing among objects. According to the natural qualities and functions of objects, or humans' daily habits or intentions, the objects with such relations are usually close to each other in the real world. We call these the associated rules.

According to our system, since the hidden objects are detected by nearby smart objects, according to Def. 5.1, associated rules may exist among them. Thus we can conjecture what the hidden object is using smart object information. For instance, when a hidden object *Hidden01* is detected by a smart object *Book-A*, since the *Boolean* “*hasFamilyRelation*” property is set to *true* for the *Book* category (see Fig. 3.3), we may conclude that *Hidden01* is also an individual *Book*. It should be noted that this kind of estimate is a low probability one (lower than attribute matching). However, because it is simple and efficient, we can still treat it as an effective method for our system.

5.4 Other Approaches to Deal with Uncertainties

Having presented an approach to deal with hidden objects, in this section we will give an outlook of several other ways to deal with the errors mentioned in Section 5.1.

(1) For technical problems that are difficult or impossible to solve, we should reveal them to end-user developers. In this way, users can understand it and will pay attention to it when creating and utilizing the services.

(2) Similar to the suggestions from previous reports [Benford *et al.* 03], programmers can tailor the created applications around the technical shortcomings or even explore them as part of our application. One example can be found in the Treasure game (see Section 4.5.1), which explores the “limited coverage” weakness of ultrasonic sensors to “hide” objects.

(3) More inconsistencies are induced by conflicting conclusions from different rules. As mentioned in Section 3.4.2, the SWRL rule language that we used in our system, is a combination of the OWL DL and OWL Lite sublanguages, and it has been given a strictly monotonic interpretation, i.e., once asserted a statement is never retracted. As a result, contradictory conclusions are allowed to be drawn under current Semantic Web semantics. To deal with this issue, we can explore the defeasible logic [Pollock 87] or the answer-set-programming technique [Eiter *et al.* 04] to introduce well-founded

5.4 Other Approaches to Deal with Uncertainties

nonmonotonic semantics into ontology-based context-aware reasoning. Preliminary progress has been reported in Antoniou et al. [03] and Eiter et al. [04].

(4) Finally, we need further error-checking and recommendation mechanisms that can periodically alert users to update the ontology definition, remind them of frequently-occurring problems, and inform them of the errors in their application settings.

Chapter 6. Evaluation Study

Having presented the implementation of our system, in this chapter, we will give an evaluation of it to measure its performance, effectiveness as well as usability.

6.1 Evaluation Tasks

The usability of *Sixth-Sense* is determined by several aspects. To take an ordinary user, called Bob, for example (because *Sixth-Sense* is a user-oriented platform), before he decides to use our system, there are several things to be examined. First, the performance of this system, for example, “*can it respond to my action in time*”, “*can it in reality detect my interaction with everyday objects*”. These questions can be answered by evaluating the knowledge infrastructure we described in Chapter 3. Second, before using our system, Bob has to firstly customize an ontology for his home. Therefore, he needs to check if he can fulfill this task utilizing the method presented in Section 3.2.3. Third, there are two programming modes (*rule-based mode* mentioned in Section 4.3 and *customization mode* mentioned in Section 4.4) in our *Open-Programming* model, and Bob will experience both of them to determine a suitable mode for him. If he can perform well the operations in the prior three steps, our system is usable to him. Finally, he wants to check if the enabled applications (see Section 4.5) of our system are attractive to him. If they are very interesting to him, he will be willing to use our system.

Content	Purpose	Subjects and Event	Method	Duration
System Performance	To determine the performance and effectiveness (including error checking mechanism) of our infrastructure	Keio Univ. 2007.10	Different settings for reasoning performance testing; rule testing to see if they work well	About one week
Ontology Customization	To determine if users can customize an ontology for their home via Protege editor	14 subjects from Keio Univ. 2008.3	Hands-On Operations, Questionnaire, Observation and Interview	20 minutes for each subject
Open-Programming Model	To determine if this model empowers users with different abilities to program artifact-based applications	(1) Keio Techno-Mall, 07 (2) 29 subjects from Keio, 2008.3-9	Hands-On Operations, Questionnaire, Observation and Interview	(1) 7 hours exhibition; (2) 40 minutes for each subject
User Experience	To obtain users’ feedback and reaction about the enabled smart artifact applications	(1) Keio Techno-Mall, 07 (2) 15 subjects from Keio, 2008.9	Demonstration, Questionnaire, Discussion	(1) 7 hours exhibition; (2) 20 minutes for each subject

Figure 6.1: Evaluation tasks

As shown in Fig. 6.1, during our two and a half years’ development experience (2006.4~2008.11) of *Sixth-Sense* system, we set out to evaluate it (according to the above

6.2 Performance Study

four aspects) through a series of events, including in-lab demonstrations, out-lab exhibitions, and user studies with subjects from Keio University. Mixed methods were used, including hands-on operations, demonstrations, questionnaires, observations during user experiences, as well as interviews and discussions at the end of each session.

We used our workspace, a 4.0×4.0 m portion of our laboratory, as the test bed (see its layout in Fig. 6.2). It can be seen that there are several skeleton objects in our test space (e.g., *Neptune-Table*), and the floor is also virtually divided into several parts (e.g., *Doorway*, *FloorScopeA*) to represent different skeleton objects.

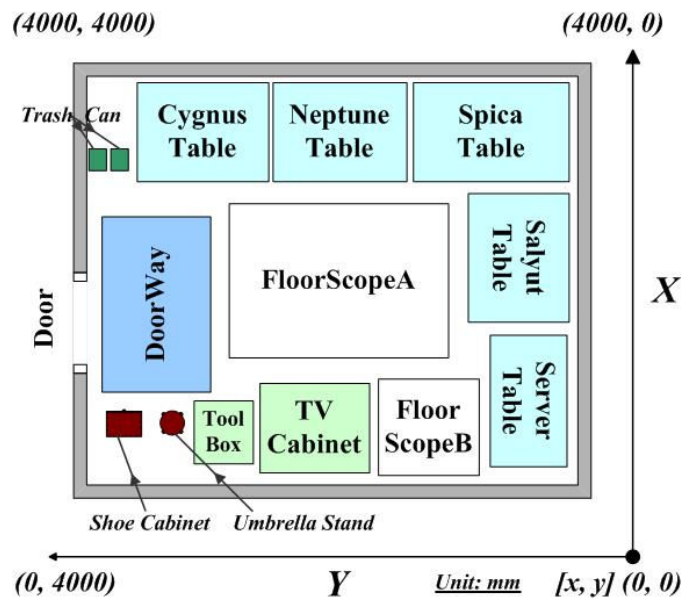


Figure 6.2: The test environment for Sixth-Sense

6.2 Performance Study

This section evaluates the performance our *Sixth-Sense* infrastructure described in Chapter 3, which involves two aspects: (1) its runtime performance; (2) the effectiveness of our rule-based reasoning mechanism.

6.2.1 Evaluation of Runtime Performance

(1) Performance of Context Reasoning

We identified context reasoning (the reasoning mechanisms are presented in Section 3.4) as a potential performance bottleneck of our infrastructure, so we did a series of experiments to evaluate its runtime performance. These experiments were conducted on two Windows workstations with different hardware configurations (1.0 GB RAM with

P4/1.0GHz, and P4/2.6GHz). We used three context data sets to evaluate our system’s scalability. These three test data sets, including the real one we used in our test environment, i.e., *SS-ONT-v1.5*, which can be parsed into about 2000 RDF triples, and two other simplified versions of *SS-ONT* (with fewer classes and instances), amount to about 1000 triples and 500 triples, respectively. We used four rule sets to test the performance of our context reasoner. The smallest rule set includes only one rule, and the biggest set has twenty rules.

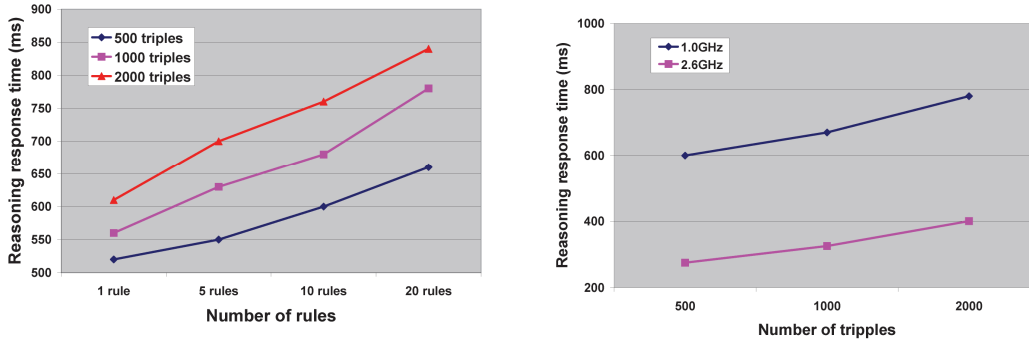


Figure 6.3: Performance of reasoning with the changes of ontology scale (left) and CPU speed (right)

The results of the experiments are illustrated in Fig. 6.3 (results are calculated as the average of five runs.). From these results, it is not difficult to conclude that logic-based context reasoning depends on four major factors: size of ontology, CPU speed, number of rules applied, and complexity of rules. More concretely, the size of ontology is determined by its *TBox* definition — definitions about OWL classes and properties, and *ABox* definition — definitions about instances, i.e., the number of smart objects and persons in a smart home. The prior part is almost the same in different homes while the latter part changes among different families. The complexity of a rule is in essence determined by the number of atoms in this rule. From the above observations, we further derive an estimation model that can be used to anticipate the reasoning time for different scales of smart spaces, as shown in formula Eq. (6.1).

$$ReasoningTime \approx \mu \frac{(Size_{TBox} + \sum_{i=1}^M (Class_Num_i + ProNum_Obj_i))^{\exp1} * (\sum_{i=1}^N Comp_Rule_i)^{\exp2}}{Speed_{cpu}} \quad (6.1)$$

Where: $Size_{TBox}$ denotes size (in triples) of the *TBox* definition of *SS-ONT*; M and N denote, respectively, the number of individuals (e.g., smart objects and persons) and the

6.2 Performance Study

number of rules; $Class_Num_i$ and $ProNum_Obj_i$ denote the number of parent classes and the number of properties for a specific individual i ; Complexity of a rule i is denoted by $Comp_Rule_i$; $exp1$ and $exp2$ are two exponents whose values are between 0 and 1 (according to the results shown in Fig. 6.3, their values can not be bigger than 1), which reflect the rate of change of the reasoning time with the growth of ontology size and complexity of rules; μ is a positive coefficient that relates the change in reasoning time to the change in its influence factors (e.g., ontology size, number of rules, etc.).

Let $exp1 = 1$ and $exp2 = 1$, we can get the upper-limit value of reasoning time, i.e., the anticipated maximum reasoning time. Suppose there is a 1000 triple-sized ontology, and 10 rules with a mean complexity of 8, we can read from Fig. 6.3 that its reasoning time is about 680 ms (P4/1.0 GHz). Using these values to replace the related variables in Eq. (6.1), we can derive an approximate value for μ : $\mu = 0.085$. The proposed estimation model is useful for us to anticipate the computational overhead for a given-sized problem. As listed in Table 6.1, for a middle-scaled smart space (a space with 50 smart objects and 50 rules), the anticipated maximum reasoning time is about 1.7s; whereas for a large-scale one (100 smart objects and 100 rules involved), this value increases to 2.8s.

Table 6.1: Anticipated maximum reasoning time in different scales of smart spaces.

Smart Spaces	Size of TBox definition (in triples)	Number of Individuals	Parent Classes (Mean)	Number of Properties (Mean)	Number of Rules	Complexity of Rules (Mean)	CPU Speed	Maximum Reasoning Time
Middle-Scale	500	50	3	7	50	8	2.0 GHz	1.7s
Large-Scale	500	100	2	6	100	7	3.0 GHz	2.8s

The performance study results reveal that rule-based context reasoning is a computation-intensive task, and the reasoning time will be human-perceivable if the scale of the smart space increases. However, for most non-time-critical applications (e.g., searching for smart objects), as their real-time requirement is not likely to be critical, a perceivable delay (one or two seconds) caused by context reasoning is acceptable. The evaluation results also suggest that, with a suitably scaled ontology size and rule-set complexity, our rule-based context reasoning mechanism can also work in some time-critical applications, for example, security and emergency situations.

(2) Performance of Context Querying

We used SPARQL query statements like Eq. (3.1) to test the performance of context querying (the querying mechanism is described in Section 3.3). The experiment result, shown in Fig. 6.4, clearly demonstrates that the increase of matched facts results in a corresponding increase of the querying response time. This is mainly because a bigger number of matched facts imply that there are more contexts (in *SS-ONT*) defined in the graph pattern of the query statement applied, which cost more matching time. Size of the ontology applied becomes another factor that influences the querying performance of our system. We also measured the loading time (time cost from OWL files to the main memory) of different-scaled ontology data sets. The result shown in Fig. 6.4 indicates that the loading time of an ontology data set is, to some extent, proportional to its size.

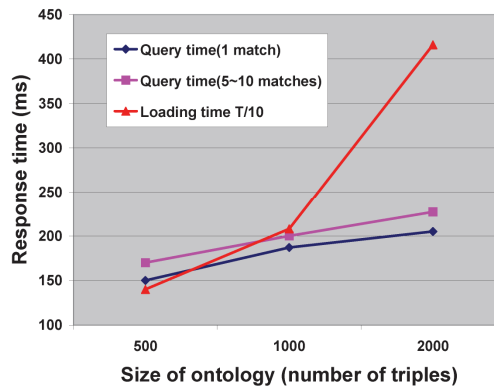


Figure 6.4: Performance of querying

Above experiment results and the proposed computational model also suggest several possible ways to improve the performance of our system. Controlling the scale of context dataset seems to be one good way to significantly reduce the context reasoning time. For example, we can separate the *SS-ONT* ontology into several sub-domain ontologies (e.g., kitchen domain and bathroom domain), and provide several resource-rich devices to process them respectively. Designing optimized rules and utilizing high-performance processors are two other effective ways to achieve a better reasoning performance.

6.2.2 Evaluation of Effectiveness

Because our context reasoning mechanism (including the error-checking mechanism mentioned in Section 5.3) is layered on a series of inference rules, the effectiveness of the defined rules will directly influence the performance of our system. In the following, we evaluate the usability of our system by testing the defined rules.

6.2 Performance Study

Eight typical inference rules explained in this thesis are tested (including several rules for hidden object detection). The experiments were performed as follows: For each inference rule R_i , we set the test time as 45 minutes, during which a situation similar to R_i 's scenario recurred 50 times. However, to test R_i 's performance in different situations, the 50 tests were performed alternately using object pairs in two different places (i.e. 25 times for each). For example, for $R5.2$, there are three objects referred (expressed as (A, B, C)), so, in the test, the two test-groups can be set as $(book, table, book)$ and $(box, floor, book)$. Following we introduce three terms to distinguish different test results.

Definition 6.1 (Diverse Test Types): In our system, an *accurate* test is one in which an object property (e.g., location) or a hidden object is detected at the right time by the right rule, it scores a true positive (TP). An incorrect claim scores a false positive (FP), and it's an *error* test. A *failed* test is one in which an inference rule should have been triggered but was not, which scores a false negative (FN).

We then used two standard metrics to summarize our system's effectiveness. *Precision* is the probability that a given inference about an object property (or a hidden object) is correct. *Recall* is the probability that our system will correctly infer a given true event (or a hidden object situation). They can be expressed in formula Eq. (6.2):

$$Precision = \frac{TP}{TP + FP}; Recall = \frac{TP}{TP + FN} \quad (6.2)$$

Table 6.2: Experiment results of effectiveness

<i>Rule Name (Rule ID)</i>	<i>True positives</i>	<i>False positives</i>	<i>False negatives</i>	<i>Precision (%)</i>	<i>Recall (%)</i>
Is_Located_On (R4.7)	48	0	2	100	96
Is_Under (R5.1)	47	5	3	90	94
Hidden_Spatial_1 (R5.2)	45	7	5	87	90
Is_Tilted (R5.3)	44	3	6	94	88
Hidden_Spatial_2 (R5.4)	42	10	8	81	84
Hidden_Force_1 (R5.5)	46	0	4	100	92
Hidden_Motion_1 (R5.6)	40	12	10	77	80
Hidden_SenderReceiver (R5.7)	42	11	8	79	84
Total	354	48	46	88	89

The results are shown in Table 6.2, where it can be seen that *Sixth-Sense* correctly inferred that an event occurs 88 percent of the time. For two inference rules ($R4.7$ and $R5.5$), there were no false positives. Of the totally 400 tests (50 tests for each rule

multiplied by 8) that actually happened, *Sixth-Sense* detected 89 percent correctly. Rules for smart objects (*R4.7*, *R5.1* and *R5.3*) performed better than the hidden object detection rules (the other five rules). This is mainly because most of the hidden object detection rules are founded on the rules about smart objects. For example, to use *R5.4*, we must first derive information about the related smart object *A* that is referred to in this rule. This includes its location, its status, and whether there is any other smart object under it, which should be previously deduced by smart object rules.

The results indicate that our rule-based detection mechanism is technically sound, and the rule-based hidden-object detection mechanism can, to a certain extent, detect the hardware errors caused by fallible sensors. By analyzing the real-time testing data, we generalized the following reasons that a failed or error test may occur.

(1) *Sensor noise*: sometimes sensor noise exceeds the threshold value we set in the rules, which may trigger an inference rule, causing an error test.

(2) *Object movement*: when a smart object is moving, it will take more time for the sensor receivers to relocate it, which may require a longer sensor update time and induce loss of important process data.

(3) *Sensor delay and data loss*: when more sensors are working, the sensor receivers have to cope with them one by one, resulting in more delays. Moreover, as the number of sensors increases, the processing capacity of the sensor system decreases. According to our experimental statistics, when 15 U3D sensors are working simultaneously, about 4% of the sensor data will be lost.

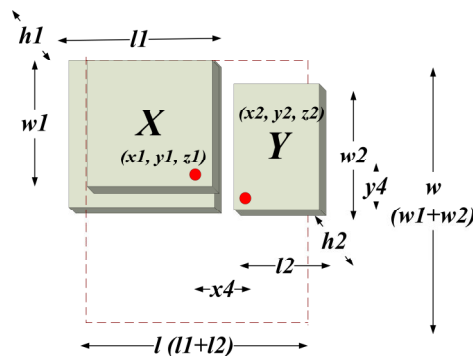


Figure 6.5: An exception case for R5.1

(4) *Inference rule defects*: some other reasons for failure may originate from the inference rules themselves. Because of the imprecise common sense knowledge background, it is sometimes difficult to create a faultless inference rule. Figure 6.5 gives

6.3 Ontology Customization

an example of an exception to *R5.1*. If the three objects *X*, *Y*, and *Z* (*Z* is under *X* and is a hidden object) are placed in this way, the horizontal distance between *X* and *Y* (i.e., (x_4, y_4)) still does not exceed the range of (l, w) , and, since *Z* is under *X*, the height difference $(z_1 - z_2)$ is also larger than the height of object *X*. Therefore, *R5.1* will be triggered and will assert that *Y* is under *X*. With this incorrect conclusion, the detection of hidden object *Z* will also fail (as the *hasUnderObject* property value to *X* is true, *R5.2* cannot be triggered).

In summary of the above evaluations, to further improve our system’s performance, better sensor technologies, transmission protocols, underlying common sense knowledge, and reasoning technology still need to be used.

6.3 Ontology Customization

To use our system, users should firstly customize an ontology for their home. This section validates the feasibility of the customization method depicted in Section 3.2.3. We set out to evaluate this with 14 students from Keio University (ages ranging from 23 to 33), all of which have at least four years’ computer experiences. During the evaluation sessions, we asked subjects to complete three hands-on tasks using the Protégé-OWL editor. The three tasks, listed in Table 6.3, involve several basic operations that are frequently used in ontology customization, for example, creating new subclasses (or individuals), specifying property domains and ranges, and specifying individual properties. We believe that if users can correctly complete these operations without or with little aid, they will be able to configure the *SS-ONT* ontology by themselves.

Table 6.3: Tasks for ontology customization

<i>Task ID</i>	<i>Description</i>
1	Assume you want to develop a new smart-artifact application, where a subclass to class “Cup”, named “CoffeeCup”, is required. Please add this subclass via the Protégé-OWL editor.
2	Please add a new object property named “hasSensor”, and set its “Domain” to “Object” and “Range” to “Sensor”.
3	Please add a new instance named “BlueCoffeeCup” for class “CoffeeCup”. Assume “BlueCoffeeCup” is equipped with a “U3D” sensor (named “U67”), please specify this property in the “Individual Editor” panel.

All subjects but one had none prior knowledge about ontology or the editing tool. One subject, whose major is computer science, told us that he had ever studied the Protégé-OWL editor in class. The hands-on tasks were carried out on a notebook pc that installed

the Protégé-OWL editor. A typical test process advanced in this way: one of our team members firstly started the Protégé-OWL editor, loaded the *SS-ONT* ontology file and switched to the “*Classes Tab*”, the subject in this session was then asked to finish the first task. When this task was finished, the editor interface was switched to the “*Properties Tab*” and the “*Individuals Tab*” for the other two tasks.

We didn’t teach anything about the use-method of the ontology editor and subjects had to find out how to use the buttons on the control panel to complete their tasks. Benefited from the rich experience accumulated from many years’ computer usage and well-designed interfaces of the Protégé-OWL editor, most users easily performed the first two tasks with few promptings. However, the relatively complicated “*Individuals Tab*” control panel confused several subjects. Though it was still easy for them to create a new “*CoffeeCup*” instance, it was a little difficult for them to specify a property for this new added object. Most subjects didn’t know how to specify a property and they looked through the interface for “*U3D*” or “*U67*”. Some promptings were provided here, such as “*All the properties are listed in this area*”, “*In the second task, you have just added a new property called ...*” Most subjects successfully completed this task under these promptings. Two subjects failed because they clicked a wrong button on the panel. Additional promptings were provided here, “*Maybe you have clicked a wrong button, why not try the adjacent one?*” Both subjects succeeded in this task after this prompting.

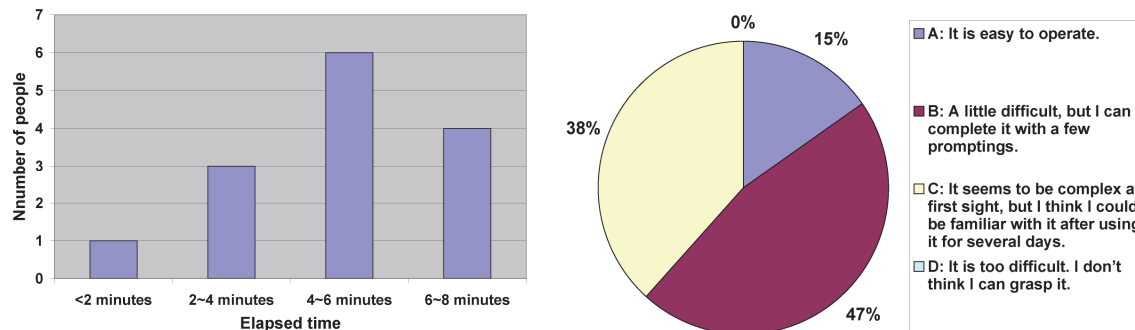


Figure 6.6: Evaluation results for ontology customization

All subjects completed the three tasks within eight minutes and for the subject who had ever learned this editor, the process shortened to be about 1.5 minutes (85 seconds). Detailed elapsed time information is illustrated in left of Fig. 6.6. Having obtained an initial impression about ontology customization, subjects were then given a detailed description about the components of the ontology model and the reasons it had to be

6.4 Open-Programming Model

configured. Note that we expressed these details in a simple language instead of using obscure technical-terms (e.g., “*ontology*” was replaced by “*knowledge base*” during the introduction). Afterwards, subjects were asked to answer the questions listed on a questionnaire sheet. One of our central aims was to find out how subjects thought about the configuration operations on the Protégé-OWL editor. Four options were given and the results are shown in right of Fig. 6.6. Two subjects believed the operations were easy and almost half the subjects thought they could complete the tasks with a few promptings. All the others felt that they could be familiar with this editor in a few days’ use. What’s more, none of them counted it as a difficult tool to master.

These results demonstrate that experienced computer users are able to customize the *SS-ONT* ontology by themselves. Though positive evaluation results, we still consider that there is some difficulty for users who have little computer experience to use this method, which is further proved by the feedback from the subject who has ever learned the Protégé editor. He argued that this editor was somewhat professional that his parents cannot easily use it. To address this, in Section 7.4.1, we suggest another possible way for user-oriented ontology customization.

6.4 Open-Programming Model

To understand the effectiveness of our *Open-Programming* platform (see Chapter 4) as a programming toolkit for both advanced users and novice users, we asked two questions:

- (1) Can users construct a new application by searching and using the shared rules and creating new rules?
- (2) Can they reprogram a shared application of interest? If so, how do they think about the solution we provide?

We set out to evaluate the above questions with 15 participates from Keio University (ages ranging from 21 to 34). The testing period extended from August to September in 2008. The subjects vary widely in gender (three are females), discipline (13 of them are non-computer students), and programming ability (11 of them have none or only a little knowledge about programming). It lasted for approximately 60 ~ 80 minutes for each subject’s test session. Each session event started by an introduction for a subject, describing him (or her) the emerging smart artifacts, artifact-based games, as well as the purpose of our programming platform. Afterwards, we handed out a four-page

questionnaire and asked the subject to perform the tasks and answer the questions written on it. Noting that to make a comparison with the two programming modes of our system (i.e., *the rule-based mode for advanced users* and *the customization mode for novices*), subjects were asked to participate in all the experiments designed for these two modes.

6.4.1 Evaluation of Rule-based Programming

To examine the rule-based programming mode presented in Section 4.3, we designed three tasks for the subjects to complete.

- We listed several rules written in the Jess language on the questionnaire to test whether they can understand them.
- To test whether the subjects can utilize existing resources, we described three events needed to be detected in a smart toy game (e.g., one event might be “*a player approaches this toy*”), and asked them to select three relevant rules from the shared-rule-browsing page (see Fig. 4.8 (a)) and give a default action setting for the selected rules.
- To determine whether the subjects can create rules by themselves, we described a scenario and asked them to create a rule accordingly. The scenario we chose is the same as rule *R4.2*'s (see Section 4.3.1), that is, “*if it rains today, alert the user to take an umbrella when detecting that he is going to leave home*”.

Following we will describe the evaluation process and the results for the three tasks.

- (1) *Reading rules*. Two typical rules we described in Section 3.4.2 (*R3.1*, *R3.2*) were selected to work for this testing. As the questionnaire fragment shows (see left of Fig. 6.7 (a)), we listed the two rules at the beginning of the related question and asked subjects if they could understand it. Furthermore, to determine if they really understood a rule, we additionally asked them to write out the meaning of a specific constant in that rule, for example, “*What does ‘25’ mean in this rule (R3.1)?*”, “*Do you know what does ‘500’ mean in this rule (R3.2)?*”. The testing results are shown in Fig. 6.8. For rule *R3.1*, almost 80% subjects could totally or basically understand it; and for the relatively complex rule *R3.2*, the percentage still arrived at a high value of 64%. The results reveal that most subjects can have a general understanding of the smart home rules written in Jess, even though they have no prior knowledge about this language. This conclusion is further validated in

6.4 Open-Programming Model

subjects’ answers to the meanings of the selected constants. All subjects succeeded in finding out the relation between “temperature degree” and the constant ‘25’ in R3.1. One subject failed to give an answer to the meaning of ‘500’ in R3.2, but all the others more or less indicated its relation with distance (some answers are coarse-grained, for example, “near or not”, “near”, and others are fine-grained, e.g., “threshold of near”).

Part 3: Creating Rules

* Here is a rule example (R1), please read it and answer the following questions:

Weather_Temperature_Degree_Rule
R1: City(?x) ^ hasTemperature(?x, ?y) ^ swrb: greaterThanOrEqual(?y, 25) ^
 → hasTemperatureDegree(?x, Hat)

(1) Do you understand it? []

A. I can understand it and even edit it for my use.
 B. Mostly understand, and I think I can totally understand it after a few hours' study of this rule language.
 C. Partly understand.
 D. Totally not understand.

(2) What does '25' mean in this rule? []

Your answer: .

(a)

Please complete a rule using the phrases listed below. Here is the description for this rule:
If it rains today, alert the user to take an umbrella when detecting that he is going to leave home.

(1) hasResidentCity(?y, ?z), (2) hasResidentCity(?x, ?z), (3) hasWeather(?z, Rainy),
 (4) hasWeather(?y, Rainy), (5) isHotDay(?z, False), (6) isLocatedNear(?x, Door),
 (7) isLocatedAt(?x, Room412), (8) hasBehavior(?x, Move), (9) Room(?y),
 (10) hasInferredActivity(?x, LeavingHome), (11) hasScheduledActivity(?x, Shopping),
 (12) Umbrella(?z), (13) Umbrella(?y)

Person(?x) ^ _____ ^ _____ ^ _____ ^ _____
 _____ → needTakeAlong(?x, ?y)

(c)

Game rules and actions:

a. When a player approaches this toy (e.g. a cup): _____ (please select a rule to detect this event),
 It reacts in this way: _____ (please specify a default sound or animation action, or a combination of them).

b. When it is tilted: _____,
 It reacts in this way: _____.

c. When it is picked up by the player: _____,
 It reacts in this way: Sound _____ (please create new audio contents for this rule, referring to the examples we give).

Sound files: A. Are you thirsty? B. Today is sunny. C. How are you? D. Thanks a lot.
 E. Please put me down. I am scared of height. F. Don't touch me! G. Bonus!

Animation files:

H. I. J. K. L. M.

(b)

Figure 6.7: Fragments of the questionnaire for rule-based mode testing

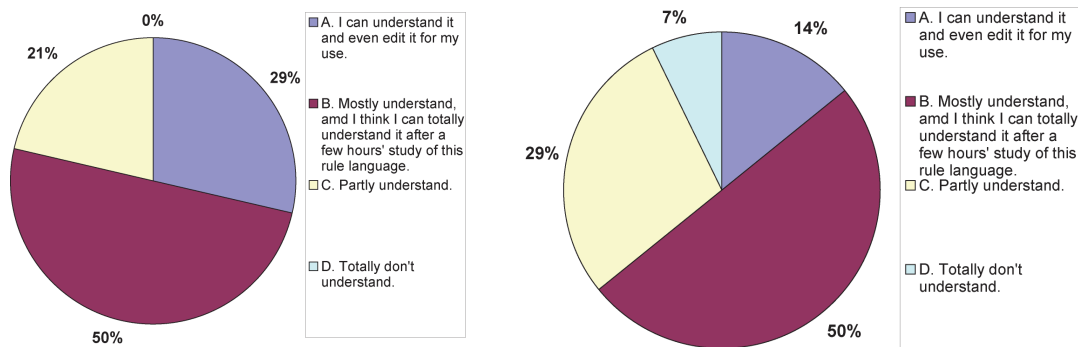


Figure 6.8: Testing results for rule R3.1 (left) and R3.2 (right)

(2) *Selecting shared resources.* The questionnaire design for this task is shown in Fig. 6.7(b). According to the given information (e.g., name, description) about the rules, all subjects except two successfully chose the three right rules from among fifteen candidates in a mean time of five minutes. This reveals that it is not a very difficult work for users to utilize shared resources when creating applications. The action-setting task seemed to be the easiest one, where testers were asked to define the sound and animation files to be played when a game event is detected. Both

resources shared in our repository and the ones created by users are allowed to be used, and a lot of interesting actions were created. For example, for the “pick-up” interaction in the game, one subject defined the audio content to be played by the toy as “Be careful! I don’t want to be hurt”.

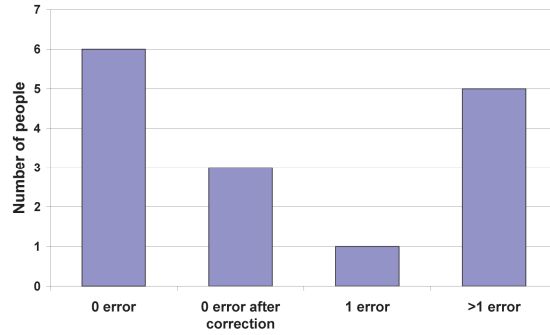


Figure 6.9: Testing result for rule creation

(3) *Creating rules.* This part was designed in the gap-filling format (see Fig. 6.7 (c)). Among the five premises of *R4.2*, only the first one, i.e., (*Person (name ?x)*), was listed on the questionnaire. Subjects were asked to fill in the other four premises by choosing the right ones from a candidate list (thirteen candidates in total). The result for this test is shown in Fig. 6.9. Almost half the subjects (six in number) gave the right answer without any guidance. Three subjects corrected their answers after the discussion at the end of their test session. The frequently asked questions during the discussions were about variables, for example, “What does ‘?x’ mean?”, “What’s the difference between ‘?x’ and ‘?y’?”. After our explanations, they realized the errors in their answers and corrected them by themselves.

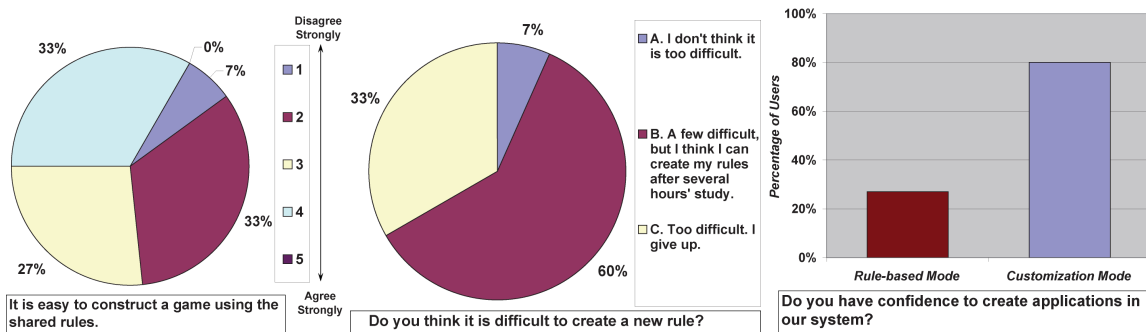


Figure 6.10: User feelings about rule-based programming

After completing the three tasks, subjects were asked about their feelings about game creation in our system. As shown in left of Fig. 6.10, about one-third subjects ($R > 3$) felt

6.4 Open-Programming Model

that it was easy to construct an application using shared rules. There are, however, about 40% of them regarded it as a difficult task ($R < 3$). This result is somewhat beyond our anticipation according to our observations during hands-on testing sessions, so we asked them for possible reasons. Most subjects said that they didn't like reading rules written in this format. One subject suggested that we should provide more information about rules (such as the principles used to design them and their use cases), which can help users understand and use them. The data illustrated in middle of Fig. 6.10 indicates that about 60% subjects thought it was a little difficult to create new rules, but at the same time they believed that they could create their own rules after a few hours' study. Five subjects (33%) considered it as difficult work and wanted to give it up, while one subject counted it as an easy task to perform. In summary, most subjects believed that they could learn to create rules after doing a few exercises. When asking testers about their confidence to game creation in our system, only 27% subjects (see right of Fig. 6.10) expressed their positive opinions ($R > 3$). This result on one hand meets the facts that advanced users only make up a small section of all users and, on the other hand, implies that we should further improve the rule-based programming method to attract more users to use it.

6.4.2 Evaluation of Customization Mode



Figure 6.11: Users in the gaming experiment (left) and the reprogramming experiment (right)
In this testing, we wanted to determine whether users, especially novice users, can reprogram a shared application by using the customization mode described in Section 4.4. To this end, we asked the subjects to customize the shared Treasure game (see Section 4.5.1) according to their imagination. This testing was performed on Treasure's configuration front-end (see right of Fig. 6.11). Subjects were asked to complete their

reprogramming task following the introductions written on this page. During the test sessions, we observed that most testers could perform this task without any guidance. But there were still several testers (four in number) who didn't perform the task according to the given requirements. For example, the requirement in Q1, “*You must specify a box and a key to act as treasure-box and treasure-box-key*”, is omitted by some of them. After finishing this, we navigated to the game-browsing page to show the tester the automatically generated rules during his (or her) customization. Many testers were surprised at this and asked us how it was realized. We then explained them our rule-interface-based customizing mechanism. Afterwards, subjects were asked to answer some questions about the customization work listed on a questionnaire.

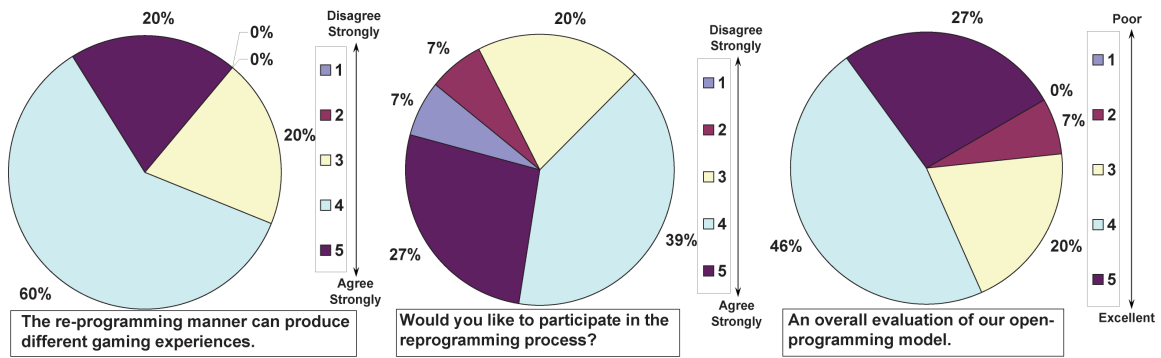


Figure 6.12: Testing results for the front-end based reprogramming

It can be seen from right of Fig. 6.10 that 80% subjects thought it was easy to reprogram a game in our system, which reflects that the configuration front-end designed for end users is simple to use. Compared to the survey result (27%) for the rule-based mode, this result suggests that much more users are enabled to participate in the activity of application creation through the customization mode. There are also 80% subjects (left of Fig. 6.12) agreed that the reprogramming manner could produce different gaming experiences, which suggests that our programming model in reality empowers novice users to create applications. To further determine the usefulness of our system, we asked testers whether they were willing to participate in the customization process. The result, shown in middle of Fig. 6.12, describes that two-thirds of them had interest in it. We finally asked testers to give an overall evaluation of our *Open-Programming* model, by comparing with the ones with a single-purpose (e.g., designed for novices or experts). As shown in right of Fig. 6.12, almost 75% users believed this model is very useful and

6.5 User Experiences

promising. This result indicates that a programming model with multi-level's participation is more acceptable than single-purposed ones.

To understand why some testers didn't follow our front-end introductions during the testing, we made a short discussion with them after answering the above questions. Here are some of their answers, *"There are too many texts displayed on this page, which makes it easy to omit one or two sentences"*, *"Does this system have multi-language support. I think it is much easier for me to use it when written in my native language"*, *"There should be a help file"*, *"Is there a flowchart to show me how to perform my task"*, *"There should be more graphics while not texts"*. From these answers, we found that the main problem arose from the design of the customization-interface, which seemed to be imperfect and unable to meet all users' requirements. This finding is further verified during the interview sessions, as mentioned in Section 6.6.

6.5 User Experiences

To determine whether users are interested in the smart artifact applications we developed, we asked them to participate in some enabled applications of our system mentioned in Section 4.5, for example, the real-world search service and the artifact-based game.

6.5.1 User Experience of the Real-World Search Service

We did a series of in-lab and out-lab demonstrations of our real-world search service. In general, most attendees' feedback was positive. As the survey result (questionnaire survey from 18 visitors, ages ranging from 20 to 55) from Keio Techno-Mall shows (see Fig. 6.13), more than 80% users were interested in the services we demonstrated. This evaluation result also implies that they expect to live in such a smart home environment. People's interest about smart home services also reflects well during our demo sessions. For example, one user was interested in our service, and after our demo, he, by himself, put a smart key to another place to see if this service still worked (i.e., to see if our service could detect the new place of this key). After getting the right information again, he said that it was a good service, but he was surprised at how this was realized. We then talked him some basic knowledge about indoor positioning.

Users also raised several critical issues as well as suggestions about our system. One user stated that the smart home was very appealing, but the current implementation of it

seemed to be complex and the cost was high to average families, because users had to deploy lots of ultrasonic receivers on the ceiling, and equipped sensors to various everyday artifacts. He thought that a simple, low cost technical implementation would greatly help for the proliferation of smart homes. We agree with his comment strongly and we believe that, with the development of wireless sensor network technology, the deployment of sensors in a home environment will be much easier and the cost of it will be greatly reduced as well. Another user (an IT engineer) suggested that to deliver “anytime, anywhere” services, more natural interaction media, while not computer screens, should be explored to transmit information to users. For example, in the real-world search service, physical interfaces, e.g., a movable spotlight on the ceiling, can be used to convey the target object’s location information to a user (i.e., the spotlight can move to the target object and project a beam of light on it).

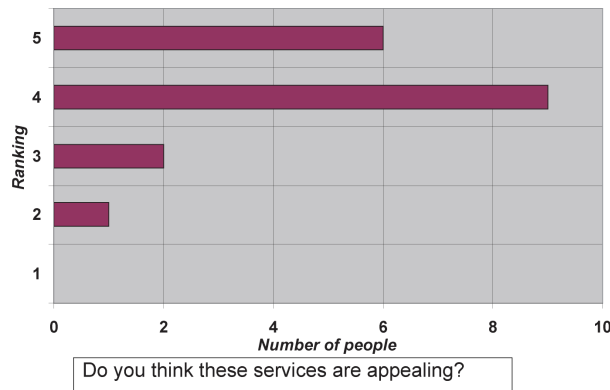


Figure 6.13: User feedback about real-world search service

6.5.2 User Experience of the Treasure Game

To investigate the prospects of artifact-based entertainments, we asked some users to participate in a predefined Treasure game (see Section 4.5.1) in our experimental gaming environment. To make it not too difficult for players to find objects, we restricted the game space to a 1.5×1.5 m area, as shown in left of Fig. 6.11. The predefined game follows the settings we depict in bottom of Fig. 4.11. Besides the four selected “hidden” objects, there were several other objects within this game field, such as a chair, a 3-tire shelf, a cup and several books, which are good places for hiding other objects. At the beginning of a testing, all the four smart artifacts were hid by our team-members in undetectable places. Subjects were asked to place the object he found on the table to ensure that it could be detected by our system.

6.5 User Experiences

On the whole, most testers successfully completed this game. However, there were also some failure situations. For example, one tester didn't place the room-key's U3D tag upright on the table, which makes it still unable to be detected by our location system. Another source of failure is caused by the ultrasonic-signal interference between our positioning system and the directional speaker, which sometimes results in error readings to the U3D location system. In this way, a hidden object may be wrongly asserted to have been found by a player. Failures like this happened twice during all the experiments.

After an experience of the game, subjects were asked about their feelings about this game. The overall impression of the game was crucial because we wanted to evaluate if people like playing artifact-based games. The results shown in left and middle of Fig. 6.14 indicate that about 80% testers were excited about this game and more than 90% of them were willing to play such games at home. Another of our aims was to evaluate the immersive factors of this game. Fig. 6.14 (right) illustrates how testers judged the mixed-reality aspects of the game. It appears that most players felt that our system provided them with an adequate and well-balanced mixture of the physical and virtual elements.

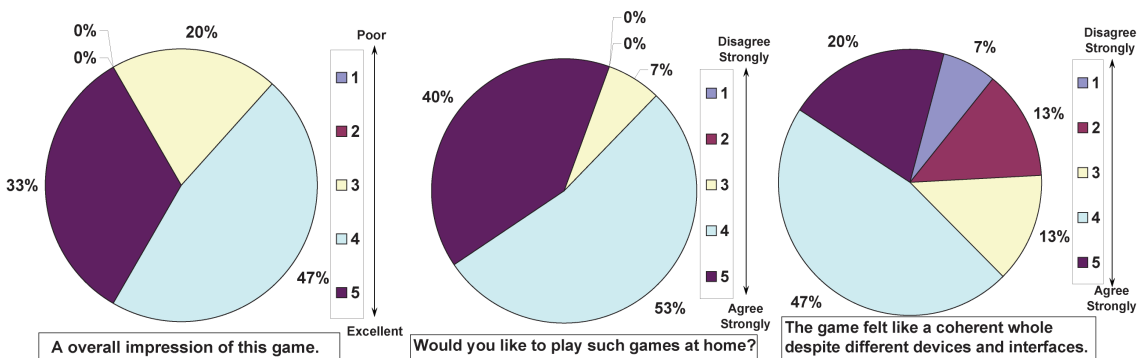


Figure 6.14: Testing results for user experience of a game

Some issues were also raised by the testers after the game session. For example, two testers were worried about our projection system, and they mentioned that the images projected on the wall would be obscure in a very bright room. Another tester stated that our Prot system might suffer from the occlusion problem when there were obstacles on the projection path. As the mean response time to players is about 1.5 seconds (latency caused by positioning and context reasoning), one player suggested that, before real actions were performed, there should be a “*Seeking...*” like message displayed to players to inform them that the objects they found had been detected by our system.

6.6 Interviews and Findings

Interviews and further discussions were made after each testing session, during which we asked subjects about their feelings on the concept of our system and their suggestions on how to improve it. Several findings were obtained during the interview sessions.

(1) Most users believed that there would be a market for our system. During the interviews, most subjects' feedback was positive. For example, after participating in the Treasure game, at least five subjects mentioned the commercially successful product — Wii, which also integrates physical activity into games. However, they meanwhile stated that our system would be more popular, because it enables users to play games in an anywhere, anytime fashion and more importantly, allows end-user programming of games, which is not supported in Wii. Testers also described several possible situations for using our system, for example, *“On my daughter’s birthday, I can hide the birthday present and ask her to find it out”*, *“Such games can bring us much funny at family parties”*. When asking subjects about the commercialization possibility of our system, many of them expressed their certainty and confidence about it, *“If the operations are simple enough, this system will be prevalent among users”*, *“The customization process can bring my family much funny”*. As smart homes are still not part of our daily life, one subject suggested that our system could be firstly installed and commercially used in specially-designed game halls. The potential of using our system as a tool for users to learn to program and a means to increase their knowledge about smart homes was proved by most testers, such as *“Learning to program by creating games is very interesting”*, *“It can help me to learn new techniques”*, etc.

(2) Not only advanced users, but also third parties could be good application creators. One tester pointed that, besides advanced users, participation of third parties, such as sensor producers, smart artifact makers and service providers, could greatly enrich the quantity and quality of smart artifact applications and games. This is a commercially sound advice, considering that third parties can also adverse their products to home users utilizing our platform.

(3) There is a need to include more graphical elements in the customization model. Many criticisms about our *Open-Programming* model came from our literal-based front-end design for novice users, which is thought to be unattractive to people. As a user-

6.6 Interviews and Findings

oriented platform, they considered that its interface should be designed in a more natural, intuitive and amusing way. Some suggestions on how to improve it were also offered. For example, one subject recommended us to visit Lively (its information is available at “<http://www.lively.com/>”), a second life platform recently developed by Google, where users can design and customize their room and avatar, and virtually interact with others on the Web. He stated that if our system could also introduce a virtual house for users to customize applications and games, it will attract much more users. This seems to be a good idea, which could be one way to improve our interface design. Moreover, we consider that, comparing with using the Protégé-OWL editor, a virtual-house-based tool also suggests a new way to help users intuitively customize the ontology for their house.

(4) Real-time response to players is important to smart artifact applications. According to the user feedback mentioned in Section 6.5.2 and the comments we gathered during the interview sessions, a user might be confused or frustrated if a smart artifact application could not respond to his actions in time. “*When there was no change of displayed images after I performed a task, I didn’t know whether there was something wrong with my actions or with the system*”. This will further decrease users’ enthusiasm to smart artifact applications. There are two possible ways to improve this. First, we can improve our context-reasoning mechanism to shorten the required response time. Second, making a compromise between “long” response time and real-time requirement, for example, we can display a “*Seeking...*” like message (a transitional response between two reaction outputs) once a user-action is detected, following the suggestions from a subject (see Section 6.5.2).

(5) A third programming mode is needed. Some testers expressed their desires to introduce graphical elements into rule creation, for instance, “*It is complex to write text-based rules. I think a graphical-connection method is better*”, “*Can I create rules using a wizard-like method*”, “*Is there a simple way to write rules without using variables*”. From these comments, we learned that there existed another requirement about programming in our system, that is, a method that has lower functionality, better operability than the text-based mode, and higher freedom than the customization mode, similar to graphical-based methods used in visual programming systems. A possible way to implement this is described in Section 7.4.3.

Chapter 7. Discussions

In this chapter, we will give a comparison of our system with related studies, and present several possible ways to improve this system.

7.1 Comparison with Related Studies

A survey of comparison of our work with other related studies (all have been introduced in Chapter 2) appears in Table 7.1. Different sensing techniques are used by them to acquire information from the physical world, including several different ways to locate physical entities (e.g., people, artifacts, and devices), they are, U3D, RFID, and pressure sensors for locating people and smart artifacts, and the Bluetooth and Ethernet technology for locating smart devices (e.g., mobile phones, faxes). Different locating technique has different merits and limitations, comparison of them lies outside the scope of this paper, detailed discussions can be found in Hightower *et al.* [01].

Table 7.1: Comparison with other related systems

<i>System name</i>	<i>Program- ming platform</i>	<i>Smart artifact system</i>	<i>Context infra- structure</i>	<i>End-user support</i>	<i>Inference engine</i>	<i>Real world search</i>	<i>Artifact- based games</i>	<i>Robust- ness of sensors</i>
Media-Cup	–	√	–	–	–	–	–	–
Smart Toolbox	–	√	–	–	–	–	–	–
Active Bats	–	√	–	–	–	–	–	–
MAX	–	√	Database	–	–	√	–	–
CoBrA	–	–	COBRA- ONT	Experts	Jess	–	–	–
Semantic Space	–	–	ULCO- ONT	Experts	Jena2	–	–	–
iCAP	Context- aware apps	–	Database	Advanced users	Ad hoc	–	–	–
Story-Room	Game creation	–	Database	Novice users	Ad hoc	–	–	–
Sixth-Sense	Games, smart object apps	√	SS-ONT	Experts, advanced, novices	SWRL, Jess	√	√	√

Besides *Sixth-Sense*, smart artifacts are also the focus of several other listed systems, however, all these systems are based on ad-hoc context infrastructures, which makes it difficult to share and reuse knowledge in them. Furthermore, the context reasoning tasks of them are implemented at the programming level, that's to say, they don't use any

7.1 Comparison with Related Studies

inference engines, which, inevitably, reduces their extendibility and increases their maintenance costs. Finally, among the listed smart artifact systems, the robustness of sensor network is only addressed in our system, as we clarified in Chapter 5.

CoBrA and Semantic Space are two existing systems that use a normalized ontology as their underlying knowledge infrastructure. These systems also provide effective context reasoning mechanisms by using inference engines. However, our system differs from and perhaps outperforms theirs in several respects. (1) These two systems are mainly designed for experts, and they don't provide any toolkit or interface for end users. (2) Their ontology definitions focus more on contexts acquired from mobile devices and software applications, while the domain knowledge of human-artifact interaction is not concerned in their systems. Therefore, they don't support rapid prototyping of smart artifact systems. (3) The techniques used in *Sixth-Sense* are more promising. Our work was conducted by the Protégé resource, including the Protégé-OWL editor for ontology editing, the Protégé-OWL API for manipulating the ontology at the programming level, and the SWRL factory mechanism to integrate with inference engines. That's to say, the Protégé team provides a free, mature and full-featured way for building and maintaining ontology-based systems. As the rule language of Semantic Web, SWRL supports rule interoperation (or sharing of rules) among different rule engines, which implies that our system can be extended to work with a variety of rule engines for different reasoning tasks (e.g., *forward-chaining*, *backward-chaining*, *non-monotonic*, etc). In contrast, this is not supported by the previous two studies. (4) Runtime performance was little discussed in previous work. However, we concluded several factors that could influence the performance of ontology-based context reasoning and proposed a computational model to estimate the maximum reasoning time under different scales of smart environments.

Over several user-oriented programming systems that supports context-aware service creation mentioned in Chapter 2 (e.g., Alfred, Jigsaw, CAMP and iCAP), we will make a comparison of our system with iCAP. First, our programming platform is founded on the *Open-Programming model*, which allows users with different abilities (e.g., advanced users and novices) to create applications in our system. However, iCAP's visual programming environment is still difficult and time-consuming for average home users to

use. Second, comparing with iCAP's database-scheme-based model, our ontology-based model is more normalized and promising for knowledge definition, as we discussed in Chapter 2. Third, benefited from our ontology-based context modeling method, our *Open-Programming* model supports the sharing of context-aware rules and services among different users, which is not supported by iCAP. Fourth, because all users share a common context structure, end users only need to perform a few simple operations to create instances and specify their properties through a mature, easy-to-use ontology editor. However, in iCAP, users have to design the context structure by themselves, such as drawing icons, specifying context categories and output types, which is difficult, error-prone and time-consuming for average users. Finally, there lacks a mechanism to detect hardware errors in iCAP, but we proposed a rule-based mechanism to deal with this.

StoryRoom is another system that supports user-oriented pervasive game design. However, our system differs from it in the following respects. First, utilizing easily-found resources in smart homes, such as everyday artifacts and user-generated contents, users of different ages can develop a wide variety of games in terms of their imagination in our system. However, StoryRoom is designed for a specific game genre — storytelling, and the props used in it are needed to be manually made by users, which somewhat restricts its prevalence. Second, our *Open-Programming* model addresses different user abilities and interests. In contrast, StoryRoom is designed for children, and the simplified programming mode it explores prohibits the creativity of many users. Third, we provide an ontology-based sharing and cooperation environment for end users, where a pervasive game created by one user can be shared and reprogrammed by others. Compared to previous systems' unshared modes, our approach greatly facilitates the development of games, and furthermore provides a stepwise learning environment for programming, that is, learning from *merely playing*, *simple configuration* to *rule-based programming*.

7.2 Open-Programming and OWL-S

OWL-S (Ontology Web Language for Services) [Martin *et al.* 04] is an effort of W3C that also seeks to construct and manipulate services based on Semantic Web techniques. By giving rich semantic specifications to Web services, OWL-S enables flexible automation of service discovery, invocation, and composition. A comparison of our

7.2 Open-Programming and OWL-S

Open-Programming model and OWL-S based studies (like [Mokhtar *et al.* 05]) is given in Fig. 7.1 and Table 7.2.

It's clear that these two approaches are different in their objectives. As a user-oriented programming platform, the *Open-Programming* model mainly concerns service sharing, end-user programming and service customization. However, OWL-S advocates placing less demand on user attention, and one of the objectives of it is to automatically enable software applications by dynamically composing services of the pervasive environment. In other words, *Open-Programming* addresses “controllability” and “personality”, while OWL-S emphasizes “high intelligence”, “autonomy” and “self-adaptation”.

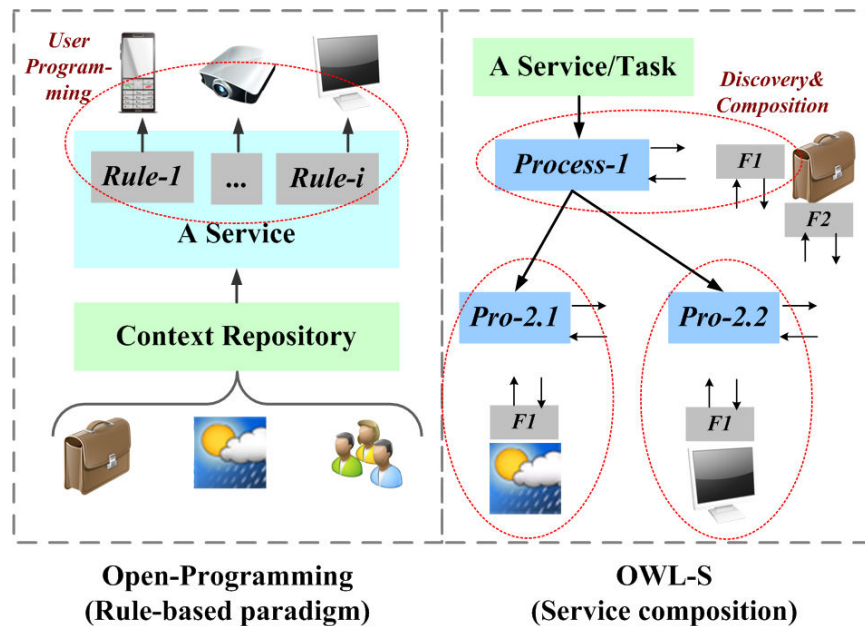


Figure 7.1: Service construction in Open-Programming and OWL-S

Table 7.2: Open-Programming and OWL-S

<i>Parameters</i>	<i>Open-Programming</i>	<i>OWL-S</i>
Purpose	Service sharing among different families, programming, and customization	Service discovery, autonomous composition, self-adaptation
Nature	Static services	Dynamic services
Semantics of a service	A smart home application	A task or a function of a device/software
Design Paradigm	Rule-based	Dynamical service composition
For End-Users	Programmable, personality, controllability	High intelligence, less user-effort

All services created in the *Open-Programming* model follow a rule-based paradigm, and the function of a service in our system is implemented by evaluating the predefined rules of it. That's to say, all running services (including user-customized ones) of our system are static services. OWL-S, by contrast, supports dynamical composition of

services. As shown in right part of Fig. 7.1, by using OWL-S, each device or software application can be viewed as a service provider that uses Web ontologies to describe (or advertise) the offered and required capabilities (or functions) of it. The description of the behavior of a user task or a service can be represented in the form of an OWL-S process model, which consists of a set of atomic processes (see Fig. 7.1). A discovery algorithm based on OWL-S can compare semantically the atomic processes of a user task with networked services (e.g., device-services), and a service composition algorithm can automatically integrate the processes of the selected services to reconstruct the process of the target user task [Mokhtar *et al.* 05]. In summary, different design paradigms of *Open-Programming* and OWL-S lead to different application fields of them. For example, the prior one can be better used in applications that concern personality and customizability, while the latter one can be explored by applications that address automated service discovery and composition. As a middle-term plan of our system, we intend to combine the advantages of these two approaches to provide high-level intelligent services for users who have little interest to control their smart homes.

7.3 Evolution of Sixth-Sense

In our work, we employed several sensing and interaction technologies to establish a prototypical smart environment. To popularize the use of *Sixth-Sense* among people, one basic hypothesis is that different smart homes are established using the same standards (or protocols) and sensing technologies. Today, smart home is far from having become a reality. However, given the trends of wireless sensor networking, low-power computing and tangible interaction, we can reasonably expect that sensors and interactive interfaces in the near future will be ubiquitously deployed in our living environment. Mature or emerging standards and products such as X10 [Helal *et al.* 05], OSGi [Gu *et al.* 04, Helal *et al.* 05], FIPA [O'Brien *et al.* 98], and RFID [Yap *et al.* 05, Gu *et al.* 04, Helal *et al.* 05], are speeding up the growth of it. Therefore, we can envision that, in future homes, artifacts will be equipped with unified sensing and interacting parts and can communicate with each other using standardized protocols. Benefiting from the *openness*, *extendibility* and *interoperability* of our ontology-based model, *Sixth-Sense* can easily be extended to reflect the contexts acquired from newly-applied technical objects, and provide interfaces

7.4 Potential Improvement Areas

to communicate with these devices. In other words, our system possesses the capability to evolve as a new technology emerges or as an application domain matures.

7.4 Potential Improvement Areas

There are several potential areas to improve our system, and we describe some of them in the following subsections.

7.4.1 User Ontology Customization

The evaluation results on ontology customization (see Section 6.3) indicate that experienced computer users can customize an ontology for their home via the Protégé-OWL editor. However, when the scale of objects and their properties enlarges, it will become a time-consuming task for users. Moreover, for users with little computer experiences, it is still difficult for them to master this tool. In light of these flaws, we are seeking for a more natural and labor-saving way for users to customize ontologies.

One way that inspires us is the barcode-based system, including traditional barcode systems prevalently used in supermarkets and 2D-barcodes (e.g., QR Codes [Kato *et al.* 05]) explored in mobile tagging systems. Taking QR-Codes for example, one such code can store at a maximum of 2953 bytes or 1817 Japanese characters (Kanji). QR Codes storing addresses and URLs may appear in magazines, buses, business cards or just about any object that users might need information about. Users with a camera phone equipped with the correct reader software can scan the image of the QR Code causing the phone's browser to launch and redirect to the programmed URL. This act of linking from physical world objects can be viewed as “*physical world hyperlinks*”.

Owing to the “*high data capacity*”, “*simplicity*” and “*physical interactivity*” merits mentioned above, we envision that a mobile 2D-barcode scanning system integrated into the *Open-Programming* model can help users quickly customize ontologies. Comparing with ontology developers and home users, the manufactures have the most intimate knowledge about their products. Most basic information about an artifact, such as its category, color, size, shape, weight, as well as performance parameters of its embedded sensors, can be encoded as barcodes when it is produced. End users can simply use a barcode-reader-equipped mobile phone to acquire the digital information of a barcode-attached object. We can also develop a client-side API to interpret the gathered object

information and automatically insert them into a user ontology utilizing Protégé- OWL API. In this way, most of the indoor objects and their properties can be automatically imported into the knowledge base, and there will be only a few properties that can not be specified by manufactures, such as the ownership of an object, privacy level of an object and human profiles, should be manually inputted by end users.

We believe that the mobile barcode-based mechanism is much simpler and can greatly ease users' burdens, and we plan to explore it in our next-step's work. However, under current technical conditions, our Protégé-OWL editor based method is still a normalized and user-friendly way for user-oriented ontology customization.

7.4.2 Artifact-based Human Behavior Recognition

Inferring human behaviors or activities using sensor networks is a crucial yet difficult application area for ubiquitous computing. In principle, current research on monitoring human behavior can be grouped into two sets: recognizing human behaviors that do not involve interactions with objects (e.g., cups, books) [Bao *et al.* 04], and recognizing those that do [Beigl *et al.* 01, Philipose *et al.* 04]. Numerous studies have performed high-level inferencing from low-level, coarse sensor data. However, progress towards rigorous activity detection has been slow, and only a few researchers have reported the results of any preliminary user testing. An emerging consensus is that fine-grained measurement of object use is a good indicator of human activities [Beigl *et al.* 01, Philipose *et al.* 04]. In view of this, *Sixth-Sense* infrastructure would be useful for supporting object-based activity inferencing. In prototyping studies of our system, we use ultrasonic location sensors, and a series of heuristic rules to infer simple human behaviors, such as *picking up something* and *moving something*. This is just a test bed to evaluate the approaches taken in our research. As one of our short-term objectives, we would like to integrate other kinds of sensors such as acceleration sensors [Bao *et al.* 04], and extend our domain ontology to support recognition of more artifact-based human behaviors.

7.4.3 A Third-Programming Mode

We learned from our evaluation that a third programming mode, which has lower functionality, better operability than the text-based mode, and higher freedom than the reprogramming mode, is required by users (as discussed in Section 6.6). Figure 7.2

7.4 Potential Improvement Areas

illustrates a possible way for this mode, which integrates the physical, intuitive nature of the physical programming mode and the code-free merit of the visual programming mode. In this mode, when a user is going to create a rule, he can firstly demonstrate the situation as this rule describes. For example, to design a rule that can give a hint when a person is carrying a key near his bed, he can firstly perform this action in his room (as shown in Fig. 7.2). At the same time, all detected real-time events and facts will be displayed on the screen of his handheld device (see bottom of Fig. 7.2). The user can then select the necessary premises from the listed events and also specify an action (e.g., displaying an image on TV) to perform. Once this is finished, a new rule is created.

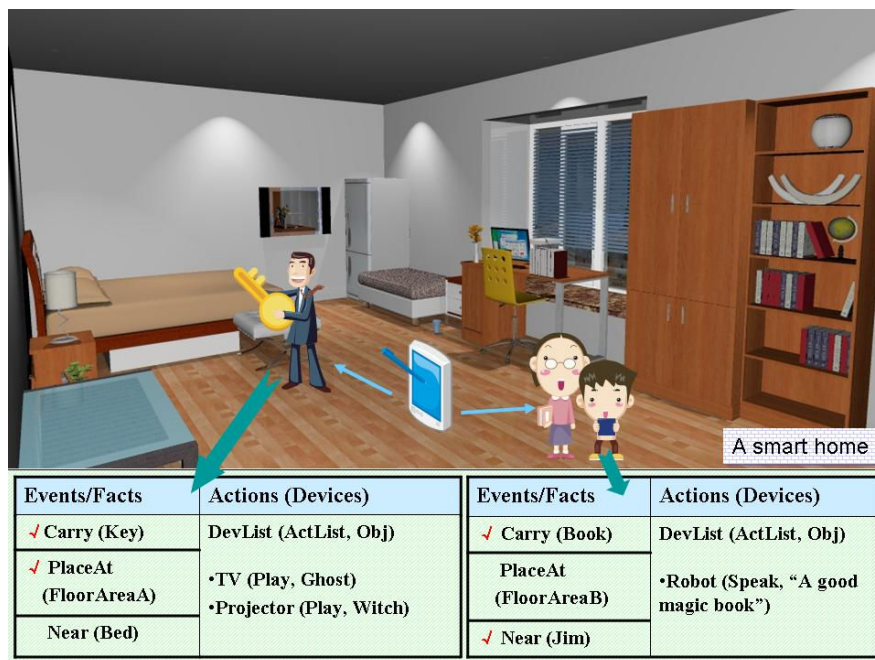


Figure 7.2: A scenario of the third programming mode

Chapter 8. Conclusions and Future Work

In this chapter, we conclude this thesis by (1) summarizing its main contributions and (2) suggesting directions for future work.

8.1 Summary

Computing is moving towards pervasive, context-aware everyday artifacts. In this thesis, we have reported our study on building a user-oriented, ontology-based programming platform for smart artifact systems.

Chapter 1 outlined the research area and main challenges in smart artifact research. Current approaches, related studies were reviewed in Chapter 2. Related work involves context-aware computing, knowledge representation, and end-user programming in smart homes. We also discussed the shortcomings in current studies in this chapter, which then lead to the motivations for the research described in this thesis. The following requirements were identified:

- A common knowledge infrastructure that supports rapid prototyping of human-artifact interaction systems is required;
- Diversity of user abilities and interest should be considered in our programming platform;
- As a programming platform, there should be strategies to deal with programming errors and fallible sensors in smart artifact applications.

These areas provide the focus of later chapters as well as our major contributions.

In Chapter 3, we described our efforts to incorporate Semantic Web technologies into the development of human-artifact interaction systems. By providing explicit ontology representation, expressive context querying, and flexible reasoning mechanisms, the *Sixth-Sense* infrastructure facilitates the development of a wide variety of smart artifact applications.

Based on the infrastructure, in Chapter 4, we described our effort on the programming environment for end users. By exploring an ontology-based programming model, called *Open Programming*, our system can support both the “*functionality*”, “*in-depth*” requirement from advanced users and the “*simplicity*”, “*high intelligence*” requirement

8.2 Future Work

from average home users. We described in detail the implementation of this programming model, including service creation, front-end design, service customization and action setting. The enabled applications, including an artifact-based game and an object search service are described to exemplify the usability of our system.

The error-checking module of *Sixth-Sense* is discussed in Chapter 5, where we described a rule-based mechanism to deal with fallible sensors in smart homes. In this approach, physical relations and logical relations among objects are explored to detect and identify hidden objects.

We also reported the experiments to evaluate the usability, performance and effectiveness of *Sixth-Sense* in Chapter 6. The evaluation study results suggest that our system can help users with different abilities to create smart artifact based context-aware services and games in our system.

Besides the two main contributions, namely a knowledge infrastructure for smart artifact systems and the *Open-Programming* model, there are also several minor contributions of our work:

- We proposed a physical-relation based approach to deal with fallible sensors in smart artifact applications;
- A rear-world search service is developed, which can help users quickly locate his belongings and monitor hidden objects at home;
- We made a series of user studies on smart home control and ontology use;
- Finally, our system investigates the prospects of artifact-based entertainment design and play in future homes.

8.2 Future Work

The area of smart artifact systems is a broad one and still largely undeveloped. This thesis has investigated some issues that may occur when living with smart artifacts in future homes. It is envisaged that some appropriate areas for future work are:

- *Ontology Storage*

In current implementation, the *SS-ONT* ontology is stored as OWL files. Operations related to this ontology, such as query and update, are implemented by loading, updating, and saving these files, which reduces the execution efficiency to

lower than that achieved using traditional databases. We therefore need to investigate more efficient methods of OWL storage and retrieval, such as Sesame [Broekstra *et al.* 02] and DLDB [Pan *et al.* 03].

- *Ontology Customization*

We plan to build a mobile 2D-barcode scanning system, which can help users quickly customize ontologies, as we discussed in Section 7.4.1.

- *Rule Editor in Open-Programming Model*

In the rule-based programming mode, users write Jess rules in a textual manner, which is unfriendly and error-prone. Therefore, we intend to develop a better Web-based Jess editor, where users can directly refer to OWL classes, properties, and individuals within an OWL knowledge base, and also have direct access to the full set of Jess functions. We believe that the improved editor will be more intuitive to users.

- *A Virtual House Interface*

As mentioned in Section 6.6, the literal-based front-end is stated to be unattractive to end users, so we want to design a virtual-house-like interface for users to intuitively customize services and games. We believe that this will attract more users to the reprogramming process.

- *A Third Programming Mode*

As discussed in Section 7.4.3, we intend to develop a third programming mode which can integrate the physical, intuitive nature of the physical programming mode and the code-free merit of the visual programming mode.

- *Long-Term Evaluation*

Finally, we plan to conduct a long-term evaluation of our system's effectiveness as an end-user programming tool in real-life house settings, during which we want to learn the challenges in terms of different domestic settings, diverse lifestyles and needs of users. We expect this can, in addition, help us to better understand the effect of *Open-Programming* model on user learning and collaboration.

Acknowledgements

This dissertation marks the end of my journey as a student at Keio. My accomplishment over the past three years is indebted to many people. Especially, I want to thank Dr. Michita Imai, for giving me the opportunity to work in his research lab at Keio. Dr. Imai is a great advisor and a fantastic friend, and in my mind, he is very nice and always with kind smiles. It was with him that I first learned how to do research and how to write scientific papers. His favorite quote “*a good study should firstly have a good ‘story’ to attract others*” encouraged me to think outside the box when I do research. Thank you, Prof. Imai, I am very grateful.

I am also deeply indebted to Prof. Yuichiro Anzai. As the president of Keio, he is very busy and has little time to spend with us. However, even during his busy schedules, he also cares about my research themes and future plans. Many thanks should also be sent to Dr. Ren Omura and Dr. Hideyuki Kawashima. Dr. Omura gives me extremely valuable remarks regarding the style and scientific contributions of this thesis. Dr. Kawashima helps me a lot on my life, study and school enrollment when I started my life in Japan.

I owe a great debt to my student-advisor, Satoru Satake, from whom I have picked up some knowledge on sensor networks, for his patience with all my foreigner’s questions and effort on guiding my research and reviewing my papers. Thanks Satake-San, I really appreciate for your kindly help. I also extend my thanks to all my colleagues and group members at Keio. Many ideas that resulted in the research presented in this thesis had their origins in discussions. In particular I appreciate Takeshi Kenda for his ideas on hidden object identification and thank Ryota Fujimura for his collaboration in the iFun game project. Many other members had helped me a lot on the life and research, to name a few: Hiroataka Osawa, Kentaro Ishii, Jun Mukai, Hitoshi Kawasaki, Yusuke Okuno, Manabu Nakamura, Yasuhide Shinagawa, Toshiyuki Shiwa, Yukiko Yamamoto and Yanagisawa Yoji. A sincere thank to all of you.

Many thanks should also been sent to the financial supporters of this study, they are, Japanese Government MEXT Scholarship, 21st century COE Program of Keio University, The Research Grant of Keio Leading-edge Laboratory of Science & Technology.

I would like to express my deepest gratitude to my dear wife Shuying Zhai, who made many compromises to let me finish my Ph.D. in Japan. She is the best cheerleader that I ever had. Whenever I was beaten by the stressful paper deadlines or research obstacles, Shuying was always there to refuel my energy. Without your friendship and love this thesis would not have been completed. Thank you, Shuying, for loving me and supporting me. I would also like to thank my dear daughter Sinyi, for the encouragement and happiness she has given to me over the last year.

Finally, my parents have always been a constant source of understanding and never-ending moral support. I know I can never repay them, but I would like to seize the opportunity to thank my family for all it has given to me.

Thanks to all of you.

2008.12

Bin Guo

References

- [Acancha *et al.* 04] S. Avancha, C. Patel, and A. Joshi. Ontology-driven Adaptive Sensor Networks. In *Proc of MobiQuitous'04*, pp. 194-202, 2004.
- [Alborzi *et al.* 00] H. Alborzi *et al.* Designing StoryRooms: interactive storytelling spaces for children. In *Proc. of the 3rd conference on designing interactive systems*, New York, USA, pp.95-104, 2000.
- [Antoniou *et al.* 03] G. Antoniou, and G. Wagner. Rules and defeasible reasoning on the semantic web. In *Proc. of RuleML Workshop 2003*, pp. 111-120, 2003.
- [Bao *et al.* 04] L. Bao, and S. Intille. Activity recognition from user-annotated acceleration data. In *Proc. of Pervasive'04*, Heidelberg, Germany, pp. 1-17, 2004.
- [Barkhuus *et al.* 03] L. Barkhuus, and A. Dey. Is Context-Aware Computing taking Control away from the User? Three levels of Interactivity Examined. In *Proc. of Ubicomp 03*, USA, pp. 149-156, 2003.
- [Beer *et al.* 03] W. Beer, V. Christian, A. Ferscha, and L. Mehrmann. Modeling Context-aware Behavior by Interpreted ECA Rules. In *Proc. of Euro-Par 2003*, pp. 1064-1073, 2003.
- [Beigl *et al.* 01] M. Beigl, H.W. Gellersen, and A. Schmidt. Mediacups: Experience with design and use of computer-augmented everyday objects. *Computer Networks*, Vol. 35, No. 4, pp. 401-409, 2001.
- [Beigl *et al.* 03] M. Beigl, and H. Gellersen. Smart-Its: An embedded platform for smart objects. In *Proc. of the Smart Objects Conference (SOC 2003)*, Grenoble, France, 2003.
- [Benford *et al.* 03] S. Benford *et al.* Coping with uncertainty in a location-based game. *IEEE Pervasive Computing*, Vol. 2, No. 3, pp. 34- 41, 2003.
- [Benford *et al.* 06] S. Benford *et al.* Can you see me now? *ACM Transactions on Computer-Human Interaction*, Vol. 13, No. 1, pp. 100-133, 2006.
- [Berners-Lee *et al.* 01] T. Berners-Lee, J. Hendler, and O. Lassila. The Semantic Web. *Scientific American*, Vol. 284, No. 5, pp. 34-43, 2001.
- [Blackwell *et al.* 01] A.F. Blackwell, and R. Hague. AutoHAN: An architecture for programming the home. In *Proc. of the IEEE Symposium on Human-Centric Computing Languages and Environments*, pp. 150–157, 2001.
- [Bobick *et al.* 99] A.F. Bobick *et al.* The KidsRoom: a perceptually-based interactive and immersive story environment. *Presence*, Vol. 8, No. 4, pp.369-393, 1999.
- [Brickley *et al.* 00] D. Brickley, and R. Guha. *Resource Description Framework (RDF) schema specification*. <http://www.w3.org/TR/RDF-schema>, 2000.
- [Broekstra *et al.* 02] J. Broekstra, A. Kampman, and F. Harmelen. Sesame: A generic architecture for storing and querying RDF and RDF schema. In *Proc. of the 1st International Semantic Web Conference*, Sardinia, Italy, pp. 54-68, 2002.

- [Brumitt *et al.* 00] B. Brumitt, B. Meyers, J. Krumm, A. Kern, and S. Shafer. EasyLiving: Technologies for intelligent environments. In *Proc. of the Second International Symposium on Handheld and Ubiquitous Computing*, Bristol, UK, pp. 97-119, 2000.
- [Chen *et al.* 04a] H. Chen, T. Finin, A. Joshi, F. Perich, D. Chakraborty, and L. Kagal. Intelligent agents meet the semantic web in smart spaces. *IEEE Internet Computing*, Vol.19, No. 5, pp. 69–79, 2004.
- [Chen *et al.* 04b] H. Chen, F. Perich, T. Finin, and A. Joshi. SOUPA: Standard ontology for ubiquitous and pervasive applications. In *Proc. of the MobiQuitous'04*, 2004.
- [Cheok *et al.* 02] A.D. Cheok, X. Yang, Z.Z. Ying, M. Billingham, and H. Kato. Touch-Space: Mixed reality game space based on ubiquitous, tangible, and social computing. *Personal and Ubiquitous Computing*, Vol. 6, No. 6, pp. 430-442, 2002.
- [Cheok *et al.* 04] A.D. Cheok *et al.* Human Pacman: a mobile, wide-area entertainment system based on physical, social, and ubiquitous computing. *Personal and Ubiquitous Computing*, Vol. 8, No.2, pp. 71-81, 2004.
- [Davidoff *et al.* 06] S. Davidoff, M.K. Lee, C. Yiu, J. Zimmerman, and A.K. Dey. Principles of smart home control. In *Proc. of Ubicomp'06*, pp. 19-34, 2006.
- [Dey *et al.* 00] A.K. Dey, and G.D. Abowd. Towards a better understanding of context and context-awareness. In *Proc. of CHI'2000*, 2000.
- [Dey *et al.* 06] A.K. Dey, T. Sohn, S. Streng, and J. Kodama. iCAP: Interactive Prototyping of Context-Aware Applications. In *Proc. of Pervasive'06*, pp. 254-271, 2006.
- [Edwards *et al.* 01] W.K. Edwards, and R.E. Grinter. At Home with Ubiquitous Computing: Seven Challenges. In *Proc. of Ubicomp'01*, pp. 256-272, 2001.
- [Eiter *et al.* 04] T. Eiter, T. Lukasiewicz, R. Schindlauer, and H. Tompits. Combining answer set programming with description logics for the semantic web. In *Proc. of KR-2004*, 2004.
- [Fontijn *et al.* 05] W. Fontijn, and P. Mendels. StoryToy the interactive storytelling toy. In *Proc. of PerGames'05*, Munich, 2005.
- [Friedman-Hill 07] E. Friedman-Hill. *Jess, the Rule Engine for the Java Platform*. <http://www.jessrules.com/jess/index.shtml>, 2007.
- [Gajos *et al.* 02] K. Gajos, H. Fox, and H. Shrobe. End user empowerment in human centered pervasive computing. In *Proc. of Pervasive'02*, pp. 134-140, 2002.
- [Gu *et al.* 04] T. Gu, H.K. Pung, and D.Q. Zhang. Toward an OSGi-based infrastructure for context-aware applications. *IEEE Pervasive Computing*, Vol. 3, No. 4, pp. 66-74, 2004.
- [Haarslev *et al.* 01] V. Haarslev, and R. Moller. Racer system description. In *Proc. of the International Joint Conference on Automated Reasoning*, pp. 701-705, 2001.

- [Harter *et al.* 05] A. Harter *et al.* The Anatomy of a Context-Aware Application. In *Proc. of 5th ACM International Conference on Mobile Computing and Networking*, pp. 59-68, 2005.
- [Helal *et al.* 05] S. Helal *et al.* The Gator Tech smart house: a programmable pervasive space. *Computer*, Vol. 38, No.3, pp. 50-60, 2005.
- [Henricksen *et al.* 02] K. Henricksen *et al.* Modeling Context Information in Pervasive Computing Systems. In *Proc. of Pervasive'02*, 2002.
- [Hightower *et al.* 01] J. Hightower, and G. Borriello. Location Systems for Ubiquitous Computing. *Computer*, Vol. 34, No. 8, pp. 57-66, 2001.
- [Hinske *et al.* 07] S. Hinske, M. Lampe, C. Magerkurth, and C. Röcker. *Classifying Pervasive Games: On Pervasive Computing and Mixed Reality, Concepts and technologies for Pervasive Games – A Reader for Pervasive Gaming Research*. Shaker Verlag, Germany, 2007.
- [Horrocks *et al.* 04] I. Horrocks, P.F. Patel-Schneider, H. Boley, S. Tabet, B. Grosz, and M. Dean. *SWRL: A Semantic Web Rule Language Combining OWL and RuleML*. <http://www.daml.org/2004/04/swrl>, 2004.
- [Hoysniemi 06] J. Hoysniemi. Games, user interface and performing arts: International survey on the Dance Dance Revolution game. *ACM Computers in Entertainment*, Vol. 4, No. 2, 2006.
- [Humble *et al.* 03] J. Humble *et al.* Playing with your bits: user composition of ubiquitous domestic environments. In *Proc. of Ubicomp'03*, pp. 256-263, 2003.
- [Ishii *et al.* 07] K. Ishii, Y. Yamamoto, M. Imai, and K. Nakadai. A navigation system using ultrasonic directional speaker with rotating base. In *Proc. of HCI07*, pp.526-535, 2007.
- [Kato *et al.* 05] H. Kato, and K.T. Tan. 2D Barcodes for Mobile Phones. In *Proc. of 2nd International Conference on Mobile Technology, Applications and Systems*, 2005.
- [Kelleher *et al.* 05] C. Kelleher, and R. Pausch. Lowering the barriers to programming. *ACM Computing Surveys*, Vol. 37, No. 2, pp. 83-137, 2005.
- [Kidd *et al.* 99] C.D. Kidd *et al.* The Aware Home: A living laboratory for ubiquitous computing research. In *Proc. 2nd International Workshop on Cooperative Buildings*, Berlin, pp. 190-197, 1999.
- [Kindberg *et al.* 02] T. Kindberg *et al.* People, Places, Things: Web Presence for the Real World. *Mobile Networks and Applications*, Vol. 7, No. 5, pp. 365-376, 2002.
- [Knublauch *et al.* 05] H. Knublauch *et al.* The Protégé OWL Experience. In *Proc. of the Workshop on OWL Experiences and Directions 2005*, Galway, Ireland, 2005.
- [Lamon *et al.* 03] P. Lamon, A. Tapus, E. Glauser, N. Tomatis, and R. Siegwart. Environmental Modeling with Fingerprint sequences for topological global localization. In *Proc. of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp.3781-3786, 2003.
- [Lampe *et al.* 03] M. Lampe, and M. Strassner. The potential of RFID for moveable asset management. In *Proc. of UbiComp 2003*, Seattle, 2003.

- [Lewis *et al.* 06] M. Lewis, D. Cameron, S. Xie, and B. Arpinar. ES3N: A Semantic Approach to Data Management in Sensor Networks. In *Proc. of 5th International Semantic Web Conference (ISWC'06)*, 2006.
- [Loke 04] S.W. Loke. Logic programming for context-aware pervasive computing: Language support, characterizing situations, and integration with the Web. In *Proc. of the IEEE/WIC/ACM Conference on Web Intelligence (WI'04)*, 2004.
- [Magerkurth *et al.* 04] C. Magerkurth, M. Memisoglu, and T. Engelke. Towards the next generation of tabletop gaming experiences. In *Proc. of the Conference on Graphics Interface*, pp. 73-80, 2004.
- [Magerkurth *et al.* 05] C. Magerkurth, A.D. Cheok, R.L.Mandryk, and T. Nilsen. Pervasive games: bringing computer entertainment back to the real world. *Computers in Entertainment*, Vol. 3, No. 3, 2005.
- [Martin *et al.* 04] D. Martin *et al.* Bringing semantics to web services: The OWL-S approach. In *Proc. of the workshop on Semantic Web Services and Web Process Composition*, San Diego, CA, 2004.
- [Mattila *et al.* 06] J. Mattila, and A. Vääänen. UbiPlay: an interactive playground and visual programming tools for children. In *Proc. of the conference on interaction design and children*, pp. 129-136, 2006.
- [Meyer *et al.* 03] S. Meyer, and A. Rakotonirainy. A survey of research on context-aware homes. In *Proc. of the ACSW frontiers 2003*, Adelaide, Australia, pp. 159-168, 2003.
- [Mokhtar *et al.* 05] S.B. Mokhtar, D. Fournier, N. Georgantas, and V. Issarny. Context-aware service composition in pervasive computing environments. In *Proc. of the International Workshop on Rapid Integration of Software Engineering Techniques (RISE'05)*, 2005.
- [Montemayor *et al.* 04] J. Montemayor, A. Druin, G. Chipman, A. Farber, and M.L. Guha. Tools for children to create physical interactive storyrooms. *Computers in Entertainment*, Vol.2, No. 1, 2004.
- [Nakadai *et al.* 05] K. Nakadai, and H. Tsujino. Towards new human-humanoid communication: listening during speaking by using ultrasonic directional speaker. In *Proc. of Robotics and Automation*, 2005.
- [Nishida *et al.* 03] Y. Nishida *et al.* 3D ultrasonic tagging system for observing human activity. In *Proc. of IEEE International Conference on Intelligent Robots and Systems*, pp. 785-701, 2003.
- [O'Brien *et al.* 98] P.D. O'Brien, and R.C. Nicol. FIPA: Towards a standard for software agents. *BT Technology Journal*, Vol. 16, No. 3, pp. 51-59, 1998.
- [O'Connor *et al.* 05] M.J. O'Connor, H. Knublauch, and S.W. Tu. Supporting rule system interoperability on the semantic web with SWRL. In *Proc. of the 4th International Semantic Web Conference*, 2005.
- [Pan *et al.* 03] Z.X. Pan, and J. Heflin. DLDB: extending relational database to support semantic Web queries. In *Proc. of the 1st PASS Conference*, pp. 43-48, 2003.
- [Philipose *et al.* 04] M. Philipose, K. Fishkin, M. Perkowitz, D. Patterson, H. Kautz, and D. Hahnel. Inferring activities from interactions with objects. *IEEE Pervasive Computing*, Vol. 3, No. 4, pp. 50-57, 2004.
- [Pollock 87] J.L. Pollock. Defeasible reasoning. *Cognitive Science*, Vol. 11, No. 4, pp. 481-518, 1987.

[Prud'hommeaux *et al.* 06] E. Prud'hommeaux, and A. Seaborne. *SPARQL Query Language for RDF*. <http://www.w3.org/TR/rdf-sparql-query/>, 2006.

[Ranganathan *et al.* 03] A. Ranganathan, and R.H. Campbell. A Middleware for Context-Aware Agents in Ubiquitous Computing Environments. In *Proc. of ACM/IFIP/USENIX International Middleware Conference*, Brazil, 2003.

[Ranganathan *et al.* 04] A. Ranganathan, J. Al-Muhtadi, and R.H. Campbell. Reasoning about Uncertain Contexts in Pervasive Computing Environments. *IEEE Pervasive Computing*, Vol.3, No.2, pp. 62-70, 2004.

[Salber *et al.* 99] D. Salber, A.K. Dey, and G.D. Abowd. The Context Toolkit: Aiding the development of context-enabled applications. In *Proc. of CHI'99*, pp. 434-441, 1999.

[Siio *et al.* 03] I. Siio, J. Rowan, N. Mima, and E. Mynatt. Digital Decor: Augmented everyday things. In *Proc. of the Canadian annual conference Graphics Interface*, 2003.

[Smith *et al.* 03] M. Smith, C. Welty, and D. McGuinness. *Web ontology language (OWL) guide Version 1*. <http://www.w3.org/TR/owl-guide>, 2003.

[Strang *et al.* 04] T. Strang, and L. Linnhoff-Popien. A Context Modeling Survey. In *Proc. of the First International Workshop on Advanced Context Modelling, Reasoning and Management*, pp. 33-40, 2004.

[Truong *et al.* 04] K.N. Truong, E.M. Huang, and G.D. Abowd. CAMP: A magnetic poetry interface for end user programming of capture applications for the home. In *Proc. of Ubicomp'04*, pp. 143-160, 2004.

[Uschold *et al.* 96] M. Uschold, and M. Gruninger. Ontologies: Principles, Methods and Applications. *The Knowledge Engineering Review*, Vol.11, No. 2, pp. 93-136, 1996.

[Wang *et al.* 04] X.H. Wang, D.Q. Zhang, J.S. Dong, C.Y. Chin, and S. Hettiarachchi. Semantic Space: An infrastructure for smart spaces. *IEEE Pervasive Computing*, Vol. 3, No. 3, pp.32-39, 2004.

[Want *et al.* 92] R. Want, A. Hopper, V. Falcao, and J. Gibbons. The active badge location system. *ACM Transactions on Information Systems*, Vol. 40, No. 1, pp. 91-102, 1992.

[Weiser 99] M. Weiser. The computer for the 21st century. *ACM SIGMOBILE Mobile Computing and Communications Review*, Vol. 3, No. 3, pp. 3-11, 1999.

[Yamada *et al.* 05] T. Yamada, T. Iijima, and T. Yamaguchi. Architecture for Cooperating Information Appliances Using Ontology. In *Proc. of 19th Annual Conference of the Japanese Society for Artificial Intelligence*, 2005

[Yap *et al.* 05] K. K. Yap, V. Srinivasan, and M. Motani. Max: Human-centric search of the physical world. In *Proc. of SenSys'05*, pp. 166-179, 2005.

[Yim *et al.* 07] J. Yim, and T.C.N. Graham. Using games to increase exercise motivation. In *Proc. of the 2007 conference on Future Play*, Toronto, Canada, pp.166-173, 2007.

Publication List

[Articles on periodicals (related to thesis)]

1. Bin Guo, Satoru Satake, Michita Imai, Lowering the barriers to participation in the development of human-artifact interaction systems, *International Journal of Semantic Computing (IJSC), Special Issue on Ambient Semantic Computing*, 2008 (in press).
2. Bin Guo, Satoru Satake, Michita Imai, Home-Explorer: Ontology-based Physical Artifact Search and Hidden Object Detection System, *International Journal of Mobile Information Systems*, Vol.4 No.2, 81-103, 2008.

[Other articles on periodicals]

3. Bin Guo, Haichang Gao, Boqing Feng, et al., Automatic Test Data Generation Based on Adaptive SAGA, *Microelectronics & Computer*, Vol.23 No.8, 10-13, 16, 2006 (in Chinese).
4. Yun Hou, Gang Gu, Haichang Gao, Bin Guo, An Improved Method of Automatic Generation for Path Coverage, *Computer Engineering*, Vol.33 No.4, 67-69, 2007 (in Chinese).
5. Haichang Gao, Boqing Feng, Li Zhu, Bin Guo, Improved adaptive GA application on automatic test data generation, *System Engineering and Electronics*, Vol.28 No.7, 1077-1081, 2006 (in Chinese).

[Articles on international conference proceedings]

6. Bin Guo, Michita Imai, Home-Explorer: Search, Localize and Manage the Physical Artifacts Indoors, In: *Proc. of IEEE 21st International Conference on Advanced Information Networking and Applications (AINA'07)*, Canada, 2007.
7. Bin Guo, Satoru Satake, Michita Imai, Sixth-Sense: Context Reasoning for Potential Objects Detection in Smart Sensor Rich Environment, In: *Proc. of the IEEE/WIC/ACM International Conference on Intelligent Agent Technology (IAT'06)*, Hong Kong, 2006.

[Others]

8. Bin Guo, Sixth-Sense: Life-based Management for Indoor Physical Artifacts, *COE Young Researcher Poster Session 2006* (Sponsored by the 21st COE Program at Keio University).
9. Li Chen, Bin Guo, Xiang Li, *Navigation of Database Projects with PowerBuilder 9*, Tsinghua University Press, Beijing, China, Jan 2005 (in Chinese).
10. Yingwei Li, Suxiang Yao, Li Jin, Bin Guo, Xiang Li, *Advanced Database Programming with Microsoft's ASP.NET(C#)*, Tsinghua University Press, Beijing, China, Jul 2004 (in Chinese).