

*Department for Informatics
University of Fribourg (Switzerland)*

Automatic Software Configuration

A Model for Service Provisioning in Dynamic and Heterogeneous Environments

THESIS

submitted to the Faculty of Science of the University of Fribourg (Switzerland) in conformity
with the requirements for the degree of
Doctor scientiarum informaticarum

submitted by

Simon SCHUBIGER-BANZ

of

Zürch (ZH), Uznach (SG), Solothurn (SO)

Thesis No. 1393
Printed at the University of Fribourg
December 12, 2002

Accepted by the Faculty of Science of the University of Fribourg (Switzerland) following the proposal of:

Prof. Béat HIRSBRUNNER, Université de Fribourg, Switzerland, Thesis Director and President of the Examination Committee;

Prof. Peter KROPF, Université de Montréal, Montréal, Canada, Examiner;

Prof. Vaidy SUNDERAM, Emory University, Atlanta GA, USA, Examiner;

Fribourg, October 11, 2002

The Thesis Director:

The Dean:



Prof. Béat HIRSBRUNNER



Prof. Dionys BAERISWYL

Acknowledgments

I wish to thank all who have helped me during the preparation of this dissertation. More specifically, I would like to thank a lot my wife Andrea and my son Noël for their support and tolerance during the ups and downs of my research. My thanks go also to my parents who kept me on track during more than two decades of schooling. I would like to thank Béat Hirsbrunner and my research group for providing me with an environment in which I was able to lead my research very freely. I would then like to thank my examiners, Peter Kropf and Vaidy Sunderam for reading and commenting on my work.

During the years I spent in Fribourg, a number of other people have also brought their more personal contribution to this work. I would like to mention (in chronological order) Stefan Müller, Matthias Specht, Oliver Hitz, Stéphane Recrosio, Oliver Krone, Lukas Theiler, Antony Robert, Sergio Maffioletti, Amine Tafat-Bouزيد, and Alessio Gaspar who were and are a great help, as students, colleagues and friends.

And last but not least thanks to our secretary team, Sylvie, Marie-Claire, Yvette, and Nicole, who were always open for a good laugh and a healthy bit of criticism.

Zusammenfassung

Diese Dissertation behandelt drei fundamentalen Problemen, die im Umfeld von Ubiquitous Computing und Web-Diensten auftreten. Es handelt sich um Heterogenität sowie statische und dynamische Konfiguration.

Das Heterogenitätsproblem wird definiert und anhand von Beispielen in den Bereichen Anwendungen, Middleware, Kommunikation und Hardware illustriert. Anschliessend werden die heute verwendeten Lösungen präsentiert.

Das statische Konfigurationsproblem tritt vor allem während der Entwicklung und der Umsetzung von Software und Hardware auf. Im Bereich des Software-Engineering wird viel Forschung betrieben, welche zu Architekturbeschreibungssprachen und vielen Software-Konfigurations-Management-Werkzeugen geführt hat. Das statische Konfigurationsproblem wird definiert und anhand von heutigen Werkzeugen beleuchtet.

Weiter hat das Konfigurationsproblem auch einen dynamischen Aspekt, im Speziellen in Umgebungen wie Ubiquitous Computing wo die Konfiguration schnell ändert oder dem Internet, wo neue Dienste täglich aufgeschaltet werden oder verschwinden. Es wird eine Definition gegeben und es werden sowohl Software-Systeme, die für dynamische Umgebungen entwickelt wurden, als auch der Aspekt der dynamisch sich ändernden Hardwarekonfiguration betrachtet.

Als wichtigstes Resultat dieser Arbeit wird das COCA Model vorgestellt. Zudem werden die sieben Hauptelemente, welche das Design und die Entwicklung von Applikationen für heterogene und dynamische Umgebungen vereinfachen, eingeführt. Diese sieben Elemente sind:

Ressourcen als die fundamentale Abstraktion innerhalb von COCA und die atomaren Einheiten, die manipuliert werden.

Kontexte sind Behälter, die Ressourcen enthalten und deren Zugriff sowie den Namensraum definieren.

Klassifikatoren sind die Grundlage für die semantische Abstraktion und assoziieren Konzepte mit Ressourcen.

Konzepte sind die semantische Abstraktion, die von den Klassifikatoren eingeführt werden und die Konstruktion von Ontologien erlauben.

Ontologien bestehen aus Konzepten, die in Beziehung mit anderen Konzepten stehen. Zusammen mit einem Inferenzmechanismus erlauben sie eine erweiterte Semantik.

Relationen werden verwendet um die erweiterte Semantik in Ontologien auszudrücken.

Aktionen behandeln den dynamischen Aspekt und erlauben die Transformation von Ressourcen zwischen Konzepten.

Die Umsetzung von COCA erlaubt sowohl das automatische Konfigurieren von Software als auch das *Adressieren durch Konzepte* welches die Konstruktion von fehlertoleranten Systemen ermöglicht. COCA hilft nicht nur bei der Umsetzung von Software-Systemen, sondern unterstützt auch die Integration von objektorientierten Systemen, sowie funktionaler und logischer Programmierung. Autonome Agenten und das Entity-Relationship-Model finden eine Korrespondenz in COCA.

Die Vorgänger von COCA werden zusammen mit drei Implementationen des Modells vorgestellt. Der Ubiquitous Computing Demonstrator ist die erste COCA-Umsetzung, die alle Elemente implementiert und einfache automatische Software-Konfiguration und das Adressieren durch Konzepte verwendet. Schliesslich wird das Modell mit einer Auswahl von Systemen verglichen, welche ein ähnliches Anwendungsgebiet wie COCA haben oder andere Gemeinsamkeiten aufweisen.

Abstract

This dissertation addresses three fundamental problems, which have to be handled in application domains like ubiquitous computing and Web services. The three problems heterogeneity, static configuration, and dynamic configuration are not only considered to persist during the near future but to increase, as similar application domains will emerge.

The heterogeneity problem is defined and examples taken from applications, programming environments, middleware, communication, and hardware illustrate the importance of the heterogeneity problems on all these levels as well as the solutions in use today.

The static configuration problem mainly appears during development and deployment of software and hardware. There is much research going on in the software engineering field to better handle the static configuration aspects of computer systems. This lead to architecture description languages and software configuration management tools which together aim at covering the entire software construction process. The static configuration problem is defined and examples of static configuration support in current systems are given.

The configuration problem has also a dynamic aspect especially in environments like ubiquitous computing where the configuration consisting of mobile devices is easily modified or on the Internet where new services appear and disappear daily. A definition of the dynamic configuration problem together with examples of software systems conceived for dynamic environments are presented as well as how changing hardware configuration is handled today.

The COCA model as the key result of this dissertation introduces seven elements helping in the design and implementation of systems for heterogeneous and dynamic environments. These elements are:

Resources as the root abstraction of COCA and the atomic unit that can be handled.

Contexts are the containers resources live in and define resource access and naming.

Classifiers are a solution of the symbol grounding problem and provide the basic level of semantic abstraction by associating concepts with resources.

Concepts are the semantic abstractions grounded by classifiers and used to construct ontologies.

Ontologies consist of interrelated concepts and provide higher level semantics together with an inference mechanism.

Relations are used to represent higher level semantics in ontologies.

Actions capture the dynamic aspects of the model by transforming resources from one concept to another.

Applying COCA allows automatic configuration of software items and addressing by concept, which in turn enables the construction of fault tolerant systems. COCA not only helps in system design, it also accommodates important programming paradigms such as object oriented, functional, and logic programming. Autonomous agent systems and the entity relationship model have also a mapping to COCA resulting in a wide coverage of important concepts currently found in computer science.

Additionally, the two forerunners of COCA as well as three implementations of the model are presented. The ubiquitous computing demonstrator as a first COCA implementation for the ubiquitous computing domain not only realizes all the elements of the model but also a simple mechanism for automatic software configuration and uses addressing by concept. Finally, the model is compared with a selection of systems used in similar areas or sharing features with COCA.

Contents

Preface	1
I Introduction and Problems	3
1 The Heterogeneity Problem	5
1.1 Introduction	5
1.2 Applications	7
1.2.1 Introduction	7
1.2.2 Unicode	8
1.2.3 XML	9
1.2.4 KIF and KQML	11
1.2.5 Filters and Converters	12
1.3 Programming Environments	12
1.3.1 Introduction	12
1.3.2 C, POSIX, and Qt	13
1.3.3 Java	16
1.3.4 .NET	17
1.3.5 Guile	17
1.4 Middleware	18
1.4.1 Introduction	18
1.4.2 CORBA	18
1.4.3 SOAP	21
1.4.4 WOS	22
1.5 Communication	23
1.5.1 Introduction	23
1.5.2 HTTP and HTML	24
1.5.3 Bluetooth	26
1.6 Hardware Abstraction	28
1.6.1 Introduction	28
1.6.2 Unix and Mac OS X	28
1.6.3 Windows 2000	30
1.6.4 Ubiquitous Computing	31
1.7 Summary	33

2	The Static Configuration Problem	35
2.1	Introduction	35
2.2	Software Configuration Management	37
2.2.1	Component Repository and Version Control	37
2.2.2	Development support	38
2.2.3	Process support	38
2.3	Architecture Description Languages	38
2.4	Tools and Languages	40
2.4.1	Introduction	40
2.4.2	Make	41
2.4.3	Autoconf	42
2.4.4	CVS	43
2.4.5	Java	44
2.4.6	Eiffel	45
2.5	Component Systems	46
2.5.1	Introduction	46
2.5.2	CORBA	47
2.5.3	COM	48
2.5.4	Java Beans	48
2.5.5	Jini	50
2.5.6	Web services	52
2.6	Summary	55
3	The Dynamic Configuration Problem	57
3.1	Introduction	57
3.2	Changing Software	58
3.2.1	Introduction	58
3.2.2	Internet	59
3.2.3	WOS	59
3.2.4	Harness	61
3.2.5	Jini	63
3.3	System Resources	64
3.3.1	Introduction	64
3.3.2	Exclusive resources	64
3.3.3	Shared resources	65
3.3.4	Computational resources	66
3.3.5	Address Space	66
3.3.6	Stable Storage	67
3.4	Changing Hardware	68
3.4.1	Introduction	68
3.4.2	Windows 2000Plug and Play	69
3.4.3	Mac OS X	70
3.4.4	PACT's XPU128	74
3.4.5	Ubiquitous Computing	75
3.5	Summary	76

II	Model and Implementations	77
4	The COCA Model	79
4.1	Introduction	79
4.2	Resources and Context	81
4.3	Classifiers and Concepts	82
4.3.1	Semantics	82
4.3.2	Classifiers and Concepts	83
4.4	Ontologies and Relations	84
4.5	Actions	86
4.6	Implications	88
4.6.1	Object Oriented Programming	88
4.6.2	Functional and Data-Flow Programming	90
4.6.3	Logic Programming	91
4.6.4	Autonomous Agents	92
4.6.5	Entity Relationship Model	92
4.6.6	Addressing by Concept	93
4.6.7	Automatic Software Configuration	94
4.7	Summary	95
5	Implementations	97
5.1	Introduction	97
5.2	WebRes	98
5.3	WebCom	102
5.4	Ubiquitous Computing Demonstrator	106
5.5	Agent Based Classifier	109
5.6	SMASH	110
5.7	Summary	112
III	Discussion and Conclusion	113
6	Discussion and Related Work	115
6.1	Introduction	115
6.2	Distributed Resource Management	116
6.3	Semantic Web	120
6.4	Smart Tags	123
6.5	Focale	124
6.6	XCM	127
6.7	UbiDev	128
6.8	Open Questions	130
	Conclusion	133

List of Figures

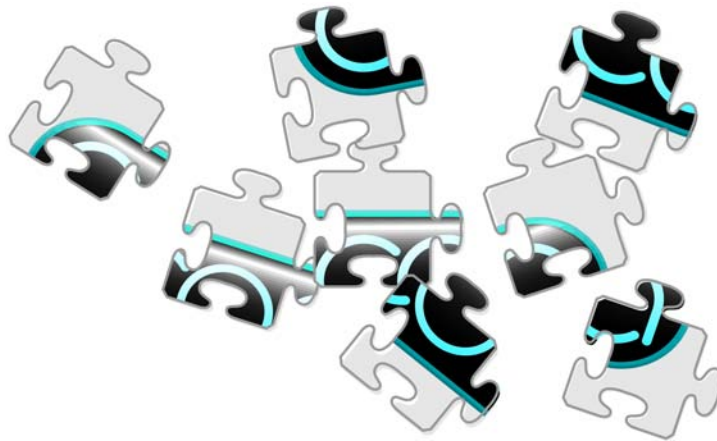
1.1	A simple ontology.	9
1.2	Emulating versus layered architecture for GUI abstraction.	14
1.3	The parts involved in a CORBA invocation.	19
1.4	Interface and implementation Repositories	19
1.5	Parts involved in a SOAP message exchange	22
1.6	Parts involved in a WOS request / response.	23
1.7	Parts involved in a HTTP request	25
1.8	Bluetooth Profile Structure	26
1.9	Windows 2000 hardware abstraction layers	31
1.10	Various ubiquitous computing devices in an application context.	32
2.1	A simple dependency graph	41
2.2	Preparing a software package for distribution with autoconf	42
2.3	The main operations performed with CVS.	43
2.4	CVS revisions, a CVS release (release_0.9, gray) and a CVS branch (1.2.1, on top).	43
2.5	Interface and implementation Repositories	47
2.6	The three main phases that lead to the use of a Jini service.	51
2.7	The ONE Web service architecture	53
2.8	The .NET Web service architecture	54
3.1	A user request propagates through two educative engines.	60
3.2	Structure of a web operating system node.	61
3.3	The harness distributed virtual machine (DVM) architecture.	62
3.4	The techniques in use today for stable storage	68
3.5	Windows 2000 Plug and Play integration (gray).	69
3.6	The relations for a SCSI I/O connection in Mac OS X.	71
3.7	The system integration of ACPI.	72
3.8	An USB configuration with three hubs and four functions.	73
3.9	The PACT architecture.	75
4.1	The basic idea behind COCA: The world is classified into concepts and actions are derived from matching concepts.	81
4.2	Union of contexts and sub-context construction.	82
4.3	Classifier operation.	83
4.4	An example of an ontology.	86
4.5	A resource transformation through an action.	87
4.6	The COCA ontology.	87

4.7	An autonomous agent and its mapping (in parentheses) to COCA.	92
4.8	Elements of the entity relationship model and their mapping to COCA.	93
4.9	A graph of actions for document conversion, DOC to GIF conversion highlighted.	95
5.1	An overview of the interactions between a resource set, two resource servers and two user interfaces.	99
5.2	The WebRes resource interface.	100
5.3	The WebRes user interface, the user is adding a printer to a resource set.	100
5.4	An example of a WebCom application and its relation to the WOS	103
5.5	An overview of the ubiquitous computing demonstrator.	106
5.6	A screenshot of the ubiquitous computing demonstrator.	107
5.7	An overview of the agent based classifier system.	109
5.8	The ontology editor used for the agent based classifier.	110
5.9	The operation of SMASH.	111
6.1	An example of a simple RDF statement.	121
6.2	An overview of the languages used in the Semantic Web.	122
6.3	An overview of the Smart Tags architecture.	123
6.4	Architecture of an observation VI.	125
6.5	Architecture of a manipulation VI.	126
6.6	An example of application imaging.	126

List of Tables

1.1	The open system interconnect (OSI) model.	24
2.1	The “.NET My Services” (Hailstorm) set of Web services.	55
6.1	Architectural layers of a homogeneous execution environment for Ubiquitous Interacting Devices (UbiDev).	129

Preface



This dissertation is the result of the research I conducted at the Department of Informatics of the University of Fribourg (DIUF) in the Parallelism and Artificial Intelligence (PAI) group. During the last few years, the field of distributed systems changed a lot and so did the direction of my research.

Web computing for example was then still in its infancy and today everybody in the industry is talking about Web services. The same holds for ubiquitous computing that was not widely known outside the research community but today a lot of people are speaking about it, even though sometimes under the name of pervasive computing. On the other side, interest in distributed *operating systems*¹ decreased and it is now widely agreed that collections of distributed services and resource management techniques running on top of established operating systems are the way to go. Nevertheless, the fundamental problems are still around and challenging.

It was mainly thanks to the freedom I enjoyed during my research and the support I received from my thesis director Prof. B  at Hirsbrunner that allowed me to follow my ideas even when they did not follow the initial proposal of this thesis. In fact, by looking at the initial proposal the outcome should now be a data-flow language for the Web operating system (WOS) - actually not at all what is presented in the next chapters.

But it is my firm believe that the result presented here is more interesting and has a broader use than just another language for the Web. By looking at three important problems in distributed computing, a new model emerged that looks at system composition and interaction from a different angle and will hopefully help in developing future application running in heterogeneous and dynamic environments.

¹A distributed operating systems does the resource management for local as well as for remote resources.

During the many discussions I had with my colleagues and the various presentations I gave, comments on this work ranged from “You can do that already with Windows”² to “This is a real paradigm shift - let’s forget OO and the rest”. In my opinion the result is somewhere in between and the range is mainly due to the problem I encountered communicating a new model. I tried to incorporate all the suggestions I received into the following chapters and hope that some points are now presented clearer than before. Still, it has to be said that the text is kept quite informal which may not be seen as the right approach for the definition of a new model. This choice was made mainly for two reasons: firstly, I find an informal definition more readable than a formal definition especially if the reader is not familiar with the specific formalism used. Clearly, this does not explain why there is no appendix with a formal definition of the model. The second and more important reason is that I consider the model not as a final and unquestioned status quo. Especially since research using the model is continuing at the DIUF in the ubiquitous computing domain, the model will certainly evolve and I am convinced that some room for interpretation will raise new and interesting questions.

This dissertation is broadly split in three parts which consists of the problems addressed, the model, and the related work:

The first part tries to motivate the subject by looking at three fundamental problems: heterogeneity, configuration, and dynamism as well as present the solutions available today. The range of examples in the first part was intentionally kept broad for two reasons: the first is motivation of the reader which will hopefully find an example he is familiar with and that helps him to understand the underlying problem. The second reason is my believe that for some domains only models addressing a broad range of issues at once may yield the necessary abstractions for dealing with systems of high complexity such as ubiquitous computing or Web services.

The second part presents the model which addresses the problems introduced in the first part with a handful of elements (seven to be exact) and gives a novel view on how systems using it can be designed. Besides addressing the problems of the first part, the model also gives a solution for the symbol-grounding problem and shows a way how applications can deal with and benefit from semantic abstractions. In order to not only have a model but also some verification, descriptions of the projects leading to the model as well as implementations of the model itself are given. It is this implementation chapter that gives a chronological overview of the forerunners and explains why the result is somewhat different from the initial proposal.

The third part finally concludes this dissertation by discussing the model in the light of related work. Several of these projects exhibit feature also found in the model although they emerged in parallel and the similarities were discovered in retrospective. This assures on the one hand the result of this dissertation and on the other hand I am hoping that this work may contribute to these projects in return .

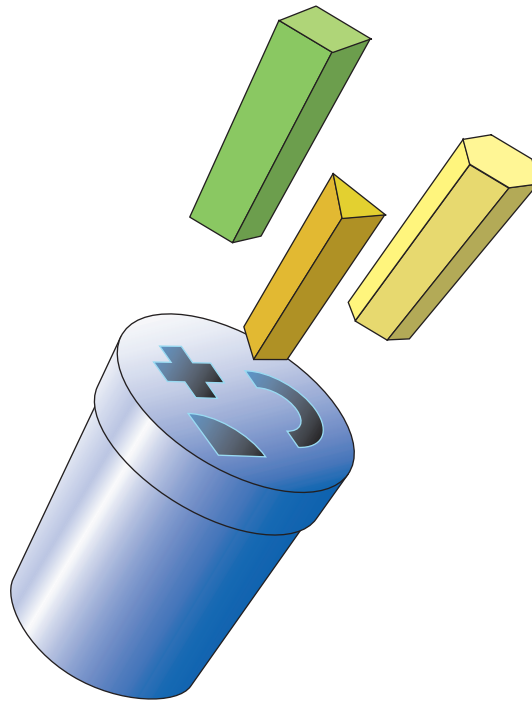
²This statement was not made by a Microsoft employee.

Part I

Introduction and Problems

Chapter 1

The Heterogeneity Problem



1.1 Introduction

This chapter looks at the first of the three problems that were leading to this thesis: heterogeneity. Although applications face heterogeneity in many ways today, in the beginning of computing heterogeneity was not a concern. The first computers in use were proprietary systems tuned for specific tasks with programs tightly coupled to the hardware. In these systems heterogeneity was totally absent.

Due to the fast advances in hardware technology, rewriting and adapting the programs became a major problem in time, quality, and cost. Obviously, the first hardware abstractions appeared, decoupling the machine peculiarities from the application task, making applications “portable” using a layered architecture. These abstractions mainly went off in two directions: CPU and memory were abstracted through compilers and languages and peripheral devices

and system resources by operating systems.

With the growth of available peripheral devices, standards emerged for physically interconnecting equipment of various vendors and interface standards soon appeared on the operating system level as well. Device drivers exposing a homogeneous interface to the application more and more replaced vendor specific libraries for accessing hardware. This is basically the situation still in place today, allowing within some limits the independent evolution of hardware and software while providing a high degree of compatibility.

It is interesting to note that programming languages today exist orthogonal¹ to the various hardware architectures whereas the major operating systems run on specific architectures only². Several languages were even specifically designed for different hardware architectures such as for example C [82]. This implies that languages have to deal with hardware (basically CPU) and operating system heterogeneity in some way. This led to a new application portability problem: even if the application was written in a portable language, various dependencies not abstracted by the language exist such as word size, endianness³, and operating system features.

The only solution so far is providing standardized languages extensions through libraries. Quite successful are the standard libraries for C and C++ that are part of the language specification [157]. But even these libraries are unable to hide all operating system details such as file name conventions and peripheral device access. Standards such as POSIX [103] try to define a least common denominator for various operating systems. Several operating systems are POSIX compliant today, but very few POSIX applications exist because the least common denominator approach is often too restrictive and applications are not competitive in features and performance compared to applications specifically written for a specific operating system. The other end is a framework providing a rich set of features by re-implementation (emulation) of missing parts. A quite prominent example is Qt [6], which is used for example in the KDE desktop environment on Linux.

With the emergence of networked computers, the heterogeneity problem started to span multiple hosts. Besides physically interconnecting the hosts, data representation and routing became additional problems. Layering was again the solution, as for example found in the prominent OSI-model [39] or in the Internet protocol (IP) as the de-facto standard for global networking. IP is the base for many higher level application protocols widely used today such as SMTP [85], POP [127], FTP [134] and HTTP [48] on platforms ranging from wrist-watches and cell phones to main-frames and super-computers.

Since data representation is orthogonal to interaction, both continue to develop independently. Higher-level communication abstraction beyond streams and packets based on

¹All major programming languages are available on most platforms.

²This is true for Windows and Mac OS. Although the Unix design is highly portable, only Linux, BSD, and Solaris are widely used on different architectures.

³Endianness means the order in which the bytes of a value larger than one byte are stored in memory. This affects integer, floating-point values, and pointers.

Little-endian (e.g. Intel and Alpha) stores the least significant byte on the lowest memory address (the word is stored little-end-first). For example, if the 32 bit value 0xC0CABABE is stored on memory address 0x1234 little-endian, the following bytes will occupy the memory positions:

0x1234	0x1235	0x1236	0x1237
0xBE	0xBA	0xCA	0xC0

Big-endian (e.g. Sparc, m68k, network byte order) stores the most significant byte on the lowest memory address (the word is stored big-end-first). For example if the 32 bit value 0xC0CABABE is stored on memory address 0x1234 big endian, the following bytes will occupy the memory positions:

0x1234	0x1235	0x1236	0x1237
0xC0	0xCA	0xBA	0xBE

distributed object models are in widespread use such as CORBA [131], RMI [161], and SOAP [65]. XML [42] is likely to become the standard for data representation and for storage as well as for network communication beside many binary standards that will continue to exist.

All these standards perform very well on the “wiring” level. Almost all entities may potentially communicate with any other entity by means of common protocols and protocol converters (gateways). But once communication is established, what should the entities talk about? In fact, if they do not agree on the meaning, is higher-level interaction even possible? Communication protocols define very well the interaction between the entities for establishing communication as well as specifying the *structure* of the data exchanged. However, they often fall short in providing clues about the meaning of the *content* or explicitly do not address content at all. What is needed for higher-level interaction is an *agreement* on that meaning. Chapter 4 will address this shortcoming. The heterogeneity problem can thus be stated informally as:

Definition 1 *In order to make two heterogeneous entities interact, conventions between these entities have to be established which define the nature of the interaction and meaning of the communication between the entities. The heterogeneity problem is to find and properly implement these conventions for both entities in question.*

The rest of this chapter looks in a top-down manner at the heterogeneity problem starting with the application level and ending with the hardware layer by presenting some exemplary solutions in use today.

1.2 Applications

1.2.1 Introduction

Applications basically have to deal with heterogeneous data representations in the following three areas:

- **Main memory:** During application execution time most if not all of the application data is held in main memory for performance reasons. The data representation is optimized for the specific architecture the application is running on by using the platform endianness, floating point formats, memory address representations, and memory alignments.
- **Stable storage:** Part of the application data is frequently written to stable storage, for example when the user saves a document or a database flushes its caches. In the same way that applications usually live longer than hardware, data tends to live longer than the application that created it. Therefore, architecture, version, and application independent encoding is desirable for data representation on stable storage. Several environments today provide support for “serialization” of object structures (for example Java [61]) but still have shortcomings when it comes to versioning and incomplete files.
- **Network:** If the application wants to communicate with other parties, communication has to follow protocols and has to use agreed data representations. Whereas main memory and stable storage representations are normally proprietary, protocols and data representations are usually standardized and define the encoding of data types as well as the possible interactions.

Much work improving interoperability has been done on several levels. For example most CPUs today use word sizes that are multiples of eight bits, encode larger entities than bytes either as big- or little endian and have adopted the IEEE floating point format [76]. A major issue of the past decades, character encoding, is likely to be overcome at least in new applications thanks to Unicode.

But still most of the file formats are proprietary. XML is likely to change this situation where inter-operation is mandatory. But structured formats will solve the problem only partially since the semantics behind the structure remains in the application that created the file. Nevertheless, there are some file formats addressing the semantics of the encoded data as well. A prominent example is PostScript [169] that is actually a programming language for an interpreter defined by Adobe. Other formats represent the semantics even without reference to an interpreter such as the knowledge interchange format (KIF [57]).

The following sections will look in some more detail at solutions of the data representation problem.

1.2.2 Unicode

In the beginning computers were mainly used for numerical calculation in military and research applications. When they entered the enterprise market string and text processing became an important task beside calculation. Every vendor in these days used its own representation for characters that rendered data exchange almost impossible. A first step toward standardization was taken by the American standard code for information interchange (ASCII) initiative. ASCII was adopted in 1967 by the international standardization organization (ISO) with a 7 and a 6 bit encoding. Unfortunately, it focused on the standardization of the English speaking part of the world using the Latin alphabet. In order to support multi-national character sets, every vendor developed its own extension to ASCII, resulting in the same problem as before when non-ASCII encoding was used.

In countries using non-western scripts like Japan, the 8 bits used for character representation by many computer systems did not even offer enough combination to represent the whole character set. These users have developed in turn double byte character sets (DBCS) where each character is represented by either one or more bytes. The encoding then defines the interpretation of a byte string, complicating string manipulation and affecting performance.

In this light the Unicode initiative emerged around 1990, with the goal to solve the problem once and for all. Unicode is a subset of ISO 10646 ⁴, which is an international standard to encode all of the world's languages correctly on computers. It supports complex scripts like Arabic, Japanese and Chinese as well as dead languages such as Sanskrit.

Unicode is usually used in UCS transformation format 16 (UTF-16), a 16-bit encoding where each character occupies two bytes and therefore has a fixed size, simplifying manipulation and improving performance. Several encodings exist for Unicode beside UTF-16, an interesting one is UTF-8 that provides backward compatibility with ASCII:

0x0000 - 0x007F	0	[bits 0..6]		
0x0080 - 0x07FF	1 1 0	[bits 6..10]	1 0	[bits 0..5]
0x0800 - 0xFFFF	1 1 1 0	[bits 12..15]	1 0	[bits 6..11] 1 0 [bits 0..5]

⁴ISO 10646 defines the universal character set (UCS), a 31-bit character set wherein Unicode represents a 16-bit subset, the basic multilingual plane (BMP).

Unicode is supported for example by Windows 2000, Mac OS X, Java and several Unix libraries. All XML parsers must support the Unicode encodings UTF-8 and UTF-16, making Unicode the standard of the future Web.

1.2.3 XML

The extensible markup language (XML [42]) is becoming the base of the “new Web” of smart data and electronic commerce. It is an approved recommendation of the World Wide Web consortium (W3C) and is used by many Web sites, applications, and industry standards. XML is a language in a long history of markup languages. The idea behind markup is to insert special “tags” into documents to simplify automatic document processing. This idea was picked up by several vendors, which in turn developed their proprietary markup.

In 1986 an alternative to these systems was issued as the standard generalized markup language (SGML). The biggest contribution of SGML beside standardization was the document type that defined (as a grammar) the document structure in terms of components and their ordering.

The hypertext markup language (HTML) is a SGML application developed in 1991, defining markup for headings, bullet lists, hypertext links, and other elements. Although HTML was a SGML application, most web browsers ignored the document structure and did not enforce any particular element ordering. This led to invalid HTML documents that were rendered differently by different browsers and made automatic processing difficult if not impossible. In addition, HTML mixes structural elements such as headings and lists with formatting instructions such as boldface and left-alignment.

XML was created to remedy the problems generated by HTML by customizing SGML:

- Specific choices of syntax characters such as “<” and “>”
- Empty elements can either have a normal start- and end tag with no content (<tag></tag>) or an empty-element tag (<tag/>)
- Tag omission is not optional
- Element ordering must follow a document type definition (DTD)
- The DTD need not to be present

A DTD provides a “grammar”, a body of rules about the allowable ordering of a document’s “vocabulary” of element types thereby defining the set of possible documents. Since DTDs are still considered not precise enough to define sets of documents a new standard around XML schemas is being developed [45, 174, 18].

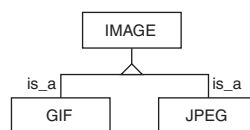


Figure 1.1: A simple ontology.

As an example consider the simple ontology⁵ depicted in figure 1.1 that may be represented in the following XML document:

```
<?xml version="1.0"?>
<!DOCTYPE ontology SYSTEM "ontology.dtd">
<ontology>
  <concepts>
    <concept name="IMAGE"/>
    <concept name="JPEG"/>
    <concept name="GIF"/>
  </concepts>
  <relations>
    <relation name="is-a">
      <concept name="GIF"/>
      <concept name="IMAGE"/>
    </relation>
    <relation name="is-a">
      <concept name="JPEG"/>
      <concept name="IMAGE"/>
    </relation>
  </relations>
</ontology>
```

Which refers to the following DTD that defines the set of all possible ontologies:

```
<!ELEMENT ontology (concepts, relations)>
<!ELEMENT concepts (concept*)>
<!ELEMENT relations (relation*)>
<!ELEMENT concept EMPTY>
<!ATTLIST concept name ENTITY #REQUIRED>
<!ELEMENT relation (concept+)>
<!ATTLIST relation name ENTITY #REQUIRED>
```

The above DTD reads as an *ontology* consists of *concepts* and *relations*. *Concepts* consists of zero or more *concept*. *Relations* consists of zero or more *relation*. *Concept* is an empty element and has an *name* which is required. *Relation* consists of one or more concepts and an attribute list consisting of a *name* which is required.

XML documents are mainly used on stable storage or as data representation over a network link. For manipulation in memory, applications use an XML parser that transforms an XML document into a document object model (DOM) that represents the XML document as a tree of objects that are easily navigated and manipulated through memory references and method invocations.

XML gives a human readable, extensible way for structuring documents as well as their verification. With additions such as the XML style sheet language (XSL), structure-based transformations are possible, resulting in a powerful conversion architecture. Unfortunately, XML addresses structure only and leaves semantics to the application.

⁵An understanding of ontologies is not required for this example. It can also be seen as an inheritance graph. Ontologies are discussed in detail in chapter 4.

1.2.4 KIF and KQML

Whereas XML simplifies the automatic processing of document *structure*, the knowledge interchange format (KIF [57]) addresses the problem of automatic processing of *meaning*. The need for describing things in computers appears in every application. Finally, all applications model some real or virtual entities and their interactions. This is one of the reasons why the object-oriented model had such a success; it provides a model that allows easy handling of “things” or “objects”.

Inside an application the meaning of these “things” are (hopefully) clear to all cooperating parties. A problem arises when multiple parties are involved that not only have to exchange data but also the meaning of the data. Natural languages are very powerful in describing “things” but statements in these languages are not always clear and subject to different interpretations.

Symbolic logic is one of the tools developed by mathematicians to describe “things”. Facts, definitions, abstractions, inference rules, and constraints allow making statements about “things” that have unambiguous interpretations. KIF as such a logic language aims at describing “things” in computer systems, e.g. expert systems, databases, or intelligent agents. Special attention was paid to make it useful as an interchange format for other representations. Several translators for domain specific knowledge formats to KIF and back exist.

KIF is a prefix version of a first order predicate calculus with extensions to support non-monotonic reasoning⁶ and definitions with a Scheme⁷ like syntax.

The example ontology of figure 1.1 may be written in KIF as:

```
(ontology (is-a (concept "GIF") (concept "IMAGE")))
(ontology (is-a (concept "JPEG") (concept "IMAGE")))
```

Which states that the ontology consists of a relation *is-a* that relates the concepts *GIF* and *JPEG* to *IMAGE*.

Complementary to KIF, the knowledge query and manipulation language (KQML) allows interaction with a knowledge base, not necessarily represented by KIF. KQML is most useful for asynchronous communication among autonomous programs such as agents to query, stating, believing, requiring, achieving, subscribing, and offering information. KQML messages are called *performative* in that they are intended to perform some action by being sent. By adding new performatives, KQML may be extended for new application domains.

For example the above KIF base may be queried by sending

```
(ask-all :language KIF :reply-with query
:content (val ontology))
```

And receiving

```
(reply :language KIF :in-reply-to query
:content ((ontology (is-a (concept "GIF") (concept "IMAGE")))
(ontology (is-a (concept "JPEG") (concept "IMAGE"))))
```

⁶In first-order logic if the database of facts is increased, the set of logical conclusion grows *monotonically*. In non-monotonic reasoning this is not necessarily true. See [58] for a more complete discussion.

⁷Scheme is a dialect of LISP.

Whereas KIF serves as a representation for domain specific knowledge, KQML is an interaction language not necessarily to be used together with KIF. It is important to note that KIF is primarily an *external* representation for knowledge. Applications using KIF usually store knowledge in other representations internally which are then transformed to and from KIF for externalization. This process is similar to applications that use XML for externalization but manipulate different structures internally. The big difference between an XML document and a set of KIF statements is that XML provides no interpretation beside the structure of the document. KIF in contrast defines semantics for a basic set of predicates that allow unambiguous interpretation of KIF statements.

1.2.5 Filters and Converters

Despite the fact that powerful structured representations such as XML exist, data exchange is still a big problem. The root of the problem is missing semantics. As long as nobody exactly knows the semantics of a *paragraph* in a Microsoft Word document for example, how can it be possibly handled by another program? Even if the other program has a notion of *paragraph* itself, they are unlikely the same and who guarantees that the semantics do not change from one version to another? Languages such as KIF provide a solution but unfortunately are not in widespread use, maybe due to the problem that specifying semantics to the last detail is a tedious task. There are also economic reasons why application knowledge is not externalized; this very knowledge might be the main asset of an application, which should not be given away for “free”.

The problem of incompatible data representations resulted in a plethora of filters, converters, and scripts, often adding to the complexity of today’s software systems. A reason for the multitude of filters and converters is the problem that perfect translation from n to m formats requires $n \times m$ programs. By using an intermediate format this may be reduced to $n + m$ programs but usually such a format does not exist. A solution lies somewhere in-between as the ubiquitous computing demo, which deals with data conversion, will illustrate in section 5.4.

Instead of speaking in terms of formats it might be better to define some properties of the source and destination and constrain the transformation in terms of these properties. For example for some type of application it is not necessary that the exact formatting of a document is preserved but its content must be invariant under the transformation. Attribute schemas as used in the WebCom project (see section 5.3) for example try to represent documents in terms of their attributes and allow the formulation of rules over these attributes. A rule engine then enforces these rules by applying appropriate transformations on the attributes. The Java activation framework (JAF [26], see also 2.5.4) takes a similar approach with data handlers detecting the nature of the data and deriving from there a set of possible commands to manipulate the data.

1.3 Programming Environments

1.3.1 Introduction

Programming environments followed closely the evolution of the hardware. On the one hand the hardware provided more and more power to the language designer and compiler writers. On the other hand the languages influenced the hardware design. In the early

days of programming, machine language was standard, meaning no abstraction at all. The programmer wrote directly the ones and zeros to main memory and started execution.

The next step was to render the ones and zeros more read- and writeable for humans by assigning abbreviations to them, called mnemonics. Obviously, programs written with these mnemonics were not directly machine processable, they had to be translated to machine language by yet another program called “assembler”. Assembly language was still strongly tied to the machine architecture; every CPU had its own set of mnemonics.

Around 1950, first high-level languages such as Fortran appeared which were largely CPU independent but required far more translation work than assemblers. Interest in high-level languages increased and started research in computer language and compiler design.

With a better understanding of programming languages and in parallel with hardware abstraction many new languages emerged. The successful imperative languages started to have an impact on the hardware design as well. One motivation behind the development of reduced instruction set computers (RISC) for example was the observation that compilers only used a small subset of the complex instruction set computers (CISC) instructions frequently. The logical conclusion was to build CPUs that only provide this subset but at a higher speed and emulating the seldom used instructions in software.

A side effect of this mutual influence was that today’s CPUs support very well imperative languages⁸ at the same time limiting the success of other paradigms such as functional programming (Haskell [79], LISP) or logic programming (Prolog [155]) which are designed around other computation models.

Beside compiled languages many successful interpreted languages are in use today such as Perl, Tcl/Tk, or PostScript. By giving up some performance, interpreted languages usually provide a high degree of abstraction. PostScript for example runs on several hundred different printers and platforms.

The following three sections will take a closer look at the abstraction provided by the three programming environments C, Java, and Guile each having its own goals and deals with the inherent heterogeneity in its own way.

1.3.2 C, POSIX, and Qt

The design of the C language was motivated by two contradicting goals: efficiency and portability. To be efficient the language had to be as close to the hardware as possible. To be portable it had to abstract the hardware as much as possible. Not an easy task at first sight and several compromises had been made by the designers to satisfy both goals. The most important task was an efficient CPU abstraction, which leads to platform dependencies such as:

- Basic data types: characters, integers, enumerations, floats and bit sets
- Endianness
- Memory alignment
- Evaluation order of expressions and function arguments

⁸Procedural, modular, and object oriented languages are also well supported by current architectures since they are just extensions of the imperative model.

Nevertheless, C and its successor C++ are quite successful maybe because the dependencies were clearly stated and it is possible to write portable code when paying attention to a few critical points.

A clever move was to keep the language itself small⁹ and to pack as much as possible into libraries. Pascal for example which was developed only a year before C had input/output and memory management still as a part of the language and libraries were vendor specific. C in contrast had from the beginning a set of standard libraries included by every compiler vendor which supplied memory management, input/output and various helpful tools such as string manipulation, mathematical functions, and sorting.

Surely, these libraries did not provide enough to write applications, requiring close interaction with the operating system. This was one reason why POSIX [103] was created, an attempt to develop a library for a portable operating system interface for computing environments. POSIX is now supported by all major Unix implementation, VAX/VMS, Windows 2000 and OS/2. POSIX.1 addresses file and terminal I/O, file and directory manipulation, the runtime environment, and process management.

Various extensions have been made to POSIX including real-time functionality, threads, security, networking, and bindings to other languages than C. Although it is possible to write interesting applications using POSIX only, real world applications consist of a mix of POSIX calls and platform specific code. The biggest contribution of POSIX is that the platform specific part can be minimized and isolated.

With the introduction of graphical user interfaces (GUIs) a new level of heterogeneity entered the picture. Since the GUI operations are very performance sensitive and vendors use these GUIs or better the proprietary “look and feel” to distinguish each from another, the result was a wide range of programming interfaces. In order to provide a cross-platform GUI abstraction, basically three approaches are possible as depicted in figure 1.2:

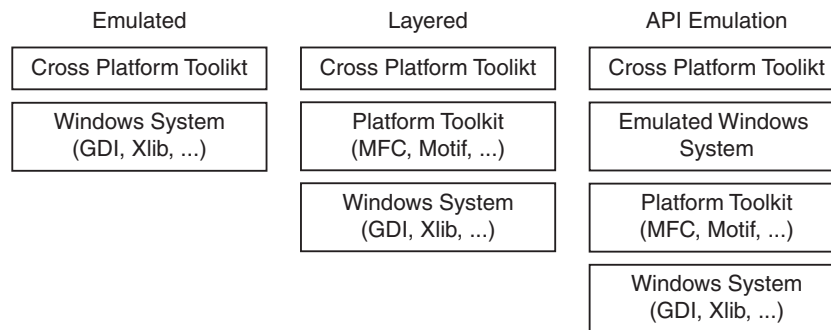


Figure 1.2: Emulating versus layered architecture for GUI abstraction.

- **Emulation:** This means using only the lowest graphical layer and emulating everything on top of it. The toolkit has to provide its own menus, buttons, dialogs, and window management implemented on top of the drawing primitives provided by the platform specific windows system.
- **Layered:** The cross platform toolkit defines a set of operations somewhere between the intersection of the features of all platform toolkits and the union of the features of all

⁹This is mainly true for C, C++ adds many features to C and is considered a large and complex language.

platform toolkits. The implementation consists in translating cross platform toolkit calls to the native toolkit where available and emulating the operations where no counterpart is available. This means that native widgets such as menus, buttons, dialogs and windows are used when available and emulated otherwise.

- **API emulation:** Still another approach is to emulate the windowing system API of one platform on another. For example a Unix/X11 MFC emulation libraries allows to quickly port applications using the Microsoft Foundation Classes (MFC) to a Unix/X11 environment.

The Qt toolkit developed by Trolltech AS [6] and widely used in the open source community is presented here as an example. Qt is a cross-platform C++ application framework implemented as a class library and provides a rich API. Qt offers a wide spectrum of features but focuses mainly on the GUI. The range of platforms supported by Qt is quite large: all major Unix versions using X11, all Windows versions since Windows 95, and an embedded version that can be targeted to any display and input hardware. Qt uses the emulation approach resulting in the re-implementation of all widgets with the specific “look and feel” of every supported platform. The “look and feels”, called styles, currently supported by Qt are: Motif, CDE, SGI, Motif Plus, Platinum (Mac OS), and Windows. Since no platform specific features are used for the “look and feel”, application can be tested with any style on any platform. It is even possible to switch the style at runtime and to write own styles. Key features of the Qt toolkit are:

- **Component Programming:** Qt objects can be seen as components using a signal and slot mechanism similar to resource linking used by WebRes (see 5.2).
- **Internationalization:** Unicode together with utility functions simplify localization in Qt.
- **Composite pattern** [54]: Qt follows the composite pattern wherein widgets form an easy to manipulate hierarchy.
- **Strategy pattern** [54]: Qt follows the strategy pattern which controls widget layout independently.
- **Qt designer:** A graphical GUI design tool is provided.
- **Device independent graphics:** Qt supports a powerful set of 2D graphics operations for drawing and bitmap manipulation.
- **Tool classes:** In addition to the GUI part Qt provides abstractions for files and directories, time and date, input/output, strings, collections, networking, and multi-threading.

Even though C introduced several platform dependencies for the sake of efficiency, the long experience with C code shows that portable code is feasible by paying attention to a few points. Using cross platform libraries such as POSIX and Qt, platform dependencies can be further reduced. C in conjunction with appropriate libraries provides an efficient abstraction while maintaining the full range of native features where necessary.

1.3.3 Java

Whereas C and third party libraries strive for source code portability, Java took a more radical approach. Instead of making the *source code* portable and hiding the heterogeneity behind a platform independent interface, Java made *machine code* portable. To not favor a specific CPU architecture Sun Microsystems defined a new *virtual* CPU called Java virtual machine (JVM).

This is not the first attempt to solve the portability problem by means of an underlying virtual CPU. For example the UCSD Pascal system [185] tried the same approach over a decade ago. Earlier attempts mainly failed due to the performance loss implied by the added interpreter layer. Only recent advances in hardware and just in time compiler technology¹⁰ allow interpreted code to perform almost as fast as native code.

The JVM and its instruction set called bytecode, have several interesting features usually not found in “real” CPUs:

- The JVM has no explicit registers, all instructions operate on the stack. An implicit stack pointer, a frame pointer and a program counter are manipulated in a controlled way as side effects of the various instructions.
- The JVM defines basic data types such as bytes (8-bit), shorts (16-bit), integers (32-bit), longs (64-bit), float (32-bit) and double (64-bit) as well as their encoding (big-endian for integer types, IEEE floating point format for float and double).
- The JVM has no memory address space model. The only way to access memory in the JVM is through references. References are created by a JVM instruction (**new**) that allocates an instance of a *class* that may contain *fields* which in turn may be accessed by JVM instructions (**getfield** and **putfield**). Arrays are handled in a similar way and every array access is range checked. Unused memory is automatically reclaimed by a *garbage collector* which is also part of the JVM.
- JVM operations are type checked, for example manipulating references with an integer instruction results in an exception.
- The JVM supports multi-threading and synchronization. Every instance of a class allocated through **new** act as a monitor. Monitors may be entered by the **monitorenter** instruction and left through **monitorexit**.
- Since there is no address space model, code is called by invoking a *method* defined in a *class* as bytecode instead of jumping to a memory address. Inside a method only relative jumps are allowed. Jumping before or past a method is intercepted by the JVM and results in a runtime exception.

The Java effort did not stop with the CPU and memory abstraction by the JVM. Java itself is an object oriented programming language designed for the JVM. But many compilers for other languages than Java exist that also generate bytecode, executable by the JVM. There are also

¹⁰Just in time compilation defers part of the compilation process (usually the compiler back-end) until runtime. In the case of Java, byte-code is *compiled* to native code of the underlying architecture upon execution instead of being *interpreted* by the JVM. A similar technique implemented in hardware and called *code-morphing* is used by Transmeta’s Crusoe CPU.

native compiler available for Java that allow execution of programs written in Java without a JVM.

As can be seen from the previous section, abstracting the CPU and memory through a language is only one part of the story; platform heterogeneity is the other part.

Java addresses platform heterogeneity on behalf of several classes for input/output, process management, and GUI. It is interesting to note that the first GUI abstraction, the abstract windowing toolkit (AWT) used a layered approach (see figure 1.2) whereas the new version (Swing) uses an emulation approach.

Java is the first successful realization of truly portable software and clearly shows the path for future development. For example the new C# language [36] recently introduced by Microsoft for the .NET [56] environment borrows many aspects from Java.

1.3.4 .NET

The key contribution of .NET [104, 149] to the heterogeneity problem is the introduction of a language independent object interface that allows from various languages¹¹ the mutual use of objects. This infrastructure called common language infrastructure (CLI) consists of the common language runtime (CLR) and an object standard for all programming languages using it. This standard consists of the common language specification (CLS) together with the common type system (CTS). As with Java bytecode, compilers for the .NET environment generate intermediate code called the Microsoft intermediate language (MSIL), which is translated to machine code upon loading. This is similar to Java's just in time compilation with the important difference that compilation is not an *option* but mandatory in .NET. This means that .NET code always executes as native code and is never suffers from the performance penalty introduced by an interpreter. On top the CLR, a common class library provides a shared set of services across all .NET languages and simplifies the integration of legacy DCOM/COM+ components and database access.

1.3.5 Guile

The former approaches are hiding heterogeneity by the definition of a platform independent, homogeneous interface. In doing so, they are on the one hand opening the number of potential platforms but on the other hand also limiting the developer's language choice by imposing a development environment that includes a specific language. Only .NET defines a language independent interface but this interface is limited to object oriented languages only.

The diversity of programming languages found today is mainly due to the fact that certain problems are more easily expressed in one paradigm than in another. So one may ask, why using one language for everything when there are specialized languages available for specific things?

The GNU extension library (Guile [20]) aims at solving the inter-language communication problem. The basic idea is to split an application in sub-problems that are then individually solved in the language best suited. Guile also helps in the case where certain functionality is available, say in a C library, but has to be accessed from another language, acting as cross-language.

¹¹Currently these are the languages supported by Microsoft's Visual Studio .NET: C++, C#, and Visual Basic .NET [24].

So what is Guile then? The short answer is that Guile is a Scheme¹² [1] interpreter distributed as a library. The power of Guile comes from its foreign language interface. It uses the C-linkage convention as a pivot point and defines how data in different language is accessed from Guile and how control is passed from Guile to other environments and back. This means for example that one can simply add (Scheme) scripting functionality to any existing program by linking with Guile. It also means that Guile can be used to bind together several applications and acting as the “smart glue” between those.

Guile deals with heterogeneity in two ways. On the one hand it explicitly addresses a heterogeneous environment where several incompatible languages exist. On the other hand it hides exactly the same heterogeneity by providing a homogeneous access to these environments by its Scheme interface.

Because Guile uses the platform dependent C-linkage convention, Guile applications have to run on the platform they were designed for. The next section will present some cross-platform solutions that may run on multiple platforms and even span multiple hosts over a network.

1.4 Middleware

1.4.1 Introduction

When network computing became more and more common, the demand for services beyond e-mail, file transfer, and printing increased. This demand was satisfied by a large number of proprietary client / server applications which reproduced the same incompatibility problem as found in the early days of computing.

In order to decouple proprietary clients and servers, a new kind of software appeared: middleware. The task of the middleware is to bind in a standardized way client and server and to provide structured message exchange, thereby replacing the proprietary protocols. The evolution followed the main stream in computer languages from message passing (assembler instructions) to remote procedure calls (procedural programming) to distributed object systems (object oriented programming).

The following three sections look at two prominent examples: CORBA and SOAP as well as at the WOS research project. CORBA takes a full-fledged approach realizing everything for everybody whereas SOAP follows more the “keep it small and simple” way. WOS is somewhere in between offering interesting features of its own.

1.4.2 CORBA

The object management group (OMG), founded 1989 as a non-profit organization aiming to achieve interoperability on all levels for an open object market. The common object request broker architecture (CORBA) is at the heart of the OMG effort. In some sense CORBA is built on top of the same idea as Guile: controlled interaction of heterogeneous environments. But CORBA pushed interaction ways further. Instead of just using linkage conventions, CORBA defines a rich distributed object model that allows network-wide interoperation of objects.

CORBA is language and platform independent and its layered architecture (see figure 1.3) helps porting it to new environments. The language independence comes through an

¹²Schem is a LISP like language with a few imperative extensions as well as some syntax modification.

interface description language (IDL). CORBA interfaces written in IDL are compiled for the target environment, which is using a CORBA service as a client or implementing one as a server. So-called CORBA language binding exists for many languages; the most prominent are C++, Java, and Smalltalk. Besides defining the transformation from IDL to the target environment, the language bindings also define the interaction of the target language with the object request broker (ORB). The main task of the ORB is to route the method invocation from the client side to the appropriate server object implementing it and to handle the object references involved. The ORB totally hides implementation and location details of the server. The ORB does not need to be implemented as a single component but provides a common interface for client and server. Different ORBs may be combined and communicate via the Internet inter-ORB protocol (IIOP) for example.

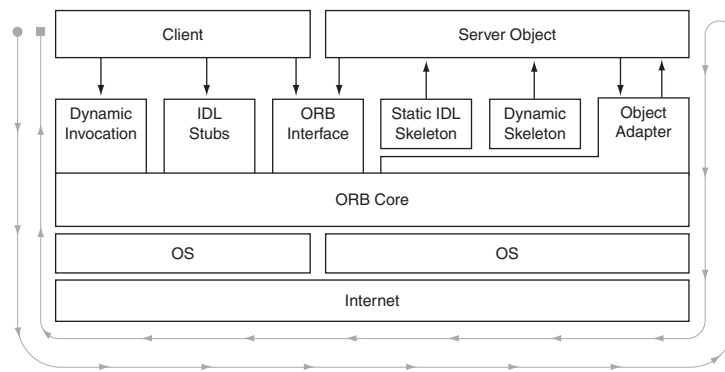


Figure 1.3: The parts involved in a CORBA invocation.

Figure 1.3 shows a client and a server object in a CORBA environment. To make a request, the client can either use the dynamic invocation interface or an IDL stub. The stub is individually generated for every CORBA interface whereas the dynamic invocation interface is uniform for all interfaces. The server object receives a request as an up-call either through the generated skeleton or through a dynamic skeleton.

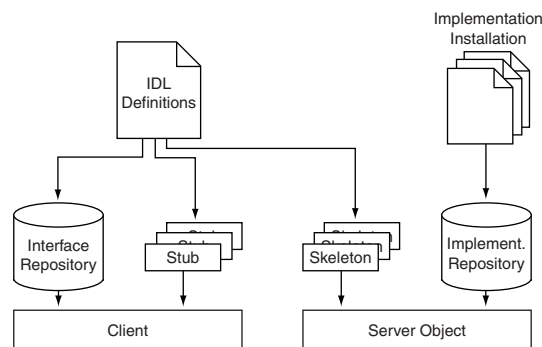


Figure 1.4: Interface and implementation Repositories

Figure 1.4 shows the relation between the interface repository that stores interface information for runtime introspection, used by the dynamic invocation interface, and the generated stubs and skeletons. Information on how to find and activate a server object is located in the implementation repository.

CORBA is basically a “wiring” standard, it regulates the interaction and representation of objects between heterogeneous environments. In order to provide higher-level abstractions beyond objects, the OMG defines the object management architecture (OMA) that consists of CORBA interfaces for services and facilities as well as application objects for horizontal and vertical markets. Some of these basic CORBA services [132] are:

- **Naming service:** All CORBA objects have a unique ID. The naming service allows assigning names to object IDs in a hierarchy.
- **Security service:** A pervasive service that provides certificate based security throughout CORBA.
- **Trader service:** A “marketplace” where providers may offer services which can in turn be found by clients through queries.
- **Object transaction service:** Supports nested transactions in a CORBA environment and allows integration for non-CORBA transactions.
- **Change management service:** This service supports version tracking for interfaces and persistent objects.
- **Concurrency service:** This service supports acquisition and release of locks on resources. Several lock modes are offered and cooperation with the object transaction service is integrated.
- **Event notification service:** The event notification service allows the creation of immutable event objects that flow from suppliers to consumers. Events flow through channels which can be filtered and support a “push” and a “pull” model of event notification.
- **Externalization service:** Serializing a graph of objects to a stream is handled by the externalization service. Objects have to implement a special interface to be externalized.
- **Licensing service:** To control legal use of third party objects, the licensing service allows an object to check if its usage complies with a given licensing policy.
- **Life cycle service:** Object graphs can be created, copied, moved, and destroyed by the life cycle service.
- **Object collection service:** A set of abstract data types such as bags, sets, queues, lists, and trees.
- **Object query service:** Similar to the trader service that locates servers, the object query service locates instances based on object attributes. The supported query languages are the object database management group’s object query language (OQL) and the standard query language (SQL) with object extension.
- **Persistent object service:** A persistent object survives the termination of the application that created it. Similar to the externalization service, the persistent object service provides an externalized form for objects but without the need to implement a special interface.

- **Properties service:** This service allows arbitrary properties to be associated with objects that implement a special interface.
- **Relationship service:** Usually objects are interrelated through references. The relationship service provides an associative model for interrelating objects similar to relational databases.
- **Time service:** System time in a distributed system is inherently inaccurate so the time service provides a reference time for time based actions in a CORBA environment.

In addition to the CORBA services the OMA also defined CORBA facilities, which are frameworks for component integration.

- **User interface:** Printing, email, compound documents, automation and scripting, and object linking.
- **Information management:** Structured storage, universal data transfer, and meta-data.
- **System management:** Instrumentation, monitoring, and logging.
- **Task management:** Workflow management, rules, and agents.

Finally, application objects serve a specific application domain and are not part of the OMG effort but outsourced to specialized organizations that then report to the OMG by proposing standards. A prominent example is the business object management special interest group (BOMSIG).

Although CORBA is very stable and supported by many vendors, implementation of CORBA services and CORBA facilities is still in the beginning. CORBA services and CORBA facilities are optional and every vendor provides its own subset. Despite that, it can be said that at least for object oriented environments, CORBA achieves a high degree of inter-working and platform independence.

1.4.3 SOAP

In principle, CORBA can be used over the Internet, for example Netscape's Communicator includes Visigenics's Visibroker ORB and various Java based ORBs for use within applets are available. The reason why CORBA is seldom used over the Internet is that firewalls usually prohibit protocols other than HTTP.

This was the main reason why the simple object access protocol (SOAP) was developed. Another reason behind SOAP is the current state of service access on the Web. The Web provides a rich set of services but they are mostly designed for human users, rendering programmed or automatic access very difficult.

SOAP provides a simple and lightweight mechanism for exchanging structured and typed information in a distributed environment using XML. SOAP consists of three parts:

- The SOAP envelope defines *what* is in a message, *who* should handle it and *whether* the parts are optional or mandatory.
- The SOAP encoding rules define how application defined data types are serialized.
- The SOAP remote procedure call (RPC) representation is a convention to represent remote procedure calls and responses.

Although SOAP can be used over various transports, currently it is only used over HTTP. In order to remain simple, SOAP does not define advanced features like object references, activation of persistent objects, and garbage collection for example.

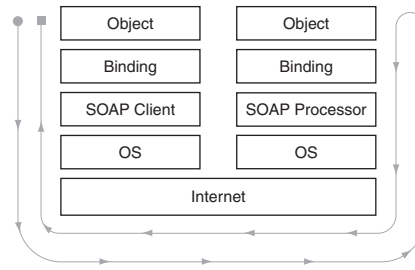


Figure 1.5: Parts involved in a SOAP message exchange

SOAP messages are one-way messages but often used in a request-response pattern such as in a HTTP POST request. Figure 1.5 illustrated the parties involved in the processing of a SOAP request. A client, for example written in Visual Basic calls the SOAP client through a language binding. The language binding defines the way in which a SOAP client is accessed within a specific environment such as Visual Basic or Java. The SOAP client then handles all the communication with the SOAP processor on the server side. The SOAP processor is usually integrated into a Web server environment. The client request is then passed through the language binding to the server object implementing the service. Many variations of this scheme are possible. SOAP services may be called without a binding by sending SOAP messages directly; a SOAP processor may implement the service without delegating it to an object and so on.

SOAP allows forwarding of messages along a message path and requires a SOAP processor to understand the exchange pattern being used, the role of the recipient, the data encoding, and the correct semantics for correct processing. Furthermore, SOAP provides versioning through an XML namespace.

In the same way CORBA is often used in conjunction with CORBAServices, SOAP is used together with other services like universal directory and discovery (UDDI) and the Web service description language (WSDL) to form Web service platforms such as Sun's ONE (see 2.5.6) and Microsoft's .NET (see 2.5.6).

As in CORBA, SOAP hides heterogeneity behind a common protocol but is much simpler. SOAP is only a message protocol and does not define a distributed object model. Some flexibility is gained by allowing optional parts to be skipped by a SOAP processor. SOAP by itself provides only the “wiring” level, service discovery and description is handled by services such as UDDI and WSDL. Section 2.5.6 takes a closer look at this relation.

1.4.4 WOS

The Web operating system (WOS) approach to global computing aims to provide solutions for global ubiquitous computing and to develop service mechanisms that meet the requirements of the net-centric view of services and processes. To account for the dynamic nature of the Internet, generalized software configuration techniques, based on a demand driven technique called education are developed for the WOS. The WOS is not an operating system in the classical sense which manages (hardware) resources but a meta-operating system running on

top of existing operating systems such as Unix and Windows.

Because of the heterogeneous and dynamic nature of the Web, the WOS has to support dynamic changes and multiple simultaneous versions [98]. The versioning problem is two-fold: *nodes* in the WOS may run different versions of the operating system kernel and *resources* may be available in different versions.

As a result the WOS uses a two layer protocol [7]. The WOS request protocol (WOSRP) is used for version queries to ensure version compatibility of WOS nodes as well as the discovery of WOS nodes using UDP broadcasts. If version compatibility is found, the WOS protocol (WOSP) is used for the actual data exchange. WOSP handles the following interactions between WOS nodes:

- Query commands used by a WOS client to interrogate another WOS warehouse.
- Setup commands to change the execution parameters of a WOS node.
- Execution commands allowing a WOS client to use resources from another node.

Communication inside the WOS is connectionless and usually a request travels through different WOS nodes until one has the right set of resources available to process the request and send back the result to the requester.

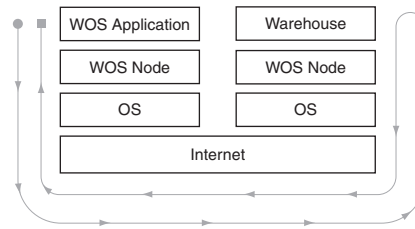


Figure 1.6: Parts involved in a WOS request / response.

Figure 1.6 illustrates the parties involved in a WOS communication. An application issuing a WOS request contacts a known WOS node, for example the WOS kernel running on the local machine, which then will pass on the request to a WOS node serving the request. The serving WOS node then satisfies the request using its warehouse. The warehouse acts as a cache for previous responses as well as storage and directory for local and remote resources provided by the WOS node.

The two WOS protocols use lower level protocol such as UDP and TCP. The WOS takes a similar approach to the heterogeneity problem as SOAP: by using common protocols heterogeneous entities are able to cooperate. WOS goes further than SOAP in the sense that it also includes in the base protocols automatic discovery and versioning. Adapters, shown for example by the implementation of a CORBA-to-WOS gateway [93], may translate the WOS protocol to other protocols. The WOS is further discussed in section 3.2.3.

1.5 Communication

1.5.1 Introduction

Communication is a broad field and involves many aspects. Two entities that like to talk to each other have to speak the same language on several levels. It starts at the physical layer

where both parties have to use the same signaling and modulation. It continuous in how bit strings are encoded on these physical links and how higher level data is encoded (see also 1.2). In the case of an end-to-end link this might be sufficient for rudimentary communication. If the endpoints are part of a larger network, questions arise how data from one endpoint reliably reach the other endpoint. The open system interconnect (OSI) model (see table 1.1) has served in the past as a reference in designing network systems.

Application	Service provided on top of a protocol
Presentation	Encoding of data (syntax)
Session	Organization of larger sequences
Transport	Quality and nature of delivery
Network	Addressing and routing
Data Link	Logical organization of transmitted bits
Physical	Electrical and physical issues

Table 1.1: The open system interconnect (OSI) model.

But simple data exchange is often not sufficient. Higher-level services require specific protocols and semantics. A few examples were presented in the previous sections.

With the addition of human users as part of the communication, client heterogeneity increases. Making services available for both human users and computers becomes a challenging task to be solved on the Web. The problems are the very dissimilar requirements of the two. Presentation for example is important for the human user but usually makes the situation worse for automatic clients. Protocols like SOAP and data representation like XML together with XSL try to solve the problem by offering appropriate transformations for human users and automatic clients. However, they provide little help for automatic *understanding* of the service provided, something human users take for granted and what is easily achieved by a good Web design.

To show the heterogeneity problems involved with communication, the following sections present two contrasting examples: HTTP together HTML that is tailored toward user interaction by means of a Web browser and a Web server and Bluetooth that provides automatic device interaction with little or no user intervention.

1.5.2 HTTP and HTML

The hypertext transfer protocol (HTTP) together with the hypertext markup language (HTML) is an application level protocol for distributed collaborative hyper media systems. It is a simple data exchange protocol with a few defined methods for resource access on remote servers. A resource or a part of it is requested by a uniform resource locator (URL). The data exchanged may be negotiated and is typed by the multipurpose Internet mail extension (MIME [22]).

Figure 1.7 shows the parties involved in a HTTP interaction. The client, usually a Web browser, requests or sends information to a HTTP server that responds with the requested resource or performs an operation. The basic operations of HTTP are:

- The **GET** method that retrieves whatever information is identified by the URL given in the request. Usually a **GET** request returns a document identified by the URL from the Web server's filesystem. But Web servers also allow on the fly generation of information

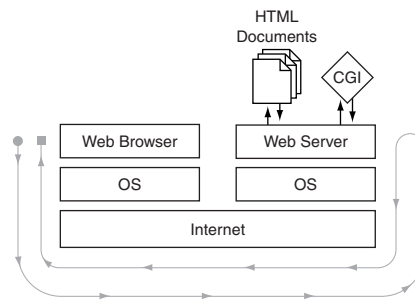


Figure 1.7: Parts involved in a HTTP request

through the common gateway interface (CGI) where the output of a process is returned instead of a document.

- The **PUT** method can be seen as the complement to the **GET** method. It requests the server to store or update the data supplied by the client under the URL given in the **PUT** request.
- The **DELETE** method asks the server to remove the resource given by the URL in the **DELETE** request. **GET**, **PUT** and **DELETE** provide the basic resource access of HTTP similar to a fileserver.
- The **POST** method asks the server to accept the data supplied by the client to be accepted as a subordinate of the resource identified by the URL given in the **POST** request. **POST** may be used for tasks such as annotation of existing resources, posting a message to a group of articles like bulletin boards, mailing lists or newsgroup, or to add records to a database. The actual operation of the **POST** method usually depends on the server implementation and the URL in question.

HTTP is mainly used to serve HTML pages but it is found in other applications as well like SOAP (see 1.4.3) that uses XML messages to access objects. Protocol extensions such as WebDAV (Web-based Distributed Authoring and Versioning [182, 183]) allow users to collaboratively edit and manage files on remote Web servers.

HTML is a data representation format for hypertext documents rendered on screen by a browser. Although HTML's success is unquestioned and demonstrated by the billions of HTML pages, it has nevertheless several shortcomings:

- HTML is a SGML application at the base but allows several shortcuts, making HTML more a recommendation than an implemented standard. This makes rendering HTML difficult for browsers and complicates automatic processing of HTML.
- HTML mixes form and content with the result that separating both is nearly impossible. A separation is often required because automatic clients usually deal with content whereas formatting helps human users to understand content.
- Because HTML is rendered entirely on the client, HTML puts considerable load on it. HTTP in combination with HTML is not a well-balanced system. A fully functional HTTP server may be written in a few hundreds lines of code but writing a Web browser

compatible with all Web pages on the World Wide Web is a challenging task even for the leading browser companies.

- Dynamic HTML (DHTML) and cascading style sheets (CSS) increase the load on the client side even more and introduce new incompatibilities.

HTML is widely used despite all these problems and supported on many platforms. Several processes are under the way by standardization bodies to correct the problems of HTML but the Web will have to continue to live with the HTML pages existing today, which are unlikely to disappear soon.

The HTML example shows the importance of content beyond protocols regulating interaction and message structure. It is not sufficient to have agreed communication protocols such as HTTP where interpretation of content is not clear but required for successful end-to-end communication. Standardization failed in the case of HTML mostly because of its latency to respond to innovation. It is therefore questionable if standardization is the right way to follow in a highly dynamic and innovative environment such as the Web.

1.5.3 Bluetooth

Bluetooth [66, 67] describes the protocol of a short-range (10 meter) frequency-hopping radio link between devices. Its primary purpose was to replace cables between low-power devices such as mice, keyboards, printers, headsets, etc. but many new applications were defined since. The Bluetooth standard is split in two parts: the *specification* describes how the technology works and the *profiles* describe how the technology is used.

It is interesting to note that the Bluetooth standard goes beyond the simple wiring level, which is where many similar network standards stop. In fact, Bluetooth defines an additional (service) layer in the OSI model between the application and the presentation layer that defines how services are implemented and accessed. These services are called *profiles* and define the interface for applications as well as between devices.

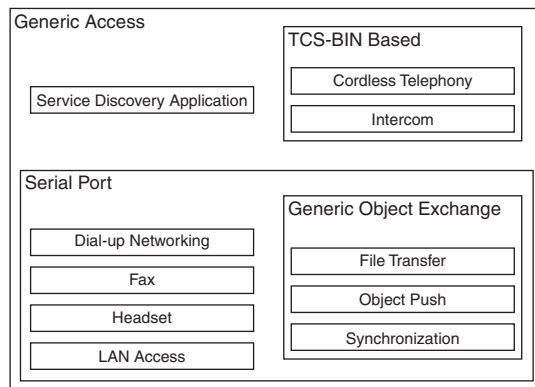


Figure 1.8: Bluetooth Profile Structure

Figure 1.8 depicts the structure of the Bluetooth profiles defined so far.

- The generic access profile defines the generic procedures related to discovery of Bluetooth devices and link management aspects of connecting to Bluetooth devices. It is the core on which all other profiles are based.

- The service discovery application profile defines the features and procedures for an application in a Bluetooth device to discover services registered in other Bluetooth devices and retrieves any desired available information pertinent to these services.
- The cordless telephony profile defines the features and procedures that are required for interoperability between different units active in the 3-in-1 phone¹³ use case. This profile also shows how it can be applied generally for wireless telephony in a residential or small office environment.
- The intercom profile defines the requirements for Bluetooth devices necessary for the support of the intercom functionality within the 3-in-1-phone use case.
- The serial port profile defines the requirements for Bluetooth devices necessary for setting up emulated serial cable connections between two peer devices.
- The dial-up networking profile defines the requirements that shall be used by devices like modems, acting as an Internet bridge.
- The fax profile defines the requirements for Bluetooth devices to be used as a wireless fax modem for sending and receiving fax messages.
- The headset profile defines the requirements for a wireless headset that can be combined with cellular phones or audio equipment for example.
- The LAN access profile defines how Bluetooth enabled devices can access the services of a LAN using the point-to-point protocol (PPP). Also, this profile shows how the same PPP mechanisms are used to form a network consisting of two Bluetooth-enabled devices.
- The generic object exchange profile defines the protocols and procedures for Bluetooth devices necessary for the support of object exchange such as file transfer, object push, or synchronization.
- The file transfer profile defines the requirements for applications providing the file transfer usage model. Typical scenarios involve a Bluetooth device browsing and manipulating objects on another Bluetooth device or transferring objects between devices.
- The object push profile defines the requirements for applications providing the object push usage model. Typical scenarios covered by this profile involve the pushing and pulling of data objects between Bluetooth devices
- The synchronization profile defines the requirements for applications providing the synchronization usage model. Typical scenarios covered by this profile involving manual or automatic synchronization of personal information management (PIM) data when two Bluetooth devices come within range.

Bluetooth recognized the need to deal with heterogeneity on a higher-level than simple wiring, defining special applications called profiles. Still, extension beyond these profiles requires a revision of the standard, which may slow down innovation in a highly dynamic field like wireless piconets and home automation.

¹³A 3-in-1-phone is a handset that can be used as either a cellular-phone, a wireless phone, or an intercom

1.6 Hardware Abstraction

1.6.1 Introduction

Software usually lives longer than hardware, which changes very quickly due to technical advances. Hardware dependencies of software can be resolved in two ways: constrain the hardware (for example the instruction set architecture) or abstract the hardware and prohibit direct access to the software.

Currently both ways are taken simultaneously. Modification of the instruction set of existing architectures were always backward compatible and thus clearly constraint by the existing software. For example a modern Intel Pentium CPU is still able to run code written decades ago. On the abstraction side, operating systems provide more and more support for hardware devices through drivers making the software largely independent of the actual hardware. The main hardware features usually abstracted by the operating systems are: CPU, main memory, stable storage, and input/output.

The picture changes considerably when the focus moves to networked scenarios such as in ubiquitous computing where no central resource management and no abstraction authority exists, when global state can no longer be established, and the configuration is continuously changing.

The following sections will look at the approach taken by three prominent operating systems, namely Unix, Mac OS X, and Windows 2000. It is followed by a selection of problems arising in a dynamic and heterogeneous ubiquitous computing environment.

1.6.2 Unix and Mac OS X

Unix was designed from the beginning with portability in mind. This included on the one hand portability of Unix applications and on the other hand the portability of the operating systems itself. Portability is achieved on a source code level mainly thanks to using C as the implementation language. The Linux kernel for example was first developed for 32-bit x86-based PCs. These days it also runs on Compaq Alpha AXP, Sun SPARC and UltraSPARC, Motorola 68k, PowerPC, PowerPC64, ARM, Hitachi SuperH, IBM S/390, MIPS, HP PA-RISC, Intel IA-64, and DEC VAX. The main hardware abstractions of Unix are:

- **CPU:** Processes are the CPU abstraction used by Unix. Every Unix process has the illusion of owning the CPU with its own set of registers, address space etc.
- **Main memory:** The virtual memory systems together with the swap mechanism of Unix gives each process the illusion of a large area of private main memory.
- **Stable storage:** Stable storage is abstracted by block-drivers implementing a block-level access to a storage device and a file system which acts as the main storage interface for processes.
- **Input/output:** Unix device drivers handle input/output in either sequential mode (character driver) or random access (block driver). Every device may implement a set of additional input/output controls handling features not accessible by either sequential or random access. Unix device drivers are mapped into the file system and may be accessed by applications as files.

Originally, hardware on Unix machines was considered static (at least during runtime) and all required device drivers for a particular system were compiled into the kernel. With the introduction of buses that allow hardware changes at runtime such as PCI 2.0, PC Card, USB, or IEEE 1394¹⁴ (see 3.4.3) Unix kernels had to become adaptive.

Linux as one flavor of Unix uses so called kernel modules that can be loaded and linked at runtime into the kernel. Bus drivers watch buses for changes and load and unload appropriate modules automatically.

When Apple designed Mac OS X which is based on a Unix variant¹⁵ it had the choice between extending the Unix driver model for dynamic loadable/unloadable drivers, reusing the Mac OS 9¹⁶ driver model, or starting from scratch. Because both older models had their shortcomings, a new driver model collected in the I/O Kit was developed.

The I/O Kit is object oriented and defines several base classes for families of devices. Writing a device driver basically means implementing or overriding methods defined in these base classes. Device families currently supported by the I/O Kit include: audio, graphics, human interface devices, serial interfaces, storage, network, universal serial bus (USB), Apple desktop bus (ADB), small computer systems interface (SCSI), AT attachment (ATA)¹⁷ and ATA packet interface (ATAPI), FireWire (IEEE 1394), PC Card (PCMCIA), serial bus protocol 2 (SBP2), and peripheral component interconnect (PCI).

The I/O Kit has many modern features but only the aspects dealing with the hardware heterogeneity will be presented here. Instead of having a set of drivers statically included in the kernel, Mac OS X uses a three-phase matching process that narrows down a pool of candidate drivers for a specific device. This device-driver matching process makes use of matching directories defined in XML. Each matching directory describes a so-called personality of a driver, which specifies the kind of devices it supports. At boot time and at any time a bus driver (for example for the SCSI bus or USB) detects a change, the I/O Kit initiated the matching process that goes through the following four steps:

1. In the *class matching* step, the I/O Kit keeps only drivers of possible classes for the change in question. For example SCSI drivers have not to be considered if a USB device was attached.
2. In the *passive matching* step, the I/O Kit uses the driver's personality to further narrow down the set of possible drivers by looking at device class identifiers for example.
3. In the *active matching* step the driver has to possibility to talk to the device and return a score reflecting the driver's ability to handle the device.
4. The I/O Kit then chooses the driver with the highest score and loads it. If the driver fails for some reason, drivers with lower scores are tried.

Thanks to its modern design, the I/O Kit provides a powerful approach handling device heterogeneity. The object oriented base classes simplify driver development because common functionality is shared between drivers and does not have to be re-implemented. The innovative device-driver matching provides generality as well as specialization. Unknown devices

¹⁴The IEEE 1394 standard is also known as FireWire and iLink.

¹⁵Mac OS X is based on FreeBSD [102] on top of a Mach [19] micro-kernel.

¹⁶Mac OS 9 is the proprietary operating system for the Macintosh line of Apple Computer.

¹⁷ATA refers to AT-bus (IBM advanced technology (AT) computer system bus) attached storage devices also known as integrated device electronics (IDE) or EIDE devices.

may be handled by a generic driver for a class of devices but are superseded by specialized drivers when available thanks to the scoring mechanism.

1.6.3 Windows 2000

The predecessor of Windows 2000, Windows NT [171] was designed with portability in mind and was available for Intel x86, PowerPC, MIPS and Alpha. The set of supported instruction set architectures was reduced with every new version; with Windows 2000 formerly known as Windows NT 5.0 supporting Intel x86 only. Largely using C and C++ as the implementation language together with a layered architecture achieved the portability of the source code.

The basic idea behind the Windows NT hardware abstraction is a small, machine specific part called hardware abstraction layer (HAL) that defines basic access procedures for motherboard and chipset specific features. The HAL provides services such as device register access, bus-independent device addressing, interrupt handling, direct memory accesses (DMA) transfers, timer and real time clock (RTC) control, and multiprocessor synchronization. Higher-level hardware abstractions are built on top of the HAL, making for example drivers portable among different hardware platforms. The higher-level hardware abstractions of Windows 2000 are:

- **CPU:** Threads are the CPU abstraction used by Windows 2000, organized with other resources in a process. Every Windows 2000 thread has the illusion of owning the CPU with its own set of registers and stack, but sharing the address space with threads in the same process.
- **Main memory:** The virtual memory systems together with the swap mechanism give each process the illusion of a large area of private main memory.
- **Stable storage:** Stable storage is abstracted by drivers implementing a block-level access to a storage device and by file systems which are device drivers too.
- **Input/output:** A framework consisting of the I/O manager, the plug and play manager, the power manager, and the HAL handle input/output.

Figure 1.9 illustrates the various parts involved in the Windows 2000 hardware management.

- The user level part handles mainly device configuration and driver installation. For example if Windows 2000 detects an unknown device, the “Setup” application prompts the user to install the appropriate drivers.
- The plug-and-play manager handles hardware configuration changes and dynamically loads and unloads drivers for physical devices.
- The I/O manager handles device independent access to physical devices by routing calls to the appropriate driver. The I/O manager handles the filesystems as device drivers as well. Device drivers may be stacked to perform filtering. A generic filter for a pointing device driver may for example scale the cursor position information thereby implementing mouse cursor acceleration.
- The power manager is responsible for system-wide power control, turning off idle devices, sending the system in suspend state, etc.

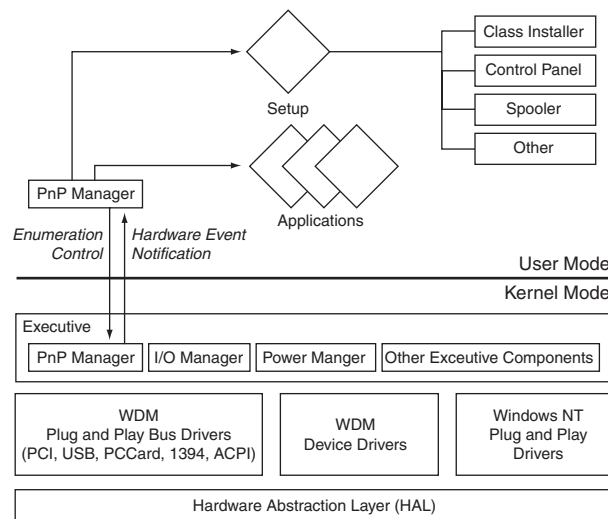


Figure 1.9: Windows 2000 hardware abstraction layers

Due to its history, Windows 2000 takes a more conservative approach to hardware abstraction than Mac OS X. Windows 2000 for example supports still the Windows NT driver model and plug-and-play was an addition instead of a generalized replacement. The HAL presents an interesting concept, making drivers portable among platforms but with support for Intel x86 only this part lost its importance except for a few specialized server systems with unique hardware. Windows offers over 100 different API's for handling devices, reflecting a good deal of the hardware heterogeneity as API heterogeneity. Drivers usually do not implement those API directly but are based on the Windows driver model (WDM) which is the base for the device specific APIs. The WDM describes how driver calls in the form of I/O request packets (IRPs) are handled, how plug-and-play support is integrated, how power-management is supported and various other issues related to configuration, reentrancy, and portability.

In Unix as well as in Windows 2000, a centralized instance has control over all resources and allows allocation decisions to be based on a global system state. The next section will present an environment where no such central instance exists and global state can no longer be established.

1.6.4 Ubiquitous Computing

Research in ubiquitous computing is towards the development of an application environment able to deal with the mobility and interactions of both users and devices. The vision of ubiquitous computing [110, 111, 112, 130, 181] relies on the presence of environments enriched by computers embedded in everyday objects (blackboards, table, walls, etc.) and by sensors able to acquire information from the context.

Ubiquitous computing devices have a tight coupling between hardware and software, on the one hand because of the resource limitations of the device itself and on the other hand due to resource dependencies of the software on specific hardware features such as pen-input or temperature measurement. For software running on ubiquitous computing devices, the abstraction requirements are twofold:

- The intra-device abstraction is similar to today's computers, solved by small-operating

system kernels. Additional constraints such as screen size, memory size, CPU-power, or power consumption may limit the application but basically intra-device abstraction has to deal with a static environment known at build time.

- The inter-device abstraction is a much bigger problem. Beside the decision of which services should be accessible from the outside, different hardware and software interfaces have to be considered as well as “smart” cooperation of devices. Additional difficulties arise because the cooperating devices may be unknown or untrusted, there are no standards to follow due to innovation, there is no global accessible state, and there is usually nothing like a “system administrator” keeping a birds eye view of the system.

On the user level, using an application and doing useful work should not require to be an expert in the organization of the software infrastructure. As much as possible the software should be self-managing and self-repairing in the face of simple transient faults.

For developers, hardware and protocol heterogeneity must be hidden from ubiquitous computing applications as good as possible. Thanks to a common design methodology the application programmer should not have to understand the entire software and hardware infrastructure, which may not be known in advance.

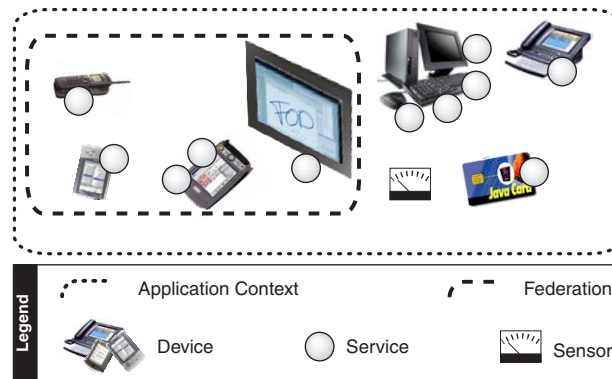


Figure 1.10: Various ubiquitous computing devices in an application context.

Ubiquitous computing applications going beyond the services offered by a single device depend on resources provided by a context, which contains other ubiquitous computing devices and services. Figure 1.10 shows various ubiquitous computing devices cooperating in a federation and providing different services. An application running in such an environment will be a collection of services linked at runtime. These links may change during execution especially if the application is composed of services running on mobile devices which may leave the application context or devices that are simply turned off independently.

Finding and accessing resources and services becomes a major problem because the application context may contain devices and features not known at the time the application was developed. A common approach is standardizing interfaces for resource access and selection. Several architectures rely on directory services combined with attribute based query services for resource selection and access protocols. Unfortunately, standardization takes a long time, usually involves several compromises and the resulting specifications are seldom open enough to allow innovation. Even agreeing on basic services such as printing is a lengthy and complicated process as recently shown by the Jini print services [80] for example. Because of that, it may be claimed that standardization will be unable to keep pace with the highly dynamic

evolution of ubiquitous computing devices as well as the upcoming Web Services. Similar problems were encountered in the past. Like the situation found with HTML today, which is nearly impossible to correct. Even implementation of standards such as SOAP still differ [151], complicating the realization of the idea of homogenous Web services. In a ubiquitous computing environment, proven communication protocols such as CORBA, Java RMI, or SOAP may be used for the “wiring” but for system composition and hardware abstraction a radically different approach is needed.

Instead of enforcing a standardized view for the application, a ubiquitous computing middleware should decouple the high-level concepts (abstractions) from the instances implemented by a context [108, 153]. The concept “nearest printer” for instance may be used no matter how a context supplies the corresponding implementation. This means that an application expresses its resource requirements in terms of its concepts instead of addressing specific resources directly for example by an URL. The application concepts are instantiated by the middleware in function of the context and may change with time and location. For example when a users walks around, the middleware will keep track of the concept “nearest printer”, switching from one physical printer to another as needed and without user intervention. Section 5.4 presents an implementation of these ideas.

1.7 Summary

From this chapter it may be concluded that heterogeneity is handled quite well. This is the case on the operating system level where even plug and play devices are often properly detected and configured with minor or no user intervention. But all the models proposed by the major operating systems limit the space for innovation. For example [5] states: “Although the I/O Kit attempts to represent the hierarchy and dynamic relationships among hardware devices and services in a Mac OS X system, some things are difficult to abstract. It is in these gray areas of abstraction, such as when layering violation occur, where the driver writers are more on their own.”

New devices not fitting into the boundaries set by the operating system have no proper way to bring themselves into the system. All these techniques have major problems when no central control or global state is available such as for example in a ubiquitous computing environment.

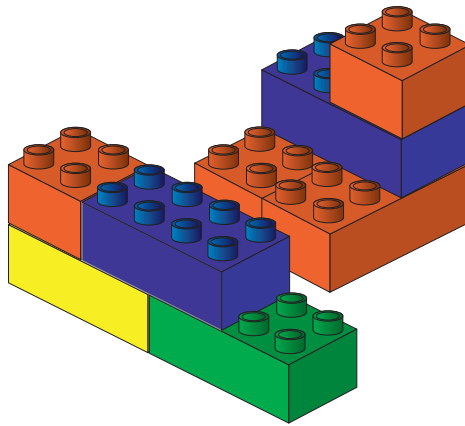
The situation is better on the language level thanks to the long experience and a quite stable hardware base (instruction set architectures do not change every few months but are stable over decades). Cross platform libraries provide enough abstraction so that portable programs for a wide range of platforms can be written. But source-code portability is not always the desired abstraction. Not every user knows how to use a software development environment and not every vendor loves to give his source code away. Java attacks the problem a layer below by defining a virtual machine, which allows applications to be distributed in binary form. Thanks to a rich class library most of the platform specifics are hidden resulting in one of the first successful cross-platform environments.

In the area of distributed computing powerful standards exists but mainly for the “wiring”. What is required in the context of distributed computing as well as in the similar context of ubiquitous computing is software that does not rely on predefined, standardized interfaces but is self-adapting. In order to make software self-adapting, semantics still scattered throughout tons of only human readable specification-documents or, even worse, in the head of a few

developers has to be moved into the system itself. Semantic based lookup and configuration will enable a plethora of new applications and simplify areas such as Web and ubiquitous computing.

Chapter 2

The Static Configuration Problem



2.1 Introduction

The previous chapter presented different levels where heterogeneity appears and examples how heterogeneity is handled by software. The main techniques used are abstraction and layering. Heterogeneity is hidden behind a common and homogenous interface defined for each layer. By using interfaces, higher layers can now deal with the abstract entities defined by the lower layers instead of the concrete heterogeneous entities below. Although the benefits of this technique are great, they come at a cost. One is decreased efficiency, since every layer adds some translation and management overhead. Another is configuration, because various dependencies between abstract and concrete entities are introduced.

This chapter looks only at the configuration problem and ignores the efficiency aspect. In the beginning of computing configuration was not a problem. There was one piece of hardware and one monolithic piece of software seen as one unit solving a task. The configuration problem started with the first hardware abstractions. Various libraries had to be loaded first into the machine before the actual program could be executed, resolving the main program's dependencies. The number of dependencies increased with higher-level languages. Not only had the right libraries to be present during run time but also the right compiler during development, with the compiler having dependencies of its own that had to be resolved.

In the following, software dependencies started to grow and finally outgrew hardware

dependencies by orders of magnitude. Whereas hardware composition problems usually could be handled physically, for example by using appropriate plugs and slots, no similar techniques were available for software.

Development and evolution of software systems became more and more complex. This lead, together with the lack of software interface standards, to the well known “software crisis” [128]. In response, development systems started to support the software developers in maintaining the dependencies and ensuring compatibility. But this support is still small and often requires manual maintenance by developers. This is mainly due to the level of abstractions these systems have to deal with. Most tools use flat files as abstraction, which provide little insight in the dependencies and compatibility issues. Only compilers and libraries usually have enough information to check various dependencies statically. Although recent languages such as Java that have typing systems that allow a lot of errors to be ruled out statically, they seldom address versioning and interface semantics for example. This means that as long as interfaces match, code can be bound independent of versions and semantics of caller and callee.

Especially in an object-oriented environment where dependencies are manifold due to inheritance, method overriding, and complex reference graphs, support beyond interface conformance is needed. White box reuse¹ as it is still common today permits developers to rule out most of these problems by introspection but is not considered “good style” because dependencies do not stop at interface level but go deep into implementation details of the reused code.

In distributed environments which are already in wide use and whose number is likely to increase in the future, white box reuse will be the exception and black box reuse will be typical. To make black box reuse practical, documentation tries to compensate the lack of interface expressiveness. Although documentation helps a number of developers to design systems and reuse code in a compatible way, documentation may still be ambiguous and not explain every detail necessary to resolve dependencies in question. As soon as system composition is no longer done by engineers that are able to understand the documentation but by the software systems itself, maybe marginally supported by an end-user, new ways of interface description have to be found.

The static configuration problem can thus be informally stated as:

Definition 2 *Given dependent entities in a computer system providing services, the static configuration problem is to find a configuration of these entities that allows correct functioning of the system as a whole.*

The following sections look in a top-down manner at the problem, starting with software configuration management, a recognized engineering discipline today. Architecture description languages are then discussed that aim at capturing the large-scale structure of software systems. A few exemplary tools and languages in widespread use underline development support available today. Component systems that inherently have to deal with composition and configuration are addressed at the end.

¹“White box reuse” in contrast to “black box reuses” refers to code reuse where the full source code is available. “Black box reuse” means no introspection, only the interface specifications are available. Various shades of gray reuse exist today.

2.2 Software Configuration Management

Software configuration management (SCM) is a supporting part of the realization of a software project. Heterogeneity, evolution of hardware, operating environment, and user requirements cause a software product to be modified many times during its lifetime. SCM is concerned with the identification, and organization of changes as well as controlling the way changes are applied in a software project. As the product evolves, changes often lead to families of related versions, several of which may need to be maintained in parallel. SCM spans the entire software life cycle from initial design through implementation, deployment, and maintenance. The importance of SCM usually grows with the size of the software projects. The number of dependencies increases often faster as the size of a software system, making manual dependency tracking very expensive.

SCM emerged as an engineering discipline for the control of the evolution of complex software systems about 20 years ago; soon after the “software crisis” [128] was identified. It was recognized that software engineering not only has to cover classical disciplines such as algorithms and data structures but also architecture, development, and system evolution. SCM as a discipline finds its implementation often in SCM systems and tools that provide services to control the evolution of software systems. The following sections will address the main areas where software projects receive SCM support.

2.2.1 Component Repository and Version Control

Large software systems consist of many different software components in different versions that have to be safely stored. Version control systems appeared as the first SCM support tools (see also 2.4.4) based on a simple idea: Each time a file is changed a *revision* is created, a file is thus a linear sequence of revisions. New lines of change can be created from any revision, resulting in a *revision tree* where each new line represents a branch.

The main features offered by these systems are a revision history, multi user management and merging. The revision history records for each file *who* changed *what*. Multi user management prevents changes from overwriting each other. The two main techniques used are long transactions where a user acquires an exclusive lock for a file (check-out), which is released after the change (check-in). The other technique is optimistic check-in where every user has a local copy of the project and version conflicts are resolved at the time the local copy is synchronized with the component repository (commit). The versions from the various users are automatically merged in the repository but may create conflicts requiring user intervention.

Although component repositories are often implemented as databases, the abstraction they deal with is a file enriched by a few attributes. In order to maintain compatibility with existing development tools like compilers and editors, files are also used for the external representation, limiting the control of these systems.

Because SCM is interested in the evolution of entire systems, file level version control is not sufficient. A valid software product normally consists of a set of files forming a valid configuration, so configuration control is required too. Support for configuration control is rare; questions like what the configuration is, how to build it and how to prove properties of it are seldom answered in an automatic way. Additionally, it may be interesting why something is a revision of something else, in particular what both have in common and where the differences are. Current systems only reply with differences of lines of code, letting the

user infer the meaning of the change.

2.2.2 Development support

An advantage of knowing the dependencies between software components is that this knowledge cannot only be used during runtime for proper software configuration but also at build time to optimize the build process. The **make** [154] tool (see also 2.4.2) for example uses make-files as a knowledge base of software item dependencies. These dependencies are then used by **make** to dynamically generate a build plan starting from non-existent items and file modification dates. Despite its success, **make** is difficult to use and inadequate in many respects. But all attempts to substantially improve it have failed so far.

Because files are too small in scope as representation for software configuration items, SCM tools offer workspace support. Workspaces are separated areas where programmers can work and usually semi-automatically interact with the component repository. The SCM system is responsible for providing the right configuration, letting the user work independently, and saving changes automatically when a user has finished.

Having independent users updating a common repository also means that conflicts have to be resolved at synchronization time. SCM systems provide cooperation support insofar that it can be controlled who can change which attribute of which component. Additionally, “smart” automatic file merge is offered.

2.2.3 Process support

As stated at the beginning, SCM spans the whole software life cycle, so there is an interest to model this life cycle in SCM systems as well. One role of a SCM system is therefore to model the software development process and to help reality to conform to this model. Two techniques are found today: state transition diagrams and activity centered modeling.

A state transition diagram describes the legal succession of states a product can go through and therefore describes the legal evolution of a system. Activity centered modeling has activities as the central points and the model describes the data and control flow between these activities [118]. Experience shows that activity centered modeling is better suited for modeling the large scale aspects of the process whereas state transition diagrams are very useful for fine grained process descriptions [44].

Besides improving the SCM tools in use today and enhancing cooperative work support for example through web integration, much work is required towards higher abstractions. Current systems mostly deal with files and lines of code, lacking higher level understanding of process and architectural aspects. Architecture description languages as presented in the next section address one part of this problem.

2.3 Architecture Description Languages

Whereas software configuration management addresses *how* to control the evolution of software systems, architecture description languages (ADLs [115]) aim at modeling *what* a software system is. The goal of ADLs is to capture large-scale structure of software systems. A widely used definition of a software architecture is given by Garlan and Shaw [55]:

Definition 3 *Software Architecture is a level of design that goes beyond the algorithms and data structures of the computation: designing and specifying the overall system structure emerges as a new kind of problem. Structural issues include gross organization and global control structure; protocols for communication, synchronization, and data access; assignment of functionality to design elements; physical distribution; composition of design elements; scaling and performance; and selection among alternatives.*

The basic building blocks of an architectural description are widely agreed and consist of components, connectors, and architectural configurations, which will be explained in the following sections.

Components

A component is a unit of computation and a data store but is not required to be both at the same time. A component in an architecture may be as small as a single procedure or as large as an entire application. It may require its own data and/or execution space, or it may share them with other components.

A component is accessed through its interface. An interface specifies the services a component provides. In order to be able to adequately reason about a component and its dependencies, ADLs also typically provide facilities for specifying component requirements such as services used and resources required.

Modeling of component semantics enables analysis, constraint enforcement, and mappings of architectures across levels of abstraction. But most ADLs provide only limited support for component semantics beyond interfaces. Semantic support found in today's ADLs are ordered event sets or special algorithm modeling languages.

Additional component constraints may be expressed in ADLs or special constraint languages such as the unified modeling language's (UML) object constraint language (OCL [138]).

Component evolution has two aspects that have to be captured by an ADL: one is the possibility of component changes to be reflected in the architecture and vice-versa. The other is to track the evolution of the architecture. This opens the question if the architecture should be versioned like any other software item or if the architecture serves as the high-level interaction point with the versioning system.

Connectors

Connectors are architectural building blocks used to model interactions among components and rules that govern those interactions. Unlike components, connectors might not correspond to compilation units in implemented systems. Connectors come in many forms e.g. message routing, shared variables, linker instructions, dynamic data structures, procedure calls, or client-server protocols.

In order to enable proper connectivity of components and their communication in an architecture, a connector exports as its interface the services it expects and it is able to connect. Connector constraints are usually specified to ensure adherence to interaction protocols, to establish intra-connector dependencies, and to enforce usage boundaries.

Configurations

Architectural configurations, or topologies, are connected graphs of components and connectors that describe the architectural structure. Connector and component interface together with other constraints define the possible configuration of a software system. In concert with semantic models of components and connectors, configuration descriptions enable assessment of concurrent and distributed aspects of an architecture, e.g., potential for deadlocks and starvation, performance, reliability, security, etc.

A goal of architecture description languages is to facilitate development of large systems, with components and connectors of varying granularity, implemented by different developers, in different programming languages, and with varying OS requirements. In consequence, ADLs provide facilities for architectural specification and development with heterogeneous components and connectors.

Tool support

To use the full potential of an ADL, several tools are required for actions like designing, visualizing, and verifying architectures. When defining an architecture, different users may require different views of the architecture. Several ADLs support two basic views of an architecture: textual and graphical.

Depending on the power of the semantic model of the ADL, analysis of certain system properties can be performed. This is especially useful in large, distributed and concurrent systems. Another aspect of analysis is enforcement of constraints. Parsers and compilers can enforce constraints implicit in types, non-functional attributes, component and connector interfaces, and semantic models.

The ultimate goal of software design and modeling is to produce the executable system. An elegant and effective architectural model is of limited value unless it can be converted into a running application. Doing so manually may result in many problems of consistency and traceability between an architecture and its implementation. ADLs therefore often come with code generation tools providing several levels of support starting from simple skeletons to almost complete applications. The degree of generated code largely depends on the semantic power of the ADL and the degree of refinement of the architecture.

It can be summarized that ADLs address a new range of problems not found in traditional programming languages. As the following sections will show, features of ADLs are more and more integrated into modern programming languages, thereby taking into account their importance.

2.4 Tools and Languages

2.4.1 Introduction

Whereas SCM systems and ADLs represent the top-down approach to the configuration problem, tools and languages are the bottom up approach. But this distinction is usually not that strict, because both interrelate in many ways and it is not yet clear what aspects are best solved at which level.

One observable trend is that specific subproblems of SCM are solved by specialized tools such as component repositories (CVS, see 2.4.4) and dependency management tools (**make**,

see 2.4.2). Another trend is that ADL features end up in programming languages and libraries such as components (CORBA, COM, and Java Beans) and interface constraints (Eiffel).

The following sections will look at two examples addressing sub problems of SCM and two languages that integrate ADL features.

2.4.2 Make

The **make** [154] tool is the first successful tool for dependency management and still in wide use today. **Make** basically operates on a declarative rule base called a “makefile” that contains rules of the form:

```
target ... : prerequisites ...
            command
            ...
```

A *target* is usually the name of a file that is generated by a command for example an executable, an object file, a stub, or a skeleton. But targets may also be symbols that represent some action to be carried out such as “clean” or “install”.

A *prerequisite* is a file or another target that is used as input to create the target. A target often depends on several files and these dependencies are represented by the prerequisites part of the rule.

A rule may have several *commands* which are the actions that **make** carries out for a rule. In that way a rule declares how a target has to be constructed when one of the prerequisites changes.

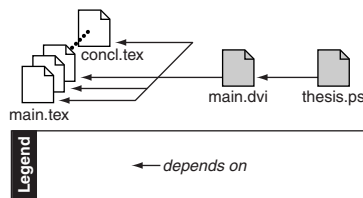


Figure 2.1: A simple dependency graph

Figure 2.1 depicts a simple dependency graph. The corresponding makefile consist of the following rules:

```
thesis.ps: main.dvi
    dvips -D 600 -Z -f < main.dvi > thesis.ps

main.dvi: main.tex ... concl.tex
    latex main.tex
```

When asked to update the target **thesis.ps**, **make** will first ensure that all prerequisites exist and are up-to-date. In this example this implies updating the target **main.dvi** that depends on several **TEX** files. To check if the target needs to be updated, **make** uses a simple heuristic that executes the action when the modification date of one of the prerequisites is newer than the target or the target simply does not exist.

Although **make** is widely used and its basic model is very simple, it has several shortcomings. The most important is that the rules usually have to be maintained by hand which makes the process error prone. Especially in today's object oriented systems where inter-object references are common, tracking dependencies becomes an enormous task. As a consequence, several SCM systems allow automatic generation of makefiles from information gathered during compilation or by other analysis techniques [107].

A second drawback is that **make** has files as its only abstraction and modification dates as the only indicator of change. Again due to the manifold dependencies of object-oriented systems, small changes may trigger many unnecessary actions. In these cases a fine-grained change control is desirable.

2.4.3 Autoconf

Autoconf [106] is a tool for producing Unix shell scripts that automatically configure software packages to adapt to a specific system. Configuration scripts produced by **autoconf** run independently of **autoconf** and require no user intervention. System features required by a software package are individually tested resulting in a flexible way of dealing with hybrid or customized systems.

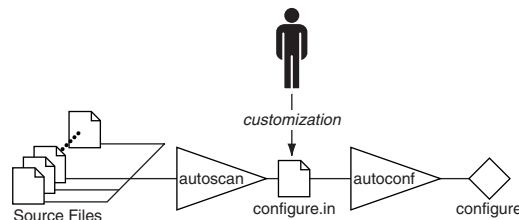


Figure 2.2: Preparing a software package for distribution with **autoconf**.

The basic process of preparing a software package for distribution is presented in figure 2.2. Tools like **autoscan**, **ifnames**, and **autoheader** help creating the initial version of a **configure.in** file. The **configure.in** lists macros defined by **autoconf** to test various system features that are required by the software packages such as executable programs, libraries, header files, typedefs, structures, compiler characteristics, specific library functions, and system services.

From the macros in **configure.in** the **autoconf** tool generates the **configure** shell script that can be executed on the platform where the software package will be deployed. The generated **configure** script tests on the target platform the features and adapts the source-, header-, make-, and other required files.

If **configure** completes without errors, all specified features were found and the software package can be deployed. This usually results in running **make** to compile the software package and **make install** to install the package. By convention, software packages have an **install** target that has as its command the actions necessary to install the package on the target system.

Autoconf takes an interesting approach by using the Unix shell as a least common denominator for executing a script (**configure**), which tests system features required by a software package. Configuration occurs under control of this script before the software package is used; ruling out many runtime configuration errors by ensuring required system properties.

2.4.4 CVS

The concurrent versioning system (CVS [28]) is an open source implementation of a component repository. The main idea, as in any component repository, is to save all revisions of software components in a central place.

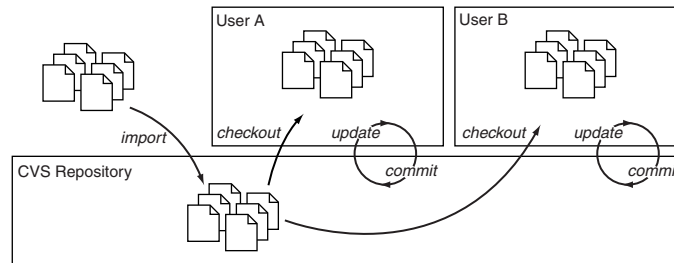


Figure 2.3: The main operations performed with CVS.

Figure 2.3 illustrates the basic operations performed with CVS. A new set of files (called a module) is added by an *import*. Once in the repository, users may concurrently create local copies of the module (*checkout*), synchronize their local copies with versions in the repository (*update*) and contribute own changes to the repository (*commit*).

Instead of using file locking to prevent that a user overwrites changes of another user, CVS takes an optimistic approach: Every user has a local copy of the repository where modifications independent of other users are made. If now a user has finished with its modification of a file, the repository is updated using *commit*. As long as no other user has committed the same file in the meantime, the new version is stored in the repository. Otherwise, a semi-automatically merge takes place. As long as the modified sections of the file are mutually exclusive, additions are merged. If the changes overlap, both versions are stored using special markers to indicate the conflicting region. CVS lets then the user decide which one to keep or to integrate both versions manually.

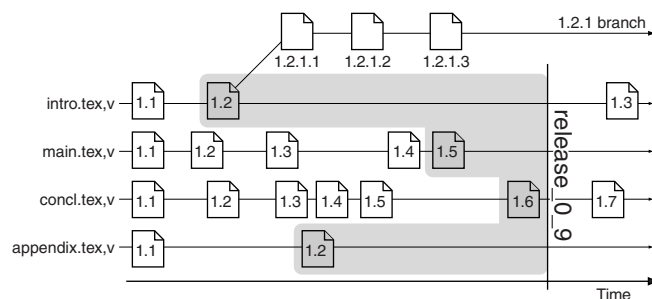


Figure 2.4: CVS revisions, a CVS release (release_0.9, gray) and a CVS branch (1.2.1, on top).

Beside file level versioning, CVS also knows release tagging. Normally every commit creates a new revision of a file by incrementing its revision number. After a while files of a module have very different revision numbers, depending on their modification frequency. What is often requested is to have a notion of a “snapshot” of a set of files, independent of each file’s revision. For example after successfully testing an application, a “snapshot” is taken before development continues. Such a “snapshot” is called a *release* and depicted in figure 2.4. Each file has its own revision history and at some moment in time a new release

“relase_0_9” is created. This release can then be referenced as a whole without knowing the revisions of the individual files.

Another feature of CVS is that not only linear revisions are supported but also variants called *branches*, existing in parallel with the main trunk of revisions. In figure 2.4 for example a new branch (1.2.1.) of the file “intro.tex” starting from revision 1.2 was created and evolves in parallel with the original branch.

CVS does a good job for file versioning and provides some support for configuration management through releases. But it leaves many questions open such as when to commit files, which files to commit, how to ensure release consistency and so on. These issues are usually solved by site policies but require some discipline from the user because CVS itself is unable to enforce them.

2.4.5 Java

Java as a modern object oriented programming language integrates some ideas from SCM and ADLs. From the SCM domain comes limited versioning supported for serialized objects, from ADLs comes the interface idea. Java also has a component model that corresponds to ADL components and connectors but this will be the subject of section 2.5.4 because it is implemented as a library and not part of the language.

As one of the first general purpose languages Java implemented a true interface concept. An interface in Java is a software contract between a client that uses a service and a server that implements the service in a class. It is important to note that an interface as a contract does not have any associated code or state; it is only a specification. Classes in contrast usually have code implementing the service and a set of instance variables (fields) that hold the state.

Interfaces form their own inheritance hierarchy similar to the class hierarchy with the difference that multiple inheritance is allowed for interfaces but not for classes. A class may therefore implement as many interfaces as it likes but only inherit from one super class.

Because Java allows objects to be serialized, another configuration problem arises. Serialization allows objects to survive the termination of the application that created them in a persistent (serialized) form. Serialization is also used when objects have to be transmitted over the network as in a remote method invocation (RMI [161]) for example. The main problem with serialization is that the class that created the serialized object may be modified at the time the object is deserialized, for example because features were added or bugs fixed.

A simple answer to this problem is to prohibit class modification for serialized objects. This means that exactly the same class that created the object is responsible for reading it back. But because data usually lives longer than the creating application, Java allows within certain boundaries modification of classes without changing their deserialization capabilities.

The default serialization mechanism of Java uses a symbolic model for binding the fields in the stream to the fields in the corresponding class. Two types of data may occur in the stream for each class: required data (corresponding directly to the serializable fields of the object); and optional data (consisting of an arbitrary sequence of primitives and objects). The stream format defines how the required and optional data occurs in the stream so that the whole class, the required, or the optional parts can be skipped if necessary. A class may redefine the `writeObject` and `readObject` methods to implement a backward compatible way for serialization and deserialization. The stream format of each class is identified by the use of a stream unique identifier (SUID). All later versions of the class must declare the SUIDs that they are compatible with.

Another SCM feature of Java is its archive format. In contrast to C for example where header files are used during compilation and libraries during runtime to check interface conformance, Java defines an archive format (JAR) that allows sets of classes together with resources like icons and sound files to be distributed as a unit. This unit (the JAR) is then used during compilation *and* runtime, ensuring compatibility.

An interesting SCM feature of Java is that it not only defines the programming language but also special “tags” in source code comments. These tags can be extracted by tools like `javadoc`, which generate documentation automatically, often eliminating the need for additional documents. The benefits are twofold: firstly, developers can easily keep the documentation in sync with the source code because it is stored in the same file. Secondly, generated documentation is much more consistent and tends to look similar even for different products, simplifying reading and navigation of documentation produced by `javadoc`.

Although Java addresses some configuration problems, the solution does not go as far as one may expect. Versioning is only implemented for serialized objects with some degree of backward compatibility. There is no way to prohibit a class to change its interface completely and break all existing contracts. Because all classes share a common namespace, it is impossible to have more than one version of a class loaded.

Having interfaces as software contracts is a good idea but interface semantics are very limited, basically they only define method signatures that follow to normal type checking rules. The next section will present Eiffel that adds more semantics to interfaces through *pre*- and *postconditions* but this mechanism is only used at runtime. Java 1.4 introduces a similar runtime mechanism called *assertions*.

2.4.6 Eiffel

Like Java, Eiffel [117] is an object oriented programming language, sharing many commonalities such as inheritance and polymorphism. But Eiffel has additional features such as multiple inheritance, genericity, and its notion of software contract. Where a software contract in Java² only limits the method signature by typing rules and thrown exceptions, Eiffel introduces a runtime mechanism through class invariants, pre-, and postconditions that increases correctness and robustness of software. Pre- and postconditions in Eiffel are a *correctness formula*:

Definition 4 A correctness formula of the form $\{P\}A\{Q\}$ means that any execution of A , starting in a state where P holds, will terminate in a state where Q holds.

More informally, this means that a *precondition* expresses the constraints under which a routine will function properly and a *postcondition* expresses the properties of the state resulting from a routine’s execution.

A post condition gives a guarantee on the part of the routine’s implementation that the routine will yield a state satisfying certain properties (at least those expressed in the post condition) assuming it has been called with the precondition satisfied.

In addition to pre- and postconditions, Eiffel also allows the definition of *class invariants*. Class invariants define properties of the state of an instance that hold when the class is not under modification. For example, a sorted array class may state as its class invariant that all elements are in increasing order. Technically, class invariants are only additions to pre- and post conditions, they are simply anded with the pre- and postconditions during execution.

²The closest notion in Java to a software contract is a Java interface.

With a class invariant I , the correctness formula of a method A would be $\{I \wedge P\}A\{I \wedge Q\}$. Nevertheless, class invariants have additional meaning because they express properties of the state of an instance independent of a specific method.

Clearly, it is desirable that pre- and postconditions can be checked statically. This would solve the static configuration problem altogether. If one could only connect components where an output's post conditions is at least as strong as the input's precondition, only correct software systems could be composed. Unfortunately, this is not possible with the intermediate functional language (IFL) used by Eiffel for expressing pre- and postconditions.

Despite that, Eiffel's software contract helps in another area where the configuration problem appears: inheritance. Eiffel enforces that a class re-implementing methods of the super class may only replace the original precondition by an equal or weaker one and the original postcondition by an equal or stronger one. This goes much further as in other languages such as Java where only type conformance of the method signature is required.

Eiffel also allows serialization of objects but provides additional help for reading back serialized object of incompatible classes. Unlike Java, which uses some built-in rules, Eiffel uses on-the-fly object conversion which goes through the following three phases:

1. **Detection** is the task of catching object mismatches at retrieval time.
2. **Notification** is the task of making the retrieving system aware of the object mismatch, so that it will be able to react appropriately.
3. **Correction** is the task of bringing the mismatched objects to a consistent state that will make it a correct instance of the new version of its classes.

Eiffel is one of the most advanced object oriented languages, integrating many features helping SCM. Eiffel usually comes with an integrated development environment (IDE), which does dependency analysis, compilation, automatic documentation and has powerful navigation techniques, further helping software configuration management.

2.5 Component Systems

2.5.1 Introduction

Surprisingly there are only a few component *languages* but several component *frameworks* as addition to existing languages. This suggests that components are orthogonal to programming languages. Some component frameworks are language or environment specific such as COM, Java Beans, and Jini, others try to provide a component framework for heterogeneous environments like CORBA and SOAP (see also 1.4). Szyperski [168] defines a software component as follows:

Definition 5 *Software components are binary units of independent production, acquisition, and deployment that interact to form a functioning system.*

Several aspects of the static configuration problem can be found in this definition:

- How are the components *deployed*? What are the necessary resources?
- How do the components *interact*? What are the dependencies?

- What is the configuration that yields a *functioning system*?

The following sections will present some component frameworks and look at how these frameworks address the above questions.

2.5.2 CORBA

Although the goal of the object management group (OMG) is to achieve interoperability on all levels for an open object market, it does not necessarily imply the creation of a component system. Nevertheless, CORBA is often seen as one. Object, service, and component are often used interchangeably in the CORBA world. The common object request broker architecture (CORBA) at the heart of the OMG effort implements “component wiring”. Another important part of CORBA is the interface definition language (IDL) that allows to express the software contract between client and server similar to the interface concept of Java (see 2.4.5).

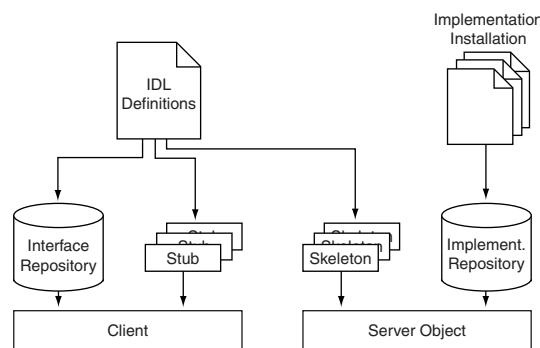


Figure 2.5: Interface and implementation Repositories

Beside being used during development where IDL files are compiled to stubs and skeletons in the target environment, IDL files are also used at runtime. Figure 2.5 shows the relation between IDL files, stubs, skeletons and the interface and implementation repositories.

The interface repository is a service that represents IDL information in a form available at runtime. Through the interface repository, a client can learn about interfaces not known at compilation time.

The implementation repository contains information that allows the ORB to locate and activate implementations of objects. The implementation repository forms together with the interface repository and the binary files of the object implementations the deployment facility of CORBA.

Interaction in CORBA is done through remote references and method invocation. Beside interface conformance no further checks are performed, leaving most of the configuration problem to the implementer.

Although CORBA provides several services for runtime system composition like the naming service, the object trader service, and the object query service; beside interface conformance little can be ensured for a configuration that it yields a functioning system. This is one of the reasons why the OMG started the development of CORBA facilities and CORBA application objects, which will specify the expected semantics for a specific set of interfaces.

The OMG also recognized the importance of versioning for CORBA. This is the reason why standardization for the change management service is under the way. SCM features like

version tracking and compatibility management will be integrated, hopefully solving some of the configuration problems.

The following section will look at Microsoft's component object model (COM) that integrates versioning already at the interface level eliminating the need for a special service.

2.5.3 COM

The component object model (COM) is the Microsoft way of component software and most Windows software has a COM interface. The "model" itself is basically a binary standard specifying how operations are found and accessed. COM itself does not specify what a component or an object is, this is subject to the *language binding*, so the term "model" is somewhat misleading. COM components are distributed in binary form as dynamic link libraries (DLL).

The central point of the language binding are *COM interfaces* that are usually defined using COM IDL, but since COM is a binary standard, this is not required. COM addresses the versioning problem already at the interface level: every COM interface has a globally unique interface identifier (IID), which serves as the "name" of an interface. This means that an IID unambiguously identifies an interface and the interface becomes immutable when an IID is generated for it. Immutable interfaces solve many binding problems; a client using an interface has the guarantee that it can access the expected services as specified at build time. Because a COM component can implement any number of interfaces, several versions of an interface may be implemented at once, allowing gradual migration. This also helps software configuration where not all clients can be migrated to a new set of interfaces at the same time.

Because interfaces usually define fine-grained aspects of components, clients have to ensure through queries that *all* required interfaces are implemented by a specific component, causing runtime overhead. For that purpose, COM introduces *categories* that are groups of interfaces that can be requested as a whole instead of every single interface in sequence. One of the reasons why *categories* are present in COM is that COM allows only single inheritance. Multiple inheritances obviously would provide the same functionality as categories. Categories have their own namespace, with similar globally unique identifiers as IIDs called CATIDs. Like COM interfaces, COM categories are immutable.

COM allows a component to define *outgoing interfaces*, interfaces the COM component would use rather than provide. A component declaring its outgoing interfaces becomes a *full connectable object* making the usually implicit dependencies explicit. This is an important step towards solving the dependency problem and helping configuration management.

Using COM to its full extent allows the creation of robust and manageable systems. Versioned interfaces ensure that contracts once established hold also in the future, categories ensure that groups of interfaces continue to exist as a group and full connected objects make dependencies explicit.

But as all other system, maybe with the exception of Eiffel, COM falls short when it comes to interface semantics. As in other systems too, only operation signatures are guaranteed, giving no additional information about the service provided.

2.5.4 Java Beans

Java Beans [69] is Sun Microsystem's component extension of Java. Java Beans is entirely written in Java and is basically a set of conventions together with some supporting services. A

Java Beans component, called a “bean” consist of:

- Support for *introspection* so that a builder tool can analyze how a bean works.
- Support for *customization* so that when using an application builder a user can customize the appearance and behavior of a bean.
- Support for *events* as a simple communication metaphor that can be used to connect up beans.
- Support for *properties*, both for customization during build time and for programmatic use at runtime.
- Support for *persistence*, so that a bean can be customized in an application builder and then have its customized state saved away and reloaded later.

The three most important features of a bean are the set of properties it exposes, the set of methods it allows other components to call, and the set of events it fires. Properties are named attributes associated with a bean that can be read or written by calling appropriate methods on the bean. The methods a bean exports are just normal Java methods, which can be called from other components or from a scripting environment. By default all of a bean's public methods will be exported, but a bean can choose to export only a subset of its public methods. Events provide a way for one component to notify other components that something interesting has happened. When the event source detects that something happens it will call an appropriate method on a previously registered event listener object. This mechanism is similar to COM's outgoing interface.

Each bean has to be capable of running in a range of different environments. Java Beans clearly distinguishes between design time and runtime environments. First, a bean must be able to run inside a builder tool. This is often referred to as the design environment. Within the design environment it is important that the bean provides design information to the application builder and allows the developer to customize the appearance and behavior of the bean through a user interface provided by the bean. Second, each bean must be usable at runtime within the generated application. In this environment, there is much less need for design information or customization. The design time information and the design time customization code for a component may potentially be quite large. Therefore Java Beans allow the design time interfaces to be supported in separate classes from the runtime interfaces. All beans must support serialization. It is always valid for an application to save and restore the state of a bean using the Java serialization APIs.

Additional services like the Java activation framework (JAF) [26] and InfoBus [30] address platform specific problems facilitating bean deployment in heterogeneous environments.

Neither Java Beans nor the Java platform define a consistent strategy for typing data, a method for determining the supported data types of a software component, a method for binding typed data to a component, or an architecture and implementation that supports these features.

This is exactly the goal of the Java activation framework (JAF). JAF provides some degree of automatic configuration of components so that a bean can adapt itself to the data and the environment it is dealing with. JAF allows determining the type of arbitrary data, to encapsulate access to data, and to discover the operations available on a particular type of data.

The InfoBus interfaces allow the application designer to create data flows between cooperating beans. In contrast to an event/response model, where the semantics of an interaction depend upon understanding a bean-specific event and then responding to that event with bean-specific callbacks to the event source, the InfoBus interfaces define very few events and have a fixed set of method calls for all components.

Beans in an InfoBus application can be classified in three types: data producers, data consumers, and data controllers. Between components, data flows in named objects are known as *data items*. Data controllers are specialized components that mediate the rendezvous between producers and consumers.

InfoBus wiring includes the following steps:

1. Establishing InfoBus participation.
2. Listening for InfoBus events. Once an object is a member of an InfoBus, it receives bus notifications by implementing an interface and registering it with the InfoBus.
3. Rendezvous on the data to be exchanged.
4. Navigation of structured data in standardized way. For example a spreadsheet and a database may be accessed in the same tabular way.
5. Retrieval of an encoding of the data value such as a Java `String` or a Java `Object`.
6. The consumer may change the data if allowed by the producer.

Java Beans realizes a component extension for Java providing design and deployment support, various kinds of interaction (method calls, events, and InfoBus) as well as some adaptive support through JAF. Because Java Beans is entirely written in Java, beans are highly portable and are deployed in environments ranging from web-terminals to mainframes. Several companies produce Java Beans for horizontal and vertical applications, realizing a component market.

2.5.5 Jini

The Jini [179] technology is not really a component system, although it fulfills the definition given above for software components and adds several interesting features in the context of the configuration problem. Jini aims at enabling federations of spontaneously networked components to communicate, interact, and share their services and functions. This implies the main aspects of the configuration problem.

As Java Beans, Jini is entirely Java based which eliminates the heterogeneity problem to some extent. Jini technology assumes a changing, long living network with an increasing population of devices, services and users. This assumption has several implications:

- No centralized administration.
- Many different, simultaneous versions of devices and services.
- Jini must be scaleable.
- Jini must be self adapting.

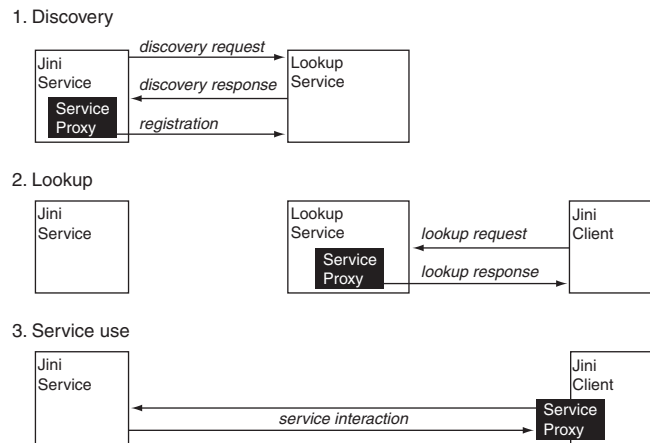


Figure 2.6: The three main phases that lead to the use of a Jini service.

At the heart of Jini is how the system is automatically configured, namely how a service is discovered by the network, published, and found by others. Figure 2.6 shows the three main phases involved in discovering and using a Jini service:

1. A service that wants to join a Jini federation multicasts over the network asking for a lookup service to respond. The set of lookup services act as a network wide directory of services available. A service then can register a proxy for itself in the lookup service, thereby publishing itself on the network.
2. A client looking for a service firstly discovers a lookup service by multicasting. Then the client requests a particular service specified by a Java interface and additional properties. The lookup service then responds with a proxy for a matching service.
3. The client can now use the service through its Java interface implemented by the proxy that hides all the communication details with the service.

Since the proxy used to talk to the service by the client is placed in the lookup service by the service itself, it knows all necessary communication details for accessing the service. The client only sees the Java interface it has requested which is implemented by the proxy.

An interesting property of this approach is that Jini unlike CORBA or SOAP does not define a “wire” protocol but only Java interfaces for accessing services. The ability to move code from the service provider to the client opens up a wide range of configuration possibilities. For example a proxy might implement the entire user interface for a service, which can then be run on the client, going beyond the options offered by a standardized programmatical interface.

Jini takes an interesting approach to the configuration problem through its dynamic service discovery and publishing as well as the proxy concept for communication. Although the techniques used by Jini potentially allow incremental extension and evolution of the system it suffers the problems inherent to Java interfaces: No semantics are provided beyond method signatures, and interfaces have to be agreed and communicated. This means that clients have to know in advance which interfaces they will use in the future, limiting the room for innovation and runtime extension of the system. In addition, Java interfaces are not versioned;

immutability of interfaces can therefore only be guaranteed by an appropriate policy. Although moving code from the service to the client opens many possibilities, not every client may feel comfortable with accepting code from any source. Jini builds on the standard Java 2 security model, which was designed for static environments and is considered inadequate for a dynamic scenario [43].

2.5.6 Web services

Universal description, discovery, and integration (UDDI [175]), the Web service description language (WSDL [37]), and the simple object access protocol (SOAP [65]) are at the heart of the so-called “Web services”. CORBA and similar standards allow the integration of heterogeneous entities, but still all entities had to agree on the same middleware standard. The chances that any of two systems not built at same site are based on the same middleware turned out to be small in practice. The result was a “component-war” of concurring systems, custom adapters and translators, all complicating system integration.

The goal of Web services is to act as a “meta-middleware” that allows interaction of services as long as they follow the Web service protocols and use the Web service infrastructure.

Building and using a Web service involves the following steps:

1. A provider creates, assembles, and deploys a web service using the programming language, middleware, and platform of its own choice.
2. The provider defines the Web service in WSDL that describes the service to clients.
3. The provider registers the service in UDDI registry that enables the provider to publish the Web service and allows clients to look up the service.
4. A client finds the Web service by searching a UDDI registry.
5. The client application binds to the Web service and invokes the service’s operations using SOAP (see 1.4.3 for more information about SOAP).

UDDI

Before a client can use a Web service, it must first locate a service provider, discover the interface and semantics and write or configure software on the client end to collaborate with the service. UDDI serves as a registry for Web services and a global, public directory of business and services called the UDDI business registry (URB) is operated by the UDDI members. UDDI offers various search mechanism such as a “white-pages” search that allows looking for specific providers, a “yellow-pages” service that allows searching by categories and a “green-pages” search that allows to look for specific services. UDDI is accessed via SOAP and defines interfaces for publishing, editing, browsing, and searching of information in the registry. UDDI is a general repository for information, often used together with WSDL but it may also contain human-oriented information like a Web page or an e-mail address.

WSDL

A client using a service needs to understand the call syntax and semantics. The Web service description language (WSDL) is an XML application, which describes the interface, semantics,

and administrative issues of a Web service call. A WSDL document contains the following key information:

- A description of the message format that is understood by the Web service.
- The semantics of the message passing like request-only, request-response, response-only.
- A specific encoding to be used over which transport such as HTTP, HTTPS, or SMTP.
- The URL of the described service, which can be used by the client to access the service.

WSDL documents are usually stored in an UDDI registry which allows the described services to be found through the UDDI search interfaces.

ONE

Sun Microsystem's open net environment (ONE [81], [167]) is a standard, which describes agreements between applications and the containers in which they run. The ONE architecture is mostly based on Java and describes an environment as depicted in figure 2.7.

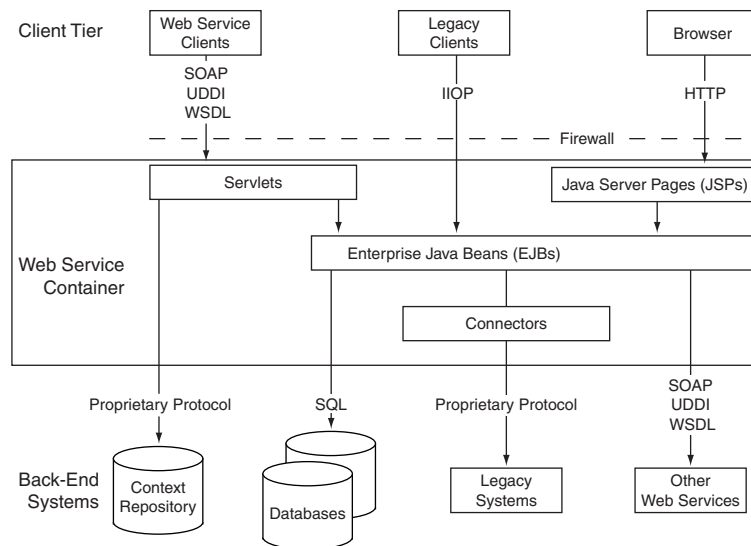


Figure 2.7: The ONE Web service architecture

- **Client Tier:** The client tier handles interaction with Web services, legacy applications, and human users. Web service clients use the Web service standards SOAP, UDDI, and WSDL. Legacy clients (CORBA, or RMI based) access the enterprise Java Beans (EJB) layer directly via the Internet inter-ORB (IIOP) protocol. Web or wireless application protocol (WAP) browsers connect to Java server pages (JSP) which render the user interface for human clients in HTML, XHTML, or WML.
- **Web service container:** This is the main part of ONE, which ensures quality of service for Web service applications such as transactions, security and persistence. The EJB layer implements the Web service logic. The service logic is built using existing infrastructure in the back-end layer, accessed by the corresponding protocols.

- **Back-end systems:** The back-end systems provide processing and data to the applications in the Web service container. The back-end consists of data base access, integration of legacy systems, and use of other Web services.

.NET

Microsoft's .NET is a collection of products that enables Web service construction. This is one of the biggest differences between ONE and .NET; ONE is a standard whereas .NET is a collection of products, although Microsoft is currently opening .NET to standardization.

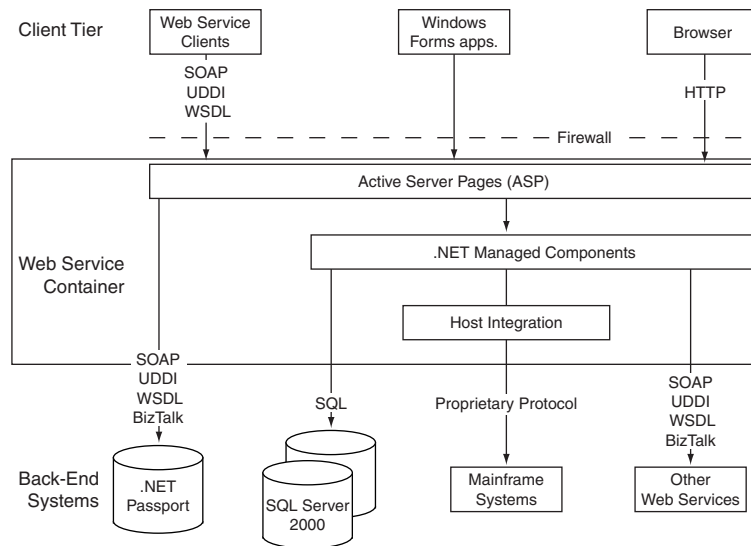


Figure 2.8: The .NET Web service architecture

Figure 2.8 shows the basic building blocks of the .NET Web service architecture.

- **Client tier:** The client tier handles interaction with Web services, heavyweight clients built using Windows forms, and Web or wireless application protocol (WAP) browsers. Browsers connect to active server pages (ASP) which render the user interface in HTML, XHTML, or WML.
- **Web service container:** As in ONE, the Web service container ensures quality of service for applications such as transactions, security, and persistence. The Web service logic is implemented with .NET managed components. Several Microsoft products such as active data objects (ADO), and COM as well as standards like SOAP, UDDI, and WSDL are supported. The service logic is built using existing infrastructure in the back-end layer, accessed by the corresponding protocols.
- **Back-end systems:** The back-end systems provide processing and data to the applications in the Web service container. It consists of data base access, integration of legacy systems, and use of other Web services.

The .NET product portfolio includes the development environment Visual Studio .NET and the .NET servers (SQL Server, Exchange Server, Commerce Server, Application Center Server,

.NET Service	Description
Profile	Name, nickname, special dates, picture, address
Contacts	Electronic relationships/address book
Locations	Electronic and geographical location and rendezvous
Alerts	Alert subscription, management, and routing
Presence	Online, offline, busy, free, which device(s) to send alerts to
Inbox	Inbox items like e-mail and voice mail, including existing mail systems
Calendar	Time and task management
Documents	Raw document storage
ApplicationSettings	Application settings
FavoriteWebSites	Favorite URLs and other Web identifiers
Wallet	Receipts, payment instruments, coupons, and other transaction records
Devices	Device settings, capabilities
Services	Services provided for an identity
Lists	General purpose lists
Categories	A way to group lists

Table 2.1: The “.NET My Services” (Hailstorm) set of Web services.

Host Integration Server, Internet Security and Acceleration, and BizTalk). Web services built on top of that are collected under the name “.NET My Services” formerly called “HailStorm” and are listed in table 2.1.

With ONE and .NET [177] two strong market forces stand behind the Web service initiative boosting its underlying infrastructure SOAP, UDDI, and WSDL. It is interesting to note that the Web services effort developed individual standards for all three problems (heterogeneity, static, and dynamic configuration) described in the first part of this dissertation:

SOAP handles heterogeneity, WSDL handles static configuration information, and UDDI allows dynamic lookup and configuration. Although all protocols provide more flexibility and semantics than Java or CORBA interfaces, they still are quite limited in their expressiveness and there is currently no way to describe service dependencies and resource requirements.

2.6 Summary

The static configuration problem persisted through the last decades of computing and is not likely to disappear. New trends such as Web services and spontaneous networking even increase the configuration problem, adding dynamic aspects, which will be the subject of the next chapter.

SCM systems provide good support for applications where all configurable items are under a single control. Architecture description languages try to capture the large scale aspects of software systems, helping the design and understanding of large and distributed applications. Programming languages start to integrate features found in dedicated systems, helping development, configuration, and deployment.

A natural result of the “software crisis” as well as SCM and ADL research was component software. Well-encapsulated software entities that can be independently deployed and connected give developers and designers higher-level abstractions than procedures or objects.

Systems like Jini even remove the connection aspect from the developer's responsibility by automatic discovery and lookup, resulting in some auto-configuration capability. The main shortcoming of all these systems is that they are based on interface descriptions with weak semantics. Usually only operations and accepted argument types are specified, leaving the description of operational aspects to user documentation. Eiffel and some ADLs provide further support in this direction.

The Web service effort tries to address the specification problem using WSDL and UDDI but it may be questioned if the expressive power of WSDL is sufficient for automatic configuration of Web services and how these technologies perform in spontaneous networked environments such as ubiquitous computing.

Chapter 3

The Dynamic Configuration Problem



3.1 Introduction

This chapter looks at dynamism that is, unlike heterogeneity and configuration, part of computing since the beginning. The success of computers is largely due to the fact that they manipulate dynamic data at an enormous speed compared to other techniques. For some time, data was the only part of the system that varied. But with the introduction of multiprogramming, system resources started to change as well. Multiprogramming added concurrent access to resources such as main memory, CPU cycles, and stable storage. Programs had to be written based on the models and abstractions defined by the operating systems instead of a static hardware specification.

The trend towards open and expandable systems increased the dynamic aspects of the configuration problem. In an open system, a program could find itself in a very different environment between two runs. System resources like main memory, stable storage, and CPU power may change due to hardware upgrades or concurrent processes.

In the beginning, peripheral devices were usually not allowed to be connected and disconnected during runtime but some of them already supported different states of availability (on- and offline) as well as individual power control. Another example is removable storage

(e.g. floppy disks and tapes), which is supported by software for quite a while. With the introduction of hot-pluggable busses such as PC Card [158], USB [31], or IEEE 1394 [173], peripheral devices are attached and detached during runtime. The user of a modern operating system expects it to adapt to these changes and automatically prompt for required software if a previously unknown device is presented to the system. In some cases whole parts of the system hardware appear and disappear during runtime, as it is the case when a laptop is connected to a docking station or used standalone.

Of course, these hardware changes are reflected on the software side as well. Loading and unloading of software components dynamically changes the operating system¹ or the runtime environment of an application². This results in a partially component based operating system.

In a networked environment the dynamic aspects increase even more. Whole parts of a software system vary along various dimensions such as version, availability, and performance for example. Many new computing environments exhibit these dynamic aspects like Web and ubiquitous computing. Future applications will have to deal more and more with a dynamic environment and adapt their capabilities in function of the resources available. This dynamic configuration problem can be stated as:

Definition 6 *Given dependent entities in a computer system providing service and a changing set of resources, the dynamic configuration problem is to maximize the quality and quantity of services in function of time and resources available.*

The following sections look in a top down fashion at the dynamic aspects found in software systems, changing system resources, and changing hardware configurations.

3.2 Changing Software

3.2.1 Introduction

Software was for a long time considered as monolithic as hardware; if changes were applied they occurred in a controlled fashion often involving system restart. This is mainly due to the static configuration problem (see chapter 2) which handles most part of software configuration still during development and deployment. Proper versioning and dependency management seldom occurs at runtime and there is only little operating system support.

With the introduction of networked environments and plug-and-play hardware, environments are becoming more and more dynamic. Software running in such environments has to be adaptive, provide some degree of auto configuration, and handle local and incremental upgrades.

The next sections present a highly dynamic area (the Internet) as well as three systems addressing the dynamic aspects of their respective environments. The Internet as an inherently heterogeneous and dynamic environment is presented first, followed by the Web operating system (WOS) which addresses the problems of Internet computing. Where the WOS tries to solve the problems as an operating system, Harness goes a level below, providing a reconfigurable, distributed virtual machine (DVM). Jini takes yet another approach by proposing a model where everything is a service, ignoring existing structures and layering.

¹Drivers add to or remove functionality from the operating system.

²Dynamically loaded libraries extend the capabilities of an application at runtime.

3.2.2 Internet

The Internet is the most prominent example of re-configurable software. The Internet is dynamic in many dimensions:

- The very protocol of the Internet, IPv4 is currently being replaced by IPv6.
- The network itself (the nodes and links) is constantly changing.
- On top of the transport protocols UDP and TCP, new protocols are defined frequently.
- Higher level service interfaces emerge almost daily.
- The service implementations and accessible data increase exponentially.

Although the Internet was very dynamic from the beginning, configuration was done mostly manual. Only a few protocols have auto configuration built in such as the address resolution protocol (ARP [23]), the routing information protocol (RIP [109]), and the dynamic host configuration protocol (DHCP [40]) for example. The current Internet still requires a lot of maintenance for basic services such as routing or naming (DNS).

What the Internet achieved well was locality of maintenance. The basic protocols are very stable and changes could always be applied incrementally, it was never necessary to “shut down” the Internet for an upgrade.

A special case is HTTP in combination with HTML forming the base of the World Wide Web. Although the World Wide Web is one of the most dynamic parts of the Internet, it is also one that deals badly with changes. Neither HTML nor HTTP provides proper versioning for resources or generalized redirection support. Dead links and inaccessible pages are part of the daily live of a Web user. The future will show if this situation improves with Web services (see 2.5.6) that build on the same infrastructure but have additional support by directory services like the UDDI registry.

Additions like the IP mobility support [133] prepare the Internet for the dynamic environment of mobile terminals. IP mobility support specifies protocol enhancements that allow transparent routing of IP datagrams to mobile nodes in the Internet. Each mobile node is identified by its home address, regardless of its current point of attachment to the Internet. The host with home address is responsible of forwarding traffic to the mobile terminal. The trend towards mobile Internet is also highlighted by the integration of IP in the general packet radio service (GPRS) which provides packet service on a GSM network, generally used for IP traffic.

3.2.3 WOS

Because the Web is dynamically changing in many directions, any attempt to design one single operating system offering a fixed set of resource-management functions will have difficulty adapting to innovation or to new demands. The characteristics distinguishing computing on the Internet from classical distributed systems are the main questions to be answered by the web operating system (WOS). Besides its highly heterogeneous nature, the Internet does not have a complete catalogue of all available resources. Moreover, such a catalogue is impossible to build because it is highly dynamic. A central decisions making for resource allocation is not acceptable or even impossible. Therefore, the Web operating system is a decentralized

that inherently supports versioning. Different versions of the operating system are running simultaneously on multiple network nodes providing different versions of services. Should for instance a given version not be capable of dealing with a particular request for a service, it can pass it on to another version. Generalized software configuration techniques, based on a demand driven technique called *eduction* are developed. Software and hardware warehouses provide the necessary components for fulfilling a service request (see figure 3.1).

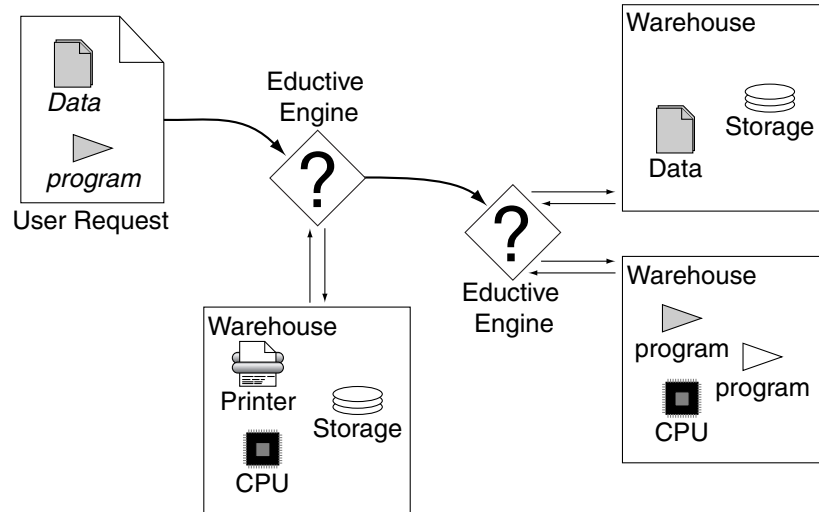


Figure 3.1: A user request propagates through two eductive engines.

As a consequence, the kernel of a WOS node is a general eductive engine, which allows interaction with many different warehouses, each offering different versions of services, resource management techniques, applications, platforms, and hardware. An eductive engine is a reactive system responding to requests from users or other eductive engines, and fulfils requests using its warehouses.

The WOSNet, the collection of all nodes running WOS kernels, consists of a collection of eductive engines together with many different warehouses. Figure 3.1 illustrates the functioning of the WOS:

A request made by a user to run a particular program, along with specified data, is sent to the closest eductive engine, which might reside on one's own machine. Upon reception of such a request, the eductive engine decides whether it is capable of dealing with the request. After all, that engine might simply be overloaded and the request might be of too high priority to wait. The engine looks in its software warehouses to determine if it actually has the requested program. If not, it might refuse service or pass on the request to one or more other eductive engines, until finally one engine accepts responsibility for the request and returns the result to the user.

WOS Nodes

As can be seen from figure 3.1, each node in the WOS plays a client and a server role. A client request may come directly from a user or may be passed on from another WOS node. Figure 3.2 depicts the general architecture of a WOS node [96].

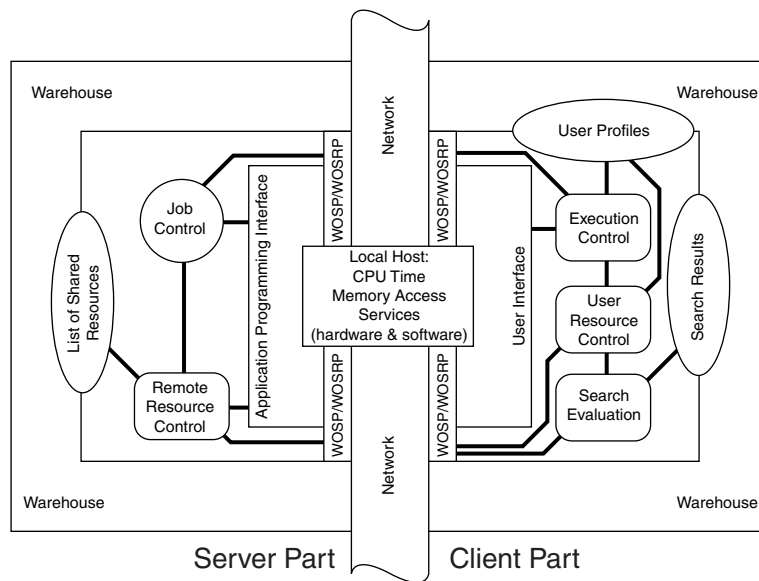


Figure 3.2: Structure of a web operating system node.

The server part of a WOS node is shown on the left and the right side represents the client part. The client side's execution control unit processes every incoming user command. It decides whether a process must be locally executed (e.g. a `ps`-command) or not. While locally executable jobs will directly proceed to the user interface of the respective operating system, all other service requests are sent to the user resource control unit. A decision will be made, whether the requested service can be fulfilled in the reserved standard user space or not. If so, a common load sharing mechanism will determine on which machine this will be done. Otherwise, a search over the net of WOS nodes must be started. This is the task of the search evaluation unit, which also evaluates the results of a search request. For that search, former search results from the local warehouse can give a satisfactory answer and will be reused. Otherwise, a net wide search must be organized. The results will be used by the user resource control unit to contact potential hosts.

A successful service execution requires that at least one remote resource control unit of a WOS node must obtain from its local warehouse a valid entry that the service can be executed on that machine for the requesting user, and must be able to allocate the necessary resources for the execution. The request can then be fulfilled there under the responsibility of the remote job control unit, which also contacts the user execution control to return the results.

3.2.4 Harness

Harness [10] is an experimental meta-computing system based upon the principle of dynamically re-configurable, object-oriented, networked computing frameworks. Harness supports reconfiguration not only in terms of the computers and networks that comprise the virtual machine, but also in the capabilities of the VM itself. These characteristics may be modified under user control via an object-oriented “plug-in” mechanism that is the central feature of the system. The motivation for a plug-in based approach to re-configurable virtual machines is derived from two observations:

First, distributed and cluster computing technologies change often in response to new

machine capabilities, interconnection network types, protocols, and application requirements, the typical dynamic configuration problem. At the system level, the capability to reconfigure the set of services delivered by the virtual machine assists in overcoming legacy problems and the incorporation of new technologies.

The second reason for the plug-in model is to attempt to provide a virtual machine environment that can dynamically adapt to meet an application's needs, rather than forcing the application to fit into a fixed environment. Harness is designed in a way that resources such as CPU power may be added during the applications run-time. In fact, the Harness reconfiguration capability and its object-oriented design allow building modular, plug-in based programming environments that can be plugged into the system on demand in order to tailor the system to the needs of the application.

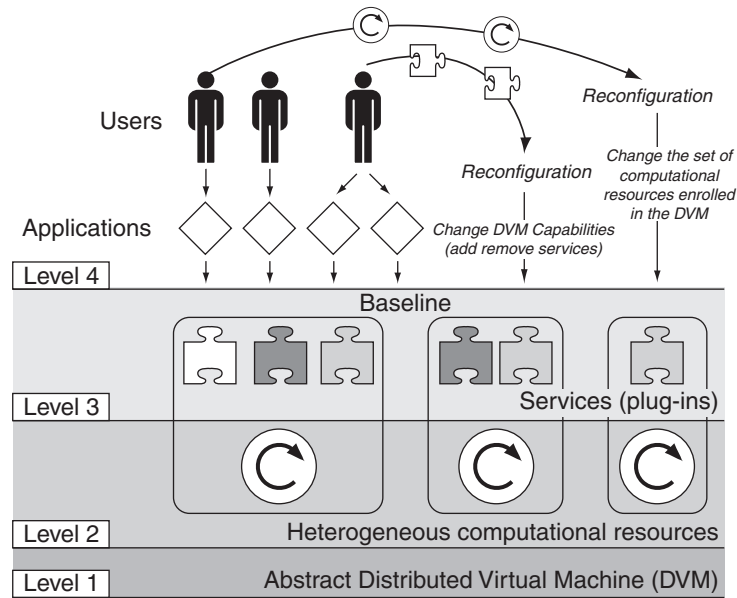


Figure 3.3: The harness distributed virtual machine (DVM) architecture.

The fundamental abstraction in the Harness metacomputing framework is the distributed virtual machine (DVM) (figure 3.3, level 1). A DVM is associated with a symbolic name that is unique within a Harness name space, but has no physical entities connected to it. Heterogeneous computational resources may enroll into a DVM (figure 3.3, level 2) at any time; however at this level the DVM is not yet ready for use by applications. To adapt to the application's needs, the heterogeneous computational resources enrolled in a DVM need to load plug-ins (figure 3.3, level 3). A plug-in is a software component implementing a specific service. By loading plug-ins that implement services, it is possible to complement the set of native services of a computational resource in such a way that all the computational resources enrolled in a DVM present a consistently homogeneous service baseline to applications (figure 3.3, level 4, baseline). Users may reconfigure the DVM at any time (figure 3.3, level 4) both in terms of computational resources enrolled by having them join or leave the DVM, and in terms of services available by loading and unloading plug-ins.

The main goal of the Harness metacomputing framework is to support the ability to enroll heterogeneous computational resources into a DVM and make them capable of delivering a consistent service baseline to users. This goal requires the programs comprising the frame-

work to be as portable as possible over a selection of systems as large as possible. The availability of services to heterogeneous computational resources derives from two different properties of the framework: the portability of plug-ins and the presence of multiple searchable plug-in repositories. Harness implements these properties primarily by leveraging two different features of Java technology. These features are the capability to layer a homogeneous architecture such as the Java virtual machine (JVM) over a large set of heterogeneous computational resources, and the capability to customize the mechanism provided to load and link new objects and libraries.

The Harness DVM is an interesting approach to the dynamic configuration problem. Instead of letting the application adapt in function of the changing environment, the environment is changed depending on the application currently executing. The advantage is that the application can be written against a guaranteed baseline of services and the responsibility to provide that environment is delegated to the DVM.

3.2.5 Jini

The scenario addressed by Jini technology [164, 166], spontaneous networking and service delivery, is inherently dynamic. The Jini technology infrastructure is built around the model of clients dynamically requesting service. A service can use other services, and services can be grouped together to provide higher-level functionality. Services are defined by a Java interface and a proxy implementing the interface (see also 2.5.5). The proxy is uploaded into the lookup service by the service provider and sent to the client upon request [165]. Once loaded into the client, the proxy handles all communication with the service.

Since Jini is entirely Java based, hardware heterogeneity is hidden and only a few additional abstractions are necessary. As a result, Jini has a very flat structure mainly consisting of support libraries and the Jini services. The other Java technologies Jini relies on are:

- Remote method invocation (RMI [161]) supplies the basic interaction mechanism of Jini services although communication between a client and a service is hidden by a proxy.
- Security is based on the Java 2 security model [60].
- Leasing [160] is used to grant guaranteed access over a time period. Leases are either exclusive or non-exclusive.
- Transactions [163] are used to group a series of operations, either within a single service or spanning multiple services.
- Distributed events [159] allow other objects to register interest in events and receive a notification of the occurrence of such an event.
- JavaSpaces [162] are often found in a Jini environment used for coordination. The implementation of the lookup service for example is based on JavaSpaces.

Jini does not distinguish between physical devices and services implemented as software only. As a result, dynamism appears on a logical level only.

Using other distributed Java technologies such as leasing and transactions, Jini allows building robust distributed application that can recover from network and service failures in a proper way.

Jini addresses dynamism in several ways through its dynamic service discovery and publishing as well as the proxy concept for communication. Because of the versioning and semantic limitations of Java interfaces, the dynamic configuration possibilities of Jini are limited to the possibilities foreseen by the developers of services and clients.

3.3 System Resources

3.3.1 Introduction

The concurrent access to system resources is the classical dynamic configuration problem handled by operating systems. System resources exhibit dynamic behavior in various areas:

- **Exclusive resources:** Applications have to be prepared that exclusive resources may be used by another application. Example of exclusive resources are printer and modem ports.
- **Shared resources:** Applications or the operating system have to coordinate access to shared resources. Examples of shared resources are the screen or the resources listed below.
- **Computing resources:** Processes in a multiprogramming environment have to deal with varying CPU power; in a distributed system, process have to be placed depending on the individual node load.
- **Address space:** Various components of an application and the operating system have to share the same address space.
- **Stable storage:** Applications have to handle changes in available stable storage space and latency.

The following sections look at the techniques used by operating systems in order to manage system resources. Because changing or plug-and-play hardware introduces new dynamic aspects not found in classic operating systems, they are the subject of a section of its own (see section 3.4).

3.3.2 Exclusive resources

There are many resources in a computer system, which can only be used by one application at a time. For example, a modem port cannot be shared between applications because multiple applications may have conflicting goals such as dialing two different numbers at the same time that cannot be handled by a modem.

Access to exclusive resources therefore has to be coordinated, or to put in other terms; the software has to be configured in such a way that exclusive resources are accessed by one software component only. Various techniques and algorithms exist to handle exclusive access to resources: With *arbitration*, one out of multiple simultaneous requests can be safely chosen. *Mutual exclusion* allows an application to perform operations on a shared object without interference from other applications. *Synchronization* forces one process to stop and wait for an event from another process or from the operating system.

Unfortunately, these techniques are not sufficient when multiple resources are used by multiple applications simultaneously. Consider two applications A_1 and A_2 each requiring exclusive access to two resources R_1 and R_2 . Application A_1 requests them in the order $\langle R_1, R_2 \rangle$ and application A_2 in the inverse order $\langle R_2, R_1 \rangle$. If now both application start at the same time no one will ever proceed because application A_1 will have access to resources R_1 but wait for R_2 , whereas application A_2 has access to resource R_2 , waiting for R_1 . This situation is known as a *deadlock* and is a system-wide phenomenon that can occur in any collection of processes that can stop and wait for resources held by other processes. The main techniques to circumvent this situation are *deadlock prevention*, *deadlock avoidance*, and *deadlock detection* [171].

Two strategies for doing *deadlock prevention* are to force a process to acquire all resources in advance, or to force a process to acquire new resources in a priority order. Obviously both strategies waste resources in the sense that often resources are locked by a process longer than really needed.

The central idea behind *deadlock avoidance* is that a system controller monitors each request for an additional resource, granting it only if there is a way to run all the processes to completion and release that resource again. Unfortunately this strategy is too expensive for many systems, especially those with many processes and many resources.

Deadlock detection tries to find circular wait conditions and abort one of the process involved in the circular wait. Deadlock detectors are easy to build, but cannot prevent the cost of having to abort some processes to free up the deadlock.

Which strategy is best depends on the application requirements. Multiple techniques should be available to allow deadlock free dynamic configuration of components under given quality of service constraints.

3.3.3 Shared resources

Shared resources are finite like exclusive resources but not in a binary way. They have at least one dimension along which they can be partitioned which enables shared access, usually with a graceful degradation of quality of service when the number of resource users increases.

For example windowing systems allow sharing of the physical screen at the cost of cluttering the screen with overlapping windows when multiple applications are using it simultaneously. Another example is a time-shared CPU that reduces the processing power available to one participant in function of the number of participants.

Various techniques are available today to handle access to shared resources, mostly implemented by the operating system. In this light, applications usually ignore the sharing problem and are programmed without inter-application coordination.

However, there exist environments where additional inter-application coordination is mandatory either due to the number of concurrent applications or the division granularity of the shared resource. Examples are the Internet where the number of applications is very high or ubiquitous computing where resources are usually scarce.

As in the classical case where the operating system handles shared resource access, handling inter-application coordination should not be the task of the Internet or ubiquitous computing application but part of the runtime environment that configures the software components depending on the context and according to user preferences.

3.3.4 Computational resources

Sharing the CPU among competing processes is known as *scheduling*. Upon an event (usually an interrupt like a clock tick, peripheral device completion notification, or system call) a scheduling algorithm has to decide which process to run next and for how long. A good scheduling algorithm should exhibit the following criteria [170]:

- **Fairness:** Make sure each process gets its fair share of the CPU.
- **Efficiency:** Keep the CPU busy 100 percent of the time.
- **Response time:** Minimize response time for interactive users.
- **Turnaround:** Minimize the time batch users must wait for output.
- **Throughput:** Maximize the number of jobs processed per hour.

Obviously some of these criteria are conflicting, a scheduling algorithm therefore has to carefully choose which one to favor.

In a distributed environment the problem of using computational resources is usually known as *load balancing* [94] or *load sharing* [176]. The goal of *load balancing* is to dynamically distribute the processes among network nodes such that certain criteria like response time is minimized or job throughput is maximized. Load balancing mechanisms are usually based on broadcasted or piggybacked load information in combination with prediction strategies. When the number of nodes increases or process migration is costly, load balancing is replaced by *load sharing* which selects a node from a subset, does not migrate processes, and does not aim at globally optimizing certain criteria.

3.3.5 Address Space

During application execution, main memory is the core area of an application's activity. Most of today's CPUs have a model of a virtual address space where different areas, often called segments, are mapped:

- **Text segment:** The instructions comprising the application. The text segment usually has a fixed size and is read-only.
- **Heap segment:** The area where the application allocates space for dynamic data structures.
- **Stack segment:** The area that is used to store invocation arguments and results as well as return addresses. In multi-threading environments there may be multiple stacks, one for each thread.
- **Shared memory segment:** Inter-process communication may happen through areas of shared memory where multiple processes can access concurrently.
- **Shared library segment:** Another form of shared memory where basically read-only code is shared among multiple processes.
- **Operating system structures:** Some operating systems make their structures accessible in the virtual address space of a process.

- **Hardware:** Some CPUs use memory mapped input output where peripheral devices appear in the virtual address space of a process.

Normally the operating systems takes care of the address space handling together with hardware support by a memory management unit (MMU) that introduces a mapping mechanism between the virtual address space presented to the application and the physical address space implemented by the hardware. However, some system like the Java virtual machine (JVM, see also 1.3.3) abandon the address space model altogether, increasing the abstraction and providing better protection against faulty memory access.

Basically the dynamic configuration problem for the address space is to map multiple dynamic regions to a static, linear region. The increasing memory demand of applications complicates the configuration problem, leading to various segmentation and overlay mechanisms. Most of the current CPUs simply increased the virtual address space (modern CPUs have an address space of 2^{64} today) in order to have some “room” for mapping. New memory models like the object-based model of the Java virtual machine together with a garbage collector show alternative paths to the relatively low-level address space abstractions available today.

3.3.6 Stable Storage

Stable storage is for a long time the service where an application dynamically allocates large amounts of space. Stable storage is basically accessed today through either a raw, a file system, or a database interface. All provide the necessary abstractions to dynamically change the physical space used.

Modern operating systems for example allow dynamic addition of physical space during runtime such as Windows 2000’s dynamic disks [119] or Linux’s logical volume management (LVM) [70].

LVM adds an additional layer on top of physical partitions. Without LVM, a physical disk contains multiple partitions. On these partitions are filesystems or they are used raw (i.e. not managed by a filesystem but with direct low-level access from the application such as a database). A logical volume manager puts the physical entities called volumes into storage pools called volume groups. The LVM in Linux can manage whole SCSI- or IDE-disks in a volume group as well as hardware- and software redundant arrays of independent disks (RAID) devices. A volume group is the equivalent of a physical disk from the systems point of view. The equivalent of partitions into which this storage space is divided for creating different filesystems and raw partitions on it is called a logical volume. If an appropriate file system such as ReiserFS is used in a logical volume, even the volume can be resized during runtime.

The relatively high-level file system and database interfaces can also be used over the network through protocols like NFS, SMB, or SQL*Net, decoupling the storage configuration from the storage access. Mainly two different approaches are in use today for network storage: storage area networks (SAN) and network attached storage (NAS) [11, 129]. Figure 3.4 summarizes the approaches for stable storage management.

In a direct attached storage (DAS) system, the application accesses the disks either directly or through a file system. An additional logical volume manager may manage the disks.

A SAN system is a dedicated storage network designed specifically to connect storage, backup devices, and servers. SAN is commonly used today to describe FibreChannel fabric

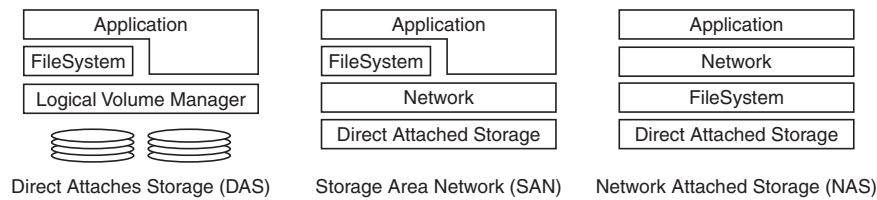


Figure 3.4: The techniques in use today for stable storage

switched networks of storage devices. A SAN is connected behind the servers through a block-level interface. The servers implement the file system interface through network protocols.

In a NAS system, storage is directly connected to a network and presents industry standard network file system interfaces like NFS and SMB over TCP/IP for example. NAS devices provide a file level interface to the outside, and use a block level interface to either a tightly coupled (DAS) or loosely coupled (SAN) storage subsystems.

Several combinations of DAS, SAN, and NAS are possible, allowing different degree of dynamic reconfiguration, availability, and safety.

3.4 Changing Hardware

3.4.1 Introduction

Bus systems are for a long time part of hardware architectures and their core extension capability. Standardization of bus systems allows the independent development of peripheral components for computer systems and provided the model for software component systems as a side effect (“software bus”).

Configuration changes were less frequent with early computers because the environment and requirements were usually quite stable. Powering off a system for maintenance and extension was acceptable. However, with the increased mobility of the devices on the one hand and high availability systems on the other hand this situation started to change. For high availability, even the shortest downtimes are unacceptable so extension has to occur during runtime. Mobile computers are frequently used in different environments, with different resources and requirements resulting in numerous changes. This resulted in the development of hot-pluggable busses such as PC Card, PCI 2.0, USB, and IEEE 1394 that allow devices to be connected and removed without removing power from the bus. Evidently, operating systems had to reflect this hot-plugging capability and implemented dynamically loadable device drivers. The changes proliferate into the applications as well, which now have to deal with the dynamic availability of services.

The following sections will look in a top down manner at two operating systems handling dynamic hardware configuration changes. Then an architecture independent standard focusing on configuration and power management is presented. Three bus systems with hot-plug capability are discussed as well as an architecture that allows reprogramming the very heart of a computer system, the CPU. Finally, an ubiquitous computing scenario shows a different area of a changeable hardware infrastructure, interconnected most likely by a wireless radio “bus”.

3.4.2 Windows 2000 Plug and Play

Windows 2000 integrates so called “Plug and Play” already explored with the consumer versions of Windows, 95 and 98 [121]. The goal behind plug and play is increased reliability and availability, reducing the need to reboot the system to complete administrative tasks such as hardware and software installation [122].

Plug and play is a combination of hardware and software that enables a computer system to dynamically recognize and adapt to hardware changes with little or no user intervention. Windows 2000 provides plug and play support for:

- *Automatic and dynamic recognition of installed hardware* in response to runtime events such as dock/undock and device insertion/removal.
- *Hardware resource allocation* based on the device requirements such as I/O ports, DMA channels, and interrupt requests.
- *Loading of appropriate drivers* based on the device identification.
- *A driver interface* that allows drivers to interact with the plug and play system.
- *Access to the power management* features.
- *Application level events* for device notification.

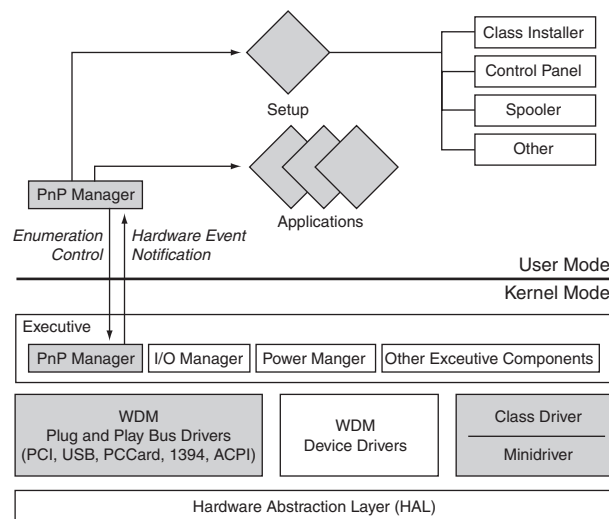


Figure 3.5: Windows 2000 Plug and Play integration (gray).

Figure 3.5 gives an overview of the parts involved in Windows 2000 Plug and Play.

- The plug and play *bus drivers* service a bus controller, adapter bridge or any device that has child devices. There is one bus driver for each bus present in the system like PCI, USB, PC Card, or IEEE 1394.
- Plug and play device drivers called *function drivers* provide the operational interface for a device. Usually a function driver is implemented as a pair of a class driver and a

minidriver. The class driver (typically provided by Microsoft) implements the common functionality required by all drivers for a class of devices and the minidriver (implemented by the device vendor) provides device-specific functionality.

- The *kernel mode plug and play manager* maintains central control, directing bus drivers to perform enumeration (scanning the bus for devices) and configuration. The plug and play manager also interacts with the device drivers to determine whether a device can be safely paused or removed.
- The *user part of the plug and play manager* maintains a device tree reflecting the hardware configuration. Configuration change events are reported to interested applications. Software installation (through Setup) is triggered for devices that do not have an installed driver yet.

Windows 2000 takes a centralized approach to configuration management, allowing configuration and power management decisions to be taken by considering the global state and the current dependencies of the system. In the light of the range and quantity of supported hardware, one has to admit that Windows 2000 plug and play performs well given the complexity of the task.

3.4.3 Mac OS X

Apple designed the hardware abstraction for Mac OS X from scratch, resulting in some interesting features. Similar to the class drivers of Windows 2000, Apple's I/O Kit defines several base classes for families of devices. Unlike Windows 2000, the I/O kit is C++ based and fully object oriented. Writing a device driver therefore basically means implementing or overriding methods defined in the base classes. See section 1.6.2 for a list of device families currently supported by the I/O Kit.

Central to the design of the I/O Kit is a modular, layered runtime architecture that models the hardware of a Mac OS X system by capturing the dynamic relationships among the hardware and software components involved in an I/O connection. The chain of interconnected services starts with the mainboard and extends through a discovery and matching process (see 1.6.2) the connection with layers of driver objects controlling the system buses (PCI, USB, IEEE 1394, etc.) and the individual devices attached to these buses. Mac OS X distinguishes between three basic objects:

- **Families** are abstractions that implement functionality common to all devices of a particular type. The I/O kit has families for bus protocols, storage devices, network services, and human interface devices.
- **Nubs** present access points and communication channels for a given protocol such as PCI, USB, or Ethernet.
- **Drivers** implement the functionality of a specific device or service, communicating through a nub with the hardware to perform I/O operations. Drivers inherit from a family and are in turn part of that family of devices.

The runtime features of the I/O Kit include dynamic loading and unloading of drivers as well as automatic resolving of a driver's software dependencies. The I/O Kit also generates

the events related to configuration changes and power management, similar to the kernel mode plug and play manager in Windows 2000. In addition, the I/O Kit maintains a dynamic database that records the graph of driver objects participating in hardware connections on a Mac OS X system and tracks the relationships among those objects.

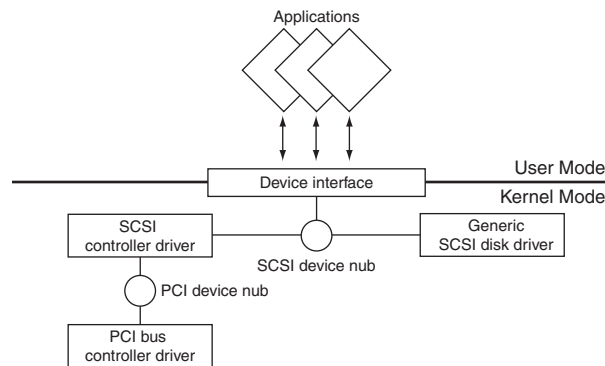


Figure 3.6: The relations for a SCSI I/O connection in Mac OS X.

A COM (see 2.5.3) compatible plug-in service provides driver access from user-space through a device interface. Figure 3.6 illustrates this relationship together with the nubs involved. For Unix compatibility, driver inodes in the `/dev` directory are dynamically maintained by the I/O Kit.

Thanks to its modern design, the I/O Kit provides many novel features for dynamic configuration like the device-driver matching process, automatic resolution of driver dependencies and matching directories defined in XML (see also 1.6.2).

Advanced Configuration and Power Management Interface

The advanced configuration and power interface (ACPI [32]) specification was developed to establish interfaces enabling operating system directed mainboard device configuration and power management of the entire system.

The principal goals of ACPI are:

- Enable all computer systems to implement mainboard configuration and power management functions, using appropriate cost/function tradeoffs.
- Enhance power management functionality and robustness over advanced power management (APM) and similar solutions.
- Facilitate and accelerate industry-wide implementation of power management.
- Create a robust interface for configuring motherboard devices.

Figure 3.7 gives an overview of the integration of ACPI in a system. Parts not directly related to ACPI are the device drivers controlling the hardware, the ACPI driver of the operating system, the kernel and the operating system directed configuration and power management (OSPM) part as well as the application talking to the drivers. The following parts are specified by ACPI:

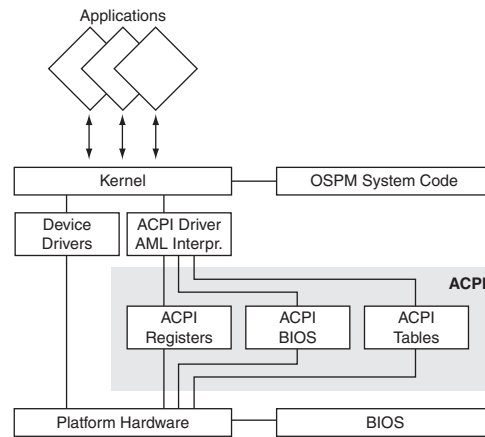


Figure 3.7: The system integration of ACPI.

- **ACPI Registers** are the part of the hardware interface described by the ACPI system descriptor tables.
- **ACPI BIOS** is the portion of the firmware that implements the ACPI specification. This code is usually used for testing and booting the machine, and providing interfaces for sleep, wake up, and some restart operations. The ACPI BIOS is also responsible for building the system description tables.
- **ACPI system description tables** describe the interface to the hardware. The description tables contain definition blocks that may use sequences of ACPI machine language (AML) code to describe operation for making the hardware functioning. AML is interpreted by the operating system (through an AML interpreter that is part of the ACPI driver in general).

ACPI provides plug and play support through device enumeration and events. In addition, ACPI also defines power, thermal, and acoustic management.

ACPI is an interesting approach to dynamic system configuration in the sense that it is operating system independent. ACPI system description tables together with AML instructions allow an operating system to discover and configure a system without any a-priori knowledge of the underlying hardware. ACPI receives wide industry acceptance and is implemented by all major PC BIOS vendors.

USB and IEEE 1394

The universal serial bus (USB [31]) was originally developed with the goal of defining an external expansion bus to simplify adding peripherals. Attaching an USB device should be as easy as plugging in a phone. Technically, the USB is a host-controlled serial bus that supports up to 127 devices with three data rates (1.5 Mb/s, 12 Mb/s, 480 Mb/s³). The key architectural parts of the universal serial bus are:

- **USB interconnect:** The USB interconnect is the manner in which USB devices are connected and communicate with the host. The interconnect defines the bus topology,

³The 480 Mb/s data rate is only supported by USB 2.0 compliant devices.

the relationship of the different USB protocol layers, the USB data flow model, and the USB schedule which arbitrates access to the interconnect. The bus topology is a tree with the USB host as its root, USB hubs as nodes and USB functions as leaves. Figure 3.8 shows a sample USB configuration.

- **USB device:** An USB device presents a standard interface for reset, configuration, and descriptive information. An USB device is either a hub or a function. An USB hub provides additional attachment to the USB and an USB function provides capabilities to the system such as a human interface device or a modem.
- **USB host:** There is only one host in a USB system, which implements the host controller in hardware and software and also a root hub which provides attachment points for additional hubs and functions.

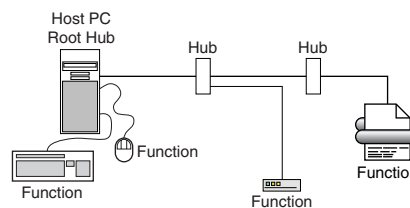


Figure 3.8: An USB configuration with three hubs and four functions.

The USB is a polled bus where the host controller initiates every data transfer. Communication is transaction oriented with an initial packet describing type and destination of the transaction, the data exchange, and an acknowledgement.

Attachment of devices to the USB is detected by polling the hubs that act as attachment points. When a device is attached, it receives a unique address during an ongoing activity called “enumeration” which also handles removal of devices. An USB device is required to provide information about its vendor, the device class, and power management capabilities. Additional information is standardized for each device class, which can be extended by vendor specific features.

The IEEE 1394 serial bus, also known as FireWire or iLink provides similar plug and play capabilities as the USB. The main difference from the user perspective are the data rates (IEEE 1394 supports data rates from 50 Mb/s up to 3.2 Gb/s⁴), only 63 instead of 127 devices, and the possibility to make peer to peer transfers without the need of a “host controller”.

Under the hood, IEEE 1394 is quite different from USB. IEEE 1394 has a memory-bus like logical architecture with an IEEE 1212 compliant addressing scheme. IEEE 1212 defines a fixed 64-bit address space with well defined control and status registers⁵ for a range of device classes.

Because IEEE 1394 has no central control, a dynamically selected root node detects bus modifications. Upon a bus reset, the physical topology is transformed into a logical tree, which

⁴The initial revision of IEEE 1394 only supported data rates up to 400 Mb/s.

⁵The structure of a IEEE 1212 address is:

10 bits	6 bits	48 bits
0 - 1022: bus number	0 - 62: node number	device memory, registers and ROM
1023: local bus	63: broadcast	

implicitly also elects the root node. Every node then assigns itself a node identifier derived from its position in the tree and negotiates speed capabilities with its neighbors.

Both buses support dynamic configuration but USB requires a controlling instance where IEEE 1394 does true auto configuration of devices. IEEE 1394 allows higher data rates but lacks the power-management capabilities of USB. However, both support similar qualities of service and transfer features making them fast, reliable, and easy to use serial buses.

PC Card and CardBus

IEEE 1394 and USB are *add-on* solutions; peripheral devices are externally connected to a computer system. PC Card and CardBus in contrast are *add-in* solutions; devices disappear inside the computer system.

PC Card and its successor CardBus were one of the first steps towards dynamic hardware configuration. Mainly found in portable devices (laptops and personal digital assistants), both implement a hot-pluggable version of the standard desktop buses with a few additions. PC Card implements an industry standard architecture (ISA) like bus and CardBus is similar to peripheral component interconnect (PCI).

The advantage of this approach is that drivers are easily portable from desktop systems to mobile systems. Once the card is inserted into the system, the hardware behaves close to the desktop equivalents. Usually the operating system takes care of the dynamic configuration of the system, detecting changes and dynamically loading and unloading drivers.

It is interesting to note that CardBay, the successor of CardBus, will integrate either IEEE 1394 or USB 2.0 [158]. This will unify a computer system's add-on as well as add-in run-time extension capabilities, leaving as main difference the form-factor.

3.4.4 PACT's XPU128

Whereas the previous systems allow the user to change the computer system's peripheral device configuration, parallel array computing technology (PACT) delivers with its XPU128 a reconfigurable CPU [68, 148].

PACT groups processing array elements in processing array clusters (PAC) of which the XPU128 has two each comprising 64 processing array elements. Each PAC has private memory, I/O, and a configuration manager. The configuration manager manages the dataflow between the processing array elements. The processing array elements consist of an arithmetical logical unit (ALU), a configuration control unit, an input, and an output register file. Array elements are interconnected by a programmable configuration, data, and event bus. Figure 3.9 gives an overview of the PACT architecture. By configuring the interconnecting bus as well as the processing elements the processor can be optimized for specific operations. Reconfiguration can occur at high frequency, allowing dynamic reallocation of CPU resources on a per-process base.

Currently only a hardware description language supports the mapping of algorithms to the PACT architecture, but a C compiler is under development.

Reconfigurable hardware has many advantages over a general-purpose architecture. Mapping algorithms directly to the hardware might enable multimedia processing even in low-power mobile devices or allow specialized high performance applications not possible with current general purpose CPUs. However, beside the programming issues, configuration and hardware allocation are non-trivial and will be a challenging task for future operating systems.

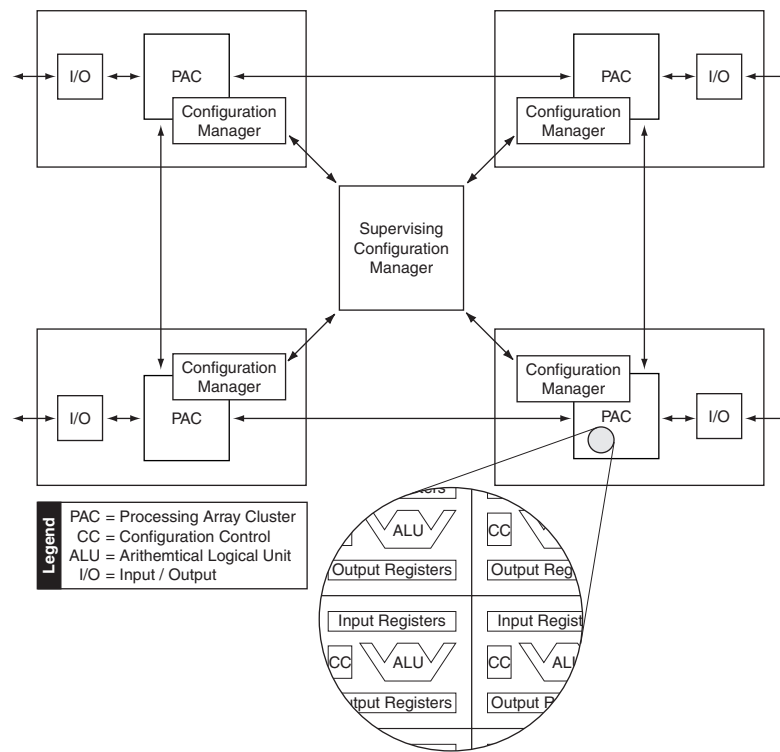


Figure 3.9: The PACT architecture.

3.4.5 Ubiquitous Computing

Ubiquitous computing (see also 1.6.4) is one of the most dynamic environments emerging today. Dynamism mainly appears in two dimensions:

- **Services:** The set of services implemented in a ubiquitous environment is constantly changing and growing. As ubiquity implies, computing will be part of our daily live and integrated in almost every physical object around us. Looking at the plethora of tools we are already using today, ubiquitous computing is likely to mirror this situation with many services. The same device may run different services, thereby changing its role. A device may for example act at one moment as a communication terminal and in another moment as a router.
- **Context:** Not only the set of services but also the composition of services in a given (spatial) context like an office, a conference room or a whole building is constantly changing. This is mainly due to the mobility of users and devices but may also be a result of the power management of individual devices, bandwidth or transmission quality constraints, etc.

Several technologies like wireless local area networks (WLAN), spontaneous networking (e.g. Bluetooth [66, 67]) as well as roaming in mobile networks like the global system for mobile communications (GSM) or the universal mobile telecommunications system (UMTS) attack the communication problems involved in a wireless communication environment.

But communication is only one of the problems. Since ubiquitous computing aims at seamless integration of devices in our daily live, making the environment “smart” requires

intelligent collaboration or composition of larger systems. This task is complicated by the fact that no central authorities are available, neither on the user side (such as a system administrator) nor on the system side (such as an operating system). It is the task of every single service to configure the environment and to talk to other services in order to realize itself. Finding suitable instances in a large, continuously changing search space becomes a challenging task. To make the ubiquitous computing vision a reality, an appropriate middleware derived from a suitable model focusing on these dynamic aspects should help applications solving these tasks. Chapter 4 proposes such a model and some prototype implementations are presented in chapter 5.

3.5 Summary

Dynamic configuration appears on various levels in computing systems and the overall trend is clearly towards more dynamism. On the one hand are systems that have to be reconfigured at runtime because service provision is so important that downtime is not acceptable, on the other hand are systems which for ease of use or changing environment conditions have to adapt while running.

Systems like the WOS with versioning and education, and Jini with dynamic discovery and use of service, are specially designed to provide service abstractions in a changing environment like the Internet. Harness takes yet another approach by proposing a distributed virtual machine as abstraction having services as plug-ins perceived by the application as machine extensions.

But not only these novel systems have to deal with dynamism. With the introduction of multiprogramming, problems such as resource reservation, sharing, and access or transient and persistent memory usage had already to be solved by earlier operating systems. Today's operating systems use well known models and algorithms for these problems but unfortunately most of them are based on centralized control and thus do not scale to networked environments.

A new aspect of dynamic configuration was introduced with changing the peripheral hardware configuration at runtime and maybe in the future not even the configuration of the CPU may remain static. These developments require a tighter interaction of operating systems and applications. Applications can no longer rely on a guaranteed level of services and treat the absence of certain functionality as error. Users expect applications to degrade gracefully in such cases, disabling certain features for example.

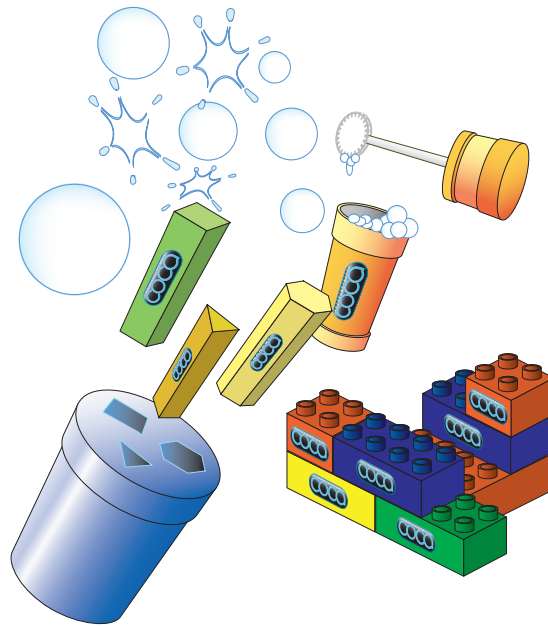
Most current systems rely on centralized solutions with their inherent shortcomings, therefore new ways have to be found in order to tackle the dynamic configuration problem. Jini, Harness, and WOS show how future decentralized systems may work.

Part II

Model and Implementations

Chapter 4

The COCA Model



4.1 Introduction

This chapter presents COCA (**C**oncepts, **O**ntologies, **C**lassifiers, and **A**ctions)¹, the key result of the research behind this dissertation. It started with the WebRes project [95], (see also 5.2) over four years ago. The goal of WebRes was to make local resources Internet-wide available with a focus on the user interface. Because WebRes intended to use the WOS for resource management, little attention was paid to resource lookup and access. After the WebRes project ended, the WOS was still not available and a new, still WOS independent successor addressing the shortcomings of WebRes followed. The contribution of the follow-up called WebCom [142] was the clear separation between component providers, service directories, and transaction

¹The COCA acronym is only one half of the story. Coordination language research at the DIUF resulted in various variants of CoLa (for **C**oordination **L**anguage). Because these languages usually deal with high-level abstractions, COCA is one way of providing them these abstractions through classifiers. This puts COCA in front of CoLa (with the obvious outcome of some well known liquid).

based resource access. WebCom used attribute sets to describe and lookup resources, similar to Jini (see 2.5.5) as well as a rule-engine enforcing relations between certain attributes of resources. Section 5.3 describes WebCom in more detail.

The key problems discovered with WebCom were heterogeneity and configuration, leading to the following three questions:

1. How can communication among services with different protocols be handled? (= Heterogeneity problem)
2. How can selection and composition of thousands of services be handled? (= Static configuration problem)
3. How can a changing environment, support for future services, and innovation be handled? (= Dynamic configuration problem)

The first part of this work underlined the importance of these three problems and also hinted the source: lack of machine processable semantics. Although various interface description standards like Java interfaces (2.4.5), IDL (2.5.2, 2.5.3), and WSDL (2.5.6) exist, few go beyond interface signatures. This basically means that clients have to infer semantics from operation names (often in English), argument types (often cryptic), and documentation (often incomplete). A situation that currently humans can handle but machines are still out of luck.

In a world consisting of computer scientists only this would do no harm but environments such as ubiquitous computing and Web services rely on “smart” automatic configuration with little or no human intervention, it is therefore important to change this situation. It is also a necessary step to make the technology accessible to everyone and to render the vision of invisible and pervasive computing a reality. Since missing semantics was identified as the key problem, the primary goal of this thesis became adding semantics to interfaces, which finally resulted in the COCA model.

Clearly, COCA did not emerge out of the void. Beside being motivated by the problems discovered with WebRes and WebCom, the model itself borrows aspects from cognitive psychology models [147]. The motivation for looking at human cognition and perception was the observation that humans perform extremely well in heterogeneous and dynamic environments. The idea was thus to adapt existing models from cognitive psychology to computer systems, in the hope to obtain a model that helps dealing with heterogeneity and dynamisms in computer systems.

Ross Quillian [136] was the first who introduced *semantic nets* [152] as a way of talking about the organization of human semantic memory. Semantic nets are a memory of concepts, where each concept is in relation with other concepts. COCA extended this model by adding *actions* as transformations between concepts and *classifiers* for symbol grounding and replacement for sensory input.

Figure 4.1 illustrates the COCA model in the light of human perception. The world is perceived as sensory input, which is interpreted or classified. Classification associates concepts with the input, abstracting the input and giving it a semantic. Thus classification transforms the perception of the real world into a meaningful symbolic representation.

Actions transform between concepts, like “drinking” from a “full cup” to an “empty cup”. Concepts such as “full cup” can be matched through classifiers with the objects in the real world and define in turn the set of actions that can be applied to them such as “drinking”.

Having observation as part of the model implies that COCA is based on an empirical understanding of the environment with the hope to enable better service provision in complex

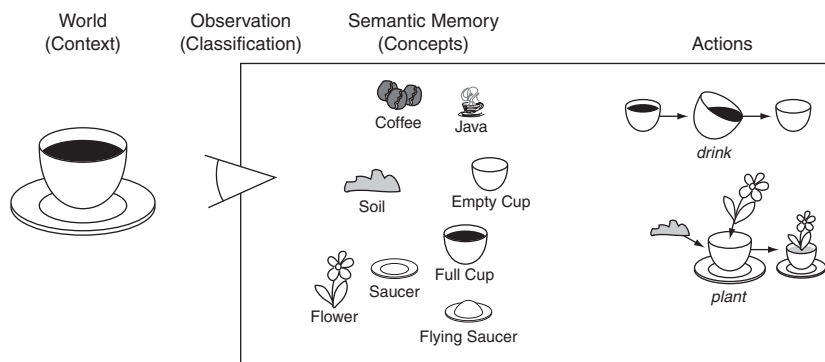


Figure 4.1: The basic idea behind COCA: The world is classified into concepts and actions are derived from matching concepts.

and dynamic environments. The trend from rationalism to empiricism [180], from complete understanding to observation and conclusion can be found in other science domains with high system complexity such as social science as well. This at least legitimates an empirical approach to computer systems.

The rest of this chapter is concerned with the COCA model and its implications. The next section introduces the environment abstractions of COCA, resources and contexts. Then follow the key notions of the model: ontology, relations, concepts, and classifiers. A discussion of the implications of the COCA model concludes this chapter.

4.2 Resources and Context

One of the starting points of COCA is its access to the environment. A knowledge base without “things” to reason about does not make much sense. “Things” are called *resources* (mainly for historical reasons) in COCA. Resources have a representation in computer systems (e.g. a name) and are accessible in some form (e.g. as a byte stream). Sets of resources are collected in a *context* that defines a namespace and the resource access. Example of contexts are the world wide Web with its URL [12] namespace and HTTP [48] as access, a database with its tables and columns as namespace and SQL [38] as access, the Internet domain name system (DNS [125]), a POP server [127], a file system, and that like.

In order to extend or restrict the perception of the environment, COCA allows some operations on contexts. A new, extended context can be constructed by the union of contexts and restricted by building a sub-context that contains resources of a specific concept only. These two operations are illustrated by figure 4.2.

Examples of restricting a context by a concept are a Web search engine that only returns Web pages that contain a certain keyword, a SQL “SELECT” statement, a segmentation algorithm of an optical character recognition (OCR) program, all JPEG files in a context, etc. Sub-context creation can be applied recursively, creating a hierarchy of sub-contexts where each level has a specific semantic, which is given by the restricting concept.

The motivation behind sub-contexts is that large and dynamic resource spaces can be restricted by an application to have only resources exhibiting certain properties in scope. An example might be a context browser that allows the user to see resources with a certain

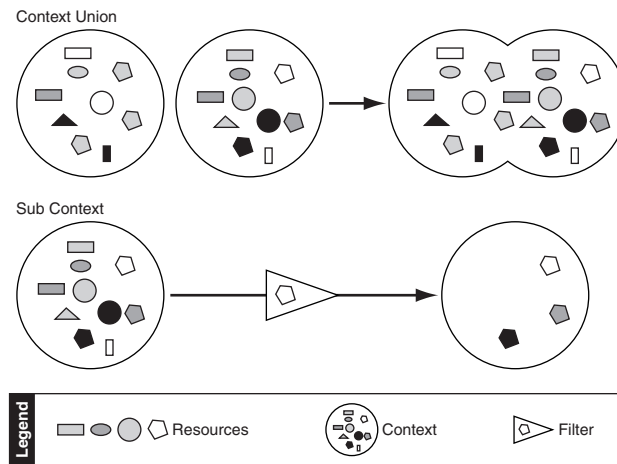


Figure 4.2: Union of contexts and sub-context construction.

semantic only such as “Java source written by me”, “five nearest color printers” or “emails from Joe”.

To summarize, resources and contexts have the following properties:

- **Resources** represent the “real world” perceived in a computer system.
- **Access:** A resource is accessible through at least one context, the context defines and abstracts the resource access.
- **Namespace:** The context that makes a resource accessible defines also a namespace for the resource.
- **Unique name:** A resource has a unique name in every context.
- **Immutable:** A resource is immutable, resource “changes” result in a changed name².
- **Context operations:** New contexts can be constructed from existing ones either by union or restriction.

The next sections will present classifiers that link resources with the COCA knowledge base, defining a symbolic representation in terms of concepts as well as their semantics.

4.3 Classifiers and Concepts

4.3.1 Semantics

The basic problem of semantics is their definition. Semantic does only exist with reference to an interpreter (operational semantics), a set of axioms (axiomatic semantics), a translation (translational semantics), a significance function (denotational semantics) [116], or a similar mechanism. KIF (see 1.2.4) for example defines a predicate calculus that gives meaning to

²Because actions (see section 4.5 below) are purely functional, no state change of the manipulated objects (resources) is allowed. The immutability property also helps to provide versioning where every version of a resource can be clearly identified and is guaranteed to remain unchanged.

KIF sentences. Because KIF and similar languages aim at “describing things in computers” [57], the foundations have to be as open as possible. One reason why these systems are not in widespread use is that defining semantics even for small domains is a complicated task. This situation may change when “semantic-libraries” one day become available that contain definitions for certain domains that can be customized, but currently this is not the case.

To escape this “define everything from scratch” problem, COCA has a very simplified yet powerful model of semantics based on classifiers.

4.3.2 Classifiers and Concepts

Instead of introducing its own semantics, COCA only defines the internal representation (concepts) of semantics and the relation to the environment (context and resources) through classifiers. Classifiers and concepts are a solution of the *symbol-grounding problem* [140], a problem intrinsic to manipulation of a symbolic representation. The symbol-grounding problem initially appears when one studies the relation between the real world and a system that uses some form of model or representation of reality to make decisions about a real world problem. The key question is how do these entities relate to each other. Can the symbols manipulated by the computer systems acquire an intrinsic meaning related to the problem?

The “real world” in COCA consists of resources that are related to concepts (symbols) by classifiers, which give the intrinsic meaning to the resources. More specifically, classifiers in COCA are functions of the form $R \mapsto C$, $R \mapsto \langle C, \mathbb{R} \rangle$, or $R \mapsto \langle C, S \rangle$ where R denotes the set of all resources, C the set of concepts defined by the classifier, \mathbb{R} the set of real values, and S the set of character strings. Every classifier defines the set of concepts C it is able to classify, giving implicitly the semantic to each concept by its execution. Figure 4.3 illustrates the operation of a classifier taking a resource as input and associating the concepts the resource is instance of with the resource in question.

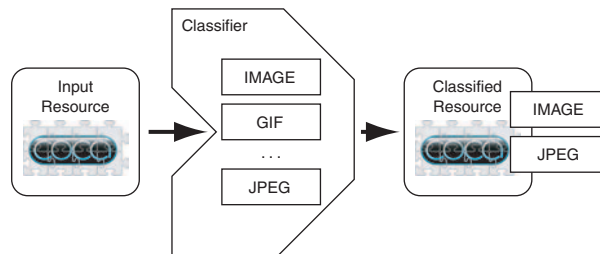


Figure 4.3: Classifier operation.

In the simplest case ($R \mapsto C$, figure 4.3) a classifier only tells of which concept a given resource is an instance of. For example a classifier that defines a set of MIME types may state that a given resource is an instance of the concept “image/jpeg” if the resource is a JPEG image.

The $\langle C, \mathbb{R} \rangle$ and $\langle C, S \rangle$ classifications are not strictly necessary but since numbers and strings represent widely used objects they are integrated into the model. The $\langle C, \mathbb{R} \rangle$ classification can be seen as associating a weighted concept with a resource ($R \mapsto \langle C, \mathbb{R} \rangle$). For example a classifier able to classify colors could assign weights in the interval $[0..1]$ for the concepts “red”, “green”, and “blue”.

Classifiers mapping resources to $\langle \text{concept}, \text{string} \rangle$ pairs ($R \mapsto \langle C, S \rangle$) are used to extract

textual information stored in resources like names or document content. A classifier for paragraphs for example may associate the concept “paragraph” and its content with a resource.

Because classifiers define concepts by their execution, every concept has an exact semantic provided the necessary resources for the classifier execution are available. The rationale behind restricting classifiers to three types of functions is the observation that a lot of semantic is already available in one of these forms. For example does it not make much sense to redefine the semantics of the concept “C-source” when there are C-compilers available that can easily decide if a file in question really contains C-source code or something else. Thus building a classifier for the concept “C-source” means simply invoking a C-compiler and looking at its exit code. This leads on the one hand to direct reuse of semantics and knowledge, encoded and available in computer programs today and it helps on the other hand constructing new semantic out of the existing pool of programs.

Classifiers have some interesting practical properties. For example they can be seen as test software assuring certain properties of a resource before using it, helping the construction of robust software. Because mapping from existing software to classifiers is relatively simple (for example by looking at the output of a program), a pool of classifiers is quickly constructed. If multiple applications share the same pool of classifiers, mutual understanding on a semantic (concept) level between the applications is possible; all applications share implicitly the semantics of the concepts defined by the shared classifier pool. If classifiers are written in a platform independent language like Java, applications may even “learn” new semantics by downloading appropriate classifiers.

4.4 Ontologies and Relations

Knowledge representation uses various terms with different domain specific definitions in order to describe the knowledge model. To help clarifying the terminology, five frequently found terms are explained here:

- **Taxonomy** (from Greek “taxis” meaning arrangement or division and “nomos” meaning law) is the science of classification according to a pre-determined system, with the resulting catalog used to provide a conceptual framework for discussion, analysis, or information retrieval. In theory, the development of a good taxonomy takes into account the importance of separating elements of a group (taxon) into subgroups (taxa) that are mutually exclusive, unambiguous, and taken together, include all possibilities. In practice, a good taxonomy should be simple, easy to remember, and easy to use.
- **Semantic nets** are a representational format that permits the “meaning” of words to be stored, so that humanlike use of these meanings is possible. Semantics nets are graphs having concepts as their nodes and relationships as edges. Semantic nets were first introduced by R. Quillian as a human cognition model for semantic memory.
- **Semiotics** (from Greek “semeiotikos” meaning the observant of signs, “semeiousthai” meaning the interpretation of signs, and from “semeion” meaning sign) is a general philosophical theory of signs and symbols that deals especially with their function in both artificially constructed and natural languages and comprises syntax, semantics, and pragmatics.

- **Ontology** is the study or concern about what kinds of things exist - what entities there are in the universe. It is derived from the Greek “onto” (being) and “logia” (written or spoken discourse). It is a branch of metaphysics, the study of first principles or the essence of things. In information technology, an ontology is the working model of entities and interactions in some particular domain of knowledge or practices. In artificial intelligence (AI), an ontology is, according to T. Gruber [64] “the specification of conceptualizations, used to help programs and humans share knowledge.” In this usage, an ontology is a set of concepts - such as things, events, and relations - that are specified in some way (such as specific natural language) in order to create an agreed-upon vocabulary for exchanging information.
- **Epistemology** (Greek “episteme” meaning knowledge and “logos” meaning theory) is a branch of philosophy that addresses the philosophical problems surrounding the theory of knowledge. Epistemology is concerned with the definition of knowledge and related concepts, the sources and criteria of knowledge, the kinds of knowledge possible, and the degree to which each is certain, and the exact relation between the one who knows and the object known.

COCA uses the term *ontology* for its knowledge base because its not taxonomy based on a fixed set of relations nor are COCA ontologies pre-defined. But classifiers are likely to use a taxonomy to internally represent the concepts they define. The COCA knowledge base is very similar to a semantic net, but unlike semantic nets, concept semantic is defined by external classifiers and has an extensible set of relations. The COCA knowledge base is clearly not an epistemology, which studies the nature, origins and limits of knowledge and is not a knowledge representation by itself. A COCA ontology is structured knowledge of interrelated concepts. Concepts have well defined meaning given by classifiers that associate concepts with resources.

Knowledge representation may range from very domain specific to holistic knowledge. Various projects try to construct a holistic ontology such as the OntoWeb (www.ontoweb.org) or the SENSUS [86] project. A holistic ontology often resembles an encyclopedia [114], something used for centuries by humans. In practice, representation of encyclopedic information in a machine processable form turned out to be difficult, this is one reason why COCA focuses only on applications in computer systems such as Web services and ubiquitous computing and is not aiming at a “one-for-all” solution.

Although a holistic ontology is not per-se excluded by COCA, a domain-specific approach is favored. Every application based on COCA usually defines its own ontology specific for the problem domain addressed by the application, keeping the ontology small and the application focused on the problem at hand. This implies local evolution of ontologies without global or semi-local coordination, nor adherence to some standard. In the case where inter-application communication is required, ontologies can be shared for example by:

- **Merging the concept name spaces:** A very simple solution which may nevertheless make sense for some domains, but it may also result in conflicting semantics for concepts having the same concept name.
- **Mutual resource classification:** Applications give mutual access to each other (context union) and each application classifies the resources of the other application according to the individual ontology.

- **Explicit declaration of concept equivalence:** Meta ontologies may be used to explicitly state equivalence of concepts in different ontologies. These meta-ontologies will require careful maintenance, likely done by humans.
- **Proving concept equivalence:** If two concepts in different ontologies are defined by the same classifier (see 4.3.2) they are considered equivalent allowing inference of further properties through each ontology's relations.
- **Inferring concept equivalence:** Concept equivalence may also be automatically inferred by observing classifiers by so-called meta-classifiers. That is, the automated version of building meta ontologies. Concepts may be considered equivalent as long as different classifiers consistently output the same concepts for a set of resources under observation.

Concepts in COCA are interrelated by relations that add inter-concept semantic. Relations in COCA are of the form $C \mapsto C$, where C denotes the set of concepts in an ontology. A relation may be either defined by enumeration (e.g. a database table), or by executable code (e.g. a program). Relations define by themselves their semantic through the enumerated concepts or the program execution. Examples of relations are the Java method `Class assignableFrom(Class)` that defines the “is-a” graph of Java classes. Another example is the “salary” relation in an employee database.

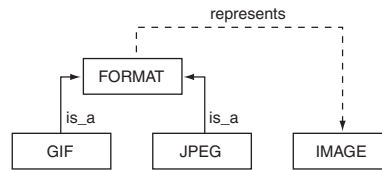


Figure 4.4: An example of an ontology.

Figure 4.4 gives another example of a simple ontology consisting of four concepts (FORMAT, GIF, JPEG, and IMAGE) and two relations (*represents*, dashed and *is_a*, continuous). The relation *is_a* relates the concepts GIF and FORMAT as well as the concepts JPEG and FORMAT. The relation *represents* relates the concept FORMAT with IMAGE. A classifier that recognizes file formats may define the GIF and JPEG concepts. Their relation with the other concepts defines the IMAGE and FORMAT concepts.

4.5 Actions

The previous sections mainly addressed the static aspects captured by an ontology. In a dynamic environment, resources not only exist, they are created, deleted, and manipulated.

In COCA, a resource manipulation is seen as a transformation of a resource, yielding a new resource. Such a resource transformation is called an *action* and is a relation of the form $R \mapsto R$ where R denotes the set of resources. Additionally, an action is constrained by the concept it accepts as argument (*input concept*) and the concept the result of the action is an instances of (*output concept*).

This requires a resource to be instance of the input concept of the action which produces an instance of the output concept as depicted in figure 4.5. The input concept acts as a

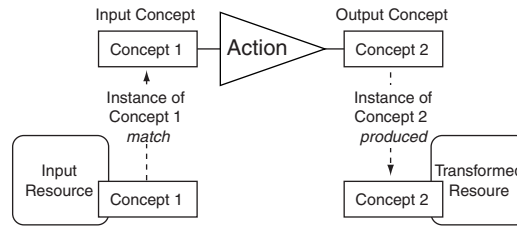


Figure 4.5: A resource transformation through an action.

precondition and the output concept as a postcondition as found for example in Eiffel's software contract (see also 2.4.6). To summarize, an action has the following properties:

- **Relation:** An action is a relation of the form $R \mapsto R$ where R denotes the set of resources.
- **Input constraint:** The input of an action is constrained by the input concept. An action accepts only a resource that is an instance of the input concept as argument. The input constraint defines the *source domain* of the action.
- **Output constraint:** The output of an action is guaranteed to be an instance of the output concept. An action only produces a resource that is an instance of the output concept as result. The output constraint defines the *target domain* of the action.
- **Stateless:** An action is stateless and side-effect free. If state has to be preserved, it has to be stored as part of a resource.

Actions can be linked when the output concept of an action matches the input concept of another action. Such a chain of actions is expected to execute under transactional semantics and appears to the application as an atomic action transforming from the input concept of the first action in the chain to the output concept of the last action in the chain.

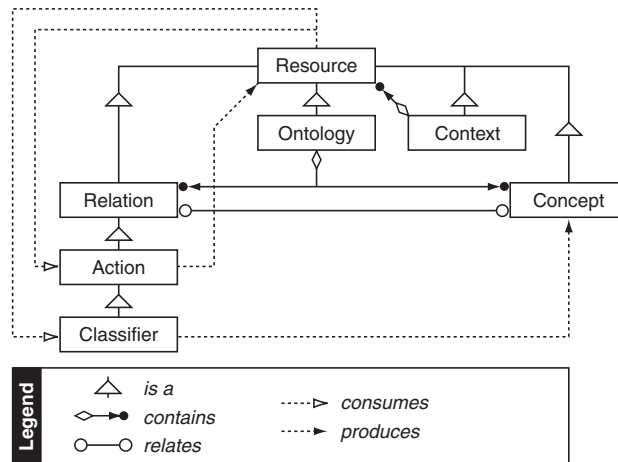


Figure 4.6: The COCA ontology.

Figure 4.6 summarizes the notions of COCA as a COCA ontology. Everything is a resource, contained in a context; an action is a specialization of a relation and a classifier is a specialization of an action. An ontology contains concepts and relations; relations relate concepts.

Actions and classifiers both consume resources. Actions also produce resources whereas classifiers produce only concepts. For simplification the $\langle C, \mathcal{R} \rangle$ and $\langle C, S \rangle$ classifications are omitted.

4.6 Implications

Examining the COCA model with the view from other models found in computer science reveals interesting mappings. Main programming paradigms like object oriented, functional, data flow, and logic programming map well to the COCA model. Furthermore, agent systems exhibit properties also found in COCA. The entity relationship model, which is the standard model behind of all today's relational database models, has also a mapping to COCA.

Other interesting implications of COCA are addressing by concept, semantic based service selection, and fault tolerance. The following sections discuss the mappings from the paradigms mentioned and other properties finally enabling automatic software configuration.

4.6.1 Object Oriented Programming

Object oriented programming maps quite well to COCA. Features found in object oriented systems are listed below and their correspondence in COCA is explained.

- **Class, type:** Object oriented programming is based on the concept of a class. A class (type) defines the set of attributes of an instance of the class and the methods that can be applied. Attributes form the state of an instance and methods allow modification of this state. Classes are organized in a class hierarchy, which defines the inheritance tree of classes.

Classes directly map to concepts in COCA and the inheritance tree consisting of “is-a” relations is easily represented within an ontology. The construction of a classifier that is able to associate a class with a concept is straightforward for object-oriented systems that support runtime type information (reflection, introspection) like Java, C++, Objective-C, CORBA, or Smalltalk. Systems not supporting runtime type information may require classifiers which access information extracted during build time or directly analyze compiled files.

- **Inheritance:** Inheritance allows classes to be organized in a “is-a” hierarchy. A subclass inherits methods and attributes from one superclass (single-inheritance) or several superclasses (multiple-inheritance). A superclass is a generalization of several subclasses whereas a subclass is a specialization of a superclass. Inheritance is the main mechanism that allows code-reuse in object-oriented systems.

Although a class-hierarchy can be easily modeled with a COCA ontology, it is not really necessary because attributes do not really exist in COCA and methods (actions) are associated dynamically. Nevertheless, an application may use the inheritance information in an ontology to select a specific action to implement polymorphism.

- **Instance, object, state:** Instances of classes, also called objects, encapsulate the state defined by a class that can be modified by methods. The state is defined by the values of the attributes of a class.

Instances in object oriented systems map to resources in COCA. The main difference is that resources obtain their “class” through classification instead of having a predefined association with a specific class.

- **Attribute, field:** Attributes represent the atomic parts forming the state of an object. Attributes may occur per class or per instance. Class attributes exist only once throughout all instances of a class whereas an instance attribute is part of the state of every individual instance.

There is no direct correspondence of an attribute in COCA. But attribute access is easily modeled within COCA. Classifiers can be used for simple attributes like numbers and strings. Complex attributes can be supported by appropriate actions.

- **Transient instance identity:** Every instance in a object oriented system has an identity that unambiguously identifies it, usually obtained during instance creation. Identities are often represented by a (memory-) reference, which are used to access the instance.

Although COCA identifies resources through a per-context namespace, the immutability property prevents a resource to “keep” its identity when its state changes. A solution to this problem is to partition the resource name into a static part that remains fixed for a resource and a dynamic part that changes with every modification of the resource.

- **Method, operation, message:** Methods implement the operations that can be applied to an instance or a class. Methods change the state of an instance or attributes of a class.

Actions in COCA provide the same functionality as methods with the advantage that they are associated at runtime with the instance in question through input concept matching. This means that the set of methods can be changed during execution in COCA, something rarely found in today’s object-oriented systems. This opens new possibilities for extension (e.g. upgrade) or restriction (e.g. security) of the set of operations at runtime.

- **Encapsulation:** Encapsulation means hiding of implementation details of attributes and methods. Encapsulation allows method implementations to be changed without changing the clients, providing some degree of data independence.

It can be argued if unconstrained change is a desirable property. COCA takes a quite different approach. Through classifiers a client only “sees” what it expects to see with the exact semantics of the concepts defined in the client’s ontology. In consequence, the degree of encapsulation in COCA depends only on the “insight” of the classifiers and can be freely chosen in a controlled manner.

- **Overloading and late binding:** Overloading means the redefinition of superclass methods in a subclass. Late binding is method selection at runtime depending on the instance in question and allows in turn polymorphism and independent compilation of classes.

Late binding is the only binding mechanism that exists in COCA. Overloading occurs in COCA when a resource is instance of several concepts that match with actions having the same output concept. If the ontology contains an inheritance tree it can be used to deterministically choose among the set of possible actions, following the “is-a” relations.

- **Instance storage:** Instance storage may be either transient or persistent. Transient means that an instance only exists during application runtime whereas persistence allows

objects to survive application termination. Object oriented systems provide varying degrees of persistence and sometimes provide versioning support for persistent objects.

If a COCA resource is persistent or transient depends on the context it lives in. Because COCA resources are immutable and the “type” is associated dynamically through classifiers, versioning is just a question of resource naming in COCA.

- **Naming:** Beside the transient object identifier, instances may also be referred by name. This is especially important in the case of persistent objects where at least for a root object an object identifier has to be obtained through a naming scheme.

Since a resource in COCA has only one name (defined by the context) there is no distinction between a transient and a persistent identification. Naming is entirely in the hands of the context.

- **Design by contract, assertion:** Design by contract [117] means a formal agreement between a class and its clients expressing each party’s rights and obligations. Eiffel for example realizes design by contract through preconditions, postconditions, and class invariants.

Classifiers in COCA can ensure arbitrary properties of a resource providing a base for a “contract” between a resource and a consumer of the resource. Actions for example ensure required properties by the input concept (precondition) and guarantee certain properties of the result by the output concept (postcondition).

4.6.2 Functional and Data-Flow Programming

In mathematics, for at least the last four hundred years, functions have played a central role. Functions express the connection between parameters (the “input”) and the result (the “output”) of a certain process. In each calculation the result depends in a certain way on the parameters. Therefore a function is a good way of specifying a calculation, which is the basis of the functional programming paradigm [74, 75]. A “program” consists of the definition of one or more functions. With the “execution” of a program the function is provided with parameters, and the result must be calculated.

Data flow programming [126] is just another way to see functional programming. Functions are the nodes of a network of interconnected inputs (parameters) and outputs (results). The main advantage of data flow programming is the easy to understand data-flow diagrams compared to deeply nested expressions.

An advantage of both is the natural parallelism [78] due to referential transparency. Because a function only depends on its inputs, every function can be evaluated independently of others.

It is also easy to make data-flow networks fault-tolerant [77]. Upon failure of either a connection or a node, the network can be automatically reconfigured to use alternative nodes and connections with the same semantics. Many languages are build on the functional or data-flow paradigm such as Haskell [79], Scheme [1], Guile [41], Lucid [178], GLU [78], or Strand [52].

The main properties of the functional paradigm and their mapping to COCA are:

- **Atoms or primitive expressions** represent the simplest entities in the system such as numbers, character strings, and primitive functions.

Resources are the “atoms” in COCA although they usually represent more complex data structures than numbers and strings. Specialization of resources such as classifiers and actions form the set of primitive functions.

- **Combination** is the mechanism to build complex expressions from simple ones.

Chaining of actions is the basic combination mechanism in COCA resulting in a data-flow network, which can also be seen as function application.

- **Means of abstraction** allow compound expressions to be named and manipulated as units.

No abstraction mechanism exists in COCA but applications are free to abstract data-flow networks and reintroduce them as named actions into the system.

More advanced functional features such as higher order functions, lazy evaluation, and pattern matching are also easily implemented with COCA.

4.6.3 Logic Programming

Logic programming is quite different from imperative or object-oriented programming: the program specifies a computation by giving the properties of a correct answer. Prolog [155] is an example of a logic programming language.

Logic programming languages are inherently “high-level” because they focus on the computation’s logic and not on its mechanics. Logic programming languages are usually based on an inference engine, which is already a powerful tool that can be exploited in developing inference engines specific to a particular universe of discourse or domain.

Logic offers the opportunity to represent data both extensionally (as an explicit fact) and intensionally (as a rule which implicitly describes the fact). The main properties of logic programming and their mapping to COCA are:

- **Axioms or facts** form a database of what is known true to the system.

Resources are the atomic “facts” in COCA.

- **Rules** state that a statement is true if certain stipulated goals and sub-goals are satisfied.

Concepts together with classifiers are simple rules that can be seen as predicates defined over resources. More complex rules can be stated with parametrizable classifiers, which may have a Prolog syntax for example. Another possibility is to encode rules as relations in an ontology.

- **An inference mechanism** allows the generation of sub-goals until they can be satisfied with facts.

There is no predefined inference mechanism in COCA but an inference mechanism can be easily integrated as an action. This has the advantage that multiple inference mechanisms can be used together.

The mapping of logic programming to COCA is the least obvious but simple declarative applications such as automatic software configuration (see 4.6.7) are easily implemented based on the COCA model.

4.6.4 Autonomous Agents

The question what constitutes an autonomous agent [143] is still open. Many definitions therefore exist. For this section the definition of [186] is adopted because it describes an autonomous agent in terms of its properties. These properties are mapped to COCA in the remainder of this section. Figure 4.7 illustrates the model of an autonomous agent with its specific features and their mapping to COCA in parentheses.

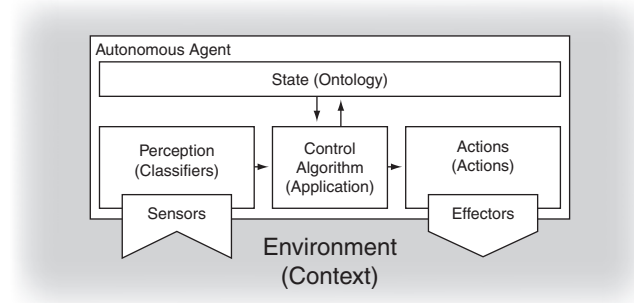


Figure 4.7: An autonomous agent and its mapping (in parentheses) to COCA.

- **Autonomy:** Agents operate without the direct intervention of humans or others, and have some kind of control over their actions and internal state.

Autonomy is out of the scope of COCA but may be part of the application.

- **Social ability:** Agents interact with other agents (and possibly humans) via some kind of agent-communication language.

Interaction in COCA is only possible through actions and classifiers. In this light a context can be seen as a coordination space where actions produce resources, which are then consumed either by classifiers or other actions. If interaction is seen as resource exchange, the ontology plays a central role because it defines the vocabulary used among agents.

- **Reactivity:** Agents perceive their environment, and respond in timely fashion to changes that occur in it.

Perception as depicted in figure 4.7 is modeled by classifiers in COCA, and the contexts represent the environment. It is up to the application to react to changes in the context.

- **Pro-activeness:** Agents do not simply act in response to their environment; they are able to exhibit goal-directed behavior by taking the initiative.

As with autonomy, pro-activeness is out of the scope of COCA but the ontology may be used to represent the agent's "mental state" to model the environment and the agent's goal-directed behavior.

4.6.5 Entity Relationship Model

The entity relationship model (ERM [29]) is the base for all current relational database modeling. Although the ERM is not a programming model, the mapping to COCA is straightforward.

The main advantage of having database models available in a COCA representation is that legacy databases can be easily integrated in applications based on COCA.

The main properties of the ERM and the mapping to COCA are:

- **Entity, entity set:** An entity in the ERM is “a thing that can be distinctly identified” [29]. Entities are either *weak entities* that only exist when some other entity exists or *regular entities* which exist independent. Entity sets group set of entities with the same properties.

Resources are the COCA representation of entities and concepts defined by classifiers form entity sets. Interestingly, Chen uses in its paper [29] the term “classified” to describe the process of constructing entity sets. The weak and strong entity properties are easily expressed in an ontology.

- **Properties:** Entities have properties which are common for all entities in an entity set. Properties draw their values from value sets also called domains.

There is no direct correspondence of a property in COCA. But property access of an entity is modeled within COCA by classifiers for simple properties like numbers and strings. Complex properties can be supported by suitable actions. Value sets are naturally represented by concepts.

- **Relationships:** Relationships are defined between entities and form relationship sets between entity sets.

Since concepts in COCA correspond to entity sets in the ERM, COCA relations directly correspond to ERM relationship sets.

- **Subtypes:** The ERM supports a type hierarchy by sub-typing which allows inheritance of properties.

As it was the case for inheritance in object oriented programming, relations in a COCA ontology can be used to model the type-hierarchy of an ERM.





ERM-Symbol	ERM Name	COCA Name
	Entity	Resource
	Entity Set	Concept
	Property	<Concept,S>, <Concept,IR>, Action
	Relationship Set	Relation
	Subtype	Relation + Concept

Figure 4.8: Elements of the entity relationship model and their mapping to COCA.

4.6.6 Addressing by Concept

Naming or addressing is a necessary step in order to access an object in a computer system. Addresses come in various forms containing different amounts of information. What is important about addresses is their *interpretation* or semantics.

The simplest form of addressing is *absolute addressing*. An absolute address contains all information necessary to locate an object. Examples of absolute addresses are memory pointers (interpreted by the CPU) or an URL (interpreted by the HTTP client and server).

Relative addresses contain only partial information, which have to be interpreted in conjunction with other information coming from the context. Examples of relative addresses are relative branches (interpreted by the CPU relative to the current program counter) or relative links in an HTML document (interpreted relative to the current document).

In order to make objects address-independent, *names* are introduced together with a mapping service (a *directory*) that contains the list of <name, address> pairs. This allows changing the address of an object without changing its name. Examples are the virtual memory management of an operating system that maintains the same logical view of an address space independent of the physical location (memory or stable storage) or the domain name system (DNS), which maintains domain names independent of the related IP addresses. Additional mapping layers may be added resulting in directories of <name, name> pairs often called *redirectors*.

A more advanced naming scheme are *queries*. Instead of accessing an object by its address or name, queries address a set of objects by attribute values. This allows gaining access to objects with certain properties without knowing their names, addresses, or even their existence. The big difference with naming is that a query may return zero or more objects, whereas naming usually is a one-to-one relationship. Databases for example implement special languages (such as SQL [38]) to formulate queries.

The shortcoming of queries is the lack of semantics. Before making the query, an agreement about the structure of the attributes and their meaning must exist. *Addressing by concept* solves this problem by providing a semantic based addressing. Instead of asking for objects with some attribute values, addressing by concept asks for objects with specific semantics. An example is the concept “nearest printer” which may result in very different objects depending on the location and the environment. COCA subcontext construction is an application of addressing by concept.

Evidently, addressing by concept may also be used with COCA to select actions. Instead of referring to a specific action by name, only actions with specific semantics are requested. Because actions are stateless and side effect free, replacing actions with identical semantics at runtime upon failure does not disrupt the application, resulting in a fault-tolerant system.

4.6.7 Automatic Software Configuration

Automatic software configuration means the automatic composition of software components in order to achieve some higher-level service. This higher-level service is only described by its properties but not how these properties can be provided. Automatic software configuration is therefore inherently declarative, the requested service exists only as a description and it is up to the system to provide a fitting implementation. Services in COCA are implemented by actions, so automatic software configuration means composition of actions. In fact, automatic software configuration itself is just another action that has as its input-concept a <concept, concept> pair describing the requested service and as its output an action representing the transformation between these concepts. The result might either be a simple action or a composition of multiple actions.

There are many ways to implement automatic software configuration. A Prolog-like inference engine may be one solution. Another much simpler is the action graph used by the

ubiquitous computing demonstrator (see 5.4).

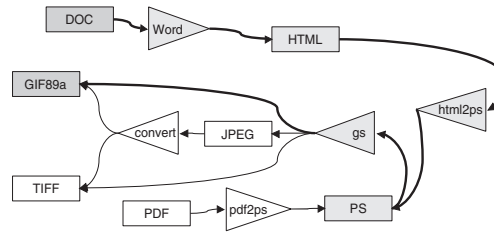


Figure 4.9: A graph of actions for document conversion, DOC to GIF conversion highlighted.

A simple action graph is depicted in figure 4.9. An action graph is constructed with actions as edges and concept as nodes. For a given <concept, concept> pair (e.g. <DOC,GIF89a>) the shortest path is computed within the graph resulting in a valid transformation between the concepts. Furthermore, weights can be added to the edges (e.g. communication or execution costs) to guide the shortest path construction and to take into account non-functional aspects.

4.7 Summary

The main contribution of COCA is the changed point of view in modeling software systems. Instead of explicitly constructing a given real or virtual environment within a software system, COCA applications have their own view (ontology) of the environment and try to match this view with the actual environment they are exposed to.

Environment access is handled by contexts, which hold resources, the atomic units COCA deals with. Classifiers are a solution for the symbol grounding problem because they give meaning to the conceptualized representation of the environment. The symbolic representation is captured by the COCA ontology, which consists of interrelated concepts. Actions allow dynamic and controlled interactions with the environment and define through input and output concepts a software contract for their use.

To close the circle, the initial three questions are answered by COCA in the following way:

1. How can communication among services with different protocols be handled? (Heterogeneity problem)

The application defines required properties through concepts which are then mapped to concrete resources in a context by classifiers. Based on concepts, the application can now deal with resources in a homogenous and abstract way.

2. How can selection and composition of thousands of services be handled? (Static configuration problem)

Ontologies allow to model any kind of dependency which can be used during deployment to ensure proper configuration. The domain specific approach of COCA together with subcontext construction ensures that an application only sees relevant resources.

3. How can a changing environment, support for future services, and innovation be handled? (Dynamic configuration problem)

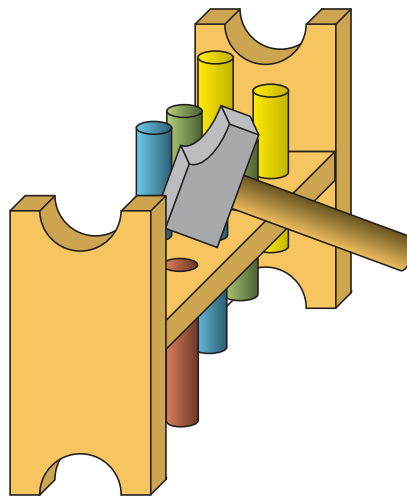
Because addressing by concept is based on semantics and not on predefined interfaces, a resource with required semantics is guaranteed to be found if available. This is especially

important in a changing environment; future services are automatically discovered if they provide at least the required semantics and innovation is handled in the same way. Automatic software configuration allows incremental extension of the system without disrupting existing services.

To conclude, COCA is a suitable model for providing service in a heterogeneous and dynamic environment such as Web services and ubiquitous computing. Integrating existing systems into COCA based applications should not be too difficult because several prominent models have mappings to COCA. The COCA implementation used for the ubiquitous computing demonstrator (see 5.4) emphasizes the practical value of the model.

Chapter 5

Implementations



5.1 Introduction

This chapter presents the various implementations of prototype systems developed during the research that lead to COCA. The development was initially driven by problems discovered in the WOS project that are inherent to computing on the Web. By addressing these problems, new questions emerged which finally lead to the COCA model. While working on the COCA model, research in ubiquitous computing started at the Department of Informatics at the University of Fribourg (DIUF). It was quickly recognized that ubiquitous computing has similar requirements and problems as Web computing. Having COCA ready, it was straightforward to put it to use in that area which resulted in the implementation of the ubiquitous computing demonstrator.

The following sections describe in chronological order the prototypes that lead to COCA as well as three applications based on the Java COCA implementation. WebRes was the first prototype realized, largely based on the ideas of WOS. The problems discovered with WebRes resulted in WebCom that made several conceptual contributions to COCA. Still, WebCom was only a prototype having the same shortcomings in terms of semantics as other similar systems around at that time such as Jini or Ninja. COCA mainly resulted as the outcome of the quest

of adding semantics to WebCom, but as can be seen from the previous chapters, the model may be applied for other systems as well.

The first implementation of COCA was done for the ubiquitous computing demonstrator, which shows the main aspects of COCA for an ubiquitous computing setting. The agent-based classifier is the subject of a diploma thesis based on COCA to explore the limits of the model in a more classic AI related application. Another application of COCA is currently on the way at Swisscom Corporate Technology using classifiers to enhance wireless application protocol (WAP) based services.

5.2 WebRes

Motivated by the idea of building a Web operating system, WebRes started at the same time and in parallel with the WOS project. It was the subject of my diploma thesis directed by Béat Hirsbrunner and Oliver Krone. Because the WOS project primarily addressed the resource lookup, access, and versioning questions, WebRes focused on other aspects such as user interaction and resource presentation. WebRes was realized with very simple resource management techniques that were planned to be replaced by the upcoming WOS.

The idea behind WebRes was to combine coordination theory with Web technology and use the Web as an interactive meeting point for resource sharing. At interface level, the user of WebRes accesses available resources such as files, CPUs, or whole applications via typical operations known from desktop user interfaces. The visibility of resources on the Web can dynamically be extended or reduced, therefore enabling collaboration functionality by changing the accessibility of formerly local resources onto a system-wide, global level.

Architecture

The Software Architecture of WebRes is depicted in Figure 5.1. There are three components involved: a resource set server as the implementation of a coordination space to manage shared resources, user interfaces for interaction purposes with the resource sets, and machine-local resource servers used to export local resources to the global Web. Note that local user interfaces and local resource servers can be used independently. A user benefitting from resources on the Web typically invokes a user interface only, whereas machines which provide certain resources automatically, that is without user intervention, may only run a local resource server.

Resources

As in COCA, WebRes has the notion of “resource” as an entity that can be manipulated by a given machine. In fact, the term “resource” persisted throughout WebRes and WebCom although it may be more appropriate to adopt a more general term such as “item” or “thing” for COCA. In WebRes, resources may have some physical or logical representations such as a printer or a file. But also abstract concepts such a CPU-power, that is a time dependent function, are considered as resources. Due to the heterogeneous nature, there exists no predefined common interface for a resource. WebRes already used the term “classification” for wrapping resources in Java classes but this was done by a programmer who defined the wrapper as an open set of properties and actions. WebRes actions were operations, which can be applied to a certain resource, or operations that the resource can perform by itself.

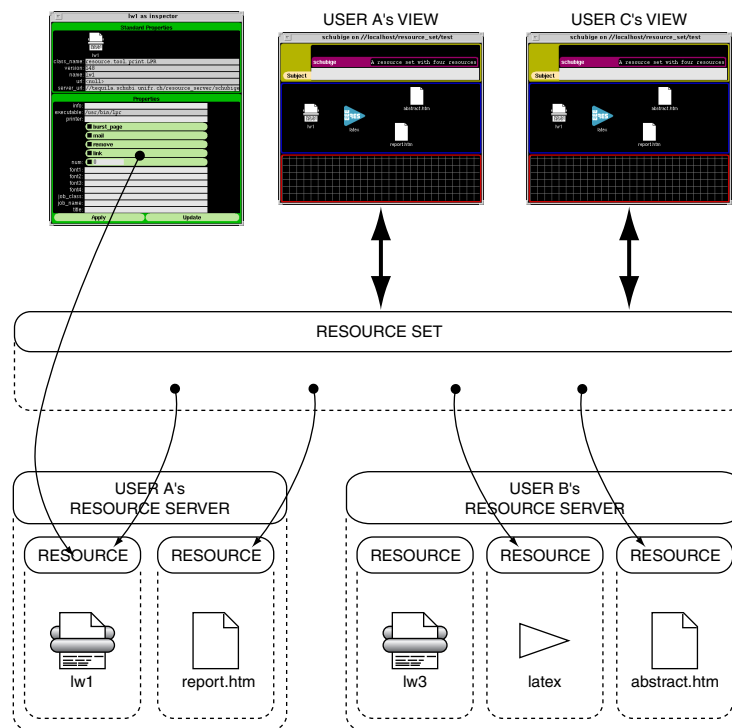


Figure 5.1: An overview of the interactions between a resource set, two resource servers and two user interfaces.

A printer for example was described by properties like the type and vendor of the printer. An action that may be applied to a printer is a status query and one of the operations that a printer can perform is printing a page.

It was easily discovered that these features map very well to an object oriented model. And this was the reason why a class hierarchy was used to organize the different kinds of resources, forming a first kind of “ontology”. Fields and methods implement properties and actions, respectively. Using Java as platform independent object framework for the wrappers turned the previously heterogeneous world of resources into a well organized, homogeneous world of objects sharing a common base interface.

The wrapper classes tried to preserve the full power and features of the native software and hardware. Since the object framework is open and extensible it was possible to integrate legacy software as well as new technologies.

Figure 5.2 shows a WebRes resource and its four key parts in more detail:

- A *presentation* provides an interactive interface to a resource on the user level. A resource has at least two standard presentations: “Iconic” and “Inspector”. Both of them are shown on the screen-shots (figure 5.3). The “Iconic” presentation is used to manipulate a resource similar to the file and program icons found in today's desktop interfaces. The “Inspector” presentations (shown on the right of figure 5.3) allows a user to view and change properties of a resource. The right hand side of figure 5.3 for example shows the “Inspector” presentation of the UNIX unzip tool. Generic implementations are provided for the standard presentations, but they can be extended or replaced by a particular resource. Resources can add additional presentations in order to provide a

RESOURCE	
Presentations (User Interface) -Iconic -Inspector -...	
Properties -read(<property>) -write(<property>,<value>) -set_readonly(<property>) -notification(<property>)	
Inputs (Methods)	Outputs (Stubs)
-...	-...

Figure 5.2: The WebRes resource interface.

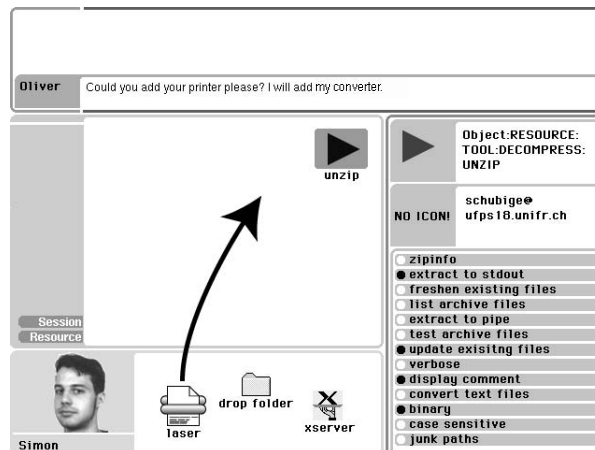


Figure 5.3: The WebRes user interface, the user is adding a printer to a resource set.

specialized user interface.

- The *properties* represent the state of a resource. The resource framework automatically collects all public fields of a class used to implement a resource. These fields are then published as properties and are accessible from the outside through the appropriate `read` and `write` calls. In addition, a property can be marked read-only and notifications allow a client to be informed upon a change of a property.
- The *inputs* of a resource are all public methods of a resource. Like the properties, the inputs and their corresponding calling interfaces are collected and published by the resource framework automatically.
- The *outputs* are stubs provided by the implementer of a resource. Outputs are typed in the same way as inputs. An output can be linked at runtime to one or more inputs if the interface matches. An execution of an output stub by a resource will result in a parallel execution of all linked inputs.

The resource framework used the runtime class information available through the Java reflection mechanism to collect and publish the features of a resource. Therefore, no extra work was required in order to describe and publish a resource besides inheriting from the `Resource` class.

Sharing

Locally available resources were managed by a resource server running on the machine where these resources were accessible. The resource server was responsible to publish the local resources and their interfaces on the network. A resource server might also act as meta-server. Instead of accessing a resource directly, it translates the calls to some other protocol such as the WOSP.

In order to allow sharing of resources, the local resources published through the resource server had to meet in a globally accessible place. Such a place is called a resource set. A resource set can be seen as a coordination space [88], a medium which enables sharing through the coordination of distributed applications.

As far as the implementation is concerned, a resource set can be located anywhere on a network and may be distributed and replicated. Resources can dynamically be added to and removed from a resource set. The resource set is responsible for tracking the location and the availability of resources. For example, if a local resource server breaks down, the resource set has to temporarily remove the resources, as they are no longer accessible.

The resource set is a network wide accessible directory of resources located on that network. Usually individual resource sets are created to solve a specific problem. This means that many resource sets exist simultaneously. Because resource sets are reused and modified over time, they are persistent.

Interaction

The success of the Web browser interface has shown the need for a graphical representation tool for resources on a network. Therefore, a graphical user interface permitted the interaction with a resource set. The user interface followed the well-known desktop metaphor.

Today's users know how to manage their local resources through a desktop-interface. This understanding was "recycled" to manage Web-wide resources in a similar way.

Beside the desktop interface, WebRes introduced a new feature not found in the current interfaces: resource linking. The pipe operator of the Unix shells is, although limited, an implementation of resource linking. WebRes allowed linking in a graphical manner and goes far beyond input/output redirection. Since all resources publish their interfaces, properties, inputs and outputs can be matched by type and interconnected arbitrarily. Resources can be linked together to form a data flow network by connecting their inputs and outputs. Such a data flow network acts as a transaction in a resource set and can be used for problem solving. Only basic support for resource linking was integrated into WebRes and stimulated further investigation of this feature in the WebCom project.

Conclusion

The approach of WebRes was inspired by coordination theory for interactive resource sharing on the Web. Everyone was able to share local resources with other users and shared resources were collected in a persistent resource set, which implemented a coordination space. Users contributing resources to a resource set were able to combine and use the resources in the resource set for interactive problem solving. Using the well-known desktop metaphor augmented by a powerful link technique to interact with the resource set facilitated collaboration of geographically dispersed people.

The object oriented "wrapper" technique provided a good way to integrate and to share heterogeneous resources on the Web. It is interesting to note that many features of COCA were already present in WebRes such as a "classification", actions and data flow networks, although they existed only as implementations.

5.3 WebCom

It was a direct step from the resource-linking feature of WebRes to a generalized data flow based system. The WebCom project as the successor of WebRes was constructed around the data flow paradigm, benefiting from the advantages of such networks in a distributed context:

- Data flow networks exhibit a natural parallelism. Because the components only depend on their inputs, every component can be executed independently of the others. The structure of the network maps well to today's network of workstations that have a similar physical structure.
- Data flow networks can be made fault-tolerant. Upon failure of either a channel or a component, the network can be automatically reconfigured to use alternative components and channels. Data buffers and stateless processing components ensure that a data flow can be restarted without loss of data.
- Data flow networks have a wide coverage in the literature. Entire languages and systems are built on the data flow paradigm [126] such as Lucid [178], GLU, [78], or Strand [52] just to mention a few.

- Data flow networks are used successfully in many real-world settings. Applications range from hardware (super scalar/pipelined CPU's) to tools (development environments) to business applications (online transaction systems).

Using a data flow network in the context of the WOS (Web Operating System [97]), was considered a suitable choice since the WOS requires a resource presentation and control interaction going beyond the current desktop systems [141]. In addition, the WOS provides distributed computing through its services which map directly to the nodes of a data flow network. Although the WOSP (WOS Protocol [7]) provides a basic means for formulating requests, a coordination mechanism is desirable in the context of the WOS. Processing requests which include several sub-requests and have specific constraints require help from an additional layer on top of the WOS.

Coordination on behalf of data flow networks was partially implemented in the WebRes project [141] and the results ¹ suggested to continue in that direction.

WOS and Object Flow Networks

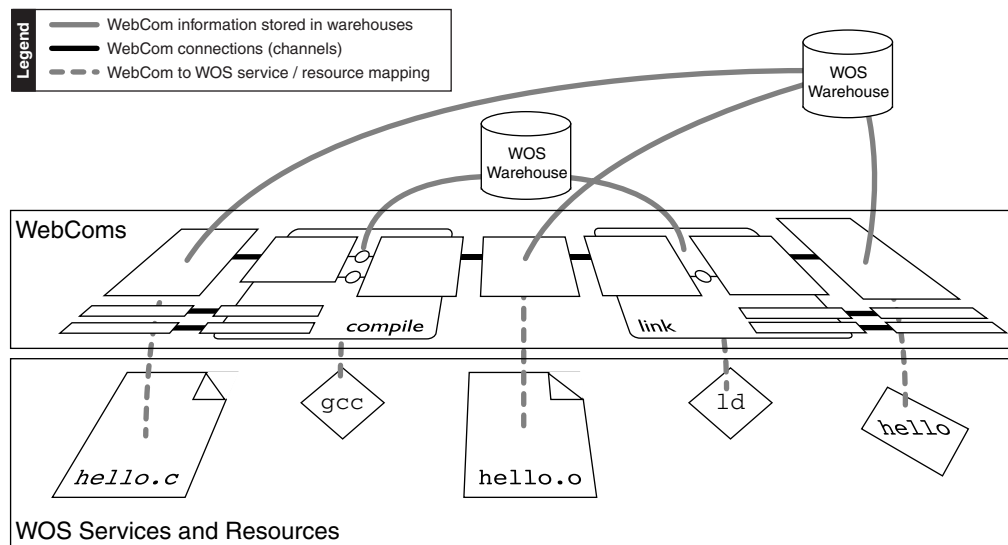


Figure 5.4: An example of a WebCom application and its relation to the WOS

An Object flow network is considered a graph with vertices called WebComs having *plugs* and edges called *channels*. Channels are created through the matching of compatible plugs in the WebCom space. WebComs are either active or passive and are attributed. A *passive* WebCom is specified by its *attribute scheme* only. Documents are typically passive WebComs. Every change in some set of attributes results in an object flow through the network. This data is processed by *active* WebComs which have an *action* in addition to their attribute schemes.

Figure 5.4 depicts the relations between the WOS and WebComs for a compilation scenario. Two active WebComs are shown, called `compile` and `link` which are connected to passive WebComs (files) represented by WOS resources `hello.c`, `hello.o`, and `hello` respectively. The active WebComs in this example are mapped to the WOS resources `gcc` and `ld`.

¹Graphically built networks are easy to understand for the user and have a direct mapping to WebRes resources and runtime linking [95].

Attributes Schemes and Objects

Because WebComs have real world representations such as programs and files, they represent some known concepts. These concepts are captured in *attribute schemes*. In this respect attribute schemes are similar to intentions in the intentional programming style [4, 150]. Attribute schemes serve as abstractions for these concepts. New attribute schemes are constructed by description or by composition using set operations. For example the attribute scheme of `hello.c` might be defined as `c-file`:

```
scheme c-file is file union unix-file union c-source;
scheme file is {
  name(Name) :- volatile, persistent, native(name, Name).
  length(Length) :- volatile, persistent, length(length, Length).
  path(Path) :- volatile, persistent, path(path, Path).
  content(Content) :- volatile, persistent, content(Content).
}
scheme c-source is {
  content(Content).
  denotation(Denotation) :- implicit, dependsOn(content);
}
scheme unix-file is {
  i-node(Inode) :- volatile, final, native(i-node, Inode).
}
```

The definition of an attribute scheme is either implicit such as in `c-source`, defined by a platform dependent (native) implementation like `unix-file` or a relation based on other attributes as in `dependsOn()` in `c-source`. Various relationships between attributes can be stated as well. For example one attribute might be just a synonym for another attribute or the value of an attribute depends on the value of another as in `denotation(Denotation) :- dependsOn(content)`. Having implicit attribute schemes without a definition makes sense because certain relationships between attribute schemes can be established without referring to a definition. For example what one normally expect from a C-compiler is that the object file has the same denotations as the source file. A C-compiler might then claim that the `denotation(Denotation)` attribute scheme is invariant under compilation. This property of a C-compiler can be stated without saying what `denotation(Denotation)` actually means. Such an attribute scheme is said to be *implicit* because there is an implicit agreement on its meaning.

Instances of attribute schemes are called *objects* and their parts are called *attributes*. The `hello.c` object might be

```
object hello is-a c-file {
  name(hello.c);
  length(3456);
  ...
}
```

Upon a change of one or more attributes of an object, a new instance of an attribute scheme is created, called a version, inheriting the unchanged attributes from the previous version of the object. The system maintains the resulting version history completely or partially depending on the versioning policy. Such an attribute change acts as a *transaction* which

usually causes several subsequent transactions initiating a data flow through WebComs. An example of a transaction is an editor that saves a new version of `hello.c`. This transaction will affect several attributes such as `content`, `length`, etc. This may trigger some actions like recompilation, which again cause new transactions changing attributes in object files.

The mapping between attribute schemes and the entities are stored in WOS warehouses as well as low-level platform specific information and higher level concepts such as files.

Object Flow Networks

An object flow network is a generalization of the data flow network in the sense that not only passive data items are allowed to flow through the network but also associated code. Having code together with data enables the system to use lazy evaluation techniques as well as remote access of attributes. Objects are dynamically constructed upon a change in some attribute set and flow through channels. Objects may be either associated with a data flow through some dynamic constructed entity such as a pipe or a TCP connection or may have representations such as an intermediate file and are also passive WebComs in that case. A WOS warehouse might also serve as an object store for channels that can not be represented by other means.

WebCom had already a form of automatic software configuration based on attribute scheme matching. Automatic software configuration in WebCom was declarative taking a set of WebComs as input and an attribute scheme as goal. By matching attribute schemes, the WebCom runtime constructed a data flow network that transformed the input WebComs to a WebCom with the specified goal attribute scheme.

For example a user specifies as input a passive WebCom, for example the file `hello.c`, with a set of appropriate attributes, and a goal WebCom, the executable file `hello`. Since the attribute scheme of the plug of a c-compiler is a subset of the attribute scheme of the source file, the two WebComs will be connected which results in an activation of the compiler. The compiler's output plug and the linker's input plug share the same attribute schemes and therefore the two plugs match. The final file is created as the result of the matching of the attribute scheme of the output plugs of the linker with the attribute scheme of the goal, the destination file `hello`. Note that a request may require certain attributes such as `denotation(Denotation)` to be invariant during the dataflow. This limits the set of possible active WebComs to c-compilers and linkers that claim the to preserve the `denotation(Denotation)` attribute.

Although never fully implemented in WebCom, the matching feature was the basic idea behind automatic software configuration in COCA. The matching approach is not new, the well known coordination language Linda [27] and the newer JavaSpaces [162] use a similar technique. But both of them lack the semantic power given by classifiers as well as the flexibility of the matching algorithm. Because matching is just another COCA action, any desired semantics can be supplied for automatic software configuration.

Conclusion

By pushing further the ideas of WebRes, WebCom sketched the base for COCA. Especially the addition of attribute schemes together with a definition language lead directly to classifiers and concepts. Due to the lack of a usable WOS implementation, WebCom was only partially realized. Thanks to its modular design, most of the code was reused in a COCA demonstration

application that implements a multi-model interface scenario in the context of ubiquitous computing, which is the subject of the following section.

5.4 Ubiquitous Computing Demonstrator

After the COCA model evolved out of WebCom, it had to be verified by an implementation to show its practical use. At the time COCA emerged, ubiquitous computing research started at the Department of Informatics at the University of Fribourg (DIUF) under the lead of Béat Hirsbrunner and Sergio Maffioletti. It was quickly recognized that similar challenges exist in ubiquitous computing as in Web computing. In order to accelerate the ubiquitous computing effort at the DIUF, the COCA model was used to implement a simple ubiquitous computing setting showing the heterogeneity, dynamic, and configuration aspects in a multi-model interface scenario.

The multi-modal interface scenario implemented by the ubiquitous computing demonstrator allows a user to establish relations between messages and devices, with a COCA based runtime automatically transforming the message to fit the current user's device. For example think of a user who associates a text file with a PDA and a phone. As soon as this file changes, the user receives a rendered version of the file on its PDA. If he is also available on the phone, the message is synthesized as voice, the user called, and the message is played over the phone.

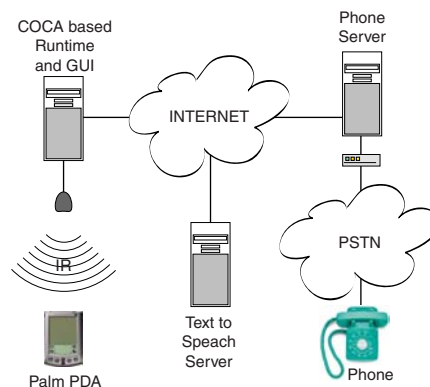


Figure 5.5: An overview of the ubiquitous computing demonstrator.

Figure 5.5 gives an overview of the ubiquitous computing demonstrator. The key parts of the implementation are:

- **COCA based runtime:** The runtime controls and interacts with the environment over the Internet and an infrared port.
- **GUI:** The GUI allows interaction and visualization of the operation of the COCA based runtime. Figure 5.6 shows a screenshot of the the GUI, visualizing different aspects of the model implementation.
- **PDAs:** A set of PalmOS based PDAs run a tiny runtime that allows them to be classified and exchange messages with the COCA based runtime over the infrared port.

- **Text-to-speech:** A text to speech server located somewhere on the Internet is used as an example of a Web based action.
- **Phone server:** The phone server allows the COCA based runtime to interact with the public switched telephony network (PSTN).

The following sections will look in more detail at how the ubiquitous computing demonstrator addresses the three aspects heterogeneity, dynamism, and configuration.

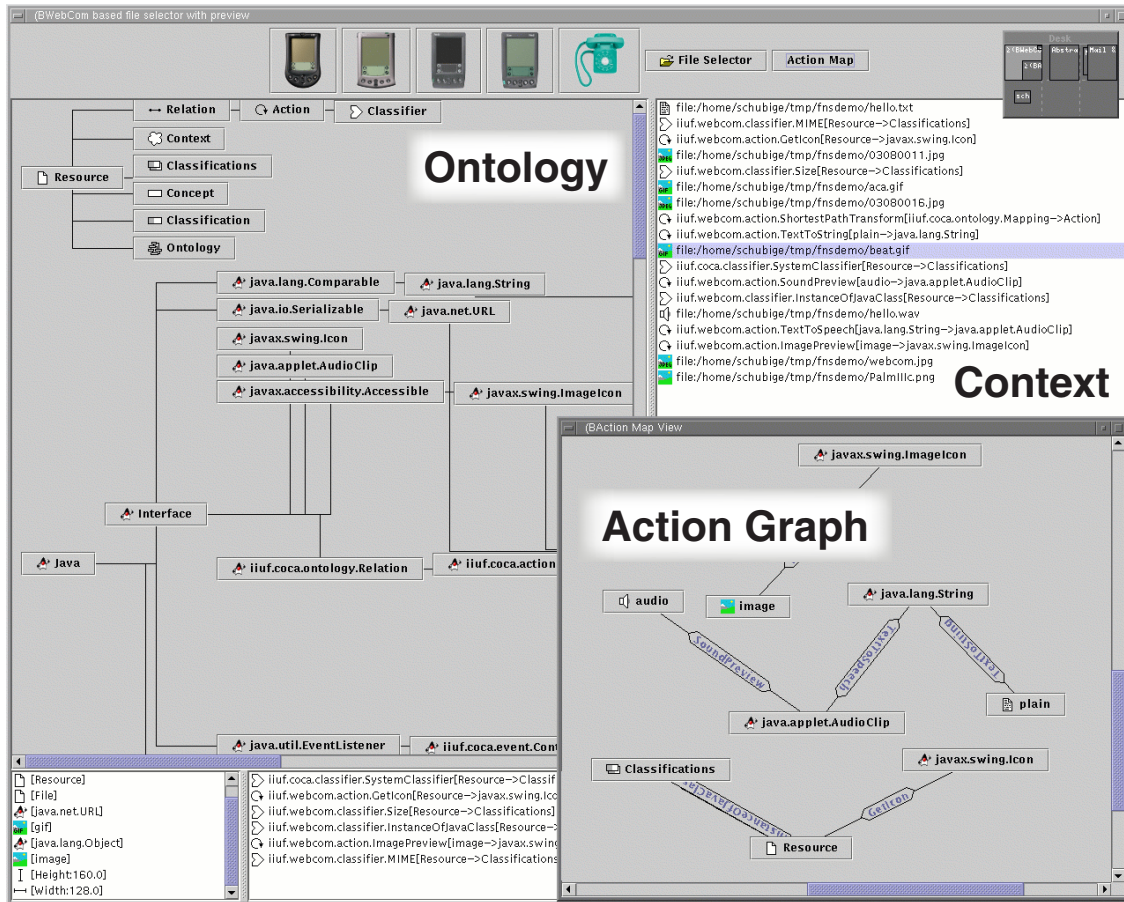


Figure 5.6: A screenshot of the ubiquitous computing demonstrator.

Heterogeneity

Heterogeneity appears basically on the hardware level in the ubiquitous computing demonstrator composed of PDAs, phones, and workstations. Only a tiny COCA runtime runs on the Palm PDAs whereas the full version runs under Java on a workstation. The text-to-speech converter is accessed by HTTP over the Internet and the PSTN integration is handled by a voice-enabled modem connected to another workstation.

Classifiers are used extensively to detect changes in the environment and to dynamically adjust the set of available resources in the contexts. The ubiquitous computing demonstrator implements a classifier for the various Palm models in order to figure out their hardware

capabilities (such as color or monochrome displays). Another classifier is able to locate an appropriate text-to-speech server on the Internet. Yet another classifiers associates MIME-types [22] with messages created by the user. For better integration with the Java environment, all Java classes loaded by the demonstrator end up in the ontology as concepts, again through a classifier.

Dynamism

Dynamisms in the ubiquitous computing demonstrator appears mainly in three areas: PDAs can be reached by infrared if within range, the text-to-speech service on the Internet may be up or down, and messages may be created. All the above items are resources in a corresponding context. In consequence, three contexts are implemented by the ubiquitous computing demo: one for what is reachable by infrared, one for the Web and one for the filesystem where messages are stored. Every context is responsible for tracking changes occurring in it and to provide context access to the COCA based runtime. The COCA based runtime unifies these three contexts and sees changes only in the unified context, hiding the peculiarities and access details involved by the specific contexts. Parts of the demonstrator only interested in particular resources such as the GUI use sub-contexts to see only PDAs for example.

Configuration

Configuration mainly consists in maintaining the action graph and providing automatic software configuration to the COCA based runtime. Maintaining the action graph is relatively simple; it is sufficient to create a sub-context with actions only. These actions then form the edges of the action graph and the input and output concepts of the actions are the nodes of the graph.

The automatic software configuration action implemented by the ubiquitous computing demonstrator is also very simple but sufficient for that case: for a $\langle \text{resource}, \text{concept} \rangle$ pair an action is returned that allows the input resource to be transformed to the requested concept. The action is found by computing the shortest path from all the concepts the resource is an instance of to the target concept. Then the shortest of all these paths is returned as an action. The returned action may be a composition of other actions (a path).

Conclusion

The ubiquitous computing demonstrator is the first implementation and application of the COCA model. The most important result of this implementation is that the model as such is realistic and indeed appropriate for ubiquitous computing settings. Another result was that the design of the application was largely simplified by thinking in COCA terms. Every problem encountered had its clear solution in COCA, helping the construction of the demonstrator in a relatively short time². Another interesting property of developing in COCA terms is that actions can be developed with very local scope. An action basically only has to know about its input and output concepts, the integration in the “big-picture” is done by the COCA based runtime and the application. This allowed for example to develop and integrate the text-to-speech service in a few hours and all parts of the demonstrator dealing with text were thereafter

²It took about 4 weeks to implement the COCA model and to build the ubiquitous computing demonstrator on top of it.

speech enabled without touching them. The positive experience gained with the ubiquitous computing demonstrator motivated other applications of COCA, which are described in the following sections.

5.5 Agent Based Classifier

To explore the limits of COCA beyond ubiquitous computing and Web applications, the agent based classifier project was launched as a diploma thesis realized by Frederic Delaloye. The goal of the agent-based classifier was to develop a COCA application for knowledge acquisition, representation, and management. The motivation of such a tool is that during research one comes across many documents, which increase the knowledge about the research subject. Knowledge continuously increases over time but it may also change direction, as research is not considered a linear process. Knowledge acquired later may change the relevance of documents read earlier.

The idea of the agent-based classifier was to provide a tool for the researcher that allows him to organize his knowledge and to have an agent that continuously classifies documents with that knowledge. Such a tool has two main uses: on the one hand it should allow to see per document the knowledge in the document in condensed form, for example the key concept discussed; on the other hand it should allow to search for specific documents based on a set of concepts.

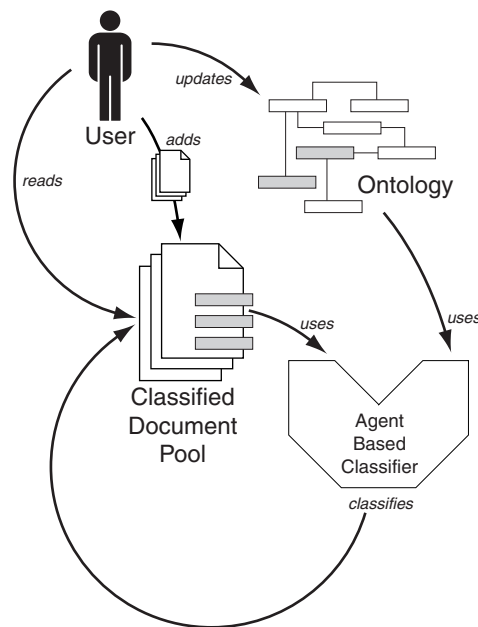


Figure 5.7: An overview of the agent based classifier system.

Figure 5.7 gives an overview of the operation of the agent based classifier. A user reads documents and updates the knowledge base (the ontology) with the knowledge acquired during reading. Documents read are added to a pool of documents, which is continuously classified by the agent. It is important to note that the knowledge acquisition of the user and the classification of the agent are independent. The user does not itself classify the documents;

it only provides the concepts and their relations through the ontology, which are then used by the agent to classify the document. This means that the user may acquire knowledge from other sources than the classified documents and that the agent can classify other documents never seen by the user.

The agent is based on the believe-desire-intention model [137] and uses the ontology provided by the user as its knowledge base to classify documents. Because the agent itself has to use the relations in the ontology for its own reasoning, relations are limited to executable code and a predefined but extensible set is provided by the system consisting of “and”, “or”, and “equals” relations.

Classifiers are also used to detect document types (PDF, PS, etc.) and actions transform documents to plain text, which is the input format of the agent. The agent itself uses low-level classifiers to extract concepts from the textual source for example by statistical and syntax analysis. As with any other COCA application, all these aspects can be dynamically reconfigured and adapted during runtime by adding classifiers and actions.

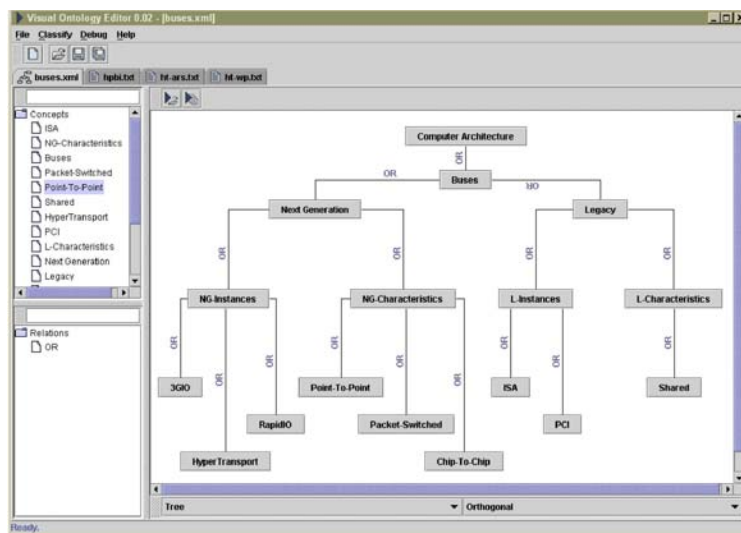


Figure 5.8: The ontology editor used for the agent based classifier.

A novel search tool was also implemented in the context of this project, which allows a user to build a concept landscape wherein each concept acts as a gravity source. Documents are then attracted by these concepts by their relevance. This allows a user to interactively skim across a large number of documents by changing the gravity landscape.

This project is currently still under the way but first results are promising, showing that COCA can as well be applied in a more classic AI domain by using classifiers for symbol grounding together with an appropriate inference mechanism.

5.6 SMASH

While writing this thesis I started to work for Swisscom Innovations, the research department of Swisscom³. As in any bigger organizations, system heterogeneity is commonplace and

³Swisscom is the biggest telecom operator in Switzerland (fixnet and mobile) with about two-thirds market share.

configuration problems are part of the daily business. Clearly, interest in models such as COCA is high and thus the SMASH (**SM**art**A**ssistants for **H**andhelds) project was launched by Innovations to investigate the usability of COCA.

The primary goal of SMASH is to increase the wireless application protocol (WAP) user experience. WAP usage is low mainly for two reasons: connection costs are relatively high because the user is charged for the full usage time instead of the data transferred. The other reason is the quality of the available services, which is still low compared to similar Web services.

The first issue is addressed by the general packet radio service (GPRS) that allows always-on connectivity and volume dependent charging. It is the usability issue where SMASH comes into play.

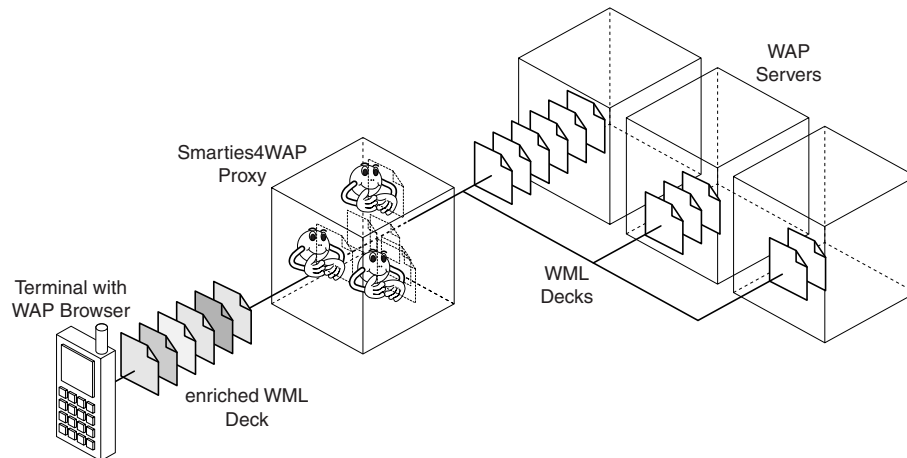


Figure 5.9: The operation of SMASH.

Figure 5.9 gives an overview of the operation of SMASH in a WAP setting consisting of servers providing content, a WAP proxy, and a user terminal. SMASH acts as a proxy for WAP that enriches wireless markup language (WML) cards with additional, context-dependent information on the fly and delivers the enriched pages to the terminal. WML cards are enriched by so called “smarties” which are a combination of classifiers and actions. Classifiers analyze a WML card within the current context and actions transform the WML card into the enriched version. Examples of “smarties” are:

- **Time-table:** Time-table applications usually require the user to enter its departure location, destination location, and times to find corresponding busses, trains, or flights. A “smarty” may detect the corresponding input fields in the WML card and supply information such as the current terminal location, current time, and a few possible destinations derived from the user’s travel history and list of future meeting locations.
- **Autologon:** Many WAP services request a user to identify itself by a user-name and password which is time-consuming on a mobile terminal. A “smarty” may detect login requests and automatically supply username and password since it is able to identify the user by its terminal. Clearly, for security reasons the user may disable this “smarty” for certain services.
- **Smart-call:** A “smarty” may detect names, e-mail addresses, and phone numbers in

WML cards and lookup the corresponding phone numbers in the user's address book. The enriched page may then allow the user to directly call a name on a WML card. This is especially interesting for mobile users reading their e-mail via WAP, allowing them to reply directly by a voice call to an e-mail sender.

These are only a few examples of what could be realized with SMASH. Because SMASH is based on COCA, it should be easy to extend when new ideas emerge. The ideal would be that new "smarties" could be developed locally even by third party vendors and simply uploaded into the proxy where they perform their work in orchestra with the other "smarties".

5.7 Summary

This chapter presented the implementation of the forerunners that lead to COCA as well as three new applications based on COCA. It shows the importance of testing ideas in real-world scenarios for verification as well as for the feedback to advance the theoretical work. It is the interplay of ideas, implementations, and problems that guided this research.

Some questions raised by the WOS project resulted in WebRes, which lead to WebCom. With the problems discovered through WebCom, the necessary abstractions were developed resulting finally in COCA. It was never intended to let COCA without an application; the question was merely where to apply the model first. Luckily, ubiquitous computing research at the DIUF provided such an opportunity and showed with success the value of COCA.

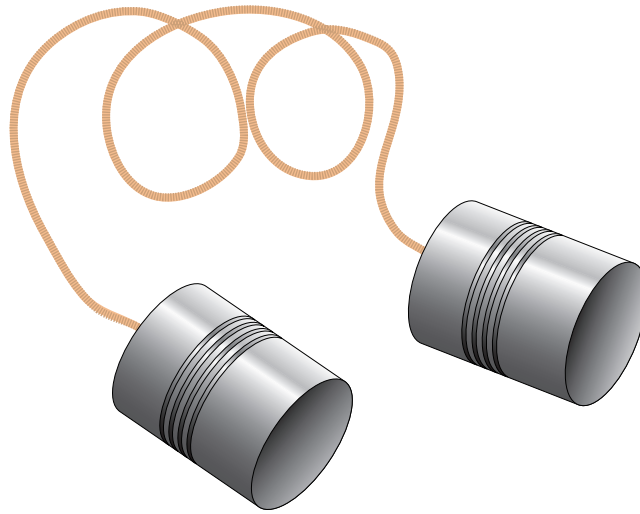
With the agent based classifier and SMASH, two new COCA application are on the way both exploring new domains where COCA could help to develop robust, auto configuring software for heterogeneous and dynamic environments.

Part III

Discussion and Conclusion

Chapter 6

Discussion and Related Work



6.1 Introduction

The previous chapters underlined the relevance of a proper handling of heterogeneity and dynamic configuration in distributed systems. COCA is a model that addresses these problems in a novel way, aiming at simplified application development, integration, and system evolution. Clearly, as a model it is not a shrink-wrapped solution that can be directly deployed. Nevertheless, the implementations so far are promising and show the value of the COCA model in the context of distributed systems.

As already apparent from the first three chapters, every sub-problem is a research domain of its own and specialized solutions exist for every single aspect. COCA tries to handle all aspects with a single model, gaining from the synergy of such an approach.

The following section looks at five research systems that by their distributed nature have to address heterogeneity and dynamic configuration.

As stated before, the main problem in current systems is the lack of machine processable semantics. Especially in the fast evolving Internet this lack is striking. The semantic Web project presented in section 6.3 shares the goal with COCA of adding semantic to an existing system, but is focused on the Web only.

Although they were heavily criticized¹, Microsoft Office XP's Smart Tags present an interesting approach of enhancing existing documents with semantic information and associated actions. In fact, Smart Tags might be seen as a simplified implementation of COCA, although both were developed independently.

Finally, the relation of this work with the research currently under the way at the DIUF is presented by the examples of the Focal project, XCM, and UbiDev.

Event though COCA provides answers for important problems in distributed computing, it also evokes some new questions which conclude this chapter.

6.2 Distributed Resource Management

The introduction of networking into operating systems increased the interest in remote resource access. Roughly three different approaches can be identified how remote resource access is provided to the user:

- **Distributed operating systems** run the same or a very similar kernel on every node in the network which controls the local resources as well as remote resources. Usually such an operating system appears to the user as a single machine with transparent resource access. Examples of distributed operating systems are Amoeba [172] or Mach [50].
- **Distributed operating system services** leverage existing operating systems features over the network, implementing services such as naming, distributed file systems, load sharing and balancing, or security. Examples of such system are the WOS (see also 3.2.3), Condor, Globus, Legion, and WebOS which are briefly presented in the following sections.
- **Distributed services** not only implement horizontal resource access but mainly vertical, application oriented high level services. Examples are CORBA (see 2.5.2), .NET (see 2.5.6), or Ninja which is presented below.

Condor

Although Condor [9] has a clear application target with high-throughput computing (HTC), its implementation provides standard operating systems services over the network. The implementation consists of libraries that intercept system calls such as file operations, which are then transferred over the network and executed on the machine providing the service together with some management and administrative applications.

In addition to that, Condor implements a checkpointing mechanism that allows applications to migrate between nodes and application restarting in the case of node or network failure.

Another interesting feature of Condor is its resource broker, which matches resource requests with resource providers by a mechanism called ClassAds. When a user submits a job to Condor it specifies along some properties such as CPU type and speed, memory footprint and that like. The nodes running Condor “advertise” their static and dynamic properties, which then can be matched with the job requirements. ClassAds are the Condor way of “addressing by query” and are based on pre-defined and implicitly agreed semantics.

¹Because Smart Tags may change the appearance of documents they infer with copyright laws.

Condor pays special attention to the rights and sensitivities of the workstation owners because a HCT system will only be successful if it does not infer with the user's use of its workstation. For that reason the user's can exactly specify the resource usage and migration policy for example during interactive work.

Globus

Globus [51] provides a vertically integrated infrastructure for high performance parallel programming through the illusion of a single virtual supercomputer available to applications. The Globus Project is a research and development project focused on enabling the application of "grid" concepts to scientific and engineering computing.

The "grid" refers to an infrastructure that enables the integrated, collaborative use of high-end computers, networks, databases, and scientific instruments owned and managed by multiple organizations. Grid computing is another way towards ubiquitous computing. The idea behind the "grid" is that computing resources should be ubiquitous available like electric power from the power grid (hence the name). Grid applications often involve large amounts of data and/or computing and often require secure resource sharing across organizational boundaries, and are thus not easily handled by today's Internet and Web infrastructures.

The Globus system consists of different components including the metacomputing directory service for locating remote resources, the Nexus [73] transparent multi-protocol communication layer, and the Globus security infrastructure which allows authentication among administrative domains.

Condor and Globus are complimentary technologies, as demonstrated by Condor-G [53], a Globus-enabled version of Condor that uses Globus to handle inter-organizational problems like security and resource management for supercomputers. Globus uses layering for abstracting the computing resources together with a set of services in order to provide a distributed computing environment. Globus lacks auto-configuration features and a lot of manual system maintenance is required.

Legion

Like Globus, Legion [63] implements a virtual super computer but takes an object-oriented approach. It is designed for a system of millions of hosts and trillions of objects tied together with high-speed links. Users working on their home machines see the illusion of a single computer, with access to all kinds of data and physical resources, such as digital libraries, physical simulations, cameras, linear accelerators, and video streams. Groups of users can construct shared virtual workspaces, to collaborate research and exchange information. This abstraction springs from Legion's transparent scheduling, data management, fault tolerance, site autonomy, and a wide range of security options.

Like the WOS, Legion sits on top of the user's operating system, acting as liaison between its own host(s) and whatever other resources are required. Legion's scheduling and security policies act on behalf of the user and reduce negotiations with outside-systems and system administration. The user has also the possibility to protect its own resources against other Legion users, so that administrators can choose appropriate policies for who uses which resources under what circumstances. To allow users to take advantage of a wide range of possible resources, Legion offers a user-controlled naming system called context space, so that users can easily create and use objects in far-flung systems. Users can also run applications written in

multiple languages, since Legion supports interoperability on the binary level between objects written in multiple languages.

Legion differs from distributed object oriented system like CORBA in the sense that it is not based on an object model with attributes, methods, inheritance, polymorphism, and so on but it uses object as its basic abstraction and provides access, naming, management and protection for it.

WebOS

The goal of WebOS is to provide a common set of OS services to wide area applications including, a global namespace, remote process execution, resource management, authentication, security, and mechanisms for resource discovery. Usually developers can rely on such services when writing for a single host environment but for wide-area application they have to come up with their own solution. To address this problem, the WebOS provides basic operating systems services needed to build applications that are geographically distributed, highly available, incrementally scalable, and dynamically reconfiguring. The basic set of services supplied by the Web OS is:

- **WebFS:** A global file system layer allowing unmodified applications to read and write to the URL name space. Cache consistency is available to applications requiring it through the AFS protocol.
- **Active names:** A mechanism for logically moving service functionality such as load balancing, resource discovery, and fault transparency from the server into the network.
- **Secure remote execution:** Since local resources may be requested by arbitrary remote users, assurances must be provided ensuring that applications are not able to violate the integrity of the local machine and that local users cannot take advantage of any user access rights provided to the programs.
- **Security and authentication:** Applications accessing remote files must authenticate their identities before access to protected files can be granted.
- **Transactions:** Applications must have well-defined failure modes. For example, an aborted remote agent should not leave a user's local file system in an inconsistent state.

WebOS like Globus, Condor, and Legion settles on a set of services to facilitate the development of wide-area applications. An interesting aspect of these projects is the social issues involved in building the systems such as trust and personal resource integrity, this is also underlined by the strong focus on security of these systems. COCA classifiers present an interesting concept when used to enforce security policies and to assure arbitrary properties of resources such as certificate based trust for example. The future will show if these systems will emerge out of the scientific field and make the idea of computing power as easy accessible as electric power a reality.

Ninja

Ninja [62] like COCA addresses the broad range of distributed Internet services, although Ninja is an implemented system and COCA is a model. Like any other distributed system,

Ninja has to deal with issues such as heterogeneity, service description, naming, and service composition. The basic building blocks of Ninja are *units* which are devices and sensors, interacting with the users and the physical environment, *active proxies* which act as an adapter between units and services and *services* which run on *bases* that are clusters of workstations. Service description and discovery is handled by a distributed directory and lack the semantic available with by classifiers. Because Ninja services run on powerful clusters of workstations, resource requirements are a minor issue. In consequence, Ninja limits service description to their programming interface. For service development, Ninja provides design patterns and distributed data structures as well as a security infrastructure.

A powerful service composition technique called *path* allows explicit and implicit (automatic) building of high-level services through combination of basic services. Path [83] is basically a mediation infrastructure between heterogeneous communication endpoints. Its development was mainly motivated by the problems encountered in communication between heterogeneous entities today. The main techniques used up to now have several shortcomings:

- **Standardization** is one approach to allow heterogeneous entities to interact by defining protocols and data formats. But Path shares with COCA the critiques: standardization is impossible for the large number of ubiquitous computing devices and standardization does not address nonstandard legacy systems.
- **Content negotiation** is more flexible in terms of how interaction actually takes place but requires an additional negotiation phase prior to the actual interaction. Again, the negotiation phase as well as the interaction itself has to be standardized.
- **Polyglot entities** understand multiple data types and protocols but still have the burden of detecting the protocol (silent negotiation) and have no way to adapt to unknown systems.
- **Least common denominator** is often the only feasible way for communication with the implicit loss of information if no idem potent transformation from the source representation to the least common denominator representation exists.

Path is built around *operators* that transform data and are similar to COCA actions, *representatives* that speak the native protocols of the endpoints (similar to COCA contexts), and a *coordinator* that discovers and initializes paths of mediators to connect endpoints. The coordinator implements some form of automatic software configuration² and mediators and representatives can be added dynamically during runtime, extending the functionality of the system.

It is easily seen that Path shares many points with COCA. However, there are also differences: Path is focused on communication and further specializes operators for the specific purposes such as *mediators* which perform data type transformation, *semantic processors* that perform operations without changing the data type (e.g. sorting), *aggregators* and *disseminators* for multicasting and accumulation of data, and *sources* and *sinks* to represent communication endpoints. Path assumes typed data and lacks an automatic “typing” mechanism like COCA’s classifiers, and automatic path composition is based on type matching compared to

²Interestingly, the Path prototype also uses a shortest path algorithm for that purpose like the ubiquitous computing demonstrator.

concept matching of COCA. Because Path is also targeting at ubiquitous computing environments, it is interesting to note that very similar solutions (COCA and Path) emerged for solving the heterogeneity and configuration problems, motivating further research in this direction.

6.3 Semantic Web

The aim of the Semantic Web [17] project is nothing less than to bring the current world wide Web to its next, semantic enriched stage, which would allow not only humans to browse but machines to reason about Web page content. The Semantic Web will not be separate from the existing Web but interweaved with it. Several other attempts exist to enrich current Web pages with meta-data to enable automatic processing of pages such as the simple HTML ontology extension (SHOE [105]) or Ontobroker [46] for example. All these approaches have in common that they rely on some “resource description” either embedded as meta-data in the Web pages or in a separate database. Resource descriptions serve as the base level on top of which an ontology is constructed. This structured information is then combined with an inference mechanism, defining some logic system allowing to reason about the encoded information. Symbol grounding is often not addressed and it is assumed that “someone” will provide the resource description. This clearly raises several questions of information validity, consistency and trust. Semantics are given by the inference mechanism that in turn also defines the expressible universe a semantic enriched Web. The following two sections will look at the resource description framework (RDF) used by the Semantic Web and at the ideas behind the Semantic Web itself.

RDF

Although everything on the world wide Web is machine-readable, this data is not machine-understandable. In consequence, it is very hard to automate anything on the Web, and because of the volume of information the Web contains, it is not possible to manage it manually.

Resource description framework (RDF [99]) is a W3C recommendation for a foundation for processing metadata; its goal is to provide interoperability between applications that exchange machine-understandable information on the Web. The resource description should make no assumptions about a particular application domain, nor define a priori the semantics of any application domain. The definition of the mechanism should be domain neutral, yet the mechanism should be suitable for describing information about any domain.

This is basically also a problem addressed by a COCA ontology. Both RDF and a COCA ontology use metadata to describe the data of some application domain (e.g. the Web). The big difference between RDF and COCA is that RDF does not say *how* this metadata is constructed, there is nothing similar to COCA’s classifiers in RDF.

RDF is a model for representing metadata as well as an abstract syntax for encoding and transporting this metadata in a manner that maximizes the interoperability of independently developed Web servers and clients. Several concrete syntaxes exist such as an XML application and a graphical notation.

In order to facilitate the definition of metadata, RDF has a class system much like object-oriented programming. A collection of classes (typically authored for a specific purpose or domain) is called a schema. Classes are organized in a hierarchy, and offer extensibility

through subclass refinement. This results in the known advantages of reuse and generalization from object oriented programming but this time for knowledge.

The basic data model consists of three object types [99]:

- **Resources:** All things being described by RDF expressions are called resources. A resource may be an entire Web page, a part of a Web page (e.g. a specific HTML or XML element within the document source), or a whole collection of pages such as an entire Web site. Furthermore a resource may be an object that is not directly accessible via the Web, like a printed book. Resources are always named by URIs plus optional anchor ids (see [12]).
- **Properties:** A property is a specific aspect, characteristic, attribute, or relation used to describe a resource. Each property has a specific meaning, defines its permitted values, the types of resources it can describe, and its relationship with other properties.
- **Statements:** A specific resource together with a named property plus the value of that property for that resource is an RDF statement. These three individual parts of a statement are called, respectively, the subject, the predicate, and the object. The object of a statement (i.e., the property value) can be another resource or it can be a literal, a simple string, or another primitive data type defined by XML. In RDF terms, a literal may have content that is XML markup but is not further evaluated by the RDF processor.

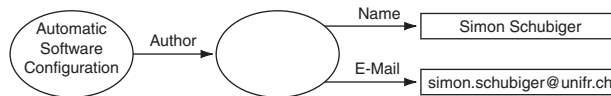


Figure 6.1: An example of a simple RDF statement.

A simple example of a graphical notation of the sentence “The person with name Simon Schubiger and e-mail `simon.schubiger@unifr.ch` is the author of ‘Automatic Software Configuration’ ” is given in figure 6.1. In this graphical notation, resources are represented with ovals and named properties by arrows. String literals are represented by rectangles. RDF allows anonymous resources such as “the person” which is the empty oval in the middle of figure 6.1.

So far, RDF allows only expressing structured information. In order to add semantic to the structure, RDF includes *schema* references that restrict and document the use of properties. Currently there is no formalization of RDF schemas and developers have to rely on informal definition of schemas.

Semantic Web

One of the driving ideas behind the original Web was that it is an information space not only for human-human communication but also that machines would participate and help. Even though humans are performing a broad range of activities by using the Web, the role of the machines is mainly limited to serve pages for human consumption. Although some information had a meaning and existed in machine processable form (for example because it was stored in a database or in a XML document) most of it is lost when rendered in HTML.

The Semantic Web [13, 14, 15, 16] effort develops languages for expressing information in machine processable form and make them available directly in that form instead of rendered HTML. The languages developed for the Semantic Web all build on top of RDF and address different aspects of storing and processing Web knowledge. Figure 6.2 gives an overview of how the different levels (with their specific language) of the Semantic Web are organized.

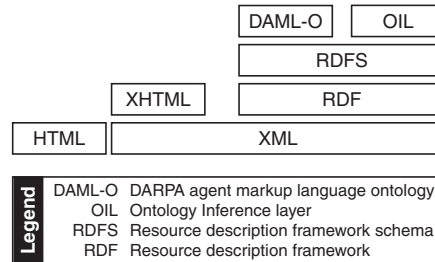


Figure 6.2: An overview of the languages used in the Semantic Web.

- **RDFS** (resource description framework schema [25]) is an extensible, object-oriented type system of terms and concepts. RDFS supports multiple inheritance and can specify domain and range constraints for properties. RDFS can be seen as set of ontological modeling primitives on top of RDF.
- **OIL** (ontological inference layer [47]) is a modeling language similar to a frame system combined with a descriptive logic reasoner. It is an extension of RDFS and targets applications such as search engines, e-commerce, and knowledge management.
- **DAML-O** (DARPA agent markup language [49]) is another high level language built on top of RDFS. DAML has constructs that make some reasoning service impossible or less efficient than OIL but allows in turn expressions not possible in OIL. The main goal of DAML is to provide a markup that allows software agents to “understand” the concepts used in a Web page.

Both COCA and the Semantic Web try to add meaning to data that should allow software agents to handle this data in a more intelligent way. It seems that there is an agreement that an ontology of interrelated concepts is an appropriate knowledge representation. How this ontology is constructed and how the symbols therein are grounded is still an open question. The Semantic Web effort favors an approach based on some kind of logic although some severe restriction might be necessary to yield an efficient reasoner. In that respect OIL and DAML are very similar to KIF. COCA takes the easy route by leaving the question of an inference mechanism open but contributes classifiers for symbol grounding which may act as very efficient “semantic shortcuts”³ making an inference mechanism unnecessary for certain application domains.

³For some application domains only very few (high level) concepts with simple relations such as subsumption and containment are sufficient and can be directly implemented by a few classifiers.

6.4 Smart Tags

Microsoft introduced Smart Tags [35, 123] in its Office XP line of products to recognize and link information across the various application in Office XP but also to provide an interface for third party addins to work with typed information in an Office XP document.

As an example for a Smart Tags application consider a user who knows that an employee's name in a Word report is the same as an employee's name in an Excel list, which is also the same as an employee's name in an e-mail message. Without Smart Tags this information is all just plain data to these applications without any further distinction. If the user now wants to find out more information about a particular employee mentioned in a report, he has to spend time launching a separate application and navigating a different user interface just to complete this simple task.

Smart Tags try to change this situation by attaching type information to plain data. The previously mentioned user can leverage the functionalities of these special applications to provide what a human knows intuitively, that is, the ability to recognize a particular string as type "person's name" and to respond with a list of possible actions. From that list the user can select the most appropriate action such as sending an e-mail message to, scheduling a meeting with, and getting the telephone number, or accessing personal records of the employee in question.

A developer in an organization is in the best position to provide knowledge and choices to users. Therefore the Smart Tags infrastructure not only implements a set of "recognizers" but also provides various interfaces for third-party addins. Addins may range from XML documents [33] that provide simple extension possibilities over custom "recognizers" [34] that identify strings of one or more specific types (for instance, a book title, chemical formula, or case number recognizer) to Web services using .NET [135].

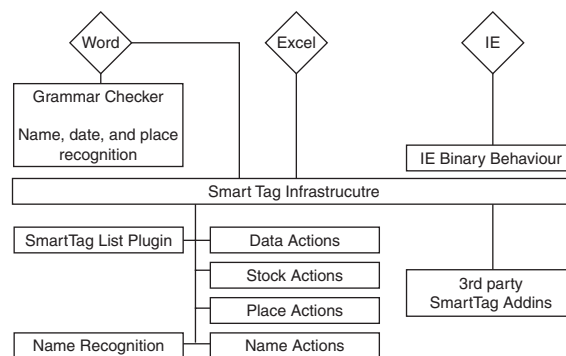


Figure 6.3: An overview of the Smart Tags architecture.

Figure 6.3 gives an overview of the Smart Tags architecture. Smart Tags use so called *recognizers* that are pieces of code that add type information to strings that are passed to them. For example, a recognizer could label "DIUF" as type **UniversityDepartment**. The type information attached to strings by recognizers is then used to compile a set of actions applicable to that string. These actions are then presented to the user as a popup menu allowing him to trigger an action. The most obvious application of Smart Tag recognizers is as plug-ins for Microsoft Excel and Microsoft Word, but Smart Tag recognition technology may be applied to a wide number of applications in the future as well, provided some extension

of the architecture.

It can easily be seen that recognizers are similar to COCA classifiers. Both operate on resources and tag the resources with semantic information. However, there are some important differences. Firstly, recognizers operate on Office XP documents⁴ only whereas COCA classifiers operate on any kind of resources. Secondly, recognizers modify the resource by attaching tags compared to the purely functional operation of classifiers.

Additionally, Smart Tags actions and COCA actions are similar, although by looking closer, a difference shows up here as well. Smart Tag actions describe only their *input* type, leaving the outcome of the action to the user's interpretation of the menu item. Because of that, software configuration is only possible at one level: either the action operates on the type or not - there is no way to combine actions as it is the case with COCA's input and output concept matching. Although the Word grammar and spelling checker supplies additional information to the recognizers such as document language, Smart Tags have no equivalent of the context abstractions defined by COCA.

It can be said that Smart Tags come quite close to an implementation of a subset of COCA; but their use and scope is limited to textual documents. Surprisingly, Microsoft did not try to extend Smart Tags to a wider range of resources such as images and sound although Office supports multi-media documents for a long time and the cross application integration possibilities would go much further than Word, Excel, and Internet Explorer only.

6.5 Focale

The Focale [100, 101] project headed by Michèle Courant and Stéphane Le Peutrec of the DIUF aims at developing interaction tools fitted with a new human computer interaction (HCI) schema, starting from the concrete situation of artificial biology.

Focale's HCI approach is based on the paradigms of autonomous agent and situated intelligence, and marks a deep renewal in the way of considering interaction. Focale shifts HCI from the client-server paradigm to the ecosystemic partnership paradigm. The main motivation for this shift is that for interacting with an evolutive computing system, new sort of interaction tools are necessary to observe, control, discover and explore it.

Focale proposes an adaptable interface model called virtual instrument (VI) following the metaphor of physical instruments that capture information broadcasted by real objects (e.g. camera, microscope, radio). The model allows to dynamically interconnect through a network with a simulation on the basis of a conceptual interface provided by this simulation. The user customizes a virtual instrument in order to choose the objects he wants to interact with, what part of these objects he wants to observe or change, and when the interconnection must be effective. In addition, the user also chooses the representation (graphical, audio, or even tactile) of the captured information as well as his manipulation interface (e.g. mouse, voice, gestures).

The notion of virtual instrument (VI) answers the need for providing a unified vision of the computing functionalities emerging from the convergence between information and communication technologies. The notion of VI addresses the broad range of Internet technology as well as ubiquitous computing scenarios but additionally aims at extending seamlessly to

⁴In fact, recognizers currently can only describe "what" they can recognize as a list of words, a cell (for Excel), or as a regular expression.

any upcoming appliance where humans and computer interact. The anthropological background of VIs is that (physical) instruments are actually defined as external and separable sensory-motor organs for modifying the modalities of the environment access of the body. Surprisingly, a computer seems not to fit this perspective. A computer is often seen as an artificial “other” which corporate with the human than being an extension of the human body. This situation was recognized as a basic obstacle for a common abstraction and lead to the three key objectives of a VI that can be summarized as [101]:

- The adoption of an anthropocentric approach on the problem of immersing an a priori inaccessible domain into the sensory-motor universe of the human body.
- The standardization of the computer among other instruments.
- The hypothesis of the physical space primacy as first primary body action domain and its corollary, that a metaphorical design of HCI starting from the physical instruments could have a facilitating effect.

Architecture and Implementation

The architecture proposed by Focale consists of four key elements allowing to model the human-computer interaction:

- **The object** of observation and manipulation, which constitute the target of the VI. In the figures 6.4 and 6.5 an artificial life (AL) experiment provides the objects as an example.
- **The observation VI** (figure 6.4) which attaches to the objects, collects, and transforms the signals emitted from the object under a specific time law.
- **The subject** of observation and manipulation which is usually the human user interacting with the system.
- **The manipulation VI** (figure 6.5) which again attaches to the objects, but *manipulates* the objects by performative actions under a specific time law on behalf of the user.

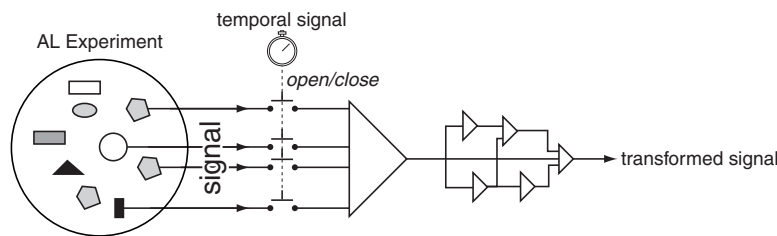


Figure 6.4: Architecture of an observation VI.

A first implementation of this architecture uses an object oriented design similar to a combination of the model-controller-view (MVC) pattern combined with the mediator pattern [54]. An application allowing interaction provides a conceptual interface that an interconnection server uses to satisfy interaction requests. A so called “mask” (see figure 6.6) acts as mediator between the semantic universe of the application and the semantic universe of the user.

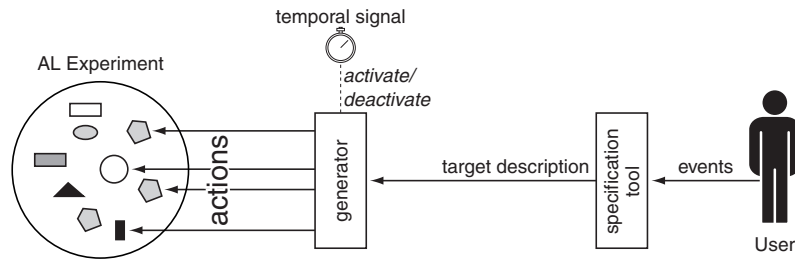


Figure 6.5: Architecture of a manipulation VI.

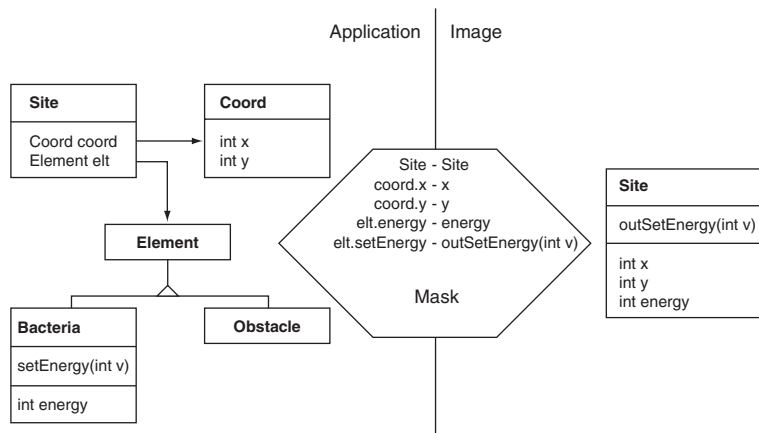


Figure 6.6: An example of application imaging.

Thereby enabling the creation of a consistent application image for the user and the controlled manipulation of the application by the user.

Figure 6.6 gives an example of an application, its image, and a mask mediating in-between. It is important to note that the mask does not only structural transformation but also *semantic* mediation, representing the application in the user's terms and vice-versa.

COCA may contribute to Focale in various ways. Firstly, COCA classifiers can be used where symbol grounding is required, mainly to capture the semantic universes of the application and the user. Secondly, a COCA ontology is an obvious choice for the representation of both universes, defining a vocabulary of terms that can then be used for mask construction. Because a mask basically mediates between concepts, it is equivalent to a set of actions. Automatic software configuration could therefore be used to construct masks with little help from the user. A user may for example construct a view of the application in terms of its own ontology. This conceptual view can then be used for automatic software configuration to find suitable combinations of actions that mediate between the user's view and the application.

6.6 XCM

Whereas COCA can be seen as attacking semantics from the bottom up, Amine Tafat-Bouzid's extended coordination model (XCM) currently under development at the DIUF takes a top down approach. Research in coordination theory has a long history at the DIUF and resulted in several coordination models and languages [72, 92, 143]. But it was not until now that a project was launched generalizing all these models that should finally result in XCM. XCM as many other coordination models builds on the following assumptions:

- Computer systems will be more and more open enabling and requiring interaction.
- Computer systems will evolve instead of having discrete life-cycles.
- Computer systems will show indeterministic behavior in certain areas.

Several coordination languages try to help in these areas but exist only as implementations, exhibiting a large gap between coordination theory and the available systems. Therefore XCM aims at closing this gap by defining a formal model for coordination which should allow theoretical study and understanding of coordination as well as instantiations that simplify implementations. Based on the thesis of Varela [113], XCM distinguishes two coordination domains:

- A organizational domain where coordination is formulated as rules for abstract situations.
- A realization domain where the organizational rules are implemented and applied to a concrete situation.

The relation between these two domains is an instantiation action of the organizational domain producing an universe of so called *entities*. An entity is the root abstraction of XCM: everything is an entity. An entity is defined by its *structure* obtained by a recursive composition of entities until some non-composite (atomic) entity is reached. The composition process of entities is expressed by operations supplied by an enclosing entity called *actions*. Every entity has in turn at least one enclosing entity with the *universe* as the only exception, which includes everything.

Entities are dynamic in the sense that their internal (enclosed) or external (enclosing) structure may change. A structural change is considered as a “state-change” of the affected entities and just another action like composition, which also affects the structure of an entity. Evolution in XCM is the sequence of state transitions of the universe, but may also be seen on a smaller scope as local evolution. Evolution can only be observed by entities with the ability to perceive state changes, implying some sort of memory allowing them to compare the current state with a previously memorized state.

This leads to the *autonomy* definition of XCM: an entity is called “autonomous” if another external entity perceives some non-determinism in its capacity to change state. XCM distinguishes between *external autonomy*, which describes changes of the enclosing entity, and *internal autonomy* that addresses structural changes of the entity in question. It is important to note here that autonomy in XCM does not exist per-se: it explicitly requires an observer that is capable to perceive the change as well as the inability of the observer to deterministically predict the change. Only when these two conditions are met by an observing entity the observed entity is considered autonomous. This also means that the same entity may be autonomous for one observer and deterministic for another.

In order to simplify instantiation of the model, some auxiliary notions are introduced by XCM. One is the notion of the *agent*, which simply describes an autonomous entity. Another is the *port* which serves as a communication endpoint between agents and is defined as a restricted set of actions. Elaborate protocols used for inter-agent communication are called *social-rules* and restrict composition of basic communication actions.

The link between COCA and XCM is less obvious mainly due to the opposite approach of the two projects. Although XCM is still in its definition phase, it is likely to have some impact on automatic software configuration because this is also a coordination activity. XCM may help to evaluate automatic software configuration algorithms for COCA in terms of expressiveness and complexity. Because automatic software configuration is mainly a search problem, the size of the search space as well as the algorithm applied are important factors for practical and efficient implementation of automatic software configuration. Having a formal model such as XCM at hand helps understanding these issues especially in the context of a dynamically changing environment where standard complexity analysis techniques are more difficult to apply.

6.7 UbiDev

The direct follow-up of the COCA research is Sergio Maffioletti’s UbiDev project, which uses COCA classifiers and actions for a homogeneous middleware that allows definition and coordination of services in interactive environment scenarios [108, 153]. The goal of UbiDev is to provide at application level a host independent interface to the underlying service-based system. Table 6.1 shows the architectural layers of the UbiDev model.

The *physical* layer represents a federation of system resources belonging to the environment. Due to the dynamic changes occurring in this layer such as network bandwidth variation, connectivity, and location, the system has to classify these resources continuously and monitor their changes. Thanks to this classification the upper layers have always a consistent description of the physical dimension.

The *service* layer takes care of analyzing the relation between the service description and the physical devices. Starting from the classification in the physical layer and in conjunction

Application	Coordinated application structure
Coordination	Interaction rules and composition
Service	Relation between service description and device
Physical	Hardware devices

Table 6.1: Architectural layers of a homogeneous execution environment for Ubiquitous Interacting Devices (UbiDev).

with the service description, the service layer determines the suitable execution environment for hosting a service; then it encapsulates the service and its execution environment into a self-contained entity on the hosting device. This allows a high degree of flexibility in device lookup, service description, and service composition.

The *coordination* layer defines a coordination space and interaction rules in order to allow applications to coordinates and combine services. This layer is composed of coordinated and coordinative entities, coordination media, and coordination laws. The main difference from classical coordination models is the addition of behavioral laws for each of the homogeneous entities defined in the service layer.

Finally the *application* layer contains the whole structure of the application in which homogeneous entities encapsulating services are coordinated. The layers below hide all the heterogeneous implementation details.

Service Layer

The core of the homogeneous abstraction is the control of the service instantiation by the *service* layer. The process of defining a homogeneous abstraction is characterized by three key elements of the service layer: *resource classification*, *EXecution Environment (EXE)* and *capsule*.

- **Resource classification:** Resource classification relies on a set of abstract concepts collected in a COCA ontology and the meaning of these concepts implicitly given by classifiers. COCA classifiers are used to decouple the high-level concepts (abstractions) from the instances implemented by a context. The main advantage of this approach in facing the resource management problem is that resources selection is based on the semantics that is given by the context. Since every application may have its own ontology (locality of knowledge) the application structure (ontology) is separated from the implementation (actions).
- **EXecution Environment (EXE):** The relation between the physical and the software dimension is expressed inside the homogeneous entity as the relation between a service and its hosting device. For expressing this relation an extension of the service concept in a service-oriented design model has been conceived. This extension called EXecution Environment (EXE) encapsulates the description of the service execution environment given by the service itself. The EXE description is composed of high-level concepts that are instantiated by the service layer in function of the context; this context may change with time and location. The process of requesting an instance of a concept takes place through addressing by concept (see 4.6.6), which assures that the host environment constraints stated by each service are taken into account by the system

and not by the application. EXE allows an extended description of a service not only in terms of functionality but also in terms of its execution environment. With service configuration based on concept matching, UbiDev is going beyond strong typing and design by contract.

- **Capsule:** A capsule represents the run-time instance of a service executing on a specific host device. It embeds a service and the access to its execution environment. Capsules are transparently and dynamically bound to different service so that all host-dependent information is moved out of the application's kernel and into the capsule itself. In this way, the system separates the naming of resources from their physical embodiment. The capsule is built around this relation; it represents an active computing environment in terms of both service and device. A capsule is a self-contained entity that interacts with other capsules and with the system basically through the coordination space defined in the coordination layer. To be fully self-contained, a capsule must incorporate the complete state of the active computing environment for its service.

The service instantiation process itself then goes through the following three steps:

1. The physical devices composing the environment are classified by COCA classifiers that associates high level concepts taken from the application's ontology with physical devices.
2. The service's EXE description is analyzed in order to find suitable instances of the high level concepts requested.
3. The service requested by the execution environment is encapsulated and a capsule is instantiated on the hosting UbiDev.

Coordination

The main advantage of hiding heterogeneity by the capsule abstraction is that the UbiDev infrastructure can present a homogeneous coordination space to the application, a unified mechanism for dynamic communication, coordination, and sharing of objects [2, 162].

This coordination space is an instantiation of the XCM's unified coordination model. By using the coordination space, applications can combine capsules to more structured entities and control their interaction with well-defined social rules. Moreover, a human interface like the manipulation instruments proposed by Focale allows the integration of the user as a coordinative entity.

The homogeneous environment and the coordination layer are the first steps towards the definition of new methodologies and paradigms in the development of ubiquitous computing applications, advancing the research started with COCA.

6.8 Open Questions

Although the current implementations have shown the feasibility and the advantages of COCA, it would be too enthusiastic to extrapolate these experiments to the entire ubiquitous computing domain as well as Web services and beyond. More instantiation of COCA have to take place to really show its potential and benefits. This also raises the first question: *What are the*

application domains of COCA in ubiquitous computing, Web services and beyond? While implementing COCA many ideas came up how designs could be simplified using COCA or how COCA could help improving existing environments but the lack of time prevented further implementations.

Another problem inherent to the application domain is *size*. Although COCA classifiers and actions can be developed with a very local scope and the larger configuration aspects are then handled by automatic software configuration, COCA does not help managing a pool of thousands of classifiers and large ontologies.

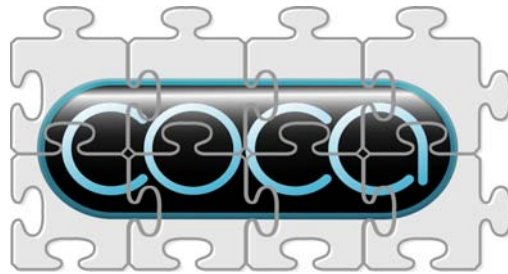
Especially in the agent based classifier project (see 5.5) the ontology size became a problem at least with the current user interface. But it may be questioned if this is only a problem of presentation or if new methods have to be found for handling large ontologies and sets of symbols. The application specific ontology approach promoted by COCA may help to alleviate this problem and ontology merging may be used to automatically construct larger ontologies but this was never tried in practice. So the questions “*how can we deal with large ontologies and classifiers pools?*” remains and requires further research before ubiquitous computing and Web services can be used at large.

Other issues involve performance. On the one hand side classifiers consume CPU power on every context change and on the other hand the inference mechanism used for automatic software configuration may also be computationally intensive depending on the search algorithm used. Classifiers usually run in parallel⁵ and effectively use otherwise idle resources but some resource properties may be very expensive to establish or even be undecidable. In such cases scheduling becomes an interesting problem because “fairness” may translate to “establishing a resource property in about the same time”, independent of the classifier complexity. The question of *what is a good scheduling algorithm for a COCA implementation?* has still to be answered and the answer might vary from application to application.

Because it is planned to continue using a COCA implementation for the ubiquitous computing middleware under development at the DIUF as well as using parts of it for the Focale project, hopefully these projects will come up with some answers.

⁵In the ubiquitous computing demonstrator every classifiers runs as a Java thread for example.

Conclusion



This dissertation addresses three fundamental problems, which have to be handled in application domains exhibiting heterogeneity and dynamism, like ubiquitous computing and Web services. The three problems heterogeneity, static configuration, and dynamic configuration are not only considered to persist during the near future but to increase, as similar application domains will emerge.

The heterogeneity problem is defined and examples taken from applications, programming environments, middleware, communication, and hardware illustrate the importance of the heterogeneity problems on all these levels as well as the solutions in use today. These solutions are mainly built around layering and standardization which both have problems of their own such as addition of new abstractions, weak semantics, and the question of integration of non-standard components just to mention a few.

The static configuration problem mainly appears during development and deployment of software and hardware. There is much research going on in the software engineering field to better handle the static configuration aspects of computer systems. This leads to architecture description languages and software configuration management tools which together aim at covering the entire software construction process. Although theoretically advanced, many ideas are not in widespread use because current development environments lack the features to extract the properties required by the configuration tools. The static configuration problem is defined in chapter 2 and examples of static configuration support in current systems are given.

The configuration problem has also a dynamic aspect especially in environments like ubiquitous computing where the configuration consisting of mobile devices is easily modified or on the Internet where new services appear and disappear daily. A definition of the dynamic configuration problem together with examples of software systems conceived for dynamic environments are presented in chapter 3 as well as how changing hardware configuration is handled today.

Without a doubt, there are many other obstacles, both technical and social, to overcome for successfully implementing the vision of Internet and ubiquitous computing but the COCA

model as the key result of this dissertation helps at least solving some of the technical aspects. The COCA model presented in chapter 4 introduces seven key elements helping the design and implementation of systems of high complexity. These elements are:

- **Resources** as the root abstraction of COCA and the atomic units that can be handled.
- **Contexts** are the containers resources live in and define resource access and naming.
- **Classifiers** are a solution of the symbol grounding problem and provide the basic level of semantic abstraction by associating concepts with resources.
- **Concepts** are the semantic abstractions grounded by classifiers and used to construct ontologies.
- **Ontologies** consist of interrelated concepts and provide higher level semantics together with an inference mechanism.
- **Relations** are used to represent higher level semantics in ontologies.
- **Actions** capture the dynamic aspects of the model by transforming resources from one concept to another.

Applying COCA allows automatic software configuration given a suitable inference mechanism and addressing by concept, which in turn enables fault tolerant systems.

COCA not only helps in system design, it also accommodates important programming paradigms such as object oriented, functional, and logic programming. Autonomous agent systems and the entity relationship model have also a mapping to COCA resulting in a wide coverage of important concepts currently found in computer science.

The model stands not in the void. The two forerunners of COCA as well as three implementations of the model are presented. The ubiquitous computing demonstrator (see 5.4) as a first COCA implementation for the ubiquitous computing domain not only realizes all the elements of the model but also a simple mechanism for automatic software configuration and uses addressing by concept.

Finally, the model is compared with a selection of systems in use in similar areas or sharing features with COCA. COCA hopefully contributes to the current and future research at the DIUF and beyond where many intersections exist.

Index

- .NET, 17, 54
- iLink, 73
- Absolute addressing, 94
- ACPI, 71
 - BIOS, 72
 - machine language, 72
 - Registers, 72
 - system description tables, 72
- Active data objects, 54
- Active names, 118
- Active server pages, 54
- Activity centered modeling, 38
- Address resolution protocol, 59
- Addressing by Concept, 93
- ADL
 - Components, 39
 - Configurations, 40
 - Connectors, 39
- ADO, 54
- Advanced configuration and power interface, 71
- American standard code for information interchange, 8
- AML, 72
- Architecture description languages, 38
- ARP, 59
- ASCII, 8
- ASP, 54
- Assembler, 13
- Assertion, 90
- Attribute, 89
- Attributes Schemes, 104
- Autoconf, 42
- Autonomous Agents, 92
- Bluetooth, 26, 75
 - profiles, 26
 - specification, 26
- Bus drivers, 69
- C language, 13
- CardBay, 74
- CardBus, 74
- Chain of actions, 87
- CISC, 13
- Class, 88
- COCA, 79
 - action, 86
 - Actions, 86
 - Classifiers, 83
 - classifiers, 83
 - concept, 83
 - Concepts, 83
 - Context, 81
 - input concept, 86
 - Input constraint, 87
 - output concept, 86
 - Output constraint, 87
 - Resources, 81
 - sub-context, 81
- Cognitive psychology, 80
- COM, 48
 - categories, 48
- Common object request broker architecture, 18
- Complex instruction set computers, 13
- Component object model, 48
- Component Repository, 37
- Computational resources, 66
- Concurrent versioning system, 43
- Condor, 116
 - ClassAds, 116
- Condor-G, 117
- Content negotiation, 119
- CORBA, 18, 47
- CORBAfacilities, 21
- CORBAservices, 20

- Correctness formula, 45
- CVS, 43
 - branch, 43
 - check-in, 37
 - check-out, 37
 - checkout, 43
 - commit, 37, 43
 - import, 43
 - module, 43
 - release, 43
 - revisions, 43
 - update, 43
- DAML-O, 122
- Data flow
 - network, 103
 - paradigm, 102
 - programming, 90
- DBCS, 8
- Deadlock
 - avoidance, 65
 - detection, 65
 - prevention, 65
- Design by contract, 90
- DHCP, 59
- Distributed operating system services, 116
- Distributed operating systems, 116
- Distributed services, 116
- Distributed virtual machine, 62
- Document object model, 10
- Document type definition, 9
- DOM, 10
- Double byte character sets, 8
- DTD, 9
- DVM, 62
- Dynamic configuration, 58
- Dynamic disks, 67
- Dynamic host configuration protocol, 59
- Eiffel, 45
- Encapsulation, 89
- Endianness, 6
- Enterprise Java Beans, 53
- Entity relationship model, 92
- Epistemology, 85
- Exclusive resources, 64
- Extensible markup language, 9
- Field, 89
- FireWire, 73
- Focale, 124
 - mask, 125
- FTP, 6
- Function drivers, 69
- Functional programming, 90
- General packet radio service, 59
- Global system for mobile communications, 75
- Globus, 117
- GNU extension library, 17
- GPRS, 59
- Grid, 117
- GSM, 75
- GUI abstraction
 - API emulation, 15
 - Emulation, 14
 - Layered, 14
- Guile, 17
- HailStorm, 55
- HAL, 30
- Hardware abstraction layer, 30
- Harness, 61
- Heap segment, 66
- Heterogeneity, 5
- Heterogeneity problem, 7
- High-throughput computing, 116
- HTML, 9, 24, 59
- HTTP, 6, 24, 59
- Hypertext markup language, 9, 24
- Hypertext transfer protocol, 24
- I/O Kit, 29, 70
- IDL, 19, 47
- IEEE
 - 1212, 73
 - 1394, 73
 - floating point format, 8
- IFL, 46
- IID, 48
- IIOP, 19
- Implementation repository, 47
- Inference mechanism, 91
- InfoBus, 50
- Inheritance, 88

- Instance, 88
- Instance storage, 89
- Inter-application communication, 85
- Interface
 - definition language, 47
 - description language, 19
 - identifier, 48
 - repository, 47
- Intermediate functional language, 46
- Internet inter-ORB protocol, 19
- Interpreted languages, 13
- IP mobility, 59
- ISO 10646, 8
- JAF, 49
- JAR, 45
- Java Beans, 48
 - customization, 49
 - events, 49
 - introspection, 49
 - persistence, 49
 - properties, 49
- Java, 16, 44
 - interface, 44
 - serialization, 44
 - virtual machine, 16
- Java activation framework, 49
- Java server pages, 53
- Javadoc, 45
- Jini, 50, 63
- JSP, 53
- JVM, 16
- Kernel modules, 29
- KIF, 11
- Knowledge
 - interchange format, 11
 - query and manipulation language, 11
- KQML, 11
- Language binding, 19
- Late binding, 89
- Least common denominator, 119
- Legion, 117
- Load
 - balancing, 66
 - sharing, 66
- Logic programming, 91
- Logical volume management, 67
- LVM, 67
- Mac OS X, 28, 70
 - Drivers, 70
 - Families, 70
 - Nubs, 70
- Make, 41
- Manipulation VI, 125
- Matching directory, 29
- Mediation, 119
 - semantic, 127
- Memory management unit, 67
- Meta-operating system, 22
- Method, 89
- Middleware, 18
- MIME, 24
- MMU, 67
- Multipurpose Internet mail extension, 24
- NAS, 67
- Network attached storage, 67
- Nexus, 117
- Ninja, 118
 - Path, 119
- Object, 88
 - constraint language, 39
 - Flow Networks, 105
 - management architecture, 20
 - management group, 18
 - request broker, 19
- Observation VI, 125
- OCL, 39
- OIL, 122
- OMA, 20
- OMG, 18
- ONE, 53
- Ontobroker, 120
- Ontology, 85
- Open system interconnect, 24
- Operating system structures, 66
- ORB, 19
- OSI, 24
- Outgoing interfaces, 48
- Overloading, 89
- PAC, 74

- PACT, 74
- Parallel array computing technology, 74
- PC Card, 74
- Performative, 11
- Plug and play, 69
- Polyglot entities, 119
- POP, 6
- POSIX, 14
- Postcondition, 45
- Precondition, 45
- Processing array clusters, 74
- Qt, 15
- RDF, 120
 - Properties, 121
 - Resources, 121
 - schema, 122
 - Statements, 121
- RDFS, 122
- Redirectors, 94
- Reduced instruction set computers, 13
- Relations, 86
- Relative addresses, 94
- Resource set, 101
- Resource set server, 98
- Revision tree, 37
- RIP, 59
- RISC, 13
- Routing information protocol, 59
- SAN, 67
- Scheduling, 66
- SCM, 37
- Semantic nets, 80, 84
- Semantic Web, 120, 121
- Semantics, 82
- Semiotics, 84
- SGML, 9
- Shared library segment, 66
- Shared memory segment, 66
- Shared resources, 65
- SHOE, 120
- Simple object access protocol, 21, 52
- Smart Tags, 123
 - Office XP, 123
 - recognizers, 123
- SMTP, 6
- SOAP, 21, 52
- Software Architecture, 38
- Software Configuration Management, 37
- Software contract, 45
- Stable Storage, 67
- Stack segment, 66
- Standard generalized markup language, 9
- Standardization, 119
- State, 88
- State transition diagram, 38
- Static configuration problem, 36
- Storage area networks, 67
- Structured knowledge, 85
- Symbol-grounding problem, 83
- Taxonomy, 84
- Text segment, 66
- Type, 88
- UbiDev, 128
- Ubiquitous Computing, 31, 75
- UDDI, 52
- UMTS, 75
- Unicode, 8
- Uniform resource locator, 24
- Union of contexts, 81
- Universal description, discovery, and integration, 52
- Universal mobile telecommunications system, 75
- Universal serial bus, 72
- Unix, 28
- URL, 24
- USB, 72
 - device, 73
 - function, 73
 - host, 73
 - hub, 73
 - interconnect, 72
- UTF-16, 8
- UTF-8, 8
- Version Control, 37
- Virtual instrument, 124
- WAP, 53
- WDM, 31
- Web operating system, 22

- Web service description language, 52
- WebCom, 102
- WebFS, 118
- WebOS, 118
- WebRes, 98
 - Resources, 98
- Windows 2000, 30
 - I/O manager, 30
 - Plug and Play, 69
 - plug-and-play manager, 30
 - power manager, 30
- Windows driver model, 31
- Wireless application protocol, 53
- Wireless local area networks, 75
- WLAN, 75
- Workspaces, 38
- WOS, 22, 59
 - node, 60
 - protocol, 23
 - request protocol, 23
- WOSNet, 60
- WOSP, 23
- WOSRP, 23
- WSDL, 52
- XCM, 127
 - actions, 127
 - agent, 128
 - autonomy, 128
 - entities, 127
 - external autonomy, 128
 - internal autonomy, 128
 - port, 128
 - social-rules, 128
 - structure, 127
 - universe, 127
- XML, 9
- XML style sheet language, 10
- XSL, 10

Bibliography

- [1] H. Abelson, G. J. Sussman, and J. Sussman. *Structure and Interpretation of Computer Programs*. The MIT Press, Cambridge, Massachusetts, 1985.
- [2] G. Agha and C. J. Callsen. ActorSpace: An Open Distributed Programming Paradigm. In *Proceedings of the SIGPLAN Symposium on Principles & Practice of Parallel Programming PPOP*, pages 23–32, San Diego, California, May 19-22 1993. ACM Press.
- [3] M. Aguilar and B. Hirsbrunner. Basic Concepts of the CoLa Coordination Language. In *Priority Programme Informatics, Information, Conference Module 3 Massively Parallel Systems*, Zurich, Switzerland, November 29-30 1994.
- [4] W. Aitken, B. Dickens, P. Kwiatkowski, O. de Moor, D. Richter, and C. Simonyi. *Transformation in Intentional Programming*. Redmond, WA 98052-6399, USA, September 1997.
- [5] Apple Computer, Inc. *I/O Kit Fundamentals*. Inside Mac OS X. Addison-Wesely Publishing Company, Inc., Reading, Massachusetts, September 2001.
- [6] Trolltech AS. *Qt - Cross-platform C++ GUI Application Framework*, 2000.
- [7] G. Babin, P. Kropf, and H. Unger. A Two-Level Communication Protocol for a Web Operating System (WOS). In *Euromicro Workshop on Network Computing*, pages 939–944, Västerås, Sweden, August 1998.
- [8] F. Bapst and O. Krone. A Coordination Kernel for Coupling Heterogeneous Programming Environments. Technical Report 97.3, University of Fribourg, IIUF, Fribourg, Switzerland, February 1997.
- [9] J. Basney, M. Livny, and T. Tannenbaum. High Throughput Computing with Condor. *HPCU news*, 1(2), 1997.
- [10] M. Beck, J. Dongarra, G. Fagg, G. Geist, P. Gray, J. Kohl, M. Migliardi, K. Moore, T. Moore, P. Papadopoulos, S. Scott, and V. Sunderam. Harness: A Next Generation Distributed Virtual Machine. *International Journal on Future Generation Computer Systems*, 15(5/6), 1999.
- [11] A. Benway. NAS Wars. *SW Expert*, pages 26–31, March 2001.
- [12] T. Berners-Lee. *Universal Resource Identifiers in WWW*, 1994. RFC 1630.
- [13] T. Berners-Lee. *Semantic Web Road map*, October 1998.

- [14] T. Berners-Lee. *Interpretation and Semantic on the Semantic Web*, January 2000.
- [15] T. Berners-Lee. *Conceptual Graphs and the Semantic Web*, January 2001.
- [16] T. Berners-Lee. *The Semantic Web as a language of logic*, January 2002.
- [17] T. Berners-Lee, J. Hendler, and O. Lassila. The Semantic Web. *Scientific American*, May 2001.
- [18] P. V. Biron and A. Malhotra. XML Schema Part 2: Datatypes. <http://www.w3.org/TR/xmlschema-2/>, May 2001.
- [19] D. L. Black. Scheduling Support for Concurrency and Parallelism in the Mach Operating System. *IEEE Computer*, 23(5):25–43, 1990.
- [20] J. Blandy. *Data Representation in Guile*. Free Software Foundation, April 1999.
- [21] H. Bögeholz. Schnelle Strippen - Die Technik von USB 2.0 im Vergleich mit FireWire. *c't, Magazin für Computer Technik*, 15:134–139, 2001.
- [22] N. Borenstein and N. Freed. *MIME - Multipurpose Internet Mail Extension*, 1993. RFC 1521.
- [23] T. Bradley and C. Brown. *Inverse Address Resolution Protocol*, 1992. RFC 1293.
- [24] U. Breymann and J. Loviscach. Die neue C-Klasse. *c't, Magazin für Computer Technik*, (4):98–105, February 2002.
- [25] D. Brickley and R. V. Guha. *Resource Description Framework (RDF) Schema Specification 1.0*. W3C, December 2000.
- [26] B. Calder and B. Shannon. *JavaBeans Activation Framework Specification Version 1.0a*. Sun Microsystems, Inc., 2550 Garcia Avenue, Mountain View, CA 94042-1100 U.S.A, May 1999.
- [27] N. Carriero and D. Gelernter. Linda in Context. *Communications of the ACM*, 32(4):444–458, 1989.
- [28] P. Cederqvist. *Version Management with CVS*. Signum Support AB, 1993.
- [29] P. P.-S. Chen. The Entity-Relationship Model - Towards a Unified View of Data. *ACM Transactions on Database Systems*, 1(1):9–36, March 1976.
- [30] M. Colan. *InfoBus 1.2 Specification*. Lotus Development Corp., February 1999.
- [31] Compaq and Hewlett-Packard and Intel and Lucent and Microsoft and NEC and Philips. *Universal Serial Bus Specification*, April 2000. Revision 2.0.
- [32] Compaq Computer, Intel Corporation, Microsoft Corporation, Phoenix Technologies, and Toshiba Corporation. *Advanced Configuration and Power Interface Specification*, July 2000. Revision 2.0.
- [33] P. Cornell. *Developing Simple Smart Tags*. Microsoft Corporation, Redmond, WA 98052-6399, USA, May 2001.

- [34] P. Cornell. *Developing Smart Tag DLLs*. Microsoft Corporation, Redmond, WA 98052-6399, USA, April 2001.
- [35] Microsoft Corporation. BRIDGE Reaches New Customers with Office XP Smart Tags and Real-Time Data. Technical report, Microsoft Corporation, Redmond, WA 98052-6399, USA, June 2001.
- [36] Microsoft Corporation. *C# Language Specification*. Redmond, WA 98052-6399, USA, May 2001. Version 0.28.
- [37] F. Curbera, D. Ehnebuske, and D. Rogers. *Using WSDL in a UDDI Registry 1.05*, 2001.
- [38] C. J. Date. *An Introduction to Database Systems*. Addison-Wesely Publishing Company, Inc., Reading, Massachusetts, 6th edition, August 1995.
- [39] J.D. Day and H. Zimmermann. The OSI Reference Model. volume 71, pages 1334–1340. IEEE, December 1983.
- [40] R. Droms. *Dynamic Host Configuration Protocol*, 1993. RFC 1531.
- [41] D. Drysdale. *Tutorial Introduction to Guile*, 2000.
- [42] B. DuCharme. *XML - The Annotated Specification*. Prentice-Hall, Inc., Upper Saddle River, New Jersey, 1999.
- [43] P. Eronen and P. Nikander. Decentralized Jini security. In *Proceedings of the Network and Distributed System Security Symposium (NDSS 2001)*, pages 161–172, San Diego, California, 2001. Internet Society.
- [44] J. Estublier. *The Future of Software Engineering*, chapter Software configuration management: A Roadmap, pages 279–289. ICSE 2000, 2000.
- [45] D. C. Fallside. XML Schema Part 0: Primer. <http://www.w3.org/TR/xmlschema-0/>, May 2001.
- [46] D. Fensel, S. Decker, M. Erdmann, and R. Studer. Ontobroker in a Nutshell. In *European Conference on Digital Libraries*, pages 663–664, 1998.
- [47] D. Fensel, I. Horrocks, F. van Harmelen, S. Decker, M. Erdmann, and M. C. A. Klein. OIL in a nutshell. In *Knowledge Acquisition, Modeling and Management*, pages 1–16, 2000.
- [48] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. *Hypertext Transfer Protocol - HTTP/1.1*, 1999. RFC 2616.
- [49] R. Fikes and D. L. McGuinness. *An Axiomatic Semantics for DAML-ONT*, December 2000.
- [50] A. Forin, J. Barrera, M. Young, and R. Rashid. Design, Implementation and Performance Evaluation of a Distributed Shared Memory Server for Mach. In *Proceedings of the Winter USENIX Conference*. USENIX, January 1989.
- [51] I. Foster and C. Kesselman. Globus: A Metacomputing Infrastructure Toolkit. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(2):115–128, Summer 1997.

- [52] I. Foster and S. Taylor. *Strand - New concepts in parallel programming*. Prentice Hall, 1990.
- [53] J. Frey, T. Tannenbaum, I. Foster, M. Livny, and S. Tuecke. Condor-G: A Computation Management Agent for Multi-Institutional Grids. In *Proceedings of the Tenth IEEE Symposium on High Performance Distributed Computing*, 2001.
- [54] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison-Wesely Publishing Company, Inc., Reading, Massachusetts, 1995.
- [55] D. Garlan and M. Shaw. *An Introduction to Software Architecture: Advances in Software Engineering and Knowledge Engineering*, volume I. World Scientific Publishing, 1993.
- [56] B. Gates. .NET today, June 2001.
- [57] M. R. Genesereth and R. E. Fikes. Knowledge Interchange Format, Version 3.0. Reference Manual. Technical Report Logic-92-1, Computer Science Department, Stanford University, 1992.
- [58] M. Ginsberg. *Essentials of Artificial Intelligence*. Morgan Kaufmann Publishers, San Mateo, California, 1993.
- [59] T. Goddard and V.S. Sunderam. WebVectors: Agents with URLs. Technical report, Departement of Math and Computer Science, Emory University, 1996.
- [60] L. Gong. *Java 2 Platform Security Architecture - Version 1.0*. 2550 Garcia Avenue, Mountain View, CA 94042-1100 U.S.A, October 1998.
- [61] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Sun Microsystems, Inc., 2550 Garcia Avenue, Mountain View, CA 94042-1100 U.S.A, 1.0 edition, August 1996.
- [62] S. D. Gribble, M. W. R. von Behren, E. A. Brewer, D. Culler, N. Borisov, S. Czerwinski, R. Gummadi, J. Hill, A. Joseph, R.H. Katz, Z. M. Mao, S. Ross, , and B. Zha. The Ninja Architecture for Robust Internet-Scale Systems and Services. *Computer Networks*, 1999. Special Issue on Pervasive Computing.
- [63] A. S. Grimshaw, W. A. Wulf, J. C. French, A. C. Weaver, and P. F. Reynolds Jr. Legion: The Next Logical Step Toward a Nationwide Virtual Computer, August 1994. CS-94-21.
- [64] T. R. Gruber. A Translation Approach to Portable Ontology Specifications. *Knowledge Acquisition*, 5:119–220, 1993.
- [65] M. Gudgin, M. Hadley, J. Moreau, and H. Frystyk. *SOAP Version 1.2 - Part 1: Messaging Framework*, 2001.
- [66] J. C. Haartsen. Bluetooth - The universal radio interface for ad-hoc, wireless connectivity. *Ericsson Review*, (3):110–117, 1998.
- [67] J. C. Haartsen. The Bluetooth Radio System. *IEEE Personal Communications Magazine*, 7(1):28–36, February 2000.

- [68] M. Hachman. Reconfigurable Processors: Computing's Holy Grail. *Dr. Dobbs's*, October 2000.
- [69] G. Hamilton. *Java Beans*. Sun Microsystems, Inc., 2550 Garcia Avenue, Mountain View, CA 94042-1100 U.S.A, July 1997.
- [70] M. Hasenstein. *The Logical Volume Manager (LVM)*. SuSE, Inc., 2001.
- [71] B. Hirsbrunner. Cola: A coordination language for heuristic parallel programming. In *SPP-IF/PPI Closing Conference*, Lausanne, Switzerland, March 19-20 1996.
- [72] B. Hirsbrunner, M. Aguilar, and O. Krone. CoLa: A Coordination Language for Massive Parallelism. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, Los Angeles, California, August 14-17 1994.
- [73] F. Hohl, U. Kubach, A. Leonhardi, K. Rothermel, and M. Schweh. Next century challenges: Nexus - an open global infrastructure for spatial-aware application. *ACM*, 1999.
- [74] P. Houdak. Conception, Evolution, and Application of Functional Programming Languages. *ACM Computing Surveys*, 21(3), September 1989.
- [75] J. Hughes. Why Functional Programming Matters. *Computer Journal*, 32(2):98-107, 1989.
- [76] Institute of Electrical and Electronics Engineers, New York. *IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Standard 754-1985*, 1985. IEEE Standards Committee 754.
- [77] R. Jagannathan and E. A. Ashcroft. Fault Tolerance in Parallel Implementations of Functional Languages. In IEEE Computer Society, editor, *21st Fault-Tolerant Computing Symposium*, Montreal, Canada, June 1991. IEEE.
- [78] R. Jagannathan, C. Dodd, and I. Agi. GLU: A High-Level System for Granular Data-Parallel Programming. Technical report, October 1995.
- [79] S. P. Jones, J. Hughes, L. Augustsson, D. Barton, B. Boutel, W. Burton, J. Fasel, K. Hammond, R. Hinze, P. Hudak, T. Johnsson, M. Jones, J. Launchbury, E. Meijer, J. Peterson, A. Reid, C. Runciman, and P. Wadler. *Report on the Programming Language Haskell 98*, February 1999.
- [80] A. Kaminsky. Jini Print Service Design. <http://www.developer.jini.org/exchange/users/jpgwg/JiniPrintService/design20000215/index.html>, February 2000.
- [81] J. Kao. *Developer's Guide to Building XML-based Web Services with the Java 2 Platform, Enterprise Edition (J2EE)*, June 2001.
- [82] B.W. Kernighan and D.M. Ritchie. *The C Programming Language*. Prentice-Hall, Englewood Cliffs N.J., 1978.
- [83] E. Kiciman and A. Fox. Using Dynamic Mediator to Integrate COTS Entities in a Ubiquitous Computing Environmen. *Springer-Verlag*, 2000.

- [84] M. Klein, D. Fensel, F. van Harmelen, and I. Horrocks. The Relation between Ontologies and Schema-Languages: Translating OIL-specifications in XML-Schema. In *Proceedings of the ECAI'00 Workshop on Applications of Ontologies and Problem-Solving Methods*, Berlin, August 2000.
- [85] J. Klensin. *Simple Mail Transfer Protocol*, 2001. RFC 2821.
- [86] K. Knight and S. Luk. Building a Large Knowledge Base for Machine Translation. In *Proceedings of the American Association of Artificial Intelligence Conference AAAI-94*, Seattle, WA, 1994.
- [87] O. Krone. *STL and Pt-PVM: Concepts and Tools for Coordination of Multi-threaded Applications*. PhD thesis, University of Fribourg, 1997. No. 1191.
- [88] O. Krone, M. Aguilar, and B. Hirsbrunner. Bridging the Gap: A Generic Distributed Coordination Model for Massively Parallel Systems. In *Proceedings of the SIPAR'95 Workshop on Parallel and Distributed Computing*, Biel-Bienne, Switzerland, October 6 1995.
- [89] O. Krone, M. Aguilar, B. Hirsbrunner, and V. S. Sunderam. Integrating Coordination Features in PVM. In *First International Conference on Coordination Models and Languages*, Cesna, Italy, April 15-17 1996.
- [90] O. Krone, M. Aguilar, and C. Renevey. The CoLA Approach: a Coordination Language for Massively Parallel Systems. In Marc Aguilar, editor, *Proceedings of the '94 SIPAR-Workshop on Parallel and Distributed Computing*, pages 29–33. University of Fribourg, October 14 1994.
- [91] O. Krone, F. Chantemargue, T. Dagaëff, and M. Schumacher. Coordinating Autonomous Entities with STL. *The Applied Computing Review*, Special issue on Coordination Models Languages and Applications, 1998.
- [92] O. Krone, F. Chantemargue, T. Dagaëff, M. Schumacher, and B. Hirsbrunner. Coordinating Autonomous Entities. In *Proceedings of the ACM Symposium on Applied Computing (SAC'98). Special Track on Coordination, Languages and Applications*, pages 149–158, Atlanta, Georgia, USA, February 27 - March 1 1998.
- [93] O. Krone and A. Josef. Using CORBA in the Web Operating System. In *Proceedings of the Workshop on Distributed Communities on the Web, Quebec, Canada*. LNCS, Springer Verlag, June 2000.
- [94] O. Krone, M. Raab, and B. Hirsbrunner. Load Balancing For Network Based Multi-threaded Applications. In *Proceedings of EuroPVM/MPI'98: 5th European PVM-MPI Conference*, Liverpool, England, September 7–9 1998.
- [95] O. Krone and S. Schubiger. WebRes: Towards a Web Operating System. In *Proceedings of Fachtagung Kommunikation in Verteilten Systemen, KIVS '99*, Darmstadt, Germany, March 2–5 1999.
- [96] P. Kropf. Overview of the Web Operating System (WOS) project. In *Advanced Simulation Technologies Conference (ASTC1999)*, pages 350–356, San Diego, California, USA, April 1999.

- [97] P. Kropf, J. Plaice, and H. Unger. Towards a Web Operating System (WOS). Technical report, Université Laval, Département d'Informatique, Sainte-Foy (Québec), Canada, G1K 7P4, 1997.
- [98] S. B. Lamine and J. Plaice. Simultaneous Multiple Versions: The Key to the WOS. In *Proceedings of Workshop on Distributed Computing on the WEB*, Rostock, Germany, June 22-23 1998.
- [99] O. Lassila and R. R. Swick. *Resource Description Framework (RDF) Model and Syntax Specification*. W3C, February 1999.
- [100] S. Le Peutrec and M. Courant. Instruments pour la vie artificielle. In *Proceedings of the Eleventh French-speaking Congress of Human Computer Interaction*, Montpellier, France, November 1999.
- [101] S. Le Peutrec and M. Courant. Revisiting HCI for Networking Computers: A First Breakthrough for Artificial Biology. In *Proceedings of the International ICSC Symposium on Biologically Inspired Systems (BIS'2000)*, Wollongong, Australia, December 2000.
- [102] S. J. Leffler, M. K. McKusick, M. J. Karels, and J. S. Quarterman. *The Design and Implementation of the 4.3BSD UNIX Operating System*. Addison-Wesely Publishing Company, Inc., Reading, Massachusetts, 1990.
- [103] D. A. Lewine. *POSIX Programmer's Guide*. O'Reilly & Associates, Inc., 1994.
- [104] J. Loviscach, H. Schulz, and K. Viola. Sunspiration. *c't, Magazin für Computer Technik*, (4):92-97, February 2002.
- [105] S. Luke and J. Heflin. *SHOE 1.1 - Proposed Specification*, April 2000.
- [106] D. MacKenzi and B. Elliston. Autoconf: Creating Automatic Configuration Scripts, December 1998. Edition 2.13.
- [107] D. MacKenzi and B. Elliston. GNU Automake, August 2001. Version 1.5.1a.
- [108] S. Maffioletti, S. Schubiger, and B. Hirsbrunner. Towards a Homogeneous Environment for Ubiquitous Interactive Devices. Technical report, Department of Informatics, University of Fribourg, Switzerland, 2001.
- [109] G. Malkin. *RIP Version 2*, 1994. RFC 1723.
- [110] F. Mattern. *Internet@Future, Jahrbuch Telekommunikation und Gesellschaft 2001*, volume 9, chapter Ubiquitous Computing, pages 52-61. Internet@Future, 2001.
- [111] F. Mattern. *Mobile Internet, Tagungsband 6. Deutscher Internet-Kongress*, chapter Ubiquitous Computing - Der Trend zur Informatisierung und Vernetzung aller Dinge, pages 107-119. dpunkt-Verlag, September 2001.
- [112] F. Mattern. *Leben in der e-Society - Computerintelligenz für den Alltag*, chapter Ubiquitous Computing - Vision und technische Grundlagen. Springer-Verlag, January 2002.
- [113] H. Maturana and F.J. Varela. *Autopoiesis and Cognition: the realization of the living*. Reidel, Boston, MA, 1980.

- [114] G. Mazzola. Humanities@EncycloSpace - Der enzyklopedische Wissensraum zur Informationstechnologie, February 1997. Empfehlungen an den Schweizerischen Wissenschaftsrat, Bern.
- [115] N. Medvidovic and R. N. Taylor. A Framework for Classifying and Comparing Architecture Description Languages. In M. Jazayeri and H. Schauer, editors, *Proceedings of the Sixth European Software Engineering Conference (ESEC/FSE 97)*, pages 60–76. Springer-Verlag, 1997.
- [116] B. Meyer. *Introduction to the Theory of Programming Languages*. Prentice-Hall, Inc., Upper Saddle River, New Jersey, 1991.
- [117] B. Meyer. *Object Oriented Software Construction*. Prentice-Hall, Inc, Upper Saddle River, New Jersey, 2nd edition, 1997.
- [118] J. Micallef and G. M. Clemm. The Asgard System: Activity-Based Configuration Management. In *SCM-6 Workshop*, volume LNCS1167, pages 175–187. Springer-Verlag, March 1996.
- [119] Microsoft Corporation, Redmond, WA 98052-6399, USA. *Enterprise-Class Storage in Microsoft Windows 2000*, 1999.
- [120] Microsoft Corporation, Redmond, WA 98052-6399, USA. *Plug and Play Design Specification for IEEE 1394*, March 1999. Version 1.0c.
- [121] Microsoft Corporation, Redmond, WA 98052-6399, USA. *Plug and Play for Windows 2000*, May 1999.
- [122] Microsoft Corporation, Redmond, WA 98052-6399, USA. *Installing Drivers and Utilities without Rebooting on Windows 2000*, August 2000.
- [123] Microsoft Corporation, Redmond, WA 98052-6399, USA. *Microsoft Office XP Smart Tag SDK v1.1*, 2001. <http://www.microsoft.com/Office/developer/platform/smartag.asp>.
- [124] M. Migliardi, S. Schubiger, and V. Sunderam. A Distributed JavaSpace Implementation for HARNESS. *Journal of Parallel and Distributed Computing*, (Special Issue - Java on Clusters), July-August 2000. Academic Press.
- [125] P. Mockapetris. *Domain Names - Implementation and Specification*, 1987. RFC 1035.
- [126] J. P. Morrison. *Flow-Based Programming: A New Approach To Application Development*. Van Nostrand Reinhold, 115 Fifth Avenue, New York, NY 10003, 1994.
- [127] J. Myers and M. Rose. *Post Office Protocol - Version 3*, 1996. RFC 1939.
- [128] P. Naur and B. Randell. In *Proceedings, NATO Conference on Software Engineering*, Garmisch, Germany, 1969.
- [129] Network Storage Solution, Inc. *The Future of NAS and SAN*, January 2001.
- [130] D. A. Normann. *The Invisible Computer*. MIT Press, 1999.

- [131] Object Management Group. The Common Object Request Broker, Architecture and Specification. Technical report, OMG and X/Open, 1992.
- [132] Object Management Group. *CORBA Services: Common Object Services Specification*, 1998.
- [133] C. Perkins. *IP Mobility Support*, 1996. RFC 2002.
- [134] J. Postel and J. Reynolds. *File Transfer Protocol (FTP)*, 1985. RFC 959.
- [135] J. Powell and T. Maxwell. *Integrating Office XP Smart Tags with Microsoft .NET Platform*. Microsoft Corporation, Redmond, WA 98052-6399, USA, October 2001.
- [136] M. R. Quillian. Computational Linguistics. *Communications of the ACM*, 12(8), August 1969.
- [137] A. S. Rao and M. P. Georgeff. BDI-agents: from theory to practice. In *Proceedings of the First Intl. Conference on Multiagent Systems*, San Francisco, 1995.
- [138] Rational Software. *Object Constraint Language Specification*, September 1997.
- [139] C. Renevey and M. Courant. Phi-Cola: Towards a Logico-Physical Coordination Language. In *Workshop on Future and Practice of Autonomous Systems*, Monte Verita, Switzerland, September 1995.
- [140] A. Robert. *EMuds: Adaption in Text-Based Virtual Worlds*. PhD thesis, University of Fribourg, Fribourg, Switzerland, 2000. No. 1272.
- [141] S. Schubiger and O. Krone. Interactive Resource Sharing on the Web. In *Proceedings of Workshop on Distributed Computing on the WEB*, Rostock, Germany, June 22-23 1998.
- [142] S. Schubiger, O. Krone, and B. Hirsbrunner. WebComs: Transactions as Object-Flow Networks for the WOS. In *Proceedings of Workshop on Distributed Computing on the WEB*, Rostock, Germany, June 21-23 1999.
- [143] M. Schumacher. *Designing and Implementing Objective Coordination in Multi-Agent System*. PhD thesis, University of Fribourg, Fribourg, Switzerland, 1999. No. 1291.
- [144] M. Schumacher, F. Chantemargue, and B. Hirsbrunner. The STL++ Coordination Language: a Base for Implementing Distributed Multi-Agent Applications. In *Proceedings of the Third International Conference on Coordination Models and Languages*, Amsterdam, The Netherlands, April 26–28 1999.
- [145] M. Schumacher, F. Chantemargue, O. Krone, and B. Hirsbrunner. STL++: A Coordination Language for Autonomy-based Multi-Agent Systems. Technical report, Computer Science Department, University of Fribourg, Fribourg, Switzerland, March 1998.
- [146] M. Schumacher, F. Chantemargue, O. Krone, S. Schubiger, and B. Hirsbrunner. The STL++ Coordination Language: Application to Simulating the Automation of a Trading System. In *Proceedings of the First International Conference on Enterprise Information Systems, ICEIS'99*, Setubal, Portugal, March 27–30 1999.

- [147] M. Sharples. *Computer and Thought: A Practical Introduction to Artificial Intelligence*. The MIT Press, Cambridge, Massachusetts, 1989.
- [148] C. Siemers. Rechenfabrik - Ansätze für extrem parallele Prozessoren. *c't, Magazin für Computer Technik*, 15:170–179, 2001.
- [149] P. Siering. Das Microsoft-Internet. *c't, Magazin für Computer Technik*, (4):86–91, February 2002.
- [150] C. Simonyi. Intentional Programming - Innovation in the Legacy Age. In *Proceedings of the IFIP WG 2.1 meeting*. IFIP WG 2.1 meeting, June 1996.
- [151] J. Snell. MS SOAP SDK vs IBM SOAP4J: Comparison & Review . windows.oreilly.com, 2001.
- [152] J. F. Sowa, editor. *Principles of Semantic Networks*. Morgan Kaufmann Publishers, San Mateo, California, 1991.
- [153] S.Schubiger, S.Maffioletti, A. Tafat-Bouzid, and B. Hirbrunner. Providing Service in a Changing Ubiquitous Computing Environment. In *Workshop on Infrastructure for Smart Devices - How to Make Ubiquity an Actuality*, September 2000.
- [154] R. M. Stallman and R. McGrath. *GNU Make - A Program for Directing Recompilation*. Free Software Foundation, Boston, MA, USA, April 2000.
- [155] L. Sterling and E. Shapiro. *The Art of Prolog*. MIT Press, 1986.
- [156] M. Stöbe. USB 2.0 kontra FireWire. *c't, Magazin für Computer Technik*, 15:128–133, 2001.
- [157] B. Stroustrup. *The C++ Programming Language*. Addison-Wesely Publishing Company, Inc., Reading, Massachusetts, second edition, 1993.
- [158] K. Stufflebeam, B. Saunders, and C. Cruz. CardBay: Blueprint for Tomorrow's Modular Mobile Add-in Technology. Technical report, PCMCIA, March 2001.
- [159] Sun Microsystems, Inc., 2550 Garcia Avenue, Mountain View, CA 94042-1100 U.S.A. *Distributed Event Specification, Revision 1.0*, July 1998.
- [160] Sun Microsystems, Inc., 2550 Garcia Avenue, Mountain View, CA 94042-1100 U.S.A. *Distributed Leasing Specification, Revision 1.0*, July 1998.
- [161] Sun Microsystems, Inc., 2550 Garcia Avenue, Mountain View, CA 94042-1100 U.S.A. *Java Remote Method Invocation Specification, Revision 1.50, JDK 1.2*, October 1998.
- [162] Sun Microsystems, Inc. *JavaSpaces Specification, Revision 1.0*. 2550 Garcia Avenue, Mountain View, CA 94042-1100 U.S.A, March 1998.
- [163] Sun Microsystems, Inc., 2550 Garcia Avenue, Mountain View, CA 94042-1100 U.S.A. *Transaction Specification, Revision 1.0*, July 1998.
- [164] Sun Microsystems, Inc., 2550 Garcia Avenue, Mountain View, CA 94042-1100 U.S.A. *Jini Architecture Specification - Version 1.1*, October 2000.

- [165] Sun Microsystems, Inc., 2550 Garcia Avenue, Mountain View, CA 94042-1100 U.S.A. *Jini Device Architecture Specification - Version 1.1*, October 2000.
- [166] Sun Microsystems, Inc., 2550 Garcia Avenue, Mountain View, CA 94042-1100 U.S.A. *Jini Technology Core Platform Specification - Version 1.1*, October 2000.
- [167] Sun Microsystems, Inc., 2550 Garcia Avenue, Mountain View, CA 94042-1100 U.S.A. *Open Net Environment (ONE) Software Architecture*, 2001.
- [168] C. Szyperski. *Component Software - Beyond Object-Oriented Programming*. Addison-Wesely Publishing Company, Inc., Reading, Massachusetts, 1998.
- [169] E. Taft, S. Chernicoff, and C. Rose. *PostScript Language Reference*. Addison-Wesely Publishing Company, Inc., Reading, Massachusetts, third edition, February 1999.
- [170] A. S. Tanenbaum. *Operating Systems: Design and Implementaiton*. Prentice-Hall, Inc., Upper Saddle River, New Jersey, 2nd edition, 1997.
- [171] A. S. Tanenbaum. *Modern Operating Systems*. Prentice-Hall, Inc., Upper Saddle River, New Jersey, 2nd edition, 2001.
- [172] A. S. Tanenbaum, R. van Renesse, H. van Staveren, G. J. Sharp, S. J. Mullender, J. Jansen, and G. van Rossu. The Amoeba Distributed Operating System. In *Proc. Workshop on Communication Networks and Distributed Systems Within the Space Environment*, Noordwijk, The Netherlands, October 1989. European Space Agency.
- [173] T. Tewell. Fire Wire: The IEEE 1394 Serial Bus. *Dr. Dobb's Journal*, Septmeber 1997.
- [174] H. S. Thompson, D. Beech, M. Maloney, and N. Mendelsohn. XML Schema Part 1: Structures. <http://www.w3.org/TR/xmlschema-1/>, May 2001.
- [175] UDDI.org. *UDDI Technical White Paper*, September 2000.
- [176] H. Unger, P. Kropf, and J. Plaice. Load Sharing for a Web Operating System. Technical report, Universität Rostock, Fachbereich Informatik, D-18051 Rostock, Germany, 1997.
- [177] C. Vawter and E. Roman. *J2EE vs. Microsoft .NET - A comparision of building XML-based web services*, June 2001.
- [178] W.W. Wadge. The Lucid Primer , 1995.
- [179] J. Waldo. The Jini Architecture for Network-Centric Computing. *Communication of the ACM*, 42(7), July 1999.
- [180] P. Wegner. Interactive Foundations of Computing, April 1997. Brown University.
- [181] M. Weiser. The computer for the 21st century. *Scientific American*, September 1991.
- [182] J. Whitehead. Collaborative Authoring on the Web: Introducing WebDAV. *Bulletin of the American Society for Information Science*, 25(1):25–29, 1998.
- [183] J. Whitehead and M. Wiggins. WebDAV: IETF Standard for Collaborative Authoring on the Web. *IEEE Internet Computing*, pages 34–40, September/October 1998.

- [184] L. B. Wilson and R. G. Clark. *Comparative Programming Languages*. Addison-Wesely Publishing Company, Inc., Addison-Wesely Publishing Company, Inc., 1988.
- [185] N. Wirth. From Programming Language Design To Computer Construction. *Communications of the ACM*, 28(2), 1985.
- [186] M. Wooldridge and N.R. Jennings. Agent Theories, Architectures, and Languages: a Survey. In M. Wooldridge and N.R. Jennings, editors, *Intelligent Agents*, number 890 in LNCS, pages 1–39. Springer Verlag, 1995.

Simon Schubiger-Banz

Bd de Pérolles 17
1700 Fribourg
Switzerland
++41 (0) 26 321 55 94 (Private)
++41 (0) 79 66 33 119 (Mobile)
simon.schubiger@swisscom.com
<http://diuf.unifr.ch/~schubige>

Curriculum Vitae

Personal information

Date of birth: 19th January 1972
Place of birth: Zurich, Switzerland
Nationality: Swiss
Marital status: Married to Andrea Banz Schubiger
Children: Noël Elia, 24th Septemeber 1999
Languages: Swiss German: first language
German: fluently spoken and written
French: fluently spoken and written
English: fluently spoken and written

Education

9.1994 - 7.1998 University of Fribourg, Switzerland
- Computer Science
- Economics
9.1993 - 4.1994 EPFL (Swiss Technology Institute), Lausanne
- Micro techniques
7.1985 - 7.1992 Kantonsschule Beromünster (High School and College)
7.1979 - 7.1985 Primary School, Buchrain and Littau

Diplomas

10.1998 Diploma in computer science of the University of Fribourg
7.1992 Maturity diploma (type: scientific) of the Kantonsschule Beromünster

Practical Experience

2001 - now Software engineer at Swisscom Innovations, Bern
1997 - 2001 Teaching assistant in computer science for scientists (1997), distributed and parallel systems (1998), operating systems (1999) and theory of programming languages (2000)
1998 - 2001 Member of the science faculty council
1998 - 2001 Database development for the science faculty
Summer 1999 Software development in the Harness team, Emory University, Atlanta, GA
Summer 1996 Multimedia development at a2i, Oron
1992 - 1996 Software engineer at Quix Computerware AG, Ebikon
- MacOS port to NeXT Hardware
- MacOS port to IBM, Motorola, FirePower Hardware
- Embedded system development
- Unix driver development
1989 - 1992 Repairman for computer equipment at Büro Vögtlin, Lucerne
1988 - now Various part-time jobs in graphics design and multimedia

Current Position

- Software engineer at Swisscom Innovations, Bern

Current Employer

Swisscom AG
Innovations
Ostermundigenstrasse 93
3006 Bern
Switzerland

Research Interests

- Web computing
- Knowledge processing and representation
- Distributed computing
- Languages and compilers

Publications

“A Model for Software Configuration in Ubiquitous Computing Environments”

S. Schubiger, Béat Hirsbrunner

Proceedings of Pervasive 2002, August 2002, ETHZ Zurich, Switzerland

“Providing Service in a Changing Ubiquitous Computing Environment”

S. Schubiger, S. Maffioletti, Amine Tafat-Bouزيد, Béat Hirsbrunner

Workshop Notes of the Workshop on Ubiquitous Computing, August 2000, International Computer Science Institute, Berkeley, CA, USA

“A Resource Classification System for the WOS”

S. Schubiger

Proceedings of the Workshop on Distributed Communities on the WEB, July 2000, Quebec, Canada

“A Distributed JavaSpace Implementation for HARNESS”

M. Migliardi, S. Schubiger, V. Sunderam

Journal of Parallel and Distributed Computing, July-August 2000, Special Issue - Java on Clusters, Academic Press

“WebComs: Transactions as Object-Flow Networks for the WOS”

S. Schubiger, O. Krone, B. Hirsbrunner

Proceedings of the Workshop on Distributed Computing on the WEB, July 1999, Rostock, Germany

“The STL++ Coordination Language: Application to Simulating the Automation of a Trading System”

M. Schumacher, F. Chantemargue, O. Krone, S. Schubiger, B. Hirsbrunner

Proceedings of the First International Conference on Enterprise Information Systems, ICEIS'99, 1999, Barcelona, Spain

“WebRes: Towards a Web Operating System”

O. Krone, S. Schubiger

Proceedings of Fachtagung Kommunikation in Verteilten Systemen, KIVS '99, 1999, Darmstadt, Germany

“Interactive Resource Sharing on the Web”

S. Schubiger, O. Krone

Proceedings of the Workshop on Distributed Computing on the WEB, July 1998, Rostock, Germany