UNIVERSITY OF CRETE
COMPUTER SCIENCE DEPARTMENT

Architectures, Methods and Tools for Creating Ambient Intelligence Environments

Ioannis Georgalis

Heraklion, June 2013

Architectures, Methods and Tools for Creating Ambient Intelligence Environments

Ioannis Georgalis
June 2013

University of Crete
Department of Computer Science

Thesis submitted in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

Doctoral Thesis Committee:
Constantine Stephanidis, Professor, University of Crete (Advisor)
Antonis A. Argyros, Associate Professor, University of Crete
Yannis Tzitzikas, Assistant Professor, University of Crete
Yuzuru Tanaka, Professor, Hokkaido University
Nicolas Spyratos, Professor Emeritus, University of Paris-South
Anthony Savidis, Associate Professor, University of Crete
Dimitris Grammenos, Principal Researcher, ICS FORTH

The work reported in this thesis has been conducted at the Human Computer Interaction (HCI) laboratory of the Institute of Computer Science (ICS) of the Foundation for Research and Technology – Hellas (FORTH), and has been financially supported by a FORTH-ICS scholarship. Additionally, this work has been supported by the FORTH-ICS internal RTD program "Ambient Intelligence Environments". Part of the work reported in this thesis has been conducted at the Meme Media Laboratory of Hokkaido University, Japan.

UNIVERSITY OF CRETE
DEPARTMENT OF COMPUTER SCIENCE

Architectures, Methods and Tools for Creating Ambient Intelligence Environments

Dissertation submitted by

Ioannis Georgalis

In partial fulfillment of the requirements for the
Doctor of Philosophy degree in Computer Science

Author:

Ioannis Georgalis

Examination Committee:

Constantine Stephanidis, Professor, University of Crete

Antonis A. Argyros, Associate Professor, University of Crete

Yannis Tzitzikas, Assistant Professor, University of Crete

Yuzuru Tanaka, Professor, Hokkaido University

Nicolas Spyratos, Professor Emeritus, University of Paris-South

Anthony Savidis, Associate Professor, University of Crete

Dimitris Grammenos, Principal Researcher, ICS FORTH

Department Approval:

Panagiotis Trahanias, Professor, Chairman of the Department

Heraklion, June 2013

Declaration of Originality

I herewith certify that all material in this dissertation which is not my own work has been properly acknowledged.

Acknowledgements

I owe many thanks to my thesis supervisor, Professor Constantine Stephanidis for his strong support and guidance through my Ph.D. studies. The ideas, guidance and support of Professor Yuzuru Tanaka were invaluable in order to shape this work to its current state. Professor Stephanidis, Professor Tanaka and Professor Nicolas Spyratos, helped me to improve my research skills with their close attention to my work and critical thinking. I am also grateful to my colleague Dr. Dimitris Grammenos for his feedback and brilliant ideas towards improving the quality of my work. My friend and colleague Nikolaos Kyriazis read early drafts of my thesis and provided many helpful comments and suggestions.

I would also like to thank Professor Anthony Savidis, Professor Antonis A. Argyros, and Professor Yannis Tzitzikas for their invaluable suggestions and comments, which also provided clear guidance towards exploring interesting future directions of my work.

Moreover, I want to thank all my friends and colleagues at the Institute of Computer Science of the Foundation for Research and Technology, Hellas (ICS-FORTH) and at the Meme Media Laboratory of Hokkaido University, Japan, who supported me and my work and helped in making the office environment stimulating, productive and pleasant.

Finally, I need to thank my parents, my sister and my friends for supporting and tolerating me all these years.

Publications Using the Proposed Work

- Georgalis, Y., Tanaka, Y., Spyratos, N., & Stephanidis, C. (2013). Programming Smart Object Federations for Simulating and Implementing Ambient Intelligence Scenarios. In C. Benavente-Peces and J. Filipethe (Eds.), Proceedings of the 3rd International Conference on Pervasive and Embedded Computing and Communication Systems (PECCS 2013), Barcelona, Spain, 19-21 February (pp. 5-15). Portugal: SciTePress. [CD, ISBN: 978-989-8565-43-3] [BEST STUDENT PAPER AWARD]
- Georgalis, Y., Grammenos, D., & Stephanidis, C. (2009). Middleware for Ambient Intelligence Environments: Reviewing Requirements and Communication Technologies. In C. Stephanidis, (Ed.), Universal Access in Human-Computer Interaction - Intelligent and Ubiquitous Interaction Environments. - Volume 6 of the Proceedings of the 13th International Conference on Human-Computer Interaction (HCI International 2009), San Diego, CA, USA, 19-24 July, pp. 168-177. Berlin Heidelberg: Lecture Notes in Computer Science Series of Springer (LNCS 5615, ISBN: 978-3-642-02709-3)
- Grammenos, D., Zabulis, X., Michel, D., Padeleris, P., Sarmis, T., Georgalis, G., Koutlemanis, P., Tzevanidis, K., Argyros, A., Sifakis, M., Adam-Veleni, P., & Stephanidis, C. (2012). Macedonia from Fragments to Pixels: A Permanent Exhibition of Interactive Systems at the Archaeological Museum of Thessaloniki. In M. Ioannides, D. Fritsch, J. Leissner, R. Davies, F. Remondino & R. Caffo (Eds.), Progress in Cultural Heritage Preservation, Proceedings of the 4th International Conference EuroMed 2012, Limassol, Cyprus, 29 October - 3 November (pp. 602-609). Berlin Heidelberg, Germany: Springer [LNCS: 7616]
- Grammenos, D., Georgalis, Y., Kazepis, N., Drossis, G., & Ftylitakis, N. (2010). The bootTable Experience: Iterative Design and Prototyping of an Alternative Interactive Tabletop. In the K. Halskov and M. Graves Petersen (Eds.), Proceedings of the 8th ACM Conference on Designing Interactive Systems (DIS 2010), 16-20 August 2010, Aarhous, Denmark. (pp. 272-281). New York: ACM Press

Abstract

Ambient Intelligence (AmI) is an emerging research field that aims to support and enhance the activities of everyday life. This new paradigm gives rise to opportunities for novel, natural and more effective interactions with computing systems. In this context, we propose a holistic framework for modeling and programming AmI environments. Through the proposed work, we aim (a) to define flexible and intuitive software abstractions for modeling all the different entities that comprise an AmI environment (architectures), (b) to suggest systematic, well-defined processes that identify and highlight all the necessary steps for developing AmI environments from the design to the deployment phase (methods), and (c) to provide programming libraries and tools that fully support and streamline the application of the defined architectures and methods (tools). The contribution of this work is twofold, as the focus is on the architectural and theoretical aspects of building AmI environments, but also on the practical issues that stem from the development process. In this regard, our approach is structured over the provision of services for exposing the resources of AmI environments through a unified homogeneous access layer, and the definition of smart objects that provide well-defined abstractions, dynamic interaction capabilities and extended composition potential for implementing AmI-related functionalities. The utilization of the concepts and tools proposed in this thesis improves existing practices (i) by maximizing the potential for incorporating heterogeneous technologies, (ii) by allowing for the comprehensive modeling of all aspects of functional AmI environments, (iii) by streamlining the whole implementation process, and (iv) by enabling the simulation of the target environment's behavior. Ultimately, these mechanisms allow for succinct implementations, minimizing the required effort for building, deploying and extending fully functional AmI environments.

Abstract in Greek

Η Διάχυτη Νοημοσύνη (ΔΝ) είναι ένας αναδυόμενος ερευνητικός τομέας που έχει ως στόχο να υποστηρίξει και να ενισχύσει τις δραστηριότητες της καθημερινής ζωής. Αυτό το νέο τεχνολογικό παράδειγμα δημιουργεί ευκαιρίες για νέες, φυσικές και αποτελεσματικές προσεγγίσεις στην αλληλεπίδραση με υπολογιστικά συστήματα. Σε αυτό το πλαίσιο, προτείνουμε μια ολοκληρωμένη πλατφόρμα για την μοντελοποίηση και τον προγραμματισμό περιβαλλόντων ΔΝ. Μέσω της συγκεκριμένης δουλειάς, ο στόχος μας είναι (α) να καθορίσουμε ευέλικτες δομές λογισμικού για την μοντελοποίηση των συνιστωσών που συνθέτουν ένα περιβάλλον ΔΝ (αρχιτεκτονικές), (β) να υποδείξουμε συστηματικές, σαφώς καθορισμένες διαδικασίες που χαρακτηρίζουν και αναδεικνύουν όλα τα αναγκαία βήματα για την ανάπτυξη περιβαλλόντων ΔΝ από το σχεδιασμό έως την τελική τους λειτουργία (μέθοδοι), και (γ) να προσφέρουμε εύχρηστες βιβλιοθήκες λογισμικού και εργαλεία για την πλήρη υποστήριξη της εφαρμογής των προτεινόμενων αρχιτεκτονικών και μεθόδων (εργαλεία). Η συνεισφορά της συγκεκριμένης εργασίας αφορά αφενός τις αρχιτεκτονικές και θεωρητικές πτυχές της κατασκευής περιβαλλόντων ΔΝ και αφετέρου τα πρακτικά ζητήματα που προκύπτουν από την διαδικασία προγραμματισμού τους. Η προσέγγισή μας στηρίζεται στην παροχή υπηρεσιών για την διάθεση των πόρων των περιβαλλόντων ΔΝ μέσα από ένα ενιαίο, ομοιογενές επίπεδο πρόσβασης και την χρήση έξυπνων αντικειμένων που παρέχουν σαφώς ορισμένες αφαιρέσεις, δυνατότητες δυναμικής αλληλεπίδρασης και σύνθεσης λειτουργιών για την υλοποίηση της λειτουργικότητας των τεχνολογιών ΔΝ. Η χρήση των εννοιών και των εργαλείων που προτείνει αυτή η εργασία βελτιώνει τις υφιστάμενες πρακτικές (i) μεγιστοποιώντας τις δυνατότητες για ενσωμάτωση ετερογενών τεχνολογιών, (ii) επιτρέποντας την ολοκληρωμένη μοντελοποίηση όλων των πτυχών της λειτουργίας των περιβαλλόντων ΔΝ, (iii) καθοδηγώντας ολόκληρη τη διαδικασία υλοποίησης, και (iv) επιτρέποντας την προσομοίωση της συμπεριφοράς του περιβάλλοντος στόχου. Οι μηχανισμοί αυτοί επιτρέπουν συνοπτικές και περιεκτικές υλοποιήσεις, ελαχιστοποιώντας την απαιτούμενη προσπάθεια για την ανάπτυξη, την εγκατάσταση και την επέκταση πλήρως λειτουργικών περιβαλλόντων ΔΝ.

Table of Contents

List of Tables

List of Figures

Chapter 1
Introduction

Ambient Intelligence (AmI) is an emerging research field that ultimately aims to make many of the everyday activities of people easier and more efficient. Ambient intelligence has been given many descriptions in the relevant literature [1], [2], [3], [4]. There is a set of traits, however, that is common among all the definitions of AmI technologies. Those features, identified in [5], characterize AmI from the user's point of view as: sensitive, responsive, adaptive, transparent, ubiquitous, and intelligent. In this regard, AmI technologies are expected to be able to fully sense the environment in which they operate (sensitive) and respond to the user's needs (responsive) in a sensible, proactive and intuitive way (intelligent). In addition, those technologies are expected to adapt to the behavior of their users based on their specific needs and preferences (adaptive), to be available everywhere at any time (ubiquitous), and be unobtrusive and invisible by "weaving themselves into the fabric of everyday life until they are indistinguishable from it" [6] (transparent).

Moreover, AmI technologies are expected to be robust, extensible and secure. They are expected to operate continuously and automatically recover gracefully from potential failures (robust), to allow for easily extending and updating their functionality without disrupting their execution (extensible), and to provide the means for protecting their operation from unauthorized third parties (secure). Although those are not among the defining traits of AmI, and are rather expected of any complete software system, they are particularly important in this case since their absence would greatly hinder the "transparency" potential of AmI technologies.

From the aforementioned defining features of AmI, ubiquity and transparency are mainly dependent on those hardware technological advancements that enable the construction of sensors and computing devices that are small enough to be embedded in everyday objects, essentially hidden from the view of humans. Wireless communication technologies also play an important role towards that direction, since the ubiquitous hidden devices need to be distributed and managed from a distance. Moreover, the transparency requirement needs

to be enforced by the way the system works and responds to the actions and needs of its users. If the system requires the user to, e.g., press a button in order to turn on the lights, then that button, as a device, is apparently not transparent to the user. However, if the same system were designed to turn on the lights when the user approaches the dark room, then the devices and different technologies involved for the implementation of this functionality are essentially transparent to the human user. Transparency, therefore, is a property of the way interaction patterns between the environment and the user are designed, which is the basic focus of the Human Computer Interaction (HCI) [7] research.

Overall, the paradigm of Ambient Intelligence clearly gives rise to opportunities for novel, more efficient interactions with computing systems. In this context, we propose a complete framework for designing and developing Ambient Intelligence environments. Through this framework, we aim to (a) define flexible and intuitive software architecture elements for modeling AmI environments and their interactions (architectures), (b) suggest systematic processes that aim to clearly define all the development steps for programming and deploying an AmI environment (methods), and (c) build intuitive, robust and sensible programming libraries and tools that will fully support the defined architectures and suggested methods for creating Ambient Intelligence environments (tools).

Through the work presented in this thesis, therefore, we aim to propose a complete software engineering approach to allow for the systematic implementation of environments that are *sensitive*, *responsive*, *intelligent*, and *adaptive* but also *robust* and *extensible*. From the aforementioned defining traits of AmI, it is beyond the scope of this work to explicitly target and discuss *ubiquity* and *transparency*, since, as mentioned before, they are the focus of different research domains and their application is not hindered in any way by our proposed approach.

Finally, the requirements and specific mechanisms for supporting *security* and *privacy* in AmI environments mainly depend on the specific types of applications that are built and operate within this context [8], [9]. There have been approaches for preserving privacy in location-based applications [10], for authenticating and controlling access to the resources available in the environment [11], and for allowing fine-grained control over the information that can be shared between different applications and services [12]. The focus of the

proposed framework is to support the implementation of the functionality of AmI environments and, although support for *security* and *privacy* is considered and discussed on many aspects of its design, the proposed approach does not provide systematic rules, guidelines and primitives for deploying provably secure applications. This is a separate open research issue and the provision of sensible security primitives and protocols that can be utilized for building secure applications is the most important part of this work's planned future amendments.

1.1 Motivation

The motivation for this work stems from the fact that there are no comprehensive all-encompassing platforms, de facto standards or guidelines for developing AmI environments. The lack of complete frameworks for basing the implementation of such environments, paired with the high degree of complexity that is inherent in their development processes, greatly hinders their potential for broader application and wide adoption. The high complexity for creating those environments comes, on one hand, from the fact that their operation requires a vast number of heterogeneous distributed computing systems cooperating seamlessly with each other in order to deliver the unified experience that characterizes ambient intelligence and on the other hand, from the diversity of the functionality and the requirements of the different AmI applications that an environment aims to support.

Approaches to building environments that implement varying subsets of the six defining traits of AmI technologies, i.e., sensitivity, responsiveness, intelligence, adaptability, ubiquity and transparency, include (a) work focusing on providing a proof of concept implementation by closely following detailed scenarios and evaluating their performance and user acceptance, e.g. [13], [14], (b) work focusing on providing a generic lightweight communication middleware for enabling the implementation of distributed services on diverse computing platforms, e.g. [15], [16], [17], (c) work that provides a richer communication middleware that either aides or automates the composition of services when those are consumed by the target AmI applications, e.g. [18], [19], and (d) work that

tries to automate completely the behavior of specific types of applications within an AmI environment through machine-learning methods, e.g. [20], [21].

Whereas their findings are very important in the AmI research field for analyzing requirements, proving the feasibility, evaluating and disseminating AmI technologies, they neither propose an integrated approach for building environments that encompass diverse functionalities nor do they provide targeted libraries and tools to simplify the implementation, deployment and extensibility potential of those environments. Moreover, the proposed distributed service-oriented frameworks lack powerful programming abstractions, cater for only a few different computing platforms and programming languages, and provide only those sets of features that are essential for realizing the functionality of their target scenarios and applications. Additionally, approaches that aim to automate either the composition of services or the behavior of applications altogether, in practice have a narrow application domain. Whereas the application of these approaches is very important in exploring specific classes of AmI scenarios and improving the accuracy of existing context-based systems, they are not very well suited in providing a practical, complete framework for building the wide range of functionalities that are expected of AmI environments.

It is our view that a software engineering approach with an all-encompassing platform is better suited for the construction of AmI environments, which, using traditional approaches, are difficult to develop, are complex to deploy and administer, and are difficult to extend in the face of changing requirements. In this context, we believe that there is a need for integrated, intuitive, well defined software elements, libraries and tools for the formulation and precise definition of the implementation process for creating AmI environments.

1.2 Objectives

Towards achieving our goal, we propose a complete framework that incorporates software libraries, novel architectural elements and tools in order to provide sensible and intuitive building blocks and guidelines for the construction of user-centric, dynamic, robust and extensible Ambient Intelligence environments.

The AmI environments we are targeting, ultimately are those environments that are able to fully sense the physical and digital context in which they operate (sensitivity), respond to the needs of their users (responsiveness) in a sensible, proactive and intuitive way (intelligence), and, finally, adapt to the behavior of their users based on their specific capabilities and preferences (adaptability).

In this context, specifically, our purpose is (a) to provide comprehensive, well-defined and flexible architectural entities for modeling all aspects of AmI environments, (b) to provide sufficient abstractions and software layers for homogenizing the access to all the resources, sensors and algorithms contained in AmI environments, (c) to provide abstractions and software layers for describing the functionality of AmI environments and how those respond to the actions of their users, (d) to provide tools that will guide and streamline the process of building AmI environments, and (e) to allow the whole framework to be easily extensible in order to incorporate future functionalities, development patterns and abstractions that will emerge from its usage.

1.3 Approach

Our approach is structured over three pivotal axes: (a) a service middleware for the creation of distributed services that will enable the exposure, under a unified homogeneous access layer, of all the software and hardware resources that are available in an AmI environment, (b) an application middleware that through its architectural abstractions, interaction capabilities and composability methods will be able to utilize the resources exposed as software services in order to deliver fully functional AmI environments, and (c) a set of programming libraries and tools that enable the implementation and deployment of the software abstractions that comprise the aforementioned layers. The high-level layered architecture of the proposed approach can be seen in Figure 1.

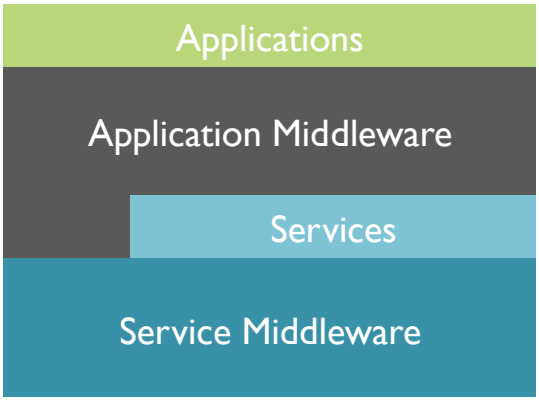


Figure 1 – Proposed high-level layered architecture for Ambient Intelligence environments

The fundamental abstractions on which an AmI environment is built, under the proposed approach, is the Distributed Service and the Smart Object. The functional role of Distributed Services (or just Services) is to provide the means for implementing the sensitivity and responsiveness of the environment. On the other hand, the functional role of Smart Objects is to provide the means for implementing the intelligence and adaptability of the environment. The modeling of the environment using these two fundamental entities was inspired by the Data, Context and Interaction (DCI) architecture [22].

The DCI architecture emerged from previous work on role-based modeling and specifically the Object Oriented Role Analysis and Modeling software engineering method (OOram) [23]. The basic idea behind the DCI paradigm is to separate the code that describes the system state from the code that describes the system behavior [24]. In DCI, the functionality of an operation is organized under three basic perspectives: the Data perspective, the Context perspective, and the Interaction perspective. The Data perspective captures the system's state and comprises of those entities that model its domain knowledge. On the other hand the Context and Interaction perspectives capture the system's behavior – i.e. how the system behaves as a reaction to a command or a sequence of commands issued by the user. In this sense, the Context comprises of the specification of a network of communicating entities that realize a user command, whereas the Interaction perspective comprises of the specification of how entities communicate when executing a user command.

Hence, in the proposed approach, the architectural role of Services is to model the slow-changing domain knowledge of the AmI environment, and the role of Smart Objects is to model the latter's rapidly-changing behavior. The aforementioned adjectives refer to the

fact that the resources that are present in an AmI environment, during its development lifecycle, do not evolve as fast as its perceived behavior. In this regard, the set of all the Services available in the environment provides the answer to the question "What the environment *is*". On the other end, the set of all the Smart Objects and they way they interact with each other provides the answer to the question "What the environment *does*".

Therefore, in practice, Services in an AmI environment implement physical things, sensors and algorithms that characterize the nature and capabilities of the environment. For example, chairs, tables, lights, displays, temperature sensors, and touch surfaces, but also human identification systems, speech recognition systems, etc. are represented and exposed to the programmer of an AmI environment as Services. Additionally, behaviors that implement the environment's reaction as a response to the user's needs or actions, e.g., "When I wave my hands the lights turn on", "When the owner is recognized the door opens", etc. are represented either as a single smart object or as a network of interconnected and interacting smart objects.

At the same time, the process of building AmI environments using Services and Smart Objects is guided by a set of graphical programming tools that further simplify and streamline the whole process while enabling the simulation and testing of the environment. Specifically, the framework provides software components for: (a) authoring and managing the descriptions of the available Services, (b) generating language-specific files for using and implementing services in the supported programming languages, (c) automatically deploying, restarting and updating services, (d) monitoring and managing the deployed services, (e) programming smart objects and their interactions, and (f) visualizing and simulating smart objects and their interactions.

1.4 Contribution

In accordance with our objectives, following the aforementioned layered approach for building AmI environments, the contributions of this work are the following:

- The definition of a sensible software architecture for the implementation of AmI environments

- The design and implementation of a type-safe communication middleware that enables the implementation of robust services that expose all the resources available in an AmI environment over a unified access layer
- The design and implementation of a new programming language, called Akkadian, for implementing fully functional smart objects and their interactions, in order to describe the functionality of an AmI environment
- The complete formulation of the development process by providing tools for guiding the implementation and deployment of services, and also, for managing and simulating smart objects and their interactions
- The provision of formal descriptions of the building blocks and the aspects of their interactions within an AmI environment

In this sense, it can be inferred that the contributions of this work are twofold. The focus is on both the architectural and theoretical aspects of building AmI environments, but also on the practical issues that stem from the actual development process. Therefore, on one hand, this work provides a formal modeling of the building blocks of the AmI environment and the key aspects of their interactions, using the Z-notation [25]. On the other hand, it provides concrete, fully functional and intuitive implementations that constitute a stable platform that substantially simplifies the development process of the inherently complex AmI environments.

1.5 Thesis Structure

The rest of this thesis elaborates upon the issues raised above and is organized as follows:

Chapter 2 describes and reviews other approaches for creating AmI environments. Additionally it provides descriptions and comparisons with existing technologies that relate to different aspects of the proposed work and its constituent concepts.

Chapter 3 introduces the requirements, design and implementation of the proposed service middleware and the tools that support the description, development and deployment of services that expose the heterogeneous resources that are available in AmI environments under a unified, robust and intuitive access layer.

Chapter 4 introduces the concept of smart objects, smart object federations and their interactions. Subsequently it describes how these smart objects can be programmed and deployed using the programming language Akkadian and its runtime. At the same time, it uses representative programs as a means to showcase the effectiveness of the proposed programming language in describing smart objects and their interactions. Additionally, this chapter discusses the concept of the external smart object that can be implemented as an external, independent hardware or software device but can seamlessly interact with the other smart objects and participate as an equal actor in the implementation of the functionality of AmI environments. Finally, Chapter 4 describes the implementation of ObjectivSim, the development and execution environment, which provides support for programming, deploying and simulating smart objects and their interactions. The functionality of an AmI environment is essentially realized as Smart objects implemented and deployed in ObjectivSim.

Chapter 5 evaluates the proposed approach and provides examples that showcase its applicability and effectiveness towards implementing and extending AmI environments.

Chapter 6, finally, summarizes the proposed work and discusses some ideas for improvements and extensions as future work.

Chapter 2
Related Work

Despite being given many different descriptions in the relevant literature, e.g. [1], [2], [3], [4], as identified in [5], Ambient Intelligence (AmI) technologies can be collectively characterized from the user's point of view as: sensitive, responsive, adaptive, transparent, ubiquitous, and intelligent. AmI technologies are expected to be able to fully sense the environment in which they operate (sensitive) and respond to the user's needs (responsive) in a sensible, proactive and intuitive way (intelligent). They are expected to adapt to the behavior of their users based on their specific needs and preferences (adaptive), to be available everywhere at any time (ubiquitous), and be unobtrusive, to the point of being, practically, invisible to the user [6] (transparent).

With varying subsets of the aforementioned traits as their defining features, ambient intelligence applications have been applied to many different domains such as: health monitoring and assistance [26], [27], energy management [28], transportation [29], [30], disaster management [31], [32], education [33], entertainment [34], [35], interactive exhibitions [36] and commerce [14], [37]. Moreover, AmI applications have been used for reimagining and implementing aspects of different physical locales such as the home [20], [38], [39], the hospital [40], the classroom [13], [41], [42], and the workplace [43], [44]. It is important at this point to make the distinction between individual AmI applications that implement a set of well-defined operations within the context of a particular domain (e.g., health monitoring, education, etc.) and AmI environments that encompass a potentially diversified set of applications in order to support and augment the varied functionality of physical locales (e.g., the home, the office, the classroom, etc.). In the latter case, a single AmI environment, such as the home, may contain a broad range of applications for supporting home automation, health monitoring and assistance, education, entertainment and office collaboration. These different functionalities on one hand have different requirements with respect to their expected outcome and reliability guarantees, but on the other hand they need to interoperate with each other in order to provide a seamless

experience within the target AmI environment. For example, a health monitoring application has to provide strong guarantees for accurate real-time monitoring of the user's health metrics while a home automation application may tradeoff accuracy for automated adaptation potential by applying probabilistic machine-learning algorithms in order to learn and adapt to the habits of the user, such as [20]. Practically, such diverse functionalities are built using different technologies and infrastructures. However, in the context of a cohesive AmI environment, from a software engineering point of view, these diverse technologies should have a common denominator that provides a unified view of all the available functionalities. There should be a mechanism that enables the environment to behave and "feel" as a single entity and not as a bag of isolated applications. This is the main goal that drives the proposed approach.

In this context, related work for building AmI environments can be categorized into the following general approaches:

- Applications focusing on providing a proof of concept implementation by closely following detailed scenarios and evaluating their performance and user acceptance
- Provision of a generic lightweight communication middleware for enabling the implementation of distributed services on diverse computing platforms
- Provision of a richer communication middleware, augmented with formalized metadata, that either aides or automates the composition of services when those are consumed by the target AmI applications
- Full automation of the implementation of behavior for specific types of applications within an AmI environment, mainly, through machine-learning methods

In this sense, existing approaches do not provide a complete, all-encompassing formulation of the implementation process for creating AmI environments. Specifically, overall, they fail to maximize the potential for incorporating heterogeneous technologies; they do not provide flexible, well-defined programming and architectural abstractions for modeling both applications and services; they do not provide adequate support for composing higher-order functionalities from diverse components; they do not allow for intuitive mechanisms for filtering, capturing and responding to the environment's context; and they sidestep many practical issues that are raised by the actual development process by providing limited

implementations of targeted programming libraries and tools for supporting the programming, deployment and testing of fully functional AmI environments. A comprehensive evaluation of existing approaches against these requirements is provided in section 2.4.

Apart from the related work towards creating AmI applications and environments, the proposed approach, with services and smart objects, also relates to other more general technologies and concepts. Therefore, this chapter will focus on approaches for creating AmI environments but also discuss technologies that are related to aspects of the proposed approach, i.e., event-driven programming languages, composition languages, and software agents.

2.1 AmI Applications and Environments

In the MundoCore project [17] applications are synthesized by services that are based on a custom lightweight publish/subscribe message exchange communication middleware. To address the high degree of heterogeneity of platforms and networking technologies, it is based on a layered design and provides a common set of APIs for different programming languages (Java [45], C++ [46], Python [47]) on a wide range of different types of devices. Applications are developed by means of subscribing and listening for the messages in any of the supported programming languages. MundoCore's lightweight implementation supports resource-constrained systems and enables the deployment of alternative transport protocols for the delivery of the exchanged messages. In relation to the proposed service middleware, MundoCore supports fewer languages, does not support natural and type-safe programming abstractions for Object-oriented programming languages, and it lacks support for automatic service deployment and update.

Similarly, AmICom [48] provides a generic and lightweight communication middleware for service consumption in ambient intelligence applications. The main purpose of AmICom is to enable different devices to expose their resources as distributed services, independently from their network architecture and without any reconfiguration of the underlying network topologies. In AmICom, the exchanged messages are type-safe and are encoded according to their type descriptions which are provided in a custom interface description language

that is similar to CORBA's IDL [49]. AmICom supports by design different transfer protocols and resource-limited devices. In relation to the proposed service middleware, it only supports C/C++ [50] and Java programming languages, its programming abstractions do not allow for a natural mapping to Object-Oriented languages, and it lacks support for service deployment and update.

The Hydra project [51] aims to support mostly mobile devices and provide a common layer over the different communication technologies that are embedded in them (i.e., Bluetooth [52], Wireless LAN [53], GSM [54], GPRS [55]). In addition to the low-level communication facilities, the Hydra framework offers higher-level methods that on the one hand provide localization and context data to applications built on top, and on the other hand perform specialized data filtering depending on personalization information. Moreover, it provides a content delivery application that can display XHTML [56] data according to user- and context-specific preferences. The content delivery system can be used and extended by applications that want to add additional functionality. Overall, Hydra aims to provide a complete software engineering approach for creating applications that are run and controlled by mobile devices. The provision of the content delivery system and the support of many different mobile devices constitute a comprehensive platform for the implementation of applications that deliver user interfaces for synthesizing and managing the resources in an AmI environment but do not provide a systematic way for modeling the interaction beyond the provided interface. Additionally, Hydra supports only the Java programming language, the exchanged messages are not type-safe and the programming abstractions do not allow for intuitive mapping onto object-oriented languages.

LAICA project's [15] main objective is to enable the control and collaboration of different resources and devices that are distributed at a city-wide level in order to implement smart city applications. LAICA's architecture is realized by two specialized types of agents to model the devices, resources and actions, and a middleware that is used for the routing of the communication messages and for providing some context to the deployed agents. The two types of the supported agents are the "sensor agents" and the "effector agents". The role of the former is to provide context information and the later to act upon changes in the context. LAICA's architecture also includes a separate, centralized control and management service that any agent may use to manipulate the behavior of any other agent. This system

was built in the Java programming language and incorporates many encoders and decoders for storing and manipulating sensor data. In this sense, the LAICA middleware only serves as a connection infrastructure and as a service provider. Therefore, comparing it with the proposed communication middleware, it uses low-level communication primitives that cannot be naturally mapped to Object-oriented languages and while it aims to be deployed in wide-scale environments, it relies on many centralized software components.

The WSAMI middleware [57], [58] proposes a solution for the dynamic composition and deployment of Web Services [59] targeting mainly resource constrained mobile devices. In WSAMI service descriptions contain both a concrete interface that describes the protocol details and the endpoint address of the service and an abstract interface that describes the messages that are supported. The abstract interface is used for dynamically composing other services through Web Service Choreography [60]. The services, either concrete or composed, are subsequently consumed by applications in the Java programming language. In our approach, service composition happens at the application middleware level though the attachment of services to Smart Objects. On the other hand, WSAMI, compared to the proposed service middleware, it does not support the delivery of asynchronous events, it only supports the Java programming language and it does not provide tools for the automatic deployment and update of services.

Utilizing Web Service technologies for AmI is also explored in the Web to Peer (W2P) system [61]. W2P implements both synchronous (request/response) and asynchronous message exchange between applications and services. While it adds support for C++ and C# [62] in addition to Java, it does not support service composition. Furthermore, it uses low-level message exchange primitives, enclosed in SOAP envelops [63], that do not allow for intuitive programming abstractions in the supported Object-oriented languages.

The main contribution of DoAmI [64], [65] is a service discovery mechanism for choosing instances of different types of services according to the (static or dynamic) configuration of the requesting applications. This discovery mechanism is implemented on top of C++ and Java implementations of the Common Object Request Broker Architecture (CORBA) [49] and it resembles in functionality the dependency injection pattern [66]. In this regard, an AmI application describes, through its configuration, the characteristics and type of the services

it uses. Subsequently, the required services are resolved and attached to the program variables by the DoAmI middleware. This implementation, despite supporting fewer languages, has many similarities with the proposed service middleware since it draws many of its design characteristics and programming abstractions from CORBA, on which both approaches are based. However, DoAmI does not support asynchronous events neither provides tools for guiding the implementation of services, their deployment and update.

The AMIGO project [18], [67] proposes an abstract architecture for a service-oriented AmI middleware which focuses on providing support for ontology-based service composition in order to establish a systematic and flexible way to consume services from AmI applications [68]. Service descriptions are semantically annotated in an extension of the OWL-S [69] markup that adds support for Quality of Service (QoS) [70] and heterogeneous service infrastructures. In AMIGO, applications are built by means of requesting and using resources, which are abstracted by the provided ontology, through the service discovery subsystem. The obtained resources and services are subsequently utilized depending on their underlying implementation technology. While the abstract AMIGO architecture does not specify the underlying service communication technology, its example applications implement Web Services in Java and C# programming languages. In this sense, the usage of the AMIGO approach resembles WSAMI mentioned before.

The agent paradigm for building ambient intelligence environments has been explored in the iDorm project [71] where embedded devices that have the capability for reasoning, planning and learning are referred to as “embedded agents”. According to this approach, all sensors and actuators in the environment are exported as XML [72] message-based request/response services from a centralized server and specialized agents running on embedded devices monitor these services in addition to the behavior of the other agents in order to observe all the changes that they are interested in and learn the conditions under which these changes were performed. Therefore, in addition to their preprogrammed rules, these agents are able to record and learn from the user’s actions what changes they should make to the environment under what circumstances. This approach is used for programming-by-example simple system behaviors, by the user, for home automation operations.

The iDorm project shares similar goals with MavHome [20], which also focuses on monitoring and learning techniques to automate the behaviors of intelligence environments for a single human user. MavHome continuously classifies perceived user behaviors (data) in order to recognize usage patterns, which are modeled as a hierarchical hidden Markov model [73], [74]. Subsequently, depending on the recognized patterns, the system chooses and evaluates the action that has to be performed in response, and any negative or positive feedback from the system or the user is fed back to the decision making component to improve the model's performance. The system utilizes a static, well defined, set of sensors and services that are accessible to the other components as CORBA [49] services.

The Gator Tech Smart House project [38], [75] aims to design and implement a fully functional AmI home environment. Gator Tech's middleware models the hardware sensors and actuators that are available in the environment as services built on top of the OSGi framework [76] in the Java programming language. The middleware also defines a separate software component for specifying on one hand the services that should be activated for different applications, and on the other hand, to specify actions that are executed when specific services produce specific values (context). Moreover, Gator Tech's middleware includes semantic descriptions for a subset of the supported Java value-types in order to automate value conversions that are essential for services to exchange compatible data (e.g. convert Fahrenheit degrees to Celsius). In this regard, applications are created by consuming the available services, composing higher-order services, and associating different contexts with actions. For this reason, Gator Tech has many similarities with our approach since smart objects can compose higher-order services and their functionality is essentially implemented as a reaction to the changes in their environment (context). Smart objects, however, leverage Gator Tech's approach by defining a high-level programming model for implementing the functionality of AmI environments and by enabling different service technologies to be composed under a single smart object. Moreover, both implementations include simulation capabilities for testing the behavior of the environment, however in the Gator Tech platform only the environment's resources can be simulated and not their behavior.

MIT's Oxygen project [77] has developed an experimental system called O2S which proposes the abstraction of Goals [78] for programming intelligent environments. Goals

encode high-level intentions which the underlying system tries to satisfy by dynamically assembling a set of generic components called pebbles. A pebble in Oxygen's middleware is more primitive than a service since it represents a single function with one input and one output (e.g. an audio-sink pebble accepts an audio stream in its input and redirects it to its audio device). Therefore, the environment is essentially composed by a set of pebbles and set of rules that describe how different goals can be satisfied. Subsequently, an application is dynamically instantiated as the process through which the current goal can be satisfied. While this approach is fundamentally different from ours, it defines concrete software abstractions for modeling and differentiating between the nature of a system and its behavior (functionality). O2S however, does not specify how the context is maintained neither provides tools to support the development of the environment.

CMU's Aura [79], [80], [81] environment defines a high-level abstraction, termed a Task, layered on top of individual services. The task layer, called Prism, provides a placeholder to capture (a) the high-level user intent by means of the services it requires in order to be satisfied, and (b) the initial state of those services. Subsequently, it employs various resource monitoring mechanisms in the Aura system to monitor and adapt underlying services to opportunistically carry out the high-level task. In this regard, Aura's architecture, like the approach in the project 2WEAR [82], is focused on the adaptation and migration of applications. Therefore, the functionality of the environment in Aura is defined as a sequence of tasks that are initiated by the user. There's no mechanism however for modeling fluid transitions between tasks by dynamically composing or interrupting them. Therefore, aside from tasks, Aura plans to introduce the notion of Activity [83] which denotes those computational abstractions that can be initiated, suspended, stored, and resumed on any computing device and can be reassigned to other users, or shared among several users. This abstraction has some similar properties with smart objects in our approach; however, the focus, in Aura project's design document, is on the definition of activities with graphical user interfaces by end users and not on managing activities and ensuring their operation in a unified environment where many interdependent and interconnecting activities take place.

The Gaia middleware [84], [85], [86] defines a programming model for applications based on the Model-View-Controller abstraction [87], [88]. Using this abstraction, applications in Gaia

are partitioned into five parts: (a) the model for representing the application logic, (b) the presentation for exposing the application data, (c) the input sensor for providing the means to interact with the application, (d) the controller for mapping input sensor events and context changes in the environment into method requests for the model, and (d) a coordinator for managing the registration (wiring) of specific instantiations of the aforementioned software components in the application's model. These parts are implemented as CORBA services that are wired together to instantiate an application (bootstrapping) in the Lua programming language [89]. Additionally, the Gaia middleware provides a separate context management service that an application (more specifically, the model of the application) may use to be notified about specific changes in the state of the environment. In this sense, Gaia reflects a philosophy similar to the proposed approach, where the application logic is separated from individual components of the environment and defines a concrete basis for modeling user-driven context-aware applications. However, this segmentation of applications increases the programming complexity of applications by distributing their logic into completely separate, isolated, loosely coupled code units and does not provide development tools to reduce that complexity. Moreover, the realization of the event distribution mechanism and the context-awareness of the environment as monolithic services hinder its coherence and extensibility potential.

In One.World [90], [91], there are multiple environments that can be hierarchically structured and incorporate applications and their persistent data. Applications are implemented by means of handling asynchronous events which are generated by the system, the application itself or by other applications. In this regard, there is a clear separation of the state of an application, its functionality and its resources (i.e. sensors, actuators, etc.), which are modeled as a set of tuples that they can emit or receive. Application data in One.World are also implemented as tuples which can be monitored and queried by applications. One.World’s treatment of data as functionally equal to asynchronous events in the context of a software entity is the same with our approach, where smart objects are able to capture both asynchronous events and modifications to their internal state. Moreover, One.World’s applications resemble the abstraction of smart objects and its environments resemble smart object federations in our approach. However, compared to our approach, One.World does not model the way environments are

encapsulated into other environments, it does not support composition of services, and the representation of context and data as query-able tuples, although flexible, does not fit well as a programming abstraction in the target implementation language, Java. In our approach, asynchronous events are higher-level and are handled by a specially-designed event-driven programming language which provides intuitive programming abstractions for managing them. On the other hand, One.World's runtime system maintains and guarantees the persistence of application data and exchanged events which greatly enhances the robustness of the system. This is a definite advantage that we plan to support in a subsequent version of our framework.

The basic building blocks of the approach followed by the PICO middleware [92], [93], [94] are the Camileun and the Delegent (sic). With the former modeling hardware devices that expose a set of well-defined functionalities, and the latter models the behavior that is imposed on camileuns by a set of rules or by the user. Furthermore, delegents can be organized in communities in order to collaborate towards a specific goal and can also collectively play the role of a single delegent. All the components communicate with each other by exchanging messages and the framework supports basic automatic, opportunistic composition of functionalities for cameleuns with streaming support. This reference architecture has common characteristics with our approach, since camileuns resemble services which play their role – in the context of an application – only through smart objects that in turn resemble delegents. Communities, additionally, resemble smart object federations in that they both provide a conceptual grouping of collaborating entities. However, there are four fundamental differences between our approach and PICO's reference architecture, which: (a) does not model the communication and organization of delegents in communities, (b) does not model how behaviors are instantiated in delegents, (c) does not specify a model for the environment's context and how it is queried and maintained, and (d) the fact that a community of delegents can play the role of a cameleun, weakens the abstraction of delegent since it effectively obviates its use for the design of applications.

The UIO architecture described in [95], provides a modeling for interconnected entities, called Ubiquitous Intelligent Objects, that can either reside in the physical space or the cyberspace. The proposed architecture does not deal with the organization, communication

or interactions between these entities, but raises an important design issue that is also considered in our approach. This issue is the modeling and implementation of the seamless coexistence and interaction between hardware-augmented physical entities (e.g. a cooking utensil, a painting, or even a phone) and pure software entities (e.g. internet search, a database, etc.). In our approach we have defined the notion of external smart objects, which can be implemented in hardware, or a mixture of software and hardware, and can interact seamlessly with other smart objects. Through this feature, the design and implementation space of an AmI environment is greatly enhanced since (a) it allows the first-class incorporation of legacy systems, (b) it enables resource limited hardware devices to offer functionality and exploit functionality offered by the AmI environment, and (c) it does not compromise the integrity of the already established architectural abstractions.

In the FedNet framework [96], [97] the environment is composed by isolated applications that are modeled as a collection of primitive Tasks. A task in FedNet references a single high-level component, called Profile, through which it expects a result. A profile, which essentially describes a state-full function, specifies two things: (a) the name and type of the function's inputs and outputs, and (b) the name and type of its state variables. When an application is deployed, the FedNet runtime opportunistically discovers a set of profiles that match the application's tasks and binds them to it. Following this association, profiles deliver their result to their associated tasks (through polling), which in turn, deliver it to the application for processing. FedNet also specifies the notion of an artifact as a physical object that is capable of reporting and affecting its state remotely (e.g. a lamp with microcontroller and Bluetooth [52] support). An augmented artifact is then defined as an artifact that supports one or more profiles that can be attached to it at runtime. In the description of its architecture, FedNet sometimes refers to augmented artifacts as smart objects but clarifies that those terms are identical. Applications, as a set of tasks, and profiles, as a set of inputs, outputs and state, are specified in separate XML-formatted documents [72] and are subsequently implemented in the Java programming language. Tasks can affect the state of their attached profiles through SOAP messages [63] and poll them for obtaining their results using the RSS protocol [98]. In this regard, FedNet pushes the extensibility potential of the applications to the resources of the environment by allowing their runtime augmentation with different profiles. This is a fundamental difference compared to our approach, where

we view the resources as the slowly-changing domain knowledge of the environment (modeled as services) and the applications, that use those resources, as the rapidly-changing environment behavior (modeled as smart objects). FedNet's implementation also provides a tool for the deployment and management of applications; however, it does not offer tools for guiding their development.

The main architectural abstractions of the CORTEX middleware [99], [100] are the Environment and the Sentient Object. A sentient object is an autonomous computing entity that can interact with the environment and with other sentient objects. All the interactions with the environment and among sentient objects happen through the emission of asynchronous events that are sent and received by the participating entities. The CORTEX middleware defines the notion of filter, through which an object can express interest in certain types of events with certain payload values. Additionally, it defines the notion of zone which introduces a means of scoping or limiting the propagation of event notifications in the system. Objects can be arbitrarily organized into zones, where a zone can be seen as a collection of objects and event notifications are only propagated within the zone of the object raising the event. In this regard, applications built on CORTEX can be seen as a subset of interacting objects within the environment. This high-level view of the environment with interacting objects that can be organized in groups and respond to asynchronous messages is very similar to ours. However, there are two fundamental differences: (a) the CORTEX middleware does not define the notion of services and a sentient object is assumed to fully contain and execute the functionality to sense and affect the environment and the other objects, (b) the notion of zone resembles the notion of a federation in our approach, however a federation is a directed multi-graph with no self-loops that on one hand, like a zone, restricts the propagation of events, but on the other hand, models the dependency relations among a group of objects allowing the environment to reason about its communication/dependency paths. CORTEX has chosen this approach, since it targets mission-critical environments with real-time guarantee requirements (e.g. cars). In the context of an AmI environment application framework, however, we can leverage this software engineering approach to increase its extensibility, adaptability and reuse potential by dynamically attaching functionality to objects through services, introducing a dynamic

event-based programming language for expressing intuitively the offered abstractions and providing tools for developing, visualizing and simulating the behavior of the environment.

2.2 Programming Languages

The proposed approach aims to provide a fully functional development and deployment tools for programming AmI environments. Towards this direction, apart from the proposed service middleware which allows the description, implementation and deployment of services with type-safe interfaces, an important part of this approach is the environment for the development and deployment of smart objects. The core part of the latter is a dynamic, event-driven programming language called Akkadian. Through Akkadian, developers can fully describe smart objects, establish smart object federations and program their interaction logic.

The proposed programming language shares many core design goals with the programming language nesC [101], which is used for programming sensor nodes in Wireless Sensor Networks (WSNs) [102]. Both are component-based, event-driven programming languages where the execution happens concurrently as emitted events reach the relevant code units. Additionally, like nesC, the execution of a command in Akkadian does not block the flow of the program. However, they have four fundamental differences. First of all, the proposed language is dynamic and weakly-typed whereas nesC, being an extension of the C programming language, is static and strongly-typed. Secondly, in nesC, the bidirectional interfaces that are either “used” or “provided” by a WSN node are essentially static libraries that are linked with the final program at compile time. In the case of Akkadian, a port that “provides” or “requests” a service, resolves the service definition dynamically, at runtime, and the target service can be implemented in any of the supported service technologies. Thirdly, in nesC, event handlers are implemented for individual events, and trigger conditions that involve multiple events have to be explicitly identified by the programmer using shared variables. On the other hand, in Akkadian, multiple events can be considered in an event handler and connected with Boolean operators to create powerful and expressive trigger conditions. Moreover, concerning the fourth fundamental difference, changes to the

memory of smart object ports, can be captured as events and participate as trigger conditions in event handlers.

Aside from the fundamental differences, the common design decisions between Akkadian and nesC are no coincidence, since WSNs share similar goals with smart objects. However, WSNs are primarily designed for implementing low-level application scenarios, realized by the redundant deployment of low reliability hardware devices. In this sense, WSNs are organized and acting autonomously while trying to achieve their goals requiring little or no user intervention [103]. On the other hand, the target of smart objects is the description and implementation of high-level user-centric AmI environments in which the actions and context of the user actively influence the operation of the smart objects in the environment [104]. Additionally, smart objects are designed to model both hardware devices and software entities and it is that potential for seamless interoperation that expands the programmer's design space and drives their effectiveness for describing high-level, user-centric scenarios.

Languages like ESTEREL [105] and LUSTRE [106] target embedded hard real-time systems following an inherently synchronous message passing paradigm. PushLogic [107] is a compiled block-structured imperative language that provides the means to centrally manage the services offered by distributed or local devices. These devices are collectively referred to as pebbles. PushLogic allows the definition of rules that are evaluated concurrently ensuring that all the assertions over its variables hold during the execution of the program. Despite having similarities with Akkadian, since both consume diverse services and evaluate concurrently their rules, PushLogic targets safety-critical systems and thus provides only static memory allocation, does not implement arrays and provides lower-level constructs and operators in order to exercise very fine-grained control over the devices it manages and to keep the memory and processing requirements of the resulting programs low. ArchJava [108] is implemented as an extension of Java for allowing the latter to dynamically invoke external software components by binding them to bidirectional interfaces, called input and output ports. However, ArchJava is only applicable to components running under the same Java virtual machine, does not support the binding of distributed services, and provides no means for message-matching and concurrent execution.

Urbiscript [109] is a prototype-based object-oriented dynamic language that includes syntactic support for concurrency and event-based programming. In this regard it provides support for emitting and capturing asynchronous events, for filtering events based on their name and payload values, and for concurrent execution of command invocations, like Akkadian. Moreover, similar to Akkadian, urbiscript supports capturing the updates of arbitrary memory locations through event-capture statements. Urbiscript was designed to be used in the Urbi robotics platform [110] for orchestrating high-level behaviors from hardware drivers and distributed software components that are implemented on top of the framework's UObject C++ middleware. Therefore, in addition to the provision of intuitive asynchronous event-based semantics, it focuses on incorporating functionality from services implemented and deployed outside the context of the language's runtime. However, compared to Akkadian and its runtime, urbiscript does not support the consumption of services implemented over many programming languages and diverse service technologies; it does not support capturing of multiple events in a single event-capture statement; it does not allow the expression of capturing combinations of events and memory locations; and it cannot handle nested event-capture statements. Additionally, whereas it provides an efficient and intuitive base for describing and implementing the behavior of a single robot it does not define any means for allowing programs to affect and send or receive events from other urbiscript programs (possibly running on other robots).

Furthermore, Akkadian, can be viewed as a composition language [111], since smart objects can offer or consume services that are either running on the same hardware or are distributed over the network. A smart object may incorporate an arbitrary number of services built on different technologies making an object described in Akkadian an effective container for composing functionalities from diverse, heterogeneous components. The composition language PICCOLA [112] shares similar goals, while supporting both imperative and functional programming styles. It also supports concurrent component invocations but does neither provide any means for filtering invocation results nor is able to utilize distributed components.

Service composability, apart from the efficient and rapid implementation of high-level functionality, also allows for easier deployment and testing of the target application as it is being built. This is an important and useful property in the field of AmI, as also identified in

many of the approaches for building AmI environments described in the previous section. The Business Process Execution Language (BPEL) [113] is a declarative, XML-based language that supports process-oriented service composition. BPEL describes a composition result (process), in terms of participating services (partners) and message exchanges or intermediate value transformations (activities). BPELJ [114], extends BPL and enables the inclusion of standard Java code inside BPEL declarations. Aside from the fact that BPEL-based languages are available only for web services, their declarative XML-based syntax makes them verbose and difficult to read. Graphical development environments for BPEL, however, mitigate the readability problem at the cost of requiring the programmer to describe the composition with graphical elements representing process workflows.

2.3 Software Agents

Another approach that has been used in the AmI domain is the agent paradigm [115], [116], [117]. Agents, due to their capability of both executing in a proactive way and reacting to environment changes are deemed as flexible and intuitive abstraction for developing AmI environments and, in general, distributed systems, e.g., [71], [118]. In this context, particular similarities can be observed between asynchronous smart objects and agents. First of all, both aim to serve as an intuitive abstraction on which complex, inherently distributed systems can be built. They provide an adequate set of concepts, abstractions and mechanisms for modeling complex systems. Secondly, like smart objects, agents are active and autonomous in nature, continuously sensing their environment and adapting or acting to its changes. Thirdly, smart objects and agents are designed to enable the creation of (functionally) flexible systems without rigid and predetermined functionalities. And lastly, they offer intuitive metaphors and support for managing organizational and functional relationships among the constituent components of complex applications.

However, agents are by design more broad and generic autonomous computing entities, aiming to provide an abstraction for many diverse application domains. This genericity necessitates the fine-grained and low-level message exchange mechanisms that are an inherent part of their implementations [119]. On the other hand, smart objects have a clearly defined role, which is to support the construction of AmI environments. Under this

premise, they provide an inherently asynchronous high-level message exchange and message filtering/matching mechanism that allows for easier description and development of distributed interacting processing entities. In this sense, agents are a lower-level abstraction than smart objects, and the latter can be considered a specialization of the former. Additionally, the inherent asynchronous execution model of smart objects, paired with the constructs provided by the proposed language, allow for a more natural description of AmI environments, which are ultimately constantly responding to external, asynchronous stimuli. Moreover, the capability of smart objects to form and inspect object federations constitutes a powerful mechanism for monitoring and reacting to the state of the environment and for explicitly expressing functional dependencies between interdependent entities. Also, allowing external objects to inspect, join and break federations allows for easy and non-intrusive extensions in AmI environments.

2.4 Discussion

The goal of this thesis is to provide a framework that incorporates software libraries, sensible architectural elements and guidelines for programming ambient intelligence environments that are sensitive, responsive, intelligent and adaptable. More specifically, apart from formally defining the participating entities and abstractions, through our approach, we aim to (a) maximize the potential of the environment for incorporating heterogeneous technologies, (b) provide sensible programming abstractions, (c) enable the implementation of higher-level functionalities by composing services, (d) provide a flexible model for representing and using the context of the user, (e) provide libraries and tools for developing and deploying services, (f) provide libraries and tools for developing and deploying applications, and (g) provide the means for simulating the functionality of the environment. In this case, “context” is any information that can be used to characterize the situation of an entity – i.e., a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and applications themselves [120].

In this section, using the aforementioned properties as the basic axis of our evaluation framework, we will assign a score value to each of the reviewed approaches for

implementing AmI environments (2.1). This value signifies how well an approach enables the provision of each property. In general, we use the symbols: “L” (Low) to signify minimum support of a feature, “M” (Medium) to signify that support of a feature was considered but it is not complete, and “H” (High) to signify that a feature is fully supported.

Specifically, the “Heterogeneity support” (Heterog.) of an approach is given the value “L” if its implementation supports the incorporation of components programmed in less than or equal to two programming languages. The value “M” is given if the approach supports three or four languages, and finally, “H” is given if the approach supports five or more. In this case, since all approaches provide layers for potentially supporting many different technologies, only the ones actually implemented are considered. Our approach supports five languages (C/C++, .NET, Java, Flash/ActionScript, and Python) for the implementation of services in the proposed middleware, but also through the abstraction of Smart Objects apart from these services, it supports the incorporation of services implemented as CLR DLLs [121], and resources implemented as REST services [122].

The “Programming abstraction completeness” (Abstract.) of an approach is given the value “L” if the implementation offers abstractions only for services and service composition but ignores applications and assumes that those are built only by consuming primitive and higher-order services. The value “M” is given if it provides abstractions for applications but it does not model how these applications can interact and interoperate together in the context of an AmI environment. Finally, the value “H” is given if the approach provides sensible abstractions for both resources and applications and specifies their interoperation in the context of an AmI environment. Our approach provides programming abstractions for resources as services and the functionality of the environment is defined through interacting smart objects, a distinct subset of which denotes an application. In this sense, the interoperation of both services and applications is well-defined in the context of an AmI environment.

The “Composability potential” (Compos.) of an approach is given the value “L” if the only way to compose functionalities that utilize different services/components is to define a new service that explicitly implements the new functionality by utilizing the required components. The value “M” is given if the approach supports composition but requires

some mediation code to be given in a high-level language or a declarative language. Finally, the value “H” is given if the approach supports semantic annotation of the functionality of the components/services and thus allows for fully automated composition of the required higher-order functionalities. Our approach falls into the second category, allowing composition of services in the context of smart objects (programmed in Akkadian). However, through this method, it supports composition even from services implemented over diverse platforms and technologies (i.e., DLLs, REST and services built on the proposed middleware).

The “Context awareness potential” (Context) of an approach is given the value “L” if the context is modeled as another service (centralized) and is construed at the application-level without any advanced filtering support. The value “M” is given if the approach models the context as a centralized entity but offers advanced mechanisms for filtering and delivering only the required information at the target application. Finally, the value “H” is given if the approach treats context as both a centralized entity and as a distributed, hierarchical property and provides advanced mechanisms for filtering and for composing higher-level contextual information. Our approach not only provides support for the latter, but also, by allowing smart object state variables to be captured as asynchronous events, allows for very fine-grained object-local control when capturing contextual information.

The “Tools for services support” (Tools S.) of an approach is given the value “L” if the implementation does not provide any software tool for automating some aspects of the development of services. The value “M” is given if the approach provides tools for guiding the development of services but not for their deployment. Finally, the value “H” is given if the approach provides tools and guidance for developing and deploying services in the environment. Our approach not only provides tools for developing and deploying services for all the supported programming languages, but also supports the automatic update of the deployed services if a newer version is available.

Similarly, the “Tools for application support” (Tools A.) of an approach is given the value “L” if the implementation does not provide any means for automating and guiding aspects of the development of applications (i.e. the functionality of the AmI environment). The value “M” is given if the approach provides the former but does not provide support for the deployment of the implemented functionality. Finally, the value “H” is given if the approach

provides tools for both the development and the deployment of an application's functionality. Again, our approach supports both aspects by offering a graphical development environment for programming smart objects that provide the AmI environment's functionality and a graphical deployment tool that visualizes and simulates their interactions.

The "Simulation support" (Simulation) of an approach is given the value "L" if the implementation does not explicitly provide any means for simulating any aspect of the target AmI environment. The value "M" is given if the approach provides the means for only simulating the resources that are available in the environment but not the latter's functionality. Finally, the value "H" is given if the approach provides the means for simulating both the resources and the behavior of the AmI environment. In our approach, resources (services) can be simulated in the context of a smart object and the provided graphical deployment tool for smart objects visualizes and can simulate all or part of the interactions in the context of the environment.

The aforementioned scores for the discussed properties for all the approaches reviewed in section 2.1 can be seen in the table below.

Table 1 - Comparison of different approaches for creating AmI Environments and applications

	Heterog.	Abstract.	Compos.	Context	Tools S.	Tools A.	Simulation
MundoCore	M	L	L	L	L	L	L
AmICom	L	L	L	L	L	L	L
Hydra	M	L	L	M	M	H	L
LAICA	L	L	L	L	L	L	L
WSAMI	L	L	H	L	L	L	L
W2P	M	L	L	L	L	L	L
DoAmI	L	L	L	L	L	L	L
AMIGO	L	M	H	M	M	M	L
iDorm	L	H	H	M	L	L	L
MavHome	L	H	H	M	L	L	L
Gator Tech	L	M	H	M	M	M	M
O2S	L	M	H	L	L	L	L

Aura	L	M	L	L	L	M	L
Gaia	L	H	M	L	L	M	L
FedNet	L	M	M	L	M	H	L
PICO	L	M	L	L	L	L	L
One.World	L	M	L	M	L	L	L
UIO	L	L	L	L	L	L	L
CORTEX	L	H	L	H	L	M	L

Chapter 3
Service Middleware for Ambient Intelligence Environments

At a technical level, ambient Intelligence (AmI) environments are comprised of a large number of diverse heterogeneous computing systems that constantly interoperate in order to provide a unified and seamless user experience. In this context, the term computing systems refers collectively to hardware devices (e.g. sensors), hybrid software/hardware systems (e.g. human tracking systems), and pure software systems (e.g. financial services). The comprising computing systems are inevitably heterogeneous, since they cut through many different research domains with many different requirements, goals and capabilities.

Requiring all the constituent components to be exclusively developed using a universal set of software and hardware technologies is impractical due to performance, suitability, sustainability and, of course, reusability reasons. The choice of technologies for developing a specific computing system is the result of balancing all the tradeoffs that will allow the final product to perform acceptably (performance), to describe its inner workings in a natural way with respect to the functions it has to perform (suitability), to be easily modified and extended in the face of changing requirements (sustainability), and to be able to be developed faster and easier by incorporating relevant technologies and components that were previously developed (reusability).

Therefore, since requiring underlying technological uniformity is impractical, it is essential to provide different means for allowing all these diverse heterogeneous components to interoperate within the context of an AmI environment. Towards this direction, we addressed the issue by developing a software platform that lies between all the diverse computing systems and the applications that utilize them and provides a common consistent view and access methods of the former to the latter, regardless of their implementation technologies. This software platform is referred to as the Service Middleware for AmI environments.

Despite being an overloaded term, middleware is a commonly used word in the context of distributed computing systems. In general, a middleware is a set of programming libraries

and programs (services) that constitute an indivisible platform offering a comprehensive abstraction over the complexities and potential heterogeneity of the target problem domain. Enabling programs written in different programming languages to interoperate seamlessly is a key design goal in most middleware platforms. As mentioned before, this is especially important in an AmI environment, where the basic system is built using diverse technologies from diverse research fields that traditionally utilize different programming languages.

In this context, the proposed middleware for Services in AmI environments provides libraries and tools to enable software developers to create services with a true Object-Oriented Application Programming Interface (API). These services can be developed and used from any program written in any of the supported programming languages which are: (a) C++, (b) .NET languages, (c) Java, (d) Python, and (e) Flash/ActionScript. The middleware effectively allows services to be distributed across the network, hiding the details of network connections and data serialization from the programmers.

The service middleware ecosystem comprises of the following four software components: (a) software libraries for facilitating communications and service distribution, (b) a set of core services for providing discovery and deployment functionality for the services infrastructure, (c) development tools that facilitate the creation, usage and deployment of services, and (d) client programs for managing and monitoring the computers that are used for running services. The details of all the aforementioned components are discussed in the following sections.

Specifically, section 3.1 details the design goals for an approach to constitute a viable platform for programming AmI services. With respect to these goals, section 3.2 reviews the technologies that were evaluated for serving as the underlying communication technology of the proposed service middleware. Section 3.3 describes the architecture and design of the proposed approach. Sections 3.4, 3.5, 3.6 describe respectively the design and implementation of the language-specific libraries that enable the development of services, the design and implementation of the infrastructure servers that support the deployment of services, and the provided tools for guiding the development and deployment of services. Section 3.7 describes the usage of all the aforementioned software entities for developing services with the proposed approach. Finally, section 3.8 provides a summary and a

discussion on the application of the proposed approach for developing services for AmI environments.

3.1 Middleware Goals

The proposed middleware, in order to constitute a viable platform for developing Ambient Intelligence services, first of all, has to support many different programming languages. Ambient intelligence, being inherently an interdisciplinary domain, incorporates a variety of technologies spanning many different research fields, which utilize components written in many different languages. For example, computer vision algorithms and programs traditionally utilize the C/C++ programming languages, whereas computer reasoning algorithms primarily use Java or .NET languages. A multi-language environment, in this sense, is essential for enabling many different research fields to expose their individual technologies and allow access to them through a uniform, well-defined set of methods. For this reason, it was deemed essential for the middleware to target a wide range of popular programming languages.

Most widespread service-oriented middleware technologies primarily support a synchronous request-response type of communication. A program, in order to invoke a method provided by a remote service, sends a request to it and waits for the processing on the service's side to finish and send back a result. The blocking nature of remote method invocation mechanics ultimately resembles the semantics and behavior of local method invocations in the majority of computer programming languages. Whereas this communication method offers an intuitive programming abstraction for services and applications to give commands to remote services and query their state, it is not sufficient. In dynamic AmI environments, there is also the need for delivering asynchronous notifications to service clients for modelling either the occurrence of inherently asynchronous events or to inform about changes to their state. Both of these communication strategies – synchronous request/response and asynchronous event-based – were deemed essential for service developers.

Concerning the validity-checking of service method invocations, there are two approaches: dynamic and static checking. In general, for an invocation of a method that belongs to a

software entity (e.g. an object, program, module, component, etc.) to be valid: (a) the referenced method has to exist in the context of containing software entity, (b) the number and types of its arguments have to match the ones expected by the instantiation of the method, and (c) the handling of the produced result has to match the type produced by the invoked method. Therefore, in the context of service invocations, dynamic checking refers to the strategy where the validity of a method invocation is checked by the service that actually contains that method only when the latter is invoked. If the provided method or type is not correct, the executing service is responsible for notifying the issuer of the invocation about the error. On the other hand, static checking refers to the strategy where the validity of an invocation is checked *before* that method is actually invoked or even before the invocation is actually compiled to the final executable. Detecting such errors as early as possible is a better approach for both the issuer of an invocation and its host, since for the former it minimizes the distance between the creation and the appearance of a programming error and for the latter it offloads the overhead of validity-checking to its clients. For that reason, it was deemed essential for the proposed middleware to support static type checking for method invocations. Specifically, for the supported statically typed, compiled languages validity-checking should be performed at compile-time and for the supported dynamic languages at runtime by the issuer of the invocation, *before* actually issuing it.

Orthogonally to the aforementioned goals, it is essential for a service middleware to provide an intuitive and simple programming model for creating and invoking (consuming) services. An intuitive, well-known model to emulate for the definition and invocation of services is the one provided by object-oriented languages for defining and using objects. In this regard, service programmers should first define and publish the interface description of a service (i.e. the set of its supported methods) which could subsequently be implemented by means of implementing a class that contains these methods in any of the supported programming languages. The client of a service should be able to use it by means of using a local object that provides the same methods as the target service and acts as a proxy to the latter. Regardless of its implementation language, a service should be usable by any of the supported programming languages.

Moreover, the provision of an automated solution for deploying finished services and making them available to the whole infrastructure is also important for the middleware's viability as an AmI service platform. The great number and diversity of the services in such an environment makes manual deployment difficult and error-prone. If a particular service depends on several other services in order to function correctly, then those dependent services must be deployed before the first one. This situation becomes more complicated for wide and deep service dependency hierarchies. For that reason, the middleware should provide a deployment mechanism through which services are started automatically when another program depends on them.

Finally, the whole lifecycle of a service, from its development to its deployment phase, should be supported and guided by the middleware platform. The particular requirements that have to be considered in this case are: (a) sharing among programmers the latest versions of the specifications of the functionalities (methods) that the services support, (b) potentially generating language specific files that will facilitate the programming of services and eliminate the need for error-prone, extraneous program code (boilerplate code), (c) providing convenient means for configuring the parameters for the deployment of the services, and (d) providing convenient means for updating older implementations of services with newer ones.

The aforementioned goals have driven the design, architecture and implementation of the proposed middleware. In this regard, their specific implementations in the proposed approach are summarized below:

- Full support for creating and consuming services in five popular programming languages (C++, .NET languages, Java, Flash/ActionScript, Python)
- Synchronous request/response (client-to-service) and asynchronous event-based (service-to-client) communication
- Statically checked type-safe invocation of Service methods
- Object-oriented programming model for creating and consuming services
- Automated service deployment and failure recovery
- Tools for describing, managing, and configuring the deployment and update of services

Moreover, in the proposed middleware, the deployment of services follows a “lazy activation” scheme in addition to the traditional “eager activation” one when services are started manually. When a program tries to access a service method, then if the hosting service is not running, it is automatically deployed and the service method is invoked without any interruption. This mechanism is also used for failure recovery. When a service crashes unexpectedly, the next time another service or application tries to access its methods, it will be automatically restarted and the invoker will successfully obtain its results without being interrupted. In addition to automatic deployment, the middleware supports automatic installation and update of services on the machines on which they should be executed. Most of the times, services, especially the ones that control specialized hardware, need to run on specific machines within the AmI computer infrastructure. Also, a specific service may be required to run on multiple machines (e.g., for load-balancing or redundancy). Therefore, the automated installation and update functionality provided by the middleware further simplifies the deployment of services in these usage scenarios. Finally, part of the middleware ecosystem is also a set of graphical tools that simplify both the development and the deployment configurations, while also providing the means for monitoring and controlling the whole computing infrastructure.

3.2 Distributed Service Technologies

For achieving the aforementioned goals it was essential to base the implementation of the proposed service middleware on top of an existing, well-understood communication technology. Basing our implementation directly over the Internet Protocol stack (i.e., TCP/IP, UDP/IP), was not a viable solution since, on one hand, its low-level abstractions do not allow for low-complexity, intuitive and robust implementations of the required invocation abstractions and, on the other hand, being a well-understood subject, there are many high-quality architectures and implementations that would provide for a much better building block for implementing the required high-level abstractions and functionalities.

The available service technologies target the construction and communication requirements of many different kinds of distributed systems crosscutting a plethora of application domains. The aim for such a broad application domain, however, comes at the cost of

having to support a large number of communication primitives, resorting to complicated programming abstractions and ultimately failing to mitigate tedious and error-prone programming practices. They justifiably provide only the least common denominator in order to cater for a very large number of often-conflicting requirements across diverse distributed application domains. For this reason, it was essential for the proposed middleware to build upon and leverage one of the available generic middleware technologies in order to achieve its goals and mitigate the aforementioned issues.

Therefore, for the underlying communication technology to constitute a solid base for building the proposed middleware, we identified four basic functional requirements. With respect to the identified goals, a generic communication technology had to provide: (a) high-quality implementations for many popular programming languages, (b) support for both synchronous and asynchronous communication strategies, (c) statically-checked type-safe service invocations, and (d) high-level object-oriented abstractions for programming and using services. Apart from these functional requirements, a purely practical issue under consideration was also the license of the technology's implementations. Since we wanted full control of the proposed service middleware, the license of the underlying technology had to be permissive enough to allow the redistribution of the whole platform under a license of our choosing.

Based on these requirements, the five distributed service technologies that were evaluated are presented in the following subsections.

3.2.1 CORBA

The Common Object Request Broker Architecture (CORBA) [49] is a standard defined by the Object Management Group (OMG) that provides a stable model for distributed object-oriented systems. Specifically, CORBA provides a set of detailed specifications to enable distributed services, written in diverse programming languages and running under diverse operating systems, to interoperate seamlessly. The abstractions described in the CORBA Object Request Broker (ORB) specification, allow for the definition and usage of distributed services to be modeled as the definition and usage of objects in object-oriented programming languages. Moreover, the CORBA standard guarantees at-most-once

semantics for method invocations, meaning that a service implementation can safely assume that if its methods are invoked remotely, they will be actually executed at most once per invocation attempt. This is a subtle guarantee that simplifies immensely the design and implementation of services, by explicitly enforcing the semantics of the provided programming abstractions.

A service is defined by means of describing its methods and their return and argument types in CORBA's Interface Definition Language (IDL). Subsequently, the standard specifies mappings between the types appearing in the IDL description and types in the target programming languages that are used to implement or consume CORBA services. Conforming implementations are then required to provide a program, called IDL compiler, which reads IDL files and produces language-specific source files that support the implementation and/or usage of services as if they were local, memory objects. Additionally, the standard also defines a plethora of core services [123] to simplify the design and deployment of distributed infrastructures. Since CORBA initially did not support asynchronous communication style, the standard Notification and Event services [123] were specified for providing centralized support for publish/subscribe architectures. Subsequent revisions added support for asynchronous messages [124], however, most CORBA-compliant implementations do not support them.

There are many CORBA implementations both commercial and open source for many different programming languages. Some of the most popular and robust open source implementations with permissive licenses include: TAO CORBA [125], JacORB [126], Java standard org.OMG.CORBA library [127], omniORB [128], and IIOP.NET [129].

3.2.2 Web Services

Web services are widely used in modern distributed systems built on top of the World Wide Web. This technology defines a simple message exchange protocol to allow applications to communicate across the Web. Services themselves are realized in terms of well-defined XML documents [72], which in turn specify the messages that can be accepted by a conforming implementation. Instead of providing a specification that offers high-level, standardized language-specific constructs for mapping service interfaces and data types,

web service aware code only needs to be able to generate and process the exchanged XML documents. The Simple Object Access Protocol (SOAP) [63], that essentially constitutes the core of the Web services architecture, only defines the format of the exchanged messages, the marshaling rules for the data that appear in the messages, and a set of conventions for performing Remote Procedure Calls (RPC) [130] over HTTP [131].

Although there are implementations of web service technologies in most popular programming languages, since the standard does not provide any specifications for them, programming abstractions and type-checking of invocations are not consistent among different implementations – even if they target the same programming language. Therefore, despite the fact that a few implementations fully support static compile-time type-checking and provide object-oriented abstractions for programming and using services, in general, they only implement the SOAP protocol and do not offer any other tools or libraries for achieving higher-level abstractions.

Concerning the communication protocol, web services support only synchronous request/response message exchange. Support for asynchronous communication is rare and non-standard among implementations. Moreover, the standard does not specify any guarantees for the communication channel, which consequently, ultimately depends on the underlying transfer protocol (i.e., TCP/IP). Whereas CORBA guarantees at-most-once semantics for method invocations, programmers building web services need to ascertain that there will be no problem for their algorithm if a method is executed more than once per invocation. Moreover, deployment of web services is complicated by the fact that most implementations require the availability of web application servers (e.g., Apache, IIS, etc.), which are very large software systems requiring complex configuration and maintenance operations.

3.2.3 Ice

The Internet Communications Engine (Ice) [132] is a true object-based domain-independent middleware platform that derives its main architecture from CORBA, but tries to improve on it by eliminating its unnecessary complexity, by adding support for asynchronous method dispatch, by supporting automatic object persistence, and by providing interface

aggregation in addition to interface inheritance. Specifically, Ice has all the aforementioned characteristics of CORBA providing static type-checking and object-oriented abstractions for using and implementing services. It eliminates some of CORBA's underspecified and underused features that even some of the popular CORBA implementations do not support (e.g. string encoding negotiation). Additionally, it adds support for asynchronous communication and interface aggregation through which a distributed object can, at the same time, implement multiple interfaces and thus be controlled differently by different programs. Interface aggregation provides an intuitive way for supporting service versioning where old programs may continue using an older version of the current service interface (i.e., the set of the supported methods and types).

Moreover, Ice leverages the set of CORBA's core services, by providing an "object persistence" service and a "software patching service". The former automatically stores the state of services in a database using application-defined transaction boundaries and the latter provides a mechanism for distributing software updates to clients.

Finally, Ice provides implementations for creating and using services in many popular programming languages and computing platforms. Initially, Ice did not support Flash/ActionScript, however, it now supports all the languages that the proposed middleware targets. The only disadvantage of Ice compared to the available CORBA implementations is its license. While open source, its implementations are released under the GPL license [133], which means that any software using them is required to be released also under the GPL.

3.2.4 Thrift

Thrift [134], used extensively in Facebook[1], is a communication middleware that emphasizes simplicity and efficiency in the delivery and invocation of distributed services. Thrift enables the creation and consumption of distributed services in many different programming languages, including the languages supported by the proposed middleware. Through an interface definition language, similar to CORBA's IDL, Thrift effectively separates the description of a service from its actual implementation, while providing statically-checked

[1] https://www.facebook.com

object-oriented programming abstractions for using and implementing services in the supported languages. Additionally, through its interface definition language, Thrift supports explicit fine-grained versioning of a service's methods and types, even at the argument-level. For example, if a newer version of a service extends a method by adding one extra argument, new programs will be able to access the extra argument, while programs built using the older version will continue to work without any problem. This fine-grained versioning support is the most important advantage of this middleware compared to the other ones.

In this regard, Thrift defines and implements only the bare minimum for achieving basic lightweight method invocations over the network with explicit versioning support. It does not support asynchronous method dispatch and at-most-once invocation guarantees. Moreover, it does not allow service methods to obtain or return references to other services. A feature supported by both CORBA and Ice. These issues prevent Thrift from offering a semantically and syntactically complete object-oriented invocation model.

3.2.5 Etch

Etch [135], which was originally part of the Cisco Unified Application Environment, is a cross-platform, language- and transport-independent framework for building and consuming network services. Etch defines and implements a Network Service Description Language (NSDL) that separates the description of a service from its actual implementation in the target language. The processing of an NSDL service description to generate the referenced interfaces allows client code to implement and invoke distributed services using object-oriented programming abstractions with static type checking. However, like Thrift, Etch is not a complete object-based middleware, as it cannot use a service object as the return value or parameter of a method, and does not guarantee at-most-once invocations.

Nevertheless, it offers support for two-way communication between a service and its clients, and simplifies security management by enabling connection authorization directives to be specified in NDSL. By supporting two-way communication, Etch effectively provides both synchronous request/response and asynchronous communication strategies. The incorporation of security primitives directly into the definition of interfaces is this

middleware's strongest point since it greatly simplifies the subsequent implementation of well-defined security protocols among service infrastructures.

However, Etch is currently a work in progress. While it initially supported only two programming languages, Java and C#, it has now expanded to include C with plans for supporting more languages in the future.

3.2.6 Discussion

With respect to the aforementioned requirements, a score value is assigned to each of the reviewed middleware technologies that were discussed in the previous subsections. In general, the following symbols are used: "L" (Low) to signify minimum support for satisfying a requirement, "M" (Medium) to signify adequate support for satisfying a requirement, and "H" (High) to signify that the requirement is fully satisfied.

Specifically, for the "Multi-language support" (Languages) requirement, the value "L" is assigned if a middleware technology's implementations support less than three programming languages. The value "M" is assigned if they support three or four, and "H" for five or more languages.

For the "Communication strategy" (Comm.) requirement, the value "L" is assigned if a middleware technology only supports synchronous request/response type of service invocations. The value "M" is assigned if only the former is supported but the specification provides adequate primitives to implement asynchronous service-to-client communication using only the middleware's libraries. Finally, the value "H" is assigned if both synchronous and asynchronous communication strategies are supported.

For the "Programming abstractions" (Abstractions) requirement, the value "L" is assigned if a technology does not support object-oriented programming abstractions for implementing and invoking services. The value "M" is assigned if object-oriented programming abstractions are partially supported (see also 3.2.4 and 3.2.5). Finally, the value "H" is assigned if object-oriented programming abstractions are fully supported by the specification and the implementations.

For the “Static type checking” (Type-Check) requirement, the value “H” is assigned if the specification supports compile-time type-checking for compiled languages and type-checking before the invocation of a service method in dynamic languages. The value “L” is assigned if the specification allows the type-checking to be performed only when the invocation reaches the actual service method.

Finally, for the “Permissive license” (License) requirement, the value “H” is assigned if the technology’s implementations have permissive licenses allowing the programs that utilize them to be released under any license their author chooses. Conversely, the value “L” is assigned if they specifically dictate and enforce the license that should be used.

A summary of the evaluation of the reviewed service middleware technologies can be seen in Table 2 below.

Table 2 - Comparison of the reviewed service middleware technologies

	Languages	Comm.	Abstractions	Type-Check	License
CORBA	H	M	H	H	H
Web Services	H	L	L	L	H
Ice	H	H	H	H	L
Thrift	H	M	M	H	H
Etch	M	H	M	H	H

Based on the evaluation above and mainly due to Ice’s restrictive distribution license, the proposed service middleware was decided to be built on top of the CORBA standard, using the aforementioned open source implementations (3.2.1), supporting C++, Java, .NET, Python, and Flash/ActionScript.

In this regard, the proposed middleware uses a well-defined subset of the CORBA specification to define its architecture, functionality and abstractions while simultaneously providing its own service registration and discovery subsystem. In addition to that, it implements a custom subsystem for asynchronous communication. Native CORBA calls are used only for method invocations. This approach greatly simplifies the development of services compared with just using the CORBA libraries directly for the following reasons: (a)

the complexities of CORBA's Application Programming Interface (API) are completely hidden from the programmer, (b) in addition to request/response communication it adds support for asynchronous, event-based communication, (c) it provides bindings for implementing graphical tools that on the one hand simplify the generation of language-specific files, while on the other hand provide a unified, consistent method for file generation regardless of the target programming language, and (d) it provides bindings for supporting automatic deployment and update of services.

3.3 Design

In the context of an AmI environment, the functional role of services is to provide the means for implementing its sensitivity and responsiveness. Specifically, services enable programs to sense the changes happening within the environment and provide the means to affect it by explicitly altering its state. At the technical level, this is achieved by exposing all the computing resources within the environment as services with a well-defined interface. Ultimately, services implement a unified access layer for monitoring and controlling physical entities, sensors, and algorithms that operate within the environment and essentially characterize its nature and capabilities.

The term physical entities, describes those physical objects that are potentially augmented with microcontrollers and can report and alter aspects of their physical state. Those augmented physical entities are a higher-level concept than sensors, since they usually embed one or more sensors in order to be able to report their state and, at the same time, they can also contain one or more actuators to enable external programs to affect their state. For example a chair in an AmI environment may contain, in addition to multiple sensors for reporting its orientation, height, back angle and its user's weight, multiple actuators for changing its orientation, height and back angle. This sensing ability and functionality is exposed to the AmI environment as a chair service that provides methods to query and affect the state of the chair, and asynchronous events to report changes to its state (external or internal).

The term sensors, which is a lower-level concept than physical entities, refers to those components that are not essentially conceptually tied to a physical object but are able to

sense some aspects of the environment. The term may refer to physical sensors[2], such as temperature sensors, distance from a fixed point sensors, etc, or to mainly algorithmic sensors such as human position tracking through multiple cameras, voice identification through one or more microphones, etc. This sensing ability is exposed to other programs as a service that contains methods to query the current state of the sensor and potentially configure parameters that affect its sensing functions, and asynchronous events to report new sensor values.

Finally, the term algorithms, refers to those entities that can neither be classified as physical things nor sensors. Although they are usually pure software constructs, they can utilize sensors, actuators or physical entities in order to deliver their functionality. The proposed classification, therefore, only concerns the perceived functionality of a resource and not its implementation details. For example a pure software algorithm resource, in the context of an AmI environment, could be a file-system service or a user profile service. On the other hand, a personal assistant service may use multiple sensors that are spread over a room to follow the user around and provide context-sensitive information. In this case services modeling algorithms contain methods for giving commands and configuring aspects of the functionality of the algorithm, and asynchronous events for the algorithm to notify that its operation has reached some well-defined checkpoints.

Table 3 - Classification and modeling of the perceived functionality of resources in an AmI environment

	Service Methods	Service Events
Physical things	Query state, change state	Report changes to state
Sensors	Configure sensing parameters	Report new sensed values
Algorithms	Send commands, Configure behavior	Report reaching a checkpoint

Apart from their perceived functionality, services modeling resources (physical things, sensors, algorithms), depending on their implementation dependencies can be characterized as primitive or high-level. High-level are those services that use (consume) other middleware services in order to deliver their functionality. At the other end, primitive are those services that do not use any other middleware service and either fully implement

[2] http://en.wikipedia.org/wiki/List_of_sensors

their behavior in a self-contained program or use other software or hardware components in order to deliver their functionality. In this regard, the only differentiating factor is whether services utilize other services within the context of the proposed middleware. If a service accesses some functionality through a different middleware technology, e.g. Google search through REST[3], but does not use any services exposed under the proposed middleware, it is classified as a primitive service.

Primitive services, therefore, utilize the functionality offered by the underlying software and hardware components through custom protocols. In this regard, the "custom linkage" in Figure 2 refers to any possible access method, through which a specific service can obtain and transmit information, e.g., linking with another software library, connecting through Bluetooth, connecting through HTTP, etc. Figure 2 also shows an example of a runtime hierarchy of two high-level and four primitive services. The gray rectangle that contains all the services denotes the entities (i.e., services) that are exposed by the middleware and can be accessed by other middleware-aware programs. Apparently, all other components are completely hidden as far as the middleware infrastructure is concerned.

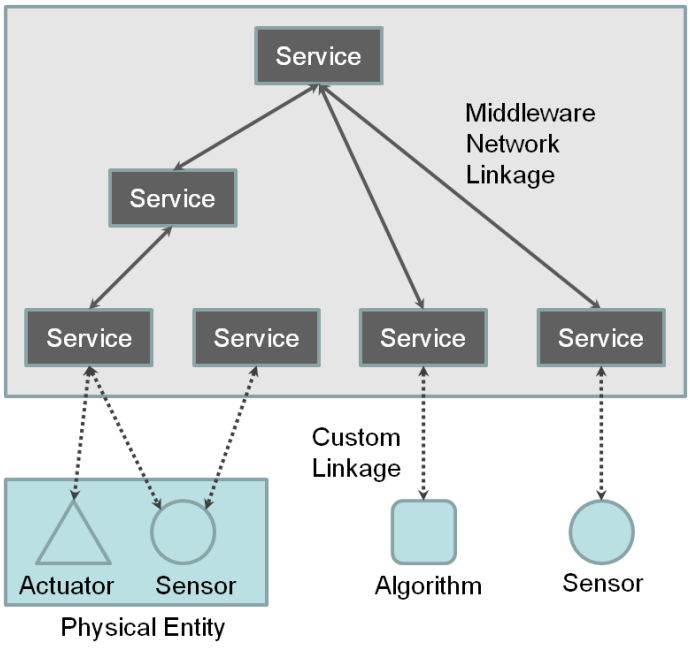


Figure 2 - Runtime architectural dependencies among resources and services

In this regard, services ultimately model the slow-changing domain knowledge providing systematic well-defined means to describe what the AmI environment is. From a software

[3] https://www.googleapis.com/customsearch/v1

engineering point of view, the description of the available services in the context of the environment, fully describe its nature and capabilities. The nature and capabilities of an environment are slow-changing ipso facto compared to the rapidly-changing way they are utilized, hence, the characterization above. This observation also drove the design of the proposed middleware to incorporate object-oriented programming abstractions with compile-time, statically-checked method definitions and invocations.

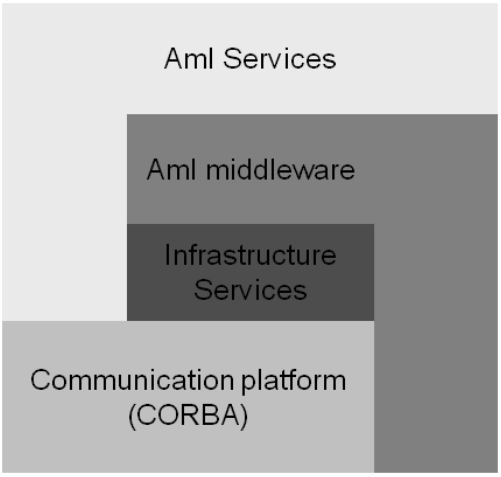


Figure 3 - Proposed middleware's layered architecture

The layered architecture of the proposed middleware can be seen in Figure 3. Implemented services use the CORBA libraries directly only for method invocations. Registration of services, obtaining a reference to a service and the implementation and management of events are provided through the proposed middleware. This functionality is supported by the infrastructure services that are an indistinguishable part of the framework. These infrastructure services are only used at a service's startup and shutdown phases and are not utilized at all during the invocation of methods and the delivery of events. Moreover, infrastructure services can be redundant, thus increasing the robustness of the infrastructure and eliminating potential single points of failure. These layers and their implementations are discussed throughout the following sections.

3.4 Libraries for Programming Services

For the reasons explained in section 3.2, the middleware's libraries for each of the supported languages are implemented on top of the Common Request Broker Architecture (CORBA). They fully utilize the abstractions, communication protocols, runtime behavior and code generation that is specified by the standard. In general, the goal for the communication libraries was to accentuate CORBA's strengths and hide its weaknesses while, at the same time, making the creation and invocation of services much easier and less error-prone.

These libraries are implemented multiple times – one for each supported programming language. Depending on the target language, a different open source CORBA implementation is used as the implementation base. Specifically: (1) the C++ implementation uses TAO CORBA [125]; (2) the Python implementation uses omniORB [128]; (3) the Java implementation uses the default CORBA runtime that is bundled with the standard Java libraries [45]; (4) the .NET implementation uses IIOP.Net [129]; and lastly, (5) the Flash/ActionScript implementation is built on top of Flash player's Microsoft Windows COM object in C# utilizing the .NET version of the middleware's libraries. Owing to the detailed specifications provided by CORBA, every language-specific implementation is interoperable with every other, allowing the proposed middleware to provide a consistent interface among all these different programming languages.

Ultimately, the provided libraries aim to eliminate the tedious, repetitive and error-prone programming tasks that are enforced by the CORBA specification for creating and using distributed objects. CORBA requires such fine-grained programming constructs since it has to provide many different communication control interfaces and primitives for catering for many diverse application domains. Since the proposed middleware has a much narrower usage scope, the underlying communication primitives and complicated interfaces can be effectively hidden from the programmer, and so can the complexities associated with CORBA programming. Moreover, inconsistent and controversial features, such as *unions*, are completely left out from the libraries and their exposed interfaces.

The following subsections describe the implementation of asynchronous peer-to-peer events on top of CORBA and the conventions, functionality and Application Programming Interface (API) for: (a) implementing and exposing services, (b) obtaining a reference to a service that has been previously exposed, and (c) invoking service methods and obtaining events.

3.4.1 Asynchronous Events

The proposed middleware extends CORBA's communication model by implementing asynchronous, event-based communication on top of CORBA's blocking request/response model. Legacy CORBA clients can receive asynchronous notifications, but this is only possible through the use of a third centralized entity – the other two being the client and the target remote object – that plays the role of a publisher/subscriber registry and message dispatcher. There are two such standard services described in the CORBA specification: the event service and the notification service [123]. We decided, however, not to use any of these standard services since they would enforce a constant dependence in a centralized entity on all deployed services, which ultimately hinders both the reliability and the performance of the AmI service infrastructure.

In the proposed middleware, asynchronous messages can be produced and dispatched by any service and delivered asynchronously to any program that has declared interest in them. Through this mechanism, the producer of an asynchronous message directly propagates it to all its consumers without the need of a centralized entity. This asynchronous type of communication begins when a message (event) is emitted by a service. On the other side, another program that had previously declared interest in that specific kind of event produced by the service is notified asynchronously and the event with its payload (i.e. its data contents) is finally delivered to its "event handler" method.

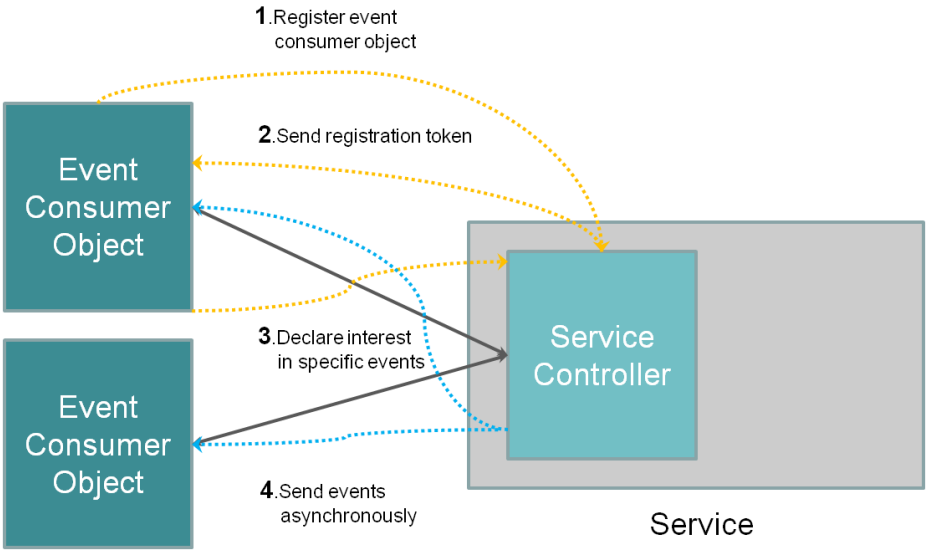


Figure 4 - Event registration and dispatching process

The participating components and the process through which the aforementioned functionality is realized, is shown in Figure 4. The service controller object, which is present in all services implemented on top of the proposed middleware, is the entry point for client programs to declare interest in the events that can be emitted. Moreover, service controller's responsibilities include dispatching of events, responding to livability checks and providing metadata associated with the containing service instance. On the other side, the event consumer object provides an interface for client programs to request specific events that are subsequently delivered to a specific program method that they implement. Initially, upon connecting to a service, the event consumer object registers itself in the service's service controller and obtains a registration token that is used to manage its event subscriptions. By registering their methods through the event consumer object's API, client programs cause the latter to declare interest in the referenced events, which upon emission from the service are delivered to the corresponding program methods. In Figure 4 above, the yellow checked arrow denotes a CORBA method invocation (blocking request/response), while the blue checked arrow denotes the emission and asynchronous delivery of an event.

3.4.2 Service Implementation

The specification of service methods and events appears in the interface definition text of the service. As shown in Figure 5, a service description, essentially comprises of the definition of the latter's methods, events, and the non-primitive types that appear as

method arguments or event payloads. The syntax of the interface description text is dictated by the CORBA IDL specification [49], however only the primitive types that appear in Table 4 are allowed. Complex types can be defined through the use of the *struct*, *enum*, *typedef*, *sequence*, and *interface* keywords. Those keywords are used for defining typed tuples with named members, enumerations, type aliases, sequences of types, and other objects with their own methods, respectively.

```
#include <ami.idl>

module Cliches {
    interface Hello {

        // Types
        //
        enum Feeling { SAD, NEUTRAL, HAPPY };

        // Methods
        //
        wstring SayHello (in wstring msg, in Feeling f);

        // Events
        //
        void Event_SaidHello (in wstring msg, in Feeling f);
    };
};
```

Figure 5 - Description of a service using IDL

When specifying a service description in the proposed middleware, the following conventions are used: (1) the first non-comment line of the description has to be the statement "#include <ami.idl>"; (2) the interface of the service that contains its types methods and events, has to be contained within a module of arbitrary name, which essentially denotes a namespace for grouping services; and (3) the supported events have to be represented as void methods, whose name begins with the prefix "Event_" followed by the name of the event and the method's arguments that denote the payload of the current event.

Table 4 - Primitive types for service interfaces

Primitive Type	Primitive Sequence	Description
boolean	BooleanSeq	Boolean true, false values
octet	OctetSeq	Raw bytes
char	CharSeq	Characters (8-bit)
wchar_t	WCharSeq	Characters, wide (16-bit)

float	FloatSeq	Floating point values (32-bit)
double	DoubleSeq	Double precision floating point values (64-bit)
long	LongSeq	Signed integer values (usually 32-bit)
long long	LongLongSeq	Long integer values (usually 64-bit)
string	StringSeq	ASCII-encoded strings
wstring	WStringSeq	UNICODE strings

Following the aforementioned description and generation of the language-specific code (see section 3.6) the specified service can be fully implemented in any of the supported programming languages. The functionality of a service is provided by the implementation of its methods, which is supported by means of implementing a class that derives from the declared service interface and implements it. Despite the fact that in most of the supported languages (C++, Java, .NET) the service implementation has to also define the body of the event methods, the required language statements are straightforward and convenience functions are provided by the proposed framework. A full implementation of the service described in Figure 5 using the C# programming language, can be seen in Figure 6. In this case, the implementation of an event method constitutes of a single statement that invokes a middleware function with the name of the event and its payload. The libraries of the proposed middleware are tentatively named “Famine” or “FAmINE”, which stands for FORTH[4] AmI Network Environment, hence the naming of the functions in the examples below.

After its implementation, a service can be exposed. In this context, a service is exposed if other programs that are implemented on top of the proposed middleware can successfully locate it, invoke its methods and receive its events. A service is exposed by (a) instantiating an object that provides an implementation of the service, and (b) invoking the “RegisterService” function that is provided by the proposed middleware. This process can be seen in Figure 6, which also shows all the invocations to the middleware’s libraries that are needed throughout the lifetime of the services implemented and exposed in the context of an executable program.

[4] http://www.ics.forth.gr

```
using Cliches.Hello_package;

public class HelloService : MarshalByRefObject, Cliches.Hello {
    public string SayHello (string msg, Feeling f)
    {
        this.Event_SaidHello(msg, f);
        return "Hello from C# (" + msg + ")" + ", " + f;
    }
    public void Event_SaidHello(string msg, Feeling f)
    {
        Ami.Famine.SendEvent(this, "SaidHello", msg, f);
    }
};

class Program {
    public static void Main(string[] args)
    {
        Ami.Famine.Initialize(args);

        var service = new HelloService();
        Ami.Famine.RegisterService<Cliches.Hello>(service);

        // Show UI or block if console application (Ami.Famine.Block())

        Ami.Famine.UnregisterService<Cliches.Hello>(service);
        Ami.Famine.CleanUp();
    }
}
```

Figure 6 - Implementing and exposing a service in C#

The aforementioned process for implementing and exposing a service is similar in the supported dynamic languages (Python and Flash/ActionScript). However, in this case the provision of an implementation body for event methods is not required. The event methods are automatically instantiated by the framework. In this regard, Figure 7 shows the full implementation and exposure of the same service in the Python programming language.

```
import Cliches, Cliches__POA

class HelloService(Cliches__POA.Hello):
    def SayHello (self, msg, feeling):
        self.Event_SaidHello(msg, feeling) # Auto-generated by the framework
        return "Hello from Python (" + msg + ")" + ", " + feeling

def main():
    ami.Famine.initialize()

    service = HelloService()
    ami.Famine.registerService(service)

    # Serve requests or use the non-blocking runOneRound() in external main loop
    #
    ami.Famine.run()

    ami.Famine.unregisterService(service)
    ami.Famine.cleanUp()

if __name__ == '__main__':
    main()
```

Figure 7 - Implementing and exposing a service through Python

3.4.3 Service Discovery and Invocation

Following the exposure of a service (see section 3.4.2), any program using the libraries of the proposed middleware can obtain a reference to it, invoke its methods and receive asynchronously its events. Obtaining a reference to an exposed service is possible through the “ResolveService” function. The function returns a reference through which the program can invoke the service’s methods and accepts an event handler object through which is notified about the occurrence of service events. In all supported languages, except C++, the event handler object implicitly declares interest in specific events by declaring methods with the “EventHandler_” prefix. A void method with the name “EventHandler_EventName” implicitly declares interest to an event with name “EventName” and is invoked asynchronously every time the “EventName” event is emitted by the resolved service. This can be seen in Figure 8, where a Java program resolves the service described in the previous section and captures its event through a separate event handler object. The only difference with event handler objects in C++, as can be seen in Figure 9, is the explicit registration of the object’s methods as handlers to the resolved service’s events.

```
import Cliches.HelloPackage.*;

class HandlerObject {
    public void EventHandler_SaidHello(ami.LocalEvent e)
    {
        System.out.println("Received: " + e.getNextArgString() +
            e.getNextArg(Feeling.class));
    }
}
class Program {
    public static void main(String[] args)
    {
        ami.Famine.initialize(args);

        Cliches.Hello servRef = ami.Famine.resolveService(Cliches.Hello.class,
            new HandlerObject());

        String result = servRef.SayHello("Hi!", Feeling.SAD);

        ami.Famine.disposeResolvedService(servRef);
        ami.Famine.cleanUp();
    }
}
```

Figure 8 - Using an exposed service in Java

Four core properties of the service resolution process and its semantics that are an important aspect of the design of the proposed middleware are the following: (1) service resolution always succeeds, as long as the type of the service is valid; (2) if a method of a

resolved service is invoked and the service is not running, then if possible the service is automatically deployed; (3) if at some point a service crashes, when the client program invokes one of its methods, then if possible the service is automatically deployed; and (4) a client program may be started before the services it uses and declare interest to their events; then, when these services are started, either automatically or manually, their events are properly registered and delivered to the client program. These properties are implemented through the use of the infrastructure services, which are an integral part of the proposed platform. Hence, the aforementioned properties and their implementation strategies will be discussed in the next section.

```
#include "Famine.h"
#include "ClicheHelloC.h"

class HandlerObject : public ami::ILocalEventHandler {
public:
    EventHandlers(void)
    {
        AddHandler("SaidHello",
            ami::HandlerFunc(this, &EventHandlers::EventHandler_SaidHello));
    }

    void EventHandler_SaidHello (const ami::LocalEvent& e)
    {
        const wchar_t* msg;
        Cliche::Hello::Feeling f;
        e.GetNextArg(msg).GetNextArg(f);
        std::cout << msg << ", " << f << std::endl;
    }
};
```

Figure 9 - Event handler object in C++

3.5 Infrastructure Servers

The proposed middleware implements two centralized services that constitute an integral part of the runtime dependencies of all services that utilize the middleware. These services are the Repository service, and the Directory service. Since these services are centralized and an indivisible part of the infrastructure, they will be referred to as servers throughout this section in order to distinguish them from the middleware services that utilize their functionality. The repository server keeps track of whether a service is running and where it is running or where it should have been running, if it is currently inactive. The directory server, as long as a service is running, keeps information about its network location (IP address and port), as well as the locations of its Service Controller object (3.4.1). Through

these two centralized servers, the proposed middleware supports network location and relocation transparency, lazy service-dependency resolution, automated service deployment and automated service recovery.

In an ideal usage scenario where there are not any failures in any of the active services, in the context of a specific service, these two servers are contacted only twice during its whole lifetime; one time when the service is initially deployed and one time when it is disposed. All communications between services from that point on are purely peer-to-peer. Therefore despite being centralized, they do not constitute a bottleneck for the whole service infrastructure. Additionally, these servers support redundant operation, which on one hand expands even more the scalability potential of the middleware and, on the other hand, greatly increases its robustness. Since there can be more than one instance of each of these servers running on different machines, when a service is unable to contact one instance, it automatically tries to contact the other ones before it gives up. Therefore, overall, these servers neither constitute a single point of failure nor do they hinder the scalability of the service ecosystem.

In addition to being contacted only during the deployment and disposal phase of services, these servers implement persistent state. An unexpected (or expected) termination of a server, first of all does not disturb the ongoing communications in the infrastructure, and secondly, when the former is redeployed, its previous state is reloaded intact and subsequent requests by services are fully handled consistently with respect to the state of the infrastructure.

3.5.1 Repository Server

The attempt of a client program to invoke a service method is initially intercepted by the repository server that subsequently checks whether the requested service is running and redirects the interested client to the service's actual network location. If the requested service is not currently running, either because it had not been started at all or because it had exited prematurely, the server requests its deployment prior to redirecting the client. The redirection is implemented through the CORBA's message forwarding mechanism [49], and is very efficient, since, once the client resolves the service, subsequent invocations from

the same client are directed to the cached actual network address and need not be intercepted again by the server. Only when the service becomes unexpectedly inaccessible from the point of view of its client, the latter falls back to resolving the service again through the Service Redirection server. In this case, the resolution process starts again from the beginning, and if the server determines that the service is not running, it attempts to activate it once more. Because of this failback mechanism and the fact that it is always available to the infrastructure, a client program may contact the servers more than two times.

This process, including the failback procedures, is performed automatically by the middleware when necessary. The client program just needs to resolve the service at the beginning in order to obtain a reference to it and from then on it can be used as if it will always be accessible. After all, if it becomes inaccessible the repository server will restart it transparently. The client program will only notice an abnormally long delay during its usual service invocations.

For the repository server to be able to start or restart a service: (a) there has to be another server, called *Service Activator*, or just Activator, running on the machine where that a specific service is supposed to run, and (b) the repository server has to be configured a priori to know on which machine a specific service is supposed to run. The tool for mapping the deployment of services to specific machines is described in the following section.

The activator server is an optional component of the middleware infrastructure. If it is available, it registers itself with the infrastructure's repository server under the name of the machine on which it is running. In this regard, it plays the role of the machine's entry point and, hence, there can be at most one activator instance per machine. Aside from the set of activators, the repository server keeps a configuration database that specifies on which machine a specific service should run and what operating system command starts the service executable in the context of the target machine. If a service is started manually, it notifies the repository server of its availability and subsequent invocation requests by client programs obtain this specific instance directly. If, however, (1) the service is not deployed, (2) there is a configuration entry in the repository server's configuration database that describes how to start that specific service, and (3) there is an activator running on the

target machine, then the repository server proceeds by contacting the activator to deploy the service on its behalf. If any of these steps fails, the client program receives an exception.

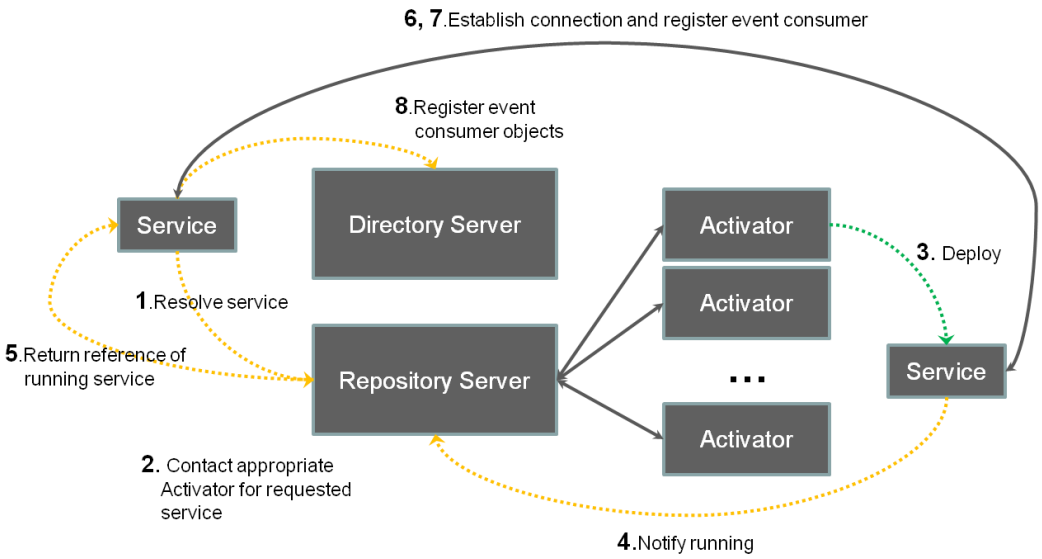


Figure 10 - The process for resolving a service to access its methods and events

This process is described in detail in Figure 10 where a service tries to resolve another service, which is not currently running. In this figure, yellow checked arrows denote a remote method invocation, the gray arrow denotes a permanent middleware connection with a service, and the green checked arrow denotes a system call (e.g., fork) in the context of a specific machine.

Moreover, the activator, prior to deploying a specific service, performs the following steps:

- Checks if the executable program of the specified service is accessible in the context of the local machine
- If the executable is not available it checks if it is available at a central file repository that is part of the middleware infrastructure and downloads it on the target machine
    - If it is not available it reports failure
- If the executable is available on the local machine, it checks if there is a newer version at the central file repository
    - If there is a newer version it downloads it and replaces the older one
- Finally, it tries to deploy the newest version of the service
    - If the deployment fails it reports failure

Through this process, all aspects of the deployment of services, including their update, can be easily controlled and tested from the programmer's computer. In addition to that, the deployment configuration process is further simplified by the provision of a graphical tool that is described in the following section.

Moreover, for more robust monitoring, control and automation capabilities over a large ambient intelligence environment, activators also offer methods to (a) turn on/off machines that are within the same subnet, through the Wake-On-Lan (WOL) network card function, (b) to start/stop and get the output of commands that are executed on the target machine, and (c) to obtain information about the characteristics of the target machine – such as operating system, processor, RAM, etc. Furthermore, it sends events to asynchronously notify interested clients about changes in the machine's state and the activation or deactivation, premature or intentional, of its hosted services.

3.5.2 Directory Server

The Directory server is an implementation of CORBA's naming service [123] and is used by the middleware to keep track of the event consumer objects and the service controller objects in the context of individual services (see Figure 4), and also to keep track of all the deployed services and their runtime service-dependencies. Whereas the set of the already deployed services can also be obtained from the repository server, the directory server in that respect contains additional information so that client programs can inspect and determine the runtime dependencies of all the deployed services. E.g., a client program can determine for a specific service which other services it invokes and receives events from.

Client programs only contact the directory server during their deployment and disposal phases in order to update its bookkeeping data. During this process a service registers its own service controller object and registers interest in the events that another service is offering. If the other service is already active, then the first service obtains the former's service controller in order to register its consumer object. On the other hand if the target service is not currently running, as soon as it is deployed, it finds out which services are interested in its events and subsequently prompts them to register their consumer objects. This is essential for enabling commutative manual deployment of interdependent services,

since the consumer of events can be deployed before their producer. The steps performed by this process can be seen in Figure 11 where a service, which is an event producer, is deployed manually and finds out that another service is interested in its events.

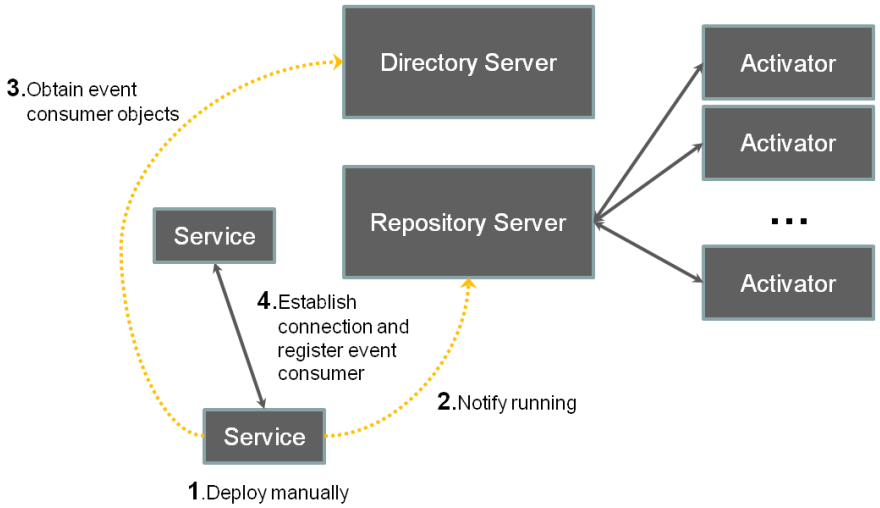


Figure 11 - Manual deployment of a service

When a client program terminates prematurely without performing its disposal phase and never recovers, the data kept by the directory server may become inconsistent. In this case, it is the responsibility of the service producing the events to clear the inconsistent data from the directory server as soon as it becomes aware of them.

3.5.3 Context and Zones

As indicated in the previous sections, a service can be resolved, i.e. discovered and being connected to, regardless of its actual network location. Only its type (that is directly associated to a language-defined type) and a free-form alphanumeric sequence need be known. The latter is referred to as the service's "runtime context" or just "context". A service's type signifies its functional role and strictly defines the set of methods, events and data-types it supports (see 3.4). All these information are described at design-time and are specified in the service's interface definition. On the other hand, a service's context is a runtime, dynamic property of a service that describes and identifies its current instantiation within an Ambient Intelligence environment. This is necessary since a service of a specific type may be instantiated multiple times within the environment either for indicating the availability of alternative implementations, for providing access to different sets of

resources, or even for implementing application-level redundancy. For example, a service of type *RoomLightsController*, which models the physical entity "lights" (see 3.3) and is responsible for controlling the lights within a specific room, has to be instantiated multiple times in an environment – once for every room that contains controllable lights. In this case, the context of the aforementioned *RoomLightsController* service can be used to uniquely identify its different instantiations. Consequently, the value of each instantiation's context can contain the name of the room that this specific instantiation controls – e.g., "kitchen", "bedroom", etc.

For simplicity we omitted the use of context for exposing and resolving services in sections 3.4.2 and 3.4.3 respectively. There is a separate version (overloading) of the middleware functions "RegisterService" and "ResolveService" that take as an argument the name of the context. If the latter is not provided, as in the previous examples, then it can be set by a command-line switch during the invocation of the service executable, or by a configuration parameter. If neither is supplied, it defaults to the name of the machine that hosts the service.

The network locations of the two servers are provided in the configuration document that specifies the different deployment zones. In this context, a zone can be seen as an isolated middleware infrastructure where services deployed in a specific zone cannot access and affect the services deployed in another zone. The primary motivation for defining the zone concept was to be able deploy and test services in a "development" zone without disrupting the infrastructure running under the "release" zone. A service can choose its deployment zone either through a command-line switch when invoking its executable, or through a configuration parameter. If neither is supplied, it defaults to the "development" zone.

The configuration mechanism mentioned above is used to provide easily modifiable, default, service-specific runtime values for the deployment of services. When a program built on top of the proposed middleware is executed, it loads the global configuration parameters that are specified in a file installed along with the middleware's libraries. Subsequently, these values are merged and replaced by the values of a configuration file in the same directory as the executable – if such a configuration file exists. The resulting values are merged and replaced by the values given as command-line parameters. Lastly, these final values are

supplied to the middleware's library for configuring the relevant aspects of the deployment of the services that are exposed by the current executable program.

3.6 Development and Deployment Tools

An important component of the proposed middleware is the service creation and deployment configuration tool-suite, called Idlematic, which supports and streamlines all the phases of the creation and deployment of a service – from the definition of its methods and events to its actual execution on the target machine. Idlematic, which keeps and exposes all the definitions of all the available service interfaces categorized by their role and their type, provides an interface description editor, a set of language specific code generators for each of the supported languages, and a service deployment configuration panel that describes the invocation parameters of a service on a specific machine.

Figure 12 shows the different visual components of Idlematic's interface with the tabbed interface for accessing the provided tools (1), the service repository for accessing the latest version of the available service descriptions (2), the service interface description editor (3), and the diagnostics console for displaying messages during the usage of the tool (4).

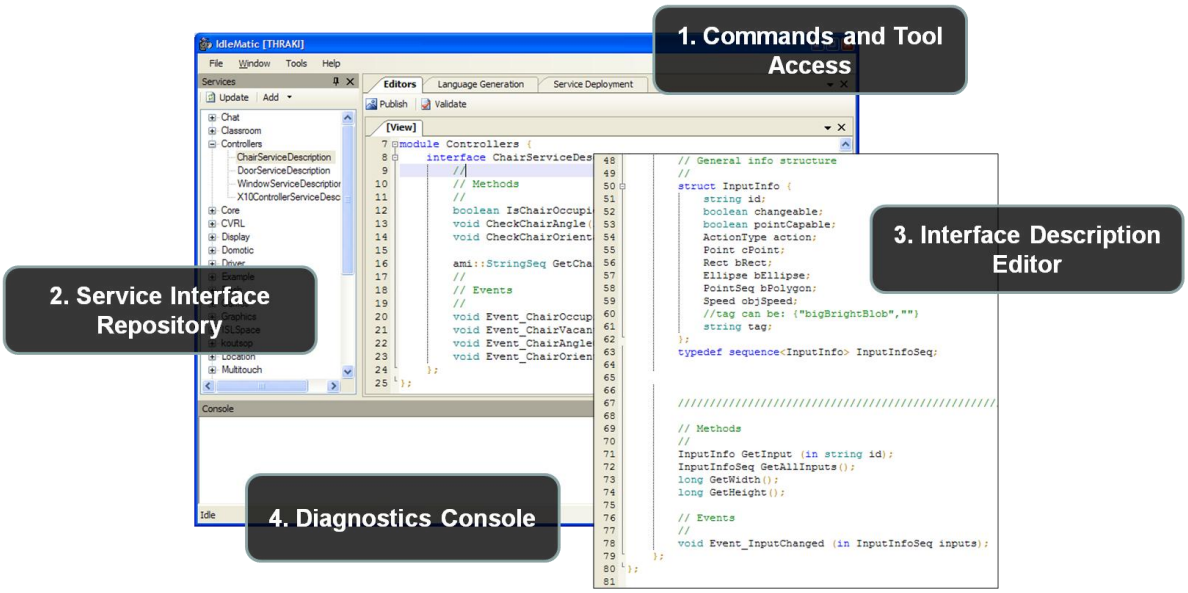


Figure 12 – Idlematic's UI with the service interface definition editor

The interface description editor offers lexical highlighting and keyword-completion functionality, assisting the programmer to fully define a service description. A service's

methods, the specialized data-types that it accepts as method arguments and the events it can emit, are described in the Interface Definition Language (IDL), which is defined by the CORBA standard [49]. The service authoring and bookkeeping process is further simplified by the automated publication and archival of the service description document in a central, version-controlled repository. Therefore, authors do not have to keep track of the individual files that comprise the definitions of their service APIs. All these definitions are automatically synchronized, as soon as they are published and the latest version is always accessible to the author through the tool's interface, as seen in Figure 12. The same figure, apart from the list of all the available services on the left side, also shows the interface definition editor in the center, the "publish" and "validate" buttons over the editor that respectively save the current interface description text and validate it for potential errors, and the "console" below the editor that displays any messages, warnings or errors concerning the actions performed through the tool.

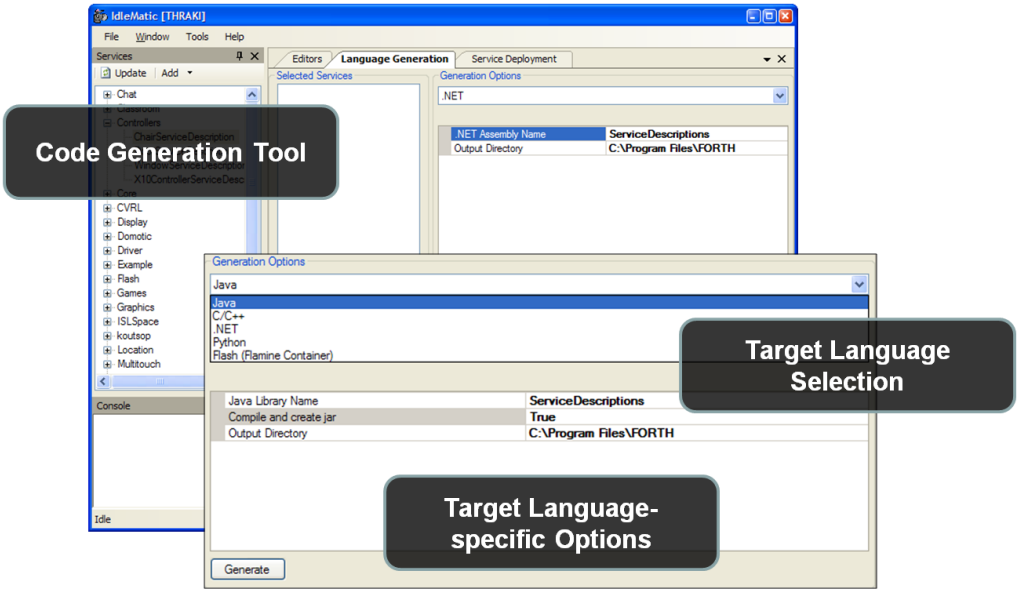


Figure 13 – Idlematic's code generation tool

In order to program the actual functionality of a service in one of the supported languages, or to use an already implemented service, the service's interface definition text has to be translated, or more precisely *mapped*, to the target programming language. This mapping, apparently, depends on the capabilities, features and conventions of the target language and it effectively enables the intuitive manipulation of the service in that language. It also allows for the type-safe implementation or invocation of a service – allowing the target

language's compiler or interpreter to type-check and point out potential errors as early as possible. The mapping process, which utilizes the interface definition language compilers of the different CORBA implementations that are used in the middleware, generates one or more language-specific files or compiled libraries that can be subsequently imported in the program that either implements or invokes the service. Although the tool uses the already available CORBA IDL compilers for code generation, it adds an extra layer of generation and checking in order to handle more effectively the concept of events, which is absent from the CORBA standard, and to enforce the specific conventions that are imposed by the proposed middleware.

The code generation tool can be seen in Figure 13, where on the left side there is the list of the selected services which the programmer needs to act upon, and on the right side the language selection list and the language-specific generation options directly below. For each selected service, the user of the tool can select whether the code generation concerns its implementation or invocation.

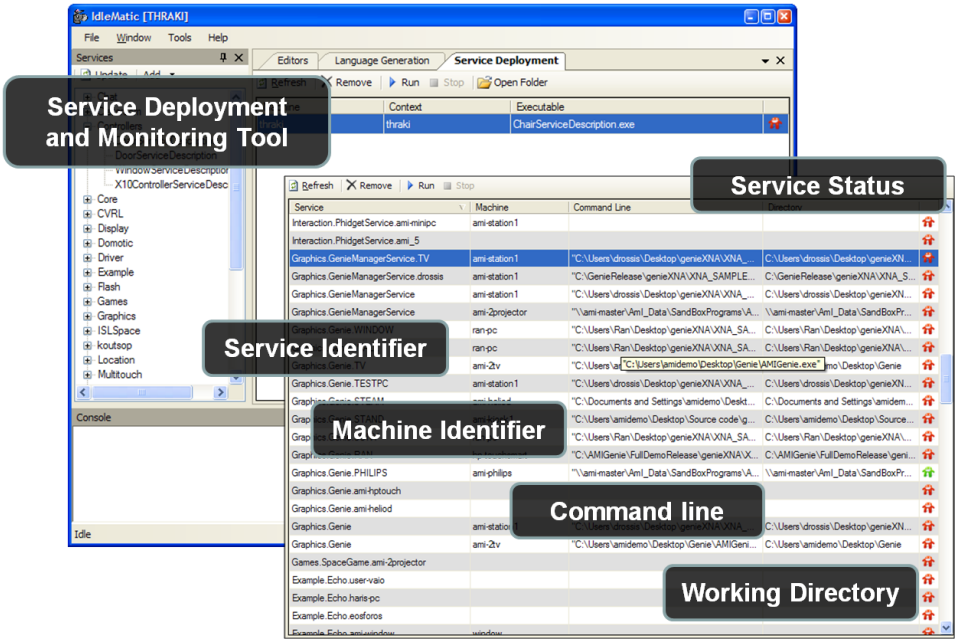


Figure 14 – Idlematic's service deployment and monitoring tool

Through the service deployment configuration tool, users can set up all the relevant parameters for enabling the automatic invocation of a service executable at a specific machine. As discussed in the previous section, this tool directly updates the database of the repository server in order to enable the latter to find out, when needed, on which machine

a service of a specific type with a specific runtime context is supposed to run. Specifically, the information needed for enabling the automatic execution of a specific service is: (1) the runtime context of the service's instance, (2) the name of the machine on which the service must be executed, and (3) the command that starts the service – i.e. the actual executable and its optional command-line parameters. All these input fields for the selected service can be seen in Figure 14 that also displays the status icon at the right side of each service instance, indicating whether the latter is running (green icon) or not (red icon). Users can test if the supplied information is correct by directly starting the service from the deployment tool, using the "Run" button over the main panel.

After the parameters for deploying a service instance are specified through the deployment tool and into the repository server's database, the latter, if a requested service instance is not running contacts the service activator that runs on the target machine. Thus, for the automatic deployment to work, the service activator has to be installed on all the machines that are meant to host one or more services. If, however, it is not available on the target machine, these specific services can be installed and invoked manually, without affecting the automatic deployment of other services that run on other machines. Moreover, the service that the activator needs to deploy does not even have to be installed at the target machine. The latest version of the executable, along with its dependencies, just needs to be placed under a central file repository and the activator automatically installs or updates it before invoking its main executable (see 3.5.1).

3.7 Example Service Implementation

Putting all the aforementioned mechanisms together, in this section, we will outline through an example the whole process of implementing and using a service in the proposed middleware. The example showcases the different communication options that a middleware-based service can utilize through its object-oriented API, and how the service's clients can use them. This example is complete. It does not omit portions of the program's text to make it more readable and clear. It is our goal, after all, to make the creation and usage of services as clear as possible with respect to each language's capabilities and conventions. Essentially, this can only be reflected in the example as long as it is complete.

Furthermore, we present different steps of the example in different programming languages that are supported by the middleware, since this provides an efficient way to showcase the different implementations without having to repeat the example multiple times. This decision is justified by the fact that this section is not meant to constitute a detailed step-by-step tutorial on utilizing the middleware to create and use services in a specific programming language, but to provide a detailed practical overview of its features from the point of view of the service programmer.

```
#include <ami.idl>

module Example {
    interface Echo {

        // Types
        //
        enum Priority { PR_INFO, PR_WARNING, PR_ERROR };

        struct AdvancedMessage {
            Priority priority;
            string msg;
            long number;
        };
        typedef sequence<AdvancedMessage> AdvancedMessageSeq;

        // Methods
        //
        wstring            EchoUnicodeString (in wstring mesg);
        AdvancedMessage    GetMessage ();
        boolean            GetMessageInArgument (out AdvancedMessage msg);

        // Events
        //
        void Event_AdvancedNotification (in AdvancedMessage msg);
        void Event_AdvancedNotifications (in AdvancedMessageSeq msg);
        void Event_NumbersChanged (in ami::LongSeq numbers);
        void Event_MultiArgNotif (in string msg, in long counter, in boolean d);
    };
};
```

Figure 15 - The interface definition of an example service

The creation of a service starts with the definition of its interface. The set of methods and events it supports, as well as all the specialized data-types that they accept as arguments or payload respectively. Figure 15 shows the specification of a service's interface definition. The service is under the logical namespace "Example" and its name is "Echo". Therefore its fully qualified type if it were to be expressed through the supported programming languages is "Example.Echo" (or "Example::Echo" in C++). The interface definition language used for the description is a subset of CORBA's Interface Definition Language (IDL). The chosen types,

methods and events are meant to showcase many of the service's capabilities for exchanging messages through the abstraction of method invocations.

The definition of a service's interface is done through the interface description-editing tool that is part of the proposed middleware (see 3.6). Once the interface is checked for errors and published, the described service is visible and can be implemented and used by all the developers with access to the tool in any of the supported programming languages. Therefore, in order to implement a published service, developers have to select through the code generation tool the target language, in which they are going to write the service's program, and declare that the code generation is for implementing the service and not for just accessing another implementation. Subsequently, the generated files (or for some languages, compiled libraries) need to be considered by the development environment during the compilation of the final service executable program. The way the latter is done depends not only on the chosen target language, but also on the specific development environment used for the implementation of the service executable program. After that, only the program text that implements the service's methods and events needs to be written.

```
using namespace Example::Echo;

// Declaration
//
class EchoImpl: public virtual POA_Example::Echo {
public:
    wchar_t*            EchoUnicodeString (const wchar_t* mesg);
    AdvancedMessage*    GetMessage (void);
    bool                GetMessageInArgument (AdvancedMessage_out msg);

    void Event_AdvancedNotification (const AdvancedMessage& msg);
    void Event_AdvancedNotifications (const AdvancedMessageSeq& msg);
    void Event_NumbersChanged (const ami::LongSeq& numbers);
    void Event_MultiArgNotif (const char* msg, int counter, bool d);
};

// Methods implementation
//
wchar_t* EchoImpl::EchoUnicodeString (const wchar_t* mesg) {
    std::wstring result = mesg;
    result += L": UNICODE MESSAGE FROM C++";
    return CORBA::wstring_dup(result.c_str());
}

Example::Echo::AdvancedMessage* EchoImpl::GetMessage (void) {
    Example::Echo::AdvancedMessage* result = new Example::Echo::AdvancedMessage;
    result->priority = Example::Echo::PR_INFO;
    result->msg = CORBA::string_dup("GetMessage from C++ called");
    result->number = 1454;
    return result;
```

```
}

bool EchoImpl::GetMessageInArgument (AdvancedMessage_out msg) {
    Example::Echo::AdvancedMessage* result= new Example::Echo::AdvancedMessage;
    result->priority = Example::Echo::PR_WARNING;
    result->msg = CORBA::string_dup("GetMessageInArgument, C++");
    result->number = 11454;

    msg = result;
    return true;
}

// Events implementation
//
void EchoImpl::Event_AdvancedNotification (const AdvancedMessage& msg) {
    ami::LocalEvent evt(1);
    evt.AppendArg(msg);
    ami::Famine::SendEvent(this, "AdvancedNotification", evt);
}

void EchoImpl::Event_AdvancedNotifications (const AdvancedMessageSeq& msg) {
    ami::LocalEvent evt(1);
    evt.AppendArg(msg);
    ami::Famine::SendEvent(this, "AdvancedNotifications", evt);
}

void EchoImpl::Event_NumbersChanged (const ami::LongSeq& numbers) {
    ami::LocalEvent evt(1);
    evt.AppendArg(numbers);
    ami::Famine::SendEvent(this, "NumbersChanged", evt);
}

void EchoImpl::Event_MultiArgNotif (const char* msg, int counter, bool d) {
    ami::LocalEvent evt(3);
    evt.AppendArg(msg).AppendArg(counter).AppendArg(d);
    ami::Famine::SendEvent(this, "MultiArgNotif", evt);
}
```

Figure 16 - The implementation of the example service in C++

Figure 16 shows the implementation of the specified example service in the C++ programming language. There are two things worth noting for this implementation. First of all, since C++ is not a garbage-collected language, the convention for memory allocations/de-allocations for a method's returned values is that it is the caller's responsibility to de-allocate the memory that was allocated by the service's object during a method call. Secondly, the implementations of the event methods, apart from any implementation-specific operations that they need to perform, have to pack all their arguments in a library-provided data-type and forward it through a library-provided function to the event delivery subsystem of the middleware.

```
def main():
    ami.Famine.initialize()

    service = EchoImpl()
    ami.Famine.registerService(service)
```

```
    # Block to serve requests, there's also the non-blocking
    # alternative runOneRound()
    #
    ami.Famine.run()

    ami.Famine.unregisterService(service)
    ami.Famine.cleanUp()
```

Figure 17 - Making the implemented service available to clients through the Python programming language

Given a service implementation, in order for it to become available to the network for being used by any other distributed client, the service program has to create an instance of the service object and register it through a library-provided function. Figure 17 shows exactly this process in the Python programming language. A service program can register many different types of services and many different instances of the same type of service. In the latter case however, it needs to specify a different runtime context for each one of them. The same figure also shows the functions for initializing and shutting down the middleware libraries.

The aforementioned program fully implements a service and makes it available to the network (it exposes it). Next, for accessing a service and being able to invoke its functions and receive asynchronously its events, the code generation tool has to be used again (see 3.6). In this case, apart from the target language, programmers have to specify that the code generation is for accessing the service and not for implementing it. Subsequently, after they import the generated files or libraries to their programming environment, they can immediately obtain a reference to the running service and start calling its methods or declaring interest for asynchronously receiving its events.

```
class EventHandlerObject
{
    public void EventHandler_MultiArgNotif(string msg, int counter, bool d)
    {
        log("Multi arg notification: " + msg + ", " + counter + ", d: " + d);
    }
    public void EventHandler_AdvancedNotification(AdvancedMessage msg)
    {
        log("AdvancedMessage: " + msg.priority + ", " +
            msg.msg + ", " + msg.number);
    }
    public void EventHandler_AdvancedNotifications(AdvancedMessage[] msg)
    {
        foreach (AdvancedMessage m in msgs)
        {
            log("\t");
            EventHandler_AdvancedNotification(m);
        }
    }
    public void EventHandler_NumbersChanged(int[] numbers)
```

```
    {
        foreach (int n in numbers)
            log(n + " ");
    }
}
```

Figure 18 - The implementation of the event handling methods of the client in C#

Figure 18 shows an example implementation of the handler methods for the example service's events in the C# programming language. In this case, the only requirement for the handler objects, in order to be called every time an event is emitted by the service, is to expose public methods with the "EventHandler_" prefix and the name of the service's event as the suffix. Also, in C#, an event handler's arguments have to exactly match the payload of their event counterparts, as specified in the services interface description document.

```
static void Main(string[] args)
{
    Ami.Famine.Initialize(args);

    EventHandlerObject handlers = new EventHandlerObject();
    Example.Echo echo = Ami.Famine.ResolveService<Example.Echo>(handlers);
}
```

Figure 19 - The C# client obtains a reference to the running service of the type specified as a template argument

Following the specification of the event handler object, a reference to a service can be obtained with the registration of the event handlers at the same time. This process can be seen in Figure 19, where the client calls a middleware function to obtain a reference to the example service while, at the same time, specifying the event handler instance that will be invoked whenever the relevant events are emitted by the service. During the resolution of a service, if the latter's runtime context is omitted, then the program's global default context is used for resolution – which is either the name of the running machine, or a string specified as a command line argument when the executable is invoked.

```
public static void main(String[] args)
{
    String str = echo.EchoUnicodeString("Hello from the Java Client in UNICODE!");
    System.out.println("Server returned unicode string: " + str);

    AdvancedMessage result = echo.GetMessage();
    System.out.println("GetMessage returned: " + result.priority.value() +
            ", " + result.msg + ", " + result.number);

    AdvancedMessageHolder argResult = new AdvancedMessageHolder();
    echo.GetMessageInArgument(argResult);
    System.out.println("GetMessageInArgument returned: " +
        argResult.value.priority.value() + ", " +
        argResult.value.msg + ", " + argResult.value.number);
}
```

Figure 20 - The Java client of the same service invokes its methods

Lastly, following the resolution of the service, its methods can be invoked at any time through the obtained reference. This can be seen in Figure 20, in which a Java client of the example service invokes its methods using the reference it had previously obtained.

3.8 Discussion

The proposed middleware was designed and implemented as a means to build distributed services for exposing heterogeneous computing system resources in ambient intelligence environments. Targeting large-scale Ambient Intelligence environments prompted the middleware's software components to provide a powerful, wide and complete set of features, while at the same time automating many tedious and error-prone programming, maintenance and deployment tasks. Through any of the supported programming languages, creating a completely new software service or adapting a legacy application into a service to plug into an AmI environment is a rapid, intuitive process that respects the capabilities, strengths and conventions of the target programming language. Since the middleware provides libraries for many popular programming languages, it already covers and unifies under the umbrella of a homogenized service ecosystem a very wide spectrum of requirements, capabilities and research domains without hindering the underlying performance, suitability, sustainability and reusability of the constituent software.

Moreover, concerning the development process for implementing and using services, apart from the tools described in section 3.6 the proposed libraries are provided through a comprehensive installer package that also provides scripts and automatically configures a set of popular development environments for each of the supported programming languages, thus effectively minimizing all the aspects of the initial setup complexities. Towards providing better support for testing the implemented programs during the development process, as part of our future work we plan to incorporate in Idlematic (3.6) the capability to automatically generate graphical user interfaces that fully implement a service that is defined in the tool, and provide appropriate interface elements for invoking all the methods and logging all the events of services defined in Idlematic. The former

provides an effective way for testing the functionality of programs that use services and the latter for testing a specific service implementation.

At the same time, the design of the whole software and tool framework effectively enables its evolution process towards encompassing even more technologies, programming models and development requirements as the vision of ambient intelligence matures and broadens. In this direction, an important future goal of the proposed middleware is the incorporation of a well-defined set of communication and deployment primitives that will provide an intuitive model for implementing Security and Privacy at the service construction and usage level. Regardless of the fact that security and privacy is primarily a property of the application, which in the case of ambient intelligence uses the environment's distributed services to perform its functions, a set of well-defined security primitives at the service construction and deployment level will certainly provide for a more fine-grained, flexible and robust security model. As mentioned in section 3.2.5, a good practical model to emulate towards implementing sensible and intuitive security primitives is the work done in the Etch middleware [135].

Particular attention has been given also to the robustness and high-availability of the middleware infrastructure. The repository and directory servers (see 3.5) are only used during the deployment and disposal phase of a service, they support redundancy and their state is persistent. Moreover, if a service terminates prematurely between invocations by other services, it can be automatically restarted without affecting the functionality of neither the infrastructure nor the dependent client programs. The repository and the directory servers are implemented in C++ on top of the TAO CORBA libraries [125], which are very portable and available under many different operating systems and processor architectures. Specifically, the standard CORBA Name Service that is provided by the TAO CORBA distribution plays the role of the Directory Server and the Repository Server is realized as a modified version of TAO CORBA's Implementation Repository Service. This makes the servers themselves portable to all the platforms supported by TAO. Therefore, for increasing the reliability of our "release" infrastructure (see 3.5.3), we have deployed the servers on a 64-bit Intel debian GNU/Linux[5] machine. In this context, in order to further

[5] http://www.debian.org

increase the robustness and fault tolerance of the infrastructure, we plan to implement a separate daemon that inspects the deployed servers to verify their proper function. If the latter cannot be verified then the daemon would be responsible for restarting either the servers, or if that does not fix the malfunction, the whole machine.

As far as performance is concerned, the proposed middleware succeeds in delivering fast, low-latency method invocations and event propagation. This is achieved through the proposed peer-to-peer design for the event distribution and, more importantly, through the high-performance CORBA communication protocols and their implementations. This is evident in the comprehensive survey on the performance of different CORBA implementations presented in [136]. Furthermore, its performance and suitability for building AmI systems from diverse heterogeneous resources is further supported by the works in [33], [35], [137], [138], which fully utilize the proposed middleware in order to deliver their functionality.

Chapter 4

Smart Objects for Implementing Ambient Intelligence Environments

In this chapter, we propose a new programming model for designing and implementing the functionality of Ambient Intelligence (AmI) environments. In the proposed approach the top-level architectural abstraction is the environment itself and not the application. Therefore, whereas we could have expressed the purpose of the programming model as an approach for designing and implementing applications in AmI environments, we would have to clarify that the notion of the application is defined as a subset of the functionality of the whole environment. The application, in this case, is not an indivisible, sealed entity but rather just an aspect of the AmI environment.

In this regard, the proposed model uses the abstraction of autonomous processing units, called smart objects, as the fundamental building block for orchestrating the required functionality through the interactions among the distributed components and resources that synthesize an AmI environment. Whereas existing approaches are based on application-centered service-oriented architectures (see 2.1), the proposed approach focuses on describing the interdependent functionalities of the environment formalizing their implementation strategies and aiming to improve the design and development of user-centric, dynamic, robust and extensible AmI ecosystems.

In addition to the definition of the model, an expressive, dynamic, event-based programming language is proposed, featuring high-level asynchronous constructs and semantics. Its features enable the precise description of the structure of smart objects and their interactions within their environment. Due to the properties of the model and the features of the proposed language, the resulting functionality of the implemented AmI environments is adaptable and extensible. The derived environments have the ability to dynamically change their behavior to cater to different user needs or to dynamically changing functional requirements, and to easily extend (or reduce) their functionality even at runtime.

Furthermore, the whole process for describing and deploying smart objects in the AmI environment is supported by graphical tools that provide full visualization of the object interactions and allow for simulating different aspects of their perceived functionalities.

When appropriate, the Z notation [25] will be used to describe more formally the model's concepts and the semantics of the proposed language. We decided to use the Z notation over π-calculus [139], since the former allows for better modeling of the form, structure and operation of the proposed concepts and abstractions. Conversely, π-calculus would be ideal for modeling the execution of a specific scenario that utilizes many different smart objects within the environment. However, since the proposed language specifically targets the description of such interactions, the use of π-calculus would be redundant. Hence, we will use Z's named schema notation where the *Name* of a schema appears on top, it is followed by the schema's variables, and the *Constraining-Predicates* appear last, below the short vertical line. Essentially, a schema represents a set of tuples *Name* = $\{x_1:T_1;\ldots;\ x_n:T_n \mid$ *Constraining-Predicates*$\}$, where $x_i:T_i$ denotes that x_i has type T_i, i.e., $x_i \in T_i$.

In the following sections we will present all the aforementioned aspects of the proposed approach. Specifically, section 4.1 describes the goals and high-level design of the model with respect to its use for developing AmI environments. Sections 4.2 and 4.3, introduce the abstraction of smart objects and their environment respectively. Using these abstractions, sections 4.4 and 4.5 describe the notion of services, memory and scope in the context of the proposed model. Section 4.6 describes the concept of the "self port", which enables individual smart objects to communicate with the environment and access its functionality. The proposed language for programming smart objects and its implementation through the provided development and deployment tools are presented in sections 4.7 and 4.8. Following that, the notion and implementation details of "external smart objects" that enable the incorporation of external entities (e.g. hardware devices) as smart objects are specified in section 4.9. Finally, section 4.10 provides a summary and discussion of the proposed model and its implementation, highlighting some potential directions for future work.

4.1 Goals and Design

In the proposed approach, the functional role of services is to provide the means for implementing the sensitivity and responsiveness of the AmI environment. At the other end, the functional role of smart objects is to provide the means for implementing its intelligence and adaptability. In this regard, sensitivity and responsiveness refer to the properties of the environment's resources (Chapter 3), whereas intelligence and adaptability refer to the properties of the environment's behavior. Ultimately, this behavior manifests itself through the manipulation of resources in response to the perceived properties of resources. Moreover, in this context, it can be inferred that whereas the architectural role of services is to model the slow-changing domain knowledge of an AmI environment, the role of smart objects is to model the latter's rapidly-changing functionality. The functionality imbued in an AmI environment is characterized as rapidly-changing since its functional requirements are constantly affected by a plethora of volatile factors which can include the results of usability tests, the incorporation of new techniques or technologies, the conformance to new regulations, etc. This observation drove the design of the proposed language for programming smart objects and their interactions. The proposed language, called Akkadian, is dynamic, weakly-typed and event-driven, allowing the immediate execution of its programs while enabling their extensibility using intuitive abstractions and techniques. In contrast to the dynamic nature of smart object implementations, the proposed middleware for services, presented in Chapter 3, provides statically-checked type-safe interfaces, reflecting the slow-changing nature of the environment's resources.

In this sense, smart objects provide a systematic well-defined means to describe what the AmI environment *does*. From a software engineering point of view, the description of the available smart objects in the context of an AmI environment, fully describe its behavior.

Moreover, apart from improving the design and implementation process for building AmI environments, another goal of the proposed approach, is to enhance their extensibility potential. This is achieved by both the extensibility potential of the smart object model and the dynamic abstractions offered by Akkadian. Moreover, another important aspect of the system is its ability to compose high-level behaviors by consuming services implemented under diverse technological platforms. Whereas the proposed approach provides full

specification and implementation of a robust and intuitive service platform, we wanted to expand the design and implementation space by providing the ability to incorporate and use other diverse functionalities without needing to "wrap" them under services built on top of the proposed middleware. Using the smart object as a composition unit described in Akkadian, the former is able to additionally exploit the functionality offered by REST services [122] and .NET DLLs [121] while enabling the incorporation of more technologies via service-provider-engine plug-ins.

Finally, another important goal that drove the design of the proposed system was the full visualization of the state of the environment and the interactions among the contained smart objects. Towards this direction, the implementation of Akkadian's runtime environment visualizes all the available smart objects, their connections and interactions. The provided user interface and the abstractions offered by the proposed language, additionally enable the simulation of a subset of the functionality of the modeled AmI environment.

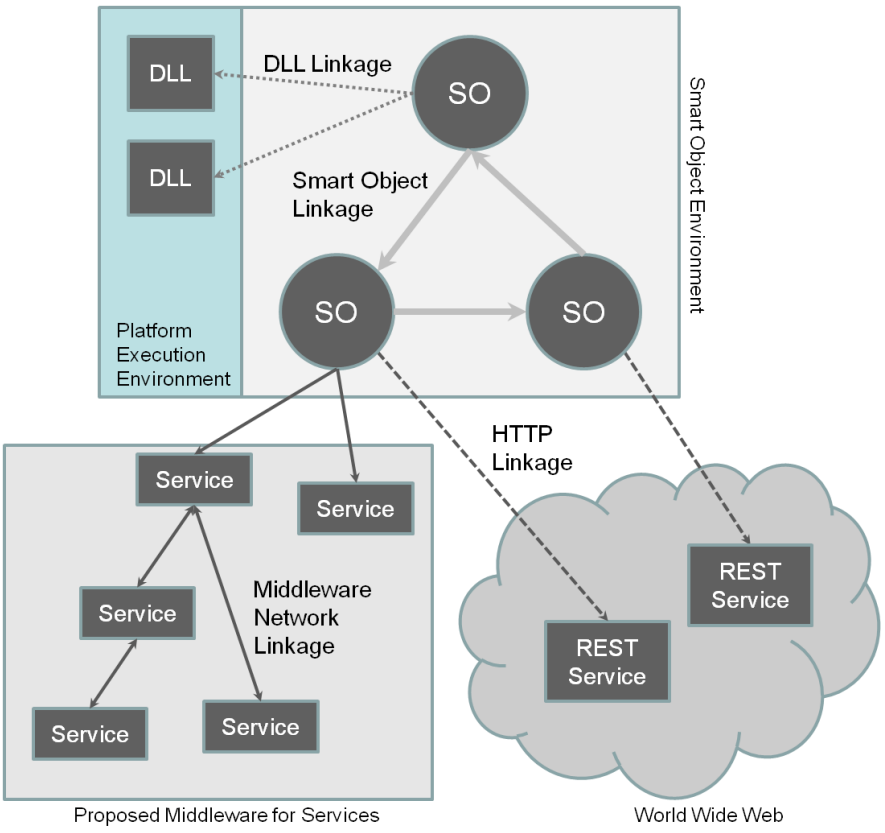


Figure 21 - Runtime architecture of the Smart Object Environment

Figure 21 shows the runtime architecture of a smart object environment. The interconnected smart objects in the figure consume the functionality exported by DLLs, by services under the proposed middleware platform, and by REST services distributed over the World Wide Web (WWW).

4.2 Smart Objects

Smart objects, at a basic conceptual level, are autonomous computing units that can operate in isolation, connect to other objects, thus creating complex structures in the form of object federations, and interact with each other within their environment [104]. A smart object can be viewed in three different ways: as a container unit, as a structural unit (passive or active), and as an interaction unit. As a container unit, an object contains memory and an arbitrary number of external services. As a structural unit, it can initiate (active) or accept (passive) connections from other objects, and, lastly, as an interaction unit, a smart object is able to issue and to execute commands, to emit and capture events, and to use the functionality that is implemented by its contained services. Smart objects are thus defined in terms of their characteristics, capabilities and behavior. Provided that they follow a basic set of operational primitives [104], [140], they can be realized as either software or hardware entities. In this thesis we will be focusing on software smart objects with the final goal being the seamless coexistence of software and hardware smart objects that federate and interact with each other in order to realize AmI scenarios.

Smart objects contain memory through which they can store and access arbitrary amounts of data. All operations on the memory are atomic and sequentially consistent [141] with respect to the containing object and the smart objects that have access to it. Moreover, objects contain an arbitrary number of services that they can invoke, retrieve the results of their invocations and receive their asynchronous events. The contained services can either be implemented within the context of a smart object or be external services that are attached to it. In this regard, services can be attached through any method as long as they provide the means to conform to the expected service model (see 4.4). In this regard, smart objects programmed with the proposed programming language (Akkadian) can only attach

external services, whereas, external objects (4.9) can also implement services within their own execution environment.

Memory and services, in the context of smart objects, are associated with an abstraction called service-providing port (PP), which essentially models the access to both of these elements. A service-providing port has a name through which it can be referenced in the context of a specific smart object. This essentially implies that there cannot be more than one service-providing port with the same name within the same smart object. Conceptually, a service-providing port seems to provide its associated service and memory to either the containing smart object or to other objects, through another abstraction called service-requesting port (RP). Similarly, a service-requesting port has a name through which it can be referenced in the context of a specific object, which in turn, cannot contain more than one service-requesting port with the same name. Conceptually, therefore, a service-requesting port seems to request a service and memory that can be provided by a service-providing port. Apparently, there can be a pair of service-requesting and service-providing ports that are associated with the same name within the context of an object.

A service-requesting port is able to use the service and the memory that are associated with a service-providing port, only if the former is connected to the latter. A connection between a service-requesting port and a service-providing port can be established only if both ports are associated with the same name (see 4.6). In this case, two ports that are able to establish a connection are said to be “matching ports”. A connection between two matching ports can also be considered as a connection between two objects. In this sense, a connection between two objects is much more than just a simple communication channel; it is an explicitly expressed relationship between an object that needs to use some functionality and one or more objects that can offer the requested functionality.

In the textual examples and figures in this chapter we will denote a service-providing port which is associated with the name “p” as “+p”, and a service-requesting port with name “p” as “-p”. This notation is also used in the proposed language (see 4.7) to refer to the corresponding ports in the context of a smart object program.

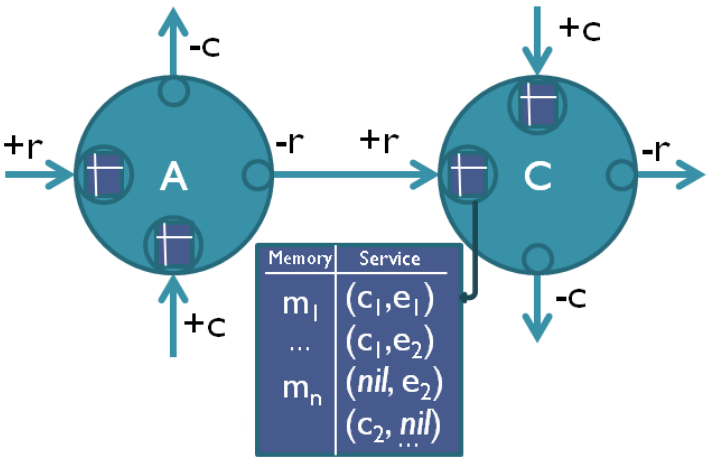


Figure 22 - Two smart objects A, C, connected through port "r"

Figure 22 shows two smart objects that have established a connection through their matching ports. In this case, we can unambiguously refer to an established connection between a service-requesting port with name “p” and a matching service-providing port, as a connection through/over port “p”. The same picture also shows the service and memory that is associated with service-providing ports. More details about these abstractions will be given in section 4.4.

Therefore, more formally, the abstraction of the smart object (SO) can be denoted with the following Z schema [25]. As mentioned before, we will use Z’s named schema notation where the *Name* of a schema appears on top, it is followed by the schema’s variables, and the *Constraining-Predicates* appear last, below the short vertical line. A schema is modeled as a set of tuples *Name* = $\{x_1:T_1;\ldots; x_n:T_n \mid \textit{Constraining-Predicates}\}$, where $x_i:T_i$ denotes that x_i has type T_i, i.e., $x_i \in T_i$. Moreover, $\mathbb{P}T$ denotes the power set of T, i.e., $A \in \mathbb{P}T \Leftrightarrow A \subseteq T$, and $X \rightarrowtail Y$ denotes a partial function from a set, X, to another set, Y.

$$[\ NAME, PP, RP, M, I, O\ ]$$
$$I^* \cong I \cup \{\ nil\ \}$$
$$O^* \cong O \cup \{\ nil\ \}$$

$$
\begin{array}{|l}
\hline
SO \\
\hline
pp : \mathbb{P} PP \\
rp : \mathbb{P} RP \\
pname : PP \rightarrow\!\!\!\!\!\!+ \; NAME \\
rname : RP \rightarrow\!\!\!\!\!\!+ \; NAME \\
mem : PP \rightarrow\!\!\!\!\!\!+ \; \mathbb{P} M \\
serv : PP \rightarrow\!\!\!\!\!\!+ \; (I^* \leftrightarrow O^*) \\
\hline
\forall x, y \in pp \bullet pname(x) = pname(y) \Rightarrow x = y \\
\forall x, y \in rp \bullet rname(x) = rname(y) \Rightarrow x = y \\
\mathrm{dom}\ rname = rp \\
\mathrm{dom}\ pname = \mathrm{dom}\ mem = \mathrm{dom}\ serv = pp \\
\hline
\end{array}
$$

In this regard, each smart object contains two sets: (*pp, rp*). The first set, *pp*, represents the service-providing ports through which a service can be used by the object or its peers, and the second set, *rp*, represents the service-requesting ports through which an object can establish a connection to another one and thus use the service that is associated by the connected service-providing port.

An element $p \in PP$ is, in turn, associated with two partial functions, *mem* and *serv*, to an element of *M* and *S* respectively, with the set *M* representing the memory of the port and the relation $S \in I^* \leftrightarrow O^*$, (where $I^* = I \cup \{ nil \}$ and $O^* = O \cup \{ nil \}$), modeling a service. The symbol "↔" represents the set of relations between two sets, i.e., $S \leftrightarrow T = \mathbb{P}(S \times T)$.

The types (sets) that are not relevant for our modeling are referenced in the declaration header using Z's square-bracket notation. The semantics of the partial function that models services in the proposed model will be described in section 4.4.

4.3 Environment and Federations

The smart object environment is represented as a labeled directed multi-graph with no self-loops. The connected components of the environment graph denote smart object federations. In a smart object federation graph, the vertices represent the smart objects, and each directed edge a connection of a smart object to another one through a port. The head of an edge is the object that requests a service from another one (through a service-requesting port) and its tail is the object that offers a service (through a service-providing port). The label of the edge represents the common name of the ports through which the

connection is established. Two objects may be connected to each other through multiple ports (multi-graph) but cannot establish a connection to a port contained in the same object (no self-loops). Moreover, a service-requesting port cannot be connected to more than one service-providing port. In this sense, the environment is represented by the Z schema below, where the symbol "#" denotes the cardinality of the set that follows it. The schema also uses Z's set-builder notation, where free variables are followed by a predicate that restricts the set's members. For example, the set-builder expression $\{ x: \mathbb{N} \mid x = x^2 \}$ is the set $\{ 0, 1 \}$.

Env

$$
\begin{array}{l}
objs : \mathbb{P} SO \\
conns : \mathbb{P}(SO \times SO \times NAME) \\
pport : SO \rightarrow\!\!\!\!\!\!+ \; \mathbb{P} PP \\
rport : SO \rightarrow\!\!\!\!\!\!+ \; \mathbb{P} RP \\
\hline
\forall x, y \in objs; \; l \in NAME \bullet (x, y, l) \in conns \Rightarrow x \neq y \\
\neg \exists x \in objs \bullet \#\{ y : SO; \; l : NAME \mid y \in objs \wedge (x, y, l) \in conns \} > 1 \\
\mathrm{dom}\ pport = \mathrm{dom}\ rport = objs
\end{array}
$$

The ability to create federations in the context of the environment is a fundamental characteristic of the smart object model. Conceptually, in an object federation graph, the direction of connection, apart from indicating which object initiated the connection, also models the fact that the source object (connection initiator) needs to *use* some functionality that is provided from the target object (connection acceptor). This explicit representation of connection direction does not imply a restriction in the communication direction. Two connected objects can exchange information between them through a connected port regardless of which object requests the associated service and which provides it.

Figure 23 shows three smart object federations that are connected through various ports within the environment. In this figure, object *I* belongs to a unary federation with itself as the only member. Moreover, this picture shows objects *E* and *F* connected through two different edges with the same name. This essentially means that object *E* contains two ports "-d" and "+d" which are connected with the matching ports "+d" and "-d", respectively, which are contained in object *F*.

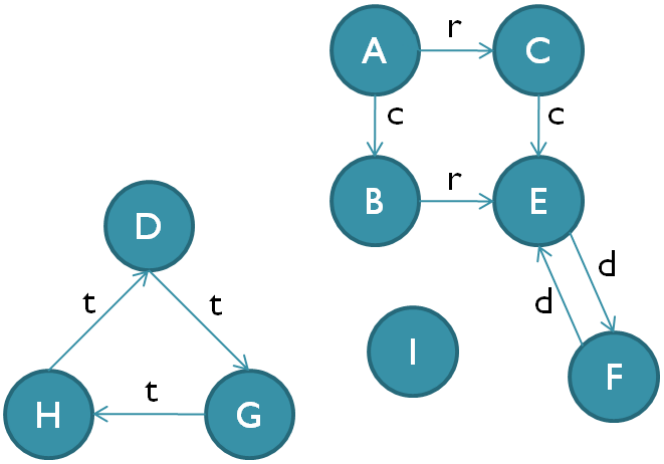


Figure 23 - Three Smart Object Federations in the Environment

Having defined the notion of the smart object and the smart object environment, we can now express through them the notion of the AmI environment and the AmI application. This is straightforward since the AmI environment (aEnv) is modeled as the smart object environment and an AmI application (aApp), which is an aspect of the operation of the whole AmI environment, is modeled as a subset of the objects contained in the smart object environment. This can be expressed through the Z axiomatic definition below, where the symbol "Ξ" includes the referenced schema in the current one, allowing the use of the variables defined previously, and also indicating that the schema's post-state is not affected.

$$\begin{array}{|l} \Xi Env \\ aEnv : Env \\ aApp : \mathbb{P} SO \\ \hline aEnv = Env \\ aApp \subseteq objs \end{array}$$

As mentioned before, we schematically represent a service-requesting port ($p \in RP$) with the associated name "p" as "-p" and a service-providing port ($p \in PP$) with the associated name "p" as "+p". Using this representation we can define a binary relation *path*: SO $\leftrightarrow$ SO that enables a smart object to refer to another smart object within its federation through a sequence of ports that denote the connections in the context of the source object's federation. This is possible, since within a single object, a port representation (e.g., -p or +p), uniquely references a single port. As mentioned before, the smart object model does not allow two service-requesting or service-providing ports with the same name to coexist in a single object. Therefore the implementation of the *path* binary relation in a federation is

defined as the concatenation of signed port names that indicate a path from a source object to a target object. In a federation graph, the direction of an edge does not restrict the capability of referencing signed port names in a *path* instantiation as the concatenation of connected ports. Examples of path relations in the context of object federations are depicted in Figure 24.

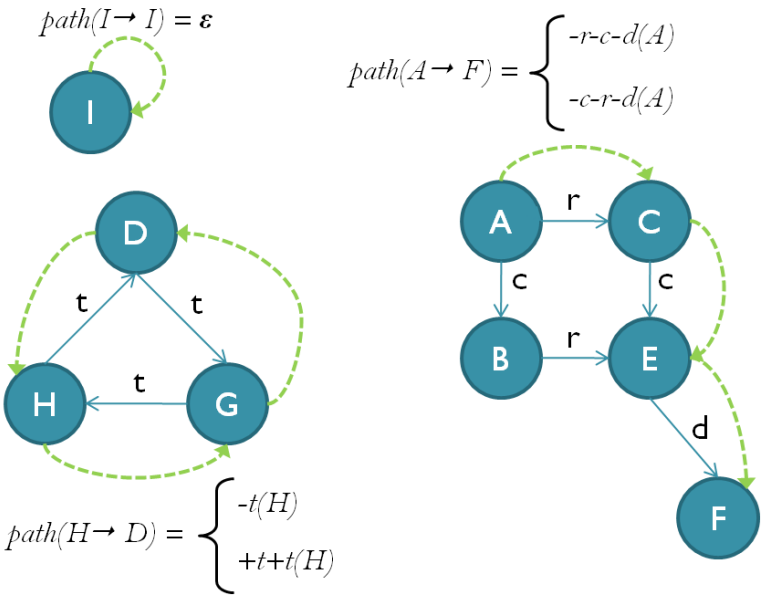


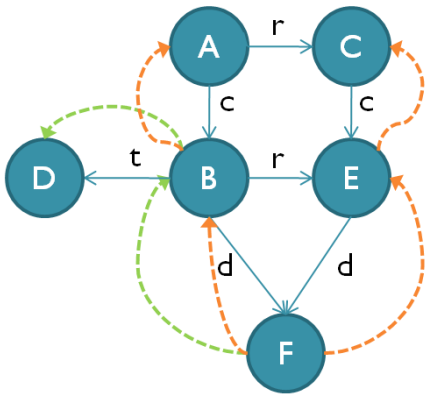
Figure 24 - Three Smart Object Federations with some example paths clearly visible

When a path is defined along the direction of connections, i.e., along the direction of the edges that represent connections in the graph-based representation of a smart object federation, the instantiation of a *path* relation uniquely and unambiguously references a single object. The reason for this is that the out-degree of a service-requesting port is guaranteed to be one. Additionally, if the in-degree of a service-providing port is one, a path defined against the direction of connections also uniquely and unambiguously references a single object.

However, since the model allows service-providing ports to have arbitrary in-degrees, ambiguities may manifest in path definitions when they reference service-providing ports with an in-degree greater than one. The path resolution process is resilient to such path definitions and should try to solve potential ambiguities by using infinite look-ahead in its path resolution algorithm. That means that when a path references a port with an in-degree greater than one, the object that is chosen among those connected to it should be the one that is able to resolve the next element of the path. If the previous statement is satisfied by

more than one object, the next path element is considered for the resolution of the ambiguity and this process continues recursively. If, even with the infinite look-ahead, the full path is still ambiguous by referring to more than one object, then any of them could be chosen (implementation-defined) to be the destination of the path reference. Subsequent resolutions of an ambiguous path are not guaranteed to consistently return a reference to the same object.

Examples of a path on which ambiguities are resolved using the infinite look-ahead and a genuinely ambiguous path can be seen in Figure 25, where the path in (a) cannot be resolved determinedly to a unique object, since in its entirety the path points to two separate objects. On the contrary, the path in (b) despite containing an ambiguous component (+d), can be resolved to a unique object since the next component (-t) removes the ambiguity.



(a) $+d+c(F) = path(F \rightarrow A \textbf{ or } C)$ [ambiguous]

(b) $+d\text{-}t(F) = path(F \rightarrow D)$ [not ambiguous]

Figure 25 - An ambiguous path that references one of two objects (a), and an unambiguous path whose one ambiguity is resolved by the infinite look-ahead (b)

In the proposed language, Akkadian, which provides an implementation of the smart object model, the resolution of ambiguous paths that refer to more than one object is well-defined. Akkadian always resolves the path to the oldest among the ambiguous objects with respect to the order in which they joined the federation.

4.4 Services and Memory

Conceptually, a service-providing port has memory, and can export and make available to its containing object a set of commands and events. An object containing a service-requesting port can access and utilize all the available memory, commands and events as soon as it establishes a connection with a service-providing port. In this context, the functionality provided by a port, i.e., its commands and events, is referred to as a "service".

Therefore, as mentioned in the previous section, services in Akkadian are modeled as a binary relation $S \in I^* \leftrightarrow O^*$. The domain of the relation represents the set of commands that can be issued through the service and the codomain represents the events that can be received through the service. The relation specifies that certain events (elements of the codomain) are generated as a result of the invocation of a command (elements of the domain). All the pairs of the form (*nil*, o), with o ∈ O, denote events that are generated by the service automatically without any previous command invocation (e.g., an asynchronous "door-opened" event). Similarly, all pairs of the form (i, *nil*), with i ∈ I, denote those commands that do not cause the service to produce any events (e.g., a "turn-off" command). Based on this, the semantics of the invocation of a command (*cmd*) that does not produce any event and the semantics of receiving an event (*evt*) that is not associated with a command are defined below. In the Z notation, the symbol "Δ" includes the referenced schema in the current one, allowing the use of the variables defined previously. The symbol "?" indicates the schema variables that work as the inputs of the operation. The symbol "⩤" denotes the domain subtraction operation, which obtains all the pairs of the second set whose first member is not contained in the first set, i.e., $R \in S \leftrightarrow T$ and $A \subseteq S$, $A \lhd R = \{ a: S; b: T \mid (a, b) \in R \land a \notin A \}$. Similarly, the range subtraction operator, "⩥", with $R \in S \leftrightarrow T$ and $B \subseteq T$, is defined as $R \rhd B = \{ a: S; b: T \mid (a, b) \in R \land b \notin B \}$. Finally, variables decorated with an apostrophe (') refer to their value in the post-state, after the evaluation of the schema.

InvokeCommand

ΔSO
$cmd?: I$
$p?: PP$

$p? \in pp$
$serv(p?)' = (\{ cmd? \} \lhd serv(p?)) \cup \{ (cmd?, nil) \}$

ReceiveEvent

ΔSO
$evt?: O$
$p?: PP$

$p? \in pp$
$serv(p?)' = (serv(p?) \rhd \{ evt? \}) \cup \{ (nil, evt?) \}$

It follows from the above that the semantics of the invocation of a command that produces an event (this notion can also be expressed as: the emission of an event that is associated with a command), is given by the following Z schema.

InvokeCommandAndReceiveEvent

ΔSO
$cmd?: I$
$evt?: O$
$p?: PP$

$p? \in pp$
$serv(p?)' = (\{ cmd? \} \lhd serv(p?)) \cup \{ (cmd?, evt?) \}$

As long as services adhere to this model, they can be fully utilized by smart objects through their service-providing ports. Thus, a service, depending on the actual smart object manifestation, may be a device driver that operates a motor, a REST service that queries a web search engine, or a CORBA service that performs withdrawals from Cyprian bank accounts when the banks are closed. In the implementation of the proposed language, different service technologies can be adapted to adhere to the model through the runtime environment's service-engine extension mechanism.

Following the definition of services in the proposed model, we can now define the notions of the *sensitivity* and *responsiveness* (behavior) of the AmI environment. As mentioned before, sensitivity (aSens) refers to the ability of an AmI environment to fully sense the context in which it operates, and responsiveness (aResp) to its ability to respond to the needs of its users. In this regard, they are modeled by the following Z axiomatic definition, where the symbol "dom" denotes the domain of a relation, and the symbol "ran" the range. In Z's set-builder notation, the expression that follows the "•" symbol denotes the resulting elements of the set. Generally, { x: T | pred(x) • expr(x) } denotes the set of all elements that result from evaluating expr(x) for all x of type T, for which pred(x) holds.

$$
\begin{array}{|l}
\Xi Env \\
\Xi SO \\
aResp : \mathbb{P} I \\
aSens : \mathbb{P} O \\
\hline
aResp = \mathrm{dom}\ \{\ o : SO;\ p : PP \mid o \in objs \land p \in pport(o) \bullet serv(p)\ \} \\
aSens = \mathrm{ran}\ \{\ o : SO;\ p : PP \mid o \in objs \land p \in pport(o) \bullet serv(p)\ \}
\end{array}
$$

Apart from the service, a service-providing port is associated with memory. A memory element m ∈ M is in turn associated with a specific name that models its position within the memory space that is associated with the port. Therefore, within the context of the port memory, only one element that is associated with a specific name is allowed. In this regard, the memory update operation is modeled by the following Z schema, where the symbol "\" denotes the set difference operation.

UpdateMem

$$
\begin{array}{|l}
\Delta SO \\
n : M \rightarrowtail NAME \\
v? : M \\
p? : PP \\
\hline
p? \in pp \land \mathrm{dom}\ n = mem(p?) \\
mem(p?)' = (\ mem(p?) \setminus \{\ m : M \mid m(m) = n(v?)\ \}\ ) \cup \{\ v?\ \} \\
\hline
\end{array}
$$

Moreover, in the context of a specific smart object, memory has the following properties: (1) it can be read/written atomically at one position; (2) it can be read/written atomically at multiple positions (batch read/write); and (3) it is sequentially consistent with respect to the order of write commands issued by different objects. In this case, sequential consistency [141] means that every smart object that can access a specific memory, sees the update operations on the same memory in the same order. It is worth pointing out that this model is weaker than strict consistency which would require that operations are seen by the smart objects in the order in which they were actually issued.

Atomic reads and writes, in this context, practically means that whenever an object *A* issues a read command on a memory element on which another object *B* issues simultaneously a write command, depending on which command (i.e. the read or the write one) arrives first, the initial object *A* will either obtain the older value of the memory cell (before it was written by *B*) or the new value (after it was written by *B*). In any case, thus, the memory model guarantees that object *A* will never receive inconsistent or corrupt data. An example

of this case is shown in Figure 26, where objects *A* and *B* issue respectively a read and a write command simultaneously.

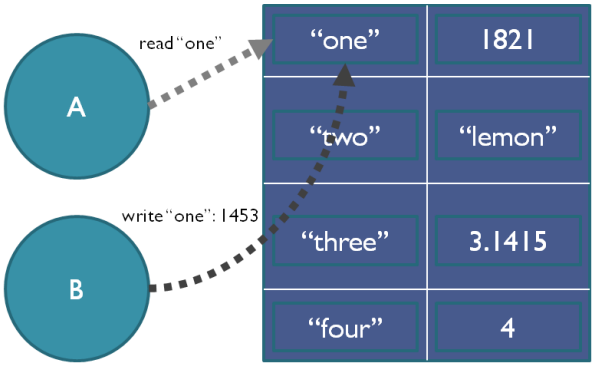


Figure 26 - Object A is guaranteed to read either 1821 or 1453

Similarly, in case of an atomic batch read command in multiple memory elements by object *A*, a simultaneous symmetric batch write command by object *B* (that writes the same cells), either updates all the cells or none with respect to the read command. Therefore, depending on the order in which the commands are processed, object *A* either reads the old values of all the elements or the new values of all the elements. The model guarantees that object *A* will never receive corrupt data or an intermixing of old and new values of memory elements. An example is shown in Figure 27, where objects *A* and *B* issue respectively a batch read and a batch write command simultaneously.

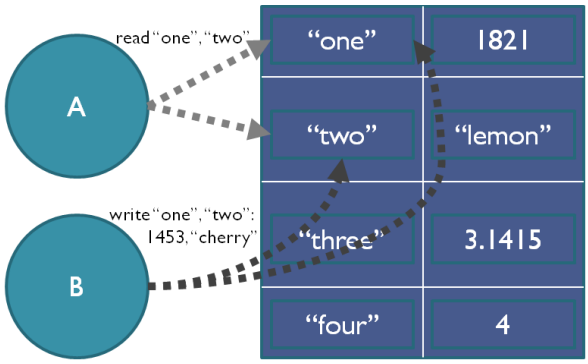


Figure 27 - Object A is guaranteed to read either (1821, "lemon") or (1453, "cherry")

In the proposed language, Akkadian, which implements the aforementioned memory model, an update command issued from an object has as a result the emission of a memory event which can be captured in the same way as the events emitted from services. This behavior is explained in section 4.7.3.

4.5 Scope

In the context of a specific smart object, *A*, scope is defined as the set of smart objects that satisfy *A*'s scope predicate and do not belong in the same federation as *A*. In general, outside the context of a specific object, scope is modeled as a binary relation whose codomain represents the objects that are within the scope of the corresponding objects in the relation's domain. Specifically, the definition of the scope relation can be seen in Z's axiomatic definition below.

$$
\begin{array}{|l}
\Xi Env \\
scope: SO \leftrightarrow SO \\
scopePred: SO \times SO \leftrightarrow \{ true, false \} \\
sameFedPred: SO \times SO \leftrightarrow \{ true, false \} \\
\hline
scope = \{ x, y: SO \mid x \neq y \wedge x \in objs \wedge y \in objs \wedge \\
\qquad \neg sameFedPred(x, y) \wedge scopePred(x, y) \bullet (x, y) \}
\end{array}
$$

Conceptually, the scope's predicate, in the context of a smart object, denotes those conditions that allow the former to sense the presence of other smart objects in the environment. In this regard, an object being within another one's scope means that the former is visible to the latter. Similarly, an object entering (or leaving) another object's scope means that the former starts (or stops) being visible to the latter. In the proposed model, objects that belong to the same federation cannot enter each other's scope even if they satisfy each other's scope predicate. The reason for this is that objects within the same federation can refer to each other through the *path* relation (see 4.3).

Practically, therefore, the implementation of scope is not fixed and also depends on the object's physical instantiation. A hardware smart object, for example, may define scope as the set of objects that are within its wireless antenna's range, as in [104] and [142], with the scope predicate, then, taking the form of a proximity relationship. On the other hand, a software smart object instantiation may define scope as the publication of an object onto a directory server on the Internet. Therefore, in general, scope may imply any method that will enable a smart object to make its presence known to other smart objects. An example schematic description of the scope from the point of view of a specific smart object can be seen in Figure 28, where the objects that belong in the same federation are not members of its scope.

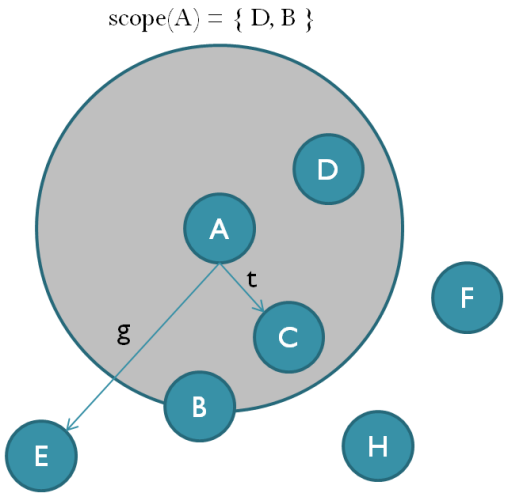


Figure 28 - The scope of object A

We have seen so far that smart objects can refer to other smart objects only through the path relation (*path*: SO ⟷ SO) and the scope relation (*scope*: SO ⟷ SO). From this follows that an orphan object (i.e., an object that belongs to the unary federation with itself as the only member), has to enter the scope of another object (or vice versa) in order to join a federation. Therefore, if we broaden the semantics of a federation to also imply *trust* among its members, then we can recursively assure that this trust relationship holds when new objects join a federation through the scope predicate. This makes the definition of the scope predicate an intuitive and robust construct for implementing security, privacy and, in general trust in the context of the smart object environment.

In Akkadian, scope is implemented as a proximity relation among smart objects in the environment, which in turn is implemented as an infinite two dimensional plane. Currently, however, it does not define any security primitives for implementing access control in the context of smart objects. Allowing the explicit definition of scope by programmers and including security primitives in Akkadian is one of the major objectives of our future work.

4.6 The Self Port

The proposed smart object model also defines a special service-providing port that has the same semantics as other service-requesting ports, with the following restrictions: (a) it has to be present in all smart objects in the environment, (b) it must be associated with a specific name that is common among all the smart objects in the environment, (c) there

cannot be any service-requesting port associated with that specific name, (d) its associated service must include a well-defined set of pairs $S' \in I^* \leftrightarrow O^*$, with $S' \subseteq S$, and (e) its associated memory must include a specific entry $m' \in M$ which is in turn associated with a specific name. This service-providing port will be referred to as the *self* port. From these restrictions, it follows that the self port cannot be used for establishing smart object federations and can only be used within the context of its containing object.

Generally, in the context of the proposed model, the self port provides the means for smart objects to connect to each other and establish federations, to break federations, and to sense whether other smart objects entered or left their scope. These abilities are provided by the associated service of the self port. In this regard, we refer to the domain of the relation $S' \in I^* \leftrightarrow O^*$ as the self port commands and to its codomain, as the self port events.

A connection between two smart objects can only be established through their ports. A service-requesting port that is not already connected, can be connected to a service-providing port of another object as long as the latter has the same name as the former, i.e. the ports are "matching ports". If these conditions are met, the *Link* self port command, successfully establishes the connection. Thus, the definition of Link command can be seen in the Z schema below, where, as before, a variable decorated with an apostrophe (′), represents its state after an update has taken place.

Link

$$\Delta Env$$
$$\Xi SO$$
$$o1?, o2 : SO$$
$$p1? : RP$$
$$p2? : PP$$

$$p1? \in rport(o1?) \land p2? \in pport(o2?)$$
$$rname(p1?) = pname(p2?)$$
$$\neg(\exists x \in objs; l \in NAME \bullet (o1?, x, l) \in conns)$$
$$conns' = conns \cup \{ (o1?, o2?, rname(p1?)) \}$$

Symmetrically, the *Unlink* command breaks an already established connection between two ports as can be seen in the Z schema below.

Unlink

ΔEnv
ΞSO
$o1?, o2: SO$
$p1?: RP$
$p2?: PP$

$p1? \in rport(o1?) \land p2? \in pport(o2?)$
$rname(p1?) = pname(p2?)$
$(o1?, o2?, rname(p1?)) \in conns$
$conns' = conns \setminus \{ (o1?, o2?, rname(p1?)) \}$

Table 5 below, contains a high-level description of the Link and Unlink commands. The result of the invocation of these self port commands (c ∈ I), are communicated to the containing object through the *Linked* and *Unlinked* self port events (e ∈ O).

Table 5 - List of commands associated with the self port

Command	Description	Parameters	Pre Conditions	Associated Event
Link	Establishes a connection to an object with a matching service-providing port	Service-requesting port, Target object reference	The target object should have a matching service-providing port	Linked
Unlink	Breaks an already established connection	Service-requesting or Service-providing port, Target object reference	The given port must be connected	Unlinked

Apart from the Linked and Unlinked events, the self port also defines the *InScope* and *OutScope* events that are received by the containing smart object when an object enters or leaves its scope respectively. The *Activated* event is emitted the first time an object is deployed in the smart object environment and becomes active. An active object in this case is an object that is able to operate fully in accordance to its specification. For example, a hardware smart object can be active as soon as it is powered-on and a software smart object when it is instantiated by its execution environment. Symmetrically, the *Deactivated* self port event is emitted just before the containing object stops being active. A summary of the events that are associated with the self port can be seen in Table 6 below.

Table 6 - List of events associated with the self port

Event	Description	Payload	Post Conditions	Associated Command
Linked	Emitted when a link is established between two matching ports	The port which was connected, The object which had the matching port	The objects must be connected through the referenced port	Linked
Unlinked	Emitted just before a link is broken between two connected, matching ports	Service-requesting or Service-providing port	The objects must not be connected through the referenced port	Unlink
InScope	Emitted when an object enters the scope of the current object	A reference to the object that entered the scope	The object must be in the current object's scope	*nil*
OutScope	Emitted just before an object leaves the scope of the current object	A reference to the object that left the scope	The object must not be in the current object's scope	*nil*
Activated	Emitted when the object is activated (e.g., deployed or powered-on)	*None*	The current object must be active	*nil*
Deactivated	Emitted just before the object is deactivated	*None*	The current object must be inactive	*nil*

Finally, the memory that is associated with the self port has to contain a specific entry that is associated with a specific name which signifies the name of the containing smart object.

4.7 Smart Object Programming Language

This section presents a dynamic, weakly-typed, event-based programming language that fully implements the smart object model defined in the previous sections. Our primary motivation was to provide an effective and intuitive way to program smart objects for implementing and simulating the functionality of AmI environments. The proposed language takes its name from the extinct Semitic language that was spoken in ancient Mesopotamia, called Akkadian [143]. Since the cuneiform writing system of Akkadian, as seen in Figure 29,

resembles networks of interconnected entities, we decided to use its name for referring to the proposed language for programming smart objects. However, despite the fact that the Unicode standard fully defines the encoding of Akkadian's cuneiform script (at the U+12000-U+123FF range), we decided against using its writing system for representing the text of smart object programs.

Figure 29 - An Akkadian inscription

Akkadian, the dynamic executable programming language proposed in this section, allows for the full implementation of the structure of smart objects and all interactions among them. The decision to propose a completely new language is justified for three reasons. First of all, since we wanted to provide asynchronous semantics to the implementation of the smart object model, it would be difficult to provide syntactically intuitive and effective event-based frameworks through existing popular general-purpose languages. Secondly, an implementation in a general-purpose language has to accommodate extraneous programming statements that increase the complexity of the text representation of a program without contributing anything to the core functionality of the objects themselves (boilerplate code). Thirdly, full control to the language's high-level constructs was necessary not only to simplify the description of smart objects and their interactions but to also accommodate for future extensions and common programming and usage patterns that will emerge from the application of the smart object paradigm into real-world scenarios. In this sense, the proposed programming language for the description of smart objects falls into the category of Domain Specific Languages (DSL) [144].

In the following subsections we will describe Akkadian's semantics and usage, highlighting some implementation details where necessary.

4.7.1 General Structure

All Akkadian's statements are defined within the context of the described smart object. The structure of an object, then, is specified by the definition of all its service-requesting and service-providing ports. Subsequently, the functionality of an object is effectively defined as the sum of all its reactions to the events it receives from its associated ports. These reactions, in turn, are defined by the Akkadian statements used for emitting and capturing events, executing commands, and reading and writing memory elements. The sequence of textual statements that follows the syntax and semantics of Akkadian, in order to describe a smart object and its functionality, is referred to in this document as the smart object program.

In a smart object program, a port is defined by the *port* keyword that is followed by a signed identifier that denotes the name of the port and whether it is service-requesting or service-providing. A service-requesting port is prefixed with the minus (-) sign and is fully defined with a single port statement. On the other hand, service-providing ports are prefixed with the plus (+) sign and can host some port-specific statements that describe their associated services and access characteristics. Specifically, service-providing ports can define (a) the service that is attached to them, (b) their arity, (c) the service specific parameters for the attached service, and (d) the service authentication parameters for the attached service. A service can be attached through the *uri* statement inside the body of a service-providing port definition and is followed by a Uniform Resource Identifier (URI) [145] that describes its location. The *arity* of a service-providing port is defined as a number that denotes the port's in-degree, i.e. the number of service-requesting ports that can be connected to it. Finally, the service-specific (*params*) and authentication (*auth*) parameters configure aspects of the usage of the referenced service, depending on its implementation technology. All the aforementioned parameters can also be specified using Akkadian's attribute expression list syntax, where a parameter keyword is followed by a colon that precedes the parameter's value. An example of the syntax for port definitions can be seen in Figure 30 below.

```
port -C;
port -L;

port +testAmIService("ami:///Example/Echo");

port +search
```

```
{
    uri "https://www.googleapis.com/customsearch/v1";
    params [ key: "BIDaSyBqGti4OGxJZ31Yyw7gMXIlsQo1sTmiZgw",
        cx: "016480373837703253633:j39k3d3lx33" ];
}

port +console(uri: "clr://mscorlib/System/IO/Console", arity: 0);
```

Figure 30 - Port definitions in Akkadian

The service definition in the context of a port effectively exposes all the methods and events of the service as port commands and port events in the context of the containing smart object. Subsequently, the smart object that contains the service-providing port or other objects connected through their matching service-requesting ports can access the port's commands and events and react to them by either invoking commands or writing to port memories. An example can be seen in Figure 31 below.

```
when Activated()
    +search.Search(q:"Akkadian");

when +search.Search(*items)
{
    +search.cached = [ q: items.query, items: items ];
    +self.Log(items);
}
```

Figure 31 - Using a service-requesting port in Akkadian

A finished smart object program can then be deployed inside the smart object environment where its Abstract Syntax Tree (AST) form is evaluated by Akkadian's runtime environment (see 4.8.1).

4.7.2 Memory and Services

The smart object memory in Akkadian is defined by the language within the context of the object's service-providing ports, following the memory model in 4.4. In this regard, ports encapsulate an associative table that links string keys to a value of any type (see also 4.7.6). An object memory can be read using the square bracket "[]" operator, which follows a port reference in order to access that specific port's memory. The same notation is used for writing to the memory, with the values to be written appearing on the right side of the assignment "=" operator. Writes and reads on a port memory are atomic operations. Moreover, multiple atomic reads and writes are supported by separating the different memory locations with the comma "," operator. In this case, instead of referencing only one

memory location, an arbitrary number of memory locations, separated by comma, can appear in a read or write operation.

Whereas the key of a memory location is always a string, the language permits a value of any type to appear as the key, due to its implicit convertibility to a string (weak-typing). If an identifier appears as a memory location key, then, if an alias exists (see 4.7.3) with the same name as the identifier, the key is considered to be the alias' value. If there is no alias with the same name, then the key is considered to be a string that has the same name as the referenced identifier. Moreover, memory can be accessed through the dot "." operator, which is followed by an identifier whose string representation references the memory location that is associated with the given name.

Similarly to commands, a service-providing port memory can be accessed by its enclosing object and all the objects that are connected to it through their service-requesting ports. Therefore, multiple objects connected through their service-requesting ports, provided that the matching service-providing port has an arity greater than one, share the same memory. Reads and writes performed to the memory of a service-requesting port that is not connected are executed successfully, as if the port was connected, and when it does connect the performed operations are merged with the service-providing port's memory.

All the aforementioned syntactic and semantic details for accessing object and port memories can be seen in the example in Figure 32.

```
port +z;
port -z;

when Activated()
{
    +z.one = 1821;

    +z["one"];                          // 1821
    +z[one] = z.one - 368;
    +z.one;                             // 1453

    +z[one, "two", three ] = 1, 2, 3;
    +z[ "one", two, "three", four ];    // [ 1, 2, 3, nil ]

    -z.pi = 3.14;
    -z.result = 1 - (-z["pi"]);
    -z["result"];                         // -2.14
}
```

Figure 32 - Reading and writing memories

Akkadian's command invocation model is asynchronous. An invocation does not block the evaluation of the program and potential results are fed back to the invoking object as events. An invocation, in the context of a port, is realized through the dot "." operator, which is followed by the name of the target service method and its arguments. Since all expressions in Akkadian yield a value, the invocation of a command produces a Boolean result indicating whether the specific invocation could be handled by the underlying service. In general, an invocation can be handled by the underlying service, if the service implements a method with the same name and that method can accept the provided invocation arguments. Therefore, the yielded value simply indicates whether the target service method can be invoked, but it does not convey any information as to whether the invocation can actually succeed. This information is communicated exclusively via events (see also 4.7.3 and 4.7.4). Two example invocations for the service implemented in section 3.7, can be seen in Figure 33 below.

```
port +testService("ami:///Example/Echo");

when Activated()
{
    +testService.EchoUnicodeString("hello world");  // true
    +testService.DoNotEchoString(content: "Bye");   // false, since it does not exist
}
```

Figure 33 - Invoking a service in Akkadian

As mentioned before, services in Akkadian programs can be attached to service-providing ports using Uniform Resource Identifiers (URIs). The schema of a URI indicates the service technology and is subsequently evaluated by the indicated service-engine to identify the location and resolve the actual service. Since service instantiations are independent from the smart object model, they may not conform to Akkadian's asynchronous invocation mechanism. In this case, the adaptation of invocations from a specific service technology to the asynchronous invocation model expected by Akkadian is handled by the runtime environment's service engines (see 4.8.2), which can be dynamically loaded on demand via plug-ins. In this regard, invocations through service engines must follow three basic rules: (1) when the execution of a method is requested, the service engine must yield control immediately to the issuing object indicating whether it can handle the method invocation; (2) potential return values of synchronous request/response method invocations must be returned to the object through one or more events whose payload carries the result of the

execution; and (3) if the invocation can be handled but the actual service (temporarily) cannot be contacted or the actual execution of the method throws an exception, then it must produce events that have the same name as the ones that would carry the result if the invocation were successful, prefixed with the sharp "#" symbol (see also 4.7.4).

4.7.3 Capturing Events

In a smart object program events are either produced from port services or from port memory updates. In the former case, port services produce an event either as a result of the invocation of a command (i.e., (c, e) $\in$ I $\longleftrightarrow$ O) or because the underlying service is capable of emitting asynchronous events (i.e., (*nil*, e) $\in$ I* $\longleftrightarrow$ O). Moreover, an invocation may produce an event in the context of the object's service-providing port if and only if the invoked method cannot be handled by the underlying service. In this case, the produced event has the same name as the name of the invocation and as payload the invocation's arguments. For example in Figure 33 above, the "DoNotEchoString" invocation that does not exist in the context of the service that is attached to the port, produces a port event that has the name "DoNotEchoString" and a payload with the string "Bye" indexed with the key "content". Moreover, an event may be explicitly produced by an Akkadian program in the context of a service providing port using the "@" invocation syntax as seen in Figure 34. In this case, instead of a service invocation, this statement has as a result the emission of an event that has the same name as the identifier that succeeds the "@" symbol and a payload value equal to the invocation's arguments.

```
port +testService("ami:///Example/Echo");

when Activated()
    +testService.@EchoUnicodeString("Hello world");
```

Figure 34 - Explicitly producing an event in Akkadian

Updating or creating a new memory location produces a memory event that indicates the modified memory location in the context of the containing service-providing port. A memory location is created the first time a value is written to it. All memory write operations in Figure 32, thus, produce a memory event.

Regardless of the reason an event is emitted, in Akkadian, its presence can be checked with the *when* statement. The statement's syntax is: ***when*** *capture-expr statement*, where

capture-expr is a Boolean expression of either a *service-event-capture-expr* or a *memory-event-capture-expr*, and *statement* is any valid Akkadian statement. In EBNF-like syntax, this can be expressed as follows:

```
when-stmt := 'when' capture-expr statement;
capture-expr := capture-expr ('and' | 'or') capture-expr
      | 'not' capture-expr
      | service-event-capture-expr
      | memory-event-capture-expr
      ;
```

Essentially, a service-event capture expression (*eventmatch*) and a memory-event (*memmatch*) capture expression are satisfied when the respective predicates below are true. In Z, the symbol "Ξ" adds the referenced schema in the current one and also indicates that its post-state is not affected. We also assume three predicates, *valuematch*, *namematch*, and *payloadmatch* that check whether a memory element, an event name and its payload are, respectively, satisfied by the relevant capture expression. The operator "ran" obtains the range of a relation.

$$
\begin{array}{|l}
\Xi SO \\
memmatch, eventmatch : \{ true, false \} \\
valuematch : M \leftrightarrow \{ true, false \} \\
namematch : O \leftrightarrow \{ true, false \} \\
payloadmatch : O \leftrightarrow \{ true, false \} \\
v? : M \\
e? : O \\
p? : PP \\
\hline
memmatch = p? \in sp \land v? \in mem(p?) \land valuematch(v?) \\
eventmatch = p? \in sp \land e? \in \operatorname{ran} serv(p?) \land namematch(e?) \land payloadmatch(e?)
\end{array}
$$

When the capture expression is satisfied by all the conjunct event-capture expressions, then the statement is said to be "matched" and its associated actions are evaluated by Akkadian's runtime. Consider, e.g., the capture statement in Figure 35 below.

```
port +t("ami:///Sensor/Temperature/livingroom");

when +t.TemperatureChanged(*sensorId, temp < +t["low_temp_thr"]) and not +t.disabled
    +self.Log("Temperature too low");
```

Figure 35 - Event capture statement in Akkadian

This statement captures an event called "TemperatureChanged" (checked by *namematch*) that is emitted by the service attached to the service-providing port with name *t*, which is

contained in the object described by the Akkadian program. Also, this statement captures two memory locations (checked by *valuematch*) that are part of the aforementioned service-providing port (*t*). Those memory locations are named “low_temp_thr” and “disabled” respectively. Interpreting the inequality and the Boolean operators, this specific statement is triggered when the “TemperatureChanged” event is emitted, its second parameter is less than the number written in memory position “low_temp_thr”, and the value of the memory position “disabled” is evaluated to *false*. The event in the above statement has two named parameters in its payload. The second one is called “temp” (for temperature) and is compared (checked by *payloadmatch*) with the value of the memory location “low_temp_thr”.

In the same example, the first parameter of the payload denotes the id of the specific sensor that reported the temperature, and since it is not needed in this specific instance, we could have omitted it completely from the event-capture expression. Alternatively, we can use the parameter alias syntax as above, an identifier prefixed with ‘*’ (in which case *payloadmatch* evaluates to *true*), in order to be able to refer to that parameter with the given identifier in the capture statement’s actions. This is the only way to define value aliases in Akkadian as the ad-hoc declaration of variables inside a smart object program is not allowed. In this regard, the state of the program is distributed among its accessible port memories.

The actions of a “when” event-capture statement are evaluated every time the capture expression is matched. Testing of whether the capture expression is matched by the emitted events happens every time an event that participates in a specific event-capture expression is emitted to the containing object. This apparently includes memory events. Therefore, in the example above, if the “low_temp_thr” memory location of port “+t” is changed, the capture statement is reevaluated.

Capture statements can be nested in other capture statements. For example, the capture statements in Figure 36 are semantically equivalent since the outer statement’s actions do not contain any other statements except the inner event capture statement.

```
port +t("ami:///Sensor/Temperature/livingroom");
port -g;

when +t.TemperatureChanged(temp < 1)
{
    when -g.Humidity(v > 0.9)
        self.Log("Low temperature, high humidity");
}

when +t.TemperatureChanged(temp < 1) and -g.Humidity(v > 0.9)
    self.Log("Low temperature, high humidity");
```

Figure 36 - Nested capture statements in Akkadian

When a capture statement is matched, then all the encapsulated capture statements become eligible for being matched, i.e., they become active. Conversely, when a matched capture statement becomes unmatched, all the encapsulated capture statements cannot be matched even if their capture expression can be satisfied by the emitted events, i.e., they become inactive.

4.7.4 Extending Commands

A command invocation in the context of a specific port may fail for the following reasons: (a) the attached service cannot be contacted at all when Akkadian's runtime system tries to invoke its method, (b) whereas the attached service can be contacted, the requested method fails being invoked, and (c) the attached service's method is invoked but during its execution it throws an exception. Total failure to contact an attached service depends apparently on its implementation technology. For a service distributed over the network, it might mean that its perceived location is incorrect, the service is not running, or that there is a temporary network failure at some point between the client and the service. On the other hand for a local service realized as a DLL, this means either that the given path was incorrect, or that the DLL itself was deleted or has become inaccessible (due to restrictive file permissions). When a service can be contacted, failure to invoke one of its methods can be either due to the client program passing wrong number or types of arguments, or due to the values of arguments failing the service method's preconditions. Finally, a method invocation in the context of a service may produce an exception, regardless of the aforementioned factors, as part of its normal execution logic.

Regardless of the reason, when an invocation fails an event is emitted in the context of the containing port. The emitted event that has the same name as the event that delivers the

result of a successful invocation of the same method prefixed with the sharp "#" character. This exceptional event carries in its payload a string with a description of the error and can participate like any other event in the smart object program's capture statements. Figure 37 shows an example that uses the service implemented in section 3.7.

```
port +testService("ami:///Example/Echo");

when Activated()
    +testService.GetMessageInArgument();

when +testService.GetMessageInArgument(result == true, *msg) and not +self.disabled
    +self.Log(msg.priority, msg.msg, msg.number);

when +testService.#GetMessageInArgument(*err) or +self.disabled
    +self.Log("Advanced notifications are disabled [", err, "]");
```

Figure 37 - Capturing an exceptional event in Akkadian

In Akkadian this mechanism is considered a command extension mechanism, since it enables the provision of additional functionality in the context of a smart object when a method invocation fails. Moreover, Akkadian enables the extension of commands in the context of a smart object program by overriding or augmenting their functionality through event capture statements. In this regard, a smart object program may override the invocation of a port command by capturing an event that has the same name as the target service method prefixed with the "@" character. This overriding event carries the invocation's arguments as its payload. In this case, instead of invoking the actual service method that is attached to the port, Akkadian emits the overriding event regardless of whether it can satisfy one or more event-capture statements. An Akkadian program can then directly invoke a service method by using the direct port invocation expression, where a port reference is followed by parentheses that contain as a first argument the name of the method, followed by the method invocation's parameters. This functionality is depicted in Figure 38, where the overridden service method is invoked only when the "disabled" memory location evaluates to *false*.

```
port +testService("ami:///Example/Echo");

when Activated()
    +testService.GetMessageInArgument();

when +testService.@GetMessageInArgument() and not +self.disabled
{
    +self.Log("GetMessageInArgument invoked!");
    +testService("GetMessageInArgument"); // Invoke the actual service method
}
```

Figure 38 - Overriding a service method in Akkadian

4.7.5 Federations, Self and Path

In an Akkadian program, events are emitted in the context of service-providing ports. These events are forwarded to service-requesting ports as long as they are connected to matching service-providing ports. A service-requesting port, thus, can be connected to a matching service-providing port through the invocation of the “Link” command in the context of the self port. Upon successful connection, both ports receive the “Linked” event. Following the model in 4.6, Akkadian’s runtime environment implicitly creates a service-providing port called “self” for every object, inside which a service implemented by the runtime environment is attached. Apart from the methods required by the model, the environment’s service instance implements additional methods that control some aspects of a deployed smart object depending on the nature of the environment and the characteristics it imbues to its contained objects. Since our implementation of the smart object environment (and Akkadian’s runtime environment) enables the representation of smart objects as visual entities inside an infinite two-dimensional plane, the environment service attached to the self port allows objects to set their position, color, displayed title, etc. More details about the implementation of the runtime environment are given in section 4.8.1. In addition to the attached service, it is the responsibility of the environment to associate the object’s name with the “name” memory location of the self port.

Figure 39 below, shows an example invocation for establishing a connection between two matching ports and accessing the names of both the current object and the object that is being connected to. This example also shows two properties of an Akkadian program’s mechanism for resolving port references. First of all, a local port reference can be represented as a simple identifier, omitting the “+” or “-“ prefix. In this case, if there are no ambiguities, the identifier uniquely references the service-providing or service-requesting port that has the same name. If there is one service-requesting and one service-providing port with the same name, then the identifier syntax chooses the service-providing port for the resolution of the port reference. The second property concerns the implicit type convertibility in the context of an Akkadian program. In the same example below, the “InScope” event captured in the context of the self port, has as a payload a reference to the object that entered the current object’s scope (see also 4.6). Therefore, since the port memory access operation for obtaining the “name” memory location is applied to the object

reference, the latter is implicitly converted to a reference to that object's self port prior to accessing the memory. More details about the implicit convertibility of Akkadian types are provided in section 4.7.6.

```
port +C("echo://control-port");
port -C;

when +self.InScope(*o) and o.name == self.name
    +self.Link(-C, o);

when -C.GoodBye()
    +self.Unlink(-C);
```

Figure 39 - Establishing a connection between two matching ports in Akkadian

Moreover, as seen in the examples in the previous subsections (e.g., in Figure 38), an event capture expression that is not explicitly qualified with a port reference is resolved against the containing object's self port.

Port references in Akkadian programs can also be resolved through paths (see 4.3). In this sense, each smart object can refer to another object in the same federation through Akkadian's path expression. In a federation, a port sequence defined in the context of a specific object can describe a path from that object to another one by means of the established connections through the ports that appear in the expressed path sequence. A path expression can be open or closed. An open path expression describes a path as a sequence of connected ports but cannot be resolved to an actual port reference or object reference since it has to be projected (applied) on an object reference first. Therefore, when an open path expression is applied to a specific object, it becomes closed and can subsequently be resolved with that object as a reference point. An open path expression can become closed with a specific object by using operator "()" on the open path value with the target object reference as the operator's parameter.

```
port +C("echo://control-port");
port -C;

when InScope(*o) and o.name == self.name
    +self.Link(-C, o);

self.openPathExpr = -C-C-C;

when self.openPathExpr(self).Linked() and +C+C(self).Linked()
    self.Log("there are three objects in front and two behind");
```

Figure 40 - Path expressions in Akkadian

This can be seen in Figure 40 where an open path expression value is stored in self port's memory and subsequently becomes closed and participates in the capture statement. This particular capture statement also contains another closed path reference and both are evaluated in the context of the containing object. In this example the self port reference applied to the closed path expression value is implicitly converted to an object reference to the port's container object.

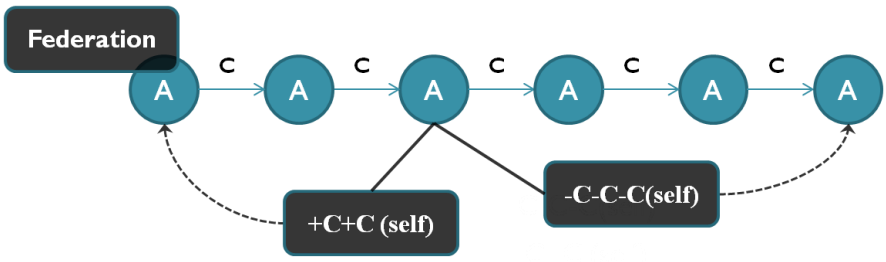


Figure 41 - Resolution of two closed path expressions within the context of a smart object

The evaluation of the program in Figure 40 has as a result the execution of the "Log" invocation inside the capture statement when the connections depicted in Figure 41 are established, in the context of the indicated object "A". In general, path expressions can participate in capture statements through which an object can capture the events of other objects. In this case, if any of the ports referenced by a path expression is not connected, then the path is invalid, i.e., it is dangling. Capture expressions that contain dangling paths cannot be matched. However, as soon as the path becomes valid, the affected expressions are evaluated and are subsequently tested against the emitted events on the target object. This is the reason the aforementioned smart object program works as expected even when, at the beginning (when it is activated), none of its ports are connected.

4.7.6 Types and Other Statements

Akkadian currently supports the types shown in Table 7. Except from the "Handler" type, all other types are implicitly convertible to strings and properly formatted strings are also implicitly convertible to the basic types. Also, all types are implicitly convertible to a list or "Struct" that contains a single element of that type. In this sense, the language's type system follows the weak typing model. The convertibility of all types can be seen in Table 8, where the symbol indicates whether the table's rows can be converted to the types

indicated in its columns. Values of all the supported types can be stored in port memories, used for arguments in invocations, or for the payload of events.

A value of type “Struct” represents a tuple of optionally named values of any type. Struct values, on one hand, behave like the “structure” (struct) type in popular programming languages (e.g. C, C#), and on the other hand, keep their values ordered allowing for sequential indexed access. The “List” type provides an ordered sequence of arbitrary length. The “Port” and “Object” types are used for port and smart object references respectively. Finally, the “Handler” value can hold an event capture statement through which can then be instantiated and reused in the context of a specific smart object or another “Handler” value (using the call operator).

Table 7 - Akkadian types

Type	Description	Supported operators
Number	Double precision real number	+, -, *, /, ** (power), <, >, <=, >=, ==, !=, call
String	Unicode character sequence	+, -, *, <, >, <=, >=, ==, !=, call
Boolean	Boolean values: true, false	and, or, not, ==, !=
List	Sequence of values	+, -, *, /, <, >, <=, >=, ==, !=
Struct	Tuple with named members	., [], +, -, /, <, >, <=, >=, ==, !=
Port	Reference to a smart object port	., [], ==, !=, call
Path	An open or closed path value	+, -, <, >, <=, >=, ==, !=, call
Handler	Capture statement as a first-class value	==, !=, call
Object	Reference to a smart object	==, !=

Values of all the supported types are immutable and equality/inequality operators (==, !=) perform deep value comparisons – not just references to values. For example comparing a list with another list, compares recursively the list’s values one by one. Also, in the case of a “Handler” value the deep comparison compares the statement’s AST representation node-by-node, since the AST representation is Akkadian’s executable form. From Table 7 above, it is worth noting that the division operator for list and struct values tests whether the former contains the value appearing on the right, which is shorthand for the comparison operators that test the subset/proper-subset relationship between the provided sequences.

Table 8 - Akkadian's implicit type conversions, read by row

	Number	String	Boolean	List	Struct	Port	Path	Handler	Object
Number	✓	✓	✓	✓	✓	×	×	×	×
String	✓	✓	✓	✓	✓	×	×	×	×
Boolean	✓	✓	✓	✓	✓	×	×	×	×
List	✓	✓	✓	✓	✓	×	×	×	×
Struct	✓	✓	✓	✓	✓	×	×	×	×
Port	✓	✓	×	✓	✓	✓	✓	×	✓
Path	✓	✓	✓	✓	✓	✓	✓	×	✓
Handler	×	×	×	✓	✓	×	×	✓	✓
Object	×	✓	×	✓	✓	✓	✓	×	✓

Logical conditions can be combined using the logical operators “and”, and “or”. A condition can be negated using the unary operator “not”.

The “for” statement can be used for iterating over a sequence of values. A “for” statement is expressed using the keywords *for*, and *in*. The *for* keyword is followed by an identifier list, which subsequently, inside the statement’s body, can be used as an alias to access the value of the current iteration. The identifier is followed by the *in* keyword, which in turn is followed by an expression-list that produces an ordered sequence of values. The iteration continues until all values in the expression list have been consumed. In case some values contained in the sequences of the expression list are consumed before the others, their associated aliases are set to the null value. An example invocation of the for-statement can be seen in Figure 42 below.

```
when Activated()
{
    for i, j, k in [ 1, 2, 3 ], [ 1, 2, 3, 4 ], [ 1, 2, 3, 4, 5 ]
        self.Log("(", i, j, k, ") ");
    //-> (1 1 1) (2 2 2) (3 3 3) (null 4 4) (null null 5)
}
```

Figure 42 – For statement in Akkadian

The “if” statement can be used to enable the conditional evaluation of a set of Akkadian statements, based on the truth-value of the condition that appears after the *if* keyword.

Finally, the "while" statement, evaluates its contained statements as long as the given condition evaluates to true. When the condition changes to false, the iteration stops.

4.8 Implementation

Akkadian's programming and runtime environment is implemented as a graphical tool in which developers can program and deploy smart objects. This tool, named ObjectivSim, contains all the necessary components for editing Akkadian programs, with the extra smart object characteristics added by the current instantiation of the environment, and executing, simulating and visualizing their federations and interactions. In ObjectivSim, when an object is defined in terms of its acquired characteristics and its program, it can be deployed inside the environment where it is activated and starts interacting with the other objects.

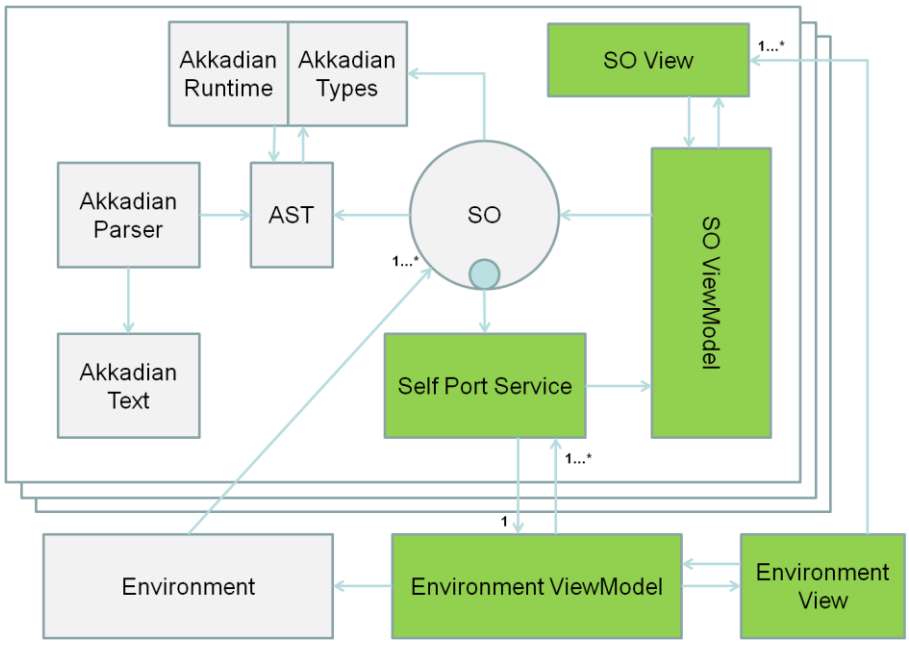


Figure 43 - Smart Object Environment Architecture

Figure 43 shows the architecture of the smart object environment including its graphical components that are provided by ObjectivSim. In the figure above, the light gray entities represent the components provided by the implementation of Akkadian's runtime, which is completely independent of any graphical components and can be reused for instantiating different smart object environments with different representations, characteristics and behavior. The green entities represent the software components that are provided by

ObjectivSim's graphical environment. Furthermore, the implementation of ObjectivSim's graphical user interface and the visualization of the smart object environment, follow the Model View View-Model (MVVM) architectural pattern [146]. MVVM facilitates a clear and distinct separation of the graphical user interface implementation from the design and implementation of the backend logic. In general, the Model in MVVM refers to either the domain model or the data access layer that represents the content and the state of the target entity to be visualized. In this regard, for the implementation of the smart object environment in ObjectivSim, the smart object (SO) and its environment (Environment) are the models in Figure 43. The View refers to the graphical elements that appear in a Graphical User Interface (GUI) and can be manipulated by the user. Finally, the View-Model is an abstraction of the view that also mediates the interactions between the View and the Model. In Figure 43, the Views and View Models are clearly marked. This architecture effectively provides another layer of abstraction for the visualization of smart objects in their environment, since View Models can be reused for realizing completely different graphical representations that can even work in parallel with the others.

An Akkadian program is first transformed by Akkadian's parser to an Abstract Syntax Tree (AST) form, which is essentially the executable form of a smart object program. In this regard, only the program's AST representation is needed by Akkadian's runtime for realizing a smart object's functionality. This further enhances the future extensibility and reusability of the implementation approach, since it enables the development of different language frontends that may describe with different syntax the different aspects of the functionality of a smart object without changing or affecting the implementation of the runtime smart object model. The AST representation of a simple Akkadian program that contains a while-statement can be seen in Figure 44.

Figure 44 - The Abstract Syntax Tree of an Akkadian while statement

Apart from the runtime environment, ObjectivSim implements graphical components for programming smart objects, visualizing the program's AST form, editing the smart object properties that are acquired by the graphical environment, and logging messages generated during the interactions between smart objects. All these graphical components in ObjectivSim can be seen in Figure 45 below.

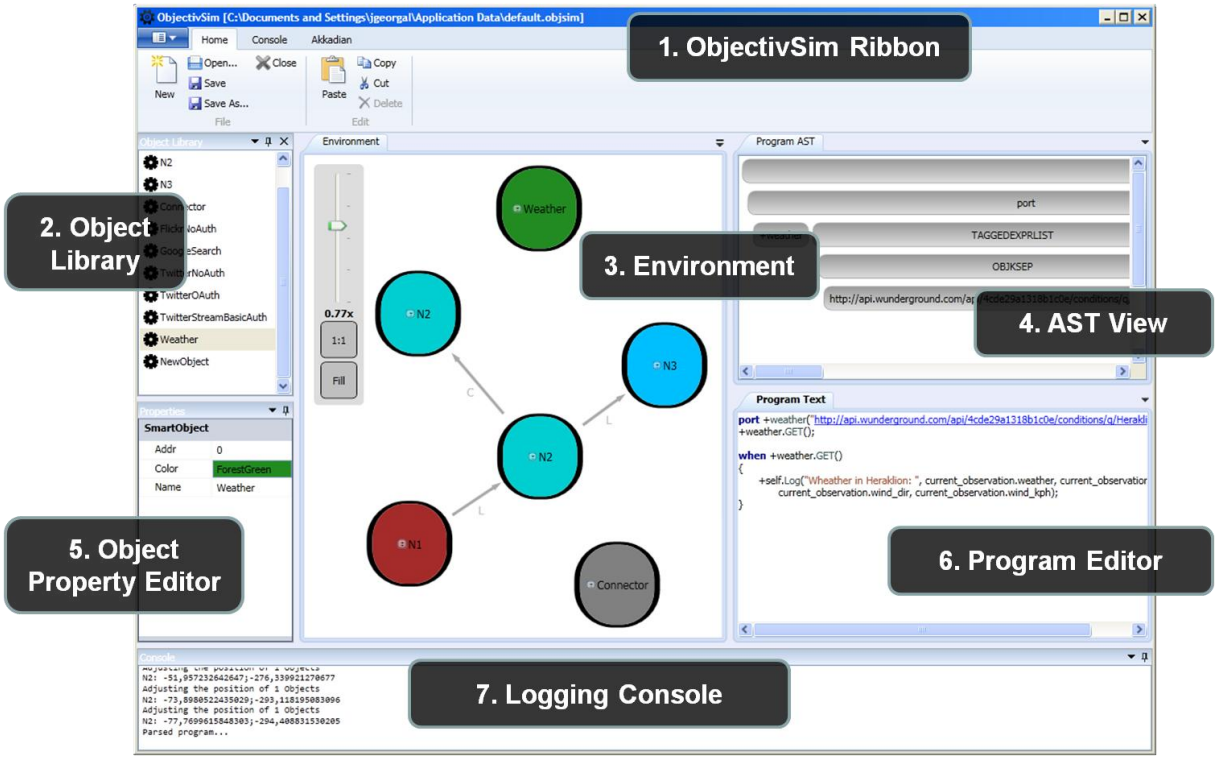


Figure 45 - ObjectivSim with all its user interface components

Akkadian's parser and runtime environment (the light gray components in Figure 43) were implemented in C# under the .NET runtime, utilizing the ANTLR parser generator [147]. ObjectivSim and all its graphical components, including the graphical instantiation of the

smart object environment (the green components in Figure 43) were implemented also in C# under the .NET runtime, utilizing the Windows Presentation Foundation (WPF)[6] for implementing the graphical user interfaces. Additionally, ObjectivSim utilizes the Microsoft Prism framework [148] for synthesizing its overall user interface from its constituent loosely coupled graphical components which are loaded on demand and attached to the application at runtime. ObjectivSim's component-based design allows the loosely coupled tools and graphical components to be implemented and evolve independently but also to be easily and seamlessly integrated into the overall application. This greatly enhances the extensibility potential and evolution path of ObjectivSim as a solid integrated platform for programming AmI environments.

4.8.1 Smart Object Environment

The smart object environment in ObjectivSim is realized as an infinite in size, two-dimensional plane. There is no limitation neither on its size nor on the number of objects it can host, and can be zoomed in and out to allow for a better view of all objects that operate in it. Figure 46 shows a zoomed out view of the environment that contains objects that have different colors, with some of them connected in federations through their ports. A smart object in the environment is represented as a round graphical colored shape with a black outline. This shape contains a title, that usually denotes the represented object's name, and a small button, which when pressed displays the current object's control panel. The control panel displays more information about the current smart object and enables the generation of synthetic events by the user through user interface elements offered by ObjectivSim's port services (see 4.8.2). A connection between two objects through two matching ports with name "p" is displayed as an arrow with the label "p", that starts from the object that contains the service-requesting port "-p" and ends to the object with the service-providing port "+p". A zoomed-in view of the environment with the aforementioned elements clearly visible can be seen in Figure 47.

[6] http://msdn.microsoft.com/en-us/library/ms754130.aspx

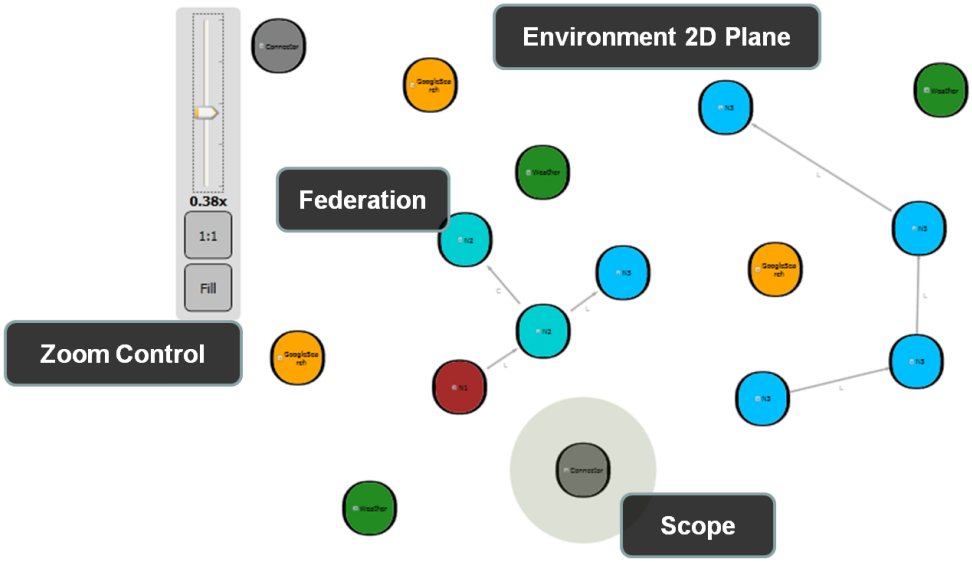


Figure 46 - ObjectivSim's smart object environment

ObjectivSim's environment offers a self-port service to each of its contained smart objects that gives them the ability to establish and break connections (Link / Linked, Unlink / Unlinked) and to be notified when other objects enter or leave their scope (InScope, OutScope). In this regard, it offers all commands and events specified by the smart object model (see 4.6). Additionally, it enables smart objects to control some of the properties that they acquire by being in operation within this specific implementation of the graphical two-dimensional environment. Owing to the design and architecture of the platform, it is possible to create alternative smart object environments (e.g., as 3D spaces) by only replacing the environment-specific components (the green components in Figure 43) and, thus, provide completely different properties to the hosted smart objects. In this instance, however, ObjectivSim's environment offers the following commands to smart objects:

- *Display*, which accepts a string value and changes the displayed title on the visual representation of the target smart object; it does not change the object's name however
- *Color*, which accepts a string value that denotes either a color's name or a string representation of a color's RGB value and changes the object representation's color
- *Move*, which accepts two number values, the first representing the change of distance along the x-axis (dx) and the second the change of distance along the y-axis (dy) and adjusts the position of the object in the environment accordingly

- *Log*, which accepts an arbitrary number of values of any type and displays on ObjectivSim's console their string representations

Additionally, it emits the event *Moved* every time the position of an object changes, either as a result of the invocation of the *Move* command or due to being dragged by the user with the mouse via the graphical representation of the smart object environment.

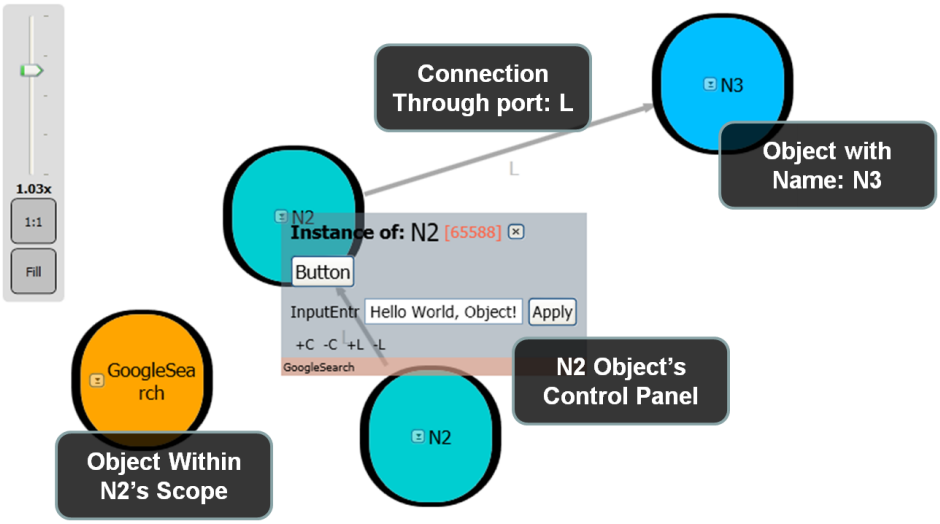


Figure 47 - A zoomed-in view of ObjectivSim's environment

In the proposed smart object environment, as mentioned before, the scope of an object (see 4.5) is realized as a proximity relation between smart objects. In this sense, an object enters another object's scope as soon as their Euclidean distance inside the smart object environment becomes shorter than a fixed, predefined value, as seen in Figure 46. Symmetrically, an object leaves from another object's scope as soon as their distance becomes longer than the predefined value. From this follows that in ObjectivSim's environment, when an object A is inside the scope of an object B, then object B is inside the scope of object A. As part of our future work, we plan to allow the definition of scope programmatically inside Akkadian programs for enabling the manipulation of its properties and incorporating security primitives for establishing a trust relationship among object federations.

4.8.2 Service Engines

Services in the context of an Akkadian program are referenced through URIs [145] whose schema uniquely identifies the technology on which the service is built. Subsequently, the

URI is resolved by the appropriate service-engine in order to locate, resolve and instantiate a service instance that can in turn be utilized in the context of smart object ports for invoking commands and delivering events. The architecture of the components responsible for instantiating services can be seen in Figure 48 below.

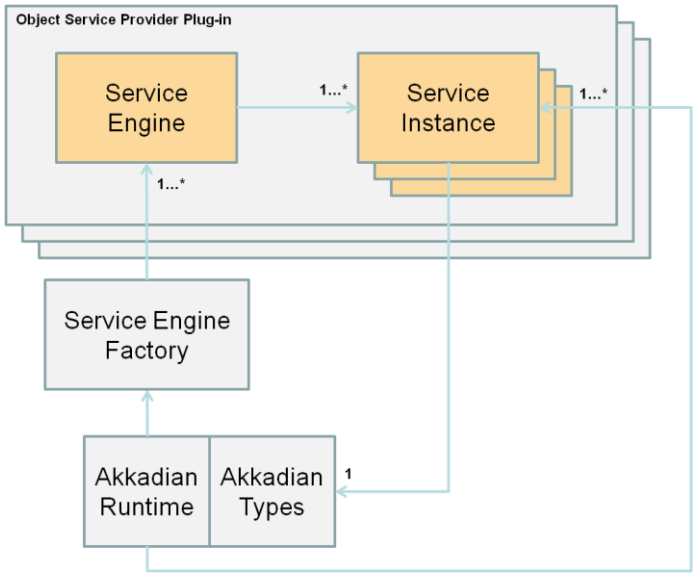


Figure 48 - Service-engine architecture

The implementation of different service-engines and the implementation of the services that they are responsible for instantiating are loaded dynamically (as DLLs) by Akkadian's runtime on demand (orange components in Figure 48). When a service with an unknown scheme with name "s-name" is referenced by a service-providing port, the service-engine factory checks whether there is a plug-in (DLL) available (in the current directory) with the suffix "*s-name*_OSP" (OSP stands for Object Service Provider). If the plug-in can be loaded successfully, the service-engine factory keeps it for future requests (i.e., the scheme is no longer unknown) and requests from it the instantiation of the service denoted by the source URI. The entry point for a service engine that handles a specific scheme is specified as class meta-data through .NET's class attribute syntax. An example definition of a service-engine that can handle the "lib" scheme can be seen in Figure 49.

```
[ServiceEngineFactory(Scheme = "lib", Author = "jgeorgal", Version = "0.1")]
public static class LibServiceEngine
{
    public static void Initialize() { }
    public static void CleanUp() { }

    public static IServiceRef Create(Uri uri, RtStructValue options)
    {
```

```
            switch (uri.Host)
            {
                case "timer": return new TimerService(uri);
                case "random": return new RandomService(uri);
                default: return new NullService(uri);
            }
        }
        public static void Destroy(IServiceRef service, Uri uri)
        {
        }
}
```

Figure 49 - Definition of a service engine for a specific scheme

In the context of Akkadian's runtime and specifically, in the context of service-providing ports, service instances are accessible through a unified interface that simply enables the invocation of service methods and the delivery of service events. For the delivery of events, service interfaces utilize Microsoft's Reactive Extensions library [149], which enables the representation of asynchronous data streams with the "Observable" abstraction through which clients are notified about the occurrence of asynchronous events. The definition of the unified interface for services can be seen in Figure 50.

```
public interface IServiceRef
{
    Uri EntryPoint { get; }
    int DefaultArity { get; }

    // First item in tuple is whether invocation was sucessful or not
    //
    IObservable<Tuple<bool, string, RtStructValue>> OnEvent { get; }

    // Return value: true means that the method invocation can be/was handled
    //
    bool CanHandleMethod(string method, RtStructValue args);
    bool InvokeMethod(string method, RtStructValue args);
}
```

Figure 50 – Definition of the interface for services

Akkadian's runtime defines and implements three basic service engines. The "NullServiceEngine" is not associated with any schema and thus cannot be explicitly associated with service-providing ports in an Akkadian program. Its purpose is being used by other service-engines for instantiating service instances when they cannot instantiate the requested service. All invocations on null-service instances return them back as errors. The "EchoServiceEngine" can be referenced through the "echo" scheme, pretends to handle all invocations and just logs them onto a file-stream or the standard output of the smart object environment. Finally, the standard "LibServiceEngine" can be referenced through the "lib" scheme and implements some standard utility services such as a timer service and a random number generator service.

ObjectivSim's smart object environment, defines two service engines; the "ButtonServiceEngine" and the "InputServiceEngine" that are associated with the schemes "button" and "input" respectively. Instances created by these service engines instantiate a button or a text input field UI element onto the containing smart object's control panel. Subsequently, when the button is pressed on the panel's UI, the associated service emits the "Pressed" event without any payload. On the other hand, when the input field's "Apply" button is pressed (see also Figure 47), the associated service instance emits the "Input" event that has in its payload the text of the input field. An example of using the aforementioned standard services can be seen in Figure 51 below.

```
port +echo("echo://testing/invocations");
port +timer("lib://timer/seconds/1");
port +button("button:///Button Name");
port +input("input:///Input Name");

when +timer.Tick()
    self.Log(msg: "Hello", from: "Timer Robot", every: "second");

when +button.Pressed()
    self.Log("Button is pressed");

when +input.Input(*value)
    self.Log("Input:", value, "was given");
```

Figure 51 - Using standard services in an Akkadian program

Apart from the aforementioned ones, three additional service-engines have been developed independently of ObjectivSim and Akkadian's runtime, each realized as a distinct plug-in loaded on demand. The first plug-in, the "CLRServiceEngine", which is associated with the "clr" scheme, uses .NET's reflection API to instantiate .NET objects, invoke their methods and receive their emitted events (i.e., those defined through the *event* keyword in C#). The CLR service engine, can additionally instantiate services that manipulate static .NET classes – e.g., the static File class[7]. For command invocations, if the corresponding .NET object-methods accept "out" parameters (i.e., references for storing values from inside a method), then they should be omitted from the port command invocation arguments and are returned, along with other return-values, as events. On the other hand, if the corresponding .NET object-methods accept "ref" parameters (i.e., references for both passing and storing values from inside a method), then they should be provided by the port command invocation arguments; they are also returned, along with other return-values, as

[7] http://msdn.microsoft.com/en-us/library/system.io.file.aspx

events. Return values from method invocations (including “out” and “ref” parameters) are fed-back to the service-providing port through an event that has the same name as the invoked method. When a .NET object-method raises an exception an event is emitted in the context of the containing port with the same name as the invoked method and the sharp “#” prefix. Figure 52 shows an example of using .NET classes in Akkadian.

```
port +file("clr:///System/IO/File");

when self.Activated()
    file.ReadAllText("C:\\AUTOEXEC.BAT");

when file.ReadAllText(*text)
    self.Log("File contents:", text);

when file.#ReadAllText(*error)
    self.Log("Some exception occurred:", error);
```

Figure 52 - Using .NET classes in Akkadian

The “AmIServiceEngine”, associated with the “ami” scheme, enables the utilization of services implemented on top of the proposed middleware described in Chapter 3. Through service-providing ports, associated with the resolved services, an Akkadian program can invoke service methods and receive service events. In this regard, the AmI service engine enables any service that is running or that can be started (see 3.5.1) in the context of the AmI environment to offer its functionality to smart object programs. Similarly to the CLR service engine, for command invocations, if the corresponding service-methods accept “out” parameters (i.e., references for storing values from inside a method), then they should be omitted from the port command invocation arguments and are returned, along with other return-values, as events. Moreover, for service-methods accepting “inout” parameters (i.e., references for both passing and storing values from inside a method), then they should be provided by the port command invocation arguments; and are also returned, along with other return-values, as events. Return values from method invocations (including “out” and “inout” parameters) are fed-back to the service-providing port through an event that has the same name as the invoked service method. When a method cannot be invoked (see 4.7.4), an event is emitted in the context of the containing port with the same name as the invoked method and the sharp “#” prefix. Figure 53 shows an example of using AmI services in Akkadian, referencing the AmI service described in section 3.7.

```
port +v("input:///Value/Hello");
port +testService("ami:///Example/Echo");

when +v.Input(*value)
{
     // out argument is omitted
    +testService.GetMessageInArgument();

     // result is return value, msg is the returned out argument (omitted above)
     when +testService.GetMessageInArgument(result == true, *msg)
        +self.Log(msg.priority, msg.msg, msg.number);

    +testService.EchoUnicodeString(value);

    when +testService.EchoUnicodeString(*result)
        +self.Display(result);

    when +TestService.#EchoUnicodeString(*error)
        +self.Log("Some error occurred", error);
}

when +testService.AdvancedNotification(*msg)
    +self.Log("Received event:", msg);
```

Figure 53 - Using AmI services in Akkadian

Finally, the "RestServiceEngine", associated with the "http" and "https" schemes, enables the invocation of REST services [122] that are accessible on the Internet. Since the invocation semantics of REST services are different than services in traditional service-oriented technologies, the name of a port invocation command does not correspond to the actual HTTP command (e.g., GET, POST, HEAD, etc.) that is projected on a URL web resource. The actual command is by default "GET" and can be overridden through the "_http_method" invocation parameter (e.g., _http_method: "POST"). The name of the invocation simply denotes the name of the event that will be emitted in the context of the containing service-providing port, which carries the result of the invoked HTTP method onto the URL resource. A port definition that targets REST services may also contain a set of parameters (params) that are always transferred along with the invocation parameters, a description of the authentication methods and credentials (auth) needed for accessing the referenced service, and a stream directive (stream) that informs the service engine that a particular REST resource will be continuously sending messages. When only part of the URL resource is referenced in the context of the service-providing port, the "_http_resource" invocation parameter may define the missing segments. An example of an Akkadian program using REST services can be seen in Figure 54 below. In this example, the "+twitterStream.Stream" event is continuously emitted throughout the lifetime of the object without the need of invoking the "Stream" command more than once. Actually, a second invocation over this

resource would have as a result the creation of a second stream that continuously delivers events in the context of the containing smart object.

```
port +twitter
{
    uri "http://search.twitter.com/search.json";
    params [ lang: "el", result_type: "mixed", rpp: 10 ];
}

port +twitterStream
{
    arity 0;

    uri "https://stream.twitter.com/1.1/statuses/sample.json";

    auth [ method: basic,
        username: "jgeorgal",
        password: "********" ];
    stream true;
}

when +twitter.Tweets(*results)
    for r in results
        +self.Log(r.text, r.from_user);

when +twitterStream.Stream(*tweet) or +twitterStream.#Stream(*err)
{
    if err != null
        +self.Log("Error reading twitter stream:", err);
    else
        +self.Log(tweet);
}

when +self.Activated()
{
    +twitter.Tweets(q:"Ναβουχοδονόσορας");
    +twitterStream.Stream();
}
```

Figure 54 - Using REST services in Akkadian

4.8.3 Development Tool

ObjectivSim, apart from the smart object environment, offers functionality for making easier the development of smart object programs in Akkadian and their deployment inside the smart object environment. Figure 45 provides a view of the Akkadian program editor (6) and the AST view (4) of the abstract syntax tree that is generated by parsing the edited Akkadian program. The program editor provides basic editing capabilities with full syntax highlighting support, implemented on top of the AvalonEdit editing component[8]. The AST view is updated in real-time as the smart object program is being edited by the programmer. The program in the editor, in turn, always refers to a specific object inside the object library and

[8] https://github.com/icsharpcode/SharpDevelop/wiki/AvalonEdit

is automatically saved within the context of the associated object. Figure 45 also shows the smart object property editor (5), which is also associated with the selected object inside the object library (2). The edited smart object properties, which are attached to objects by ObjectivSim's implementation of the smart object environment, are saved automatically in the context of the associated object. As soon as the programmer finishes editing the selected object's Akkadian program and its properties, the object can be dragged from the object library and dropped inside the smart object environment where it is activated.

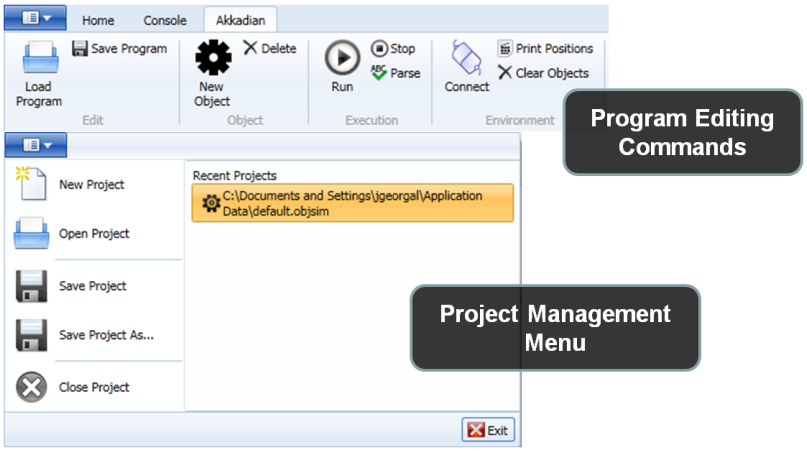


Figure 55 - Development support in ObjectivSim

Moreover, the whole setup of ObjectivSim's components, all the objects inside the object library and all the deployed objects inside the smart object environment, can be saved and loaded as a single project file. Figure 55 above, shows the project management menu and some convenience editing commands provided by ObjectivSim's development component.

4.9 External Smart Objects

In the previous sections, we demonstrated the methodology through which smart objects programmed in Akkadian can incorporate and utilize the functionalities offered by the resources available in AmI environments. The service abstraction in the smart object model, its layered architecture and modular implementation in the context of Akkadian's runtime, also allowed the utilization of resources beyond those provided by the AmI environment, extending the design and implementation space to Internet resources and .NET objects. Towards this direction, we propose the concept of External Smart Objects for further expanding the design space for incorporating into the AmI environment functionality

offered by mobile devices. Conceptually, this notion can be seen in Figure 56 below, which, in the context of the design and implementation space of AmI environments, shows the resources at the core (designed and implemented as services), followed by the smart object design space (designed and implemented as smart object programs in Akkadian), which is finally expanded to the mobile device design space (designed and implemented as external smart objects).

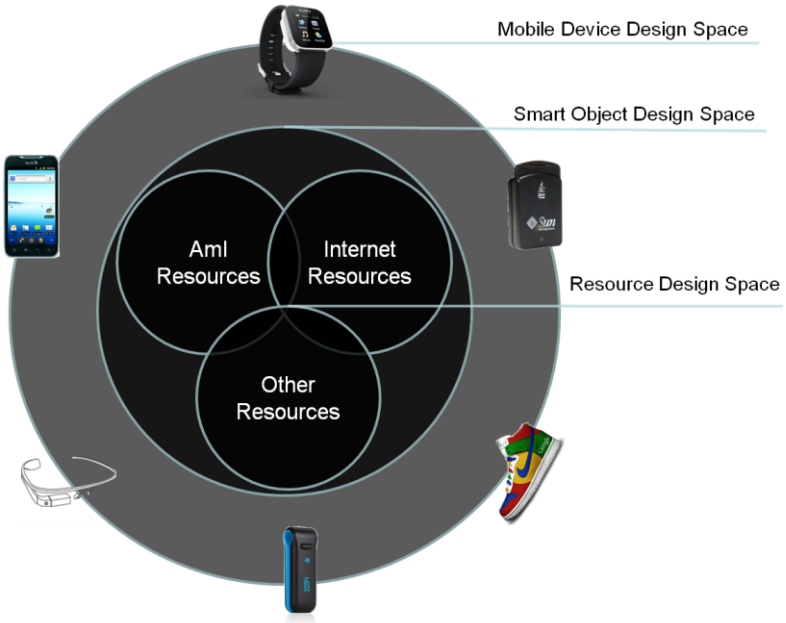


Figure 56 - Expanding the design space with external smart objects

The motivation for proposing the notion of external smart objects stems from the fact that mobile devices can be naturally modeled as smart objects but are not conceptually well-suited for being modeled as services. First of all, they cannot be considered as AmI resources and are not inherently part of the AmI environment. Their key characteristics are that they are self-contained, operate autonomously and move around, potentially entering and exiting AmI environments multiple times throughout their operational lifetime. Secondly, the notion of mobility and ad-hoc provision and removal of functionality within the AmI environment cannot be easily expressed with services, especially in the proposed service middleware (Chapter 3), where, by design, services are persistent and robust even in the case of unexpected failures. This design and the resultant guarantees are ideal for modeling the omnipresent AmI resources, but ultimately fail to intuitively model mobile devices. Furthermore, even if we introduced different kinds of services with relaxed guarantees in order to model those devices as “mobile services”, we would come across

issues stemming from the resource-limited nature of many kinds of mobile devices. Thirdly, due to the aforementioned resource limitations and the fact that mobile devices are usually self-contained with respect to the functionality they utilize and the operations they perform, an attempt to model them exclusively as smart objects implemented in Akkadian that use the device's functionality through individual services exported by the device is also an inadequate solution.

In this regard, we propose the modeling of mobile devices as external smart objects that on one hand fully implement their functionality inside their own context but can, at the same time, operate inside the smart object environment, offer their functionality to all other smart objects and use the functionality of all other smart objects, regardless of whether those objects are implemented in Akkadian or are external self-contained devices. Therefore, from the point of view of the smart object environment, external smart objects are indistinguishable from smart objects programmed in Akkadian.

External smart objects are realized in terms of the asynchronous messages they exchange over a remote connection with a proxy smart object that links the external mobile device with the smart object environment and vice versa. The smart object environment provides an endpoint for establishing remote connections with mobile devices, which result in the deployment of a smart object that maintains a persistent bidirectional link with the mobile device and effectively acts as its proxy within the environment. This process is depicted in Figure 57. In this regard, a remote program (running on a mobile device) and its proxy smart object connected via a remote bidirectional link are collectively referred to as an "external smart object". An external smart object that is powered-off or is disconnected from its proxy is immediately deactivated and removed from the smart object environment.

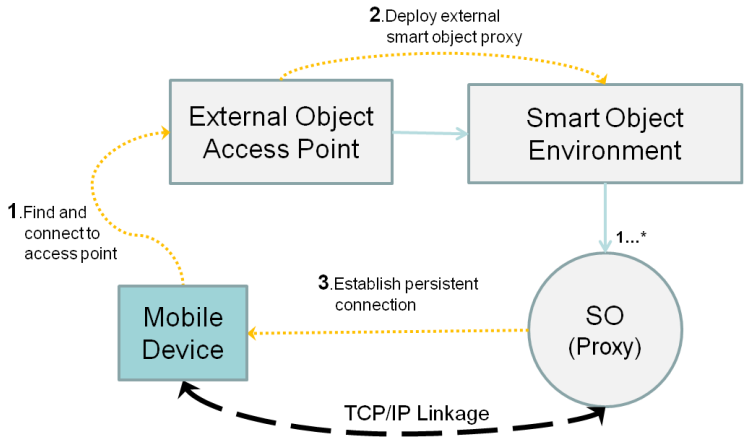


Figure 57 - Process for deploying an external smart object in the environment

In our implementation of external smart objects, we have realized the remote bidirectional link as a TCP/IP connection over the network. The use of TCP/IP is justified by the fact that it is widely supported by even the most resource constrained devices and allows for reliable delivery of messages even over noisy wireless network infrastructures. Additionally, through our definition of the bidirectional link abstraction, it is possible to extend our implementation for enabling connections through Bluetooth [52] or ZigBee [150] protocols. The remote mobile device in this case does not have any dependencies on any of the libraries provided by the proposed service middleware or the smart object environment. The only requirement is being able to establish low-level connections with remote TCP/IP servers. Subsequently, the communication is realized over a lightweight line-delimited text-based protocol.

The proxy smart object deployed in the environment as the result of a mobile device request is also associated with an Akkadian program and behaves identically to any other object except that it defines the notion of extrovert service-requesting and service-providing ports. An extrovert service-requesting port is a service-requesting port that forwards all its received events and memory update operations over an established bidirectional link. In the context of a proxy smart object, all service-requesting ports are extrovert. On the other hand, an extrovert service-providing port is a service-providing port that is not associated with any service and simply forwards all the command invocations and memory access operations over the established bidirectional link. In the context of a proxy smart object, all service-providing ports that are not associated with a specific service are extrovert. Finally, the self port of a proxy smart object behaves as both an extrovert port, forwarding all

memory access operations and events, and as an introvert port, handling the command requests that are implemented by its associated environment service.

Extrovert and introvert (service-providing) ports are created in the context of an external smart object through the transmission of text messages delimited with newline characters that contain valid Akkadian statements. A remote mobile device, in the context of an external object, can invoke services that are provided through its introvert service-providing ports or its connected extrovert service-requesting ports, again by transmitting valid command invocation Akkadian statements (see 4.7.2). When an external object throughout its lifetime exchanges messages only for invoking commands, emitting events and updating memory in the context of its contained ports, it is referred to as a *pure* external smart object. An object that in addition to the aforementioned behavior also defines and transmits Akkadian "when" statements for handling and responding to events and memory operations within the context of its proxy smart object, it is referred to as a *hybrid* external smart object. A hybrid external smart object, thus, effectively implements part of its functionality within the context of an Akkadian program and part in the context of the remote mobile device. At the other side, all messages transmitted asynchronously to the remote mobile device (notifications) are comprised of a single character that denotes the type of the notification and a triple that contains the port that the notification concerns, the name associated with the notification and the value of the notification. All notifications are delimited with the newline character. The description of the notification protocol can be seen in Table 9 below, where RP denotes service-requesting ports and PP denotes service-providing ports.

Table 9 - External smart object notification protocol description

Notification Type	Character	Port	Name	Value
Memory Update	.	RP or PP	Memory location	Memory value
Event Received	@	RP or PP	Event name	Event payload
Command Invocation	$	PP	Invocation name	Invocation parameters

An example of creating a hybrid external smart object that contains a "when" statement to react when an object enters within its scope can be seen in Figure 58, which also shows the

notifications sent back to the remote mobile device due to emitted events and the invoked command in the context of the hybrid object's self port. The remote object in this case is simulated through a terminal application we have developed that connects to the smart object environment and accepts commands in order to send Akkadian statements to its proxy object. The remote external smart object simulation program prints on its terminal all the received notifications. In this case, the implemented terminal application does not use any of Akkadian's or ObjectivSim's libraries. It only uses low-level TCP/IP sockets in order to connect, interact and ultimately play the role of an external smart object within ObjectivSim's environment.

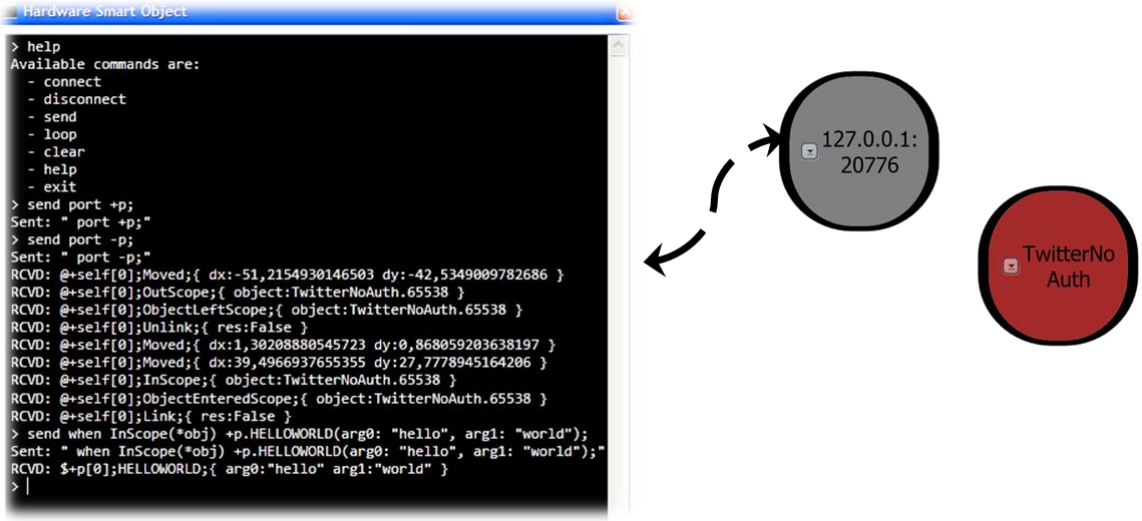


Figure 58 - Creating a hybrid external smart object through the external smart object simulator

4.10 Discussion

In this chapter, we introduced the abstraction of smart objects as a fundamental structural and functional element for implementing the functionality of ambient intelligence environments. We defined the core concepts of the smart object model utilizing the Z notation [25] as the model's specification language, through which we could formally define the notion of an AmI environment, its applications and its functionality. Aiming at improving the efficiency of the application of the smart object model for describing the functionality of AmI environments, we designed and implemented an inherently asynchronous, dynamic, event-based programming language called Akkadian that allows for the accurate and intuitive description of the structure of smart objects and their interactions. Through the

comprehensive architectural abstractions and modular implementation of Akkadian's runtime, we defined the notion of service-engines and we proposed and implemented a fully graphical smart object environment in which smart objects can operate and interact. Through the dynamic incorporation of diverse service-engines, smart objects utilizing Akkadian's runtime are able to exploit the functionality offered by diverse services built on top of diverse technological platforms. The proposed smart object environment, realized as an infinite two-dimensional plane, enables the visualization of smart objects and their interactions allowing for simulating smart object functionalities through direct interaction onto the environment's user interface. ObjectivSim, the tool under which the environment is provided, also offers comprehensive user interface components for facilitating the design and development of smart object programs. Furthermore, for integrating the functionality of external mobile devices in an intuitive and practical manner, we defined the notion of external smart objects and allowed their seamless interoperation with other objects within the smart object environment.

Compared to existing approaches, the smart object abstraction and its application through Akkadian, naturally models the distinct computational elements and the utilization of resources that synthesize applications in the AmI domain, allowing for robust, flexible and intuitive implementations. Distinct and diverse hardware devices and software components can be directly modeled as smart objects, leading to improved interoperability. Since they are able to dynamically form and break federations, discover federations, and interact with each other, smart objects-based applications are by design extensible and adaptable. Moreover, the capability of the model to attach and use any service through smart object ports greatly enhances its composability potential. Ultimately, the goal is to provide a comprehensive, intuitive programming model and tools for implementing the horizontal and vertical functionality of highly dynamic, adaptable and extensible AmI environments.

4.10.1 Future Work

Towards improving the proposed approach with the utilization of the smart object model through Akkadian and its tools ecosystem, we plan to turn our focus to four different directions: (a) security, (b) persistence, (c) semantic annotation, and (d) environment-wide programming.

Specifically, concerning security in the smart object model, we plan on extending the notion of the scope relation for providing security and privacy primitives, allowing thus the definition of the notion of trust among smart object federations. The definition of security and privacy conditions could be expressed in the context of an Akkadian program and enforced by the implementation of the smart object environment. Scope, in the context of the smart object model, is ideal for controlling and enforcing trust among objects, since it constitutes the only way for smart objects to discover other smart objects and interact with them. The second method of discovering and referring to other objects through the *path* relation (see 4.3), is guaranteed to maintain the trust relation due to the latter's transitive nature. E.g., if object A trusts object B and object B trusts object C, then object A should also trust object C.

Persistence, in the context of smart objects and the smart object environment, refers to the ability of maintaining their state and recovering from potential temporary failures. Currently, in our implementation of the smart object environment, if ObjectivSim exits unexpectedly due to either an internal programming error or due to an operating system or hardware malfunction, the environment's state is reset. Previous interactions among smart objects and established federations are completely "forgotten". However, the decentralized nature of the smart object model and the modeling of its state through the well-defined notions of memory and services, allow for a comprehensive and intuitive persistence model. In this regard, for implementing persistence, it is enough for individual smart objects to maintain a persistent record of the memory and the services that are associated with their ports.

In our implementation of the smart object model, individual smart objects receive and process the events emitted in the context of their service-providing ports. This essentially pushes the requirement of reasoning for composing higher-order events to the utilized services and the smart object programs implemented in Akkadian. Although this mechanism is flexible, robust and intuitive, allowing the composition of higher-order functionality from diverse service technologies within the context of Akkadian programs, we could leverage it substantially by introducing a formal semantic annotation method of the functionality offered by services attached to service-providing ports. This semantic annotation mechanism would be completely orthogonal with respect to the proposed approach for programming smart objects and would provide a Meta layer for being able to use higher-

order events directly in the context of Akkadian programs, without requiring their prior recognition and definition through Akkadian "when" statements. This meta-event generation capability would be one level of abstraction above port events utilizing knowledge maintained by the smart object environment. This capability would further simplify programming smart objects in Akkadian without altering or compromising the latter's structure, semantics or workflow. Finally, it could also improve the reusability potential of meta-events in the context of complex AmI environments.

With environment-wide programming, we refer to the ability of the environment to respond to the creation and presence of smart object federations with specific structural properties within its context. The definition of smart object federations as labeled directed multi-graphs with no self-loops, allows for a wide range of meta-operations that can exploit their structure in order to implement orthogonal functionalities within the context of an AmI environment. In this regard, we plan to extend the definition and implementation of the smart object environment in order to provide the means for the latter (a) to sense the presence of federations with specific graph structures, (b) to be able to establish connections between smart objects that operate within its context, and (c) to be able to break connections between smart objects that operate within its context. Since this additional functionality is an exclusive property of the environment, it can be incorporated orthogonally into the proposed approach without altering or compromising its definition and implementation. The inspiration for exploring this direction comes from [104] which essentially considers the process of forming smart object federations as catalytic reactions within the context of the environment. Therefore, as future work we plan to investigate whether the definition of the notion of environment-wide side-effects of federation formations, viewed as "catalytic reactions", can be used to leverage the process of developing AmI environments. Catalytic reactions, as a high-level design and interaction mechanism may be exploited by the smart object ecosystem in order to provide a common, efficient and intuitive solution to frequently occurring interaction patterns between smart objects. The potential of being able to associate functionality with catalytic reactions is, therefore, the provision of high-level "meta-recipes" for further simplifying the implementation of AmI environments. Apart from being used for implementing functionalities, this mechanism may also facilitate the formulation of environment-wide

assertions that will serve as global invariants [151] for systematically validating aspects of the AmI environment's integrity and responding to erroneous or unexpected conditions.

4.10.2 Connection Logic Taxonomy

Furthermore, apart from environment-wide programming, the abstraction and taxonomy of catalytic reactions for smart object federations proposed in [104], provides a comprehensive and systematic reference for describing the different patterns that appear in the connection logic of smart objects, which is applied in Akkadian programs for establishing federations. Therefore, in this subsection, we will define those connection logic patterns in terms of catalytic reactions and provide the relevant Akkadian program through which they are realized. In this regard, we can subsequently quantify the way an object A connects to another object B in terms of the following catalytic reactions: (a) autocatalysis, (b) catalysis with context, and (c) catalysis with context and stimulus.

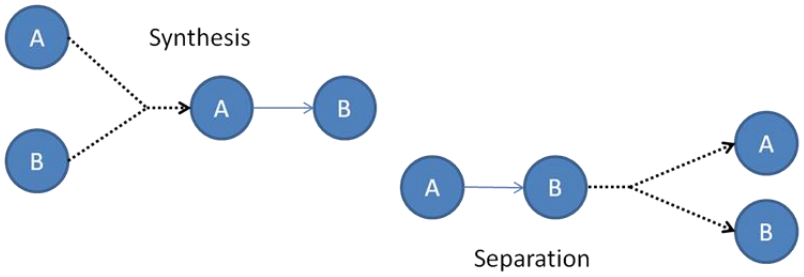


Figure 59 - Autocatalysis

Autocatalysis, defined in the context of a smart object A, refers to the connection logic pattern implemented in A's program, where the object itself initiates a connection to object B when the latter enters its scope. Conversely, object A is responsible for breaking the connection to object B. The outline of this connection logic can be seen in the Akkadian program outline of Figure 60 below.

```
port -L;

when InScope(*o) and o.name == "B" and OtherConnectionConditions
    +self.Link(-L, o);

when DisconnectionConditions
    +self.Unlink(-L);
```

Figure 60 - Autocatalysis Akkadian program outline

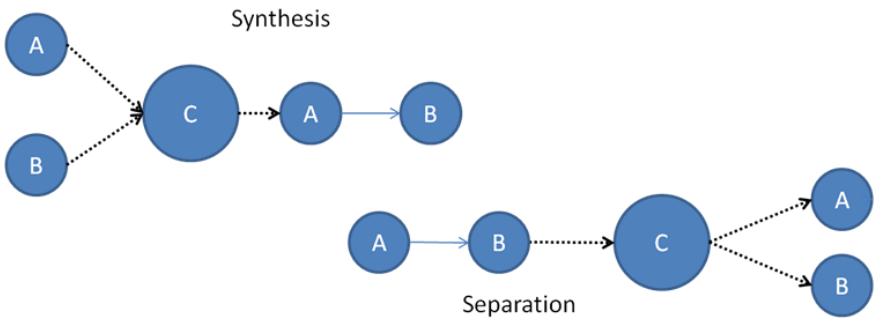


Figure 61 - Catalysis with context

Catalysis with context, defined in the context of a smart object C, refers to the connection logic pattern implemented in C's program, where the object establishes a connection from an object A to an object B when both objects enter C's scope. Conversely, if the federation of objects A and B enters C's scope, then object C is responsible for breaking their connection. The outline of this connection logic can be seen in the Akkadian program outline of Figure 62 below.

```
when InScope(*o) and (o.name == "A" or o.name == "B")
    +self[o.name] = o;

when OutScope(*o) and (o.name == "A" or o.name == "B")
    +self[o.name] = null;

// Connection outline
//
when +self.A != null and +self.B != null and OtherConnectionConditions
    +self.A.Link(-L, +self.B);

// Disconnection outline
//
when +self.A != null and -L(self.A).Linked(*o) and o.name=="B" and DisconnectionConditions
    +self.A.Unlink(-L);

when +self.B != null  and +L(self.B).Linked(*o) and o.name=="A" and DisconnectionConditions
    +self.B.Unlink(+L, o);
```

Figure 62 - Catalysis with context Akkadian program outline

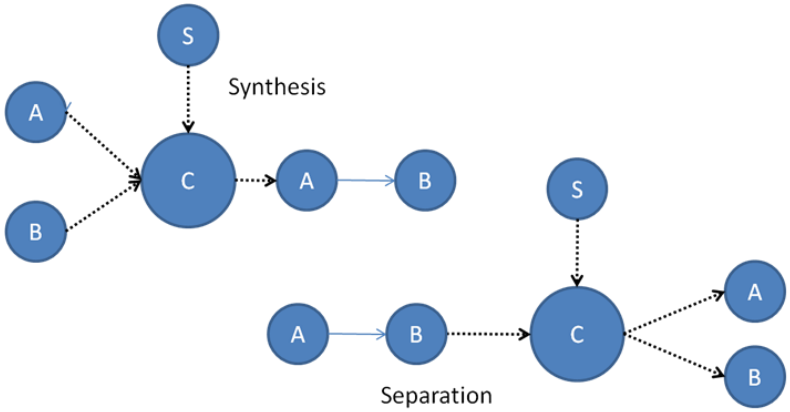


Figure 63 - Catalysis with context and stimulus

Finally, catalysis with context and stimulus, defined again in the context of smart object C, refers to the connection logic pattern implemented in C's program, where the object establishes a connection from an object A to an object B when both objects enter C's scope and there is at the same time an object S within object C's scope. Conversely, if the federation of objects A and B enters C's scope, then object C breaks their connection provided that an object S is at the same time within object C's scope. The outline of this connection logic can be trivially derived from the previous program outline in Figure 62, by replacing "OtherConnectionConditions" with "InScope(*o) and o.name == S and OtherConnectionConditions" and "DisconnectionConditions" with "InScope(*o) and o.name == S and DisconnectionConditions".

Chapter 5
Evaluation

In Chapter 3 and Chapter 4, we presented our approach for creating ambient intelligence (AmI) environments using services and smart objects as the fundamental building blocks. The proposed service middleware was designed and implemented for enabling the full utilization of heterogeneous computing system resources that are available in AmI environments. Via the provided libraries, implemented for many popular programming languages, creating a completely new software service or adapting a legacy software system into a service to plug into an AmI environment is a rapid, intuitive process that respects the capabilities, strengths and conventions of the target programming language. In this respect, it covers and unifies under the umbrella of a homogenized service ecosystem a very wide spectrum of requirements, capabilities and research domains without hindering their underlying performance, suitability, sustainability and reusability potential. Subsequently, we introduced the abstraction of smart objects as a fundamental structural and functional element for implementing the functionality of AmI environments. Aiming at improving the efficiency of the application of the smart object model we proposed an inherently asynchronous, dynamic, event-based programming language called Akkadian that allows for the accurate and intuitive definition of smart objects and their interactions. In this context we implemented a fully graphical smart object environment in which smart objects can operate and interact. Finally, we showed the extended capabilities of the platform to incorporate and compose diverse functionalities by enabling the utilization of many different service technologies and allowing the modeling of remote mobile devices as fully capable smart objects.

Following the definition of our approach, in this chapter we will evaluate its effectiveness towards constituting a comprehensive and complete framework for creating AmI environments. In this regard, we will first disseminate and discuss the features and advantages of the proposed approach against a set of well-defined criteria, permeable to the process of building AmI environments. Secondly, we will showcase the effectiveness of

the framework through targeted examples, and finally, we will compare the proposed approach with traditional methodologies, highlighting its comparative advantages.

Specifically in section 5.1 we will discuss and quantify the features of the proposed approach that satisfy the high-level requirements for supporting the development of AmI environments (identified in section 2.4). In sections 5.2 and 5.3, we will indicate the advantages of the proposed framework against a well-defined set of external and internal criteria, permeable to the process of building AmI environments. Subsequently, in sections 5.4, 5.5 and 5.6 we will showcase the effectiveness of the proposed approach through example scenarios. In section 5.7 we will outline and compare the implementation of a specific AmI environment under the perspective of both the proposed approach and a class of more traditional approaches utilizing high-level general-purpose languages. Finally, we will conclude in section 5.8 with a summary and discussion.

5.1 Requirements Evaluation

In section 2.4, we identified a set of requirements that were deemed necessary for supporting the creation of AmI environments. Under this specific perspective, our aim was to build a comprehensive, complete framework to (a) maximize the potential of the environment for incorporating heterogeneous technologies, (b) provide sensible programming abstractions, (c) enable the implementation of higher-level functionalities by composing services, (d) provide a flexible model for representing and using the context of the environment, (e) provide libraries and tools for developing and deploying services, (f) provide libraries and tools for developing and deploying applications, and (g) provide the means for simulating the functionality of the environment.

Concerning the heterogeneity potential, our approach succeeds in delivering and utilizing diverse functionalities over three different axes: (1) the proposed service middleware supports five languages (C/C++, .NET, Java, Flash/ActionScript, and Python) for the implementation of services; (2) smart objects utilizing Akkadian's runtime, in addition to the services implemented on top of the proposed middleware, enables the incorporation of services implemented as CLR DLLs [121], and resources implemented as REST services [122]; and (3) the definition and implementation of external smart objects enables the full

utilization of the functionality offered by mobile devices or, in general, any software system capable of establishing TCP/IP connections.

The proposed approach succeeds in providing sensible, intuitive and complete programming abstractions for developing AmI environments (1) by enabling the robust modeling of the environment's static resources as distributed, fault-tolerant and statically type-checked services; (2) by enabling the modeling of the dynamic, volatile functionality of the environment as active smart objects defined under a dynamic weakly-typed event-based programming language; (3) by providing a practical and intuitive modeling of mobile devices as external smart objects operating under the well-defined semantics of the smart object model; and (4) by conceptualizing and defining the proposed abstractions in a formal way.

The composability potential of our approach is focused on the proposed event-based smart object programming language, Akkadian. In this regard, Akkadian succeeds in facilitating composition (1) by enabling the utilization of many different service technologies within its context; (2) by delivering high-level event-based semantics with robust event-filtering support; (3) by supporting sensible implicit type conversion operations; and (4) by adding a second layer of type convertibility in the context of the service-engine that handles services implemented under the proposed middleware (see 4.8.2). Towards improving composability support, as discussed in section 4.10.1, the orthogonal incorporation of semantic annotation capabilities in Akkadian programs paired with a comprehensive knowledge repository, maintained in the context of the smart object environment, can simplify and automate many aspects of the process for composing higher-order functionality.

Support for context awareness is an inherent design aspect of Akkadian's syntax and semantics. In this regard, Akkadian succeeds in maximizing context awareness and handling potential (1) by systematically defining the notion of services as a command-event relation (see 4.4), enabling though its asynchronous design intuitive syntactic and semantic abstractions for the uniform handling of context modifications; (2) by supporting advanced event-filtering capabilities for differentiating between different context states; and (3) by allowing smart object memory operations to be captured and handled along with events in a uniform way. This approach allows for very fine-grained object-local control when capturing contextual information.

Moreover, the proposed approach provides guidance and support for implementing and deploying services (1) by providing a development tool for describing the functionality of services and validating its syntactic correctness; (2) by handling the management of service descriptions in a central repository with versioning support; (3) by providing tools for generating language-specific files during the implementation or utilization of services; (4) by providing tools and infrastructure for the automatic, on demand deployment of services; and (5) by supporting automatic updating of the services implemented on the proposed service middleware.

Apart from supporting the development of services, the proposed approach facilitates the development of applications as aspects of the functionality of the AmI environment (1) by proposing a targeted domain-specific programming language for programming smart objects; (2) by providing a development environment that simplifies and guides the design and implementation of smart object-based functionalities; (3) by enabling the deployment of the implemented functionality from within the development tool; and (4) by visualizing the presence, context and interactions among the deployed smart object programs.

Finally, the proposed programming language and the implementation of the smart object environment as a graphical two-dimensional plane provides comprehensive simulation capabilities (1) by allowing the conditional definition of ports and their attached services, which can be dynamically replaced in the context of an Akkadian program; (2) by offering graphical user interface elements in the context of individual objects, allowing the emission of synthetic events via direct user interaction over the deployed smart object instances; and (3) by providing an external smart object simulator, realized as a connection command terminal, that enables the transmission and acquisition of arbitrary Akkadian statements and events.

5.2 External Evaluation

Apart from satisfying the aforementioned requirements, it is important to discuss the performance of the proposed approach against a set of external evaluation criteria. In this context, the external evaluation criteria of the proposed approach concern the way the latter is used by its users – i.e., the developers designing and building AmI environments.

These criteria should reflect the practical demands and expectations of the developers when utilizing the proposed framework to implement fully functional environments. Under this perspective, we will discuss the features of the proposed approach under the following criteria: (a) rapid development potential, (b) maintainability potential, (c) reliability potential, (d) portability potential, (e) learnability potential, (f) reusability potential, (g) extensibility potential, (h) adaptability potential, and (i) efficiency. In this case, rapid development refers to how fast users are able to make progress while programming AmI environments. Maintenance refers to the effort needed in order to setup and maintain the development-deployment workflow while building AmI environments, aside from programming the actual functionality. Reliability refers to how well the framework shields its users from developing and deploying erroneous functionality that makes the environment unusable. Portability refers to the ability of utilizing the components implemented over the proposed approach on top of different computing platforms and the ability to incorporate into the proposed framework functionality developed over different architectures, libraries, platforms and technologies. Learnability refers to how easily can developers grasp the concepts and methodology suggested by the proposed approach. Reusability refers to how easily a specific component, developed over the proposed framework or any other technology, can be reused towards offering its functionality for creating AmI environments. Extensibility refers to the ability of the proposed approach to extend or reduce the functionality offered by AmI environments with minimum overhead for the programmer but without disrupting the already deployed functionality. Adaptability refers to the amount of support provided by the proposed framework for enabling the adjustment of the behavior of the environment based on contextual information. Finally, efficiency refers primarily to the capability of the approach to deliver AmI environments that are responsive and scalable.

The proposed approach enables rapid development in the following ways: (1) by providing libraries and tools for developing services in the supported programming languages following their conventions and programming style; (2) by providing tools for automatically deploying and updating services; (3) by proposing a domain-specific high-level dynamic programming language with comprehensive semantics; (4) by allowing the easy deployment of smart object programs within their environment; (5) by visualizing and enabling the simulation of interactions among smart object programs operating in the environment; and

(6) by allowing the incorporation and utilization of many diverse functionalities implemented under diverse technological platforms. These features essentially minimize extraneous programming statements (boilerplate code) by being focused on solving the issues appearing in their narrow application domain.

Easy maintenance in the proposed approach is supported (1) by keeping and managing the service descriptions in a central repository where the latest version is always accessible to programmers; (2) by providing tools that automatically generate supporting files for implementing and using services in the supported programming languages; (3) by providing a full development environment for programming managing and deploying smart objects; and (4) by proposing a comprehensive architecture that completely separates the development strategies for exposing the resources in the environment from those for implementing the latter's functionality.

Regarding the reliability potential of the proposed approach, there are two dimensions: (1) at the service development and utilization layer (also true in Akkadian), implementations and invocations of services are statically type-checked revealing potential errors before deployment or invocations; and (2) at the infrastructure level of the services, the proposed approach supports redundancy and automatic recovery from individual service failures. Towards improving the reliability support, as discussed in section 4.10.1, we plan to support persistence in the context of the smart object environment.

Portability in the proposed approach is achieved (1) by utilizing portable libraries as the foundation for implementing the proposed middleware, which makes it instantly usable under many different platforms (except for the Flash/ActionScript libraries that are implemented on top of Microsoft's COM technology and run only under the Windows operating system); (2) by proposing a separate domain-specific language for implementing the functionality of AmI environments, with well-defined semantics that enable alternative implementations of its runtime environment; and (3) by providing support for external smart objects that utilize a simple text-based, line-delimited lightweight TCP/IP protocol, which is ubiquitous among a widely diverse range of computing platforms regardless of how limited their resources are.

Concerning the process of implementing and using services, the learnability potential is maximized (1) by enabling the implementation and utilization of services using the target language's abstractions and conventions; (2) by providing support for five different popular programming languages, offering programmers the ability to choose the language they are most comfortable in; and (3) by providing tools that streamline and guide the development process. Concerning the use of Akkadian as a programming language for implementing the functionality of AmI environments, aside from the provided development tool, the learnability potential is supported by its well-defined high-level semantics and its intuitive utilization of the environment's context. Akkadian's learnability potential will be showcased in this chapter's example programs.

Reusability in the proposed approach is supported (1) by providing libraries for implementing services in five popular programming languages, allowing the reuse of previously developed systems within the context of the AmI environment; (2) by enabling the seamless interoperation among different service technologies within the context of smart object programs; and (3) by allowing the utilization of functionalities through a simple text-based protocol in the context of external smart objects. Towards improving the reusability potential of the proposed approach, we plan to incorporate meta-programming features in Akkadian for simplifying the development of commonly occurring object interaction patterns. In this context, a taxonomy of recurring connection patterns can be seen in section 4.10.2.

Extensibility in the proposed approach is supported (1) by separating the description of service interfaces from their implementation, allowing for the realization of different functionalities under conforming uniform service interfaces; (2) by enabling the incorporation of diverse heterogeneous service technologies under a unified well-defined service abstraction in the context of the smart object model; (3) by enabling the dynamic creation of smart object federations that provide functionality and context to its connected peers; and (4) by allowing high-level abstractions for monitoring and handling context modifications in the domain of Akkadian programs. In this regard, the concept of environment-wide programming, discussed in section 4.10.1, could be utilized towards improving the extensibility potential of the proposed approach.

The adaptability potential of the proposed framework is facilitated (1) by allowing diverse functionalities to be exposed in the AmI environment under uniform interfaces in the context of distributed services that are resolved at runtime; (2) by proposing a dynamic weakly-typed programming language that allows for flexible interaction and invocation patterns; (3) by supporting the dynamic association of different services with service-providing ports in the context of Akkadian programs; (4) by enabling conditional port definitions that can dynamically and seamlessly replace the functionalities offered by smart objects to their connected peers; and (5) by allowing smart objects to extend and override the functionality offered by other smart objects within their federations.

Finally, the efficiency of the proposed approach is maintained (1) by basing the implementation of services on top of the CORBA architecture [49] supported by high quality, open source performant implementations; (2) by enabling the scalability of the service infrastructure by means of redundant services; and (3) by defining the functionality of the AmI environment in terms of self-contained smart objects that enable the parallel execution of not only their associated services but also their own programming logic.

5.3 Internal Evaluation

Apart from the above external criteria from the viewpoint of the AmI environment's programmers, it is important to discuss the performance of the proposed approach against a set of introspective, pragmatic internal criteria that concern its form and feature-set regardless of how the latter is utilized within its domain. In this regard, the internal criteria consider and express the viewpoint through which the intrinsic value of the approach is divorced from its didactic or utilitarian function. Therefore, in this section we will discuss the characteristics of the proposed approach that facilitate (a) readability, (b) writability, (c) simplicity, (d) orthogonality, (e) consistency, (f) expressiveness, and (g) abstraction potential. In this context, the readability of the approach refers to how well someone can understand the architecture and function of an already implemented environment. Writability refers to the ability of the proposed approach to express with precision the required functionalities, without excessive verbiage or tedious setup tasks. Simplicity is closely tied with readability and writability but also concerns the provision of a minimal number of well-targeted

primitive concepts and features for implementing AmI environments. Orthogonality refers to how well the approach enables the combination of its concepts and features in a meaningful way, in order to avoid erroneous usage and undesired side-effects. Consistency refers to the ability of the proposed approach to avoid expressing related behaviors and concepts in different, inconsistent ways. Expressiveness is closely tied with readability and writability but also contains the notion of succinctness towards modeling and implementing AmI environments using the proposed approach. Finally, abstraction potential refers to how well the proposed approach supports a well-defined high-level set of modeling and control abstractions for handling its target application domain. In this case, the modeling abstraction potential was already discussed in section 5.1.

The readability of the proposed approach is facilitated (1) by implementing libraries and tools for developing services in different programming languages and providing a natural mapping to each language's abstractions and conventions; (2) by proposing a domain-specific programming language with high-level syntactic conventions and well-defined semantics; and (3) by providing tools that enable the monitoring of both the services and the smart objects that are operating in the AmI environment

The writability of the proposed approach is again facilitated by the provided service libraries, tools and syntactic conventions of the domain-specific programming language, but also (1) by enabling the handling of diverse services under a unified access methodology in the context of smart object programs; (2) by eliminating the need of tedious setup tasks while deploying services and smart objects; and (3) by implementing comprehensive development guidance through the proposed design and modeling methodology.

In addition to being supported by the same features that enhance the readability and writability potential of the proposed approach, simplicity is further enforced (1) by defining Akkadian as a small language with a carefully chosen, minimalistic set of features that does not hinder its applicability to the target problem domain; and (2) by allowing the incorporation of additional functionality through a minimalistic lightweight text-based line-delimited TCP/IP protocol.

The orthogonality potential of the proposed approach is enhanced (1) by proposing clear comprehensive architectures and methodologies for separating the notion of resources in

the context of the AmI environment from the its actual functionality; (2) by allowing the independent, modular and loosely coupled incorporation of new functionalities within the context of smart object programs through plug-ins; (3) by enabling smart objects to dynamically form and break federations, and thus, operate under different isolated contexts; and (4) by enforcing through the implemented tools a clear separation of the smart object environment from the distinct smart objects that operate in it, allowing for alternative implementations with different characteristics and properties. Towards improving the orthogonality potential, as discussed in section 4.10.1, environment-wide programming and semantic service descriptions can be utilized for implementing additional behaviors without disrupting the structure and semantics of the ones already implemented.

The consistency of the proposed approach is enforced (1) by allowing the implementation and utilization of services with the same functionality under five different programming languages in a uniform way that respects the semantics and abstractions of the target language; (2) by providing a consistent well-defined service interface in the context of smart object programs that can host functionalities from diverse services; (3) by enabling the handling of memory operations and invocations via the same mechanisms and syntactic abstractions utilized by events; and (4) by defining and supporting the notion of external smart objects that are operating and conform to the well-defined smart object model.

Apart from the features that support the readability and writability of the proposed approach, expressiveness is further supported (1) by providing comprehensive abstractions in Akkadian for intuitively capturing and filtering the received events within the context of smart object programs; (2) by proposing an event-based, inherently asynchronous programming language that naturally models the sensitivity operands of AmI environments; (3) by allowing the side-by-side uniform handling of memory operations as events using the same intuitive high-level programming statements.

Finally, we discussed the features of the proposed approach against the modeling abstraction potential in section 5.1. Concerning the control abstraction potential, the proposed framework enables the intuitive modeling of the control flow of the functionality of AmI environments (1) by proposing an event-based programming language that naturally models the asynchronous nature of contextual notifications within the AmI environment;

and (2) by unifying the handling of memory accesses, method invocations and event emissions within the context of smart object programs, utilizing the semantics of Akkadian's "when" statement.

5.4 Chain Replication

In this section we will show an example of using Akkadian to describe smart object federations and, specifically, an example implementation of their connection logic. Section 4.10.2, gave a description of the primitive connection strategies that appear often during the process of establishing smart object federations through matching ports. Towards composing higher-order connection strategies from the aforementioned patterns, in this example we will show a more complex scenario that implements a specialization of the smart object connection algorithm, presented in [140].

Given a sequence of an arbitrary number of smart objects connected through port *L* towards the same direction, i.e., a connected chain of n objects, the path $-L-L-L... -L = -L^{n}$, evaluated in the context of the first object, will reference the last (n^{th}) one, the Akkadian program presented in this section, enables them to make an exact replica of their chain – provided that the environment contains enough instances of the required objects, and that those objects can enter the scope of the objects that participate in the initial chain.

The original chain can be formed using catalysis with context as the connection strategy through an object named "Connector" that connects the two objects that enter its scope along the original direction of the previously established connections through port *L*. Figure 64 shows the creation process of the original chain using the "Connector" object inside ObjectivSim's smart object environment.

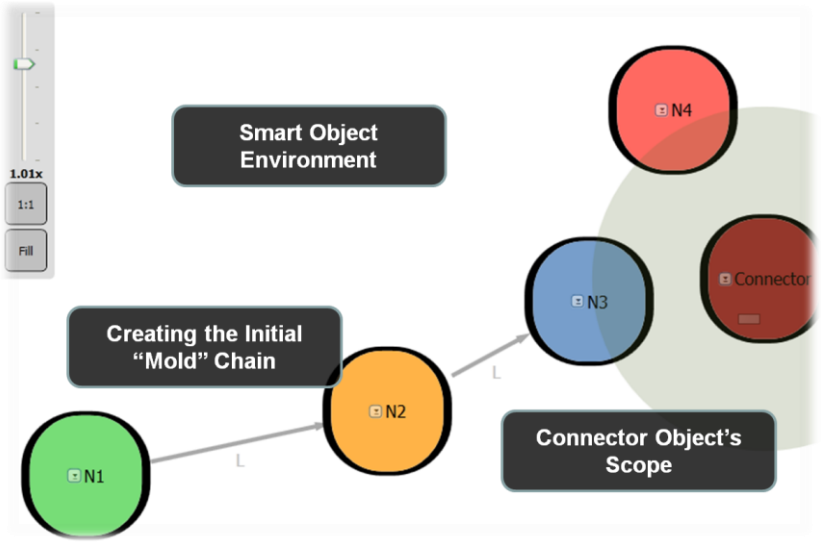


Figure 64 - Creating the original chain with the Connector object

Therefore in this case the connector's Akkadian program for establishing the connections is the following:

```
when InScope(*o) and +L(o).Linked()
    self[First] = o;

when InScope(*o) and not +L(o).Linked()
{
    if self[Second] != null
        self[First] = o;
    else
        self[Second] = o;
}

when OutScope(*o)
{
    if o == self[First]
        self[First] = null;
    else
        self[Second] = null;
}

when self[First] and self[Second]
    self[First].Link(-L, self[Second]);
```

All the participating objects, N1, N2, …, Nn, are using the same Akkadian program for expressing their connection logic. Therefore, in the first part of the Akkadian program, objects are assigned a role depending on their relative position in the chain:

```
port +L;
port -L;

when Activated()
    self.role = Free;
```

```
when +L.Linked()
{
    if self.role == Start
        self.role = Middle;
    else
        self.role = End;
}

when -L.Linked()
{
    if self.role == End
        self.role = Middle;
     else
        self.role = Start;
}
```

The *Activated* event is emitted the first time an object is placed in the environment and its program is evaluated – i.e., when the object is activated. At first, the object is not connected through any other object, and thus its "role" in the chain is marked as "Free". In Akkadian, when an identifier is not an alias for an event payload (see 4.7.3), it is equivalent to a string with the same name. Subsequently, when an object connects to the current object through port *+L*, then, provided that *–L* is unconnected, the current object is surely at the end of the chain, and so it is assigned the role "End". Conversely, when an object connects through port *–L* and *+L* is unconnected; then, the current object is at the beginning of the chain and is assigned the role "Start". If both *+L* and *–L* are connected, then the current object is assigned the role "Middle". This can also be seen in Figure 65, where object *N1* has the role "Start", *N4* the role "End" and all others "Middle".

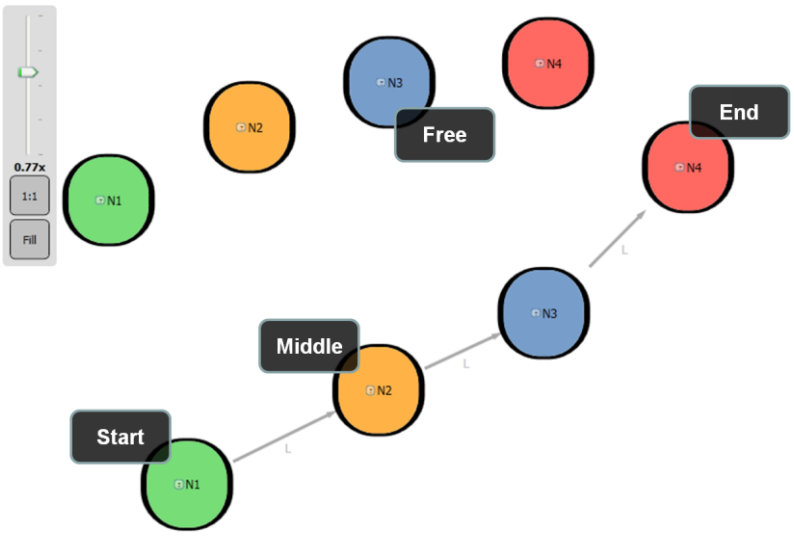


Figure 65 - A connected object chain and the role of the objects

After assigning roles, the key point in the algorithm is for objects participating in the chain to connect to other "Free" objects with the same name through a port *–C*. This happens as soon as those free objects enter the scope of the object in the chain. Following that, when the next object connected through *-L* connects with another object through port *–C* and the current object has an object connected through its own *–C* port, then those child objects are connected to each other through their *L* ports, thus gradually forming a replica of that part of the chain.

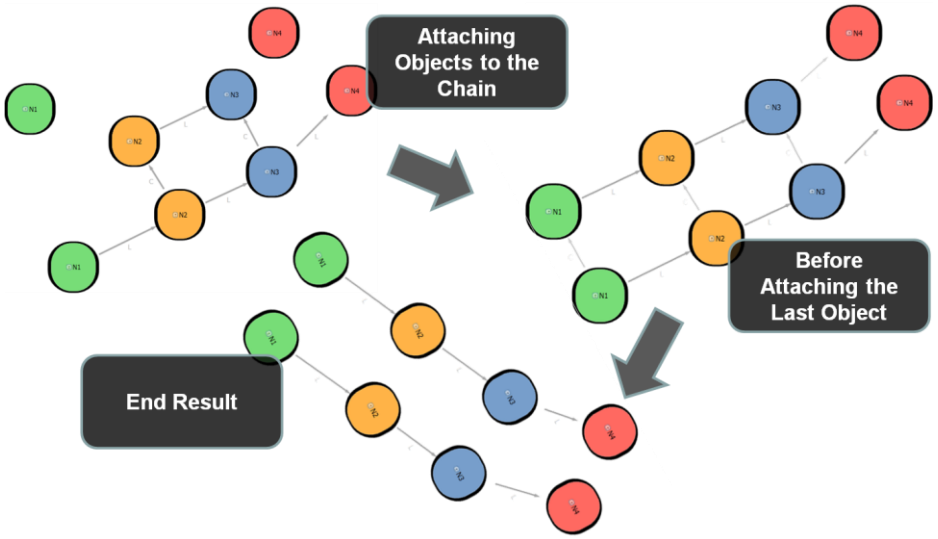


Figure 66 - Snapshots during the connection proccess

The Akkadian program that describes the aforementioned process can be seen below, where the last two "when" statements describe the terminal conditions and proceed in breaking all the connections through the *C* ports in order to finalize the replication of the chain (see Figure 66). The key part of the Akkadian program in this case is the transmission of a "token" from the start-object of the chain to the end-object, which is gradually propagated to the next object as soon as the current object connects with another one through port *–C*. When the token reaches the end-object, then it means that the replication is finished and so the end-object breaks the connection through port *–C* and, recursively, every previous object that observes that disconnection, also disconnects its own *–C*.

```
port +C;
port -C;

when InScope(*o) and o[name]==self.name and self.role !=Free and o.role == Free
{
    self.Link(-C, o);
    if self.role == Start
```

```
        -L.token = true;
}

when -L-C(self).Linked(*nextObj) and -C.Linked()
    -C+self(self).Link(-L, nextObj);

when +L.token and -C.Linked()
{
    if self[role] == End
        self.Unlink(-C);
    else
        -L.token = true;
}

when -L-C(self).Unlinked()
    self.Unlink(-C);
```

5.5 The Movie Scenario

In this example we consider a simplified version of an AmI environment, realized as the following scenario. When the user approaches a movie screen and performs a hand wave gesture with both hands, the lights turn off and a movie starts playing. While a movie is playing, the waving movement of the user's right or left hand instructs the movie screen to start playing the next or previous movie in the playlist, respectively. When the user moves away from the screen, the movie stops playing and the lights turn back on. Subsequently, when the user approaches an internet stand, the display shows information about the movies previously watched. Had the user not used the movie screen prior to approaching the internet stand, no information is displayed.

Breaking down this scenario, first of all, we assume that the following services are implemented and can be utilized by smart objects: (1) a Human tracking service that can track the position of the user inside a room, (2) a Video playing service that controls a screen, maintains a playlist and is able to play movie files on the screen, (3) a Human-gesture recognition service that is able to tell apart different hand gestures performed by a human, (4) an Internet service that is able to present information on a screen about a specific topic, and (5) a Lights service that is able to control the room's lights.

Having the aforementioned services available, we can fully implement the scenario with software smart objects. Specifically, we can model the scenario using three smart objects: (1) a "Movie Human" smart object whose role is to move around the room, control the video screen and the internet stand, (2) a static (unmovable) "Movie Stand" object that

offers the video service and can control the lights, and (3) an “Internet Stand” object that just offers the Internet service mentioned before. Implementing the scenario with these smart objects, apart from being able to utilize the services through them, requires the “Movie Human” to be able to “move” in the environment, following the user’s physical movement into the room. This can be achieved by using the events of the Human tracking service:

```
port +track("ami:///Sensors/HumanTracking/room1");

when +track.PosChanged(*dx, *dy)
    self.Move(dx, dy);
```

In this example, we will show the implementation of a simulation of this scenario. In this case, the movement of the object representing the user can be simulated by dragging (with the mouse) the visual representation of the object in ObjectivSim. Additionally, the Human gesture service can be simulated by providing buttons on the visualized objects, and lastly, we can show the actions performed on the video and internet services by altering the color and label of the participating objects. The buttons that trigger the simulated behavior of the gesture service are declared in Akkadian as ports with the “button” scheme:

```
port +GestureWave("button:///Wave");
port +GestureLeft("button:///Left");
port +GestureRight("button:///Right");
```

The object representing the user can, therefore, determine when it should access or disconnect from the video service and the internet service through the *InScope* and *OutScope* events:

```
port -Video;
port -Internet;

when InScope(*obj)
{
    if obj.name == "MovieStand"
        self.Link(-Video, obj);
    else if obj.name == "InternetStand" and self[movies] != []
        self.Link(-Internet, obj);
}
when OutScope(*obj)
{
    if obj.name == MovieStand
        self.Unlink(-Video);
    else if obj.name == InternetStand
```

```
        self.Unlink(-Internet);
}
```

Finishing the description of the object representing the user, the following statements describe how the object interacts with the movie stand and the internet stand:

```
when Activated()
    self.movies = [];

// Movie stand
//
when -Video.Linked() and +GestureWave.Pressed()
    -Video.PlayMovie();

when -Video.Linked() and +GestureLeft.Pressed()
    -Video.PreviousMovie();

when -Video.Linked() and +GestureRight.Pressed()
    -Video.NextMovie();

when -Video.FinishedMovie(*m)
    self.movies = self.movies + m;

// Internet Stand
//
when -Internet.Linked()
    -Internet.DisplayInfo(self.movies);
```

Subsequently, we simulate the behavior of the “Movie Stand” smart object as follows:

```
port +Video("echo:///Algorithm/VideoPlayer/room1");
port +lights(uri: "echo:///Thing/Lights/room1", arity: 0);

when Activated()
{
    self.Display(self.name);
    self.Color("#F0EAD6");
}

when +Video.Linked()
{
    lights.TurnOffAllLights();

    self.Color("#49796B");
}

when +Video.Unlinked()
{
    lights.TurnOnAllLights();

    self.Display(self.name);
    self.Color("#F0EAD6");
}

when +Video.PlayMovie()
{
    self.Display("Playing");
```

```
    self.Color("#417DC1");
}

when +Video.PreviousMovie()
    self.Display("Previous");

when +Video.NextMovie()
{
    +Video.@FinishedMovie("The Life of Brian");

    self.Display("Next");
}
```

Lastly, for simulating the “Internet Stand” smart object, in this case, we change the color of the object and display the title movie of the last movie watched:

```
port +Internet("echo:///Algorithm/InternetScreen/room1");

when Activated()
{
    self.Display(self.name);
    self.Color("#D9603B");
}

when +Internet.DisplayInfo(*items)
{
    for i in items
    {
        self.Log(i);
        self.lastItem = i;
    }
    self.Display(self.lastItem);
    self.Color("#ADFF2F");
}
when +Internet.Unlinked()
{
    self.Display(self.name);
    self.Color("#D9603B");
}
```

Figure 67 shows some snapshots from simulating the scenario in ObjectivSim’s implementation of the smart object environment, where initially the “Movie Human” smart object does not connect to the “Internet Stand” object since it does not contain any movies in its “self.movies” memory position.

Figure 67 - Snapshots from the movie scenario

Furthermore, we can extend the capabilities of the scenario by adding a new "Movie Help" object, as seen in Figure 67. In this case, when the "Movie Stand" object is within the scope of the "Movie Help" object, the latter can listen for the former's events and extend the capabilities of any interaction in which the former participates. In this particular example, the "Movie Help" object does not have any ports and when it detects that somebody connected to the "Video" port of the "Movie Stand" object, it changes its color and displays a helpful text message. When the "Movie Stand" object goes out of the scope of the "Movie Help" object, the latter cannot monitor the former's events. Apparently, the Akkadian program of all the other participating objects is not affected at all. The "Movie Help" object's Akkadian program is given below:

```
when Activated()
    self.Color("#E4717A");

when InScope(*obj) and obj.name == "MovieStand"
{
    when +Video(obj).Linked()
    {
        self.Display("Use Wave/Left/Right");
        self.Color("#CFB53B");
```

```
    }
    when +Video(obj).Unlinked()
        self.Display("G'Bye");
}

when OutScope(*obj)
{
    self.Display(self.name);
    self.Color("#E4717A");
}
```

Moreover we can further adapt the functionality of the “Internet Stand” object at runtime and decide dynamically whether it will simulate internet search functionality or actually implement it. In this regard, the following Akkadian program for the “Internet Stand”, depending on the value of the memory location “self.simulationMode” behaves as either a simulated object as before or it actually connects with an Internet search engine and requests results:

```
when self.simulationMode
    port +Internet("echo:///Algorithm/InternetScreen/room1");

when not self.simulationMode
{
    port +Internet
    {
        uri "https://www.googleapis.com/customsearch/v1";
        params [ key: "**zaSyBq****OGxJp31****gMXI_s****TmiZgw",
                    cx: "01648****0770025063***x9****mm" ];
    }
}

when Activated()
{
    self.Display(self.name);
    self.Color("#D9603B");
}

when +Internet.@DisplayInfo(*items)
{
    for i in items
    {
        self.Log(i);
        self.lastItem = i;
    }

    +Internet.Search(q:self.lastItem);

    when +Internet.Search(*items)
    {
        self.Log(items);

        self.Display(self.lastItem);
        self.Color("#ADFF2F");
    }
}
when +Internet.Unlinked()
```

```
{
    self.Display(self.name);
    self.Color("#D9603B");
}
```

Apparently, subsequent modifications of the value of the memory location "self.simulationMode" at runtime have as a result the dynamic replacement of the "+Internet" port for either simulating the service's functionality or actually fully implementing it as an internet search REST service.

5.6 External Chat Objects

This application of smart objects, implements its functionality by utilizing external smart objects that operate inside the smart object environment. Specifically, we consider an ambient chat application where multiple users carrying a mobile device that has texting capabilities move around the AmI environment, spontaneously forming and breaking ad-hoc chat groups with other proximate users. In this regard, the users, depending on their position inside the environment, form independent clusters. Each cluster represents a chat group where text messages written by a member of the group are delivered to all its other members but are invisible to users that belong to other clusters. A conceptual representation of ten users inside the AmI environment that have formed three chat groups by virtue of their current position can be seen in Figure 68.

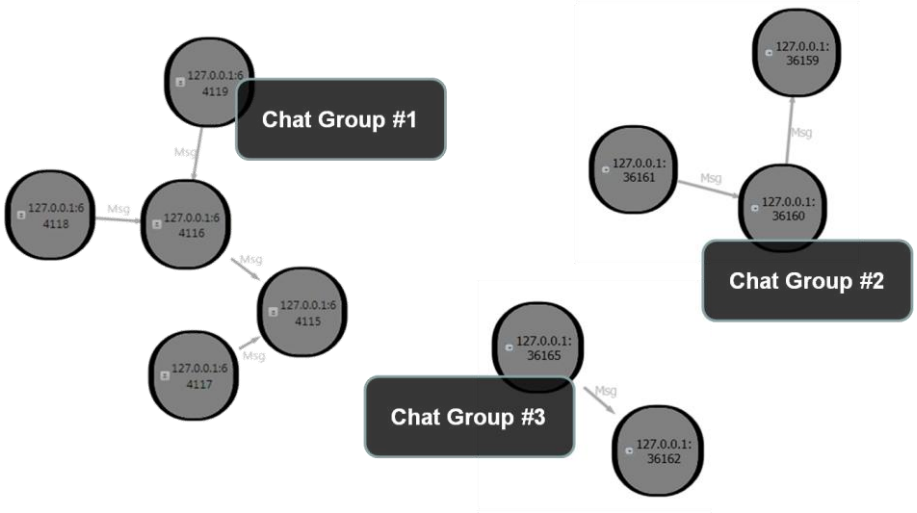


Figure 68 - Three chat groups in the smart object environment

Users distancing themselves from all the members of their current chat group are effectively quitting the group and stop receiving messages. Subsequently, when they approach a member of another group, they automatically join that group. This functionality can be modeled using external smart objects that are realized through the implementation of the text-based asynchronous protocol discussed in section 4.9. In this regard, we have realized external smart objects as independent application instances that only use a low-level TCP/IP socket for sending and receiving asynchronous messages in order to communicate with their corresponding proxy smart objects. The application that simulates the mobile chat device does not have any dependencies neither on Akkadian and its runtime, nor ObjectivSim. Figure 69 shows the different user interface (UI) elements that comprise the external chat smart object, which also includes the capability to track its movement over the desktop in order to simulate the movement of the users carrying the chat devices.

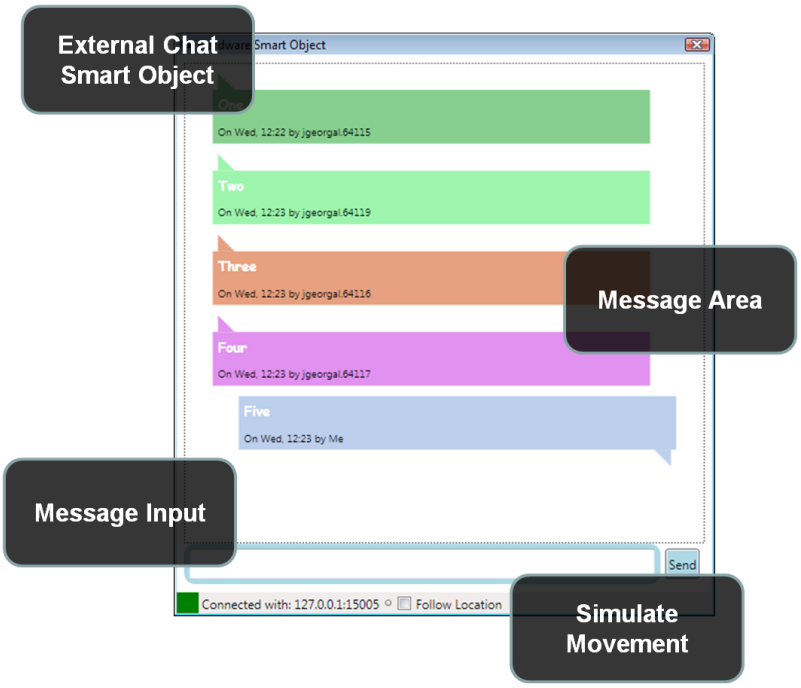


Figure 69 - An external smart object for chat

As soon as the chat device connects with its proxy smart object, it sends the following Akkadian statements as text over the established remote link:

```
port +Msg(arity: 10);
port -Msg;

when +self.OutScope(*obj)
    +self.Unlink(-Msg, obj);
```

```
when +self.InScope(*obj)
    +self.Link(-Msg, obj);
```

These Akkadian statements on one hand define an extrovert service-providing port (see 4.9) through which the external smart object implements a command for displaying messages on its screen, and can accept connections from other objects in order to form chat groups with them. On the other hand, the extrovert service-requesting port is used for establishing connections with the matching service-providing port of the other external smart objects. Additionally, the transmitted Akkadian statements, attempt a connection of the current external smart object through its extrovert service-requesting port, as soon as another object enters its scope. Thus, the connection strategy follows the autocatalysis pattern, described in section 4.10.2. Furthermore, since the object transmits event-capture statements it is considered a hybrid external smart object (see 4.9).

In a smart object federation that is comprised of external chat devices, when a message is sent by the user, the propagation algorithm works as follows:

- The current object invokes the “RouteMessage” command on its service-requesting port
- When the connected peer receives the invocation on its service-providing port, in turn invokes the “RouteMessage” command on its own service-requesting port
- When the invocation is performed onto an unconnected service-requesting port (i.e., the object is the root of the federation tree), the event “RouteMessage” is emitted in the context of that service-requesting port
- When an object receives the “RouteMessage” event in the context of its service-requesting port, it displays the message on its screen and emits the “RouteMessage” event on its service-providing port

Since the last step has as a result the emission of the “RouteMessage” event in the context of the service-requesting ports that are connected with the matching service-providing port, this step is performed recursively, until all members of the smart object federation have received the message. This algorithm is expressed in the Akkadian program below:

```
-Msg.RouteMessage(msg: "hello", from: "jgeorgal.5621"); // Example invocation
```

```
when +Msg.RouteMessage(*msg)
    -Msg.RouteMessage(msg);

when -Msg.RouteMessage(*msg)
{
    +self.DisplayMessage(msg); // Implemented in the external object
    +Msg.@RouteMessage(msg);
}
```

Therefore, if the above statements are transmitted upon the connection of the chat device with its corresponding smart object proxy, then the latter only has to (a) invoke the "RouteMessage" command on its service-requesting port when a message is sent by the user and (b) handle the "DisplayMessage" command which is invoked in the context of its extrovert self port. In the context of a chat device, the sequence of the received asynchronous messages through the lightweight text-based, line-delimited protocol can be seen below, where the message that needs to be handled is indicated in green color.

```
SENT: "-Msg.RouteMessage(msg: "hello", from: "jgeorgal.5621");"
RCVD: $+Msg[1];RouteMessage;{ msg:"hello" from:"jgeorgal.5621" }
RCVD: @-Msg[1];RouteMessage;{ msg:"hello" from:"jgeorgal.5621" }
RCVD: $+self[0];DisplayMessage;{ msg:"hello" from:"jgeorgal.5621" }
RCVD: @+Msg[0];RouteMessage;{ msg:"hello" from:"jgeorgal.5621" }
```

Figure 70 shows five external chat devices that have formed a chat group, represented in the environment by their proxy smart objects, exchanging text messages with each other.

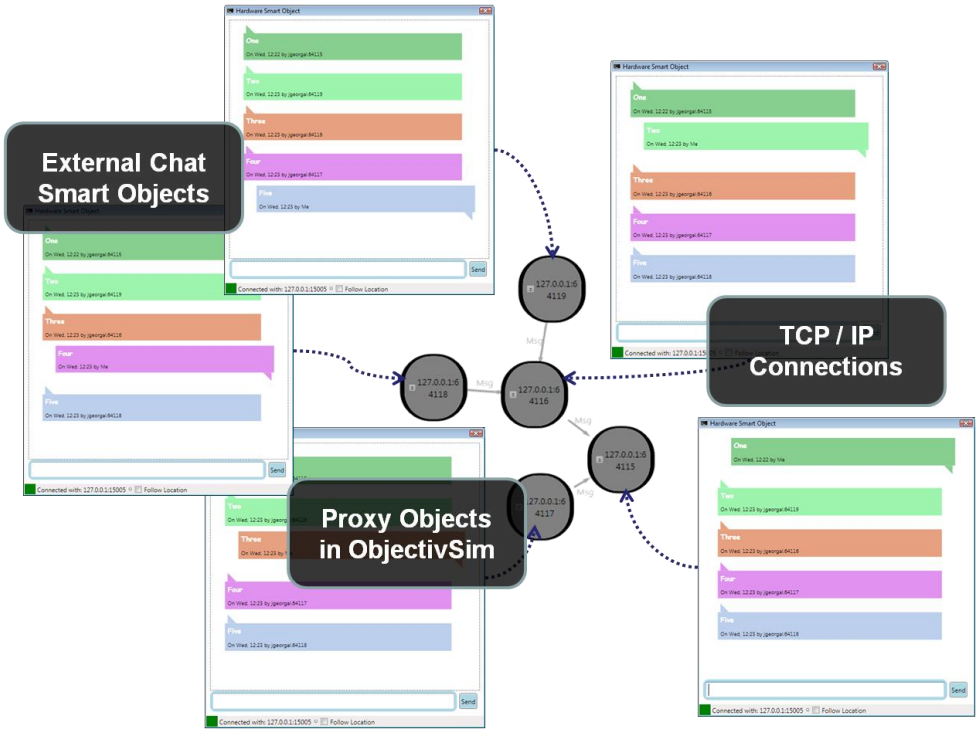


Figure 70 - External smart objects deployed into the smart object environment

Alternatively, the external chat devices may choose to implement the aforementioned message propagation algorithm completely within their own context. In this case, they have to handle the invocations and events on their extrovert ports and transmit the appropriate Akkadian statements. In this regard, in the context of a chat device, the sequence of the received and sent asynchronous messages through the text-based protocol can be seen below, where the messages that need to be handled are indicated in green color. In this case, the use of the "DisplayMessage" command invocation would be redundant since it is implied by the emission of the "RouteMessage" event in the context of the object's extrovert service-requesting port.

```
SENT: "-Msg.RouteMessage(msg: "hello", from: "jgeorgal.5621");"
RCVD: $+Msg[1];RouteMessage;{ msg:"hello" from:"jgeorgal.5621" }
SENT: "-Msg.RouteMessage(msg: "hello", from: "jgeorgal.5621");"
RCVD: @-Msg[1];RouteMessage;{ msg:"hello" from:"jgeorgal.5621" }
SENT: "+Msg.@RouteMessage(msg: "hello", from: "jgeorgal.5621");"
RCVD: @+Msg[0];RouteMessage;{ msg:"hello" from:"jgeorgal.5621" }
```

Since these remote chat devices are implemented as external smart objects operating within the smart object environment, they can communicate and interact with all other objects within the latter regardless of whether they are realized as software smart objects,

pure external smart objects, or hybrid external smart objects. Therefore, we can showcase this seamless interaction by defining a software smart object that can join any chat group and send a message every second to all members of the group as such:

```
port +timer("lib://timer/seconds/1");
port -Msg;

when InScope(*obj)
    +self.Link(-Msg, obj);

when OutScope(*obj)
    +self.Unlink(-Msg, obj);

when +timer.Tick()
    -Msg.RouteMessage(msg: "Hello", from: "Timer Robot");
```

5.7 A Smart Office Comparison

In this example we will outline the implementation of the AmI environment described in [44] using smart objects programmed in Akkadian and an external smart object, modeling the user's mobile phone. Our purpose for discussing this implementation is to draw direct comparisons between our approach and generalized classes of service-oriented AmI systems that implement their applications in high-level general-purpose programming languages without having clearly defined abstractions for modeling applications.

The original Smart Office environment, described in detail in [44], constitutes an ideal test case for the following reasons: (a) it implements a comprehensive fully working smart office environment with non-trivial, interesting interactions, (b) it was implemented in a high-level, well-designed and efficient general-purpose programming language (C# under .NET), (c) its architecture clearly separates the implementation of the functionality of the system from its utilized resources, and (d) although it utilizes functionality implemented as TCP/IP synchronous request/response servers, it fully wraps their low-level message exchange mechanics into .NET classes that enables their utilization from client code as if they were local language objects.

In this regard, the smart office environment is realized through direct user interaction with the desk that contains a foldable pad and the user's mobile phone. The desk-pad recognizes the objects placed on top of its surface (through RFID tags) and responds to their presence

by performing actions that depend on the current, ongoing task. Every task in the system begins with the identification of a specific object that is placed on the desk-pad's surface. Additionally, the system projects context-sensitive information on the desk-pad through a ceiling-mounted projector. In this case, the desk-pad can be unfolded, providing an extended projection area for displaying additional information.

The interaction steps, in the context of the smart office environment, that are initiated by the user's mobile phone are the following:

- The user can take a photograph using a mobile phone
- When the user places the phone on the desk-pad, the system recognizes and connects to it
- The user can perform a gesture to move the most recent photograph on the desk-pad's surface
- The user can perform a gesture to move the photograph to the screen across the room or to the photo-frame near the desk

The smart office environment also supports the delivery of presentations, utilizing the desk-pad and the screen that is located across the office's conference table (see Figure 71). The interaction steps of the presentation delivery task are the following:

- The user places a leaflet on the desk-pad, which is recognized by the system and is perceived as an indication to start the associated presentation
- The system dims the lights and starts the presentation on the screen across the room
- If the user removes the leaflet from the desk, the presentation is also displayed on the desk-pad's surface
- The user can change slides by using hand gestures with left hand, for moving to the previous slide, and right hand, for moving to the next
- Using an infrared pen that is tracked by a ceiling-mounted camera, the user can create hand-drawn annotations on the current slide

- If the desk-pad is unfolded, the top half shows the presentation and the bottom half shows the slide's comments; if it is folded again, only the current slide with its annotations are visible
- When the presentation ends the lights are turned back on

Another task that is initiated and controlled by the desk-pad supports the playback of video files on the environment's screen. In this case, the interaction steps are the following:

- The user places on the desk-pad a video token that, on one hand, uniquely identifies the associated video file and, on the other hand, dims the lights and initiates its playback on the screen
- Using hand gestures with the left or right hand, the user can pause or resume the video respectively
- Using a distance-sensor, which is mounted at the corner of the conference table, the user can control the volume of the video's sound
- Removing the video token from the desk-pad, instructs the system to turn the lights back on and stop the reproduction of the video file

Finally, the desk-pad recognizes some additional RFID augmented artifacts in order to:

- Start a videoconferencing session on the big screen
- Start an email application on the desk
- Start a game on the big screen

These last three applications are not tightly integrated into the system and support interaction only through keyboard and mouse events that are synthesized in the context of the current application through external wired buttons and translation of the infrared pen's movements as low-level operating system mouse events (see also [44]).

The general setup and aspects of the interaction of the smart office environment can be seen in Figure 71 below.

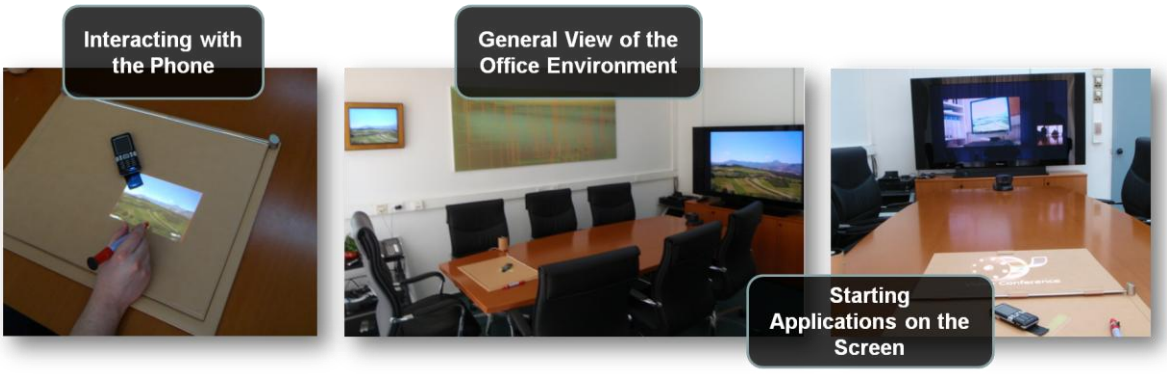


Figure 71 - The Smart Office AmI environment

The smart office environment's architecture can be seen in Figure 72, where the different aspects of the interaction are controlled by the different task states, which receive the messages from the sensing components of the environment and control it through its action components. A subsequent task state is chosen by either the current task state or by the office application component that on one hand initiates and supervises the active tasks and on the other hand communicates with the user interface (UI) components in order to display the application's graphical elements on the desk-pad. Essentially the action and sensing components (the light blue components in Figure 72), are instantiated as TCP/IP servers that play the role of services within the infrastructure. As mentioned before, these components are wrapped in .NET objects and therefore their invocation is realized through typical object method calls and their notifications are received through .NET events. In this regard, the application logic is effectively separated by the actual sensors and actuators that are available in the office environment. For this reason, we can draw direct comparisons between the implemented application logic (the light gray components in Figure 72) and an alternative implementation through the proposed approach, utilizing smart objects.

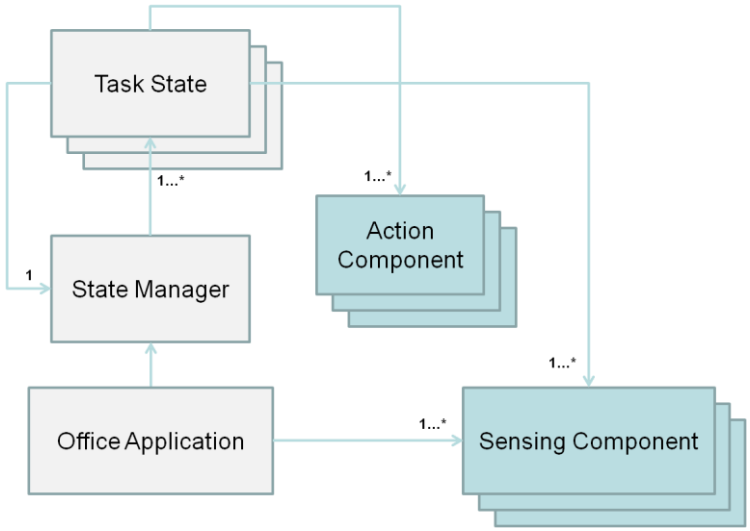


Figure 72 - Smart office environment's architecture

For this comparison, first of all we will assume that we have all the office environment's services available through the proposed middleware (Chapter 3). Apparently, the actual programming of a service and its client counterpart in the original implementation of the smart office environment suffers from many issues, including: (a) extraneous programming statements for instantiating and managing TCP/IP sockets, (b) overhead from the implementation of the communication protocol, (c) overhead from the implementation of the serialization/de-serialization protocol for the content of the exchanged messages, and (d) overhead from the need to implement an object-oriented interface to wrap the functionality of the service client code. Specifically, we can consider the reference implementation of the "Presenter" service that is responsible for delivering presentations in the office environment. In this case, the reference implementation follows the interface given below, minus the "CurrentSlideChanged" event.

```
#include <ami.idl>

module Test {
     interface Presenter {

          // Methods
          //
          boolean OpenPresentation(in string presentation);

          long GoToSlide(in long slideNo);

          boolean NextSlide();
          boolean PreviousSlide();

          string GetCurrentSlideComments();

          void HidePresentation();
```

```
            void ShowPresentation();

            long GetCurrentSlideNo();
            long GetTotalSlides();

            // Events
            //
            void Event_CurrentSlideChanged(in long slideNo);
        };
};
```

The reference service implementation is programmed in the C# programming language, utilizing a custom network library that provides high-level access to .NET's network sockets[9] and handles the encoding and decoding of the messages exchanged over the established TCP/IP streams. This network library comprises of 631 pure lines of code (LOC). These are the LOC that are comprised exclusively of C# statements, completely ignoring the empty lines, used for indentation purposes, and the comments, used for documentation purposes. Despite the high-level abstractions that the network library provides, which simplify network programming substantially, the reference implementation of the above service comprises of 267 pure LOC. For the LOC calculation we have also removed all the lines that perform calculations and invoke the UI components that actually carry out the requested operations. In this sense, the reference implementation of the service achieves the provision of a natural and intuitive access model for invoking the former's operations as if it were a local memory object. For providing exactly the same access model for the "Presenter" service, its implementation over the proposed middleware in C# comprises of 71 pure LOC, including the LOC required for its interface description in Idlematic (see 3.6). Additionally, the implementation over the proposed middleware provides at-most-once invocation semantics (see 3.2.1) and includes the provision of the "CurrentSlideChanged" event which is absent from the reference implementation.

Provided that the client wrappers are available in the reference implementation, it is safe and fair to assume their existence in the proposed middleware since the invocation interface of the already implemented services as .NET objects would be identical to the invocation interface of services implemented on top of the proposed middleware (one-to-one mapping).

[9] http://msdn.microsoft.com/en-us/library/system.net.sockets.aspx

Therefore focusing only on the application's logic and ignoring the implementation of the services and the UI components, the whole smart office application environment comprises of 1,252 pure lines of code (LOC). For the LOC calculation we have also removed all the lines that perform UI-specific calculations, e.g., for normalizing the coordinates received from the camera-based tracking service. Finally, these lines do not contain any code for handling the infrared pen's tracking events since they are received by a completely separate program and are fed back to the operating system as typical mouse events.

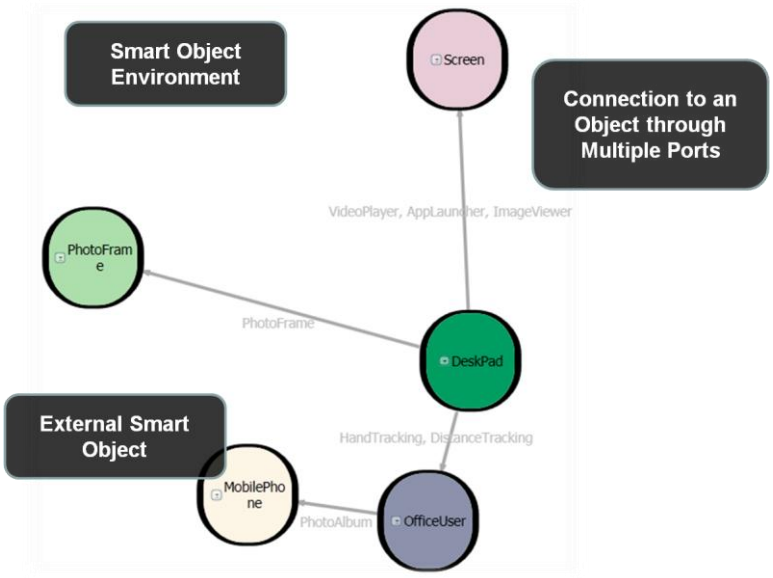


Figure 73 - Interaction setup of the smart office environment using smart objects

An implementation of the aforementioned scenario using smart objects could be realized as a federation of five smart objects:

- An "Office User" smart object that is responsible for accessing the mobile phone's photo album, obtaining the most recent photo and exporting the hand-tracking and distance-sensing services
- A "Mobile Phone" external smart object that makes available the mobile phone's functionality in the context of the smart object environment; in this case we only care about the photo album functionality of the phone
- A "Screen" smart object that contains service providing ports for a video-playing service, an application launcher service, an image viewer service, and a presentation viewer service

- A “Photo Frame” smart object that exports the service that is responsible for exploiting the physical frame’s functionality
- A “Desk Pad” smart object that uses the functionality of all the other objects in order to deliver the functionality of the smart office environment

The setup of the smart objects implementing the smart office scenario can be seen in Figure 73. In this regard, most of the environment’s logic is implemented in the context of the “Desk Pad” object, since in this specific scenario it is the desk-pad’s interface that arbitrates the interaction and this should also be reflected onto its smart object-based implementation.

For this implementation the Akkadian program of the “Office User” smart object can be seen below:

```
port +HandTracking("echo:///Sensor/HandTracking/office");
port +DistanceTracking("echo:///Sensor/DistanceSensor/office");

// Mobile Phone services
//
port -PhotoAlbum;

when InScope(*o) and o.name == MobilePhone
    self.Link(-PhotoAlbum, o);

when OutScope(*o) and o.name == MobilePhone
    self.Unlink(-PhotoAlbum, o);

when +self.Moved(*dx, *dy)
    -PhotoAlbum+self(self).Move(dx, dy);

when -PhotoAlbum.AddedPhoto(*photo)
    self.MostRecentPhoto = photo;
```

Part of the Akkadian program of the “Desk Pad” object that handles the task of displaying the most recent photo from the mobile phone can be seen below:

```
when +rfid.Recognized("Mobile Phone") and +desktopUI.AnimationFinished(*x, *y)
    +desktopUI.ShowTickMark(x, y);

when +self.User.MostRecentPhoto and +rfid.Added("Mobile
Phone") and +desktopUI.Gesture("DragTickMark")
{
    -PenTracking.DisableTickMark();
    +desktopUI.ShowPhoto(self.User.MostRecentPhoto);
}

when +self.User.MostRecentPhoto and
          +desktopUI.Gesture(name == "FlickPhoto", direction == "North")
    -ImageViewer.ShowImage(+self.User.MostRecentPhoto);
```

```
when +self.User.MostRecentPhoto and
          +desktopUI.Gesture(name == "FlickPhoto", direction == "East")
    -PhotoFrame.ShowInstantImage(+self.User.MostRecentPhoto);

when +rfid.Removed("Mobile Phone")
{
    -ImageViewer.HideImage();
    -PhotoFrame.HideInstantImage();
}
```

The full implementation of the smart office environment using the aforementioned smart objects programmed in Akkadian comprises of 148 pure LOC. The reason for this, one order of magnitude, improvement is on one hand justified by the expressiveness and effectiveness of Akkadian and the smart object model in general and, on the other hand, by the extraneous code statements that are necessary in general-purpose, compiled, statically typed, class-based programming languages (i.e., boilerplate code). Also, the reference implementation necessitated the development of the state manager infrastructure to support the environment's architecture (see Figure 72). Furthermore, for applications that involve extensive mobile interactions, we expect even greater performance improvements from the utilization of the proposed approach.

5.8 Discussion

In this chapter we evaluated the proposed approach by discussing its features against some well-defined evaluation criteria. Specifically, we first evaluated the proposed approach against the high-level functional requirements, which we had identified in section 2.4. These characteristics were deemed essential for a complete approach towards supporting the programming of AmI environments. Subsequently, we assessed the performance of the proposed approach against a set of external, more generic, evaluation criteria that should be expected by the users of the framework in order for the latter to constitute an effective, intuitive and efficient development platform. Complementing the external criteria, we identified and discussed the features of the proposed approach against a set of internal, introspective criteria that should be considered in any well-rounded approach or method, regardless of how the latter is utilized within its application domain.

In this context, Figure 74 provides a general view by showing the number of features (y-axis) that the proposed approach implements, which specifically target and support the aforementioned evaluation criteria (x-axis).

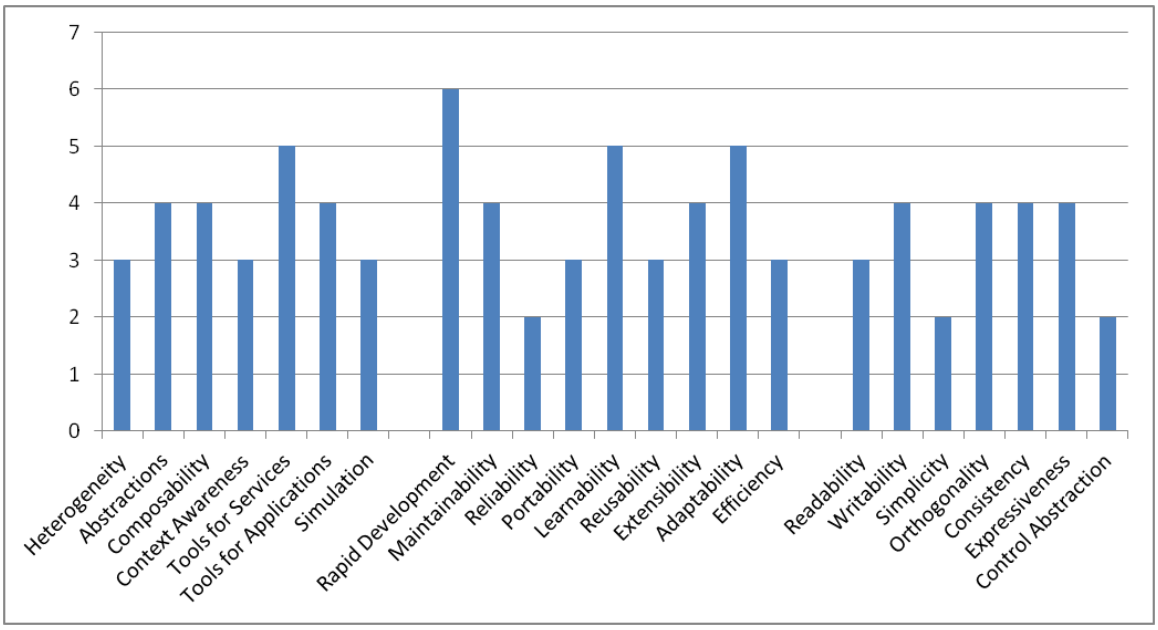


Figure 74 - Number of features that target and support specific evaluation criteria

Moreover, we showcased the characteristics and efficiency of the proposed approach through three examples with completely different goals and implementation strategies. The chain replication example showcased the flexibility and expressiveness of the proposed programming language for implementing the connection logic of smart object federations that can in turn sustain the interactions among smart objects for realizing the functionality of AmI environments. The movie scenario example showcased the expressiveness of the proposed approach for decomposing and implementing fully functional AmI environments that can be easily extended. This example also showcased strategies for adapting the functionality of individual smart objects at runtime without disrupting or affecting the functionality of the environment as a whole. Finally, the external chat objects application, showcased the effectiveness and agility of the approach for incorporating external mobile devices into the smart object environment. These mobile devices, utilizing an efficient asynchronous lightweight text-based TCP/IP protocol, can fully expose their functionality and utilize all the other available functionalities in the context of the smart object environment.

Finally, we outlined the implementation of a smart office environment using a high-level general purpose programming language utilizing the environment's resources via networked services. Subsequently, we outlined the implementation of the same environment utilizing the proposed approach with services and smart objects. Comparing the reference implementation with the smart object-based implementation, we observed an order of magnitude improvement in the number of the lines of code (LOC) required for the programming of the functionality of the environment.

Chapter 6
Conclusions

In this thesis, we proposed the design and implementation of a complete framework for modeling and implementing ambient intelligence (AmI) environments. Towards realizing this framework we defined flexible and intuitive architectural elements for modeling all the different entities that capture the structure, semantics and functionality of an AmI environment. We suggested systematic implementation processes supported by well-defined programming abstractions for realizing a robust architectural foundation and, finally, to further support and streamline all the phases of designing and programming AmI environments, we implemented a suite of comprehensive libraries and tools.

In this regard, in the context of AmI environments, we defined the concepts of services and smart objects. Services were provided as a means to model the slow-changing domain knowledge of the AmI environment and implement its sensitivity and responsiveness. Smart objects provided the means to model the environment's rapidly-changing behavior and implement its intelligence and adaptability. The proposed service middleware exposes the inherently heterogeneous computing system resources under a unified homogeneous platform. Its design and implementation succeeds in automating many tedious and error-prone programming, maintenance and deployment tasks, while allowing the development of services in five popular programming languages. Through any of the supported programming languages, creating a completely new software service or adapting a legacy application into a service to plug into an AmI environment is a rapid, intuitive process that respects the capabilities, strengths and conventions of the target programming language. We formally defined the core concepts of the smart object model, through which we could also define the notion of an AmI environment, its applications and its functionality. Aiming at improving the efficiency of the application of the smart object model, we designed and implemented an inherently asynchronous, dynamic, event-based programming language called Akkadian that allows for the accurate and intuitive description of the structure of smart objects and their interactions. Through Akkadian and its runtime, smart objects are able to fully utilize the functionality offered by diverse services built on top of diverse

technological platforms. The whole process for programming and deploying smart objects in their environment is driven by the provided modular graphical platform, called ObjectivSim. In the context of ObjectivSim, the proposed smart object environment enables the visualization of smart objects and their interactions, allowing for the simulation of smart object functionalities through direct interaction onto the environment's user interface. Finally, we defined the notion of external smart objects for expanding the design and implementation space of AmI environments by integrating within their context the functionality of external mobile devices.

Compared to existing approaches, the smart object model and its application through Akkadian, paired with the proposed service middleware, provide a practical and robust approach for naturally modeling the distinct computational elements and the utilization of resources that synthesize applications in the AmI domain, allowing for flexible, intuitive and succinct implementations. In this regard, we improved existing approaches (a) by maximizing the potential of the environment for incorporating heterogeneous technologies, (b) by providing sensible programming abstractions, (c) by enabling the implementation of higher-level functionalities by composing diverse services, (d) by providing a flexible model for representing and using the context of the user, (e) by providing libraries and tools for developing and deploying services, (f) by providing libraries and tools for developing and deploying the functionality of AmI environments, and (g) by providing the means for simulating the functionality of the environment.

6.1 Summary of Achievements

The achievements of the proposed approach and its implementation are the following:

- The definition of a comprehensive software architecture supported by intuitive abstractions for the implementation of AmI environments
- The design and implementation of programming libraries and tools for implementing type-safe services in five popular programming languages
- The design and implementation of a middleware infrastructure that enables the automatic deployment, update and failure recovery of the available services

- The formal definition of the smart object model that is used for modeling the functionality of AmI environments
- The design and implementation of a new high-level event-based programming language, called Akkadian, for implementing smart objects and their interactions
- The modular design and implementation of the smart object environment allowing the instantiation of smart objects through different implementations with different characteristics and properties
- The component-based design and implementation of ObjectivSim that incorporates a fully functional smart object development and deployment environment
- The implementation of the smart object environment in the context of ObjectivSim, enabling the visualization and simulation of smart object interactions
- The definition and implementation of the notion of external smart objects, realized through a lightweight text-based asynchronous message exchange protocol
- The complete formulation and definition of the development process of AmI environments through the use of the provided architectures, methods and tools

Overall, the achievements of this work are twofold. They are focused on both the architectural and theoretical aspects of building AmI environments, but also on the practical issues that stem from the actual development process.

6.2 Future Work

The future work in the context of the proposed approach and its implementation concerns both the proposed service middleware and the definition and implementation of the smart object model. In this context, for the proposed service middleware we aim to support through the provided tools the capability to automatically generate graphical user interfaces that fully implement a service with a given access interface or provide interaction elements for invoking all its methods and displaying all its emitted events. The former provides an effective way for testing the functionality of programs that use services and the latter for testing a specific service implementation. Another important future addition in the proposed middleware infrastructure and the provided libraries and tools, is the incorporation of a set of well-defined communication, access and deployment primitives

that will provide an intuitive and robust model for realizing security and privacy at the service construction and usage level. A set of well-defined security primitives at the service construction and deployment level will provide for a fine-grained and flexible security model in the context of the AmI environment.

Towards improving the proposed approach regarding the utilization of the smart object model via the proposed programming language, Akkadian, its runtime and tools ecosystem, our aim is to target security, persistence, semantic annotation, and enable the capability for environment-wide programming.

Specifically, concerning security in the smart object model, we plan on extending the notion of the scope relation for providing security and privacy primitives, allowing, thus, the definition of the notion of trust among smart object federations. Persistence, in the context of smart objects and the smart object environment, refers to the ability of maintaining their state and recovering from potential temporary failures. The decentralized nature of the smart object model and the modeling of its state through the well-defined notions of memory and services, allow for a comprehensive and intuitive persistence model. Moreover, introducing formal semantic annotation syntax for the functionality attached to smart object ports would allow for the introduction of a meta-programming layer enabling the utilization of higher-order events directly in the context of Akkadian programs, without requiring their prior explicit recognition and definition through the language's statements. This capability will further simplify the programming of smart objects in Akkadian and improve the reusability potential of meta-events in the context of complex AmI environments. Environment-wide programming, as future work, refers to the ability of the environment to respond to the creation and presence of smart object federations with specific structural properties within its context. In this regard, we plan to extend the definition and implementation of the smart object environment in order to provide the means for externally manipulating the interactions between smart objects in response to the formation of federations with specific structural properties. Through this mechanism, we anticipate the identification and provision of high-level "meta-recipes" for further simplifying the implementation of AmI environments and facilitating the formulation of environment-wide assertions for systematically validating aspects of their integrity and response to erroneous or unexpected conditions.

6.3 Closing Remarks

Ultimately, the cornerstone of the proposed approach, interweaved into all aspects of its design and implementation, is the provision of a canvas on which the architecture and development of ambient intelligence environments becomes a lucid, untainted picture that enables the systematic identification and extraction of frequently emergent patterns. These emergent patterns should be subsequently exploited for sustaining the evolution of the proposed architecture, methods and tools towards improving the creation process of ambient intelligence environments. In this regard, we consider that the proposed approach provides an ideal framework for encompassing all aspects of this coveted, necessary and inevitable evolution process.

Appendix A

Akkadian Grammar

```
akkadianProgram : stmt* EOF;

//*********************************************************************

stmt
     : PORT (ID|PORTID) ('(' taggedexprlist ')')? portStmtList
     | WHEN logicalExpression stmtlist

     | WHILE logicalExpression stmtlist
     | FOR idlist IN logicalexprlist (EXCEPT! logicalexprlist)? stmtlist
     | IF logicalExpression stmtlist: ELSE stmtlist)?

     | logicalExpression ';'
     | BREAK ';'
     | CONTINUE ';'
     ;

stmtlist
     : stmt
     | '{' stmt* '}'
     ;

WHEN: 'when';
PORT: 'port';

WHILE:    'while';
FOR:      'for';
IN:       'in';
EXCEPT:   'except';
IF:       'if';
ELSE:     'else';

BREAK:    'break';
CONTINUE: 'continue';

//*********************************************************************

portStmtList
     : portStmtStmt
     | '{' portStmtStmt* '}'
     | ';'
     ;

portStmtStmt
     : ID  primaryExpression ';'
     ;

//*********************************************************************
// Expressions
//
logicalExpression: booleanOrExpression (CONTAINS booleanOrExpression)*;
booleanOrExpression: booleanAndExpression (OR booleanAndExpression)*;
booleanAndExpression: equalityExpression (AND equalityExpression)*;
equalityExpression: relationalExpression ((EQ|NE) relationalExpression)*;
relationalExpression: additiveExpression ((LT|LE|GT|GE) additiveExpression)*;
```

```
additiveExpression: multiplicativeExpression ((ADD|SUB) multiplicativeExpression)*;
multiplicativeExpression: powerExpression ((MUL|DIV|MOD) powerExpression)*;
powerExpression: unaryExpression (POW unaryExpression)*;

LT   :    '<';
LE   :    '<=';
GT   :    '>';
GE   :    '>=';
EQ   :    '==';
NE   :    '!=';
AND :     '&&' | 'and';
OR   :    '||' | 'or';
ADD  :    '+';
SUB  :    '-';
MUL  :    '*';
DIV  :    '/';
MOD  :    '%';
POW  :    '**';
NOT  :    '!' | 'not';
CONTAINS: 'contains';

//*********************************************************************

unaryExpression
     : primaryExpression
     | NOT primaryExpression
     | SUB primaryExpression
     ;

primaryExpression
     : '(' logicalExpression ')'
     | term
     ;

//*********************************************************************

term: termPrefix (
               OBJLIT ID
               | OBJELEM logicalexprlist ']'
               | ASS logicalexprlist
               | CALL taggedexprlist ')'
          )*;

fragment termPrefix
     : ID
     | MUL ID
     | STRING
     | NUM
     | BOOLEAN
     | NULL
     | PORTID+
     | '[' taggedexprlist ']'
     ;

OBJELEM: '[';
OBJLIT: '.';
CALL:     '(';
ASS: '=';

//*********************************************************************

structElem: (structKey ':')? logicalExpression;
fragment structKey: ID|STRING;
taggedexprlist: structElem? (',' structElem)*;
```

```
idlist: ID (',' ID)*;
logicalexprlist: logicalExpression (',' logicalExpression)*;

////////////////////////////////////////////////////////////////////
// Primitive literals
//
NUM  : ('0'..'9')+ ('.' ('0'..'9')+)? | 'inf';
BOOLEAN: 'true' | 'false';
NULL: 'null';

STRING: '"' ( EscapeSequence | (~('\u0000'..'\u001f' | '\\' | '"' )))* '"';

PORTID: ('+'|'-') IdSequence;
ID:  ('@'|'#')? IdSequence;

WS: (' '|'\r'|'\t'|'\u000C'|'\n');

COMMENT: '/*' .* '*/';
LCOMMENT: '//' ~('\n'|'\r')* '\r'? '\n';

////////////////////////////////////////////////////////////////////
// Fragments
//
fragment IdSequence: ('a'..'z' | 'A'..'Z' | '_') ('a'..'z' | 'A'..'Z' | '_' | '0'..'9')*;
fragment EscapeSequence: '\\' ('n' | 'r' | 't' | '\"' | '\\' | UnicodeEscape);

fragment UnicodeEscape: 'u' HexDigit HexDigit HexDigit HexDigit;
fragment HexDigit: ('0'..'9'|'a'..'f'|'A'..'F');
```

Appendix B

Smart Office Objects Outline

Office User:

```
port +HandTracking("echo:///Sensor/HandTracking/office");
port +DistanceTracking("echo:///Sensor/DistanceSensor/office");

// Mobile Phone services
//
port -PhotoAlbum;

when InScope(*o) and o.name == MobilePhone
    self.Link(-PhotoAlbum, o);

when OutScope(*o) and o.name == MobilePhone
    self.Unlink(-PhotoAlbum, o);

when +self.Moved(*dx, *dy)
    -PhotoAlbum+self(self).Move(dx, dy);

when -PhotoAlbum.AddedPhoto(*photo)
    self.MostRecentPhoto = photo;
```

Desk Pad:

```
port +rfidForPadUnfolding("echo:///Sensor/RFIDReader/officedeskpad");

port +objTracking("echo:///Sensor/ObjectTracking/office");

port +speech("echo:///Algorithm/SpeechSynthesizer/office");
port +lights("echo:///Thing/Lights/office");
port +desktopUI("echo:///Algorithm/DeskPadUI/office");

when self.useUIRFIDEvents
    port +rfid("input:///ItemValue/Added/Removed");

when not self.useUIRFIDEvents
    port +rfid("echo:///Sensor/RFIDReader/office");

when Active()
    self.useUIRFIDEvents = true;

// Office User's services
//
port -HandTracking;
port -DistanceTracking;

when InScope(*o) and o.name == OfficeUser
{
    self.Link(-HandTracking, o);
    self.Link(-DistanceTracking, o);
```

```
    self.User = o;
}

when OutScope(*o) and o.name == OfficeUser
{
    self.Unlink(-HandTracking, o);
    self.Unlink(-DistanceTracking, o);
    self.Unlink(-PenTracking, o);
}

// Screen's services
//
port -VideoPlayer;
port -AppLauncher;
port -ImageViewer;
port -PresentationViewer;

when InScope(*o) and o.name == Screen
{
    self.Link(-VideoPlayer, o);
    self.Link(-AppLauncher, o);
    self.Link(-ImageViewer, o);
    self.Link(-PresentationViewer, o);
}

// Photo frame's services
//
port -PhotoFrame; // We could have also used the service directly

when InScope(*o) and o.name == "PhotoFrame"
    self.Link(-PhotoFrame, o);

// General handling of Object placements on the desk-pad
//
when +rfid.Added(*item) and +objTracking.Position(*x, *y)
{
    +speech.Say(item + " Recognized!");
    +desktopUI.Animation(*x, *y);
}

when +rfid.Removed(*item)
    +desktopUI.HideTickMark();

when +rfidForPadUnfolding.Added("Unfolded Pad")
    +desktopUI.ExtendDesktopArea();

when +rfidForPadUnfolding.Removed("Unfolded Pad")
    +desktopUI.ReduceDesktopArea();

// Mobile phone behavior
//
when +rfid.Recognized("Mobile Phone") and +desktopUI.AnimationFinished(*x, *y)
    +desktopUI.ShowTickMark(x, y);

when +self.User.MostRecentPhoto and +rfid.Added("Mobile
Phone") and +desktopUI.Gesture("DragTickMark")
{
    -PenTracking.DisableTickMark();
    +desktopUI.ShowPhoto(self.User.MostRecentPhoto);
}

when +self.User.MostRecentPhoto and +desktopUI.Gesture(name == "FlickPhoto", direction
== "North")
    -ImageViewer.ShowImage(+self.User.MostRecentPhoto);
```

```
when +self.User.MostRecentPhoto and +desktopUI.Gesture(name == "FlickPhoto", direction
== "East")
    -PhotoFrame.ShowInstantImage(+self.User.MostRecentPhoto);

when +rfid.Removed("Mobile Phone")
{
    -ImageViewer.HideImage();
    -PhotoFrame.HideInstantImage();
}

// Presentation behavior
//
when +rfid.Added("Leaflet")
{
    -PresentationViewer.Start("FORTH");
    +lights.Dim();

    self.PresentationInProgress = true;
}

when +rfid.Added(item != "Leaflet") and self.PresentationInProgress
{
    self.PresentationInProgress = false;

    -PresentationViewer.Stop();
    +desktopUI.PresentationStop();
    +lights.TurnOn();
}

when +rfid.Removed("Leaflet")
    +desktopUI.PresentationStart("FORTH");

when +desktopUI.PresentationAnnotationPoint(*x, *y)
    -PresentationViewer.SetAnnotationPoint(x, y);

when self.PresentationInProgress and -HandTracking.Recognized(*pos)
{
    if pos == "Right"
    {
        +desktopUI.PresentationNextSlide();
        -PresentationViewer.NextSlide();
    }
    else if pos == "Left"
    {
        +desktopUI.PresentationPrevSlide();
        -PresentationViewer.PrevSlide();
    }
}

// Video Behavior
//
when +rfid.Added("VideoToken")
{
    -VideoPlayer.Start("Video Token");
    +lights.Dim();
}

when +rfid.Removed("Video Token")
{
    -VideoPlayer.Stop();
    +lights.TurnOn();
}

when +rfid.Added("Video Token") and -HandTracking.Recognized(*pos)
{
```

```
    if pos == "Right"
        -VideoPlayer.Resume();
    else if pos == "Left"
        -VideoPlayer.Pause();
}

when +rfid.Added("Video Token") and -DistanceTracking.DistanceChanged(*offset)
    -VideoPlayer.AdjustVolume(offset: offset * 10);

// Other applications
//
when +rfid.Added("Videoconference Token")
    -AppLauncher.StartApplication("Videoconference");

when +rfid.Removed("Videoconference Token")
    -AppLauncher.StopApplication("Videoconference");

when +rfid.Added("Game Token")
    -AppLauncher.StartApplication("Game");

when +rfid.Removed("Game Token")
    -AppLauncher.StopApplication("Game");

when +rfid.Added("Mail Token")
    +desktopUI.MailOpen();

when +rfid.Removed("Mail Token")
    +desktopUI.MailClose();
```

Screen:

```
port +VideoPlayer("echo:///Algorithm/VideoPlayer/officescreen");
port +AppLauncher("echo:///Algorithm/AppLauncher/officescreen");
port +ImageViewer("echo:///Algorithm/ImageViewer/officescreen");
port +PresentationViewer("echo:///Algorithm/PresentationViewer/officescreen");
```

Photo Frame:

```
port +PhotoFrame("echo:///Thing/PhotoFrame/office");
```

Mobile Phone:

```
port +PhotoAlbum;
port +Phone;
port +PosTracking;

when +PosTracking.PositionChanged(*dx, *dy)
    +self.Move(dx, dy);
```

Bibliography

[1] E. H. L. Aarts and J. L. Encarnação, *True Visions: The Emergence of Ambient Intelligence*. Springer, 2008.

[2] K. Ducatel, M. Bogdanowicz, F. Scapolo, J. Leijten, and J. C. Burgelma, *Istag: Scenarios for Ambient Intelligent in 2010*. ISTAG, 2001.

[3] E. Aarts, "Ambient intelligence: a multimedia perspective," *IEEE MultiMedia*, vol. 11, no. 1, pp. 12 – 19, Mar. 2004.

[4] M. Friedewald, O. Da Costa, Y. Punie, P. Alahuhta, and S. Heinonen, "Perspectives of ambient intelligence in the home environment," *Telematics and Informatics*, vol. 22, pp. 221–238, 2005.

[5] D. J. Cook, J. C. Augusto, and V. R. Jakkula, "Ambient intelligence: Technologies, applications, and opportunities," *Pervasive and Mobile Computing*, vol. 5, no. 4, pp. 277–298, Aug. 2009.

[6] M. Weiser, "The computer for the 21st century," *Scientific American*, vol. 265, no. 3, pp. 94–104, 1991.

[7] S. K. Card, T. P. Moran, and A. Newell, *The psychology of human-computer interaction*. CRC, 1986.

[8] R. Campbell, J. Al-Muhtadi, P. Naldurg, G. Sampemane, and M. D. Mickunas, "Towards Security and Privacy for Pervasive Computing," in in *Software Security — Theories and Systems*, M. Okada, B. C. Pierce, A. Scedrov, H. Tokuda, and A. Yonezawa, Eds. Springer Berlin Heidelberg, 2003, pp. 1–15.

[9] M. Langheinrich, "Privacy by Design — Principles of Privacy-Aware Ubiquitous Systems," in in *Ubicomp 2001: Ubiquitous Computing*, G. D. Abowd, B. Brumitt, and S. Shafer, Eds. Springer Berlin Heidelberg, 2001, pp. 273–291.

[10] G. Myles, A. Friday, and N. Davies, “Preserving privacy in environments with location-based applications,” *IEEE Pervasive Computing*, vol. 2, no. 1, pp. 56–64, Jan-Mar.

[11] K. Ren, W. Lou, K. Kim, and R. Deng, “A novel privacy preserving authentication and access control scheme for pervasive computing environments,” *IEEE Transactions on Vehicular Technology*, vol. 55, no. 4, pp. 1373–1384, July.

[12]J. I. Hong and J. A. Landay, “An architecture for privacy-sensitive ubiquitous computing,” in *Proceedings of the 2nd international conference on Mobile systems, applications, and services*, New York, NY, USA, 2004, pp. 177–189.

[13] G. D. Abowd, “Classroom 2000: An experiment with the instrumentation of a living educational environment,” *IBM Systems Journal*, vol. 38, no. 4, pp. 508–530, 1999.

[14] S. Keegan, G. M. P. O’Hare, and M. J. O’Grady, “Easishop: Ambient intelligence assists everyday shopping,” *Information Sciences*, vol. 178, no. 3, pp. 588–611, 2008.

[15] G. Cabri, F. De Mola, L. Ferrari, L. Leonardi, R. Quitadamo, and F. Zambonelli, “The LAICA Project: An ad-hoc middleware to support Ambient Intelligence,” *Multiagent and Grid Systems*, vol. 4, no. 3, pp. 235–247, Jan. 2008.

[16] G. Szeder, “Low-Level Distributed Data Transfer Layer: The ChilFlow Middleware,” in in *Computers in the Human Interaction Loop*, A. Waibel and R. Stiefelhagen, Eds. Springer London, 2009, pp. 297–305.

[17] E. Aitenbichler, J. Kangasharju, and M. Mühlhäuser, “MundoCore: A light-weight infrastructure for pervasive computing,” *Pervasive and Mobile Computing*, vol. 3, no. 4, pp. 332–361, Aug. 2007.

[18] N. Georgantas, S. B. Mokhtar, Y. Bromberg, V. Issarny, J. Kalaoja, J. Kantarovitch, A. Gerodolle, and R. Mevissen, “The Amigo Service Architecture for the Open Networked Home Environment,” in *5th Working IEEE/IFIP Conference on Software Architecture, 2005. WICSA 2005*, 2005, pp. 295 –296.

[19] H. Chen, T. Finin, and A. Joshi, “An ontology for context-aware pervasive computing environments,” *The Knowledge Engineering Review*, vol. 18, no. 03, pp. 197–207, 2003.

[20] G. M. Youngblood, D. J. Cook, and L. B. Holder, “Managing Adaptive Versatile environments,” *Pervasive and Mobile Computing*, vol. 1, no. 4, pp. 373–403, Dec. 2005.

[21] A.-M. Vainio, M. Valtonen, and J. Vanhala, “Proactive fuzzy control and adaptation methods for smart homes,” *Intelligent Systems, IEEE*, vol. 23, no. 2, pp. 42–49, 2008.

[22] T. Reenskaug and J. O. Coplien, “The DCI architecture: A new vision of object-oriented programming,” *An article starting a new blog:(14pp) http://www. artima. com/articles/dci_vision. html*, 2009.

[23] T. Reenskaug, P. Wold, and O. A. Lehne, *Working with objects: the OOram software engineering method*. Manning, 1996.

[24] T. Reenskaug, *The Common Sense of Object Orientated Programming*. 2008.

[25] J. M. Spivey, *The Z notation: a reference manual*. Hertfordshire, UK, UK: Prentice Hall International (UK) Ltd., 1992.

[26] M. E. Pollack, “Intelligent technology for an aging population: The use of AI to assist elders with cognitive impairment,” *AI magazine*, vol. 26, no. 2, p. 9, 2005.

[27] D. H. Stefanov, Z. Bien, and W. C. Bang, “The smart house for older persons and persons with physical disabilities: structure, technology arrangements, and perspectives,” *Neural Systems and Rehabilitation Engineering, IEEE Transactions on*, vol. 12, no. 2, pp. 228–250, 2004.

[28] A. I. Dounis and C. Caraiscos, “Advanced control systems engineering for energy and comfort management in a building environment—A review,” *Renewable and Sustainable Energy Reviews*, vol. 13, no. 6–7, pp. 1246–1261, Aug. 2009.

[29] J. Krumm and E. Horvitz, “Predestination: Inferring destinations from partial trajectories,” *UbiComp 2006: Ubiquitous Computing*, pp. 243–260, 2006.

[30] S. A. Velastin, B. A. Boghossian, B. P. L. Lo, J. Sun, and M. A. Vicencio-Silva, “Prismatica: Toward ambient intelligence in public transport environments,” *Systems, Man and*

Cybernetics, Part A: Systems and Humans, IEEE Transactions on, vol. 35, no. 1, pp. 164–182, 2005.

[31] J. Augusto, J. Liu, and L. Chen, “Using ambient intelligence for disaster management,” in *Knowledge-Based Intelligent Information and Engineering Systems*, 2006, pp. 171–178.

[32] V. Jones, G. Karagiannis, and S. Heemstra de Groot, “Ad hoc networking and ambient intelligence to support future disaster response,” 2005.

[33] G. Margetis, P. Koutlemanis, X. Zabulis, M. Antona, and C. Stephanidis, “A smart environment for augmented learning through physical books,” in *Multimedia and Expo (ICME), 2011 IEEE International Conference on*, 2011, pp. 1–6.

[34] C. Magerkurth, A. D. Cheok, R. L. Mandryk, and T. Nilsen, “Pervasive games: bringing computer entertainment back to the real world,” *Computers in Entertainment (CIE)*, vol. 3, no. 3, pp. 4–4, 2005.

[35] D. Grammenos, Y. Georgalis, N. Kazepis, G. Drossis, and N. Ftylitakis, “The booTable experience: iterative design and prototyping of an alternative interactive tabletop,” in *Proceedings of the 8th ACM Conference on Designing Interactive Systems*, 2010, pp. 272–281.

[36] D. Grammenos, X. Zabulis, D. Michel, P. Padeleris, T. Sarmis, G. Georgalis, P. Koutlemanis, K. Tzevanidis, A. Argyros, and M. Sifakis, “Macedonia from Fragments to Pixels: A Permanent Exhibition of Interactive Systems at the Archaeological Museum of Thessaloniki,” *Progress in Cultural Heritage Preservation*, pp. 602–609, 2012.

[37] R. T. Watson, L. F. Pitt, P. Berthon, and G. M. Zinkhan, “U-commerce: expanding the universe of marketing,” *Journal of the Academy of Marketing Science*, vol. 30, no. 4, pp. 333–347, 2002.

[38] S. Helal, W. Mann, H. El-Zabadani, J. King, Y. Kaddoura, and E. Jansen, “The Gator Tech Smart House: a programmable pervasive space,” *Computer*, vol. 38, no. 3, pp. 50 – 60, Mar. 2005.

[39] B. de Ruyter and E. Aarts, “Ambient intelligence: visualizing the future,” in *Proceedings of the working conference on Advanced visual interfaces*, New York, NY, USA, 2004, pp. 203–208.

[40] D. Sánchez, M. Tentori, and J. Favela, “Activity recognition for the smart hospital,” *Intelligent Systems, IEEE*, vol. 23, no. 2, pp. 50–57, 2008.

[41] A. F. Bobick, S. S. Intille, J. W. Davis, F. Baird, C. S. Pinhanez, L. W. Campbell, Y. A. Ivanov, A. Schütte, and A. Wilson, “The KidsRoom: A perceptually-based interactive and immersive story environment,” *Presence*, vol. 8, no. 4, pp. 369–393, 1999.

[42] Y. Shi, W. Xie, G. Xu, R. Shi, E. Chen, Y. Mao, and F. Liu, “The smart classroom: merging technologies for seamless tele-education,” *IEEE Pervasive Computing*, vol. 2, no. 2, pp. 47–55, 2003.

[43] A. Fox, B. Johanson, P. Hanrahan, and T. Winograd, “Integrating information appliances into an interactive workspace,” *Computer Graphics and Applications, IEEE*, vol. 20, no. 3, pp. 54–65, 2000.

[44] D. Grammenos, Y. Georgalis, N. Partarakis, X. Zabulis, T. Sarmis, S. Kartakis, P. Tourlakis, A. Argyros, and C. Stephanidis, “Rapid Prototyping of an AmI-Augmented Office Environment Demonstrator,” *Human-Computer Interaction. Ambient, Ubiquitous and Intelligent Interaction*, pp. 397–406, 2009.

[45] K. Arnold, J. Gosling, and D. Holmes, *The Java programming language*, vol. 2. Addison-wesley Reading, MA, 2000.

[46] B. Stroustrup, *The C++ programming language*. Addison-Wesley Longman Publishing Co., Inc., 1997.

[47] G. Van Rossum, *Python programming language*. 1994.

[48] J. Koch and I. Fliege, “AmICom-Middleware Support for Ambient Communication.”

[49] Digital Equipment Corporation, Object Management Group, and X/Open Company, *The Common object request broker: architecture and specification, revision 1.1*. New York, NY: John Wiley, 1992.

[50] B. W. Kernighan and D. M. Ritchie, *The C programming language*. Prentice Hall, 1988.

[51] M. Eisenhauer, P. Rosengren, and P. Antolin, "A Development Platform for Integrating Wireless Devices and Sensors into Ambient Intelligence Systems," in *6th Annual IEEE Communications Society Conference on Sensor, Mesh and Ad Hoc Communications and Networks Workshops, 2009. SECON Workshops '09*, 2009, pp. 1 –3.

[52] S. I. G. Bluetooth, "Specification of the Bluetooth System, version 1.1," *http://www. bluetooth. com*, 2001.

[53] I. C. S. L. M. S. Committee, *Wireless LAN medium access control (MAC) and physical layer (PHY) specifications*. IEEE Std, 1997.

[54] M. Mouly, M.-B. Pautet, and T. Foreword By-Haug, *The GSM system for mobile communications*. Telecom Publishing, 1992.

[55] C. Bettstetter, H.-J. Vogel, and J. Eberspacher, "GSM phase 2+ general packet radio service GPRS: Architecture, protocols, and air interface," *Communications Surveys & Tutorials, IEEE*, vol. 2, no. 3, pp. 2–14, 1999.

[56] W. W. W. Consortium, "Xhtml 1.0: The extensible hypertext markup language," *A Reformulation of HTML 4 in XML*, vol. 1, 2000.

[57] D. Sachetti, R. Chibout, V. Issarny, C. Cerisara, and F. Landragin, "Seamless access to mobile services for the mobile user," in *Proc. of IEEE Int. Conference on Software Engineering*, 2004, pp. 801–804.

[58] V. Issarny, D. Sacchetti, F. Tartanoglu, F. Sailhan, R. Chibout, N. Levy, and A. Talamona, "Developing Ambient Intelligence Systems: A Solution based on Web Services," *Automated Software Engineering*, vol. 12, no. 1, pp. 101–137, 2005.

[59] F. Curbera, M. Duftler, R. Khalaf, W. Nagy, N. Mukhi, and S. Weerawarana, “Unraveling the Web services web: an introduction to SOAP, WSDL, and UDDI,” *Internet Computing, IEEE*, vol. 6, no. 2, pp. 86–93, 2002.

[60] A. Arkin, S. Askary, S. Fordin, W. Jekeli, K. Kawaguchi, D. Orchard, S. Pogliani, K. Riemer, S. Struble, and P. Takacsi-Nagy, “Web service choreography interface 1.0,” *World Wide Web Consortium*, p. W3C, 2002.

[61] G. Vanderhulst, K. Luyten, and K. Coninx, “Middleware for ubiquitous service-oriented spaces on the Web,” in *Advanced Information Networking and Applications Workshops, 2007, AINAW’07. 21st International Conference on*, 2007, vol. 2, pp. 1001–1006.

[62] A. Hejlsberg, S. Wiltamuth, and P. Golde, *C# language specification*. Addison-Wesley Longman Publishing Co., Inc., 2003.

[63] D. Box, D. Ehnebuske, G. Kakivaya, A. Layman, N. Mendelsohn, H. F. Nielsen, S. Thatte, and D. Winer, *Simple object access protocol (SOAP) 1.1*. 2000.

[64] M. Anastasopoulos, H. Klus, J. Koch, D. Niebuhr, and E. Werkman, “DoAmI-a middleware platform facilitating (re-) configuration in ubiquitous systems,” in *System Support for Ubiquitous Computing Workshop. At the 8th Annual Conference on Ubiquitous Computing (Ubicomp 2006)*, 2006.

[65] M. Anastasopoulos, D. Niebuhr, C. Bartelt, J. Koch, and A. Rausch, “Towards a reference middleware architecture for ambient intelligence systems,” in *ACM conference on object-oriented programming, systems, languages, and applications*, 2005.

[66] M. Fowler, *Inversion of control containers and the dependency injection pattern*. 2004.

[67] F. Ramparany, R. Poortinga, M. Stikic, J. Schmalenstroer, and T. Prante, “An open context information management infrastructure the IST-amigo project,” in *Intelligent Environments, 2007. IE 07. 3rd IET International Conference on*, 2007, pp. 398–403.

[68] S. Ben Mokhtar, A. Kaul, N. Georgantas, and V. Issarny, “Efficient semantic service discovery in pervasive computing environments,” *Middleware 2006*, pp. 240–259, 2006.

[69] D. Martin, M. Burstein, J. Hobbs, O. Lassila, D. McDermott, S. McIlraith, S. Narayanan, M. Paolucci, B. Parsia, and T. Payne, “OWL-S: Semantic markup for web services,” *W3C Member submission*, vol. 22, pp. 2007–04, 2004.

[70] S. Shenker, “Specification of guaranteed quality of service,” 1997.

[71] H. Hagras, V. Callaghan, M. Colley, G. Clarke, A. Pounds-Cornish, and H. Duman, “Creating an Ambient-Intelligence Environment Using Embedded Agents,” *IEEE Intelligent Systems*, vol. 19, no. 6, pp. 12–20, Nov. 2004.

[72] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau, “Extensible markup language (XML),” *World Wide Web Journal*, vol. 2, no. 4, pp. 27–66, 1997.

[73] L. Rabiner and B. Juang, “An introduction to hidden Markov models,” *ASSP Magazine, IEEE*, vol. 3, no. 1, pp. 4–16, 1986.

[74] S. Fine, Y. Singer, and N. Tishby, “The hierarchical hidden Markov model: Analysis and applications,” *Machine learning*, vol. 32, no. 1, pp. 41–62, 1998.

[75] S. Helal and C. Chen, “The Gator Tech Smart House: enabling technologies and lessons learned,” in *Proceedings of the 3rd International Convention on Rehabilitation Engineering & Assistive Technology*, New York, NY, USA, 2009, pp. 13:1–13:4.

[76] Osg. Alliance, “Osgi service platform, core specification, release 4, version 4.1,” *OSGi Specification*, 2007.

[77] M. L. Dertouzos, “The future of computing,” *Scientific American*, vol. 281, no. 2, pp. 36–47, 1999.

[78] U. Saif, H. Pham, J. M. Paluska, J. Waterman, C. Terman, and S. Ward, “A case for goal-oriented programming semantics,” in *Workshop on System Support for Ubiquitous Computing (UbiSys’ 03), 5th International Conference on Ubiquitous Computing (UbiComp 2003), Seattle, WA, USA*, 2003.

[79] J. P. Sousa and D. Garlan, “Aura: an architectural framework for user mobility in ubiquitous computing environments,” 2002.

[80] D. Garlan, D. P. Siewiorek, A. Smailagic, and P. Steenkiste, “Project aura: Toward distraction-free pervasive computing,” *Pervasive Computing, IEEE*, vol. 1, no. 2, pp. 22–31, 2002.

[81] M. Satyanarayanan, “Pervasive computing: Vision and challenges,” *Personal Communications, IEEE*, vol. 8, no. 4, pp. 10–17, 2001.

[82] A. Savidis and C. Stephanidis, “Distributed interface bits: dynamic dialogue composition from ambient computing resources,” *Personal and Ubiquitous Computing*, vol. 9, no. 3, pp. 142–168, 2005.

[83] J. P. Sousa, V. Poladian, D. Garlan, B. Schmerl, and P. Steenkiste, “Steps toward activity-oriented computing,” in *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, 2008, pp. 1–5.

[84] M. Román, C. Hess, R. Cerqueira, A. Ranganathan, R. H. Campbell, and K. Nahrstedt, “Gaia: a middleware platform for active spaces,” *ACM SIGMOBILE Mobile Computing and Communications Review*, vol. 6, no. 4, pp. 65–67, 2002.

[85] M. Román, C. Hess, R. Cerqueira, A. Ranganathan, R. H. Campbell, and K. Nahrstedt, “A middleware infrastructure for active spaces,” *Pervasive Computing, IEEE*, vol. 1, no. 4, pp. 74–83, 2002.

[86] S. Chetan, J. Al-Muhtadi, R. Campbell, and M. D. Mickunas, “Mobile gaia: a middleware for ad-hoc pervasive computing,” in *Consumer Communications and Networking Conference, 2005. CCNC. 2005 Second IEEE*, 2005, pp. 223–228.

[87] T. M. H. Reenskaug, “User-oriented descriptions of Smalltalk systems,” *Byte, 6*, vol. 8, 1981.

[88] G. E. Krasner and S. T. Pope, “A description of the model-view-controller user interface paradigm in the smalltalk-80 system,” *Journal of object oriented programming*, vol. 1, no. 3, pp. 26–49, 1988.

[89] R. Ierusalimschy, L. H. De Figueiredo, and W. C. Filho, “Lua-an extensible extension language,” *Software Practice and Experience*, vol. 26, no. 6, pp. 635–652, 1996.

[90] R. Grimm, J. Davis, E. Lemar, A. Macbeth, S. Swanson, T. Anderson, B. Bershad, G. Borriello, S. Gribble, and D. Wetherall, “System support for pervasive applications,” *ACM Transactions on Computer Systems (TOCS)*, vol. 22, no. 4, pp. 421–486, 2004.

[91] R. Grimm, J. Davis, B. Hendrickson, E. Lemar, A. MacBeth, S. Swanson, T. Anderson, B. Bershad, G. Borriello, and S. Gribble, “Systems directions for pervasive computing,” in *Hot Topics in Operating Systems, 2001. Proceedings of the Eighth Workshop on*, 2001, pp. 147–151.

[92] M. Kumar, B. A. Shirazi, S. K. Das, B. Y. Sung, D. Levine, and M. Singhal, “PICO: a middleware framework for pervasive computing,” *Pervasive Computing, IEEE*, vol. 2, no. 3, pp. 72–79, 2003.

[93] S. Kalasapur, M. Kumar, and B. Shirazi, “Seamless service composition (SeSCo) in pervasive environments,” in *Proceedings of the first ACM international workshop on Multimedia service composition*, 2005, pp. 11–20.

[94] S. Kalasapur, M. Kumar, and B. A. Shirazi, “Dynamic service composition in pervasive computing,” *Parallel and Distributed Systems, IEEE Transactions on*, vol. 18, no. 7, pp. 907–918, 2007.

[95] M. Wang, J. Cao, J. I. Siebert, V. Raychoudhury, and J. Li, “Ubiquitous intelligent object: Modeling and applications,” in *Semantics, Knowledge and Grid, Third International Conference on*, 2007, pp. 236–241.

[96] F. Kawsar, T. Nakajima, J. H. Park, and Y.-S. Jeong, “A Document Based Framework for Smart Object Systems,” in *Second International Conference on Future Generation Communication and Networking, 2008. FGCN ’08*, 2008, vol. 1, pp. 178 –183.

[97] F. Kawsar, T. Nakajima, and K. Fujinami, “A document centric approach for supporting incremental deployment of pervasive applications,” in *Proceedings of the 5th Annual International Conference on Mobile and Ubiquitous Systems: Computing, Networking, and Services*, ICST, Brussels, Belgium, Belgium, 2008, pp. 23:1–23:11.

[98] D. Winer, “RSS 2.0 Specification,” *Berkman Center for Internet & Society at Harvard Law School.< http://blogs. law. harvard. edu/tech/rss*, 2003.

[99] P. Verissimo, V. Cahill, A. Casimiro, K. Cheverst, A. Friday, and J. Kaiser, “Cortex: Towards supporting autonomous and cooperating sentient entities,” 2002.

[100] M. Wu, A. Friday, G. S. Blair, T. Sivaharan, P. Okanda, H. Duran-Limon, C.-F. Sørensen, G. Biegel, R. Meier, and E. FET, “Novel component middleware for building dependable sentient computing applications,” in *ECOOP’04 workshop on component-oriented approaches to context-aware systems (online proceedings)*, 2004.

[101] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler, “The nesC language: A holistic approach to networked embedded systems,” *SIGPLAN Not.*, vol. 38, no. 5, pp. 1–11, May 2003.

[102] C. Intanagonwiwat, R. Govindan, and D. Estrin, “Directed diffusion: a scalable and robust communication paradigm for sensor networks,” in *Proceedings of the 6th annual international conference on Mobile computing and networking*, 2000, pp. 56–67.

[103] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci, “Wireless sensor networks: a survey,” *Computer Networks*, vol. 38, no. 4, pp. 393–422, Mar. 2002.

[104] Y. Tanaka, “Proximity-Based Federation of Smart Objects: Liberating Ubiquitous Computing from Stereotyped Application Scenarios,” in in *Knowledge-Based and Intelligent Information and Engineering Systems*, vol. 6276, R. Setchi, I. Jordanov, R. Howlett, and L. Jain, Eds. Springer Berlin / Heidelberg, 2010, pp. 14–30.

[105] F. Boussinot and R. de Simone, “The ESTEREL language,” *Proceedings of the IEEE*, vol. 79, no. 9, pp. 1293 –1304, Sep. 1991.

[106] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, “The synchronous data flow programming language LUSTRE,” *Proceedings of the IEEE*, vol. 79, no. 9, pp. 1305 –1320, Sep. 1991.

[107] D. Greaves and D. Gordon, “Using simple pushlogic,” in *WEBIST 06: Proceedings of the second international conference on web information systems and technologies*, 2006.

[108] J. Aldrich, C. Chambers, and D. Notkin, “ArchJava: connecting software architecture to implementation,” in *Proceedings of the 24rd International Conference on Software Engineering, 2002. ICSE 2002*, 2002, pp. 187 –197.

[109] J.-C. Baillie, A. Demaille, Q. Hocquet, and M. Nottale, “Events! (Reactivity in urbiscript),” *arXiv:1010.5694*, Oct. 2010.

[110] J.-C. Baillie, “Urbi: Towards a universal robotic low-level programming language,” in *Intelligent Robots and Systems, 2005.(IROS 2005). 2005 IEEE/RSJ International Conference on*, 2005, pp. 820–825.

[111] O. Nierstrasz and T. Meijler, “Requirements for a composition language,” in in *Object-Based Models and Languages for Concurrent Systems*, vol. 924, P. Ciancarini, O. Nierstrasz, and A. Yonezawa, Eds. Springer Berlin / Heidelberg, 1995, pp. 147–161.

[112] F. Achermann, M. Lumpe, J. Schneider, and O. Nierstrasz, *PICCOLA - a Small Composition Language*. 1999.

[113] F. Curbera, Y. Goland, J. Klein, F. Leymann, Thatte, and S. Weerawarana, “Business process execution language for web services, version 1.1,” 2003.

[114] M. Blow, Y. Goland, M. Kloppmann, F. Leymann, G. Pfau, D. Roller, and M. Rowley, “BPELJ: BPEL for Java technology.” [Online]. Available: http://www.ibm.com/developerworks/library/specification/ws-bpelj/. [Accessed: 17-Aug-2012].

[115] M. R. Genesereth and S. P. Ketchpel, “Software agents,” *Commun. ACM*, vol. 37, no. 7, pp. 48–53, 147, 1994.

[116] H. S. Nwana, “Software agents: An overview,” *Knowledge engineering review*, vol. 11, no. 3, pp. 205–244, 1996.

[117] J. M. Bradshaw, Ed., *Software agents*. Cambridge, MA, USA: MIT Press, 1997.

[118] L. Bellissard, N. De Palma, A. Freyssinet, M. Herrmann, and S. Lacourte, "An agent platform for reliable asynchronous distributed programming," in *Proceedings of the 18th IEEE Symposium on Reliable Distributed Systems, 1999*, 1999, pp. 294 –295.

[119] F. Bellifemine, A. Poggi, and G. Rimassa, "JADE–A FIPA-compliant agent framework," in *Proceedings of PAAM*, 1999, vol. 99, p. 33.

[120] A. K. Dey, "Understanding and using context," *Personal and ubiquitous computing*, vol. 5, no. 1, pp. 4–7, 2001.

[121] E. Meijer and J. Gough, "Technical overview of the common language runtime," *language*, vol. 29, p. 7, 2001.

[122] R. T. Fielding, "Architectural styles and the design of network-based software architectures," University of California, 2000.

[123] O. M. G. CORBAservices, "Common object services specification," *Object Management Group*, 1995.

[124] S. Vinoski, "New features for CORBA 3.0," *Communications of the ACM*, vol. 41, no. 10, pp. 44–52, 1998.

[125] D. C. Schmidt, D. L. Levine, and S. Mungee, "The design of the TAO real-time object request broker," *Computer Communications*, vol. 21, no. 4, pp. 294–324, 1998.

[126] G. Brose, "JacORB: Implementation and design of a Java ORB," in *Proceedings of DAIS*, 1997, vol. 97, pp. 143–154.

[127] A. Wollrath, R. Riggs, and J. Waldo, "A Distributed Object Model for the Java^ T^ M System," *Computing Systems*, vol. 9, pp. 265–290, 1996.

[128] D. Grisby, S.-L. Lo, and D. Riddoch, "The omniORB version 4.1 User's Guide," *Apasphere Ltd. and AT&T Laboratories Cambridge*, 2006.

[129] "IIOP.NET." [Online]. Available: http://iiop-net.sourceforge.net/. [Accessed: 16-Mar-2013].

[130] R. Srinivasan, "RPC: Remote procedure call protocol specification version 2," 1995.

[131] T. Berners-Lee, R. Fielding, and H. Frystyk, *Hypertext transfer protocol--HTTP/1.0*. May, 1996.

[132] M. Henning, "A new approach to object-oriented middleware," *Internet Computing, IEEE*, vol. 8, no. 1, pp. 66–75, 2004.

[133] R. Stallman, "Gnu general public license," *Free Software Foundation, Inc., Tech. Rep*, 1991.

[134] M. Slee, A. Agarwal, and M. Kwiatkowski, "Thrift: Scalable cross-language services implementation," *Facebook, Jan*, 2007.

[135] "Apache Etch." [Online]. Available: http://etch.apache.org/. [Accessed: 16-Mar-2013].

[136] P. Tůma and A. Buble, "Overview of the CORBA performance," in *Proceedings of the 2002 EurOpen. CZ Conference*, 2002.

[137] D. Grammenos, X. Zabulis, D. Michel, T. Sarmis, G. Georgalis, K. Tzevanidis, A. Argyros, and C. Stephanidis, "Design and development of four prototype interactive edutainment exhibits for museums," in in *Universal Access in Human-Computer Interaction. Context Diversity*, Springer, 2011, pp. 173–182.

[138] D. Grammenos, G. Margetis, P. Koutlemanis, and X. Zabulis, "Paximadaki, the game: creating an advergame for promoting traditional food products," in *Proceeding of the 16th International Academic MindTrek Conference*, 2012, pp. 287–290.

[139] R. Milner, J. Parrow, and D. Walker, "A calculus of mobile processes, i," *Information and computation*, vol. 100, no. 1, pp. 1–40, 1992.

[140] J. Julia, Y. Tanaka, and N. Spyratos, “Formalization of an RNA-inspired Middleware for Complex Smart Object Federation Scenarios,” presented at the PECCS, 2012, pp. 96–105.

[141] L. Lamport, “How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs,” *IEEE Transactions on Computers*, vol. C–28, no. 9, pp. 690 – 691, Sep. 1979.

[142] Y. Tanaka, “Proximity-based ad hoc federation among smart objects and its applications,” in *Proceedings of the 5th international conference on Soft computing as transdisciplinary science and technology*, 2008, pp. 1–2.

[143] J. Huehnergard and C. Woods, “Akkadian and Eblaite,” *The Cambridge Encyclopedia of the World’s Ancient Languages*, pp. 218–280, 2004.

[144] M. Mernik, J. Heering, and A. M. Sloane, “When and how to develop domain-specific languages,” *ACM Comput. Surv.*, vol. 37, no. 4, pp. 316–344, Dec. 2005.

[145] T. Berners-Lee, R. Fielding, and L. Masinter, *Uniform resource identifiers (URI): generic syntax*. RFC 2396, August, 1998.

[146] J. Smith, “WPF apps with the model-view-ViewModel design pattern,” *MSDN magazine*, no. 2009, 2009.

[147] T. J. Parr and R. W. Quong, “ANTLR: A predicated-LL (k) parser generator,” *Software: Practice and Experience*, vol. 25, no. 7, pp. 789–810, 1995.

[148] B. Brumfield, G. Cox, D. Hill, B. Noyes, M. Puleio, and K. Shifflett, *Developer’s Guide to Microsoft® Prism 4: Building Modular MVVM Applications with Windows® Presentation Foundation and Microsoft Silverlight®: Building Modular MVVM Applications with Windows® Presentation Foundation and Microsoft Silverlight®*. Microsoft Press, 2011.

[149] E. Meijer, “Reactive extensions (Rx): curing your asynchronous programming blues,” in *ACM SIGPLAN Commercial Users of Functional Programming*, 2010, p. 11.

[150] Z. Alliance, “ZigBee specification,” *ZigBee Document 053474r13*, pp. 344–346, 2006.

[151] B. Meyer, “Applying ‘design by contract’,” *Computer*, vol. 25, no. 10, pp. 40 –51, Oct. 1992.