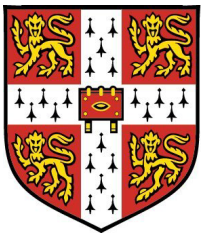
Abstracting Application-Level Security Policy for Ubiquitous Computing

David Jonathan Scott

Robinson College

This dissertation is submitted for the degree of

Doctor of Philosophy
at the
University of Cambridge



Declaration

The dissertation is not substantially the same as any I have submitted for a degree or diploma or any other qualification at any other university. Further, no part of the dissertation has already been or is being concurrently submitted for any such degree, diploma or other qualification.

Chapter 3 and Chapter 4 are similar to papers [145, 144] co-authored with Alan Mycroft (my supervisor) and Alastair Beresford. Apart from a small piece of code written by Alastair (highlighted in the text) all work described in these chapters was developed entirely by me, in discussion with my supervisor. The work described in Chapter 5 was built upon original ideas contained within papers [146, 148, 147] co-authored with Richard Sharp. However, the ideas described together with all the implementation and analysis is substantially my own work.

Other chapters are solely my own work even if they share ideas developed jointly with my supervisor. The Section "Author Publications" lists all publications sharing ideas with this thesis along with authorship.

This dissertation contains less than 62,000 words, including appendices, tables, footnotes, equations and bibliography. It contains 60 figures.

This work was supported by the Schiff Foundation and AT&T Laboratories Cambridge Ltd.

David Scott, September 2004.

Author Publications

The research presented in this thesis has also been published in the following papers (in chronological order):

SCOTT D. AND SHARP R. Abstracting Application-Level Web Security.[1] In *The Eleventh International World Wide Web Conference Proceedings* (WWW2002) (2002), pages 396–407, ACM Press.

SCOTT D. AND SHARP R. Developing Secure Web Applications. In *IEEE Internet Computing Magazine* special issue on *The Technology of Trust* (2002), pages 38–45, ©2002 IEEE.

SCOTT D., BERESFORD A. AND MYCROFT A. Spatial Security Policies for Mobile Agents in a Sentient Computing Environment.[2] In *Proceedings of FASE* (2003), vol. 2621 of *LNCS*, ©2003 Springer-Verlag.

SCOTT D., BERESFORD A. AND MYCROFT A. Spatial Policies for Sentient Mobile Applications. In *Proceedings of IEEE 4th International Workshop on Policies for Distributed Systems and Networks* (2003), pages 147–157, ©2003 IEEE.

[1] Received the *Best Paper* award at WWW2002.

[2] Received the European Association of Software Science and Technology (EASST) *Best Software Science* Paper award.

SCOTT D. AND SHARP R. Specifying and Enforcing Application-Level Web Security Policies. In *IEEE Transactions on Knowledge and Data Engineering*, Volume 15, Issue 4, pages 771–783, July–Aug 2003, ©2003 IEEE.

Abstract

In the future world of Ubiquitous Computing, tiny embedded networked computers will be found in everything from mobile phones to microwave ovens. Thanks to improvements in technology and software engineering, these computers will be capable of running sophisticated new applications constructed from mobile agents. Inevitably, many of these systems will contain *application-level* vulnerabilities; errors caused by either unanticipated *mobility* or *interface behaviour*. Unfortunately existing methods for applying security policy – network firewalls – are inadequate to control and protect the hordes of vulnerable mobile devices. As more and more critical functions are handled by these systems, the potential for disaster is increasing rapidly.

To counter these new threats, this thesis champions the approach of using new application-level security policy languages in combination to protect vulnerable applications. Policies are abstracted from main application code, facilitating both analysis and future maintenance. As well as protecting existing applications, such policy systems can help as part of a security-aware design process when building new applications from scratch.

Three new application-level policy languages are contributed each addressing a different kind of vulnerability. Firstly, the policy language MRPL allows the creation of *Mobility Restriction Policies*, based on a unified spatial model which represents both physical location of objects as well as virtual location of mobile code. Secondly, the policy language SPDL-2 protects applications against a large number of common errors by allowing the specification of per-request/response validation and transformation rules. Thirdly, the policy language SWIL allows interfaces to be described as automata which may be analysed statically by a model-

checker before being checked dynamically in an application-level firewall. When combined together, these three languages provide an effective means for preventing otherwise critical application-level vulnerabilities.

Systems implementing these policy languages have been built; an implementation framework is described and encouraging performance results and analysis are presented.

Acknowledgements

Firstly I'm very grateful to my supervisor, Alan Mycroft, whose guidance and advice have been invaluable throughout my time as a student in Cambridge. Thanks are also due to Andy Hopper who admitted me as a PhD student in the first place and who allowed me the freedom to pursue my interests.

Thanks to all my fellow students (in both the Laboratory for Communication Engineering and the Computer Laboratory) and colleagues (in both AT&T Laboratories Cambridge and Intel Research Cambridge) who have been so inspiring over the last few years. Particular thanks are due to Alastair Beresford for all his proof-reading efforts and Richard Sharp for supplying coffee and interesting discussion. Thanks are also due to Kieran Mansley, Alastair Tse, Alastair Beresford and Anil Madhavapeddy for all their ... relaxing... late-afternoon games of Quake III up in SN17.

Finally, thanks to my family for all their encouragement and support; they'll be glad that perhaps now I can finally get myself a proper job!

Contents

CHAPTER 1

Introduction

Tiny embedded computers are found in a host of everyday items including wristwatches, alarm clocks, mobile phones, network routers and washing machines. Very few of these devices are networked. The situation is changing rapidly; the recent development of integrated on-chip wireless network interfaces will soon drive the cost of networking these embedded computers to almost nothing. Hordes of mobile embedded devices will soon be able to run the same style of distributed programs found today on networked PCs. This is the world of *Ubiquitous Computing*.

Many researchers believe that an environment permeated with powerful, invisible, networked computers will be able to support whole new types of useful applications. However as well as great potential for useful applications, there is also great potential for harmful ones. Already figures from CERT (see Figure 1.1) show an alarmingly rapid rise in the number of computer security incidents – typically infections with viruses and trojans – a situation bound to get worse as networked computers proliferate. So-called "zero-day exploits" where malicious code to exploit a vulnerability is released before the vendor is made aware of the flaw are becoming common. Recently "Cabir" [19], the first mobile phone virus, has been observed by anti-virus software firms. Although this particular virus

contained no malicious payload, future mobile viruses could do anything from sending vast numbers of premium-rate text messages from the victim's phone (incurring the relevant charges, as happened with the ill-fated copy protection mechanism in the mobile game *Mosquitos* [109]) to spreading automatically through the contact phone numbers stored in the handset. With only a few networked computers per person (i.e. desktop, laptop, PDA, mobile phone) users can always exercise the ultimate sanction by pulling the plug on an aberrant machine. What sanction is left when there are thousands of networked computers per person and some of them are safety or mission-critical?

Traditionally networks are secured by surrounding them with firewalls which filter network packets depending on their source and destination. In a recent report [127], Gartner analyst John Pescatore wrote that although firewalls have been effective so far at blunting network-level attacks, the popularity of HTTP tunnelling and Web-Services will push future attacks up to the application-level. In the near future old-style firewalls will likely cease to be effective. Users need to protect and control their applications with new types of *application-level* security policies.

> The thesis of this work is that abstracting application-level security policy (which we define to be policy written to govern both the *mobility* and *interface behaviour* of systems) is an effective technique for guarding against application-level vulnerabilities which would otherwise plague future ubiquitous computing systems.

This chapter begins with an outline of relevant hardware trends in Section 1.1 providing the context for a description of the field of Ubiquitous Computing in Section 1.2. Section 1.3 describes some terminology useful for discussing distributed systems while Section 1.4 gives a brief outline of research into policy systems. Section 1.5 describes the trend towards ever greater software mobility which is followed by a discussion in Section 1.6 of how these factors have all led to traditional network firewalls being unable to effectively protect vulnerable applications. Abstracting application-level security policy is described in Section 1.7 and motivated as a useful technique in Section 1.8. Section 1.9 describes the contribution and Section 1.10 the structure of the rest of this thesis.

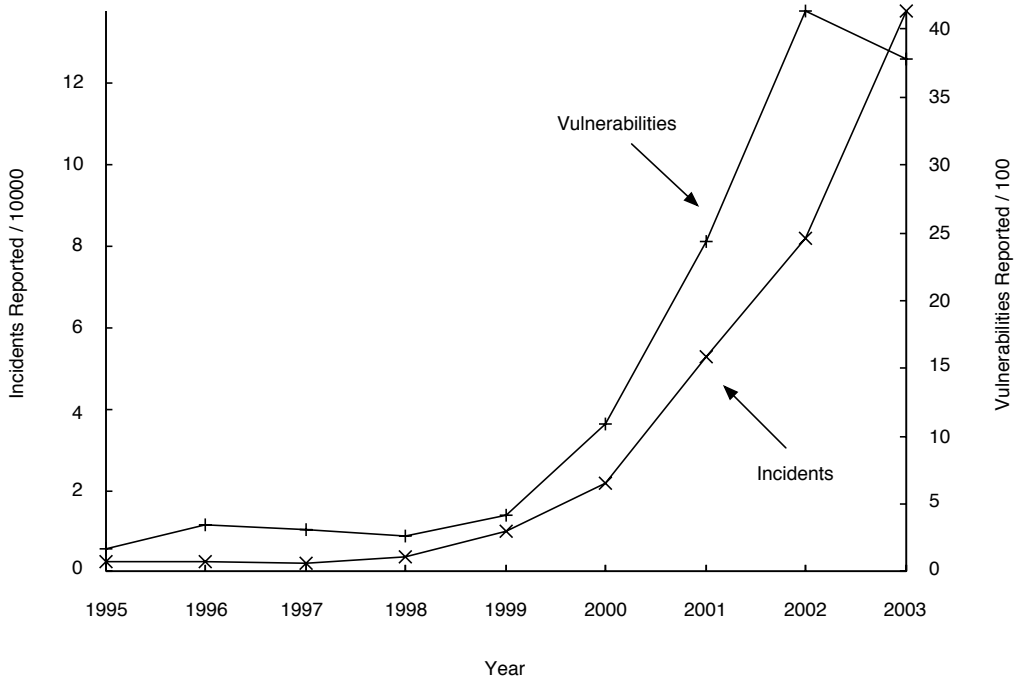


Figure 1.1: CERT/CC Statistics on Reported Incidents and Vulnerabilities obtained from `http://www.cert.org/stats/` for the period from 1995–2003.

1.1 Hardware Trends

The Semiconductor Industry Association [150] predicted that around 1 billion transistors will be manufactured for every man, woman and child on Earth in the year 2010. These transistors will likely not be in conventional CPUs; Tennenhouse predicted [162] that the production of microcontrollers – tiny embedded CPUs – will outnumber traditional desktop CPUs by 50 to 1 in the year 2000. These microcontrollers are found today in many products including network routers, cellphones, alarm clocks, washing machines, vending machines, heating systems, car engines, security systems and many others. These tiny machines are becoming increasingly sophisticated (following Moore's Law) and some now run Commercial Off-The Shelf (COTS) operating systems like VxWorks [10] or ITRON [4]. Although many of the devices are not currently networked (except network routers) it seems likely that, with the falling cost of wireless networking technologies like Bluetooth [2] and IEEE 802.11 [93] soon these devices will be equipped to communicate with one-another and with the rest of the global Internet. Due to their small size and weight, many of these devices will also be physically mobile, either carried by people (in the form of a laptop), under their own power (in the form of a robot) or even drifting on the wind (in the form of a "Smart Dust" [102] particle).

1.2 Ubiquitous Computing

The potential capabilities of a huge network of communicating, mobile, embedded devices are truly staggering. In August 1999, Neil Gross published an article in Businessweek [78] which contained the prediction:

> Hundreds of thousands of PCs working in concert have already tackled complex computing problems. In the future, some scientists expect spontaneous computer networks to emerge, forming a "huge digital creature".

In his seminal article in Scientific American Weiser described [175] a world in which computers cease to be the focus of attention but instead fade into the background, "weaving themselves into the fabric of everyday life until they are indistinguishable from it". This vision of *Ubiquitous Computing* is perhaps the

one Licklider imagined [110] in his influential 1960 article titled “Man-Computer Symbiosis”:

> The hope is that, in not too many years, human brains and computing machines will be coupled together very tightly, and that the resulting partnership will think as no human brain has ever thought and process data in a way not approached by the information-handling machines we know today.

Unfortunately there are already signs that trouble is brewing with the first generation of networked embedded devices: Arce discusses [15] how it may be possible to remotely compromise and seize control of the computer within a Cisco router. Vulnerabilities in core network elements are bad enough but worse could still be to come: Borriello and Want describe [27] how in the future tiny ingested computers controlled by Web-Services may perform the time-release of drugs into the bloodstream. Vulnerabilities in these systems would be devastating.

1.3 Terminology: Software Layers, Levels and Protocols

Many systems, particularly networked ones, are considered to be constructed in *layers*, sometimes known as *levels*, arranged in a stack. Each layer is a module which uses the interfaces of the layer below in order to expose new interfaces to the layer above. Layers at the same level in different systems communicate using *protocols*. Protocol messages emitted from a particular layer in one system are routed through lower layers on their way to the same layer in peered systems. Layers are intended to be abstract entities; their internal details are hidden and only concrete interfaces are exposed.

The ISO Open Systems Interconnect (OSI) seven layer model is a famous reference design for networked systems[1]. The seven layers, from highest-level to lowest, are: application, presentation, session, transport, network, datalink, physical. The physical layer considers issues such as the voltages required in electrical transmission circuits and the rate at which bits can be modulated. The datalink layer deals with the organisation of bits transmitted, including error-checking and

[1] ISO defined a concrete set of protocols corresponding to their abstract model. Although these protocols never became popular the model is still used today as a reference to compare other systems.

framing. The network layer takes care of message routing across nodes in a network. The transport layer arranges for any necessary data retransmissions in the event of messages being dropped. The session layer considers grouping messages into higher-level structures such as byte streams or request/response messages. Finally the application layer performs everything else, i.e. the "real work" of the application. In this thesis we consider the application-level software to consist of two aspects: a *mobility* aspect which controls the location of code and an *interface behaviour* aspect which controls how a component interacts with other components.

1.4 Policy Systems

Policies are widely used throughout computer systems for many different purposes. In general, a policy system usually consists of a set of high-level rules written in mathematical logic which describe the way a system should behave in different circumstances. For example, an access control policy [138] would describe how and when users may gain access to particular resources. An example access control policy would be one which states (in some concrete mathematical form) that only users who have supplied the correct password and are in a particular group may access the printer between 9am and 5pm on a tuesday. Access control policies are either *discretionary* or *mandatory*. A discretionary policy allows users control over who can access the objects they "own" while a mandatory system is configured by the system administrator and no-one else may change the permissions—not even the user who owns the objects.

Information flow policies are rules which constrain how data may move between applications and application components. Denning and Denning [50] designed a compile-time program analysis phase which checks that information can only legally flow between variables (data may flow by direct assignment or by indirectly influencing the value of another variable) if the variables belong to certain security classes (e.g. data may flow from a variable of class *public* to a variable of class *top secret* but not the other way around).

Role-based access control (RBAC) [73] introduced the concept of a role which is a associated with a particular position within an organisation and granted a set of access rights. Users are assigned to roles either statically or dynamically

(for example, after supplying supplementary passwords) rather than being granted access rights directly.

XACML [66] is a policy system which has recently been defined by an industrial consortium to control access to parts of the XML documents. The process of exchanging XML documents constitutes the core of the Web-Service protocols and together with XML encryption [94] and XML signatures [18], XACML is a significant part of the Web-Service security model. XACML is highly flexible; individual users or roles may be given access to read or write documents at arbitrary granularity, down to individual document elements.

The Policy Description Language (PDL) [111] uses a paradigm called *event-condition-action* (ECA) originally developed for active databases. Events (such as a login attempt) are matched against the ECA rules to generate appropriate actions, which could include generating further events. Simple event combinators are provided to allow rules to match combinators or sequences of events.

Network Firewall policies are similar to ECA policies where events fire when packets are received and the actions include: (i) drop the packet; (ii) reject the packet by sending back an error; (iii) let the packet pass unmodified; or (iv) transmit the packet with some modification.

1.5 Mobile Agents

As networked computers proliferate there is not only a trend towards greater *physical* mobility but also greater *logical* mobility. Over time, many different architectures for distributed software have been proposed including *client-server, remote execution, code on demand* and *mobile agents*. These architectures differ primarily in the amount of code mobility (i.e. logical mobility) they support. Client-server is the most static — supporting no code movement at all; client and server simply communicate via request and response messages. Remote execution allows a client to send a program to the server while in a "code on demand" system a program is downloaded from the server to a client. Mobile agent systems sport the most mobility and flexibility; agents are autonomous programs which decide for themselves when to move between hosts on the network and when to stay in one place and send messages instead.

Mobile Agents are attractive for many reasons [107] including overcoming ef-

fects of latency, lowering bandwidth requirements, increasing fault-tolerance and maximising use of distributed resources. As time passes each of these aspects is becoming more important. Consider latency: unlike bandwidth which can be increased through ever more sophisticated transmission schemes, communications latency has a hard lower-limit determined by the speed of light and the distance between distributed nodes. As machines get faster individual communications become increasingly expensive in terms of CPU cycles. Sending a whole program (as an agent) is therefore more economical than sending multiple requests serially to a remote server. For fault-tolerance, agents can replicate themselves and run on several nodes of a network simultaneously; this ability should be particularly useful as shrinking CMOS feature sizes is leading to more unreliable chips [55]. At the present time data is being generated in vast quantities by scientists (e.g. from genome sequencing and particle physics experiments) by businesses (e.g. from customer profiling) by governments (e.g. from citizen profiling) and by individuals (e.g. in the form of digital media). In many applications data is generated so quickly that it must be processed by distributed nodes in a system known as a computational *Grid* [67]. Autonomous mobile agents are an attractive means of programming these Grid applications [105].

Unfortunately the increased flexibility of mobile agent systems comes with a cost; it is not clear how best to control such a highly mobile, fluid system. There are many open security challenges; for example it is not clear how best to protect hosts from malicious agents and agents from malicious hosts (more details may be found in the literature [141, 41]). However as time passes the lure of Mobile Agents will get ever stronger and more difficult to resist.

1.6 Current Network Firewalls Insufficient

A *firewall* is an entity which imposes policy at the network boundary between systems. The original design of the Internet was based on the *end-to-end* [137] principle in which all intelligence is contained in the hosts while the network itself is dumb. The network consisted entirely of simple packet routing engines corresponding to the *network-level* in the jargon of the ISO/OSI model described in Section 1.3. This design has no explicit firewalls; each host is responsible for its own security and may communicate with any other host. Explicit firewalls

were introduced later in an attempt to centralise security responsibilities rather than leaving security to individual hosts. Unfortunately this creates tension between two groups of users: those charged with security who run the firewalls and those running the actual applications. In such a situation an arms race can easily develop; a firewall blocking a particular protocol provides an incentive to modify the application to *tunnel* the blocked protocol over some other still permitted protocol. It is perhaps no surprise then that Web-Services [75], a new middleware platform, sends all data over HTTP [22]: the same protocol used by the World Wide Web (WWW) and which is almost universally permitted. Therefore as far as any application using Web-Services is concerned, network firewalls are transparent.

Mobility is also a problem for current network firewalls. If mobile agents (possessing logical mobility) are allowed to migrate across a firewall then they are able to send traffic directly to local machines without being subject to network security policy imposed by the firewall. Similarly, programs running on physically mobile machines (e.g. on laptops) may also bypass firewall policy when the machines move. Figure 1.2 shows graphically a representation of a traditional networked environment in which fixed hosts are connected in small trusted networks separated by firewalls. Each isolated local network operates as an *island of trust* and the firewalls form natural *trust boundaries*, imposing policy on the traffic flowing between the networks. Figure 1.3 shows the same diagram augmented with a multitude of mobile agents and devices able to move both physically and logically. Clearly network-level firewalls are no longer sufficient to protect vulnerable applications from mobile attack.

1.7 Abstracting Application-Level Policy

In this thesis the term "application-level policy" is used to refer to policy written to govern the *mobility* and *interface behaviour* of systems. Application-level policy does not deal with lower-level security issues such as the choice of encryption protocol used to prevent eavesdropping or the nature of the packet filtering rules contained within a network firewall. Application-level policy deals with issues such as (i) where can a program execute; (ii) where can it move to; and (iii) what messages may it safely exchange with other programs.

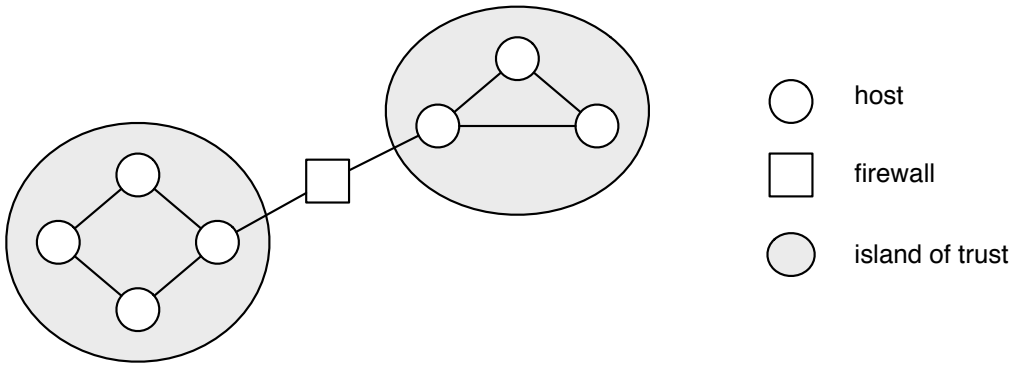


Figure 1.2: Diagram represents traditional distributed computing; fixed hosts are connected in small trusted networks joined together by firewalls.

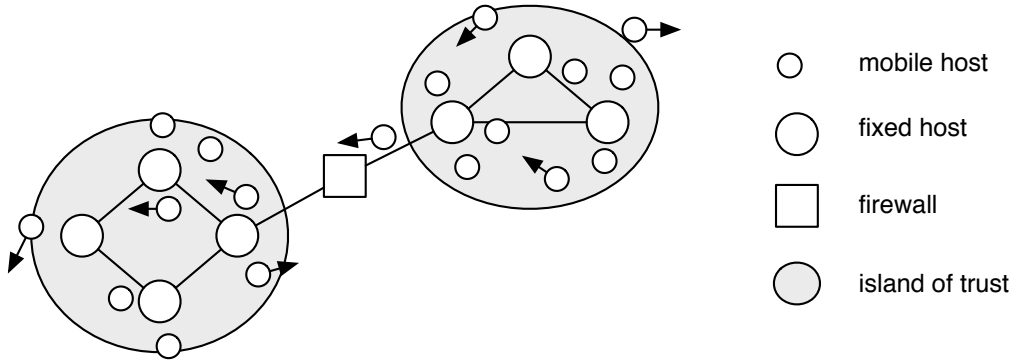


Figure 1.3: The same diagram from Figure 1.2 augmented with a host of mobile devices capable of crossing between islands of trust.

"Abstracting" policy refers to a general technique for maintaining a single policy specification separate from an entire application codebase. The policy is kept in one piece to facilitate reasoning and analysis—a task which would be much more difficult if the policy was entwined with the main application code.

1.8 Motivation for Abstracting Application-Level Policy

Application-level policy is required for three reasons. Firstly, the trend towards greater mobility of both hardware devices and software agents is reducing the effectiveness of current firewall policies written at the network or transport-level. Network-level policies necessarily base their decisions only on the source or destination addresses of traffic while transport-level policies (like those implemented by the OpenBSD stateful firewall [7]) are able to filter traffic based on TCP-level flow information e.g. the presence or absence of a `SYN` flag indicating a connection setup (more will be said about TCP connection setup later in Section 2.6.2). Secondly, modern middleware platforms like Web-Services are being designed specifically to work around network-level firewalls, tunnelling through them by pretending to be normal HTTP traffic[2]. Finally, more and more vulnerabilities are being found and exploited at the application-level; a trend which is predicted to get worse [127]. Application-level policy, rather than network-level policy is required to protect applications.

Abstracting application-level policy affords us a number of advantages. It allows developers to untangle security-related code from a large codebase, facilitating design, analysis and maintenance. In situations where an application is constructed from separate components (consider a shopping cart component used in a web-application) it may be difficult to modify the individual components since they may be written in different languages or only available in binary form..

Abstracting application-level policy can help mitigate the effects of zero-day exploits. When a component vendor wishes to release a new version with a critical bug fixed, great care must be taken to avoid accidentally creating additional problems—a difficult task when the vendor's customers have widely different setups and rely on different features of the component. The vendor is placed in a very difficult position: release a fix early and possibly introduce more problems

[2] "The Net interprets censorship as damage and routes around it." – John Gilmore

with some customers or release late and leave all customers vulnerable to attacks for longer. Abstracting policy offers a third way: customers can write policy, customised for and tested by themselves on their individual configurations, allowing them to protect themselves in the short-term while they wait for an official fix to come from the vendor.

A downside to the approach of abstracting policy is that it requires extra maintenance; i.e. when the application is modified so too must the policy be modified. However this task need not be as onerous as it might seem. Policy maintenance could be simplified using automatic tool support. For example, a tool could be used to track changes to an application source and remind the administrator to check particular parts of the policy when the source is changed. The task of initially creating policy could also be simplified with a tool which passively monitors "normal" application behaviour and produces a default policy which is permissive enough to allow all the behaviour it observes. Such policy could then be customised by a human policy designer.

1.9 Contribution

This thesis explores policy languages and mechanisms to govern both the *mobility* and *interface behaviour* of software at the application-level. This thesis specifically contributes:

1. the Unified Spatial Model (USM): a spatial model capable of representing simultaneously the physical locations of objects such as people and the virtual locations of mobile agents. The USM is the foundation of the Mobility Restriction Policy Language (MRPL) – a language which allows policies to be written to limit the movements of mobile agents in an environment augmented with location sensors, like that displayed graphically in Figure 1.3;

2. the language SPDL-2 which allows the expression of application-level interface security policies in which per-request and response validation and transformation rules can be written to protect web-applications from many kinds of common security vulnerabilities; and

3. the language SWIL which allows the flow of control across an application interface to be described as a sequence of operations accepted by a finite

state automaton. The automaton can be both analysed statically with a model-checker and dynamically checked with an application-level firewall.

Together this model and these policy languages allow abstract application-level policy specifications to be written to protect against many kinds of vulnerability. These application-level policy systems have been implemented; implementation details and performance results are provided.

1.10 Outline

The rest of this thesis is structured as follows: Chapter 2 provides technical background information important for understanding the following chapters. The next four chapters are divided into two groups of two; the first two chapters focus on the *mobility* while the second two focus on the *interfaces* of applications. Chapter 3 begins by describing the Unified Spatial Model (USM) capable of representing both the physical location of computers and the logical locations of mobile agents in the same framework. This model is used subsequently in Chapter 4 as the basis of the Mobility Restriction Policy Language (MRPL) in which policies may be written capable of limiting the movements of agents in both physical and virtual space.

The following two chapters deal with the interfaces of applications. Chapter 5 introduces the policy language SPDL-2 (Security Policy Description Language version 2) which allows the specification of per-request and response validation and transformation rules, protecting applications from many common forms of security vulnerabilities. Chapter 6 introduces SWIL (Stateful Web Interface Language) which allows the flow of control across an application interface to be described as a sequence of inputs accepted by a finite state automaton. SWIL secures an application interface through a combination of up-front model-checking and dynamic policy enforcement.

Chapter 7 describes the implementation of the systems described in Chapters 3 through 6 along with some encouraging performance results. Finally Chapter 8 discusses future work and concludes. Related work and general discussion is provided in the relevant chapters.

CHAPTER 2

Technical Background

This chapter provides detailed technical background information on a number of core concepts used throughout the rest of this thesis, specifically Mobile Agents (in Section 2.1), Location Sensing (in Section 2.2), Interfaces (in Section 2.3), Web Protocols (in Section 2.4), Computer Security (in Section 2.5) and Model-Checking (in Section 2.6).

Previously Section 1.5 motivated the use of mobile agents as an increasingly useful and economical way to structure distributed programs. With this in mind, this chapter begins in Section 2.1 with a detailed description of key Mobile Agent concepts together with some simple application examples. Section 1.1 argued that *physical* mobility (as well as mobile agent-style *logical* mobility) is set to increase significantly as devices continue to shrink and proliferate. Mechanisms to control this physical mobility will necessarily rely on the ability to sense where devices are in physical space. Therefore Section 2.2 describes a number of mechanisms for location sensing and provides several political arguments for why these location sensing devices will become commonplace. Later work in Chapter 3 builds upon this foundation, creating the Unified Spatial Model (USM) capable of representing both the physical locations of objects and the logical locations of agents simultaneously. The model is used in Chapter 4 as the basis for the Mobility

Restriction Policy Language (MRPL) which allows the creation of *mobility restriction policies* capable of limiting the movement of mobile agents throughout an environment augmented with location sensors. We observe that, just as early operating systems had little need for file protections until they became multi-user, mobility policy only becomes critical as systems become more mobile.

In addition to moving between hosts, Mobile Agents are also able to communicate across points known generically as *interfaces*. Section 2.3 describes a simple interface formalism capable of representing a number of common interface operations. The concepts of an interface *proxy* and an interface *firewall* are both introduced. The policy languages SPDL-2 in Chapter 5 and SWIL in Chapter 6 are both implemented by an interface firewall system described in Chapter 7. While the concept of an interface is generic, real-life systems necessarily will use some concrete set of protocols. To facilitate a more interesting discussion (and without loss of generality) this thesis makes the simplifying assumption that future distributed, mobile applications will opt to use the standard web protocols for communication. These protocols are described in detail in Section 2.4. The term *web-application* is introduced to mean any application using the web protocols (HTTP and URIs) for communication and the form of a web-application interface is described using the formalism of Section 2.3. Web interfaces say nothing specific about security and leave a number of crucial interface assumptions undocumented causing a large number of application vulnerabilities; these problems are fixed by the policy languages SPDL-2 in Chapter 5 and SWIL in Chapter 6.

Section 2.5 describes traditional security principles and outlines several reasons why engineering secure systems is difficult in practice. A simple example is used to demonstrate how security is not in general composable and how it is only ever as strong as the weakest part of a system. Recall the thesis statement from Chapter 1 which talked about *application-level* security vulnerabilities. As briefly described in Section 1.3, software is often constructed in a layered fashion, with system software (e.g. network stacks) layered beneath application-specific code. Application-level vulnerabilities are by definition vulnerabilities present only in the application-specific code. Both the mobility restriction policies in Chapter 4 and the interface policy languages in Chapters 5 and 6 are application-level security mechanisms.

property	description
reactive	responds to external stimuli in the environment
autonomous	takes its own decisions
goal-oriented	has a purpose
continuous	runs all the time
communicative	communicates with other agents and/or people
learning	changes its behaviour based on its previous experience
mobile	able to migrate between hosts
flexible	actions are not scripted
character	believable “personality” and emotional state

Figure 2.1: Table of properties of mobile agents as proposed by Franklin et al [71].

Finally, Section 2.6 describes *model-checking*, a technique for the automatic verification of communicating systems. The syntax of the PROcess MEta LAnguage (PROMELA) is described and a simple example system is specified. Various properties of the system are mechanically verified using the SPIN model checker. PROMELA and SPIN are revisited later in Chapter 6 where they are used to verify properties of interface security policies written in SWIL.

2.1 Mobile Agents

There is no general consensus amongst researchers or practitioners on what exactly constitutes a mobile agent or how an agent is different from a normal software program. Rather than providing a concrete definition, Franklin et al [71] propose a number of properties which they believe an agent should have. These properties are displayed in the table in Figure 2.1. Some of these properties (specifically character, learning, goal-orientation) are obviously inspired by Artificial Intelligence (AI) research while others (mobility and communicativity) are lower-level implementational properties.

This thesis adopts the simple definition:

> A Mobile Agent is an autonomous program capable of moving between hosts on a network and communicating with other agents.

The term “autonomous” is used to imply that mobile agent programs decide themselves when to move unlike *process migration* systems (e.g. SPRITE [54]) in which the hosts decide when processes should move. The diagram in Figure 2.2

shows software running on two hosts labelled “system A” and “system B”, each including a mobile agent server. The software on each host is represented as a sequence of layers ranging from lowest-level and least abstract at the bottom to highest-level and most abstract at the top. Each layer communicates conceptually with the corresponding layer in the peered system although in practice messages are routed through lower layers. In the system depicted the underlying network connecting the two systems is ethernet. In the lowest level (corresponding with the ISO OSI physical layer) electrical signals are sent via coax ethernet cables. The ethernet protocol adds framing and checksumming information and corresponds to the OSI datalink layer. Both systems are using the internet protocol suite TCP/IP which corresponds with the OSI session, transport and network layers combined together[1].

Each mobile agent server provides an execution environment (sometimes described as a *place*) for the agents running inside. Servers provide all the relevant APIs needed to allow agents to operate and provide any necessary access to local resources. Agents expose *interfaces* through which they may communicate with other agents. Two types of message are sent between agent servers: messages sent from one agent to another and messages co-ordinating the transfer of agents. The message formats are handled by the mobile agent servers which correspond to the OSI presentation layer. The agents themselves correspond to the OSI application layer.

The security implications of representing software systems as a set of layers will be discussed later in Section 2.5.4. In this thesis we shall assume that all application-level communication is achieved through the web protocols, described later in Section 2.4.

In addition to communication primitives, mobile agent systems typically support the following basic but not standardised operations:

create: creates a new agent from scratch

clone: duplicate the current agent (like `fork()` under Unix)

migrate: move the agent to a new host

[1]TCP provides an error-free flow-controlled byte stream abstraction on top of IP which provides an unreliable datagram service.

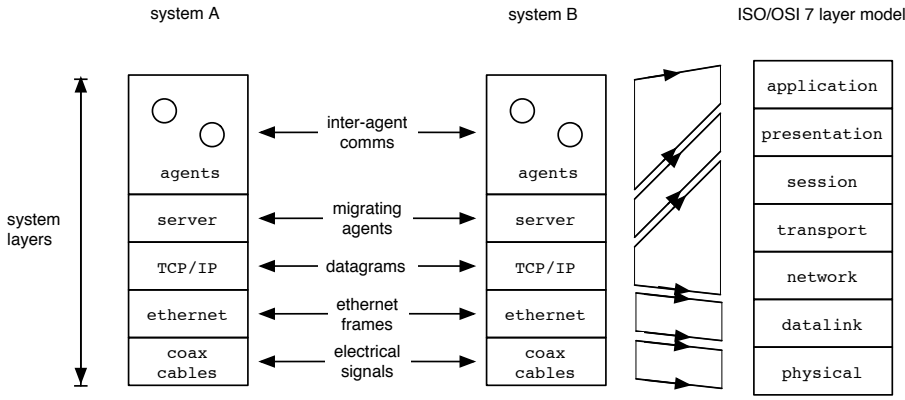


Figure 2.2: Diagram showing hosts, agent servers, agents and communication channels. Conceptually agents communicate directly with other agents but in practice the communications are routed via lower layers.

recall: forcibly returns an agent to its initial location

terminate: kills a running agent

Mobile agent implementations vary in many ways. Firstly, there is no standard mobile agent security mechanism; each mobile agent system provides its own. A second point of variation is the choice of implementation language. For interoperability, interpreted bytecode-based systems are common e.g. Java Aglets [106] and interpreted C autonomous objects [25]. Other systems define their own languages e.g. Telescript [176] and Obliq [31].

Wojciechowski's Nomadic Pict [177] is noteworthy in that it is a two-level agent programming language; it has a low-level sublanguage which deals with agent migration and location-dependent communication while a high-level language supports only location-independent primitives. The high-level language is translated into (or implemented in terms of) the low-level language by a mapping known as an *infrastructure*. Applications can be recompiled with different infrastructures to explore different migration and communication strategies—an infrastructure may be thought of as a kind of abstract mobility and communication policy. Later work by Unyapoth [167] considered semantics in detail and demonstrated how it was possible to prove the correctness of an infrastructure mapping for an arbitrary high-level source program.

Linked to the choice of implementation language, a third variation is the type of mobility offered [87]. Some systems are said to offer *full mobility*: the migration of all state including that within the kernel (i.e. open files and network connections). Another variation is *strong mobility* in which a system is able to introspect and migrate running code. The remaining systems offer *weak mobility*, insisting that agents perform manual shutdown and restart. For example, Java based systems only offer weak mobility because Java does not allow the introspection of the method call stack of running code. Even these distinctions are not very discriminating: Bettini et al describe how strong mobility may be implemented on top of weak mobility with a syntax-level translation [23].

Mobile agent code is handled in several different ways. Some systems transmit code along with an agent's state during migration. Alternative mechanisms include downloading the code on-demand, perhaps on a class-by-class basis – a common technique if the agent system is based in Java. A further alternative is to download the code from a third party server.

To give a flavour of how mobile agents may be used, the following four sections describe simple example applications.

2.1.1 Example: Flight Booking

In this example a mobile agent is created by a user who wishes to travel somewhere by air. The agent is written containing the user's preferences (e.g. date of journey, class of ticket etc.) and budget (i.e. how much they are willing to pay). The agent migrates to servers controlled by airlines from where it can efficiently interrogate local flight databases. The agent can migrate between airline servers to compare prices and find the most appropriate deal for their user. The process is illustrated graphically in Figure 2.3. The user first creates an agent (represented by a circle) which migrates between agent servers (rounded boxes) as indicated by the filled arrows. In the illustration the agent migrates between three different airline agent servers in turn, querying local databases (represented by the unfilled arrows) before returning to the original agent server.

Provided the agent itself is small compared with the database data, this design requires less overall network bandwidth than the traditional client/server design in which the databases are all interrogated remotely from a fixed location. The main

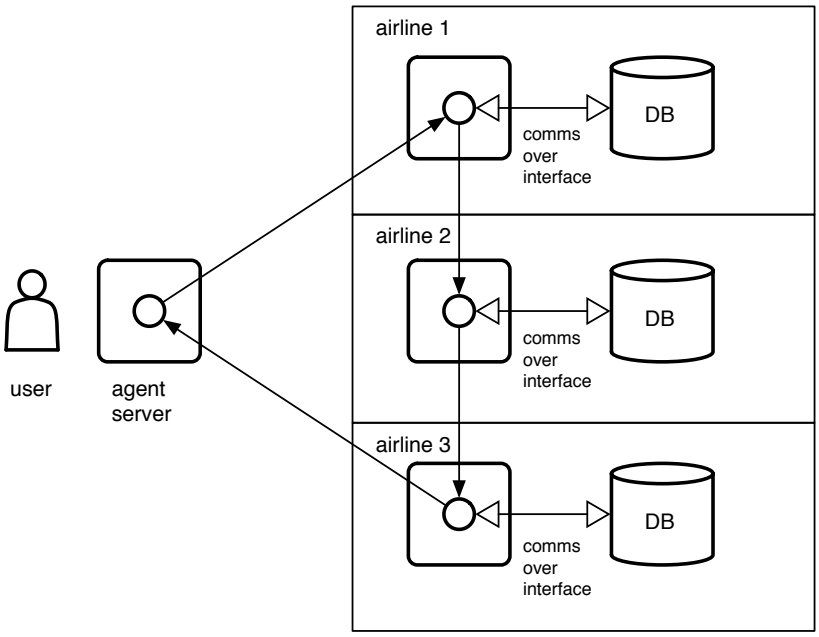


Figure 2.3: Use of a mobile agent to query remote databases.

disadvantage is that a malicious airline site may disassemble the agent to discover information the user would rather keep secret, for example how much money the user is willing to pay.[2] A malicious airline server has other potential avenues of attack open to it. It might block the agent's exit, particularly if it intends to migrate to a competitor's servers. Alternatively it might corrupt the agent, modifying any data stored supplied by competitors. To address these problems the user must always assume the worst[3]and send fresh copies of agents to each airline server and then only analyse the results on the user's own trusted computer.

2.1.2 Example: Secure Database Searching

Yannopoulis et al [180] proposed a system called PIVOTS: Private Information Viewing Offering Total Safety. The system uses mobile agents to enable sophisticated searches to be performed against private data collections while limiting the amount of possible data leakage.

[2]Advocates of price discrimination might claim such transparency is a good thing [124].

[3]Later on in Section 2.5 building secure systems is described as "programming Satan's computer".

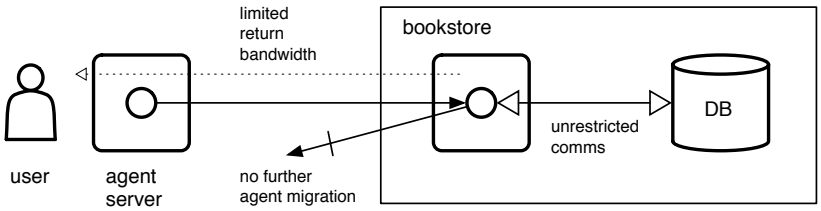


Figure 2.4: Use of a mobile agent to securely perform full-text searching on private data.

Their running example is a bookstore which contains, but does not want to make freely available, the full-text contents of books. In a situation where the full-text *can* be freely published a user or a third-party search engine can easily perform arbitrary queries against the text. However, this is not possible in the case where the data is private. In the PIVOTS system, users can write agents (or have them written by a third-party) which analyse certain properties of the text. The agents migrate to the bookstore servers from where they can run sophisticated searching algorithms against the full-text contents. Crucially, agent communications are limited and an agent can only send back simple responses such as, "you would like this book". Therefore they argue that the risk of data leakage is approximately the same as that experienced by a traditional book shop, in which people can wander in and physically thumb through the contents of books on the shelf.

The process is illustrated graphically in Figure 2.4. The user creates an agent which migrates to an agent server running in the bookstore. The agent then enjoys unrestricted communication with the back-end database which contains the full-text contents of the books. The agent cannot leave the bookstore server (for it might take private data back with it) but it may send back a simple message, represented by the light, dashed, unfilled arrow.

2.1.3 Example: Controlling ATM Networks

Halls and Rooney [81] describe how mobile agents within a system called the Tube [80] may be used to control and monitor a network of ATM switches. The agent code and execution state is transmitted between server processes in the form of continuations in the language Scheme. Agents can query the state contained within network switches, specifically examining the tables mapping input con-

nection numbers (known as Virtual Channel Identifiers in ATM jargon) and ports to output connection numbers and ports. The agents can detect inconsistencies and correct the tables. The system proposed has two main benefits compared with a traditional centralised approach: (i) the distribution ensures the system has no single point of failure; and (ii) the overall amount of network traffic required is reduced.

2.1.4 Example: Location-Oriented Multimedia

In his thesis, Halls [80] describes how mobile code may be used in a "location-oriented multimedia" system in which video and audio components follow the user as he/she moves around the environment. Applications in the proposed system are able to monitor the physical locations of users wearing Active Badges [172] and migrate multimedia streams and user interfaces accordingly. Possible applications include: a music playing application which always plays music physically near the user; and a video conferencing system which is able to track a user and select the most appropriate video camera feed as they walk around. The example of a mobile music playing application will be revisited later, in Section 4.6.

2.2 Physical Location-Sensing

In addition to possessing logical mobility as exemplified by Mobile Agents, future systems are likely to also be physically mobile. This section describes how and why such future devices will be able to sense their location as they move around the environment,

There exist many devices which can be used to measure physical location information with a variety of different accuracies, latency, levels of reliability and cost. A good taxonomy of sensor systems has been written by Hightower and Boriello [84]. Sensors differ in the exact physical characteristic they measure: some compute distance between objects (e.g. a measuring tape), some give a notion of proximity (e.g. an RFID tag), some measure 3D co-ordinates (e.g. a Polyhemus tracker[4]) while others determine containment of objects (e.g. Active Badges [172]). Some location-sensing techniques arise as a side-effect of

[4]Polhemus tracker product information available from `http://www.polhemus.com/`

other technologies (typically wireless communication systems), for example IEEE 802.11 [93] as used in the RADAR [16] project and GSM Location Based Services (LBS) [79]. Systems designed primarily for location-sensing include the Global Positioning System (GPS) [86], Active Bats [173], Ultra Wide Band systems like PAL650 [65] and Cricket [130].

Examples of location sensing systems are given in the following sections.

2.2.1 Example: GPS

The US Military's Global Positioning System (GPS) uses a set of low earth orbit satellites to transmit signals to passive ground receivers. Each receiver must have a clear line of sight to several satellites; therefore the system only works well outdoors, away from trees and tall buildings. The more satellites in view the more accurate the data can be. A minimum of 3 satellites is required for a 2-D position and 4 satellites for a 3-D position.

The GPS satellites transmit their positions (calculated from sophisticated orbit models computed by ground stations) simultaneously, using on-board atomic clocks for accurate timing. The signals are transmitted using a chipping code over a large frequency range making the system very robust to jamming. Receivers correlate the incoming signals with the known chipping sequence to recover the data and calculate the relative times of arrival of each signal. The positions of the satellites and the relative times of arrival are then used to calculate the position of the receiver.

Standard inexpensive hand-held GPS units are accurate to 10-20m 95% of the time but *differential GPS* which calibrates against a receiver at a well-known fixed location can be accurate to 1m.

2.2.2 Example: Active Bat

The Active Bat system comprises a network of ceiling-mounted ultrasound receivers and small battery-powered tags called "Bats". Bats emit ultrasound signals over a small frequency range when triggered over a radio channel. The ceiling-mounted receivers use time of flight measurements to estimate the distance between receivers and individual Bats. Since the receivers are in well-known positions, these distances can be used to multilaterate the absolute positions of the bats. The bats can be located with an accuracy of 3cm 95% of the time at a maxi-

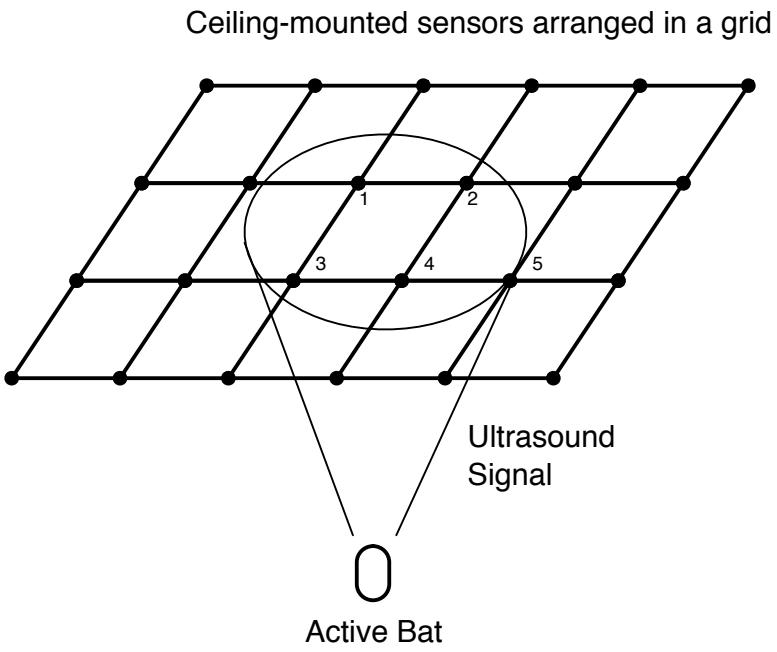


Figure 2.5: Diagram showing how an ultrasound signal emitted by an Active Bat is received by multiple ceiling-mounted receivers.

mum frequency of 50Hz.

The diagram in Figure 2.5 shows an Active Bat emitting a narrowband ultrasound pulse directed at ceiling-mounted receivers. In the scenario shown, the pulse is received by sensors labelled 1,2,3,4 and 5. Each room has a temperature sensor allowing the system to accurately calculate the speed of sound in air. Based on this calculation, the system can then calculate the distances between the Bat and all five receivers and then use multilateration to compute the absolute Bat position.

2.2.3 Example: Radio Received Signal Strength (RSS)

Radio communication systems such as IEEE 802.11 [93] (also known as WiFi) and Bluetooth [2] provide a means to monitor the strength of signals from other network nodes. The signal strengths received from several fixed nodes can be fed into a radio propagation model to estimate the position of the mobile receiver. For example, power received from a transmitter in free space using omnidirectional

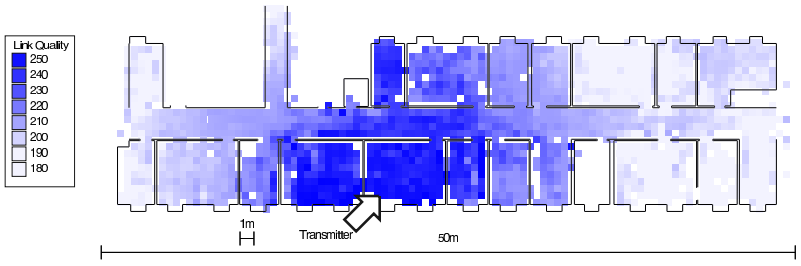


Figure 2.6: Plot showing received signal strength (link quality) of a mobile Bluetooth receiver relative to a fixed basestation. Picture courtesy of Alastair Tse and Kieran Mansley.

antennas is well known to be inversely proportional to the square of the distance from the receiver.

Indoors the situation is more complicated. Walls, floors, doors, windows and furniture all attenuate signal and cause reflections; the severity of these effects depend on the frequency of the radio and the exact materials the objects are constructed from. Two objects may look identical at visible wavelengths but may have vastly different effects on radio wavelengths. The diagram in Figure 2.6 shows received signal strength from a class 1 Bluetooth device as a function of position (measured with an Active Bat) in the Laboratory for Communication Engineering in the University of Cambridge. The signal clearly drops over a range of about 10-15m but even samples quite close to the receiver vary unpredictably, probably due to the placement of furniture and the movements of people[5].

WiFi signal strength can be used to infer location outdoors as well as indoors. The PlaceLab project[6] uses a large user-created database mapping WiFi basestation identifiers to latitude and longitudes. If a mobile receiver spots a known basestation (typically in a densely populated urban environment) then it can infer its position is within 100m of base station.

[5]Note that organic tissue contains a large percentage of liquid water which readily absorbs 2.4GHz microwave energy, an effect exploited by microwave ovens. This effect also means that humans attenuate WiFi and Bluetooth signals.

[6]More information can be found at `http://www.placelab.org/`

2.2.4 In Future Many Devices Will Possess Location Information

In the US the Federal Communications Commission's Enhanced 911 program [60] mandated that mobile phones be locatable *by the network* to within 100m 67% of the time and locatable *by the device* to within 300m 95% of the time. The network can locate devices by multilaterating phone positions given the time of arrival of mobile signals while the devices can locate themselves by incorporating GPS receivers. It has been suggested that adding GPS to a mobile handset will cost no more than 10 dollars [168]. Economies of scale and similar pending initiatives by other governments (such as in the EU) strongly suggest that future mobile phones (and perhaps all future mobile communications devices) will have some built in location-sensing capability.

Without adding an explicit location system like GPS, any embedded device with a wireless network interface will automatically be able to use signal strength measurements to infer at least proximity to other wireless devices. If one of these nearby devices is a GPS-equipped mobile phone then the embedded device will be able to infer its absolute position.

For these reasons, this thesis makes the assumption likely to hold in the near future that location-sensing systems (whether explicit like GPS or implicit like Bluetooth RSS) are found in most of the networked, embedded devices throughout the environment. This thesis makes the further two related assumptions that (i) the resolution of the location sensors is sufficient to distinguish between adjacent rooms in a building; and (ii) that user privacy concerns have been properly addressed elsewhere (e.g. using Beresford and Stajano's concept of a *mix-zone* [21] in which a trusted system deliberately mixes location sightings within defined physical areas, obscuring the identities of the people involved).

2.3 Interfaces

An Interface is used to describe how one component (e.g. an agent) may communicate with another. Earlier in Section 2.1 Mobile Agents were described as programs which could both move and communicate, although the mechanism and style of communication was not discussed. In this section a simple formalism for Interfaces is presented in the style of Remote Procedure Calls (RPCs) [121] which is suitable to describe such agent communication.

A useful interface should specify what assumptions the component makes about its environment (referred to as *input assumptions*) as well as what guarantees it is prepared to make about its output (referred to as *output guarantees*). These assumptions normally take the form of type specifications for each interface operation. Informally, when the outputs of one interface are connected to the inputs of another we require the output guarantee of the first to satisfy the input assumptions of the second. In the case where the input assumptions and output guarantees only specify types, this corresponds to simple interface typechecking. However input assumptions and output guarantees can be arbitrarily complicated e.g. they could describe the acceptable order in which functions are called using automata [46]. They also correspond to the use of preconditions (`REQUIRES`) and postconditions (`ENSURES`) in the SRC Extended Static Checker (ESC) [51].

2.3.1 Formalism for Interfaces

This section describes the simple interface formalism including both a simple syntax and a set of common interface operations.

Interface Syntax

An interface I is defined as a set of types ($t_1 \ldots t_k$) and functions ($f_1 \ldots f_n$) whose parameters have direction (either `in` or `out`), types and names. We allow the pre-defined types int and $string$, custom abstract types and the type constructors $list$ and $\times$ for constructing list and cartesian product types respectively. Valid interfaces satisfy the constraint that all types visible in function signatures are either built-in or defined in the interface itself. Each function has a conjunction of atomic formulae representing explicit assumptions and guarantees. Figure 2.7 shows an example interface specification. Note that an interface specification does not say anything about the *implementation* of that interface; the implementation is considered as a black-box.

The input assumptions of the operation f_1 are twofold. Firstly, the input has type t_a and the predicate $P(a)$ holds. Similarly the output guarantees are that the output will have type t_b and the predicate $Q(b)$ holds. Additionally we impose the constraint that all the types mentioned in the functions $f_1 \ldots f_n$ must be declared in the interface.

```
interface I {
    type t_1 = ...
    ...
    type t_k = ...
    f_1(in t_a a, out t_b b) assume { P(a) }
        guarantee { Q(b) }
    ...
    f_n(in t_c c, out t_d d) assume {...}
        guarantee {...}
}
```

Figure 2.7: Example Interface consisting of types $t_1 \ldots t_k$ and functions $f_1 \ldots f_n$.

Operation: Hiding

Individual interface types and operations may be hidden so that they are no longer accessible by callers. For example consider the following interface I':

```
interface I' inherits I {
    hide f_1
}
```

Interface I' inherits all the types and functions from interface I with the exception of the operation f_1 which becomes hidden. Note that this does not mean the implementation of f_1 cannot be called; only that interface I' no longer mentions it. If a client can access interface I directly then f_1 will be available. Types can also be hidden provided the constraint – that all types mentioned in operation signatures are visible – is not violated. Therefore in order to hide a type used by a function it is first necessary to hide that function.

Operation: Extending

In the following example, interface I' inherits all the types and functions of interface I and adds a type t and a function g. It is assumed that t and g do not already exist in I.

```
interface I' inherits I {
    type t = ...
    g(in t arg) assume {...} guarantee {...}
}
```

Operation: Replacing

Operations and types can be replaced using the same syntax used to add operations and types in Section 2.3.1. For example:

```
interface I' inherits I {
    type t = ...
    f1(in t arg) assume {...} guarantee {...}
}
```

In this example I' inherits from interface I adding a new type t and redefining operation f_1, changing its type, assumptions and guarantees. Note that this operation is only valid if no other function is visible which still references the old type t – when a type is replaced all functions which reference it must also be replaced.

2.3.2 Interface Proxies and Firewalls

The term *proxy* is used here to be an entity which intercepts requests for another entity and if necessary[7] forwards the request back to the original entity. Crucially, a proxy exposes exactly the same interface as the original entity. The term *firewall*, on the other hand, refers to a more generic processing entity which is capable of changing the interface of an application.

The diagram in Figure 2.8 shows four cases which are referenced later in the thesis. The first shows a client interacting with a server exposing interface I. The second case shows an intermediate proxy between the client and the server. All requests and responses are processed by this entity but crucially from the point of view of the client, the interface is not changed: the client still interacts with the server through interface I. In the third case the client interacts through a firewall, which changes the interface exposed by the application from I to I'; a configuration which might represent an upgraded client interacting with a legacy server. In the fourth case the client still uses interface I but the server now exposes interface I'; a configuration which might represent an upgraded server interacting with a legacy client.

[7] A *caching proxy* may be able to respond to the request itself with cached data and avoid contacting the original server.

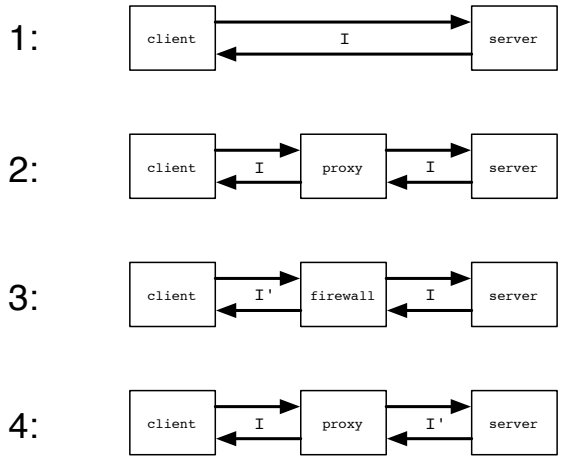


Figure 2.8: Diagram showing four configurations. The first involves a client invoking an interface I on a server. The second case interposes an interface proxy between the client and server, leaving the interface unchanged. The third case uses a firewall rather than a proxy causing the client to see an interface I' rather than I. The fourth case involves an interface firewall used to convert the interface I' exposed by an *upgraded* server into the original interface I expected by an unmodified client.

2.4 The World Wide Web

This thesis describes application-level policy languages which address both the mobility and interface behaviour of ubiquitous computing systems. The description of mobile agents in Section 2.1 and location sensing systems in Section 2.2 provide all the necessary background for understanding the Unified Spatial Model in Chapter 3 and the Mobility Restriction Policy Language in Chapter 4. The interface behaviour policy systems SPDL-2 and SWIL described in Chapters 5 and 6 are based upon the web protocol suite, described in this section. The use of a concrete set of protocols simplifies the presentation while the specific use of the web protocols permits the use of more compelling commercial examples. However the techniques presented in this thesis are not limited to apply to only the web-protocols; rather they apply to any similar distributed system.

This section describes the web protocols in fine detail and describes an *approximation* of the interface exposed by a web-based application in terms of the formalism described earlier in Section 2.3. The approximation is only needed to

simplify the work presented in later chapters and does not reduce the power of the policy systems described. Examples in later chapters will assume knowledge of protocols and the interface details described here.

At a minimum all distributed systems must provide three things: (i) a protocol for transmitting data between entities; (ii) a method for naming entities and resources; and (iii) a standard data format. The WWW uses the HyperText Transfer Protocol (HTTP) to transfer typed data objects; Uniform Resource Identifiers (URIs or URLs or URNs) to name objects; and the HyperText Markup Language (HTML) to describe the the structure and content of hypertext documents. Note that HTTP and URIs may be used to transfer data other than hypertext (such as sound or video files) but the following presentation will focus exclusively on HTML — this is because HTML documents contain URIs pointing to further data objects while other files (such as sound, graphics or videos) are simply 'leaf' nodes, containing no further links. Each of these components is described in the following sections.

2.4.1 HTTP

According to RFC 2616 [64], HTTP is an "application-level protocol for a distributed, collaborative, hypermedia information system". HTTP is designed to run over a reliable transport protocol, typically the internet Transmission Control Protocol (TCP/IP). Messages are designed to be self-describing; properties of the payload data are listed in message *headers* including cachability hints and content type indication. HTTP is a very flexible protocol which can be used to interface with legacy systems such as File Transfer Protocol (FTP) servers and Network News Transport Protocol (NNTP) servers, providing users with a common and consistent interface across disparate systems.

As described earlier in Section 1.6 many current firewalls are configured to let HTTP through and to block other traffic – a situation that has encouraged application writers deliberately to encapsulate their protocols within HTTP. IP itself can even be tunnelled over HTTP creating the possibility of making Virtual Private Networks (VPNs) spanning multiple organisations ignoring firewalls (and associated firewall policy).

Protocol Roles

HTTP defines a number of roles, including client, server and proxy. An HTTP client is any entity which can create a request and send it somewhere. Correspondingly, an HTTP server is any entity which can receive a request and reply with a response. Clients and servers need not be distinct entities. For example, HTTP proxies are both clients and servers; they receive requests from clients by acting as a server and then forward the requests by acting as a client. Like the interface proxies described in Section 2.3.2, HTTP proxies may not change the interface exposed by an application, performing only simple caching functions. Entities which can perform interface transformations are also known as firewalls.

2.4.2 Naming Resources

Uniform Resource Identifiers (URIs) are the primary means by which objects on the web are named. The general structure of a URI is as follows:

```
scheme : scheme-specific-part
```

The first part, `scheme`, names a family of URIs. Example schemes include `http`, `ftp` and `mailto`. The `scheme-specific-part` is defined by the specification of the scheme. A common set of URI schemes share a hierarchical syntax for the scheme-specific part:

```
// authority path ? query
```

The `authority` usually refers to a host by DNS name, the `path` is an optional '/'-separated sequence of names analogous to a filesystem path and the `query` is some optional, arbitrary data relative to the named path. Examples of URIs include

```
mailto:dave@recoil.org
ftp://www.cl.cam.ac.uk/pub/
http://www-lce.eng.cam.ac.uk/
```

The optional `query` part of an HTTP URI contains '&'-separated `key=value` pairs known as *query parameters*. The URI `http://www.example.com/foo?a=b&c=d` has two such parameters: `a=b` and `c=d`.

Query parameters are used to pass information to *Web Applications*, described later in Section 2.4.4. For the rest of this thesis the term URI shall refer to the scheme, authority and path of HTTP URIs (e.g. the part `http://www.example.com/foo`) without the parameters, which will be considered separately.

Many RFCs and technical documents refer to URLs and URNs as well as URIs. Uniform Resource Locators (URLs) are technically a subset of URIs, specifically those which identify an object by the combination of its primary access mechanism and address (e.g. `ftp://www.cl.cam.ac.uk/pub`). Uniform Resource Names (URNs) are another subset of URIs, specifically those which identify a document by a *pure name* i.e. a name which has no visible location information[8] (e.g. a telephone number without the area code is a pure name[9]).

Although classifying URIs into URLs and URNs seems appealing, many URIs found on the Internet are neither strictly one nor the other. For example the URI corresponding to an entry in a telephone directory might be `http://directory.org/phone/07771234567`, an identifier which contains some location information (i.e. access host `directory.org` via HTTP) as well as a part which is more like a URN (i.e. `07771234567`). This confusion has persisted, with the terms URI, URL and URN often used interchangeably. The general term URI shall be used exclusively in this thesis to refer to all names: pure or impure.

Protocol Messages

HTTP messages come in two flavours: requests and responses, depicted graphically in Figure 2.9. Both request and response messages consist of a series of CR-LF terminated string lines (the header) followed by a blank line and then an arbitrary payload. The first header line is mandatory and indicates the message type (e.g. whether it is a request or response) while the remaining header lines contain a series of key-value pairs. According to the HTTP RFC [22], some head-

[8]To quote Needham and Mullender [120]: "a pure name just identifies. An impure name guides."

[9]Note that earlier implementations of the telephone system may have encoded some location information in telephone numbers to aid call routing; now that it is possible for numbers to be ported between network operators the names are pure.

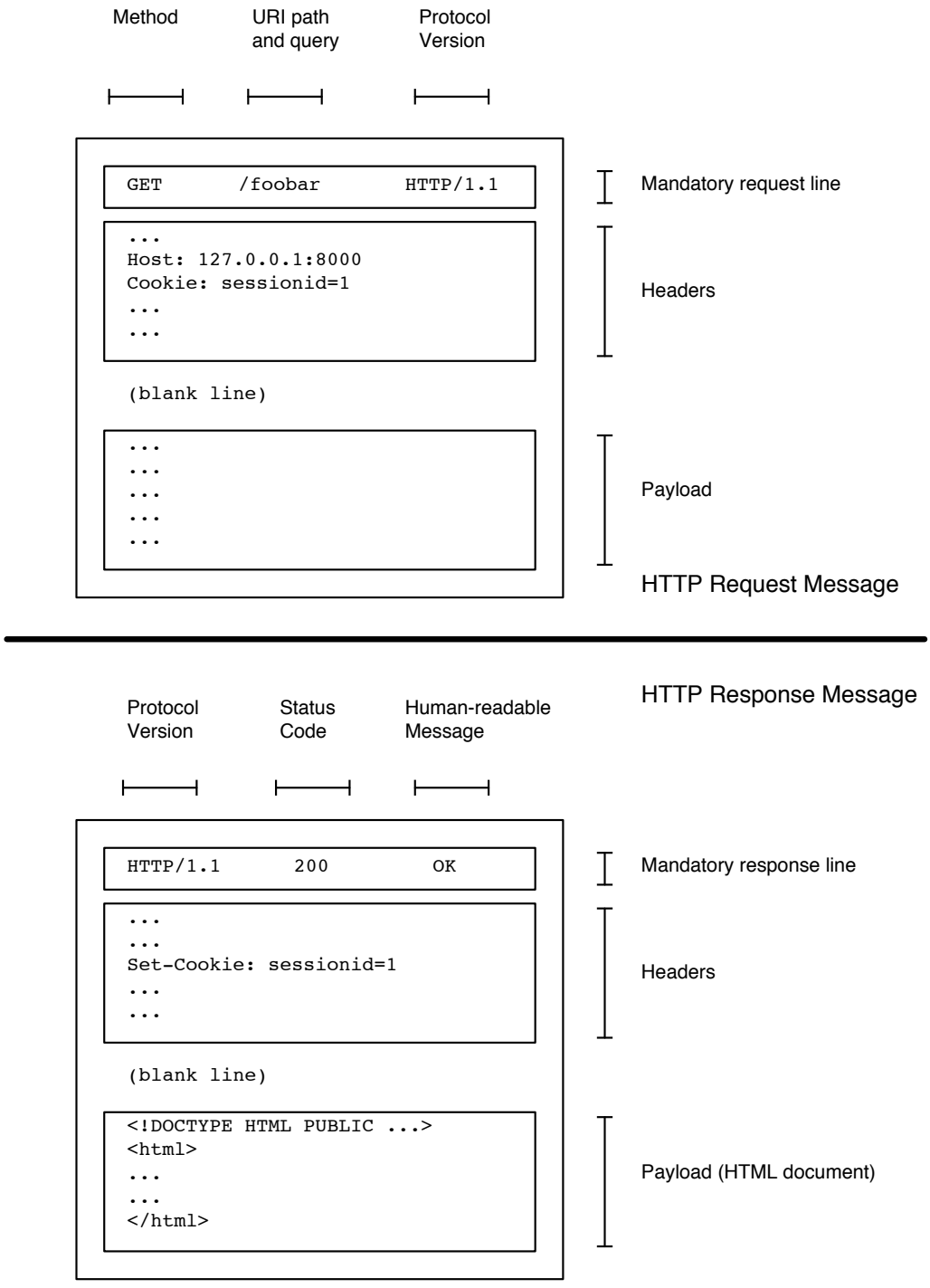


Figure 2.9: HTTP message formats. Both request and responses consist of a number of CR-LF terminated string lines followed by blank line then a payload.

ers are mandatory, some are encouraged while others are entirely optional. An implementation will typically ignore unknown headers. The exact encoding used by the payload is specified in the header (using the `Content-Encoding` key) and may be anything from straight ASCII text to the output of a compression program like `gzip`.

The mandatory first header line of both requests and responses have one feature in common: they both contain the generating protocol version, albeit at different places in the first line. This protocol version is intended to allow interoperability with future protocol versions. The general features of requests and responses shall now be described in more detail.

The mandatory first line of a request consists of a *method* (roughly speaking a type of action to perform) and the path and query from a URI (URIs were described previously in Section 2.4.2). The rest of the URI – the authority – is not in the first line but in the `Host:` header. Applications are free to either invent their own custom method types or to use one of a pre-defined set which includes the following:

GET : indicates the request is a simple data retrieval operation which should not have side-effects on the server.

POST : invokes an action on the server (identified by the URI) with arguments supplied in the payload.

By convention when a user presses the submit button on a form rendered by a web-browser, the completed form values are uploaded to the server by the browser as either query parameters of a GET message or as part of the payload of a POST message. For simplicity this thesis considers the former – form parameters are uploaded as the URI query parameters – and hence the request payload can be ignored.

The mandatory first line of a response message consists of the protocol version together with a status code and a human-readable message. The human-readable message is intended to help the user understand the message and is conventionally ignored by non human-controlled web clients. There are predefined status codes for a number of common scenarios including: (i) request was completed successfully; (ii) the URI was not found on the server; and (iii) the object identified by

```
GET /first HTTP/1.1
Host:  127.0.0.1:8000
Connection:  keep-alive
User-Agent:  Mozilla/5.0 (Macintosh; U; PPC Mac OS X;
en-us)
Accept:  */*
Accept-Language:  en-us, ja;q=0.62, de-de;q=0.93, de;q=0.90
```

Figure 2.10: The headers of the first HTTP request message.

the URI has moved to another place. As explained earlier, we assume here that all response payloads contain only HTML data.

One type of header is especially important and appears in both requests and responses. Originally invented by Netscape Communications Corporation, a *cookie* is a key-value pair sent to a client by a server in a `Set-Cookie:` response header. By convention a cookie-compatible client will return this data to the server in future request headers. Cookies may be used for any purpose but common uses include storing user preference information and to associate together requests from the same client, providing the illusion of session state over the HTTP protocol.

Example Request and Response Messages

This section describes examples of HTTP messages in the context of a simple series of interactions between a client and a server. First, imagine the client invokes a **GET** request targeting URI `http://127.0.0.1:8000/first` on the server. The headers of the HTTP request message generated are displayed in Figure 2.10. The mandatory first header line indicates that the message is a **GET** request using protocol version 1.1. By concatenating the `Host:` header and rest of the mandatory first header line we can see the message is targeted at the URI `http://127.0.0.1:8000/first`. The `Connection: keep-alive` header is a hint sent from the client that the underlying TCP connection stay open after the current transaction completes for efficiency reasons[10]. The `User-Agent:` line indicates the request was generated by an Apple Mac-

[10]The TCP protocol uses a *slow start* algorithm and it is therefore often faster to reuse the same connection once it has increased its send window size rather than constantly opening and closing new connections.

```
HTTP/1.1 200 OK
Date:  Tue, 28 Oct 2003 11:57:57 GMT
Server:  Apache/2.0.40 (Red Hat Linux)
Set-Cookie:  sessionid=1
X-Powered-By:  PHP/4.2.2
Transfer-Encoding:  chunked
Content-Type:  text/html; charset=ISO-8859-1

<html>
<body>
<a href="http://127.0.0.1:8000/second">
click here
</a>
</body>
</html>
```

Figure 2.11: The headers of an HTTP response message.

intosh computer running the browser called “Safari” while the `Accept:` and `Accept-Language:` headers describe which content types and languages are preferred in the response.

Continuing the example, Figure 2.11 shows the the response sent back by the server. The mandatory first line indicates the server is replying using protocol version 1.1 and the status code is 200 with human-readable message “`OK`”. The rest of the headers indicate the server type, type of content and content encoding type. The `Set-Cookie:` header contains the key-value pair `sessionid=1` which the server expects the client to return in subsequent requests. The payload of the response contains an HTML document (HTML is described in more detail below in Section 2.4.3) which contains a single link with URI `http://127.0.0.1:8000/second`.

Assuming the payload HTML is rendered by a web-browser and the user clicks the link, the web-browser will produce a second HTTP request like that shown in Figure 2.12. The second request differs from the first request in two respects: (i) the URI query has changed from `/first` to `/second`; and (ii) the second request has the `Cookie:` header which returns the data `sessionid=1` to the server. The server will be able to use this cookie to customise the next response

```
GET /second HTTP/1.1
Host:  127.0.0.1:8000
Connection:  keep-alive
Cookie:  sessionid=1
User-Agent:  Mozilla/5.0 (Macintosh; U; PPC Mac OS X;
en-us)
Accept:  */*
Accept-Language:  en-us, ja;q=0.62, de-de;q=0.93, de;q=0.90
```

Figure 2.12: The headers of the second example HTTP request message.

to the user.

2.4.3 HyperText Markup Language

The HyperText Markup Language (HTML) is a language for creating structured hypertext documents defined by the W3C [179]. HTML documents are primarily intended to be viewed by humans using graphical web-browser programs. HTML documents contain content, formatting information[11] and references to other documents. The language HTML is specified using the Standard Generalised Markup Language[12].

An example HTML document may be seen in Figure 2.13. The document is a tree of *elements* of the form `<name> ... </name>`. The text `<name>` is known as an *opening tag* and `</name>` is known as a *closing tag*. Elements with no contents can be written in the short form `<name/>`. Some elements have *attributes* which are key-value pairs written as `<name key1=val1 key2=val2>` where the element `name` has two such pairs: (`key1`,`val1`) and (`key2`, `val2`). The example also contains a *hyperlink* – a reference to another document – written using the syntax `<a href="URI">label</a>` where `label` is human-readable text to be displayed by the web-browser program and `URI` is the URI to be followed when the user selects the hyperlink.

Users with graphical web-browsers can interact with servers through HTML forms as well as hyperlinks. HTML forms allow the user to fill data into text fields, select items from drop down lists and toggle checkboxes. Special submit

[11] Presentation information can also be abstracted using *style sheets*, not discussed further here.

[12] More information relating HTML and SGML can be found at `http://www.w3.org/TR/REC-html40/intro/sgmltut.html`

```
<html>
<head>
<title> This is an example page </title>
</head>
<body>
<h1> This is a title </h1>
This is some ordinary text.
<a href="http://www.example.com/">hyperlink</a>
<form target="http://www.example.com/"
onSubmit="...">
<input type="text" name="foo" value="bar"/>
<input type="submit"/>
</form>
</body>
</html>
```

Figure 2.13: An example HTML document

buttons allow the form data to be uploaded to the server using a POST request. In the example of Figure 2.13 there is a single form element. The element has two attributes: `target` and `onSubmit`. The URI to which the data will be sent is given by the `target` attribute and the `onSubmit` attribute allows the HTML author to include some client-side Javascript code which is executed once the user presses the submit button but before the data is uploaded to the server. Javascript code is commonly used to prevalidate the contents of a form before incurring the latency of a full network round-trip to the server. If a form fails the Javascript validation check then the error will be reported to the user through a pop-up dialog and the HTTP request will be aborted.

Relation to XML

At first glance XML and HTML look very similar. In fact they are different both in terms of syntax and in terms of purpose. XML has much simpler syntax than HTML and is therefore much easier to parse. Every XML document must have properly nested elements and no tags may ever be left out. In contrast, HTML is more flexible and in some circumstances tags may be left out, their presence being inferred automatically by the parser. Due to the legacy of the WWW and the many web-pages containing invalid HTML, many web-browsers accept invalid HTML

without complaints, parsing them using heuristics. XML parsers, by contrast, are much stricter.

HTML and XML are meant for different purposes. HTML is designed specifically for encoding hypertext documents whereas XML is a generic mechanism for encoding data for any purpose. Each XML document is associated with a Document Type Definition (DTD) which describes the set of acceptable element names, how the elements may nest and the attributes each element should have. An XML parser which checks the XML document is valid with respect to the DTD is known as a *validating parser*.

2.4.4 Web-Applications

This thesis uses the following simple definition:

> A Web-Application is any application which uses URIs to name remote procedures and HTTP to transmit requests and responses.

Both web-application clients and servers can take many forms. A client can be anything from a graphical user program to a command-line URI fetching tool. A server can be anything from a web-server running a set of scripts to a graphical program presenting a remote control interface over HTTP. Some servers associate URIs with static HTML pages while others dynamically generate responses at runtime. Some types of client will parse HTML received in responses from the server. A *web-spider* is a type of client which will attempt to traverse all links from a starting page either in search of a particular object or to build up a database of reachable pages. A *web-browser* is another type of client which will parse and render the HTML responses on the user's screen. The web-browser invites the user to fill in forms and click on links to generate further HTTP requests.

2.4.5 Web-Applications: Handling Application State

HTTP is a stateless protocol. Many applications are inherently stateful and need to create the illusion of a coherent session between the application and the user. The only way to do this is to send session information to the client either as a cookies or as part of an HTML form. This client-side state is then returned to the server (as either a cookie or URI parameter) when the next URI is invoked. Such client-side state can take three different forms:

1. an opaque identifier referencing server-side state;
2. data cryptographically protected against modification; or
3. plaintext data.

Each of these options has advantages and disadvantages. The first — an opaque identifier — has a security advantage: the client has no way of directly accessing the data pointed to by the identifier since it exists only on the back-end server. However it also has two major disadvantages. Firstly the machine storing the server-side state becomes a bottleneck; every request involves consulting (and possibly updating) this shared state. This becomes especially critical if the site is busy and the state is replicated and must be kept in sync across several servers. Secondly the server must take responsibility for deleting the state at the right time – a tricky operation since a user might *bookmark* a session and attempt to return later, only to find the session state has been deleted.

The second option listed involves sending application state to the client cryptographically protected to guarantee its *integrity* (integrity is explained in detail later in Section 2.5.2). This option has the advantage that it involves storing no state on the server (which otherwise might limit how many clients the server can support), except any secret keys necessary to verify the integrity of the data. The main disadvantage of this method is the state might be large and therefore take significant time to protect and transmit.

Finally, the third option – keeping the data on the client side in plaintext – has the advantage of simplicity. Indeed many kinds of session state can be stored safely on the client in this way e.g. user preference data. However, this method is totally unsuitable for any data which, if modified, could disrupt the workings of the application. More will be said about this problem later in Chapter 5.

2.4.6 Interfaces of Web Applications

This section shows how an approximation of the interface of a Web Application may be described using the formalism from Section 2.3. Section 2.4.4 defined a web-application to be any application which uses URIs to name remote procedures and HTTP to transmit requests and responses. HTTP requests and responses were described in detail in Section 2.4.2.

First consider an HTTP request. The request consists of four parts: (i) a method (i.e. GET, PUT, POST or an application-defined custom method name); (ii) a URI; (iii) the rest of the headers; and (iv) the payload. Application data can therefore be sent to the server as (i) names of custom methods; (ii) URI query parameters; (iii) headers; or (iv) within the payloads. Individual web-applications can use any arbitrary combination of these. Rather than include all of these separately in our interface description we instead assume that, for simplicity, all application data is contained solely within the URI query parameters and one particular header – the `Cookie:` header (described earlier in Section 2.4.2 this header contains an arbitrary key value pair which the server wishes the client to add to its future requests). This assumption does not reduce the power of the policy systems described in later chapters; they could easily be expanded to consider all these extra possibilities, treating them exactly the same way they do query parameters and cookies. Cookies shall be represented as key-value pairs using the type

$$type\ \ cookies\ =\ (string \times string)\ list$$

We assume that user supplied data is encoded as part of the URI *query parameters*. In the URI `http://www.example.com/foo?a=b&c=d` the part after the `?` is the query string containing the two parameters `a=b` and `c=d`. Query parameters are key-value pairs which may be represented by the type:

$$type\ \ parameters\ =\ (string \times string)\ list$$

The remainder of the URI (the non-query part) shall be considered as a simple type:

$$type\ \ uri\ =\ string$$

Therefore an HTTP request has the type

$$type\ \ request\ =\ uri \times parameters \times cookies$$

An HTTP response message may be broken down into three parts: (i) a status code; (ii) the rest of the headers; and (iii) a payload. We shall ignore the status

code (implicitly assuming it to be 200 – success) and all the headers except the `Set-Cookie:` header. For the sake of simplicity we consider the payload to contain an HTML document which contains a single HTML form[13]. A form consists of a URI and a set of named controls, the contents of which are uploaded to the server when the form is submitted. The form may be represented by the type:

$$type\ \ form\ =\ uri \times parameters$$

Therefore the response may be represented by the type:

$$type\ \ response\ =\ uri \times parameters \times cookies$$

Note that, under our assumptions, a $response$ is of the same type as a $request$.

Figure 2.14 shows the interface I exposed by a web-application with URIs=$\{U_1 \ldots U_n\}$, each URI is represented by a function which expects a request and generates a response but otherwise makes no assumptions or guarantees. The name of each URI in interfaces shall be commonly abbreviated using the *path* of the URI. For example a URI `http://www.example.com/view` would be represented using a function with name `view`.

2.5 Security Principles

Chapter 1 provided statistics from CERT showing a large year on year increase in reported security incidents and vulnerabilities. Arguments were presented that traditional approaches to security – specifically network firewalls – are insufficient to deal with new threats arising due to the increased logical and physical mobility of communicating entities. To better understand these threats and the solutions proposed in later chapters this section provides some background on the field of computer security.

First of all we attempt to define the word "security" itself. One definition of security taken from the Oxford English Dictionary is as follows:

> the condition of being protected from or not exposed to danger; safety.

How might someone or something be exposed to danger? Exposure to danger

[13] A link may be considered a primitive type of form consisting of only a single submit button.

```
interface I {
      type cookies = string
      type uri = string
      type parameters = (string × string)list

      type request = uri × parameters × cookies

      type response = uri × parameters × cookies

      U1(in request req, out response res) assume { }
            guarantee { }
      ...
      Un(in request req, out response res) assume { }
            guarantee { }
}
```

Figure 2.14: Interface of a web-application with URIs=$\{U_1 \ldots U_n\}$ using the formalism of Section 2.3.

could arise in one of two ways: (i) through a direct action; or (ii) through inaction. In the first case, the entity whose action creates a dangerous situation is commonly known as an *attacker*. In the second case there is no explicit attacker; the danger arises through lack of an preventative action. To understand how inaction might be dangerous, consider a nuclear power plant safety system which reduces the reaction rate by lowering reaction-inhibiting control rods. If the system is prevented from responding during an emergency then a reactor meltdown may become inevitable.

The definition of security above suggested that security is synonymous with safety. It is worth noting that in computer science the term "safety" is used in other contexts. A program is said to be "type-safe" or "memory-safe" if it does not violate any typing rules or perform illegal (i.e. out of bounds) memory accesses. Systems which are "type-safe" or "memory-safe" are free from certain classes of run-time error. A program which is not type-safe or memory-safe may well exhibit unintended behaviour which could lead to security problems. For example a program which is not memory-safe might be vulnerable to a *buffer overflow* attack in which parts of the program's code and data are overwritten with data supplied by an attacker effectively subverting the process.

A more specific definition of *computer security* comes from Bruce Schneier in the forward to Ross Anderson's book on Security Engineering [13]. The definition concentrates on the properties of an attacker, the entity whose action leads to a dangerous situation:

> Security involves making sure things work, not in the presence of random faults, but in the face of an intelligent and malicious adversary trying to ensure that things fail in the worst possible way at the worst possible time ... again and again.

This idea – of constantly fighting against an intelligent and malicious adversary – is famously described as "Programming Satan's Computer" by Needham and Anderson [14].

Core to computer security are the concepts of *confidentiality, integrity* and *availability* (CIA) which refer to keeping data secret, ensuring data remains intact and ensuring systems are responsive. These concepts are described in the following sections: confidentiality is described in Section 2.5.1, integrity in Section 2.5.2 and availability in Section 2.5.3. We are usually concerned with the security of complete systems rather than separate components. In particular this thesis is focused on security of mobile communicating systems and so a brief introduction to distributed systems security is given in Section 2.5.4. Engineering secure systems is difficult. Several reasons are suggested for this including (i) lack of composability (described in Section 2.5.6); and (ii) the security of a system only being as strong as that of the least secure subcomponent (described in Section 2.5.5). This section finishes with a discussion of Application-Level Security in Section 2.5.7 and a brief note on the relation between security, control and policy in Section 2.5.8.

2.5.1 Confidentiality

Confidentiality involves limiting the disclosure of data by ensuring that it only becomes known to authorised users. A common method to ensure confidentiality is to *encrypt* data known as the *plaintext* with a secret *key* known only to the authorised users to produce encrypted text known as the *ciphertext*. A good encryption scheme will make it difficult or impossible to recover the original data from the encrypted data without knowledge of the key.

The split between the encryption algorithm and the key is somewhat arbitrary. In his article on military cryptography [104] Kerckhoff put forward a number of security principles. The most famous principle (summarised by Claude Shannon as "the enemy knows the system") is that one should assume the enemy already knows the workings of the security system (i.e. the algorithm) and therefore overall security should only depend on the secrecy of the *key*. The usual assumption is that the key is some entirely arbitrary, application instance-specific, secret piece of data whereas the rest of the system is fixed and well-known.

A very simple example of an encryption algorithm is the *Vernam* cipher also known as the *one-time pad*. The algorithm requires that the length of the key in bits is the same as the length of the plaintext data. The ciphertext output of the algorithm is given by the bitwise exclusive OR of the plaintext and the secret key. The decryption process is also trivial; the bitwise exclusive OR of the ciphertext and the key yields back the plaintext. This cipher is is noteworthy in that it is one of a small class of ciphers[14]for which it has been proved mathematically that the plaintext can only be determined from the ciphertext through knowledge of the key.

Encryption by itself is not always enough to guarantee confidentiality. It is possible to imagine a system in which simply the presence of an encrypted message implicitly conveys information about the value of confidential data. In this situation an attacker may never need to discover the key or break the encryption algorithm in order to attack the system.

2.5.2 Integrity

Integrity refers both to data having sensible values (e.g. a sensible date value would consist of a valid combination of day, month and year fields) and the data not being corrupted. The definition of the set of sensible data values entirely depends on the application concerned. Data corruption can occur either naturally (e.g. as an error on a communication channel) or maliciously (e.g. through the deliberate action of an attacker). Data can be protected from accidental corruption by appending an error-detection code which has the property that errors in

[14]The ciphers vary in the function used to combine the data with the key. In addition to exclusive OR, the groupwise addition and subtraction of n bits modulo 2^n also works.

the source data are very likely to require a change in the matching detection code. Data received with a non-matching detection code are considered to have been corrupted. So-called hash functions (one-way functions) are often used for this. A commonly-used hash function is MD5 [136].

Malicious data corruption is more difficult to counter. A simple hash like MD5 is insufficient because hash functions are assumed to be public knowledge (see Kerckhoff's Principle in Section 2.5.1) and easy to compute. An attacker would still be able to corrupt the data provided they also recomputed the corresponding hash. An alternative is to encrypt the hash value with a secret key to form a *Message Authentication Code* (MAC). Only authorised parties possess the key and can generate and verify matching MACs. It is possible to use a hash function to build a MAC using an algorithm such as HMAC [20].

2.5.3 Availability

Availability refers to ensuring a system can respond to requests in a timely manner (recall the example of the nuclear plant safety system described earlier). Note that having a system which fails to respond may be even worse than having no system at all if the users have grown to rely upon it. There are many reasons why a system may fail to respond. An accident or natural disaster might disable a system by compromising its power supply. Alternatively, a malicious user might launch a *Denial of Service* (DoS) attack, flooding a system with bogus requests and causing it to become overloaded.

2.5.4 Distributed Systems from a Security Perspective

At the beginning of this chapter a diagram was drawn in Figure 2.2 depicting communicating mobile agents. The system was drawn in layers, with agents at the highest layer and the physical hardware at the lowest layer. This section revisits this idea with a simple example of a distributed system which is then used as a reference for subsequent security-related discussion. A distributed system by definition consists of multiple components connected by a network. The network providing the interconnect should properly also be considered a component with properties determined by the network type; for example a trusted local area network has vastly different performance and security characteristics to the untrusted global Internet. Each component is structured in a number of layers from most

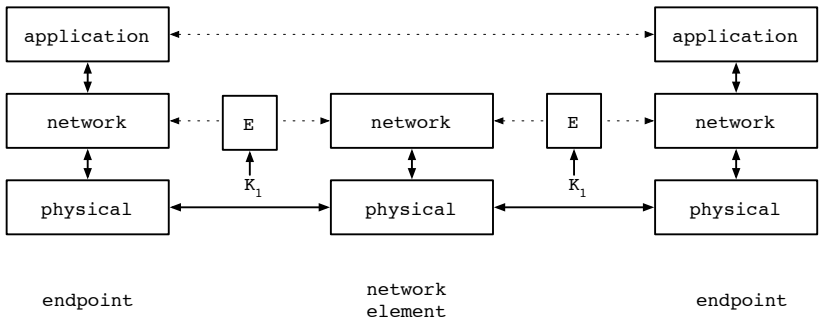


Figure 2.15: Diagram depicting a distributed system with 3 components: 2 endpoints and 1 network element. Each component consists of a number of software layers. Solid arrows indicate actual communication between components. Dashed arrows indicate how layers within components conceptually communicate with peer layers in other components.

abstract at the top to least abstract at the bottom.

Figure 2.15 shows a simple distributed system containing 5 components: 2 endpoints, 1 network element and 2 encryption modules (labelled "`E`"). In this example the endpoints have 3 layers labelled `application`, `network` and `physical` representing roughly the application code, the OS network code and the physical network interface respectively. These layers correspond to the ISO OSI layers (first introduced back in Section 1.3) of the same names, the rest left out for simplicity. The network element has everything except the application code. The solid arrows represent actual communication; vertically through each stack and horizontally across the network. The dashed arrows indicate how layers conceptually communicate with peer layers in other components, even though their messages are physically routed via lower level layers. In this example the encryption modules are intended to protect the confidentiality of the network communications. Both of the encryption modules are configured with the same secret key: K_1.

2.5.5 As Strong as the Weakest Link

The security of a system is only ever as strong as its weakest link. This follows from the assumption that a sensible attacker will choose to attack the system where it is weakest i.e. where the attack has the greatest chance of success. A corollary

of this is that a system designer must work to defend *all* parts of a system, making them equally hard to attack. An interesting example of this principle in action is the case of the famous physicist Richard Feynman who noticed a number of secret document safes containing top-secret nuclear information while working at the Los Alamos research facility. In his book [63] Feynman describes how these safes, although they *looked* very solid and secure were vulnerable to simple lock-picking attacks. A thief required only a small piece of metal to open the lock rather than the sophisticated cutting gear required to break through the walls.

Consider the distributed system design from Figure 2.15. An attacker might try to break the encryption algorithm being used by the encryption modules to gain access to the information being transmitted across the network. However, if the cipher is strong and the key handled safely then this attack is likely to be very difficult. Imagine the distributed system is a web-application running over HTTPS (i.e. HTTP over SSL). SSL uses strong ciphers and handles session keys securely: keys are transmitted protected by public key cryptography, where the public key of the server is itself signed by a trusted third party. It is difficult to attack SSL directly. Instead, an attacker may be able to find and exploit a vulnerability in the web-application software (many examples of such vulnerabilities are given later in Chapters 5 and 6). Attacking the system at the application layer completely avoids the need to break the cipher or guess the SSL session key. More discussion of application-layer security vulnerabilities is presented later in Section 2.5.7.

2.5.6 Composability

Security is not composable. Consider the distributed system displayed in Figure 2.15. Imagine a decision is taken to increase the security of the network-level encryption and so an extra pair of crypto modules are purchased and installed in series with the existing modules. A fragment of the resulting system is displayed in Figure 2.16. Furthermore, imagine the encryption modules use the "one-time pad" cipher first mentioned in Section 2.5.1 in which the plaintext is bitwise exclusive-ORed with a secret key. Recall that the "one-time pad" cipher is provably completely secure in the sense that the plaintext cannot be recovered from the ciphertext without knowledge of the key. Now imagine that, in a configuration error, the keys used by both encryption blocks are the same i.e. $\mathtt{K}_1 = \mathtt{K}_2$.

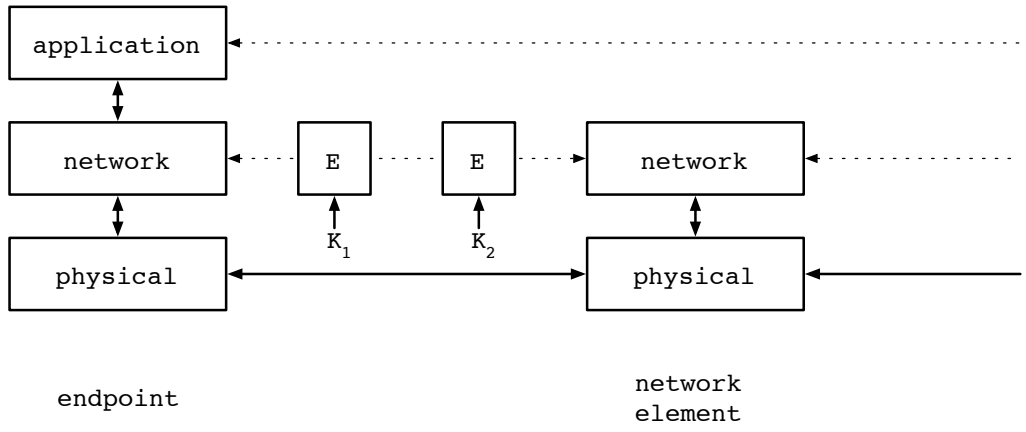


Figure 2.16: Diagram depicting part of the system from Figure 2.15 where an extra network-layer encryption module has been added in series with the existing module.

The net effect of both encryption blocks is to exclusive-OR the plaintext twice which results in the secret data being output in the clear. Thus, albeit in a rather contrived example we have managed to take two perfectly secure systems and compose them in such a way as to form a perfectly insecure system.

2.5.7 Application-Level Security

Earlier in Section 2.5.5 it was argued that the security of a system is only ever as strong as that of the weakest link. An example was described in which the messages sent on the network were encrypted but the application running on top (a web-application) had an *application-level* vulnerability. Rather than defeat the cryptography, an attacker only had to exploit the application-level vulnerability.

Exact definitions of application-level security vulnerabilities vary. SPI Dynamics report [132] define application-level security vulnerabilities as those weaknesses created when the constituent components of an application are combined together. In Chapter 1 arguments were presented that future systems are likely to be constructed out of communicating mobile agents. Therefore for the rest of this thesis the following definition will be used:

> An application-level vulnerability is a weaknesses in a system caused by unintended component movement or interface communication.

In their report, "Application-Level Firewalls Required", Gartner [127] describe how current security measures are targeted at the network-level. They operate by blocking network packets destined for vulnerable services. Gartner predict that in the future systems will be attacked primarily at the application-level; i.e. rather than attack the system hosting the service (where the system is protected by a network-firewall and security is the strongest) crackers will instead attack the service itself.

There is much empirical evidence that current systems are vulnerable to application-level attack. According to a ZD-NET survey [113] 30-40 percent of all e-commerce sites and according to Internet Security Systems (ISS) [95] 11 widely-deployed shopping carts are vulnerable to simple application-level attacks. The fact that so many vulnerabilities are being found in applications should not really be surprising. There are many more different applications in existence than there are operating systems or network stacks. Most web-applications are bespoke pieces of software which deployed on commodity operating systems. Operating systems tend to be heavily studied for security vulnerabilities; for example the OpenBSD Project [6] continually performs security audits of its core operating system code. For the remainder of software which is not so carefully scrutinised, this thesis advocates the use of application-level policy languages to protect against many common kinds of vulnerability.

2.5.8 Security, Control and Policy

Control is an important aspect of security. Control can be considered as the ability to respond to a threat or potentially unsafe situation. To control a system is to be able to exercise power or authority over the system, to check that everything is working properly and to orchestrate corrective action or impose a suitable restraint if necessary. Should a system develop a fault (through either accident or malicious attack) an ability to control the system (e.g. to shut it down cleanly) may permit a reduction in the amount of harm caused.

2.6 Model Checking using PROMELA and SPIN

Model checking [42] is a technique for the automated verification of *reactive systems* where a reactive system is one which consists of components communicating

with each other and their environment, like the layered systems described earlier in Section 2.5.4. In order to model check a system one must first represent it by a suitable model. The models described here are all finite, however there exist infinite-state model checking algorithms which explore the state space lazily. The model must be detailed enough to exhibit the interesting behaviour faithfully while remaining as small as possible. Desirable properties are represented as temporal logic[15] formulae and therefore model checking corresponds to checking the truth of the formulae with respect to the model. Model checker programs operate by exhaustively searching the state-space of a model and may never terminate in a reasonable time if the model is too complex and the state-space is too large.

The finite models used in model checking are based on Kripke structures defined as the four-tuple:

$$M = (S, S_0, R, L)$$

where S is a finite set of states, $S_0 \subseteq S$ is a finite set of initial states, $R \subseteq S \times S$ is a transition relation between states such that $\forall s \in S . \exists s' \in S . R(s, s')$ and $L : S \rightarrow 2^{AP}$ is a function mapping every state to a set of *Atomic Propositions* (APs) which are true in those states. For a formula f written in temporal logic, model checking corresponds to verifying that $M \models f$. Common choices for temporal logics include Computation Tree Logic (CTL) and Linear Temporal Logic (LTL).

2.6.1 PROMELA

PROMELA [88] (PROcess MEta LAnguage) is the language used by the SPIN (Simple Promela INterpreter) model checker to specify models. In addition to checking LTL properties, SPIN can also verify that PROMELA models are free of deadlocks, race conditions, assertion violations and that liveness properties hold (i.e. "good" states happen eventually). The rest of this section describes the main features of PROMELA needed for the rest of this thesis.

A PROMELA program consists of a finite number of processes, interconnected by channels. Processes consist of blocks of individual atomic statements; the statements from two processes running concurrently are interleaved arbitrarily. Multiple statements can be grouped together into larger blocks using the keyword

[15] Temporal logics are modal logics where each distinct universe corresponds to a system state at a particular point in time.

Type	Size in bits	Signed or unsigned
bit	1	unsigned
bool	1	unsigned
byte	8	unsigned
short	16	signed
int	32	signed

Figure 2.17: Table describing basic PROMELA data types.

`atomic`; such blocks are guaranteed, if executed, to be executed atomically and not interleaved with other statements.

Variable declarations

Variable declarations are scoped and may appear either globally or inside individual process specifications. Each declared variable has a type drawn from a set including (i) basic types; (ii) structured types; and (iii) channel types. The basic types are: `bit`, `bool`, `byte`, `short` and `int`. Details of the basic types may be found in Figure 2.17.

Variables associated with a special type called `mtype` are associated with a global set of symbolic constants. For example the following PROMELA program fragment:

```
mtype = { SYN, ACK, FIN }
mtype a = SYN;
```

declares a variable `a` of type `mtype` which has been assigned the value `SYN`. Structured types include records and arrays. Record types are declared using the keyword `typedef` in the following way:

```
typedef Structure {
  bool flag;
  int value;
}
```

In this example, a new record type named `Structure` has been declared with fields `flag` and `value` accessed using the syntax `x.flag` and `x.value` on a variable `x`. An array type can be declared using the C-like syntax:

```
bool flags[N];
```

where `N` is the number of elements in the array.

Finally, channel types are represented by the type `chan`. In the following program fragment:

```
chan a;
```

the variable `a` has been declared as a channel type.

Defining and running processes

Processes are declared using the keyword `proctype`. Processes have a list of formal parameters and a block of statements. Processes are instantiated through the keyword `run`. The same process specification may be instantiated several times with different parameters through repeated calls to `run` within special `init` blocks, where the `init` block is analogous to the `main` function in a C program.

Blocking behaviour

Each individual atomic statement contained within a process specification may be either a command or an expression. Each has a notion of "executability"; being said to be either enabled or blocked, depending on the state of the program. Assignments and unconditional branches (using the `goto` keyword) are always enabled. Expressions are enabled if they evaluate to true e.g. the expression

```
x == 1;
```

blocks until the variable `x` has the value `1`. Conditions are written using the syntax:

```
if
:: (stmt_0) -> block_0
...
:: (stmt_n) -> block_n
fi
```

For each branch $0 \ldots n$, the branch is enabled if the corresponding guard statement ($stmt_0 \ldots stmt_n$) is true. If more than one branch is enabled at the same time then one enabled branch is chosen non-deterministically.

Channel Communication

Channels are declared using syntax like the following:

```
chan c = [1] of { mtype, bit };
```

In this example, the channel `c` is declared as a channel with a buffer length of 1 accepting messages which are pairs of `mtype` and `bit`. Channels can be created with any finite buffer length greater than or equal to zero. A buffer length of zero indicates the channel is synchronous — i.e. a write to the channel blocks until another process simultaneously reads — while a buffer length of greater than zero indicates the channel is asynchronous.

Sending and receiving are performed using the `!` and `?` operators respectively. The following command:

```
c!SYN,1;
```

transmits the pair $\langle$ `SYN`, `1` $\rangle$ on the channel `c` (recall the channel `c` was declared above to accept pairs of `mtype` and `bit` values). If the channel is empty then this command is enabled and will not block. If the channel is full (i.e. in the case of channel `c` a full channel would contain only one message) then the command is blocked. Note also that if more than one write is blocked on a full channel and another process performs a read, one of the blocked write commands is selected non-deterministically to complete i.e. there is no input queueing or notion of fairness for blocked I/O commands.

Values can be read from the channel `c` using the following syntax:

```
c?a,b;
```

The variables `a` and `b` should be of type `mtype` and `bit` respectively[16]. As well as a standard receive operation, PROMELA supports a non-side-effecting receive operation which behaves like a normal receive but without removing the message from the channel. The following command:

```
c?<a,b>;
```

copies the next message in the channel (a pair of `mtype` and `bit`) into the variables `a` and `b`. Future reads will read the same value.

[16]Like C, PROMELA will perform automatic casts of types to other compatible types. This mechanism is ignored to simplify the presentation.

Assertions

Assertions may be added to PROMELA programs using the syntax:

```
assert(e);
```

This assertion causes the program to abort if the expression `e` evaluates to true in the current execution of the program.

2.6.2 Simple PROMELA example

This section demonstrates the capabilities and syntax of PROMELA via a simple example, based on the Internet protocol TCP. TCP is a peer-to-peer protocol[17] for establishing reliable stream-based connections over datagram networks. TCP has many advanced features including error recovery, flow and congestion control which will not be modelled here. Instead TCP shall be modelled as a simple client-server system in which a client connects to a server and then disconnects, transmitting no data.

Figure 2.18 shows an example PROMELA model of a system comprising two components: a client and a server. The components are represented by processes started atomically using `run` statements in the `init` block at the bottom of the program text. The processes are connected by two channels: `c_to_s` and `s_to_c` for client to server and server to client communication respectively. The client proctype transmits a `SYN` before receiving from the channel. If the client receives a `RST` (i.e. if the server rejects the connection) then the client jumps to the label `done` and exits. If the client receives a superfluous `SYN` then the assertion `assert(false)` is triggered and the computation aborts. If the client receives an `ACK` then it transmits a `SYNACK`, confirming the connection before transmitting a `RST`.

The server process blocks waiting for a client to attempt a connection by sending a `SYN` message before non-deterministically choosing between two options: (i) rejecting the connection by sending a `RST` and jumping to the label `done`; and (ii) accepting the connection by transmitting an `ACK` before receiving a `SYNACK`,

[17] Although many people talk about "clients" and "servers" in the context of TCP it itself has no such concept. Connections can be initiated by either party and even initiated simultaneously by both sides.

at which point the connection is considered open. The server then expects a RST[18] from the client, closing the connection.

Figure 2.19 shows two example *message sequence charts* corresponding to potential execution paths through the PROMELA program from Figure 2.18. The topmost chart shows the classic TCP 3-way handshake (SYN, ACK, SYNACK) before the connection is closed by the client sending the server a RST message. The bottommost chart shows the server rejecting the client's connection request by replying to the SYN with a RST.

2.6.3 Model checking with SPIN

Once a valid PROMELA program has been created it is possible to mechanically verify some useful properties with the model checker SPIN. Three general techniques shall be covered here but the list is by no means exhaustive. The techniques are:

- detecting invalid end states;
- detecting assertion violations; and
- detecting lack-of-progress cycles.

Invalid End States

An *invalid end state* is one in which either (i) all spawned processes have exited leaving unread data in a channel; or (ii) the system is deadlocked i.e. all processes are active but blocked. Invalid end states often happen when a message arrives on a channel at an unexpected point; for example, if the client process in Figure 2.18 were to send anything other than a SYN message at the beginning then the server would remain blocked and the system would deadlock.

Assertion Violations

The client process in Figure 2.18 has the command `assert(false)` which is triggered if the client receives an unexpected SYN when it is expecting either a ACK or a RST. SPIN is able to search through every possible execution sequence and verify that this assertion is never triggered.

[18] Normally a TCP connection is put into a special *closing* state first using a FIN message but that will not be modelled here.

```
mtype = { SYN, ACK, SYNACK, RST }

proctype client(chan input; chan output){
    output!SYN;
    if
    ::  input?RST -> goto done; /* refused */
    ::  input?SYN -> assert(false); /* impossible */
    ::  input?ACK -> skip;
    fi;
    output!SYNACK;
    /* established */
    output!RST;
done:
    skip;
}

proctype server(chan input; chan output){
    input?SYN;
    if
    ::  true -> { output!RST; goto done }
    ::  true -> output!ACK;
    fi;
    input?SYNACK;
    input?RST;
done:
    skip;
}

chan c_to_s = [1] of {mtype}
chan s_to_c = [1] of {mtype}

init {
    atomic {
        run client(s_to_c, c_to_s);
        run server(c_to_s, s_to_c);
    }
}
```

Figure 2.18: A simple example PROMELA program consisting of two processes: `client` and `server` interconnected by two channels. The program models a small part of the TCP protocol.

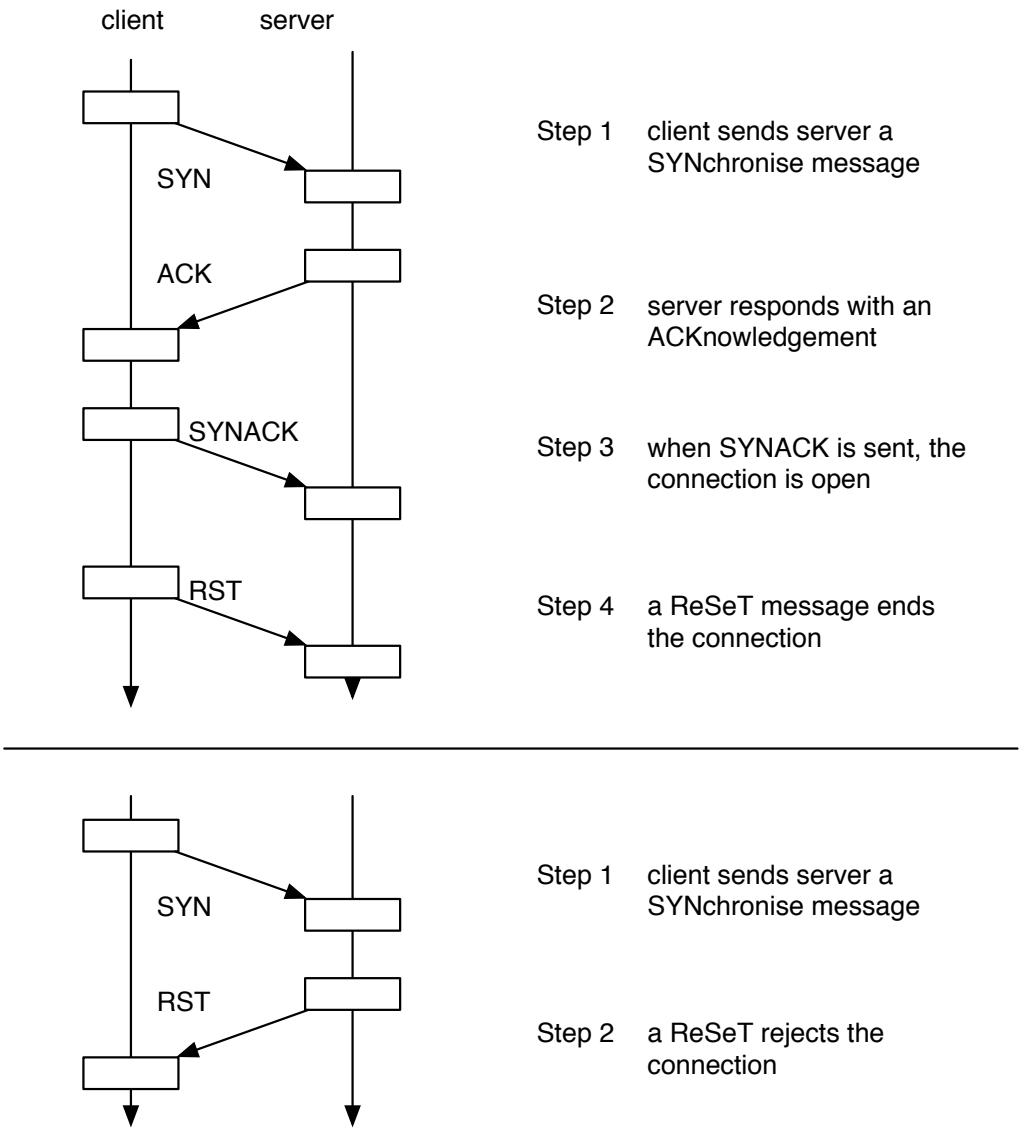


Figure 2.19: Two message sequence charts corresponding to two possible runs of the program in Figure 2.18. [*Top*] shows a connection created and then shut down; [*Bottom*] shows a rejected connection.

Lack-of-progress Cycles

SPIN can detect livelocks in programs as well as deadlocks and assertion failures. A livelock is defined to be any infinite loop in the program (i.e. a cycle of states in the state machine) which does not go through a specially marked *progress* state. In SPIN terminology a livelock is known as a lack-of-progress cycle. A state is marked as a progress state by attaching a label with the prefix `progress_`. In the TCP example from Figure 2.18 the client and server could be made more realistic by adding a label `start:` at the beginning and a `goto start` command at the end. The modified processes would be able to loop forever, continually starting up and shutting down connections. As well as being a closer approximation to reality, this would likely help find bugs where messages from a previous connection prevent a new connection from being successfully started. However now that the program contains infinite loops, SPIN has no way to distinguish desirable looping (continually starting and stopping connections) from undesirable, buggy loops where (for example) the server enters an infinite loop part-way through establishing one particular connection. To resolve this difficulty we need to change the label at the beginning from `start:` to `progress_start` which signals to SPIN that the only valid infinite loops must go through this state and any other loop represents livelock.

2.7 Summary

This chapter provided background on several topics essential for understanding the following chapters. Mobile Agents – software programs which can move themselves between hosts on the network – were described in detail. Since future applications are likely to be physically as well as logically mobile, the ability for a device to sense its location is becoming more important. Several possible location-sensing mechanisms were presented along with several arguments as to why future devices are likely to be equipped with such mechanisms. A simple formalism for describing communication interfaces was presented and web-application were described as a concrete example. Relevant topics within the field of computer security were discussed and a brief introduction to model-checking – a technique for automatically verifying properties of communicating systems – was presented.

CHAPTER 3

A Spatial Model

This thesis argues that abstracting application-level security policy (policy written to govern both the *mobility* and *interface behaviour* of systems) is an effective technique for guarding against application-level vulnerabilities which it was argued, would otherwise plague future ubiquitous computing systems. This chapter and Chapter 4 concern the security issues involving the *mobility* aspect of systems; Chapters 5 and 6 focus on those involving the *interface behaviour*.

Two distinct types of mobility have been introduced already. Physical mobility, in which entities running computer programs move in three-dimensional space was introduced first in Section 1.1. It was argued that physical mobility will inevitably increase as hardware devices proliferate and become ever smaller. Logical mobility, in which mobile agents running on computers move themselves between hosts was introduced first in Section 1.5 and described in more detail in Section 2.1. Both types of mobility are inherently related; the logical location occupied by a running mobile agent must be associated with a particular CPU which itself is a physical object with a defined physical location at a given point in time. The focus of this chapter is to produce a unified representation of space known as the Unified Spatial Model (USM) which allows users and applications to reason about both physical and logical movement within the same framework. Users will

be able to use this framework to write new kinds of security policies, explored in more detail in Chapter 4, which combine elements of traditional security policy ("no entry to unauthorised personnel") with computer security policy (e.g. " no access to this resource"). Later in Chapter 7 *sentient mobile applications* are introduced which use this framework to *sense* their environment and *react* accordingly.

This Chapter describes the Unified Spatial Model (USM) capable of representing simultaneously the disparate worlds of physical spaces (complete with notions of physical location, containment and proximity of objects) and *virtual* spaces i.e. those inhabited by mobile agents. The Chapter is structured as follows: Section 3.1 describes the main features of the spatial model. Section 3.2 describes how the model reacts to represent changes in the world and Section 3.3 describes how this model relates to others from the literature.

3.1 Modelling the World

We model the world as a tree of nested *entities*, analogous to *ambients* in the Ambient Calculus [34]. The Ambient Calculus is described in the related work section at the end of this chapter in Section 3.3.1. We begin our description by defining the following set of terms:

entity name: label used to name entities which may have further entities nested inside. Entities form a nesting hierarchy and sequences of entity names describe a route through the hierarchy. Examples of entity names include the names of physical places, computers and mobile agents. By convention we use η to range over all entity names and lower-case letters a, b, c, d, e to range over mobile agent entity names.

entity: description of a spatial location. Entities might consist of nested entities with an entity name, sibling entities in parallel, special entity *factories* or *void* (i.e. nothing). Note that, in a similar fashion to the Ambient Calculus, we are not restricted to describing only physical places but can represent any bounded region where activity happens. For example an office containing people may be described as an entity, as can a virtual machine containing mobile code. We use capital letters X, Y, X', Y' to range over entities.

path: sequence of entity names describing a route through the entity hierarchy naming a specific entity. Paths are written in the form of a sequence $\langle \eta_1, \ldots, \eta_n \rangle$ and are described further in Section 3.1.3.

path expressions: regular expression-like facility to efficiently name a set of entities. Path expressions are described further in Section 3.1.3.

We divide our entity names into *sorts* each representing a different kind of object. The sorts provide a simple well-formedness constraint on the model by restricting how entities may nest. The exact sorts used in any deployed system will depend on the kinds of things being modelled (e.g. an aviation-based system may introduce the sort "aircraft") but for expository purposes we restrict ourselves to the following:

room: a physical volume of space corresponding to a building, office etc.

person: an autonomous physical entity able to move between **room**s (a human or a robot) perhaps able to carry other entities

workstation: an immovable physical object which can host computer processes

laptop: a mobile physical object which can host computer processes

context: a virtual (or physical) machine capable of running mobile code

agent: a piece of mobile code

We write $e \lhd s$ to mean entity name e is of sort s. The formula $SortContainable(s_1, s_2)$ holds when entities of sort s_2 may be nested inside entities of sort s_1. This formula is defined graphically in Figure 3.1. Intuitively, it says that physical objects may nest in the obvious way (e.g. a **workstation** may nest inside a **room** and a **laptop** may nest within — i.e. be carried by — a **person**) and that computers (**workstation**s and **laptop**s) host virtual machines (**context**s) which in turn host **agent** processes. Note that agents do not contain any contents themselves; this is an artifact of the prototype implementation described later. A future version could remove this restriction.

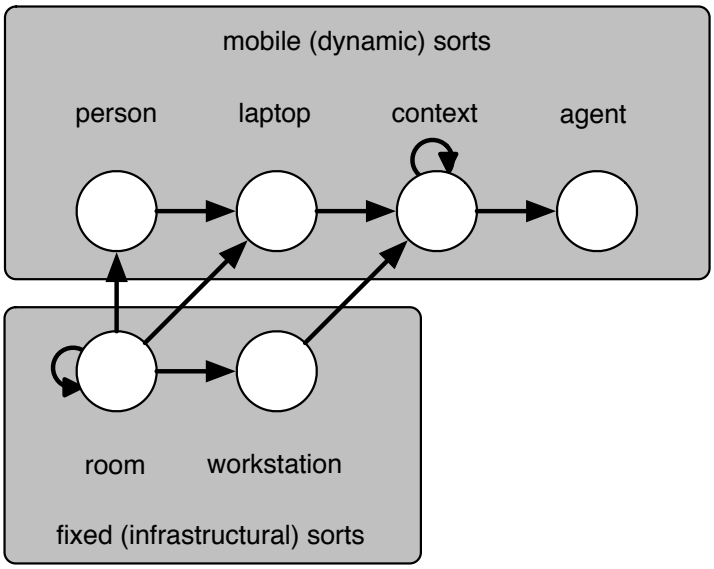


Figure 3.1: The relation $SortContainable$.

A state of our world model may be written down with the following syntax (where η ranges over a set of entity names):

$entity$	$\leftarrow$	$entity \mid entity$	(siblings)
$entity$	$\leftarrow$	$\eta[entity]$	(nesting in a place η)
$entity$	$\leftarrow$	**0**	(void)
$entity$	$\leftarrow$	$!\eta[entity]$	(entity factory)
$model$	$\leftarrow$	$\eta[entity]$	(spatial model)

By convention we consider an entity factory $!\eta[X]$ (where $\eta \lhd$ **agent**) to be a special kind of entity which can create other entities, i.e. the factory $!\eta[X]$ can spawn the entity $\eta[X]$. Note that every mobile agent which wishes to be created dynamically must be associated with at least one of these factory entities.

A spatial model (represented by $model$) consists of an entity name (the *root* entity name) plus entity contents. For the spatial model $M = y[X]$ with name y and contents X we say that M is *well-sorted* if $y \lhd$ **room** and $WS($**room**$, X)$

where the predicate WS is defined informally as follows:

$$\begin{array}{lcl} WS(s, \mathbf{0}) & \leftarrow & true \\ WS(s, X \mid Y) & \leftarrow & WS(s, X) \wedge WS(s, Y) \\ WS(s, \eta[X]) & \leftarrow & \exists s'.\eta \lhd s' \wedge SortContainable(s, s') \wedge WS(s', X) \\ WS(s, !\eta[X]) & \leftarrow & \exists s'.\eta \lhd s' \wedge SortContainable(s, s') \wedge WS(s', X) \end{array}$$

In particular note that an entity factory is only well-sorted if the entity it generates would also be well-sorted in its place and that all entities can contain an empty entity **0**. Observe that the syntax for entities is similar to the subset of the ambient calculus which has no active processes and which describes only the structure of space, like that used in the semistructured data format described in [33].

As is conventional in mobility theory we next define a congruence relation, $\equiv$, under which entities are equal up to simple rearrangements of parts. In addition to reflexivity, symmetry, transitivity and context ($X \equiv Y \implies \eta[X] \equiv \eta[Y]$) this relation (often referred to as a structural congruence relation) admits the following rules:

$$\begin{array}{ll} X \mid Y \equiv Y \mid X & \text{(commutativity)} \\ X \mid \mathbf{0} \equiv X & \text{(zero)} \\ X \mid (Y \mid Z) \equiv (X \mid Y) \mid Z & \text{(associativity)} \end{array}$$

$$\frac{X \equiv X', Y \equiv Y'}{X \mid Y \equiv X' \mid Y'} \qquad \text{(parallel)}$$

3.1.1 Example

Consider a simple environment containing people, computers and several mobile agents. A graphical depiction of the model corresponding to this world at a particular time is displayed as follows:

```
World [ Bob's office [ 0 ]
      | Charlie's office [
          Charlie [ 0 ]
        | Alice   [ laptop  [ default [ agent [ 0]]]]
        | laptop  [ default [ ! music player factory [ 0]]]
        | PC      [ default [ secret agent [ 0]]
                  | audio   [ music player [ 0]]]
```

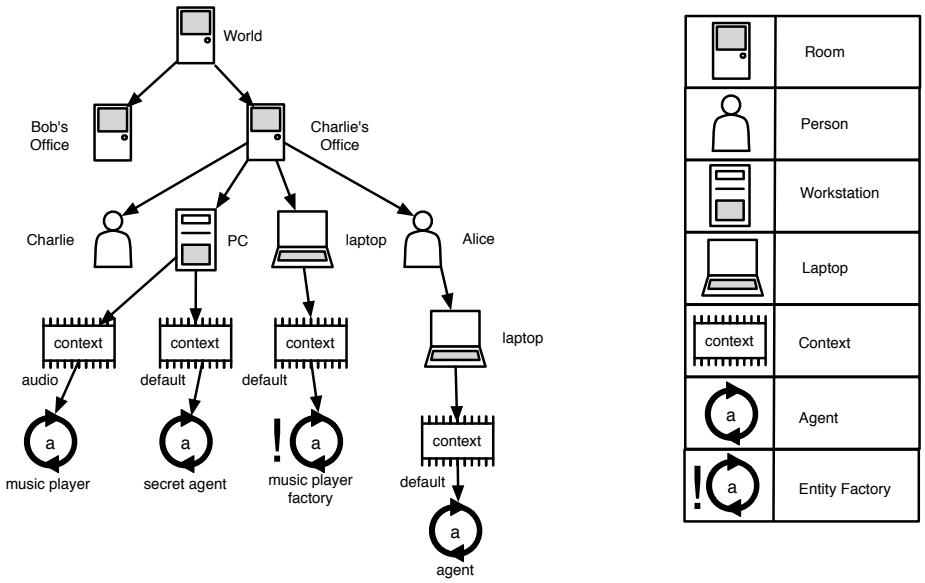


Figure 3.2: An example world configuration in graphical form.

```
    ]
]
```

There are various things to note about this configuration:

1. Alice is carrying a laptop inside Charlie's office. This laptop is currently running some mobile agent code. This mobile agent might have arrived in this room by one of two processes: (i) it may have been physically moved by Alice carrying the laptop; or (ii) it may have migrated itself deliberately after the laptop arrived in the room.

2. The PC in Charlie's office has been configured with an additional **context**, called `audio`.

3.1.2 Naming

For simplicity entity names have been presented as short, flat identifiers written in english; names such as `Alice`, `laptop` and `music player`. The actual names chosen are likely to be more complicated in practice. Policy systems like the one described in the following chapter will likely need some mechanism to re-

fer to individual objects, to either allow or disallow access to protected resources[1]. Therefore it is important to be able to create and refer to unique and unforgeable names to prevent the policy system being subverted.

On the other hand allowing non-unique names facilitates policies and conventions which apply to whole groups of things at once. For example, we may like to write a policy governing a particular class of mobile agent which is straightforward if all instances of the class of mobile agent are represented by the same entity name.

Furthermore it may be useful to represent the same object with multiple entity names collocated within the spatial model. For example, a music playing agent written by Alice might have two names: `Alice's agent` and `music player` representing two useful properties about the agent, specifically that it was written by Alice and is capable of playing music. Policies may then be written to govern either (i) all of Alice's agents; or (ii) all music playing agents rather than having one policy for each individual agent.

One possible implementation of such a flexible naming scheme would rely on digital signatures. Imagine that an agent signed by a particular key is represented by a particular entity name. If the secret part of the key is kept secret then the signature—and hence the name—is unforgeable. If the secret part of the key is made public then anyone can create entities with that particular name. A agent may be signed by multiple independent keys, reflected in the model by multiple collocated entity names.

For simplicity in the rest of this chapter we will continue to use simple english names (like `music player`) to name entities which are assumed to be unique .

3.1.3 Paths and Path Expressions

We uniquely specify a single entity name by providing a path from the root entity name using the nesting relation, $\downarrow_M$. We say that $a \downarrow_M b$ if b is a child of a in the model M, i.e. b is contained within one level of nesting of a. Therefore a path from the root $p = \langle \eta_1 \ldots \eta_n \rangle$ selects an entity name if $\eta_1 \downarrow_M \eta_2 \wedge \ldots \wedge \eta_{n-1} \downarrow_M \eta_n$. For example, in the diagram in Figure 3.2 a sequence of entity names uniquely

[1] Consider an agent called `secret agent` which is allowed access to some privileged resource by a policy based on the model presented here. If an attacker could make another agent with the same name then they will be able to access the resource illegally.

specifying the entity `music player` could be written

$$\langle \ \texttt{World}, \ \texttt{Charlie's office}, \ \texttt{PC}, \ \texttt{audio}, \ \texttt{music player} \ \rangle$$

Path expressions, similar to regular expressions, are used to quantify over a set of paths. We first define the $\downarrow_M^*$ operator as the reflexive transitive closure of $\downarrow_M$ and then write the syntax of path expressions as follows:

$$\begin{array}{rcll} element & \leftarrow & \eta & \text{(entity name)} \\ & | & \{\eta, \eta\} & \text{(disjunction)} \\ & | & * & \text{(any)} \end{array}$$

$$\begin{array}{rcl} expression & \leftarrow & element \\ & | & \bullet\bullet\bullet\texttt{/}expression \\ & | & element\texttt{/}expression \end{array}$$

We define the *matching set* of a path expression $expression$ as a set of paths $paths$ using the following rules:

- the trivial path element η *matches* an entity name η;
- the element $\{\eta_1, \eta_2\}$ *matches* the entity name η if $\eta_1 = \eta$ or $\eta_2 = \eta$;
- the element * *matches* any entity name;
- an expression of the form $element$ matches a path $\langle\eta\rangle$ if $element$ matches η;
- an expression of the form $\bullet\ \bullet\ \bullet\texttt{/}expression$ matches a path $\langle\eta_1, \ldots, \eta_k, \eta_{k+1} \ldots, \eta_n\rangle$ where $expression$ matches $\langle\eta_{k+1} \ldots, \eta_n\rangle$ for some value of k; and
- an expression of the form $element\texttt{/}expression$ matches a path $\langle\eta_1, \eta_2, \ldots, \eta_n\rangle$ where $element$ matches η_1 and $expression$ matches $\langle\eta_2 \ldots, \eta_n\rangle$.

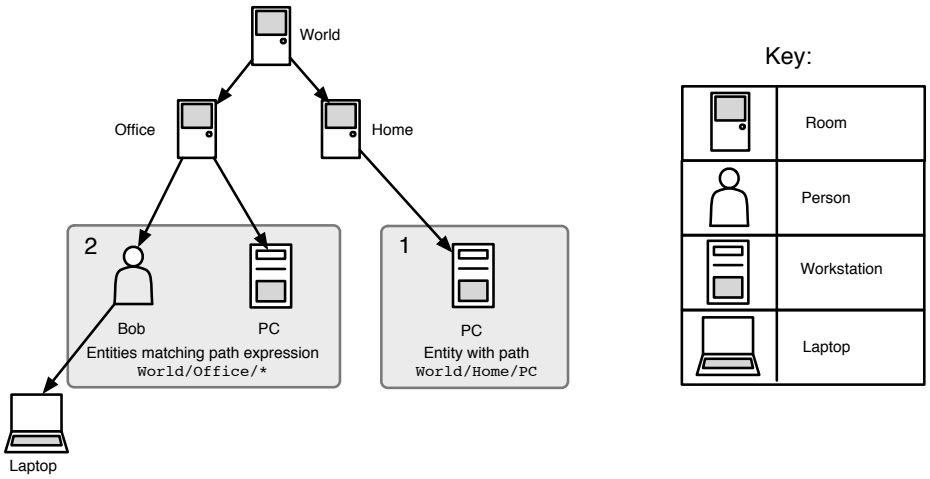


Figure 3.3: Graphical example of entities named by paths and path expressions.

Figure 3.3 shows an example spatial model with some entities highlighted. Highlighted group number 1 corresponds to the singleton entity name `World/Home/PC` while highlighted group number 2 corresponds to two names which match the path expression `World/Office/*`.

Path expressions provide a similar function to that of XPath [178], used for naming elements of XML documents. They allow us (*i*) to list concisely names whose paths diverge at a point (using disjunction) e.g. $a/\{b, c\}$ matches the same names as those matched by a/b or a/c; (*ii*) to have paths with dislocations (using `/ • • • /`) e.g. $a/ \bullet\bullet\bullet /b$ can match the same entities as those matched by a/b as well as $a/c/d/b$; and (*iii*) to quantify over names without directly listing them e.g. $a/*$ can match the same names as a/b, a/c and a/d.

3.2 Updating the World Model

The model is updated dynamically to reflect the real-time configuration of the environment by an implementation described in more detail in Chapter 7. The physical world is monitored by location sensors, a concept first introduced in Section 2.2. Changes in the physical world – specifically the movements of objects – are then reflected by changes in the model (more details of the implementation are forthcoming in Chapter 7). In addition we assume that mobile agents may be programmatically created, killed or migrated, constrained only by the installed

security policies. We define legal updates to the world by a labelled transition relation, $\xrightarrow{\gamma}$ over entities. We use labels to represent the side-effects of transitions, in particular the emission ($emit(\gamma)$) and reception ($receive(\gamma)$) of an agent during migration. The absence of a label on a transition indicates the lack of side-effects. A valid transition must have no labels at the top level – labels must always be matched and cancelled by the rule (migrate) described below. For brevity we write $a \leftrightarrow b$ if the transition is reversible i.e. if both $a \rightarrow b$ and $b \rightarrow a$ are legal transitions. The implementation (described in more detail in Section 7) ensures that every event that occurs is represented by a legal transition.

Now for agent creation with entities X, Y, Z and entity names a, b, c where $a \lhd$ **person**, $b \lhd$ **room** and $c \lhd$ **laptop** (i.e. a is of sort **person**, b of sort **room** and c of sort **laptop**) we define the following rules:

$$\begin{array}{lcll} a[X] \mid b[Y] & \leftrightarrow & b[a[X] \mid Y] & \text{(walk in/out)} \\ a[X] \mid c[Y] & \leftrightarrow & a[c[Y] \mid X] & \text{(pick up/put down)} \end{array}$$

In plain terms these rules describe how a person may freely walk into and out of rooms and pick up or drop any portable physical objects (represented by entities of sort **laptop**).

For simplicity everything that can happen to a mobile agent (i.e. being created, killed, migrated etc.) is considered as a sequence of primitive operations of the following two types: (*i*) leaving a particular context; (*ii*) entering a particular context. For example an agent creation is considered a single event – the new agent entering its initial context, next to the factory that created it. Killing an agent is a single leaving event. An agent migration from a to b is considered a sequence of two events: (*i*) leaving the source context a; and (*ii*) entering the destination context b directly afterwards.

With a view to defining mobility security policies we assume the existence of a pair of infix predicates, *can_enter* and *can_leave* which for a given agent with entity name d and context with entity name e behave as follows:

d	*can_leave*	e	holds if the policies allow d to leave the context e
d	*can_enter*	e	holds if the policies allow d to enter the context e

Mobility policies may be "plugged-in" to the model simply by defining these predicates.

For entity names $d \lhd$ **agent**, and $e \lhd$ **context** we write the rule:

$$e[!d[Y] \mid X] \quad \rightarrow \quad e[d[Y] \mid !d[Y] \mid X] \quad \textit{iff } d \textit{ can\_enter } e \quad \text{(agent created)}$$

This rule asserts that agents may be created in those places containing an appropriate agent factory (represented by $!d[Y]$) provided the new agent is allowed to enter the surrounding context. Similarly, intensional agent destruction (i.e. agent suicide) is only permitted if the agent is allowed to leave the containing context, as described by this rule:

$$e[d[Y] \mid X] \quad \rightarrow \quad e[X] \quad \textit{iff } d \textit{ can\_leave } e \quad \text{(agent killed)}$$

Agent migration between contexts is handled by the following rules:

$$\begin{array}{llll} e[d[Y] \mid X] & \stackrel{emit(d[Y])}{\longrightarrow} & e[X] & \textit{iff } d \textit{ can\_leave } e \quad \text{(agent leaves)} \\ e[X] & \stackrel{receive(d[Y])}{\longrightarrow} & e[d[Y] \mid X] & \textit{iff } d \textit{ can\_enter } e \quad \text{(agent enters)} \end{array}$$

$$\frac{X \stackrel{emit(W)}{\longrightarrow} Y \quad Y \stackrel{receive(W)}{\longrightarrow} Z}{X \rightarrow Z} \text{ (migrate)}$$

Note that the act of migration is a compound operation where the side-effect $emit(W)$ must be matched by a corresponding side-effect $receive(W)$. Therefore migration may only happen if the policies allow both the leaving step and the arriving step; it is impossible for the agent to get stuck somewhere in between. It is important to emphasise that only the results of toplevel transitions are visible to applications – applications cannot see any intermediate states of the model. We get away with this because our work so far has focused on a trusted "intranet"-style environment where complications due to unreliable network communication and partial failure are minimised.

Agent migration could be represented differently if we allowed agents to simply climb the entity hierarchy and then walk down again – the approach taken in the Ambient Calculus. This would allow us to simplify our rules by removing the labels on our transition relation. However, allowing an agent to move any-

where in the hierarchy could lead to violations of the sorting rules (described in Section 3.1). Additionally there is a subtle semantic difference with respect to the security policies: by using the "teleporting" approach described here, only the configurations at the start (the leaving step) and at the end (the arriving step) are relevant. If the agent were to have to walk from one place to another then the migration could potentially be blocked by a policy attached to an entity somewhere in the middle. Sometimes this effect is desirable; for example in the case of a firewall, agents would have to explicitly move through the firewall and be subject to its policy, rather than teleport past it, avoiding its policy.

To complete our description of how the model can be updated we have the following rules where X' and Y' are entities:

$$!\eta[X] \quad \rightarrow \quad !\eta[X] \mid \eta[X] \quad \textit{iff } \eta \not\triangleleft \textbf{agent} \quad \text{(non-agent entity created)}$$

$$\frac{X \xrightarrow{W} Y}{\eta[X] \xrightarrow{W} \eta[Y]} \text{ (nested update)} \qquad \frac{X \xrightarrow{W} Y}{X \mid Z \xrightarrow{W} Y \mid Z} \text{ (parallel update)}$$

$$\frac{X' \equiv X, X \xrightarrow{W} Y, Y \equiv Y'}{X' \xrightarrow{W} Y'} \text{ (update } \equiv)$$

Informally the first rule states that non-agent entities may be created in entity factories (note that agent entities may only be created if allowed by the security policies, using the rule (agent created) described earlier). The other three rules state that transitions may occur anywhere in the entity nesting hierarchy, in parallel with arbitrary other entities up to structural congruence. Note that the labels on the transitions are preserved but must eventually be cancelled further up the tree (by the rule (migrate)).

3.3 Other Modelling Techniques

Spatial models have been investigated within several research fields including mobility theory, ubiquitous computing and spatial database theory. Mobility theorists are interested in models of distributed computation in order to reason about distributed software in the presence of variable network latency and partial failure. Ubiquitous computing researchers found they needed some form of middleware

to bridge the vast semantic gap between (often low-quality) location data received from environmental sensors and the (high-quality, high-level) data required by their applications. Finally spatial databases have been studied at great length due to their usefulness in fields as diverse as geography and electrical engineering.

Accordingly related work is divided into three sections: the first deals with recent models of distributed computation involving explicit location, the second with ubiquitous computing middleware and the third discusses spatial databases. Each section describes the related work and contrasts it to the model presented here.

3.3.1 Theoretical Models

Many mobility models have been proposed in the literature, some of which are described in this section. We begin with a general overview and then proceeed with detailed descriptions of particular models in the following sections.

The π-calculus [116] is an early system created to help model distributed, communicating systems. The π-calculus is a *process* calculus in which concurrent active processes communicate over named synchronous channels. Channel names are treated as first-class data items and can be transmitted across other channels, providing a simple way to encode mobility. The π calculus was the basis for the programming language Pict [129].

Many variants of the π-calculus have been investigated, including the asynchronous π-calculus [90] and the distributed π-calculus [135]. The latter added several new concepts including migration, site failure, explicitly-located channels and channel names associated with permissions. The nomadic π-calculus [167], (on which the language nomadic Pict [177] is based) separates communication primitives into two types: *location-dependent* and *location-independent*. A mapping known as an *infrastructure* is used to implement the location-independent primitives in terms of the location-dependent ones.

Amadio et al [12] created a variation of the π-calculus with located channels and allowed location names to be treated as first-class data items and transmitted over channels. The calculus was used to model the locality and failure behaviour of the distributed programming language Facile [72].

The Seal calculus [37] is an extension of the π-calculus with hierarchical lo-

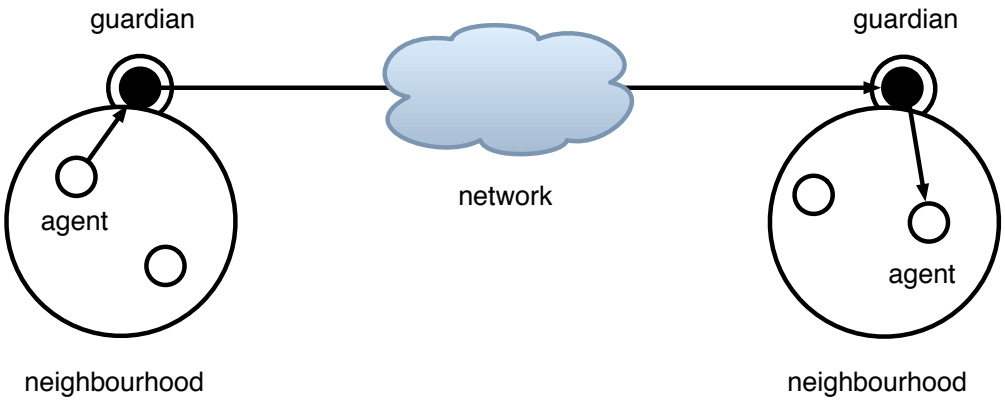


Figure 3.4: The main components of the Mob$_{adtl}$model. The large circles are neighbourhoods, the small white circles are agents, the small black circles are guardians. The arrow shows a communication path between a sender agent in one neighbourhood and a receiver in another.

cations (known as *seals*), mobility and resource access control. Processes can only communicate if they are collocated in the same seal or are in a parent-child relationship in the seal hierarchy.

The following sections describe a small sample of mobility models and contrasts them to the approach taken here.

Mob$_{adtl}$

Semprini et al [62] describe a model called Mob$_{adtl}$ used to design mobile, network-aware applications. They model the world as a flat (not hierarchical) list of *neighbourhoods* each under the control of a *guardian*. Mob$_{adtl}$ neighbourhoods are home to agents which can communicate with other agents and migrate to other neighbourhoods. All communication and migration between neighbourhoods must be routed through guardians and guardians are not necessarily directly connected to each other by point-to-point links but rather form a directed graph of nodes. Remote communication or migration may then be vetoed by any guardian on the path from sender to receiver through the guardian network. Guardians are the primary means to impose communication and mobility policy on the system. The diagram within Figure 3.4 graphically depicts neighbourhoods, agents, guardians and a communication between two agents in different neighbourhoods.

The Mob$_{adtl}$ system is specified formally using the Mark toolkit [61] written

on top of the theorem proving system Isabelle [126]. It supports designing applications by refinement starting with a specification of generic components (agents and guardians) together with a *co-ordination theory*. The co-ordination theory contains a set of assertions written in the spatio-temporal logic ΔDSTL(x) [118] which talk about the possible states of the distributed system without requiring a global clock. A typical formula might be:

$$\mathrm{m}\, p \quad \mathrm{LT} \quad \mathrm{n}\, q \quad \wedge \quad \mathrm{m}\, r \tag{3.1}$$

which says that a property p holding in component m causes properties q and r to hold in future states of components n and m respectively. Each refinement step aims to reduce the complexity of the co-ordination theory by distributing aspects of it across the individual components. The eventual aim is to finish with the co-ordination theory containing only those assumptions that can be guaranteed by the underlying network stack and middleware e.g. that messages are delivered reliably and in the original order.

There are a number of differences between Mob$_{adtl}$ and the model described in this thesis. Firstly the Mob$_{adtl}$ model uses a flat list of environments (called neighbourhoods) whereas the model described in this chapter uses a tree of nested entities inspired by the Ambient Calculus. Secondly the Mob$_{adtl}$ system is inherently generic whereas the system proposed here is targeted towards environments which have pervasive location sensing systems (i.e. modern *UbiComp* environments). Thirdly Mob$_{adtl}$ aims to formally develop applications through careful refinement steps whereas this thesis proposes the abstraction of security policies (both for mobility and communication properties) from the source code of *existing* applications.

The Ambient Calculus

The Ambient Calculus [34] is a calculus for describing the movement of processes and devices through and within *administrative domains* where an administrative domain may be considered the granularity at which policy is applied. The concept of an administrative domain corresponds to the term *context horizon* used by Shirky [154] to describe the boundary between "local" and "remote" in the world of Web-Services. It is no coincidence that work on Ambients was inspired ini-

tially by observations of the World Wide Web; Cardelli and Gordon claim [36] that the primary problem with web-programming is neither mobility nor communication (for both have realisable technological solutions) but rather in specifying and enforcing suitable policy as computations leave one domain and enter another.

In the Ambient Calculus an ambient is defined to be a *bounded* place where computation happens. Examples of possible ambients include web-pages, processes, address ranges of memory and computers, bounded by their physical cases and connectors. In the model, ambients may be nested within other ambients, possess names which can be used for access control (as part of *capabilities*), contain active threads or processes and may be migrated wholesale (i.e. including contents) to other locations. A simple example of an ambient could be written

$$n[in\ m.P \mid Q] \quad | \quad m[R] \tag{3.2}$$

where n, m are the names of ambients, brackets are used to denote nesting and $\mid$ denotes parallel composition of ambients. The sub-expression $in\ m.P$ means "execute an action associated with capability $in\ m$ and then perform P"; in this case the capability $in\ m$ denotes the capability to cause the enclosing ambient to enter sibling ambient m. Therefore this ambient expression can reduce to

$$m[n[P \mid Q] \mid R] \tag{3.3}$$

The model proposed in this chapter is heavily inspired by but is a more restricted and domain-specific version of the Ambient model. Entities, like ambients may be nested. Capabilities in the ambient calculus include permission to enter, leave and *open* an ambient; enter and leave capabilities are similar to the *can_enter* and *can_leave* predicates in the model presented here. The main difference stems from a different underlying philosophy: the ambient calculus is intended to be powerful enough to encode arbitrary computation while the model presented here is intended only to represent configurations of the world for the purposes of writing and dynamically checking policy.

Join Calculus

The join-calculus [68] is an asynchronous variant of the π calculus [116] aiming to be implementable in a distributed programming environment. The distributed join-calculus[69] is a further variation which adds the notions of named locations, migratable agents and partial failure (the presence or absence of which they describe as the "litmus test" for models of distributed computation).

The distributed join-calculus models agents as locations and allows locations to be nested inside other locations forming a tree. Migration is modelled by atomically moving a location (complete with contents) wholesale to another site. The notion of a location in the distributed join-calculus therefore has much in common with the notion of an ambient in the ambient calculus and therefore also with the entities in the model presented here.

As with the ambient calculus, the main difference is one of motivation: the distributed join-calculus is intended to represent the core of a distributed programming language whereas the focus here is to model configurations of the world for the purposes of writing and dynamically checking policy.

3.3.2 UbiComp Middleware

Traditional middleware aims to abstract away low-level details of network communication (e.g. choice of underlying protocol or naming of endpoints) from applications in order to make the task of writing distributed applications simpler. Yau et al suggest [181] that UbiComp middleware aims to exploit user *context* to foster dynamic integration between resource-poor devices and a resource-rich infrastructure in a way transparent to the application and hence make the task of writing generic ubiquitous computing applications easier. Context is a nebulous concept, reflected in the commonly-used definition from Dey and Abowd [52]:

> any information which can be used to characterise the situation of an entity

Dey and Abowd propose four primary types of context: (i) identity (of the interacting entities); (ii) location (in space); (iii) activity (i.e. actions being performed by entities); and (iv) time. Two of these types – identity and time – predate the study of UbiComp and hence are the most well understood. Activity inference is

hard problem; Hull et al report [92] that even inferring simple activity information from a mix of low-level sensor readings is difficult to make robust. Perhaps unsurprisingly a lot of effort in the UbiComp community has therefore been focused on monitoring and processing location i.e. spatial context.

3.3.3 The Location Stack

The Location Stack [85] is a "layered software engineering model for location-aware applications", similar in spirit to the OSI reference model for networking. The stack aims to take advantage of commercially available location systems and integrate research in sensor fusion allowing applications to receive combined sensor input which is more accurate or has greater coverage than any one sensor system can provide by itself. The system includes standard notions of measurement (e.g. distances between objects, frames of reference), standard algorithms for positional estimation (e.g. multilateration) and standard query types (e.g. containment and proximity). The system aims to preserve estimates of uncertainty as measurements are processed and to provide some kind of activity inference.

3.3.4 SPIRIT

The SPIRIT [11] system is a typical example of a *spatial indexing system* developed originally at AT&T Laboratories Cambridge. Primitives in the SPIRIT model are spatial containers (three-dimensional volumes of space) represented as polyhedra which may be mobile or fixed statically to the environment. The world is represented as a quad tree-like structure allowing the system to efficiently compute dynamic intersections between containers. Users register interest in these container intersection events (e.g. partial overlap) and receive notifications when events of interest occur. The SPIRIT system receives events mainly from Active Badges [172] (personal IR transmitters) and Active-Bats [174] (radio-triggered ultrasound-emitting tags) and events are emitted by callbacks in the CORBA [123] middleware system.

At a higher-level, the SPIRIT system contains a sophisticated object database which knows about many different types of physical object. Desks, chairs, walls, computers, telephones, monitors, windows and doors are all represented in the SPIRIT model. A typical SPIRIT client is the classic "active map" program which displays a top-down view of an office complete with the positions and orientations

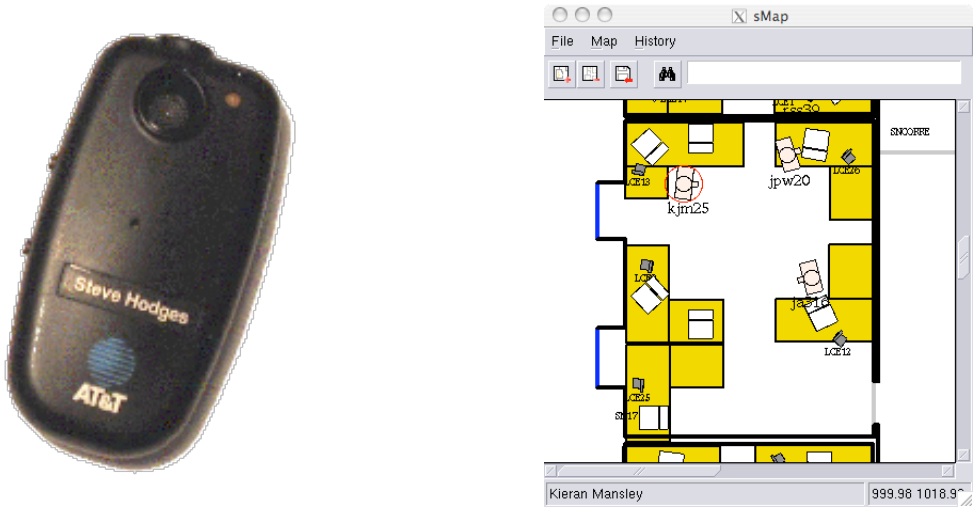


Figure 3.5: On the left: an example active bat. On the right: a screenshot of the "active map" program.

of people and equipment, updated in real time. Figure 3.5 shows an active bat on the left and a screenshot of the "active map" program on the right. Note that the map is showing a zoomed-in view of an office in the Laboratory for Communication engineering where 3 people are present, represented by usernames of `kjm25`, `jpw20` and `ja316`. Desks, computer terminals and telephones are also visible.

Relevance to this work

There are a number of differences between the model exposed by middleware like SPIRIT and the Location Stack and that proposed in this chapter. The differences arise due a difference in focus; both the SPIRIT model and Location Stack are intended to represent the physical configuration of the world as accurately as possible. In contrast the model presented here is intended to be an abstraction of reality where it is not necessary for the model to only represent objects with physical presence; virtual objects like mobile agents are easily incorporated, along side physical objects.

Both SPIRIT and the Location Stack are sophisticated systems. In SPIRIT sets of objects may partially overlap with each other while the location Stack associates notions of uncertainty with its measurements. These sophisticated features

complicate the process of reasoning about the models. In contrast the model presented here has been kept deliberately simple — supporting only simple nesting — to aid reasoning and analysis.

Both the Location Stack and SPIRIT model the world at a single level of granularity whereas the model presented here can represent some parts of the model in high-detail while using more coarse-grained representations for the rest. Coarse-grained representations have the advantage that they change at a slower rate than fine-grained representations and hence increase application *scalability*, by allowing a single server to support more clients. In this respect, the model presented here is similar in intent to recent work by Katsiri and Mycroft [103] which proposed building a two-layer logical representation where the lower layer contains accurate physical information and the higher layer maintains logical inferences using rules registered by client applications. The hope is that the higher level inferences will be more coarse-grained and hence change less often than the low level data, permitting the system to scale. The system proposed by Katsiri and Mycroft is intended to bridge the semantic gap between low-level sensor readings and the high-level knowledge required by applications whereas the model presented here is deliberately intended to be as simple as possible in order to act as a base for reasoning and writing security policies.

3.3.5 Spatial Databases

According to Shekhar et al [151] a Spatial Database Management System (SDBMS) aims to manage effectively and efficiently any data related to space. Such systems have been studied for over 20 years and have found extensive uses in many disciplines including geography (Geographical Information Systems), urban planning, astronomy, biology (mapping living organisms), pharmaceutical research and electrical engineering (e.g. VLSI and CAD).

Spatial databases primarily differ on the kinds of data they store. Some are *field-based* and store information such as rainfall or temperature as a function of spatial position. Others are *object-based* and divide up spaces into discrete entities with notions of boundaries between objects. The latter database type is evidently the most closely related to the entity-based world model presented here.

Spatial databases support numerous query types; for example the OGIS sys-

tem [125] adds object intersection tests, distance calculations, convex hull calculations and calculations of spatial unions, intersections and differences to SQL. Like the SPIRIT system described earlier in Section 3.3.4, spatial databases employ sophisticated spatial indexing methods in order to process queries efficiently. Many datastructures are used for spatial indexing, including BSP-trees, R-Trees and Quad-trees [139] (as used in SPIRIT). Spatial databases are designed to be flexible and generic, suitable for many types of applications. The model presented here is intended only to allow simple reasoning about spatial security policies and does not aim for such wide application. However by defining a suitable interface, a spatial database fed with accurate sensor information could be used as a back-end for the model presented here.

3.4 Summary

This chapter described a spatial model which provides a unified representation of space, capable of representing both the physical and logical movement of people, things and mobile agents. The model allows users to write new kinds of security policies which combine elements of traditional security policy ("no entry to unauthorised personnel") with computer security policy (e.g. " no access to this resource"). The model itself if policy-agnostic; policies may be plugged-in by defining the predicates *can_enter* and *can_leave*. A sophisticated *mobility restriction policy* is described in Chapter 4.

As well as providing hooks for implementing security policies, applications may also use the model to monitor the environment in which they exist. A special type of application known as a *sentient mobile application* will be described later in Chapter 7 which uses this framework to *sense* their environment and *react* accordingly. Implementation details for both the spatial model and sentient applications are described later in Chapter 7.

CHAPTER 4

Mobility Restriction Policy Language

This chapter builds upon the spatial model described in Chapter 3 to define *mobility restriction policies* governing the permitted movements of mobile agents in an environment equipped with physical location sensors (of the kind introduced in Section 2.2). Together with Chapter 3 this chapter contributes an application-level policy system governing the *mobility* of applications. The following two chapters contribute application-level policy mechanisms governing the *interfaces* of applications. Chapter 7 describes the implementation of the system described here.

The structure of this chapter is as follows: Section 4.1 outlines why mobility restriction policies are necessary and describes a number of scenarios in which mobility restriction policies would shield applications from attack. Section 4.2 outlines the threat model associated with this chapter. Section 4.3 describes how physical location is already used to grant people access to resources and how this technique can be extended seamlessly to mobile agents. Section 4.4 describes the language for writing policies consisting of assertions about the state of the spatial model and a corrective action executed upon assertion violation. Assertions can be violated either through the *physical movement* of a computational host or the *migration* of an agent. It is recognised that, in a realistic environment, policies

may be written by different people, with different motivations and for different purposes. Accordingly Section 4.5 describes a metapolicy for resolving conflicts between different users within a typical office scenario. Finally Section 4.7 describes security mechanisms found in other mobile agent systems and contrasts these with the system proposed here.

4.1 Motivation

The policies described in this chapter are motivated by three key trends and observations. Firstly, as first explained in Section 1.1, improvements in device miniaturisation and cost are leading to the proliferation of tiny, powerful embedded computers which can be found in everything from digital watches to fridge freezers.

Secondly, computers are becoming increasingly *physically* mobile. Current users have portable computers: laptops, PDAs and mobile phones which are mobile only by virtue of being *deliberately* carried by people. As devices become smaller one can imagine tiny computers which are *accidentally* carried by other entities (e.g. "smart dust" [102] – tiny sensor particles carried by the wind). Fully autonomous mobile computer devices known as robots are already commonplace today in factories and are likely to become more common as technology improves.

Mobile Agents were motivated initially in Section 1.5 as a sensible and economical way to structure distributed programs and then described in detail in Section 2.1. Mobile Agents are said to possess *logical* mobility. Agents can programmatically move themselves ("migrate") independently of hosting computers.

Finally, modern ubiquitous computing environments equipped with location sensors (of the kind described in Section 2.2) are capable of hosting new kinds of malicious applications not encountered before. The following subsections describe two possible new malicious applications which highlight the need for new policy mechanisms.

4.1.1 Sentient Spy

A *Sentient* application is one which can perceive its environment and react accordingly [91]. The Sentient Spy application is defined to be one which perceives the current physical location of a target person and reacts by migrating to a computer

nearby. It then activates the microphone on the computer, attempting to record any conversations which it then transmits over the network back to its controller. As the target person moves so the spy application moves to follow. Note that is it not sufficient to secure the physical workstations near potential targets to protect against this attack. Another user carrying a PDA could become the unwitting host of the sentient spy application by walking into the same room as the target and allowing it to migrate onto the PDA.

4.1.2 Human Denial of Service

A Denial of Service attack is one which prevents legitimate users from being able to interact with a particular service. Denial of Service attacks are commonplace on the Internet where it is possible to block access to particular networks or machines by flooding them with unwanted traffic, drowning out legitimate traffic. A Human Denial of Service attack is created by using a number of mobile agents which constantly monitor the physical location of a target user. The agents attempt to migrate to computers near the target user and deliberately allocate resources to prevent[1] the user from gaining access to them. For example an agent could use Bluetooth [2] interfaces to prevent the user from being able to use local IEEE802.11 networks [93] or could play lots of audio noise at high volume to prevent the user being able to talk to other people. Note that such an attack relies on the computers within the environment being configured to allow the free movement of agents. The earlier database searching example described in Section 2.1.2 relied on being able to prevent all agents from leaving their current execution environment; this may be considered as a simple mobility restriction policy. In this chapter we expand upon this idea, allowing the expression of more sophisticated mobility restriction policies suitable for preventing unwanted agent migrations in an environment permeated with hosts, agents, people and location sensors.

4.2 Threat Model

This section summarises the threat model behind the mobility restriction policies described in this chapter. The model is based within a single organisation or com-

[1]While a Bluetooth interface is scanning for other nodes enough interference is generated to prevent any IEEE802.11 nodes from successfully receiving any data. Therefore continuous scanning would cause a denial of service.

pany. The people within the organisation are associated with physical resources: specifically offices and computers (laptops, desktops, PDAs) etc. and are associated with computational resources in the form of mobile agents. We make the assumptions that

- the hardware, operating system and support libraries on all the computers are trusted;
- an organisation-wide location system exists which is accurate and responsive and which generates unforgeable location sighting events;
- all mobile hardware and personnel are tracked by the location system; and
- all mobile code runs in a Virtual Machine or sandbox under the control of the OS and system libraries (i.e. the code cannot "break free" from system control).

We do not consider interface communication here, only mobility.

The threats addressed happen when:

1. a mobile agent is physically carried into a different room and gains access to the physical space without any access control check e.g. such code can then access the "shared audio channel" in the room by playing music or recording sound;
2. a user physically moves (presumably after passing an access control check by possessing a key for a physical lock) into the same room as a computer on which an agent is running—the agent now has access to the physical space near the user; and
3. a mobile agent explicitly migrates to a new host physically near a particular user.

All three cases are symptoms of the divide between two worlds: the physical world of people under the control of locks and keys; and the ethereal world of agents with their traditional access control systems. The aim of the mobility restriction policy language is to bridge this divide and prevent threats like those listed above.

4.3 Location-based Access Control

In normal life humans are already familiar with access to resources being governed by physical location, a form of *location-based access control*. For example many physical security arrangements work by creating a guarded perimeter which only authorised people may cross. Once through the perimeter (and presumably after satisfying a security check), access to resources is granted automatically without subsequent checks. This work allows this principle to be extended seamlessly from the physical world of people to the ethereal world of mobile agents.

In order to exploit this principle, the spatial model of Chapter 3 is extended with two extra facilities: (i) the notion of sets of *privileges* associated with individual entities and inherited by entities nested within; and (ii) the notions of an entity being "jailed" and "released". Each of these will be described in the following two sub-sections.

4.3.1 The $privs$ function

Privileges are defined through a partial function, $privs : entity\ name \rightarrow privilege\ set$, which is only defined on names associated with **context** entities and gives the set of privileges which are granted automatically to any **agent** nested inside. Upon movement away from the **context** (either through intentional movement or through eviction) the privileges are revoked.Note that, in contrast to capability-based systems, it is not possible for an agent to delegate access rights to anyone else. Examples of privileges include "can_play_sound" and "can_record_sound" corresponding to the ability to play and record audio respectively. Through creating appropriate new **context**s in the spatial model and assigning appropriate privileges we gain the ability to control access to arbitrary resources. Note that we consider all agents to have permission to execute at all times; even the most deprived agent can therefore use this ability to request to migrate to a **context** which is associated with more access.

4.3.2 Jailing and Releasing

The act of jailing an agent is considered as a migration into a special context called *jail* and releasing is a migration out again. The special *jail* contexts are created dynamically when needed and associated with no privileges i.e. $privs(jail) = \{\}$.

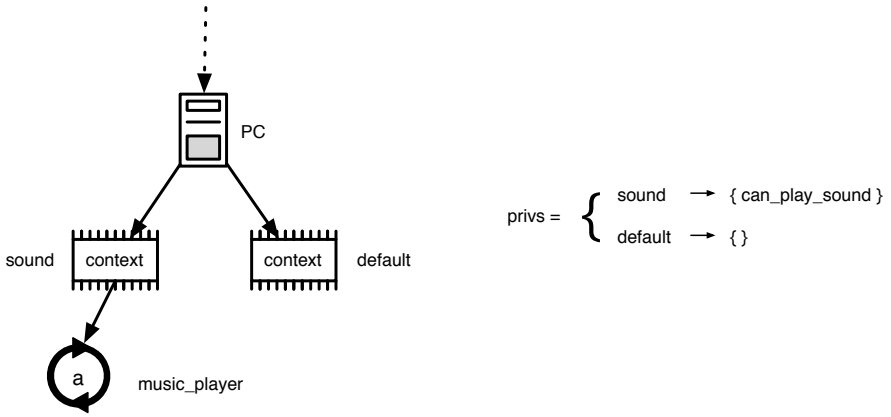


Figure 4.1: Diagram showing a PC with two **context** entities, one conferring the can_play_sound privilege on a music playing agent nested inside.

Note that although agents in jail contexts lose all their privileges (by virtue of leaving the context where the privileges were granted) they are still able to execute and in particular may request to be released again. The acts of jailing and releasing are represented by the following two extra spatial model update rules in addition to those defined in Section 3.2 and a single extra structural congruence rule:

$$\begin{array}{lcll} e[d[Y] \mid X] & \rightarrow & e[jail[d[Y]] \mid X] & \textit{iff } d \textit{ can\_leave } e \quad \text{(jailed)} \\ e[jail[d[Y]] \mid X] & \rightarrow & e[d[Y] \mid X] & \textit{iff } d \textit{ can\_enter } e \quad \text{(released)} \end{array}$$

$$e[jail[X] \mid jail[Y]] \equiv e[jail[X \mid Y]]$$

Note that, to be jailed, an agent must be allowed by the security policies to leave its current context. There is no guarantee the agent will ever be released again; release may only occur if the agent has permission to reenter the original context. The extra structural congruence rule asserts that multiple entities with name $jail$ in the same place may be freely combined into a single such $jail$ entity.

4.3.3 Example: Sound Playing

The diagram in Figure 4.1 depicts a fragment of a spatial model configuration. There is a PC entity containing two sub-entities, one of which is associated with the privilege can_play_sound via the $privs$ function. Note that if the agent were to

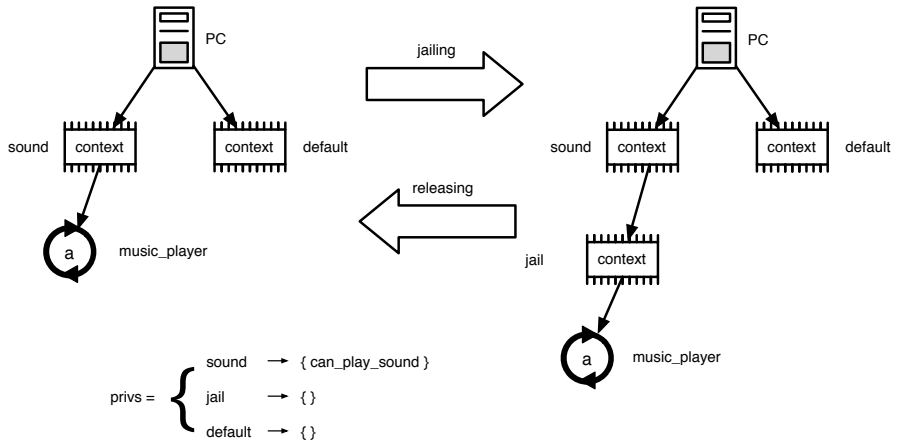


Figure 4.2: Diagram showing the agent from the example in Figure 4.1 may be jailed and released.

migrate to the other context, it would lose this privilege. Later we will see how it can be forced to migrate by the application of mobility restriction policy. The diagram in Figure 4.2 shows the same agent being jailed and then released again.

4.4 Mobility Restriction Policies

A mobility restriction policy is defined as a 4-tuple

$$\langle location, formula, times, onfail \rangle$$

where $location$ is a path expression (see Section 3.1.3) designating a set of specific entities where the assertion given by $formula$ should hold. If, with respect to the time period described by $times$, the assertion becomes violated (e.g. by the physical movement of an object) then the system will attempt to execute the command described in the field $onfail$.

It is worth noting that policy is intended to be used in one of two ways: either (i) an agent which is attempting to migrate to another host has its movement prevented; or (ii) a physical movement (such as a person carrying a laptop from one room into another) results in the violation of a policy which is "patched up" by the execution of the $onfail$ action. The $onfail$ action may be thought of as a last-ditch attempt to fix an unpreventable and undesirable situation.[2]

[2]One may imagine a system in which physical movements too may be prevented through au-

Later in Section 4.5 it is described how policies may be written (probably by different users) which conflict with each other in certain situations. Some mechanism is required which resolves these conflicts and decides which policies should be enforced at the expense of which other policies. Of course this implies that not all policies can actually be enforced in practice. Although outside the scope of this thesis, an automatic tool could be created to analyse policies and check for nonsensical policies and also potential conflicts before they happen. For example, it is possible to write policies which make untrue assertions about the physical world (e.g. a policy could insist that a certain office does not exist when it actually does). Knowing a little about how the world model behaves (e.g. that rooms never move) a tool would be able to detect some nonsensical policies up-front. A tool could also search through possible configurations of the world for situations which cause policies to be in conflict (just as a model checker enumerates the state space of a model searching for assertion violation). Potential problems could be flagged up and users could then revise their policies accordingly.

The policy field $times$ can contain one of two possible types of values: $Always(\Delta t)$ and $Sometime(from, to, \Delta t)$ (where $from$,to and Δt are all in seconds and $\Delta t > 0$). In both cases the parameter Δt specifies how much "reaction" time the system has before the policy $onfail$ action is executed. The value $Always(\Delta t)$ indicates that the assertion $formula$ should hold for all time during which the system is running. The value $Sometime(from, to, \Delta t)$ states that $formula$ should hold[3] at some point in the time interval between the times $from$ and to. Note that we do not rely on having global time synchronisation and allow individual agents to operate on their own local clocks. Therefore agents (particularly those conserving battery power) need to be given some time (represented by the Δt) to respond to policy violations.

To understand the necessity of having the "reaction" time, Δt, first consider a hypothetical scenario in which an agent is running on a battery-powered PDA with intermittent wireless network access (perhaps the PDA deliberately runs the

tomatic control of door locks. However consider that, due to Health and Safety regulations, in the event of a fire (alarm) the door controls would have to be disabled, enabling policy violations.

[3]This is similar to the concept of *obligation* in traditional Role-Based Access Control (RBAC) systems i.e. it states that someone *should* perform some action during some time interval.

network in a low-power mode or the radio coverage is relatively poor). Imagine that a policy exists which insists that for all time (i.e. the $onfail$ field is set to $Always(\Delta t)$ for some $\Delta t > 0$) the agent currently running on the PDA is not within a particular room. Furthermore consider that the database of installed policies is running on a mains-powered high-performance server elsewhere in the building and *not* on the PDA. Imagine that a user carries the PDA into the room in which the agent is forbidden. It is possible that the new location of the PDA will be sensed by the environment during a period when the PDA's network interface is inactive. Therefore the policy will be violated without the software on the PDA realising. Assuming that the agent running on the PDA wishes to comply with the policy in the first place, then it is polite to grant it a (hopefully short) grace period in which to comply before the $onfail$ action is activated. This grace period is represented by the Δt.

The policy field $onfail$ specifies an action to take should the policy be violated. The action can be of the following types:

- $Log(message)$ causes a message to be written to a log;
- $Kill(pathexpr)$ asks the system to terminate agents identified by the path expression $pathexpr$;
- $Jail(pathexpr)$ requests agents named by $pathexpr$ be jailed; and
- $Create(path)$ requests the agent factory named by $path$ create an agent.

For both the $Kill$ and $Jail$ values we adopt the convention that if the path expression has a missing initial element (i.e. it starts with `/•••/`) we automatically prepend the full path to the specific entity the formula is currently being applied to. For example if the policy $location$ field is a`/*` and the policy is violated at $\langle a, b\rangle$ then the $onfail$ expression $Kill$ `/`c is expanded to $Kill$ a`/`b`/`c i.e. a request to terminate *only* the entity named by $\langle a, b, c\rangle$ and not any other element (e.g. $\langle a, d, c\rangle$). This ability to refer to previously matched data in a pattern is also found in other systems using regular expressions, e.g. Perl [171].

The policy field $formula$ contains an expression written in a simple spatial modal logic similar to the Ambient Logic [35]. The core syntax is as follows,

where η ranges over entity names:

$$
\begin{array}{rcll}
formula & \leftarrow & \mathbf{T} & \text{(true)} \\
 & | & \neg formula & \text{(negation)} \\
 & | & formula \vee formula & \text{(disjunction)} \\
 & | & \mathbf{0} & \text{(void)} \\
 & | & \eta[formula] & \text{(named entity)} \\
 & | & !\eta & \text{(named agent factory)} \\
 & | & formula \mid formula & \text{(composition)} \\
 & | & \Diamond formula & \text{(somewhere modality)}
\end{array}
$$

For convenience the following additional syntax is defined in terms of the core:

$$
\begin{array}{rcll}
\mathbf{F} & \equiv & \neg\mathbf{T} & \text{(false)} \\
a \wedge b & \equiv & \neg(\neg a \vee \neg b) & \text{(conjunction)} \\
\Box a & \equiv & \neg\Diamond\neg a & \text{(everywhere modality)}
\end{array}
$$

These constructs may be familiar to those versed in modal logics, but their meaning is summarised in the following section.

4.4.1 Satisfaction

We say that an entity E satisfies the logical formula f (i.e. the formula f holds at E) by writing $E \models f$. Intuitively, we may think of a formula f as *matching* an entity E if $e \models f$. The relation, $\models$ is defined informally as follows:

- $E \models \mathbf{T}$ for any entity e
- $E \models \neg f$ if $E \models f$ does not hold
- $E \models f \vee g$ if either $E \models f$ or $E \models g$
- $E \models \mathbf{0}$ if $E \equiv \mathbf{0}$ i.e. e is "nothing"
- $E \models !\eta$ if $E \equiv !\eta$
- $E \models \eta[f]$ if $E \equiv n[M]$ and $\eta = n$ and $M \models f$

- $E \models f \mid g$ if $E \equiv N \mid M$, $N \models f$ and $M \models g$
- $E \models \Diamond f$ if $\exists e'.e \downarrow^* e'$, $E' \equiv e'[Y]$ and $E' \models f$

So the formula **T** matches any entity (which may itself be a parallel composition of entities) and the formula **0** only matches "nothing" (or "void") i.e. the absence of anything. The formula $f \mid g$ matches E if e can be written as the composition of two expressions N and M (remember the equivalence relation $\equiv$) such that f matches N and g matches M. The formula $\Diamond f$ matches E if there is an entity E' somewhere in the tree rooted at E where E' matches f.

4.5 Policy Conflict

If we allow individual users to write their own security policies then we must also provide a mechanism to resolve policy conflicts when they arise. Conflicts between rules in our system are similar to those found in Active Databases [45]. Many mechanisms have been proposed, ranging from simple numeric priority schemes to more complex algorithms comparing rules based on their generality [96] (e.g. the more general rule holds except when the less general does not or the other way round). There is no single best strategy that works perfectly in all circumstances. Therefore the scheme presented here is not claimed to be perfect, but rather as an example of the kind of scheme possible in a system like this. Some implications (not all of which are desirable) of the scheme presented here are outlined in the examples in the following section. Our main goal is to make the system be intuitive enough for ordinary users to understand. Security policies in our system are based on a spatial modal logic therefore we also use a spatial mechanism for arbitrating between conflicting policies.

Recall that we model the state of the world as a nested tree of entities (see Section 3.1). We observe that within a real life enterprise people too are often arranged into a hierarchy, with the boss at the top, managers in the middle and normal employees at the leaf nodes. In such an organisation, a manager would be able to set a policy which would override those of subordinates but which could itself be overridden by the boss. These two hierarchies, one describing the world and one describing the people, can be linked together by associating entities with

a set of people ("owners" or "administrators") via a function

$$owners : entity\ name \rightarrow person\ set$$

such that for an entity name η we have $owners(\eta) = \{person_1, \ldots, person_k\}$ where $person_{1\ldots k}$ are the direct "owners" of η. In a typical configuration, the boss would "own" the root entity while normal employees would "own" their individual offices. Our scheme for arbitrating between conflicting policies may be informally described as:

> For a proposed change to entity name η, policies instituted by a user $u'' \in owners(\eta'')$ override those policies instituted by a user $u' \in owners(\eta')$ where $\eta'' \downarrow^* \eta'$ and $\eta' \downarrow^* \eta$ as long as $\eta'' \neq \eta'$ and $u'' \neq u'$.

Recall from Section 3.2 that the installed security policies may be represented by a pair of predicates, *can_leave* and *can_enter* which, given an agent and a context hold precisely when an agent is allowed to leave or enter the context respectively. Both of these predicates are computed in the following way: For a proposed change in the configuration at context named c (e.g. an entity wishes to leave c) we first compute the set of users who "own" any of the entity names on the path $\langle \eta_1, \ldots, \eta_n \rangle$ from the "root" entity name η_1 which designates c

$$users = \bigcup_{k=1}^{n} owners(\langle \eta_1, \ldots, \eta_k \rangle)$$

Each user $u \in users$ is allocated a single vote on the proposed change. Note that this effectively means that although users may write policies about entity names they do not "own" these policies will be easily overridden by other users who *do* "own" these entity names. A user u votes for the proposed change if the number of their policies which are in violation decreases or the number remains the same but the agent requesting the migration is owned by them, votes against if the number in violation increases and abstains otherwise. We define a function $vote(user)$ as

follows:

$$vote(user) = \begin{cases} -1 & \text{if } user \text{ votes against the proposal} \\ 0 & \text{if } user \text{ abstains} \\ +1 & \text{if } user \text{ votes for the proposal} \end{cases}$$

We then compute the value of

$$overall\ vote = \sum_{i=1}^{n} \sum_{o \in owners(\langle \eta_1, \ldots, \eta_i \rangle)} prio(i) vote(o)$$

where $\langle \eta_1, \ldots, \eta_i \rangle$ refers to the ith entity name on the path $p = \langle \eta_1, \ldots, \eta_n \rangle$ and the function $prio(i)$ gives the priority of owners of this entity name. One possible priority function is given by $prio(i) = x^{-i}$ where x is a tunable vote weighting factor. The parameter x determines how many people who "own" an entity name η_n are needed in order to equal the vote of a single person who "owns" a "more important" entity name η_{n-1}. If $x > max_i\,(|owners(\eta_i)|)$ then it is impossible for the owner of a more important entity name to be overridden by a group of people who own a less important entity name. The system will allow the proposed change if $overall\ vote \geq 0$ and veto it otherwise.

4.6 Example

In this section we demonstrate the kinds of policies which are expressible in our system by means of a series of examples set in a typical shared workplace environment. A snapshot of the world configuration is presented in Figure 4.3. The top-level entity is named `World` and contains child entities `Bob's office` and `Charlie's office` representing the offices of users named Bob and Charlie respectively. We assume that ordinary employees by default "own" the entities corresponding to their offices and for the sake of an interesting example we further assume that Bob is the boss and also "owns" the top-level entity, `World`.

A user, called Alice, deploys a "follow-me" music playing mobile agent which follows her around, playing music where she goes. She is worried about the agent failing to follow her when she moves between rooms and so writes the following policy to cause the system to monitor the agent:

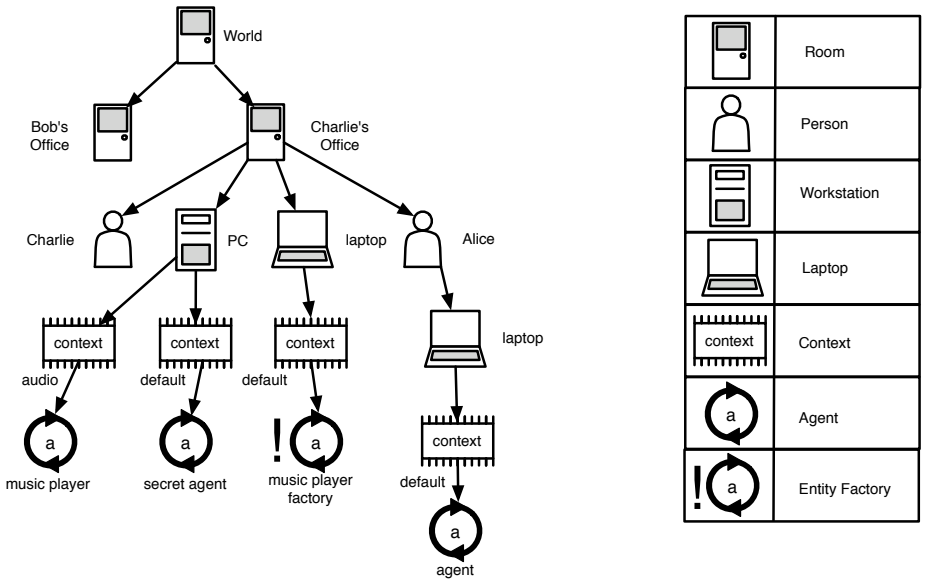


Figure 4.3: An example world configuration in graphical form.

$$\begin{array}{lrl} \langle & \mathit{location} & = \quad \texttt{World}, \\ & \mathit{formula} & = \quad \Diamond(\text{Alice}[\mathbf{T}] \mid \Diamond\texttt{music player}[\mathbf{T}] \mid \mathbf{T}), \\ & \mathit{times} & = \quad \mathit{Always}(10\ \mathit{seconds}), \\ & \mathit{onfail} & = \quad \mathit{Log} \qquad\qquad\qquad\qquad\qquad\qquad \rangle \end{array} \tag{4.1}$$

> "for all time, I am in the `World` and an agent called `music player` should be in the same space as me. If this is not true for more than 10 seconds, log the error"

The generous 10 second reaction time affords the agent an opportunity to notice Alice has moved (by receiving an event from a networked location server), stop playing music, record how much of the current track it has already played, find a new machine to migrate to and to migrate itself to its new home. If the reaction time period is shortened then the probability increases that a log entry is created even when the agent is working properly but fails to react in time. Note that the agent may fail to react for many reasons, including: the task scheduler on the hosting device may neglect to give the agent any CPU time; the network may temporarily fail, delaying the arrival of the location sighting event; or perhaps

Alice moves into a room in which the agent simply cannot find a suitable host to run on.

Remember that $E \models f \mid g$ holds whenever f and g match the two children of E and that **T** matches anything, including **0**, the absence of anything. In the formula above the third **T** means that the formula will hold irrespective of whatever else is in the same space as Alice.

The consequences of this policy are summarised as follows:

1. When the `music player` attempts to migrate, the system prevents the agent from *leaving* the same room as the user. Recall from the beginning of Section 4.4 that the system is assumed to be able to block agent migration requests which are judged to violate policies.. Note it does not directly force the agent to move properly (these are mobility *restriction* policies), it just stops it from moving inappropriately.

2. Upon observing Alice move to a new room the system generates a new entry in the system log if it has not followed her within 10 seconds. The log entry may help Alice in debugging her errant agent.

3. If the agent is running on a portable device which is picked up by someone else and moved to another room (recall that physical movements cannot be blocked unlike agent migration requests and therefore must be "patched up" afterwards using the $onfail$ action) then a new entry will be generated in the system log if the agent has not migrated to a machine in the same room as Alice within 10 seconds.

4. If Alice moves to a room which already has a `music player` agent the system will not complain even if Alice's agent fails to follow her. Recall the earlier discussion about naming in Section 3.1.2; therefore it is important that only one agent called `music player` exists at a time.

Consider a second user, Bob, who is Alice and Charlie's boss. Bob prefers peace and quiet where he works. To prevent wandering music playing agents disturbing him he writes a rule:

$$\begin{array}{lcll} \langle & \mathit{location} & = & \texttt{World/*}, \\ & \mathit{formula} & = & \Box\neg\texttt{Bob}[\mathbf{T}] \vee (\Diamond\texttt{Bob}[\mathbf{T}] \wedge \Box\neg\texttt{audio}[\neg\mathbf{0}]), \\ & \mathit{times} & = & \mathit{Always}(3\ \mathit{seconds}), \\ & \mathit{onfail} & = & \mathit{Jail}\ \texttt{/}\bullet\bullet\bullet\texttt{/audio/*} \quad \rangle \end{array} \tag{4.2}$$

> "if ever I'm in an office with a music playing agent, jail the agent if it has not left within 3 seconds"

The policy $\mathit{location}$ field `World/*` causes the rule to be applied to all children of the entity named `World`, i.e. in the diagram in Section 3.2 this corresponds to all the offices, $\langle$`World`, `Bob's office`$\rangle$ and $\langle$`World`, `Charlie's office`$\rangle$. The same formula is applied individually to each of these entities. The formula $\Box\neg$`Bob`$[\mathbf{T}]$ holds if the entity `Bob` is nowhere inside the office; the formula $\Diamond$`Bob`$[\mathbf{T}]$ holds if the entity `Bob` is *somewhere* inside the office and the formula $\Box\neg$`audio`$[\neg\mathbf{0}]$ holds if there is not a non-empty `audio` context anywhere within the office. Taken together, the whole $\mathit{formula}$ may be read as

> Either Bob is not inside the office concerned (in which case there is no violation) or he is inside the office but there is no sound playing.

If the policy is violated in the office named x then the onfail action is expanded to Jail `World/`x`/`$\bullet\bullet\bullet$`/audio/*` causing audio playing agents inside office x to be jailed.

The consequences of this policy are summarised as follows:

1. If a `music player` agent attempts to migrate inside the same office as Bob the request will be denied, assuming that his policy is not overridden by anyone more senior in the company.

2. If a `music player` agent running on a laptop or PDA is physically moved inside his office by someone else, that agent will be jailed.

Now consider what will happen when Alice enters Bob's office. Clearly the two policies 4.1 and 4.2 now conflict. Alice's mobile agent will attempt to migrate

inside Bob's office so the system will apply the conflict resolution rules described in Section 4.5. Assuming the system knows that Bob "owns" the entity named `World` (since he is the boss) his policy will override those belonging to Alice and the migration request will be blocked.

Imagine a third user, Charlie, with more malicious intent. This user attempts to lure hapless agents into his domain and then trap them there forever. He decides to target Alice's music playing agent and writes the following:

$$\begin{array}{lrll} \langle & \mathit{location} & = & \texttt{World/Charlie's office}, \\ & \mathit{formula} & = & \Diamond(\texttt{music player}[\mathbf{T}]), \\ & \mathit{times} & = & \mathit{Always}(3\ \mathit{seconds}), \\ & \mathit{onfail} & = & \mathit{Log} \qquad\qquad \rangle \end{array} \tag{4.3}$$

> "for all time the `music player` agent should remain inside my office."

Consider what happens when Alice is enticed into Charlie's office for a coffee and biscuit. Initially Alice's `music player`'s request to migrate into Charlie's office is accepted since it does not violate any policy (in fact it causes rule 4.3 to no longer be in violation – an improvement!) When Alice leaves the office the music player attempts to follow her. Charlie's and Alice's rules are now in direct conflict; Alice's rule is violated if the agent stays and Charlie's rule is violated if the agent goes. Note that both rules have an entirely passive onfail action. Unfortunately for Alice since Charlie "owns" his office his policies take priority and therefore the agent's request to leave is denied. What can Alice do? The only solution for Alice in this situation is to appeal to a higher authority – in this case Bob – someone whose policies are ranked higher than Charlie's. Bob may write a policy to evict Alice's agent, overriding Charlie's wishes.

4.7 Related Work

Many people have developed mobile-agent frameworks which included support for some kind of mobility security policies. Dag Johansen, one of the developers of the influential TACOMA [101] system recently wrote in a 10-year retrospective [99] that

> more than 100 mobile agent systems have been built, where the majority, to be modest, hardly provides any incremental contribution to the community as a whole.

Many of the systems Johansen referred to were developed independently, by different teams and using different languages and tools. However they all implemented more-or-less the same functionality. Johansen claims that the lack of a satisfactory solution to mobile-agent security problems together with a general lack of a *killer-app* and not the lack of support for a particular language or system prevented the large-scale deployment of mobile agent systems.

Rather than describe 100 near-identical mobile agent systems it suffices to mention only a few classified into the following groups: (i) traditional mobile-agent systems as described by Johansen; (ii) proposals for distributed programming languages; and (iii) middleware systems supporting object migration. Each of these groups shall now be described and their support for security policies contrasted with the mobility restriction policies presented here.

4.7.1 Traditional Mobile-Agent Systems

Traditional mobile-agent systems is the name used here for those systems which aim to create generic mobile-agent middleware for writing agents in existing languages. They have many features in common. They focus mainly on mechanism: how to transmit agents between sites (running on different platforms); how to protect sites from aberrant agents; and how to specify security policies to guard accesses to local resources. These systems are differentiated through many subtle features. One such feature is the type of mobility offered [87] (first mentioned in Section 2.1). Some systems are said to offer *full mobility*: the migration of all state including that within the kernel. Another variation is *strong mobility* in which a system is able to introspect and migrate running code. The remaining systems offer *weak mobility*, insisting that agents perform manual shutdown and restart. Even these distinctions are not very discriminating; Bettini et al describe how strong mobility may be implemented on top of weak mobility with a syntactic translation [23].

Two example systems shall be described, one academic system (TACOMA) and one industrial system (Aglets).

TACOMA

The TACOMA [101] (Tromsø And COrnell Moving Agents) project had several major revisions over a 10 year period, the features of each version are summarised in a summary paper [100]. The most recent version allowed agents to be written in a variety of programming languages and supported communication over TCP. TACOMA provided weak mobility; agents were charged with handling their own serialisation rather than relying on runtime support for freezing running code (which would prevent agents being written in many languages e.g. Java). Agent state was handled through "briefcases" which could be left permanently on one site or transmitted between sites. Briefcases contained nested "folders" containing application data.

Special "firewall agents" are used to enforce application-specific security policy on arriving agents. Depending on the security policy in force, arriving agents might find themselves executing in a private location (like a sandbox) where they cannot communicate with other agents (in TACOMA agents must be in the same location in order to communicate). In the system described here, a sandbox would correspond to a location with restricted privileges (via the $privs$ function) and a mobility policy which prevented any agents contained inside from migrating out of the sandbox.

Several aspects of TACOMA are shared with this work. The TACOMA project imagined mobile agents running on PDAs [97] and mobile phones [98], equivalent to the portable **laptop** entities used in the USM described here. The hierarchical data stored in TACOMA briefcases and transported between sites could be represented by creating a new sort of entity ("**data**") and allowing agents to have these entities nested inside. The TACOMA framework differed from the system described here in three significant ways. Firstly, TACOMA considered only mobile agents and agent servers; it had no access to physical sensor information and therefore made no attempt to support a unified representation of both physical objects and mobile agents like that described in Chapter 3. Secondly, TACOMA did not directly associate locations with access to specific resources instead preferring to restrict access to resources depending on whether a mobile agent has been signed. TACOMA has no equivalent to the $privs$ function described in Sec-

tion 4.3. Thirdly, TACOMA firewall agents would impose mobility policy by preventing certain agents from migrating to specific locations. The mobility restriction policies described in Section 4.4 also copes with unrestricted *physical* movement by triggering a corrective action *after* a policy has been violated.

Aglets

Aglets [106] (a portmanteau word derived from "agent" and "applet") is a Java-based API for building mobile agents released by IBM. The Aglet system has much in common with TACOMA with the major difference that it is based entirely upon Java. Security policies in the Aglet system are based upon the Java-2 security model [74] in which code is selectively trusted or not depending on its origin and/or the presence of recognised signatures. This signature-based scheme is totally different to that proposed here; it does not associate virtual locations with privileges nor does it attempt to handle physical movement as well as agent migration, a feature essential to cope with the expected hordes of networked mobile devices outlined in Chapter 1.

4.7.2 Distributed Programming Languages

The term *distributed programming language* is used here for projects which aim to create altogether new languages suitable for developing distributed (often agent-based) systems. The following sections describe three such systems and describes both how they approach security and how the approaches taken differ from the system proposed here.

Telescript

The Telescript system [161] is an object-oriented programming language designed to support a remote programming model offering *strong mobility*. Security in the system is addressed by the use of safe language features such as the lack of pointers, types , automatic memory management, built-in authentication mechanisms and a system of capabilities known as "permits". Telescript has the concept of "regions" each of which can have their own security policy just as **context** entities here are associated with sets of privilege. Both Telescript and the system described here allow policy to be written to prevent agents migrating to particular places. However unlike the system presented here, telescript does not directly

support policies written in terms of spaces and does not support physical regions – all regions in telescript are entirely virtual.

Obliq

Obliq [32] is a dynamically-typed language built on top of Modula-3 Network Objects at DEC SRC. It aimed to exploit the existing Modula-3 middleware facilities including remote object naming, invocation and garbage collection. Obliq introduced distributed lexical scoping: it allows procedures to be sent to remote sites complete with free variables which remain bound to their original definition sites. Issues involving security were delegated to the underlying network objects middleware. The middleware supports security through (i) callee authentication; (ii) access control lists for individual objects; and (iii) non-forgeability of references to secure network objects. This very traditional approach to security is completely different from and orthogonal to that proposed here. Chapter 1 argued that the environment will be filled with physically and logically mobile programs necessitating mechanisms to limit the mobility of software, hence the mobility restriction policy presented in this chapter. Agents will still need to be able to authenticate each other and control access to their internal state and so a system like that used by Obliq and secure network objects may run in parallel with the system proposed here.

JoCAML

Conchon et al produced a mobile-agent system for Objective-Caml based on the Join Calculus [43] (the Join Calculus was described earlier in Section 3.3.1). Furthermore, on top of JoCAML Fournet created a proof-of-concept implementation [70] of the Ambient Calculus. The Ambient Calculus was described in detail in the previous chapter in Section 3.3.1. JoCAML and the Ambient Calculus encoding are powerful tools for creating distributed systems.

Facile

Facile [72] is a high-level, higher-order programming language based on Standard ML [117] with the addition of a model of concurrent processes based on Milner's CCS [115]. Facile has the concept of a node – effectively a virtual processor – on which processes may execute. Individual processes communicate and synchronise

using typed channels; any value may be transmitted, including user-defined types, channel names and functions. In addition to possessing a robust implementation, Facile (the first release of which was known as "Facile Antigua" [164]) has been studied extensively from a theoretical standpoint. As a programming language, Facile does not include built-in support for security policies; however it could be used to create an implementation of a system like that presented here.

Distributed π

First mentioned in Section 3.3.1, the Distributed π calculus (Dπ) incorporates notions of remote execution, migration and site failure into the π calculus. Channels in Dπ are explicitly located; using a channel requires knowledge of both the name of the channel and name of the location. All names (for both channels and locations) have permissions detailing what can be done with the name. Defined permissions include:

snd : to send data along a channel

rcv : to receive data from a channel

run : to run a thread at a location

newc : to create new channels at a location

subl : to place sublocations at a location

mig : to move a location complete with all threads and sublocations

halt : to kill a location, stopping all threads from running

When names are communicated, certain permissions are communicated too. A type system is employed to ensure that well-typed terms only use received names in a manner consistent with the permissions received along with the name. Dπ and the system proposed here have one obvious feature in common: both systems represent locations as trees. However unlike Dπ in which all locations are under the control of the programmer, in the system presented here some locations correspond with mobile physical objects and are simply observed, not controlled. A major difference between the two systems is that Dπ programs are type-checked

at compile-time whereas MRPL policies are enforced dynamically at run-time against agents whose program code may not be available to the system for any up-front analysis.

KLAIM

KLAIM (A Kernel Language for Agents, Interaction and Mobility) [49] is a mobile-agent formalism based upon Linda. KLAIM supports multiple *located* tuple spaces and operators for building processes. A KLAIM program (also called a *net*) consists of a collection of *nodes*; each of which has a process component and a tuple space component. In KLAIM terminology, the concrete name of a node is a *site* and a *locality* is the symbolic name of a node. Localities are considered first-class data and can be dynamically created and communicated over the network. Programs may be written using the X-KLAIM [24] tools and compiled into Java for execution. Programs are written in terms of localities and are independent of the actual physical distribution of nodes. The net primitives address distribution and co-ordination issues, including the visibility of localities and the mappings of localities onto sites.

KLAIM uses a simple type system to statically enforce security properties [122]. Type-checking is a two phase process; the first phase deduces process intentions (e.g. read, write, intention to migrate etc.) with respect to the various localities they may interact with; the second phase checks whether each process actually has the necessary rights (granted by the administrator) to perform the intended operations.

KLAIM has some similarities to the system proposed here. Both KLAIM and the MRPL attempt to control access to localised resources and to prevent some process migrations (in KLAIM a migration would be blocked if the process had insufficient access to the intended migration target whereas here migrations would be blocked if found to violate MRPL policies). However KLAIM and the system proposed here have a number of philosophical differences. The KLAIM system analyses a fixed network of sites up-front in a compile-time type-checking phase whereas we take a more dynamic approach, enforcing policies at run-time. Rather than only consider networks of sites, MRPL policies are written also in terms of physical spaces and can react to the physical movements of people and

computers.

4.7.3 Middleware-based Approaches

This section provides an example of a piece of middleware which supports *object* migration. Object migration systems allow the state of an object to be moved from one host to another and typically have some request redirect mechanism whereby requests sent to the old object location get transparently rerouted to the new location. From the point of view of other objects interacting with the migrating object, nothing has happened save a slight pause as the object moved. Object migration systems can be used to implement mobile agents systems supporting weak mobility; i.e. those which require agents to assist in their own shutdown and restart post-move.

LocALE

The LocALE [47] (Location-Aware Lifecycle Environment) framework provides a CORBA [123]-based mechanism to control the life-cycle (i.e. creation and destruction) and location of software objects residing on a network. LocALE defines the notion of a *Location Domain* – a group of machines physically located in the same place. This aspect of the LocALE system could be represented easily in the spatial model presented in Chapter 3; a Location Domain would correspond to a set of **workstation** entities with a common parent entity.

The LocALE project addressed two primary concerns: (i) providing the underlying (CORBA-based) mechanism for causing objects to move seamlessly from one object server to another; and (ii) using a vision-based environmental sensing system (called TRIP [48]) to trigger objects to change location. The LocALE system made the simplifying assumption that everything operates within the same trust domain and hence no specific support for security was required. The system could use whatever CORBA security systems are available on the hosts (such as transport-level encryption via SSL) in much the same way as Obliq could use the underlying feature-set of the secure network object system. This thesis makes the different assumption (outlined in Chapter 1) that many mobile programs will not be trusted – or equivalently contain errors which cause them to act in an untrustworthy fashion – and hence we require mechanisms like the mobility restriction policies described in this chapter to remain in control.

4.8 Summary

This chapter used the spatial model introduced in Chapter 3 as the basis for creating *mobility restriction policies* which allow the movements of mobile agents to be controlled in an environment augmented with physical location sensors. The spatial model was extended to associate entities with access to resources (via the function $privs$ defined in Section 4.3) allowing the same policies to be used to govern both movement and resource access. Policies are intended to be written by individual users with potentially conflicting goals. A metapolicy system to resolve policy conflicts was presented in Section 4.5 and was demonstrated by means of a hypothetical example in Section 4.6.

This thesis argues that application-level security policies governing the mobility and interface behaviour of applications are an effective technique to prevent vulnerabilities which would otherwise plague future ubiquitous computing systems. Potential threats enabled by physical and logical mobility were outlined in Section 4.1 and the mobility restriction policy language was argued by means of a series of examples to be an effective mechanism to prevent this class of attacks. A prototype implementation of the system described here has been created and described in Chapter 7. This chapter concludes the study of the mobility aspect; the following two chapters focus on the *interface behaviour* of applications.

CHAPTER 5

Security Policy Description Language v 2

This thesis introduces new application-level security policy systems to govern both the *mobility* and *interfaces* of applications. Chapter 3 and Chapter 4 discussed the mobility aspect; this chapter and Chapter 6 focus on the interface aspect.

Many contemporary web-applications possess application-level security vulnerabilities. According to a ZD-NET survey [113] 30-40 percent of all e-commerce sites and according to Internet Security Systems (ISS) [95] 11 widely-deployed shopping carts are vulnerable to simple application-level attacks. These vulnerabilities may be classified into a small number of categories, including client-side modification, SQL attacks, Cross-Site Scripting (XSS) and Forceful Browsing attacks. Each type of vulnerability stems from an undocumented interface assumption; an assumption that a malicious user can violate often to great effect and which this thesis addresses in a systematic manner.

Unfortunately, although each vulnerability is simple in nature it is nevertheless difficult to fix all the errors scattered across a large codebase. A realistic application is likely to be constructed of multiple components, each written by different people in different languages. Additional difficulties arise when components have been bought and are only available in binary form, preventing any analysis of the

sourcecode.

This chapter introduces a language called SPDL-2 which allows security policy to be created to protect an entire web-application interface. SPDL-2 allows the specification of per-request and response validation and transformation rules which can protect applications from all of the vulnerabilities listed above with the exception of Forceful Browsing, the subject of the following chapter. SPDL-2 policies are kept separate from the main application codebase and can be enforced against applications even without access to the original sourcecode and irrespective of programming language(s) used. Using SPDL-2 application integrators and installers can proactively fix bugs in third-party components without waiting for the original vendors to produce a fix[1].

The structure of this Chapter is as follows: Section 5.1 describes the threat model used by Chapters 5 and 6. Section 5.2 describes the common security vulnerabilities addressed by SPDL-2: namely Client-side Modification, SQL Attacks and XSS. It is argued that these vulnerabilities are evidence of the existence of *forgotten assumptions* within the apparently simple interfaces exposed by these applications. Using the interface framework described in Section 2.3 the general-form of interface modification required to fix these problems is described in Section 5.3. The language SPDL-2 is introduced in Section 5.4 and the capabilities are demonstrated through a case study in Section 5.5. Section 5.6 finishes with a discussion of alternative security mechanisms.

5.1 Threat Model

This section summarises the threat model behind the interface security policies described here and in the following chapter. We begin by classifying and describing the people involved in the production, maintenance and use of an e-commerce system on the Internet:

component vendors who sell black-box components (e.g. an e-commerce shopping cart);

[1]Vendors have to produce fixes that work on all installations without needlessly breaking functionality. It is often easier to fix the bug locally because (i) some functionality may be unused and breaking it does not matter; and (ii) the application with the fix can be tested readily.

application integrators who create sites by connecting components together with custom application logic;

system administrators who configure and run machines in a corporate environment;

network operators who run the public network; and

users who access the application over HTTP.

We assume that the application integrators and system administrators work for the same company—the company who wish to run the application—and are therefore fully trusted. The component vendors are assumed to be semi-trusted third-parties who—although they cannot guarantee to write perfectly secure software—nevertheless do not deliberately introduce trojans into their products. The network operators and users are completely untrusted and it is assumed that they are prepared to act maliciously to gain any advantage by exploiting vulnerabilities in the application. The diagram in Figure 5.1 shows the structure of the system and highlights which group of users is responsible for each part. Note that we do not consider mobility in this description; we only consider interface behaviour. Note further that in the diagram, responsibility for the application-level policies has been split between the application integrators and the system integrators to reflect the fact that the policies are of interest to both groups: the system administrators who have traditionally been responsible for security and the application integrators whose application the policies are written specifically for.

We assume that transport-level security (e.g. SSL) is used to protect communications between the users and the servers and therefore that the network operators are not able to eavesdrop on traffic. We do not consider Denial-of-Service attacks where legitimate users are blocked from accessing the servers by floods of bogus network traffic. We assume the system administrators stay on top of the relevant OS and server software patches and manage their network firewall rules so that the server machines may not be exploited directly. We do not consider the security of the users' machines to be important; the worst case scenario is that an attacker takes over computers belonging to other people and acts on their behalf. We do not distinguish attacks coming this way from attacks coming directly from

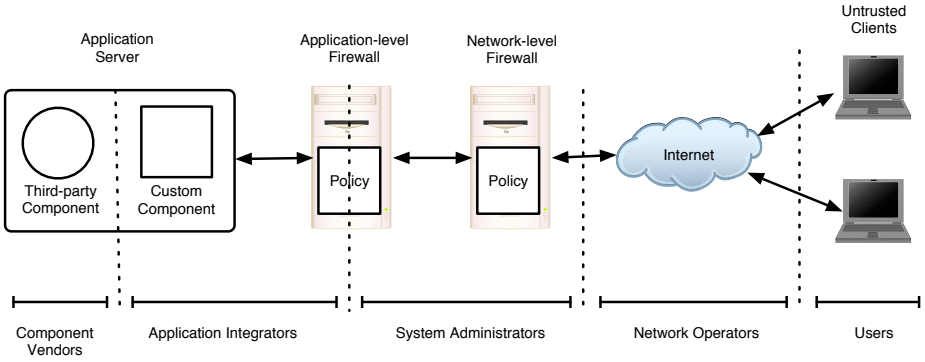


Figure 5.1: Block diagram showing the structure of a hypothetical system highlighting the users responsible for each component. Arrows indicate interface communication.

the attackers' own machines[2]. We assume that physical attacks are not possible because the servers are locked away. Therefore the remaining threats are due to:

- vulnerabilities in the third-party components; and
- vulnerabilities in the custom logic glueing the components together.

We assume that these application-level vulnerabilities can be used to subvert the application and leave it under the complete control of an attacker, even if all else in the system is secure. The two policy languages, SPDL-2 and SWIL described in this and the following chapter aim to address these remaining interface vulnerabilities. Note that policy enforcement is always done on the application-level firewall on the server-side. Some additional policy enforcement may be done on the client-side too—to save the cost of a round-trip to the server—but in all cases, policy is always checked on the server side *as well*.

5.2 Common Security Vulnerabilities

This section contains a set of example web-application security vulnerabilities found in many applications today. This set is not exhaustive; for a more exhaustive list the reader is referred to the Open Web-Application Security Project (OWASP)

[2]Of course the difference between these two cases is in the potential for *catching* those responsible; an attacker using someone else's machine is presumably much harder to catch than an attacker using their own machine.

"top 10" vulnerabilities list [9]. The examples presented here are intended to provide a flavour of the problem and motivate the subsequent discussion.

5.2.1 Client-side Modification

Section 2.4.5 described the need to store application state on the client-side to provide the illusion of a coherent session over the inherently stateless protocol HTTP. Application state can either be stored as a cookie or within a component on an HTML form. HTML forms contain several types of stateful components including textboxes, list selections, checkboxes, press buttons and even special *hidden* elements, designed specifically for storing client-side state invisibly. All of these may be changed. Many online e-commerce applications are vulnerable to so-called "hidden price field"' attacks [113] where users are able to modify the prices of goods stored in hidden fields on the client-side.

In addition to client-side state some applications use client-side *code* written in the language Javascript. Such code is commonly used to prevalidate user input by checking all the input fields have appropriate values before sending the data to the server; useful to avoid unnecessary and slow round-trips to the server. However, it is unsafe to rely on this code catching errors as it can also be changed by a malicious user, neutralising whichever checks it performs. Any application which does rely on client-side code may be vulnerable to attack.

Defence

Client-side Modification attacks happen when application-developers mistakenly assume that code and data on the client side will not be modified. These attacks can be defended against by carefully checking all data (even hidden data) on the server-side; the server should never rely on the honesty of the client.

5.2.2 SQL Attacks

SQL Attacks are sometimes known as "metacharacter attacks" or "quoting attacks". Web-applications often include database components. For example, in an e-commerce application a database would be used to store the shop catalogue, inventory and a list of customer orders. Database interfaces are often written in terms of Structured Query Language (SQL) statements which are dynamically interpreted by the database engine. Applications construct these SQL queries, trans-

mit them to the database which responds with a table of records which match the query. In a web-application the queries often involve data obtained from the user, for example the password supplied by the user to log into their account. Consider an application which contains code like the following:

```
$query =
"SELECT * FROM users WHERE username='"+$username+"'"+
                    " AND password='"+$password+"';";
```

where the variables `$username` and `$password` are bound to the user's username and password as entered on a login HTML form. Imagine a malicious user types their username as

```
Administrator'; \#
```

and leaves their password blank. After minor textual reformatting the SQL query constructed by the system will be

```
SELECT * FROM users WHERE username='Administrator';
# ' AND password = '';
```

In SQL the character # indicates the following data up to the end of line is a comment. Therefore this SQL query is equivalent to

```
SELECT * FROM users WHERE username='Administrator';
```

which will successfully retrieve the system administrator's user account from the database, without checking the password.

Defence

SQL attacks happen when developers fail to safely incorporate user data with SQL queries. They can be defended against by carefully separating all SQL metacharacters (", ' etc.) from user input through careful escaping or changes to the database interface (e.g. using precompiled SQL statements rather than relying on dynamic interpretation).

5.2.3 Cross-Site Scripting

Section 5.2.1 described how client-side Javascript code is often used to enhance applications by pre-validating form input thus avoiding a costly round-trip to the server to catch simple type errors. The Javascript security model is based upon the *same origin policy*. This policy states that Javascript code from a particular site can only access private state belonging to the same site. The intention is to prevent Javascript written by a malicious user from stealing another application's session state.

The web was intended to be an interactive medium which encourages everyone to publish their own information. This is exemplified by the proliferation of bulletin-board applications which invite users to post messages in various discussion threads. Unfortunately if a malicious user is able to publish a message containing Javascript code then when a normal user views the message the malicious Javascript will appear to have come from the bulletin-board site and the same origin policy will allow the Javascript to access the session state of the bulletin-board application.

The term *session hijacking* refers to a technique for impersonating another user by stealing their session information somehow. As discussed in Section 2.4.5, such session data is often stored on the client-side due to the stateless nature of HTTP.[3]

Once Javascript code can access the session information it can be trivially sent to an attacker, for example by fetching a URI with the secret information as a query parameter. This subversion of the same-origin policy is known as Cross-Site Scripting (XSS).

Note that XSS attacks can be used in conjunction with client-side modification attacks: consider an application which requires users to register before being granted access by entering personal details such as their full name and address. A typical HTML form for this purpose would contain separate HTML text boxes for forenames, surnames, title and address. Many sites impose length restrictions

[3]Some websites tie sessions to IP addresses, avoiding the need to store any additional client-side state (note the IP address is another form of client-side state). Unfortunately the use of web-proxies, Network Address Translation (NAT) and load-balancers render this technique almost useless in practice. Additionally, IP addresses (and therefore sessions) can be forged.

on fields like the title field; after all, most titles are quite short e.g. Mr. Mrs. Rev. Dr. Prof. etc. It seems superficially unlikely that much malicious Javascript code would physically fit in such a short field. Correspondingly a web-application designer might decide not to bother filtering that particular field. However, if the length restriction is only imposed on the client-side then it will be possible to modify the form, remove the restriction and use the changed form to upload a much larger than expected title field, containing the malicious code. If successful, anyone who views the malicious user's title on screen may also accidentally (and transparently) execute the malicious Javascript.

Defence

XSS attacks result from improper filtering of input data. Avoiding XSS attacks involves carefully filtering every piece of data submitted by a user removing all suspect code and HTML tags which might be displayed by the application.[4] The application server must remove all data received from a client A which, if sent to another client B, would be interpreted by client B's browser as something other than "normal" content.

5.2.4 Forgotten Assumptions: the root cause of vulnerabilities

The vulnerabilities described here are all caused by application developers making unwarranted assumptions about client behaviour. Within a single organisation it may be safe to assume that clients always behave in a predicable way but on the Internet where clients and servers normally run within different organisations a secure server should never trust a client or make any assumptions about its behaviour.

The assumptions made by developers in the above vulnerabilities fall into three groups:

1. query parameters (as entered on forms) are transmitted as text but often represent higher-level types (e.g. a credit card numbers) which need to be checked;

2. freeform textual data is often assumed to be free of special metacharacters

[4] A difficult task because the server must filter everything the client may interpret as a script. Many different browsers exist, each of which has its own set of bugs and undocumented features.

(like # in SQL);

3. data stored on the client-side (as cookies or hidden form fields) is modifiable.

A client-side modification attack (Section 5.2.1) is possible when developers either forget to check the types of form parameters (item 1) or forget that data on the client-side is modified, even if it has been marked as hidden (item 3). SQL attacks (Section 5.2.2) and XSS attacks (Section 5.2.3) occur when developers forget that users can enter special characters (item 2) – fragments of SQL or Javascript – which cause code to be executed either on the back-end server or third-party clients.

5.3 Formalising interface assumptions

Previously in Section 2.4.6 an interface exposed by a web-application was described consisting of a set of functions of the form:

$$URI(\texttt{in}\ request\ \texttt{req}, \texttt{out}\ response\ \texttt{res})\ \texttt{assume}\ \{\ \}\ \texttt{guarantee}\ \{\ \}$$

This interface is very generic, insisting only that clients send valid HTTP requests (represented by the argument of type $request$) and guaranteeing only that the application will generate valid HTTP responses (represented by the type $response$). The existence of the vulnerabilities described in Section 5.2 is evidence that many real applications have more complicated interfaces with additional and unchecked assumptions about client behaviour.

In this Section we make web-application interface assumptions explicit by creating a new interface which acts as an interface firewall, checking the assumptions before up-calling to the original interface. We define the following predicates:

$Typecheck(u, req)$ holds iff the parameters and cookies within request req are valid with respect to the SPDL-2 policy associated with URI u

$Statecheck(u, req)$ holds iff all MAC-protected client-side state within request req mentioned in the SPDL-2 policy associated with URI u is intact

$Escaped(u, req)$ holds iff all metacharacter escaping has been performed on the request req as specified by the SPDL-2 policy associated with URI u

$Protected(u, res)$ holds iff the client-side state mentioned in the SPDL-2 policy associated with URI u and stored within the HTML form in the response res has been protected by a MAC.

The generic interface for a web-application was displayed in Figure 2.14. The new interface with additional assumptions and guarantees is displayed in Figure 5.2. Each of the assumptions ($Typecheck(U_1, \texttt{req})$, $Statecheck(U_1, \texttt{req})$, $Escaped(U_1, \texttt{req})$ corresponds to an explicit check made by the application-level firewall. If any of the assumptions do not hold (i.e. the corresponding check fails) then the request is blocked and the application is protected. Similarly, the guarantee made about the response ($Protected(U_1, \texttt{res})$) corresponds to a transformation applied to the response – annotating client-side state with MACs – before returning to the caller.

5.3.1 Example

Consider part of an e-commerce application consisting of 2 URIs whose paths[5] are `viewdetails` and `commit`. The URI `viewdetails` displays details of a transaction, prompting the user for confirmation while the URI `commit` finalises the transaction by debiting the user's credit card and storing the order in a back-end database. The raw generic interface would have the following form:

```
interface Application{
  viewdetails(in request req, out response res)
  commit(in request req, out response res)
}
```

Imagine the application stores all session data on the client-side as hidden form fields. The form returned by `viewdetails` will look something like the following:

```
<form target="commit">
  <input type="hidden" name="productID" value="1234"/>
```

[5] A URI `http://www.example.com/view` has path `view`.

$$
\begin{array}{l}
\texttt{interface}\ I\ \{ \\
\quad \texttt{type}\ \mathit{cookies} = \mathit{string} \\
\quad \texttt{type}\ \mathit{uri} = \mathit{string} \\
\quad \texttt{type}\ \mathit{parameters} = (\mathit{string} \times \mathit{string})\mathit{list} \\
\\
\quad \texttt{type}\ \mathit{request} = \mathit{uri} \times \mathit{parameters} \times \mathit{cookies} \\
\\
\quad \texttt{type}\ \mathit{response} = \mathit{uri} \times \mathit{parameters} \times \mathit{cookies} \\
\\
\quad U_1(\texttt{in}\ \mathit{request}\ \texttt{req}, \texttt{out}\ \mathit{response}\ \texttt{res}) \\
\qquad \texttt{assume}\ \{\ \mathit{Typecheck}(U_1, \texttt{req}), \\
\qquad\quad \mathit{Statecheck}(U_1, \texttt{req}), \\
\qquad\quad \mathit{Escaped}(U_1, \texttt{req})\} \\
\qquad \texttt{guarantee}\ \{\mathit{Protected}(U_1, \texttt{res})\} \\
\quad \ldots \\
\quad U_n(\texttt{in}\ \mathit{request}\ \texttt{req}, \texttt{out}\ \mathit{response}\ \texttt{res}) \\
\qquad \texttt{assume}\ \{\ \mathit{Typecheck}(U_n, \texttt{req}), \\
\qquad\quad \mathit{Statecheck}(U_n, \texttt{req}), \\
\qquad\quad \mathit{Escaped}(U_n, \texttt{req})\} \\
\qquad \texttt{guarantee}\ \{\mathit{Protected}(U_n, \texttt{res})\} \\
\}
\end{array}
$$

Figure 5.2: A web-application firewall interface with URIs=$\{U_1 \ldots U_n\}$ using the formalism of Section 2.3.

```
  <input type="hidden" name="price" value="19.99"/>
  ...
  <input type="submit">Submit order</input>
</form>
```

The form `target` field indicates that the values contained within the form should be sent to the URI `commit` when the user presses the submit button. Notice the two hidden text fields; one stores the integer ID of the product being purchased and the other stores the price as a floating point number.

The application is potentially vulnerable in the following ways:

- it may not check the types of the `productID` and `price` fields, leading to undefined behaviour in the event the user changes the types of these values; and
- the `price` field (representing the price of an item in a shopping cart) is left on the client-side and may be modified by a user keen to get an unauthorised discount.

We may protect the application by creating an interface firewall with an interface like that shown in Figure 5.2 and producing suitable definitions for the predicates:

- $Typecheck(\texttt{commit}, \texttt{req})$,
- $Statecheck(\texttt{commit}, \texttt{req})$,
- $Escaped(\texttt{commit}, \texttt{req})$ and
- $Protected(\texttt{viewdetails}, \texttt{res})$.

The predicate $Protected(\texttt{viewdetails}, \texttt{res})$ should guarantee the hidden field data returned by the `viewdetails` URI is protected by a MAC. The $Typecheck(\texttt{commit}, \texttt{req})$ predicate should guarantee the types of the parameters: `productID` is an integer and `price` is a floating point number. $Statecheck(\texttt{commit}, \texttt{req})$ verifies the client-side state protected by a MAC (i.e. by the $Protected(\texttt{viewdetails}, \texttt{res})$ predicate) is still intact. This example does

not contain an SQL Attack (Section 5.2.2) or XSS (Section 5.2.3) vulnerability and so we require

$$\forall req.Escaped(\texttt{commit}, req) = \mathbf{T}$$

MACs allow servers to verify that state stored on the client-side (in the form of hidden fields) is returned unmodified. A well-written server would be expected to store well-typed data in such MAC-protected fields; returning to the previous example, one can see that the `productID` field was an integer while the `price` field was a floating-point number. However, it is possible to imagine a buggy server which accidentally generates badly-typed hidden form data. Such badly-typed data might lead to an application-level error when the data is returned unmodified to the server. Therefore it is still useful to typecheck all data, even though some is MAC-protected and cannot be modified by malicious users. Typechecking therefore serves two purposes: (i) it prevents users entering badly-typed data into form fields; and (ii) it prevents a buggy application from sending itself badly-typed data through hidden form fields.

The next section describes the language SPDL-2 – Security Policy Description Language version 2 – which is used to define concretely the request/response validation constraints and transformations represented by the predicates mentioned above.

5.4 SPDL-2 Overview

The language SPDL-2 (Security Policy Description Language version 2) allows the specification of both per-URI validation constraints and data transformation rules. The first version of SPDL was presented at the Eleventh International World Wide Web conference (WWW2002) [146] and had several deficiencies which have been addressed by the design of version 2. In particular SPDL-2

- allows policies to be defined hierarchically, factoring out common elements and leading to more readable specifications;
- allows fine-grained control over expensive HTML-modification operations specifically the use of Javascript (see Section 5.4.1) and MAC insertion (see Section 5.4.1); and

- uses the same language for both validation and transformation (see Section 5.3 expressions for consistency.

SPDL-2 descriptions may be written by hand or with the aid of automatic tools. Once written, policies can be compiled into executable code and dynamically loaded into an application-level firewall which sits between the Internet and the back-end web-server. The following sections describe the SPDL-2 language in detail.

5.4.1 Security Policy Description Language Version 2

At the top level, an SPDL-2 specification is an XML document (XML and HTML were briefly compared in Section 2.4.3). The full DTD for SPDL-2 may be found in Appendix A. The document consists of a single `<site>` element which in turn contains a collection of `<policy>` elements. Each `<policy>` element contains a group of related `<uri>` and `<cookie>` elements and optionally `<parameter>` and further nested `<policy>` elements.

For each `<uri>` element a number of `<parameter>` elements are declared. The attributes of a `<parameter>` element with attribute `name` $= p$ place constraints on data passed via URI parameter p:

- The `maxlength` and `minlength` attributes specify maximum and minimum length constraints.
- Setting `required` to "`Y`" specifies that p must always contain a (non-zero length) value;
- Setting `MAC` to "`Y`" specifies that the value of p must be accompanied by a Message Authentication Code (MAC) [143] generated by the server. This provides assurance of data integrity; i.e. the user is prevented from changing the value of the parameter to arbitrary values .
- The `type` attribute specifies the data-type of p (currently either `int`, `float`, `bool` or `string`).

The `method` attribute determines whether the specified constraints apply to p passed as a GET-parameter (i.e. a URI argument) or a POST-parameter (i.e. returned from a form).

For example, consider the following policy fragment:

```
<policy name="example" description="...">
   <uri prefix="http://www.example.com/foo">
      <parameter name="p1" maxlength="4"
                 type="int" required="Y"
                 MAC="N" />
      <parameter name="p2" method="POST"
                 maxlength="3" type="string" />
   </uri>
</policy>
```

This example specifies constraints on parameters passed to URIs with prefix "`http://www.example.com/foo`". The first `<parameter>` element defines constraints to be applied to a parameter named `p1` passed by either GET or POST; the second `<parameter>` element defines constraints to be applied to a POST parameter named `p2`. A larger example of a policy definition can be found in the case study of Section 5.5. The case study explicitly demonstrates how policies can be nested within each other in order to abstract common parameters (e.g. Session IDs etc.) from a group of related URIs.

The attributes of the `<parameter>` element are intended to cover the majority of validation constraints required in practice. However one can imagine some circumstances in which more flexible constraints are required; this is the purpose of the `<validation>` element. Within `<validation>` elements *validation expressions* may be written in a general purpose *validation language*. SPDL-2 uses a simple, call-by-value, applicative language which is essentially a simply-typed subset of Standard ML [117] to express validation expressions. Note that the exact details of the language are not important and a commercial version of the system may choose to replace ML with a more mainstream industrial language, like Java.

The abstract syntax of the validation language is displayed in Figure 5.3. All well-formed expressions have type `bool`; this restriction is enforced by a compile-time type-checking phase in which all badly typed expressions are rejected. If the expression evaluates at run-time to *true* then this signifies that the

$$
\begin{array}{rcll}
e & \leftarrow & x & \text{(variables)} \\
 & | & c & \text{(constants)} \\
 & | & f(e_1, \ldots, e_k) & \text{(function calls)} \\
 & | & \texttt{getparam}.c & \text{(value of GET parameters)} \\
 & | & \texttt{postparam}.c & \text{(value of POST parameters)} \\
 & | & \texttt{this} & \text{(value of this field)} \\
 & | & e_1 \ \langle \texttt{op} \rangle \ e_2 & \text{(binary infix operators)} \\
 & | & \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 & \text{(conditionals)} \\
 & | & \texttt{let}\ d \ldots d\ \texttt{in}\ e\ \texttt{end} & \text{(local declarations)} \\
 & & & \\
d & \leftarrow & \texttt{val}\ x : t = e & \text{(immutable bindings)} \\
 & | & \texttt{fun}\ f(x_1 : t, \ldots, x_k : t) : t = e & \text{(function definitions)} \\
 & & & \\
t & \leftarrow & \texttt{int} \mid \texttt{float} \mid \texttt{string} \mid \texttt{bool} & \text{(types)}
\end{array}
$$

Figure 5.3: The Abstract Syntax of the Validation Language.

parameter contains valid data. Invalid data should cause the validation expression to evaluate to *false*. Validation expressions are executed in an environment in which all HTTP GET parameters named `x` may be referenced as `getparam.x`, HTTP POST parameters named `x` as `postparam.x` and the value of the enclosing parameter is referred to by the special name `this`.

The SPDL-2 system contains a simple standard library of convenience functions including:

- Arithmetic operators `+`, `-`, `*` and `/` which can be applied to both integers and floating point values. String concatenation is represented by the infix operator `++`.
- Relational operators `lt`, `gt`, `le` and `ge` can be applied to integers, floating point numbers and strings (under the standard lexicographic ordering).
- The function `String.length(s)` returns the length in characters of string `s`.
- The function `String.format(s,regexp)` returns true iff `s` matches the form specified by regular expression, `regexp`.

- The function `String.mid(s,l,r)` returns the substring of `s` which starts at character `l` and finishes at character `r` inclusively. (Characters of `s` are numbered from 1).

- Functions are provided to convert between different types. For example, `String.fromInt(i)` returns the string representation of integer `i`.

- Function `isdefined(p)` takes a parameter (e.g. `getparam.p` or `postparam.p`) and returns a boolean indicating whether `p` is defined (i.e. has been passed to the URI in the HTTP request). Using an undefined parameter as an argument to any other function or operator leads to a dynamically generated error message.

Transformation rules for parameters and cookies are written in the same language as validation expressions but are often much simpler. The standard library contains a number of functions covering some of the common cases. For example, if it was necessary to remove all spaces from a parameter `p` and then SQL-escape the result the specification would look something like:

```
<transformation>
  let fun filter(s: string, char: string):string =
      let val first:string = String.mid(s, 1, 1)
          val rest:string  = String.mid(s, 2,
              String.length(s) - 1)
      in  if (fst = char) then filter(rest, char)
                else first ++ (filter(rest, char))
      end

  in  Transform.SQL( filter( this, " " ) )
  end
</transformation>
```

The standard library contains the following functions:

- The function `Transform.EscapeSingle(s)` returns a copy of the string `s` where all single quotes have been replaced with their HTML character encoding.

- The function `Transform.EscapeDouble(s)` returns a copy of the string `s` where all double quotes have been replaced with their HTML character encoding.

- The function `Transform.HTML(s)` returns a full HTML-encoding of string `s` where every meta-character has been replaced by an HTML character encoding.

- The function `Transform.HTMLpartial(s)` returns a copy of the string `s` where all meta-characters have been replaced by character encodings (like `Transform.HTML`) with the exception of those associated with a small number of allowed tags (including simple style tags, `<b>`, `<u>`, `<i>` and anchors of the form `<a href="..."> ... </a>`).

- The function `Transform.SQL(s)` returns a copy of the string `s` where all SQL meta-characters such as '`;`' are escaped.

HTML-encoding is a particularly important transformation since inadvertently forgetting to HTML-encode user-input leads directly to XSS vulnerabilities, described earlier in Section 5.2.3. In recognition of the importance of this transformation, the convention is that *all* parameters are HTML-encoded unless explicitly specified otherwise in the security policy. To turn off HTML-encoding one must set the `htmlencode` attribute of the `transformation` element to `N`. For example one may write:

```
<transformation htmlencode="N" /transformation>
```

The DTD in Appendix A specifies that policies within an SPDL-2 document consist of a series of `<uri>` and `<cookie>` elements. `<uri>` elements have already been discussed in detail; in a similar fashion, `<cookie>` elements allow designers to place validation constraints on cookies returned from clients' machines. Earlier in Section 2.4.2 the assumption was made that cookies are global across entire applications i.e. all cookies are returned to the application in every request. This allows MACs to be generated for all state together.

Client-side Form Validation

Whenever requested by the policy (through the policy attribute `javascript="Y"`), the firewall inserts Javascript code to perform client-side validation checks. (Recall that the insertion of Javascript is only intended to enhance usability – data is always rechecked by server-side code to avoid the kind of Client-side Modification attacks described in Section 5.2.1). The inserted code checks most of the SPDL-2 constraints: types, lengths and all custom constraints written in the validation language. The resulting program is inserted into the `onSubmit` attribute of a `<form>` tag (described earlier in Section 2.4.3) unless such an attribute is already present – in which case an error is generated.

Message Authentication Codes

Recall that SPDL-2 policies allow URI parameters to be marked indicating that they must only receive data which is accompanied by a *Message Authentication Code* (MAC) [143] to ensure the integrity of state stored on the client-side. This corresponds to the predicate $Protected$ described in Section 5.3.

The implementation (discussed further in Chapter 7) uses the HMAC [20] algorithm to generate MACs by securely combining the protected data with a secret key and a timestamp. In this way an attacker is unable to generate new MACs without first finding out the secret key. A major danger still facing this system is avoiding *replay attacks* [158] where clients replay messages already annotated with MACs in unexpected contexts. Two steps are taken to avoid such attacks:

1. A time-stamp is included in the MAC to guarantee message *freshness*.

2. Rather than generating separate MACs for each individual protected field, a single MAC is generated for all protected client-side state bundled together. This protects the integrity of the whole message, protecting against cut-and-splice attacks (in which MAC-annotated fields are swapped into other messages).

Despite these preventative measures, the responsibility for ensuring that replay attacks are not damaging ultimately rests with the security policy designer. For example, in the case study of Section 5.5 a MAC is generated for *both* the

`productID` *and* `Price` fields. Although users can replay such messages this results in multiple purchases of the same product for the *correct* price. The intention is that the MAC prevents the `Price` and `productID` being modified independently.

SPDL-2 requires that HTML pages fetched from URIs that are contained in a single `<policy>` block may only contain links to MAC-protected URIs[6] which are found in the same `<policy>` block (or in a nested `<policy>` block). By forcing the application's URIs to be partitioned in this way an important performance optimisation is facilitated which is discussed later in Chapter 7.

Server-side State

The previous section described how client-side state may be protected using a MAC generated from a secret key and a timestamp. The secret key used is a piece of state stored on the server-side (strictly-speaking the state is stored within the interface-transforming firewall, rather than the application process itself). One may wonder whether *all* state should be stored in this way, rather than transmitted to the client at all.

The primary difference between the state represented by the secret key and the state stored in the hidden form fields is that the secret key is considered to be a constant shared across a lot of independent user sessions, whereas the state found in hidden form fields is application and session-specific. Consider the example described earlier in Section 5.3.1 which had two hidden fields: `price` and `productID` representing the price of a good and a database key respectively. If two users are purchasing different goods from the application at the same time then they will have different `price` and `productID` fields in their respective requests but, *crucially*, the MACs will be generated with the *same* secret key.

A single shared secret key for MACs allows us to build a scalable implementation. A cluster of interface-transforming firewalls can be used, all pre-configured with the same key. Since the key is shared it will not matter that logically related interactions (e.g. those pertaining to the same session) could be handled by different firewall nodes. By contrast, if all application state (like the `price` and `productID`) were stored within the firewalls then this state would have to be

[6] A MAC-protected URI has at least one `<parameter>` with its `MAC` attribute set to 'Y'.

shared dynamically between the firewall nodes. Accessing the state would be a bottleneck, inhibiting the scalability of the system.

The timestamp stored within the MAC is intended to guarantee message *freshness* and to limit the scope for replay attacks. The timestamp can be implemented in one of two different ways. It may be used to contain

1. the exact time the MAC was generated (assuming clocks are synchronised across the firewalls); or

2. a message sequence number.

The former approach allows the firewalls to reject replayed requests that are more than a certain number of seconds old. This obviously does not prevent all replays but it has the advantage of scalability—no extra state is required in the firewalls. The second approach allows firewalls to reject all duplicate responses, completely preventing all replays but at the cost of requiring extra state in the firewalls. The implementation of the system described in Chapter 7 uses the former, stateless approach. It must be emphasised therefore that the primary function of the MACs is to prevent hidden fields changing separately (e.g. allowing a different `productID` for the same `price`) and not to prevent all replays.

5.5 Case Study

To illustrate the methodology presented in this Chapter a hypothetical e-commerce system is considered. The hypothetical application is first partitioned into groups of URIs (see Figure 5.4) corresponding to:

- *static HTML pages* including "welcome" pages, an "about" page and all static multimedia content (e.g. images, videos, music);

- a group of *dynamic pages* further subdivided into

  - an off-the-shelf 3rd party shopping cart component capable of credit card transactions;

  - an in-house PHP component.

To make a more realistic example let us assume that:

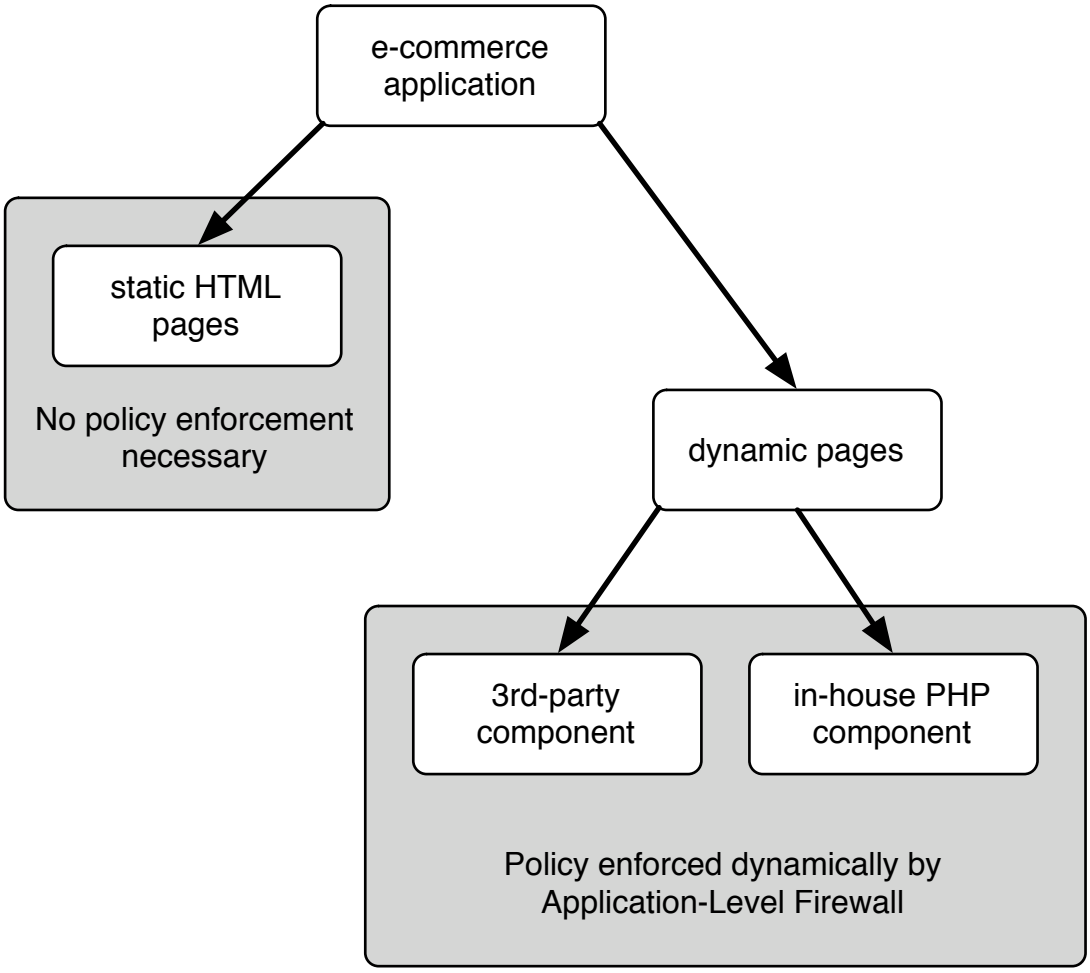


Figure 5.4: High-level structure of a simple e-Commerce web-site.

- both the in-house components and the 3rd party component are configured to set a cookie, `sessionKey`, to monitor user movement through the site for marketing research purposes;
- the off-the-shelf shopping cart component is supplied in a binary-only form and therefore cannot be modified.

Let us further assume that the site in question is vulnerable in the following ways:

1. The in-house PHP code (used to view an online shopping catalogue) has missing input validation code and can be used to execute arbitrary SQL against the back-end database using the attack described in Section 5.2.2.
2. The `sessionKey` cookie is predictable since it is created using a time-seeded random number generator; clients can spoof other active sessions by modifying the value of the cookie in their browser.
3. Javascript can be embedded in the `surname` field of the shopping cart login page which, when viewed on the company's intranet, leads to XSS vulnerabilities.
4. The Client-side Modification attack (described earlier in Section 5.2.1) can be used to reduce the price of items in the shopping cart component.

5.5.1 Designing the Security Policy

The static pages can be described simply with a single `<policy>` block containing a straightforward list of URIs; no processing is required for these pages.[7] The dynamic pages, on the other hand, necessitate a more complex policy description. In the example application described above, the dynamic pages may be subdivided into two sets of pages: those corresponding to the 3rd party component and the in-house PHP code. Each of these sets should be mapped onto its own `<policy>` block and then if either component is upgraded the policy changes required are limited to a single block. `<policy>` blocks may be nested and inner

[7] Note that these pages will also silently ignore the `sessionKey` cookie.

blocks inherit the parameter and cookie specifications from outer blocks. Since the `sessionKey` cookie is common to all the dynamic pages it is desirable to nest the policy specifications for the dynamic pages within a parent `<policy>` block which declares this shared cookie. The resulting SPDL-2 XML document has the following high-level structure:

```
<site name="e-Commerce website" ...>
  <policy name="Static pages" javascript="N">
    <uri prefix="About.html"   />
    <uri prefix="Contact.html" />
    ...
  </policy>
  <policy name="Dynamic pages" ...>
    <cookie name="sessionKey" MAC="Y" maxlength="16"
            type="int"/>

    <policy name="dynamic enforcement">
      <policy name="3rd party component">
        <uri prefix="CreditCard.asp" />
        ...
      </policy>
      <policy name="in-house PHP">
        <uri prefix="ViewShoppingCart.asp" />
        ...
      </policy>
    </policy>
    <policy name="static enforcement">
      <uri prefix="Login.asp"         />
      <uri prefix="Buy.asp"           />
      ...
    </policy>
  </policy
</site>
```

All that remains is to fill in the individual `<uri>` elements by writing the appropriate validation and transformation code. To see how this is done, consider the final step in the purchasing process in the shopping cart, a continuation of the example first used in Section 5.3.1. Imagine that users are sent an HTML form requesting their surname, credit-card number and its expiry date. The price and product-ID are stored in hidden form fields on the form. For example, when purchasing a product with `productID` $= 1234$, the form sent to the client might be as follows:

```
<form method="POST"  target="commit">

  <input type="hidden" name="productID"  value="1234">
  <input type="hidden" name="price"      value="19.11">

  <input type="text"   name="surname">
  <input type="text"   name="CCnumber">
  <input type="text"   name="expires">

 <input type="submit">Submit order</input>
</form>
```

Once purchases have been made, an order record is entered into the company's back-end database which can be subsequently viewed on their local intranet.

The SPDL-2 fragment corresponding to the form's target URI (`commit`) is presented in Appendix B. Each of the parameters shown in the form above are declared and a number of validation and transformation rules specified. A typical validation element looks like the following:

```
<parameter name="productID" method="POST"
           maxlength="10" minlength="1"
           required="Y" type="int" />
```

This fragment indicates that the field `productID` should be between 1 and 10 characters long, should always be present and should be a valid integer. Most of the rest of the SPDL-2 specification is self-explanatory although a few points are

worth noting. Firstly the `<validation>` element for the `price` field simply states that negative prices are not allowed. Secondly the more complicated validation expression for the `CCnumber` field is an implementation of the Luhn-formula commonly used as a simple validation check for credit-card numbers. Thirdly the validation expression for the `expires` field ensures that it is of the form *mm/yy* and also checks that the month is in the range 1–12.

Through repeating this process for all `<uri>` elements all of the system's vulnerabilities (described above) may be fixed without modifying any of the application code:

1. The form- and cookie-manipulation attacks can no longer be used since the `price` and `productID` fields along with the `sessionKey` cookie can all be protected by having their `MAC` attributes set to "Y".

2. The `surname` field can be HTML-encoded, preventing XSS attacks.

3. SQL attacks are prevented by applying the transformation, `SQLEncode` (see Section 5.4.1), to escape quotes in all relevant fields.

If specified in the SPDL-2 specification (by setting `javascript="Y"`), Javascript code will be inserted to check validation rules on the client-side. In this example Javascript is generated to ensure that credit-card numbers satisfy the Luhn-formula, that expiry dates are of the form *mm/yy*, that the `surname` field contains a non-zero-length value etc. Note that if extra validation constraints are required, they can simply be added once to the policy specification. Using conventional tools and techniques, the addition of extra validation constraints may require them to be coded multiple times (once in Javascript for client-side validation and certainly at least once in the web application's source code).

5.6 Related Work

Related work is divided into two categories. The first category comprises proposals for new methods for developing web-applications from scratch which attempt to abstract away many of the difficulties of traditional web-programming. The second category comprises tools for analysing and protecting *existing* applications. SPDL-2 falls into the second category.

5.6.1 New Application Frameworks

Graunke et al

Graunke et al [77] describe a model of web interactions in which each application consists of a set of URIs, each of which is a function which takes a request and produces a result containing a single form. This is the same model of web interactions presented earlier in Chapter 2. Graunke use this model to propose a type system for a new web-scripting language in which each form is associated with a record type and the server prevents the execution of any program which might generate a form which violates this typing. Their focus is on preventing run-time errors caused by programming errors, not on preventing run-time errors caused by malicious users modifying otherwise-correct forms. As a side-effect of their different focus they also purposely ignore issues arising from the use of client-side storage, which is an important part of the system proposed here.

BigWig

The `<bigwig>` project [1] consists of domain-specific languages and tools for the development of web services. A part of the `<bigwig>` project, *PowerForms* [28], allows constraints (expressed as regular expressions) to be attached to form fields. A compiler generates both client-side Javascript and code for server-side checks.

There are several key differences between `<bigwig>` and the system proposed here. Firstly `<bigwig>` is intended for the development of new web-applications whereas the system here is able to retrospectively attach security policy to existing applications. `<bigwig>` lacks a general-purpose validation language using instead a system of regular expressions. Validation constraints in `<bigwig>` and SPDL-2 can both be compiled to Javascript for client-side execution but `<bigwig>` only supports server-side validation when the application itself is written in `<bigwig>`. SPDL-2 policies can be applied no matter what language the application is written in.

WASH/CGI

WASH/CGI [163] is a system for building web-applications entirely in Haskell. It is structured into *sub-languages* each handling a different aspect of applica-

tion development. The document sub-language handles creating valid XHTML documents, the session sub-language provides a session abstraction, the widget sub-language allows forms to be typed (in a similar manner to Graunke at al [77]) and the persistence sub-language handles server-side and client-side storage. WASH/CGI therefore provides all the facilities needed to develop sophisticated applications and has no obvious security problems itself. However creating secure distributed application is still difficult (as argued in Chapter 2). It is still possible to compose together individually secure components to produce an application with vulnerabilities. For example it would still be possible to use WASH/CGI to produce an application which uses client-side state insecurely, like in the earlier e-commerce example of Section 5.3.1. Systems like SPDL-2 are therefore still useful even when such advanced tools are used to construct systems.

5.6.2 Dynamic approaches

Sanctum AppShield

Sanctum Inc. produce a product called AppShield [140] which, like the system described here, protects a system by interposing an interface firewall. However, despite this apparent similarity, there are significant differences between the two systems: we take the *programmatic approach* of specifying a security policy explicitly; in contrast AppShield has no policy description language or compiler and attempts to infer a security policy dynamically. Whilst this allows AppShield to be installed quickly, it limits the tasks it can perform. In particular, since there is no policy description language for describing validation or transformation rules, AppShield knows very little about what constitutes valid parameter values in HTTP-requests and can only perform simple checks on data returned from clients. AppShield is intended as a plug-and-play tool which provides a limited degree of protection for *existing* websites with application-level security problems. In contrast, we envisage our approach as also being useful in the design process of new applications i.e. when components are being composed together to make new systems.

WebScarab

WebScarab is a freely available Java-based application penetration-testing framework. It is complementary to the system described here: where SPDL-2 is intended to define the behaviour of an interface firewall to *protect* an application, WebScarab is intended to locate the errors in an application so they can be fixed. It contains an HTTP proxy, web-spider and a set of predefined application security tests. WebScarab is not intended to be fully-automatic (the developers say, "There is no shiny red button in WebScarab") but rather as a base on top of which developers can write application-specific tests. WebScarab is used for off-line analysis, not as a permanent application-protecting firewall like AppShield.

5.7 Summary

This chapter introduced SPDL-2, a language for writing expressive security policies governing web-application interfaces. SPDL-2 allows the specification of per-request and response validation constraints and transformation rules which protect web-applications from client-side modification attacks, SQL attacks and cross-site scripting (XSS) attacks. As justified by the extended case study, policies written in SPDL-2 are effective even when an application is constructed from multiple components, written in different languages and when components are only available in binary form. Implementation details are presented later in Chapter 7.

One attack, "forceful browsing" was mentioned briefly at the start of this chapter but not elaborated. This attack and its prevention are the subject of the following chapter.

CHAPTER 6

Stateful Web-application Interface Language

This chapter is the second and final chapter dealing with *interfaces* of applications. The previous chapter introduced the policy language SPDL-2 which allows stateless per-request and response validation and transformation rules to be specified for web-applications. SPDL-2 policies are able to protect applications from several common forms of application-level vulnerabilities, including client-side modification, SQL attacks and XSS. Mentioned briefly in the previous chapter, *forceful browsing* attacks involve submitting legitimate, well-typed[1] requests to an application in an unexpected order, causing the application state to become invalid. This chapter describes this class of attack in detail and presents a new policy language called the Stateful Web-application Interface Language (SWIL) designed specifically to thwart this kind of attack.

At the most basic level, SWIL describes the acceptable order of operation invocations as that accepted by a deterministic finite state automaton. SWIL is designed to be both easily translatable into PROMELA – the input language of the SPIN model-checker – as well as directly executable by an application-level firewall. Using SPIN, policies in SWIL are readily analysed before being installed, enabling policy designers to verify the system is free from important classes of

[1]The requests are well-typed in the sense that they pass SPDL-2 parameter validation checks.

error like deadlocks, livelocks and assertion violations.

Built on top of SWIL, the SWIL Meta-programming System (SMS) provides a high-level mechanism to automatically generate low-level SWIL programs. SMS consists of an embedding of SWIL in Objective-Caml (O'Caml)[2] [108] together with a library of useful functions which generate common application control sequences.

SWIL programs can do more than simply block erroneous HTTP requests; they can also be used to guide the actions of an HTML-based user interface transcoder. The transcoder is capable of removing all HTML elements (i.e. links and forms) whose normal use[3] would always violate the installed security policy. Finally, to demonstrate the complete SWIL-based system, a series of examples involving a hypothetical e-commerce site are presented and discussed.

The structure of this Chapter is as follows: Section 6.1 begins with a detailed discussion of forceful browsing attacks leading into an overview of the SWIL system contained in Section 6.2. Technical details of both SWIL and SMS are described in Section 6.3, followed by a description of the translation into PROMELA in Section 6.4 and the dynamic user interface transformation procedure in Section 6.5. The system is demonstrated by means of an extended example in Section 6.6. Related work is found in Section 6.7 while section 6.8 concludes this chapter.

6.1 Motivation

Earlier in Section 2.4.6 the interface for a web-application was described as a collection of URIs, each of which is represented as a function which takes a single input of type *request* (representing HTTP requests) and returns a single output of type *response* (representing HTTP responses):

$$URI(\texttt{in}\ request\ \texttt{req}, \texttt{out}\ response\ \texttt{res})\ \texttt{assume}\ \{\ \}$$
$$\texttt{guarantee}\ \{\ \}$$

[2]Standard ML (on which SPDL-2 validation expressions are based) and O'Caml are very similar, differing in minor syntax and library support.

[3]Normal use is discussed later but specifically excludes activities such as modifying application HTML in a text editor and generating custom HTTP requests.

No other assumptions (represented by the keyword `assume`) or guarantees (represented by the keyword `guarantee`) were documented in the default web interface description. Chapter 5 described how many assumptions about what constitutes valid input values typically remain undocumented in real applications. These assumptions often manifest themselves in the form of application-level vulnerabilities which an attacker can exploit. The policy language SPDL-2 was designed to allow the specification of stateless per-request and response validation and transformation rules and when enforced by a special application-level firewall (described later in Chapter 7) is able to prevent many of these common attacks.

SPDL-2 only deals with argument (parameter) validation. In the description in Chapter 5 nothing was said about the acceptable order of execution of the interface functions; rather it was assumed that a user could invoke the operations in an arbitrary order with arbitrary parameters, provided they satisfied the validation constraints.

Unfortunately some application designers *do* make assumptions about the acceptable order of function invocation. When a user invokes a function from the interface, the response typically contains an HTML document comprising links and forms pointing to further interface functions. Users *normally* click on these links to continue their session with the application. It is easy to forget that users can manually type in URIs at any time, open new windows or click on their browser's "back" button, potentially confusing the application[4]. Many current stateful applications attempt to detect when the client and the server get out of synchronisation and return an error message. However, as is the case with the other vulnerabilities discussed in Chapter 5 it is difficult to ensure this potential error case is handled properly everywhere inside a large application. An attacker need only find one vulnerability in the application to mount a *forceful-browsing* attack.

[4]It should be noted that a fully stateless application – one in which all application state is stored on the client-side – is less likely to be confused than an application which possesses mutable server-side state which cannot be rolled back to a previous configuration. It should also be recognised that not every application can be stateless e.g. a flight booking application will need to commit a transaction to a database at some point.

6.1.1 Forceful Browsing Examples

Imagine a hypothetical e-commerce website which has a three stage checkout procedure, displayed diagrammatically in Figure 6.1. The first stage (labelled "Step 1") displays the contents of the user's shopping cart and offers the user the chance to make a purchase. The next stage ("Step 2") asks the user to enter credit card details while the last stage ("Step 3") asks for a delivery address. The normal path through the application (from "Step 1" to "Step 2" to "Step 3") is illustrated with the solid arrows. The web-site designer intends that goods are only dispatched once both payment and delivery details are confirmed.

Imagine a scenario where a user manages to guess the URI pointing to the delivery address page. Such a user will be able to request the delivery address page without having filled in the credit card details first. This alternate path through the application (from "Step 1" to "Step 3" bypassing "Step 2") is illustrated in Figure 6.1 by the dashed arrow. If the application has been written securely then the unexpected request ("Step 3") will be rejected and the user forced to go back and enter their payment details. However, if the application developer has assumed that the only way to reach the delivery address page ("Step 3") is by pressing the button on the credit card page ("Step 2") and the code contains no explicit check then the application is vulnerable to a forceful-browsing attack. In this example the forceful-browsing attack may allow a malicious user to receive goods without paying for them.

It is worth noting that forceful browsing attacks are not always malicious. Many web-applications use client-side Javascript code to open new web-browser windows, a particularly common technique in the travel industry. Consider a hypothetical travel booking application which opens a new window displaying flight details every time a user performs a database search. Furthermore imagine the application designer intends that each search result is either accepted (by clicking on a "buy" button) or discarded (by closing the window) before any further searches are made. A possible sequence of events is shown diagrammatically in Figure 6.2, with time running horizontally from left to right. At the first stage ("Step 1") the user enters some search criteria and presses the button marked "search". The second stage ("Step 2") a new window has been opened containing the details of

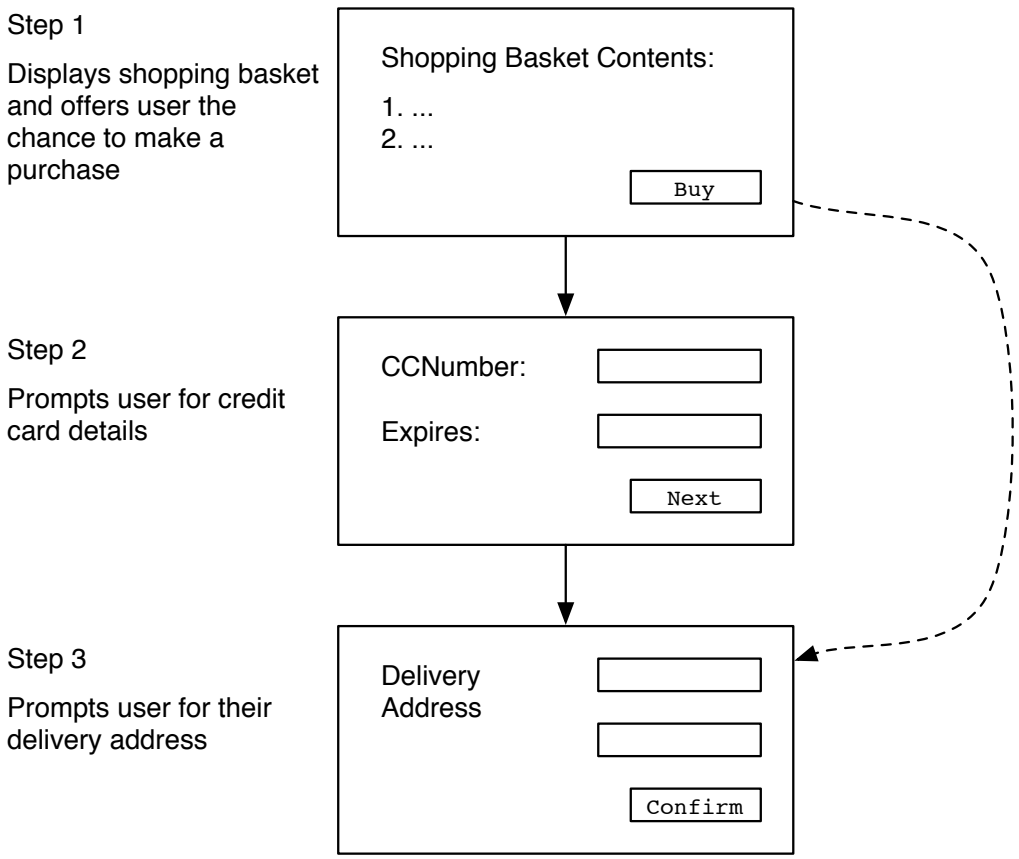


Figure 6.1: A hypothetical e-commerce website with a three stage checkout procedure. Each step has a form which points to the following step. The intended path through the procedure is displayed using the solid arrows. An alternate path, which may constitute a forceful browsing attack is displayed using a dashed arrow.

a flight (labelled “Flight Details 1”) alongside the original application window. Rather than discarding this window or accepting the flight, the user returns to the original application window and enters different search criteria. The results of the second query appear in another new window (labelled “Flight Details 2”) at the third stage (“Step 3”). Imagine the user prefers the first result to the second and so discards the window “Flight Details 2” and presses the button marked “Buy” on the window “Flight Details 1”. At the final stage (“Step 4”) the user has to confirm the transaction and pay for the flight. The question is: which flight has the application booked? From the point of view of the user the first flight has been booked. However if the application contained server-side state relating to the current search results then it might misinterpret the request and accidentally book the second flight. In effect, the application has suffered a forceful browsing attack simply because a user has clicked on *legitimate* links (i.e. those provided by an application) in an unexpected order – no guessing of URIs or other dubious behaviour was required on the user’s part.

Defence

Forceful-browsing attacks occur when developers forget that users can invoke interface functions at any time, with any arguments (provided they satisfy whatever parameter validation scheme is in force). They are a direct consequence of the mismatch between the stateful nature of many applications and the statelessness of the HTTP protocol, a problem exacerbated by the the flexibility of web-browsers (specifically the ability to open new windows, use bookmarks and type URIs at will). The attacks can be prevented by ensuring that the requests received from the user are consistent with the internal state of the application. In the three stage checkout procedure example above, the application should reject requests for the delivery address stage (“Step 3”) without first receiving the payment details (“Step 2”). In the travel-booking example, the application should *either* reject requests which correspond to stale search results i.e. once the window “Flight Details 2” is opened, all requests related to the window “Flight Details 1” should be rejected *or* it should use enough client-side state (protected as necessary with MACs as in Chapter 5) to book the flight the user actually chose.

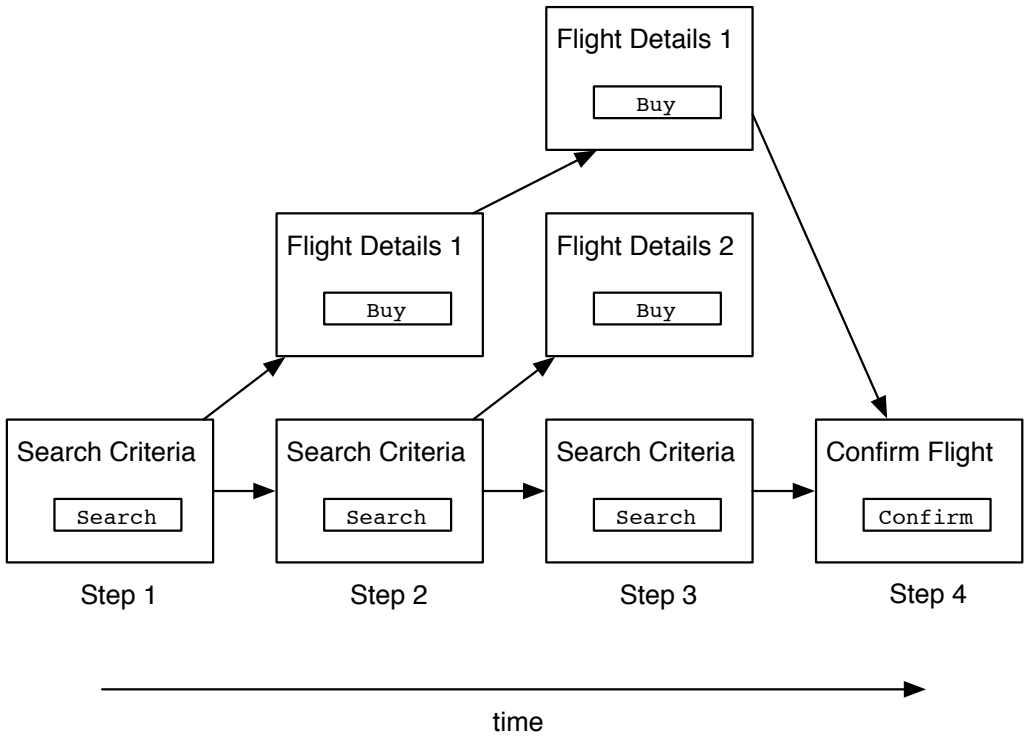


Figure 6.2: A sequence of interactions with a hypothetical travel-booking web-application. Time runs horizontally left to right. At each point in time a number of windows are open, displayed vertically.

6.2 Overview

This chapter introduces a form of web-application security policy based on Alfaro and Henzinger's Interface Automata [46] which allows developers to define the acceptable order of URI invocations as that accepted by a deterministic finite state automaton. A simple language called Stateful Web-application Interface Language (SWIL) for defining such automata is presented and the language is embedded in O'Caml — a variant of ML. The embedding facilitates *meta-programming*; high-level ML programs can be written which generate complete low-level SWIL programs. To demonstrate the usefulness of this technique, a small library of functions has been created each of which represents a common application control sequence. The functions in the library can be easily composed together to generate sophisticated SWIL programs.

SWIL has special-purpose features intended for web-applications interacting over HTTP: specifically an HTTP request type and manipulation functions. The language has been designed to be easy to translate directly into PROMELA, the language accepted by the SPIN model-checker as well as simple to execute dynamically on an application-level firewall.

A high-level overview of the system is presented graphically in Figure 6.3. After generation using the O'Caml embedding, SWIL policy is compiled first to PROMELA for off-line analysis with the SPIN model checker. PROMELA is used to verify a number of useful properties including: (i) the error state (encountered when an unexpected HTTP request is received) is unrecoverable; (ii) the automaton does not suffer from livelock or deadlock; and (iii) key steps (such as authentication) cannot be bypassed by forging a URI. Once the policy has been analysed in sufficient detail and the relevant properties verified it is compiled into an O'Caml program for dynamic execution on an application-level firewall, alongside any desired SPDL-2 policies.

Policies can be designed at different levels of granularity and installed in parallel. Such a compound policy ensures that only HTTP requests accepted by *all* policy automata will be allowed through to the application. Policies can be activated and deactivated at run-time in response to external demands. Finally, policies executed by the application-level firewall are not only passive monitors of application

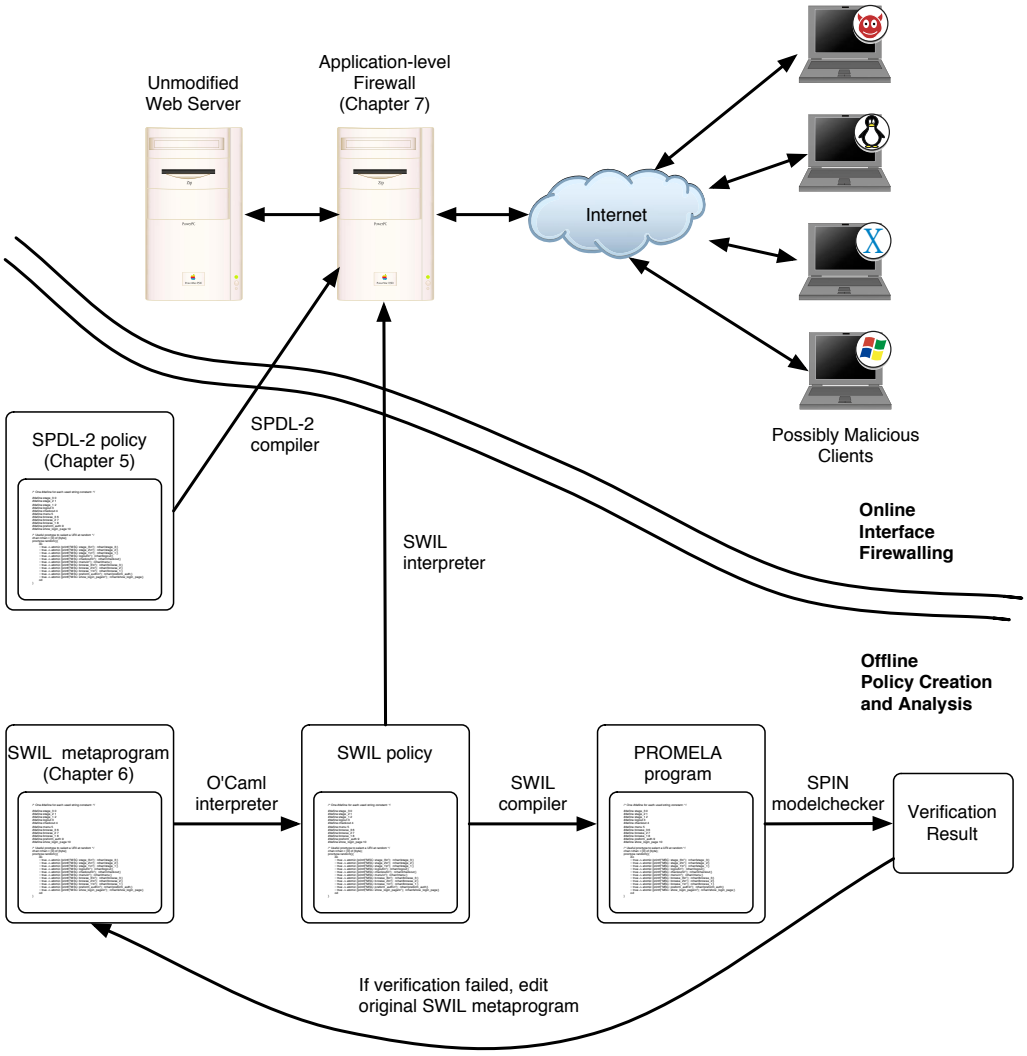


Figure 6.3: Overview of the system outlined in Chapters 5, 6 and 7. Note that the bottom part of the diagram describes offline policy creation and maintenance while the upper part shows the main application datapath.

behaviour but take a more active role. A strength of the design presented here is that policies may be used to dynamically transform the appearance of graphical user interfaces expressed in HTML, removing interface elements (links and forms) whose use correspond to transitions inevitably leading to future error states.

6.2.1 As Interface Modification

Recall that the interface exposed by a web-application was described in Section 2.4.6 as a set of functions of the form:

$$URI(\texttt{in}\ request\ \texttt{req}, \texttt{out}\ response\ \texttt{res})\ \texttt{assume}\ \{\ \}$$
$$\texttt{guarantee}\ \{\ \}$$

This interface is very generic, insisting only that clients send valid HTTP requests and guaranteeing only that the application will generate valid HTTP responses. Section 5.3 described how a number of common attacks resulted from hidden assumptions not documented in the interface and how the attacks could be prevented by programming a suitable interface firewall. This section shows how the forgotten assumption behind forceful browsing attacks – that users can invoke interface operations at any time – can be expressed using the same interface formalism.

Valid sequences of interface invocations are represented by a policy based on a finite state automaton. Transitions between states of the automaton are triggered by HTTP requests; the lack of a valid transition from a state indicates an error condition. Input HTTP requests are represented as elements drawn from a finite alphabet Σ using an *abstraction function* of type $request \rightarrow \Sigma$. The exact form of the alphabet and the abstraction function are policy-specific. More will be said about this later in Section 6.5.

Note that the input of the automaton is taken entirely from HTTP requests, ignoring HTTP responses. The reason for this is convenience; it is much easier to parse HTTP requests and extract useful information (e.g. the target URI, query parameters and cookies) than it is to parse and extract useful content from the HTML payload contained within the HTTP responses. However, it is possible to imagine a more complicated system which possesses a series of templates or XPath queries which are matched against the HTML payload data and the results of which are fed into the automaton, but for the sake of simplicity the remainder

of this presentation will ignore this possibility.

A policy is represented by an automaton (Q, q_0, A, T) where Q is a set of states, $q_0 \in Q$ is a special initial state, $A : request \rightarrow \Sigma$ is the abstraction function and $T : Q \times \Sigma \rightarrow Q$ is a partial function which returns the next state given the current state and the abstract representation of a request. For each combination of state and possible input there is either no successor state or one single successor state.

The policy can be imposed by an interface firewall in the style of Section 5.3 by first defining a number of types, functions and prolog-style predicates. First of all we require a mechanism to associate HTTP requests originating from the same session. There are several concrete techniques used in practice to achieve this but we represent the process abstractly by a function $getsession : request \rightarrow \mathbb{N}$ which maps individual HTTP requests onto sessions represented by natural numbers. Several possible methods to track user sessions were described earlier in Section 2.4.5 and a concrete implementation will be discussed later in Chapter 7. Each active session is associated with a mutable state value stored on the firewall which is updated as valid transitions occur. This updating is treated as a special side-effect of the predicate $UpdateState(s, q')$ which updates the state value of session s to value q'.

Note that the sessions and their associated policy automata are maintained on the firewall and not directly accessible by the user or their browser. A user only possesses an unforgeable reference to a session which is obtained when they first access the application and which is stored in a cookie. Users cannot clone their sessions, since the session state is not stored on their computer. They can copy the session reference by sharing their cookie with others with the effect that multiple browsers would interact with the same session at the same time. Note that this requirement to store per-session state is a major point of difference with the SPDL-2 system described in the previous chapter. In practice sessions are likely to be timed out after periods of inactivity and their state garbage-collected with the effect that subsequent requests from the same user will cause a fresh session to be created. More will be said about session management later in Section 7.2.3.

The full list of predicates used in the interface firewall definition are summarised below:

$SessionID(req, s)$ holds iff s is a valid session corresponding to the HTTP request object req i.e. $s = getsession(req)$

$CurrentState(s, q)$ holds iff q is the current automaton state corresponding to the session named by s

$Abstract(req, \sigma)$ holds iff $\sigma = A\ req$ i.e. $\sigma \in \Sigma$ is the abstract representation of the request req

$Valid(q, \sigma)$ holds iff $(q, \sigma) \in dom(T)$ i.e. a valid transition exists in the transition function[5]

$NextState(q, \sigma, q')$ holds iff $q' = T(q, \sigma)$ i.e. q' is the new state of the automaton

$UpdateState(s, q')$ holds always with the side-effect that the state of the automaton state corresponding to session s is updated to q'.

Figure 6.4 shows the general form of a firewall interface with URIs $\{ U_1, \ldots, U_n \}$ using the predicates defined above, necessary to protect applications from forceful browsing attacks. Two points are worth noting about the specification:

1. the identifiers $q, q,', \sigma$ are bound locally in each `assume` block; and
2. to avoid concurrency issues, each `assume` block is evaluated in isolation, to prevent two near-simultaneous requests reading the same initial state value q.

6.3 SWIL Technical Details

This section describes SWIL policies in detail. This section assumes knowledge of PROMELA and SPIN, first described back in Section 2.6. Section 6.3.1 describes the core language syntax and is followed by Section 6.3.2 which introduces the notion of a well-formed *program fragment* and *program*. Since SWIL is a low-level language, Section 6.3.3 introduces the high-level SWIL Meta-programming System (SMS) based on an O'Caml embedding which greatly eases the production of SWIL policies. SMS comes complete with a library of common control

[5]The automaton is deterministic as indicated by the fact that $NextState$ is a function.

```
interface I {
    type request = ...
    type response = ...

    U_1(in request req, out response res)
        assume {
            CurrentState(getsession(req), q),
            Abstract(req, σ), Valid(q, σ),
            NextState(q, σ, q'), UpdateState(getsession(req), q')
        }
        guarantee {}
    ...
    U_n(in request req, out response res)
        assume {
            CurrentState(getsession(req), q),
            Abstract(req, σ), Valid(q, σ),
            NextState(q, σ, q'), UpdateState(getsession(req), q')
        }
        guarantee {}
}
```

Figure 6.4: A web-application firewall interface with a URIs $\{ U_1, \ldots, U_n \}$ written using the formalism of Section 2.3. The identifiers q, q', σ are considered to be bound locally in each `assume` block.

sequence abstractions which may be easily combined together to create complex SWIL programs.

6.3.1 SWIL

The abstract syntax of the language SWIL is shown in Figure 6.5. The language is derived from PROMELA as used in the SPIN model checker with a few simple adjustments including (i) the removal of features in PROMELA allowing the expression of non-determinism; and (ii) the addition of built-in primitives for handling HTTP request messages.

The main syntactic categories are commands c and side-effect free expressions e. A program p consists of a set of global variable declarations d, a set of labelled command blocks b and an initial command. All variables are initialised with a constant literal (either an integer, string, boolean or a special null request) and further assignments are restricted to values of the same type. The infix binary operators `+`, `-`, `*` and `/` are defined on integers[6], `and` and `or` on booleans and the equality operator `=` is defined on all built-in types: integers, strings, booleans and requests.

The special command `x := NextRequest()` assigns to the variable `x` the next request sent by the user, directly corresponding to reading from a synchronous channel in PROMELA. The command `x := PeekRequest()` is similar except it does not *consume* the request; subsequent calls to `PeekRequest()` or `NextRequest()` will receive the same data again. These two commands are similar to PROMELA commands `c?x` and `c?<x>` described in Section 2.6.1. Request objects are also treated specially in SWIL. They can be assigned to other variables and introspected with the special `GetField` function. The expression `GetField(`var`, "x")` where var is an identifier containing data of type $request$ and `"x"` is a constant string returns the field named `x` from var. By convention we say that every request object has a field called `uri` which is the URI of the request. We also say that a request parameter called `y` can be accessed by `GetField(`var`, "param.y")`. Note that strings are constant and immutable and can only be compared against other strings. In the translation into PROMELA

[6]Note that should a divide-by-zero error occur, the program behaves as if the `error` command has been executed.

described later in Section 6.4, each unique string (including field names) is translated into a unique integer.

The command `x := native f(`$e_1, \ldots, e_k$`)` assigns to variable `x` the result of calling native function `f` with arguments given by expressions $e_1, \ldots, e_k$. The implementation of the function `f` must be written in every language targeted by the system; e.g. if targeting PROMELA then the implementation will be written as a PROMELA `proctype` and if targeting O'Caml then the implementation will be written as an O'Caml function. Typically a function implemented in PROMELA would employ non-determinism and be much simpler than the deterministic O'Caml equivalent. The `proctype` in PROMELA would likely be an abstraction of the corresponding code written in O'Caml.

To understand the usefulness of this mechanism, consider a web-application which requires users to authenticate before accessing the rest of the application. Checking the username and password could be performed by a call to a native function taking two strings, representing the username and password, and returning a boolean value, true if the credentials are valid and false otherwise. In a PROMELA model we may represent this by a non-deterministic choice: either the credentials match or they do not. In O'Caml we represent this as a deterministic concrete function which looks up the username and password in the accounts database.

The rest of the commands are: (i) assignments of the form `x :=` e which assign the result of evaluating expression e to variable `x`; (ii) sequencing with '`;`'; (iii) conditionals with `if then else`; (iv) `skip` which does nothing; (v) `error` which jumps to a built-in error state; and (vi) `goto` which jumps to another labelled block. The special error state jumped to by the `error` command is equivalent to jumping to the following program fragment:

```
error_state:
  skip;
  goto error_state;
```

Note that no further requests are read (via `PeekRequest()` or `NextRequest()`), preventing the application from processing any further HTTP requests. There is no way to recover from this state except deleting the

$$
\begin{array}{rcll}
const & \leftarrow & integer \mid string \mid boolean \mid null & \text{(constant literals)} \\
\\
e & \leftarrow & x & \text{(variables)} \\
 & \mid & const & \text{(constants)} \\
 & \mid & \texttt{not}(e) & \text{(logical negation)} \\
 & \mid & e_1 \, \langle \texttt{op} \rangle \, e_2 & \text{(binary infix operators)} \\
 & \mid & \texttt{GetField}(x, string) & \text{(request field access)} \\
\\
c & \leftarrow & x \texttt{ := } e & \text{(assignment)} \\
 & \mid & x \texttt{ := NextRequest()} & \text{(next)} \\
 & \mid & x \texttt{ := PeekRequest()} & \text{(peek)} \\
 & \mid & x \texttt{ := native } string(e_1, \ldots, e_k) & \text{(native)} \\
 & \mid & c_1 \texttt{ ; } c_2 & \text{(sequencing)} \\
 & \mid & \texttt{if } e \texttt{ then } c_1 \texttt{ else } c_2 & \text{(conditionals)} \\
 & \mid & \text{error} & \text{(error)} \\
 & \mid & \text{skip} & \text{(skip)} \\
 & \mid & \text{goto } label & \text{(goto)} \\
\\
b & \leftarrow & label\texttt{:}c & \text{(block)} \\
\\
d & \leftarrow & \texttt{val } x = e & \text{(variable declaration)} \\
\\
p & \leftarrow & \texttt{let } d_1 \ldots d_n \texttt{ and } b_1 \ldots b_n \texttt{ in } c & \text{(program)}
\end{array}
$$

Figure 6.5: The Abstract Syntax of the Language SWIL.

current automaton state and restarting from the beginning.

6.3.2 Well-formed Programs

Before describing a well-formed program we first introduce the concept of a well-formed *program fragment* where a program fragment is defined to be a pair of variable declarations and blocks:

$$type\ fragment = d\ list \times b\ list$$

A program fragment is said to be well-formed if the following constraints are met:

1. each variable is defined at most once;

2. each label is defined at most once;

3. every variable is associated with a single static type: one of integer, string, boolean or request;

4. every variable of type request (initialised to $null$) must be assigned to by a `NextRequest()` or `PeekRequest()` before a `GetField` in every possible execution path within every block; and

5. every call to a native function f must have the same number of arguments, same argument types and same result type.

A program consisting of a program fragment and a command is said to be well-formed if the fragment is well-formed and also:

1. each variable is defined exactly once;

2. each label is defined exactly once; and

3. each label referenced by a `goto` is defined.

Rather than use a set of well-formedness rules, an alternative approach would be to employ a type system to achieve the same effect. A type system could be created which would allow program fragments to be combined together if their associated typing environments are compatible (e.g. enforcing the constraint that each variable has at most one definition).

6.3.3 The SWIL Meta-programming System

SWIL is a simple low-level language which does not directly support sophisticated mechanisms for code-reuse. However rather than always writing programs entirely by hand it is possible to combine together existing fragments of SWIL programs using a meta-program in a higher-level programming language. This technique allows common interface patterns to be abstracted and easily shared across applications. To achieve this goal, the SWIL Meta-programming System consists of an embedding of the language SWIL in O'Caml allowing program fragments to be generated and composed together to create complex SWIL programs. SWIL syntax is available as a set of recursive disjoint union types and

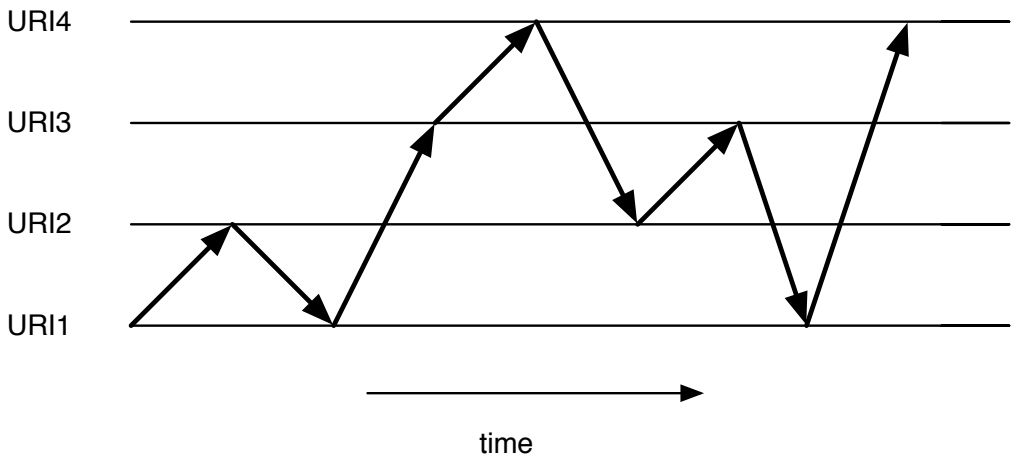


Figure 6.6: Graphical depiction of a browsing pattern allowed by the program fragment generated by the O'Caml sequence function. Four application URIs are represented vertically, time runs horizontally and the arrows indicate a path taken through the application by a user

.

there are types $label$, $fragment$ and $program$ corresponding to label names, well-formed fragments and programs respectively.

The utility of this mechanism is demonstrated by way of examples. The rest of this section describes two functions which can be used to generate SWIL program fragments. Each function is intended to capture the essence of (or 'abstract out') a common pattern of interaction with a web-application.

The first function, called sequence, has the following O'Caml signature:

$$\texttt{sequence} : uri\ list \rightarrow label \rightarrow label \rightarrow fragment$$

The function abstracts a sequential interaction pattern in which a list of URIs ($\{U_1 \ldots U_n\}$) represented by the first parameter of type $uri\ list$ must be visited essentially in order (i.e. U_1 before $U_2 \ldots$) but with limited support for backtracking. An example of the scheme is depicted graphically in Figure 6.6. In the figure time runs horizontally and 4 application URIs labelled URI1, URI2, URI3, URI4 are represented vertically. As time progresses, the user browses between the URIs in the sequence URI1, URI2, URI1, URI3, URI4, URI2, URI3, URI1, URI4 represented by the arrows.

The code generated by the `sequence` function enforces the following rule: If the user has visited at some point during the current run of the sequence the URI U_k where $k < n$ then revisiting any URI within the range $U_1 \ldots U_k$ is permitted. However any attempt to visit U_{k+1} before a visit to U_k will be blocked. To understand how this kind of interaction sequence might be useful, consider the e-commerce payment example first described in Section 6.1.1. The user has to fill in several pages of information (including payment and shipping details) before being able to complete a purchase. The security policy created by the `sequence` function will prevent the user from skipping the payment stage and going straight for the dispatch of goods, preventing a possible forceful browsing attack. However the policy generated by the `sequence` function still allows the user to backtrack (through either links explicitly included on the pages, entering URIs by hand, recalling bookmarks or using the browser back button) between already-visited pages in the payment procedure, useful to correct mistakes. The policy also blocks any attempt to fetch a URI not mentioned in the $uri\ list$ as this would also be ambiguous i.e. should the application cancel the transaction or simply postpone it?

The two parameters of type $label$ represent the entry and exit labels respectively. The sequence is activated by an explicit `goto` to the entry label and the fragment finishes by jumping to the exit label when the final URI is successfully called. When composing together program fragments in the policy creation system, the system verifies that the resulting program is well-formed (i.e. the labels match up correctly etc.) using the rules described above in Section 6.3.2.

In SMS evaluating the expression

```
sequence [ "one"; "two"; "three" ] "entry" "exit"
```

generates a program fragment like the following, where the two variables `tmp` and `req` are fresh:

```
entry:
  tmp := 0;
  goto loop;
loop:
  req := ReadNext;
```

```
if (GetField (req, "uri") = "one"){
  tmp := 1;
  goto loop;
} else { skip };

if (GetField (req, "uri") = "two"){
  if (tmp < 1) { error } else { skip };
  tmp := 2;
  goto loop;
} else { skip };

if (GetField (req, "uri") = "three"){
  if (tmp < 2) { error } else { skip };
  tmp := 3;
  goto exit;
} else { skip };
error
```

The implementation uses the integer counter variable `tmp` to store the highest URI index visited so far. The label `entry` marks the entrypoint to the fragment and the fragment is left either by the `error` command or a jump to the label `exit`. Attempts to access forbidden URIs invoke the `error` command. The diagram in Figure 6.7 displays graphically an automaton which corresponds to this program fragment. Each state (represented by a circle) is associated with a particular value of the variable `tmp`. State transitions (represented by the directed edges) are labelled with the URI expected. The special label `else` signifies a transition that occurs when a URI appears which is not associated with any other edge. In every state (except the final state where the sequence is exiting) there is an `else` label pointing to a special error state (in the program fragment this corresponds with the command `error`). Once inside the error state there is no transition out again[7].

The second O'Caml function, called `browse`, has the same signature as

[7] A user-friendly system might opt instead to allow the user to backtrack out of the error state.

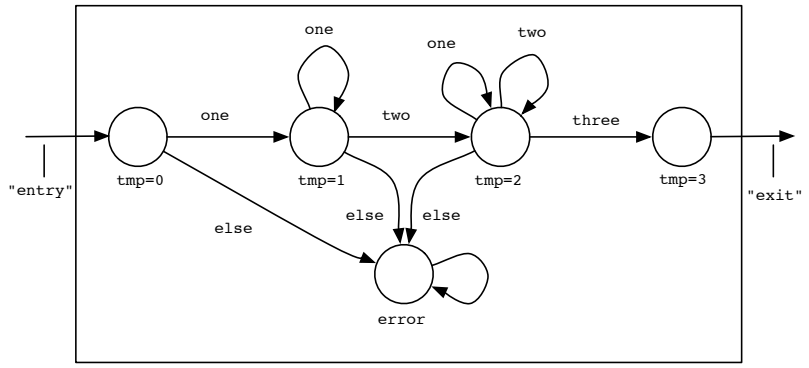


Figure 6.7: Automaton representing program fragment `sequence [ "one"; "two"; "three"] "entry" "exit"`. States are represented by circles and labelled arrows indicate state transitions where the label is the name of the consumed URI. Arrows without labels indicate a transition which consumes no URI. Two conventions are used to preserve determinism. Firstly we insist that all the labelled arrows from each individual state must have distinct labels and secondly we insist that if the source state has arrows with labels, these transitions are followed in preference before any unlabelled arrow transition.

`sequence` i.e.

$$\texttt{browse} : \mathit{uri\ list} \rightarrow \mathit{label} \rightarrow \mathit{label} \rightarrow \mathit{fragment}$$

This function represents a free-form browsing pattern in which the user is allowed to visit any URI from the list specified in the first parameter. Like the `sequence` example, the two arguments of type label specify entry and exit labels respectively. Control passes to the exit label if the user fetches any URI not in the set specified by the first argument of type $\mathit{uri\ list}$. The final URI is handled specially; rather than being consumed by the generated program fragment (as all URIs discussed so far have been) it is deliberately left unconsumed, available for the following program fragment.

The program fragment generated by evaluating the following expression in O'Caml:

```
browse [ "one"; "two" ] "entry" "exit"
```

looks like the following, where the variable `req` is fresh:

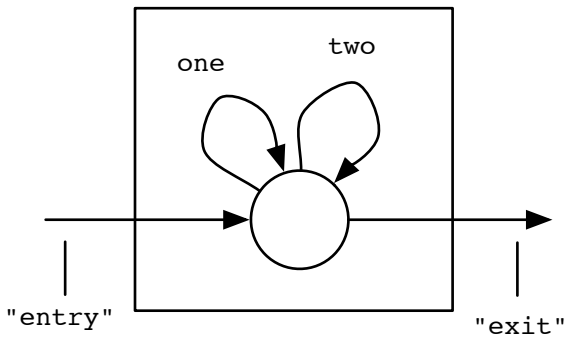


Figure 6.8: Automaton representing program fragment `browse [ "one"; "two"] "entry" "exit"`.

```
entry:
  req := PeekRequest();
  if (GetField (req, "uri") = "one"){
    req := ReadNext();
    goto entry;
  } else { skip };
  if (GetField (req, "uri") = "two"){
    req := ReadNext();
    goto entry;
  } else { skip };
  goto exit;
```

It operates by first peeking to see if the next request has a URI of either `one` or `two`. If it has, the request is discarded by calling `ReadNext()`. Otherwise the request is left and control passes to the label `exit`. The associated (and very simple) automaton is displayed graphically in the diagram within Figure 6.8. It has only a single state and all labelled edges (corresponding to the URIs within the URI list) point back to this single state.

6.4 Translation into PROMELA

This section describes how a SWIL policy may be translated into PROMELA for analysis by the SPIN model checker. An introduction to PROMELA and SPIN was given earlier in Section 2.6.1 and knowledge of both shall be assumed throughout

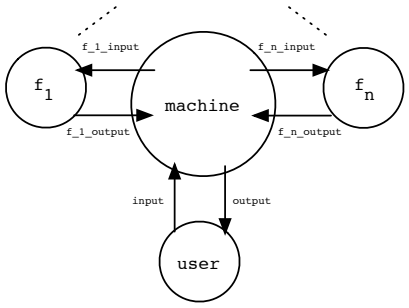


Figure 6.9: General structure of the translation of an automaton into PROMELA. Circles represent PROMELA `proctypes` while labelled arrows represent channels: the arrow indicates direction while the label gives the channel name.

this section.

Recall that a PROMELA model consists of a finite number of processes (defined through the `proctype` keyword) connected together by simply-typed channels with optional buffering. Each process body may be thought of as a set of labelled blocks with control-flow handled using the `goto` keyword. The features and syntax of SWIL are intentionally similar to PROMELA with only a few specific differences notably (i) SWIL policies are deterministic whereas PROMELA models may be non-deterministic; and (ii) models described here have support for HTTP request types and native functions, examples of the use of which will be described later in Section 6.6.

After translation into PROMELA the general structure of the automaton produced is depicted graphically in Figure 6.9. Each PROMELA `proctype` (process) is represented by a circle. Circles are connected by labelled arrows: the arrow gives the channel name and the arrow-head indicates the direction of flow of data; i.e. it points from sender to receiver. The bulk of the code (i.e. all the labelled blocks) is contained within the `proctype` named `machine`. The user is represented by an empty `proctype` labelled `user` with two channels: one for sending requests to the application and the other for receiving a boolean result: `true` indicating the request was accepted by the policy and `false` indicating the program has entered an error state. Each native function called by the automaton becomes an empty `proctype`, labelled $f_1 \ldots f_n$ in the diagram. Each of these native functions must be implemented, perhaps by a process which reads

its inputs, ignores them and non-deterministically generates a random output. The exact implementation depends on the nature of the analysis being performed, an example is provided later in Section 6.6.1.

The mechanical translation into PROMELA requires two passes. The first pass computes the following:

1. the types of all variables used in the program
2. the set of all request fields accessed by calls to `GetField`

$$allfields = \{field\_1, \ldots, field\_k\}$$

3. prototypes for all native functions ($f_1 \ldots f_n$)

$$\begin{aligned} allnatives \quad = \{ \quad & f_1(a_{11} : t_{11}, \ldots, a_{1q_1} : t_{1q_1}) : t_{1q_{return}}, \\ & \ldots, \\ & f_n(a_{n1} : t_{n1}, \ldots, a_{nq_n} : t_{nq_n}) : t_{nq_{return}} \quad \} \end{aligned}$$

4. a unique integer for each string used in the program

$$hash : string \rightarrow integer$$

Since PROMELA is conventionally fed through the C pre-processor a `#define` is created for each string e.g. the string "foo" may be represented by the following:

```
#define FOO 0
```

The request fields are computed because PROMELA HTTP requests are ***flattened*** into individual variables, one for each field. PROMELA does not support a string type; the function $hash$ maps constant strings onto simple integers (in the above example "foo" was mapped onto 0).

The second phase performs the actual translation via the set of recursive functions: $\mathcal{T}[\![-]\!], \mathcal{D}[\![-]\!], \mathcal{E}[\![-]\!], \mathcal{O}[\![-]\!], \mathcal{C}[\![-]\!], \mathcal{X}[\![-]\!], \mathcal{P}[\![-]\!]$ over types, declarations, expressions, operators, commands, native functions and programs respectively. The first five of these are displayed in Figure 6.10.

$$
\begin{array}{lll}
\mathcal{T}[\![integer]\!] & \rightarrow byte \\
\mathcal{T}[\![string]\!] & \rightarrow byte \\
\mathcal{T}[\![boolean]\!] & \rightarrow bool
\end{array}
$$

$$
\begin{array}{lll}
\mathcal{D}[\![\texttt{val } x = e : request]\!] & \rightarrow & \mathcal{T}[\![string]\!]\ x\_field\_1 = \mathcal{E}[\![\text{“”}]\!]; \\
 & & \ldots \\
 & & \mathcal{T}[\![string]\!]\ x\_field\_k = \mathcal{E}[\![\text{“”}]\!]; \\
\mathcal{D}[\![\texttt{val } x = e : t]\!] & \rightarrow & \mathcal{T}[\![t]\!]\ x = \mathcal{E}[\![e]\!]
\end{array}
$$

$$
\begin{array}{ll}
\mathcal{E}[\![x]\!] & \rightarrow x \\
\mathcal{E}[\![string\ x]\!] & \rightarrow hash(x) \\
\mathcal{E}[\![not(e)]\!] & \rightarrow \texttt{!}(\mathcal{E}[\![e]\!]) \\
\mathcal{E}[\![e_1\ op\ e_2]\!] & \rightarrow (\mathcal{E}[\![e_1]\!]\ \mathcal{O}[\![op]\!]\ \mathcal{E}[\![e_2]\!]) \\
\mathcal{E}[\![\texttt{GetField}(x, string\ y)]\!] & \rightarrow x\_y
\end{array}
$$

$$
\begin{array}{llllll}
\mathcal{O}[\![and]\!] & \rightarrow & \&\& & \mathcal{O}[\![or]\!] & \rightarrow & || \\
\mathcal{O}[\![+]\!] & \rightarrow & + & \mathcal{O}[\![-]\!] & \rightarrow & - \\
\mathcal{O}[\![<]\!] & \rightarrow & < & \mathcal{O}[\![>]\!] & \rightarrow & > \\
\mathcal{O}[\![=]\!] & \rightarrow & == & & &
\end{array}
$$

$\mathcal{C}[\![x$ `:=` $y : request]\!]$ $\rightarrow$ `x_field_1 := y_field_1; ...`

$\mathcal{C}[\![x$ `:=` $e]\!]$ $\rightarrow$ `x =` $\mathcal{E}[\![e]\!]$

$\mathcal{C}[\![x$ `:= NextRequest()]\!]` $\rightarrow$ `input?x_field_1, ..., x_field_k`

$\mathcal{C}[\![x$ `:= PeekRequest()]\!]` $\rightarrow$ `input?<x_field_1, ..., x_field_k>`

$\mathcal{C}[\![x$ `:= native` $f(e_1, \ldots, e_n)]\!]$ $\rightarrow$ `f_input!`$\mathcal{E}[\![e_1]\!]$`, ...,`$\mathcal{E}[\![e_n]\!]$`; f_output?x`

$\mathcal{C}[\![c_1\texttt{;}c_2]\!]$ $\rightarrow$ $\mathcal{C}[\![c_1]\!]\texttt{;}\mathcal{C}[\![c_2]\!]$

$\mathcal{C}[\![$`if` e `then` c_1 `else` $c_2]\!]$ $\rightarrow$

```
if
:: (E[[e]]) -> {C[[c1]]}
:: else -> {C[[c2]]}
fi
```

$\mathcal{C}[\![$`error`$]\!]$ $\rightarrow$ `goto error`

$\mathcal{C}[\![$`skip`$]\!]$ $\rightarrow$ `skip`

$\mathcal{C}[\![$`goto label`$]\!]$ $\rightarrow$ `goto label`

Figure 6.10: The PROMELA translation functions $\mathcal{T}[\![-]\!], \mathcal{D}[\![-]\!], \mathcal{E}[\![-]\!], \mathcal{O}[\![-]\!], \mathcal{C}[\![-]\!]$.

The first function $\mathcal{T}[\![-]\!]$ translates simple (i.e. not HTTP request) types (Figure 6.10). Simple PROMELA types were first described in Section 2.6.1. As previously noted, HTTP messages are translated into PROMELA by flattening them into individual fields. The first translation pass computed the complete set of HTTP request fields accessed throughout the program: $\mathit{allfields} = \{\mathit{field\_1}, \ldots, \mathit{field\_k}\}$. This information is used by the translation of variable declarations through the function $\mathcal{D}[\![-]\!]$ (Figure 6.10). Note that each field is assumed to have type string and defaults to the translation of the constant string "". Also note that a request called x with a field called f is transformed into x_f; and it is assumed that variable x_f does not already exist. Violations of this rule must cause a translation error. Expressions are translated straightforwardly into PROMELA via the function $\mathcal{E}[\![-]\!]$ (Figure 6.10) where the function $\mathcal{O}[\![-]\!]$ converts binary operators. Commands are translated via the function $\mathcal{C}[\![-]\!]$.

Notice in particular how variables of type request are decomposed into a set of variables, one for each individual possible field access. This affects both the assignment, next and peek rules. In the `NextRequest` rule the individual fields are read from a channel labelled `input` (displayed graphically in Figure 6.9) which is a unit-length buffered channel. The processes representing the user (`user`) and the policy (`machine`) are kept in lock-step using synchronous communication over the `output` channel. Calling a native function requires two operations: a marshalling step where the arguments are evaluated and sent over a request channel (for function f named `f_input`) and secondly a boolean result is read from a result channel (named `f_output`). The `error` command is implemented by jumping to a special block called `error` described later.

Labelled blocks are translated trivially via the function $\mathcal{B}[\![-]\!]$:

$$\mathcal{B}[\![\texttt{label:}c]\!] \;\rightarrow\; \texttt{label: } \mathcal{C}[\![c]\!]$$

Every native function is translated into a skeleton `proctype` via the function

$\mathcal{X}[\![-]\!]$:

$\mathcal{X}[\![$`native` $f(a_1, \ldots, a_n)]\!] \rightarrow$

```
    proctype f(chan input; chan output){
    loop:
        input?arg_1, ..., arg_n;
        ...
        output!true;
        goto loop;
    }
```

This skeleton does nothing except unmarshal its input arguments and return the single result `true`. When analysing a realistic system it is expected that this skeleton will be replaced with something more useful; an example of this will be described later in Section 6.6.1.

Finally the function $\mathcal{P}[\![-]\!]$ completes the translation and is displayed in Figure 6.11. This final translation mostly adds necessary boilerplate: the definition of the global `error` label, all the `proctype` and channel definitions and finally the `atomic` block which instantiates each `proctype` with the appropriate channels. The `init` block is a PROMELA idiom similar to the `main` function in the C programming language.

At this point a syntactically correct PROMELA model is available, albeit one which exhibits no interesting behaviour whatsoever; if executed the `machine proctype` would block on the first attempt to read a request from the `user proctype`. To make the system more useful the following steps must now be performed:

1. the `user proctype` should be completed with a model of user behaviour to analyse;

2. each native function `proctype` should be filled in with some logic specific to the application domain;

3. properties of interest should be devised and then analysed with SPIN.

```
P⟦let d_1,...,d_n and b_1,...,b_m in c⟧ →
    D⟦d_1⟧;...;D⟦d_n⟧;
    proctype user(chan output){
        do ::  true -> skip; od
    }
    proctype machine(chan input;
                     chan f_1_input; chan f_1_output;
                     ...
                     chan f_n_input; chan f_n_output){
        T⟦string⟧scratch_field_1, ...  scratch_field_k;
        C⟦c⟧
    error:
        output!false;
        input?scratch_field_1, ...  , scratch_field_k;
        goto error;
        B⟦b_1⟧...B⟦b_n⟧
    }
    X⟦f_1(a_11 : t_11,...,a_1q_1 : t_1q_1) : t_1q_return⟧
    ...
    X⟦f_n(a_n1 : t_n1,...,a_nq_n : t_na_n) : t_nq_return⟧
    chan input = [1] of {T⟦string⟧, ...,T⟦string⟧};
    chan f_1_input = [0] of {T⟦t_11⟧, ...,T⟦t_1q_1⟧};
    chan f_1_output = [0] of {T⟦t_1q_return⟧};
    ...
    chan f_n_input = [0] of {T⟦t_n1⟧, ...,T⟦t_nq_n⟧};
    chan f_n_output = [0] of {T⟦t_nq_return⟧};

    init{
        atomic{
            run f_1(f_1_input, f_1_output);
            ...
            run f_n(f_n_input, f_n_output);
            run user(input);
            run machine(input,
                        f_1_input, f_1_output,
                        ...,
                        f_n_input, f_n_output);
        }
    }
```

Figure 6.11: The definition of the function $\mathcal{P}[\![-]\!]$.

In many cases the model of the user will be completely non-deterministic i.e. the user can do anything. This allows us to check properties of the form "this can never happen, nomatter what the user does". Other models of the user can be written in order to check that "this is still possible if the user acts sensibly" i.e. the policy has not needlessly broken the application.

6.5 Dynamic Execution

SWIL programs are designed to be easy to translate into PROMELA for off-line analysis and easy to be interpreted by an O'Caml program running on an application-level firewall. The O'Caml translation is broadly similar to the PROMELA one; each (deterministic) PROMELA `proctype` simply becomes an O'Caml `Thread` communicating via `Event` channels (O'Caml's `Thread` and `Event` handling is based on Reppy's Concurrent-ML [133]). Since all non-determinism in SWIL programs must be contained within native function calls, only these need to be written manually in O'Caml. We make the additional restriction that the O'Caml implementations of the native functions are purely functional (i.e. free of side-effects). Purely functional native functions allow speculative execution by cloning the state and then feeding the automaton additional inputs to see how it behaves, without worrying about unintended side-effects. More will be said about this in the next section.

Program state is represented by the simple type

$$type\ \ state = (id, id\ expr)\ Hashtbl.t$$

where id is the type of identifiers, $id\ expr$ the type of expressions and $Hashtbl.t$ is a hashtable type constructor.

As with the PROMELA translation, user behaviour (`proctype user` in PROMELA) and all native functions need to be defined by hand. The thread running the user code looks like the following:

```
let rec user_thread () =
    let request = read_from_user () in
    let request' = make_request_object request in
    Event.sync(Event.send output request');
```

```
match (Event.sync (Event.receive input)) with
| OK -> begin
    send_to_webserver request;
    let response = read_from_webserver()
    in
    send_to_user (transform response);
    user_thread();
  end
| Error -> raise Error_Exception
```

The function `read_from_user` reads a raw HTTP request from the user and converts the result into a special request object – containing only those fields accessed by calls to `GetField` – before sending it on the channel `output` connected to the main program thread. This step corresponds directly to using the abstraction function mentioned back in Section 6.2.1.

The result (read from channel `input`) is either `OK` if the request was accepted or `Error` if the program entered the error state. If an error occurs an exception is thrown which halts the program. Otherwise, the request is forwarded to the back-end webserver via `send_to_webserver`. The response is read back through `read_from_webserver` and filtered through a special function `transform` (described in the following section) before being sent to the user via `send_to_user`.

6.5.1 Dynamic Response Transformation

Recall how an interaction between a user and a web-application consists of a series of HTTP requests and HTTP responses, requests containing parameters and responses containing HTML documents. Although the user can generate any HTTP request at any time (e.g. by recalling a bookmark or simply entering a URI by hand) the HTML within the response contains hints for further HTTP requests as links and forms. There is an unwritten contract between the application and the user which states that the user is allowed to invoke any of these operations; otherwise, why would they be present? Unfortunately if a SWIL policy is being imposed by a firewall then it is possible for the application to generate links and forms whose use would be rejected by the firewall.

In the generated O'Caml code described above in Section 6.5, the special function `transform` parses HTTP response messages containing HTML and filters all those forms and links which, if clicked on, would definitely be rejected by the installed security policy. The result is that the user sees a restricted view of the application, where links and forms are deleted if selecting them would certainly fail. As a side-effect, this makes SWIL policies useful for more than simply enforcing security properties; for example, parts of web-sites may be selectively disabled in a user-friendly manner by installing simple policies which block requests corresponding to those parts of the site.

A very simple conservative analysis can be used to determine whether a particular link or form should be removed, using the information known when the page is generated (i.e. before any extra information is filled in by the user). Links and forms were first described back in Section 2.4.6. In both cases they may be represented by a type

$$type\ \ link\ =\ form\ =\ uri \times parameters$$

where

$$type\ \ parameters\ =\ (string \times string)\ list$$

The primary difference between links and forms is the origin and purpose of the $parameters$. In the case of a link the parameters are all contained within the HTML and not displayed at all when the page is rendered. In normal operation (i.e. assuming the user is not examining the HTML in a text editor) the only opportunity the user gets to see the parameters is when the user clicks on the link. When a link is clicked, a web-browser typically copies the link to the URI bar [8] as the new page is loaded. Like links, forms contain some parameters which are invisible and not intended to be edited (the "hidden" form elements) but they also contain parameters which the user is supposed to modify, through text input controls. Therefore when a page is being transformed by the application-level firewall

[8] Some web-browsers allow the URI bar to be hidden from view using Javascript code. Sometimes web-browsers are embedded into other applications as components without the URI bar. For simplicity we ignore these cases.

each link and form may be considered to be an instance of type

$$uri \times (string \times (string\ option))list$$

where the parameters are key-value pairs, the keys are of type $string$ while the values are of type $string\ option$ where the value $None$ indicates the user is supposed to fill this value in and the value $Some\ x$ indicates the value is known already.

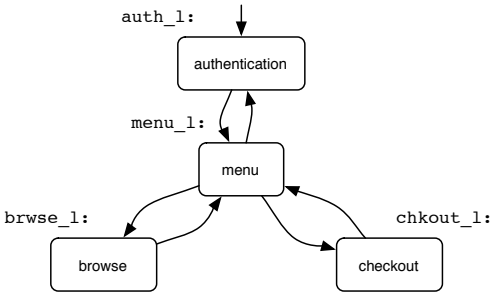
The analysis is performed by duplicating the current state of the firewall and speculatively executing the program as if the request had been sent as-is by the user. The speculation aborts immediately if a `GetField` command attempts to access any part of the request which is unknown (i.e. a $None$) value. Therefore the result of the speculation can be any one of three possibilities: (i) the request was accepted; (ii) the request was rejected; or (iii) insufficient information is known to decide between (i) and (ii). In the case of both (i) and (iii) the link or form is left intact; only if the request was definitely rejected is the link or form deleted from the page.

It should be noted that removing links or forms using this technique might lead to user confusion if some of the remaining text or images on the page refer to the deleted elements. If this happens then either the technique should not be used or a more application-specific user interface transform – which also removes these remaining problematic elements – should be applied instead.

6.6 Example

In this section a simple e-commerce application is described and a fine-grained SWIL policy is built which represents the acceptable order of URI invocations within the application. The application has the following characteristics:

1. users are required to login at the very beginning;
2. once authenticated, users are presented with a menu of further options;
3. users can choose to browse around the site adding items to their shopping cart; and
4. when finished shopping, there is a 3-step checkout scheme, similar to that mentioned back in Section 6.1.1.



Module	URIs
authentication	`show_login`
	`do_auth`
	`logout`
menu	`menu`
checkout	`checkout`
	`checkout{1,2,3}`
browse	`browse`
	`browse{1,2,3}`

Figure 6.12: General structure of the e-commerce application described in Section 6.6. [*Left*]: the application consists of 4 modules displayed along with entrypoint labels. [*Right*]: table lists the URIs which logically belong in each module.

The general structure of the application is described in Figure 6.12. The leftmost part of the figure shows how the application may be divided into four logical modules, labelled "authentication", "menu", "browse" and "checkout". Each module will correspond to a SWIL program fragment with entrypoint labels `auth_l`, `menu_l`, `brwse_l` and `chkout_l` respectively. The rightmost part of the figure contains a table which lists the URIs contained within each module. The syntax `checkout{1,2,3}` is used as a compact way of representing the three strings `checkout1`, `checkout2` and `checkout3`. Note that, as explained in Section 2.4 URIs are of the form `scheme://authority/path?query`; for brevity single words are used for URIs in the examples.

The first application module, authentication, involves three URIs: `show_login`, `do_auth` and `logout`. For the user to log into the application, the first two of these URIs are required: the first to return an HTML login page to the user and the second to process the user-supplied username and password. This procedure can be modelled with a SWIL fragment like the following:

```
auth_l:
  x := ReadNext;
  if (GetField(x, "uri") <> "show_login")
    then { error } else { skip };
  x := ReadNext;
  if (GetField(x, "uri") <> "do_auth")
```

```
    then { error } else { skip };
  t := native("check_authentication",
              GetField(x, "param.username"),
              GetField(x, "param.password"));
  if (t) { goto menu_l; } else { error }
```

This program fragment expects two requests in order with URIs `show_login` and `do_auth` respectively. The first corresponds to the user fetching the page with the login window while the second is fetched when the user enters their username and password and presses the login button. The second request assumes the existence of parameters `username` and `password` which are sent to the native function `check_authentication`. If this native function returns false then the application enters the error state, otherwise it jumps to the label `menu_l`, representing the menu module.

The menu module is very simple. It presents a list of options to the user and can be modelled with the following short program fragment:

```
menu:
  x := ReadNext; uri := GetField(x, "uri");
  if (uri = "menu") then { skip } else { error };

  x := ReadNext; uri := GetField(x, "uri");
  if (uri = "logout") then { goto auth_l }
      else { skip };
  if (uri = "checkout") then { goto checkout_l }
      else { skip };
  if ( (uri = "browse1") or
       (uri = "browse2") or
       (uri = "browse3") ) then { goto brwse_l }
      else { skip };
  error
```

The user is first expected to request the `menu` URI which displays the menu HTML itself. Afterwards if the user fetches the URI `logout` then control jumps to the label `auth_l`, the authentication module. If the user fetches the URI

`checkout` then control jumps to the label `checkout_l` in the checkout module. The last option is to browse the product catalogue; if the user requests any of the URIs `browse_1`, `browse_2` or `browse_3` then control jumps to the label `browse_l` in the browse module.

The browse module, corresponding to label `brwse_l` is easily handled using the `browse` SWIL generating function described earlier in Section 6.3.3. Unlike the specific example displayed graphically in Figure 6.8, this application allows the user to browse around pages named by URIs `browse_1`, `browse_2` and `browse_3`. In the meta-programming system the expression:

```
browse [ "browse_1"; "browse_2"; "browse_3" ]
       "brwse_l" "menu_l"
```

generates a program fragment with entrypoint `brwse_l` which allows the user to move between the URIs `browse_1`, `browse_2`, `browse_3` at random. When the user requests a different URI, control jumps to the label `menu_l` defined above.

The final module of the application is the checkout module which can be handled by the program fragment generated by evaluating the following expression in the meta-programming system:

```
sequence [ "checkout_1"; "checkout_2"; "checkout_3" ]
         "checkout_l" "menu_l"
```

This expression evaluates to a program fragment which has a similar form to the example in Figure 6.7, with different URIs and label names. A detailed diagram of the resulting automaton can be seen in Figure 6.13. The states in the diagram (represented by circles) are divided into 4 groups corresponding to the 4 modules of the system. The start state is highlighted with a double circle. Each transition is signified by a labelled edge, the label being the URI expected. The state marked with the cross represents the call to the native function `check_authentication`. Not shown on the diagram is the error state which the machine jumps to if it receives an unexpected URI or if the `check_authentication` function returns `false`.

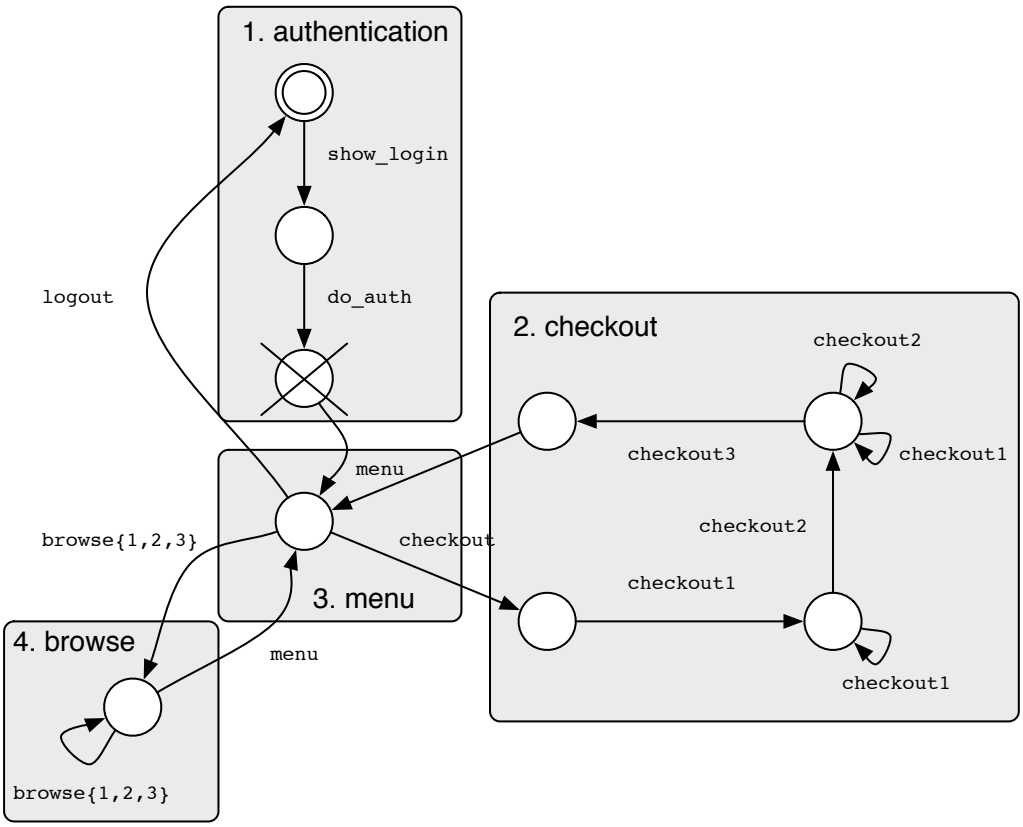


Figure 6.13: Graphical depiction of the automaton described in Section 6.6. Each state is represented by a circle. The start state is marked by a double circle. The state with the cross represents the call to the native function `check_authentication`. For clarity the error state is not represented on the diagram.

6.6.1 Analysis with SPIN

The translation of the SWIL program into PROMELA results in a skeleton; leaving the processes `check_authentication` and `user` undefined. The function `check_authentication` represents the process for determining whether a particular username and password is valid while `user` represents the behaviour of a user interacting with the system. This section describes how interesting properties of the automaton can be mechanically verified using the SPIN model checker by implementing these `proctypes` and writing some auxiliary code.

Recall how the PROMELA translation in Section 6.4 flattened requests into a set of variables, one for each distinct request field access via calls to `GetField`. In the following sections requests are therefore triples of `username`, `password` and `uri`. Each of these is a string in SWIL mapped onto byte constants in PROMELA ($\mathcal{T}[\![string]\!] \rightarrow byte$). Although URIs within the program become unique integers, a series of `#define`s are created allowing the use of symbolic names throughout the examples in this section.

For convenience, a special `proctype` and channel are introduced using the following code:

```
chan rchan = [0] of {byte};
proctype random(){
        do
        :: true -> rchan!menu;
        :: true -> rchan!logout;
        ...
        od;
}
```

This process runs forever in a loop, non-deterministically selecting a URI (represented by symbolic names `menu`, `logout`, ...) and sending it on the channel `rchan`. This can then be used as the basis for a general model of a user, capable of invoking interface functions at random.

1. Error state is non-recoverable and absence of livelock

This section describes how to simultaneously verify two desirable properties: the inability to recover from the error state and the absence of livelock. Entering the error state is intended to be a non-recoverable operation that happens when the user sends a request not anticipated by the policy. The error indicates a mismatch between the user's model of the application and the policy designer's model of the application. For safety it is important that the user session is not continued. Livelock occurs in the shopping application example if the automaton enters an infinite loop ignoring all requests sent by the user. The consequences of livelock are severe: from the point of view of the user the application would appear to halt while in fact the automaton would waste all available CPU-cycles on the application-level firewall.

To check for livelock with SPIN it is first necessary to mark a set of program states as *progress* states where infinite loops are only allowed if they go through at least one of these special states. Figure 6.6.1 shows PROMELA code representing a non-deterministic user process in which the first state in the main loop of the `user proctype` is marked as a progress state. Note that the variable `uri` holds a single byte value representing a constant string in the original SWIL program (recall that in SWIL all strings are static and constant). This `proctype` loops forever sending random URIs (read from the channel `rchan`) on its output channel. The username and password are left as `0` and subsequently ignored by the `check_authentication proctype` described below. The variable `result` keeps track of whether the `machine` has entered an error state. If it has then all future results (read from the `input` channel) are expected to be `false`, since it should not be possible to recover from the error state. If the `machine` is able to recover from an error state then this process will be able to halt in an *invalid end-state* (in SPIN terminology) i.e. one blocked attempting to read a `false` value from the `input` channel when it contains a `true` value. Invalid end-states were first described in Section 2.6.3.

The `check_authentication proctype` is written as follows:

```
proctype check_authentication(chan input; chan output){
        byte username, password;
```

```
proctype user(chan input; chan output){
        bool result = true;
        byte uri;
progress_loop:
        if
        :: result -> input?result;
        :: else -> input?false;
        fi;
        rchan?uri;
        output!0, 0, uri;
        goto progress_loop;
}
```

Figure 6.14: A PROMELA representation of a user which loops forever reading random URIs and writing them to the output channel. The main loop is marked as a *progress* state indicating that infinite loops are only allowed if they pass through this state.

```
loop:
        input?username,password;
        if
        :: true -> output!true;
        :: true -> output!false;
        fi;
        goto loop;
}
```

This `proctype` loops forever reading usernames and passwords, ignoring them and non-deterministically choosing whether the authentication succeeded or failed.

Once these `proctype`s have been defined the two properties can be verified by SPIN by

1. checking for the absence of invalid end-states which confirms the special SWIL error state is non-recoverable (i.e. nomatter what actions the user takes and whether or not they ever properly authenticate); and

2. checking for the absence of non-progress cycles (non-progress cycles were

first described in Section 2.6.3) which confirms the policy never gets stuck in a loop, ignoring user input.

2. Authentication cannot be bypassed

It is useful to verify that users must properly authenticate (i.e. have a valid username and password) before being able to use the application. Put another way we desire that users who do not properly authenticate cannot do anything other than attempt to log in.

The first step is to write the `check_authentication proctype` as follows:

```
proctype check_authentication(chan input; chan output){
  do
  :: input?username,password -> { output!false; }
  od;
}
```

This process reads in a username and password and always outputs the value `false`, as if the username and password are always wrong. The second step is to write the `user proctype` to represent possible user behaviour. The `user proctype` may be written as follows, where `SHOW_LOGIN`, `DO_AUTH`, `CHECKOUT`, `CHECKOUT2`, `CHECKOUT3` correspond to the strings "show_login", "do_auth", "checkout", "checkout2", "checkout3" mapped to unique integers using `#defines` by the PROMELA translation described earlier in Section 6.4:

```
bool failure = false;
proctype user(chan output; chan input){
  bool scratch;
  input?scratch;
  do
  :: true -> output!0,0, SHOW_LOGIN; input?scratch;
  :: true -> output!0,0, DO_AUTH;    input?scratch;
  ...
  :: true -> output!0,0, CHECKOUT;   input?false;
  :: true -> output!0,0, CHECKOUT2;  input?false;
```

```
  :: true -> output!0,0, CHECKOUT3;  input?false;
  od;
}
```

The two `0` parameters represent the username and password which are totally ignored by the `check_authentication` process. The input channel returns a boolean value: `true` if the request is valid and `false` otherwise. Note how this `proctype` uses non-determinism to model every possible sequence of URIs sent to the application. The two URIs `SHOW_LOGIN`, `DO_AUTH` are the only ones that *may* succeed, since neither requires authentication. Note however that if the user has already caused the system to enter an error state then they should fail. For the rest of the URIs, we require they *always* return `false`. If any of these URIs ever returned `true` it would indicate the user managed to access the application without properly authenticating first.

The property we set out to prove can now be verified by mechanically checking the resulting program has no invalid end-states.

3. Checkout must be completed once begun

Once a user has added items to the shopping cart they may begin the checkout sequence, arranging payment and delivery details before finally confirming the transaction. The SWIL program described in Section 6.6 aimed to ensure this sequence could not be interrupted i.e. the user should be prevented from leaving the sequence partly completed.

To verify this, the user is represented by the same code used in Section 6.6.1, in Figure 6.6.1. The code non-deterministically chooses URIs (from the `rchan` channel) and fires them at the automaton, keeping track of whether the system has entered the error state.

The `proctype machine` needs to be modified, with the following snippet of code added at the beginning of the checkout procedure:

```
assert(!in_checkout);
in_checkout = true;
```

At the end of the checkout routine the following snippet is added:

```
assert(in_checkout);
in_checkout = false;
```

Finally, the variable `in_checkout` is declared at the beginning of the program:

```
bool in_checkout = false;
```

The desired property – that the checkout sequence is non-interruptible once begun – can now be verified by asking SPIN to check the assertions always hold. Note we are partially depending on the structure of the application to justify this claim. If the user leaves the sequence part of the way through then we rely on the fact that the user will always be able to return to the menu and restart the sequence. Therefore if the sequence is interruptible SPIN will be able to find an error trace where the user enters the sequence again, after interrupting it the first time, triggering the assertion failure.

6.6.2 Scalability

This section discusses the applicability of the techniques described in this Chapter to larger e-commerce websites. For expository purposes the example application described in this Chapter was kept small and simple and used only 12 distinct URIs; a realistic commercial site is likely to use many more.

In the example described here, users were required to log on before being able to access any part of the site (apart from the login screen). A realistic commercial site is likely to allow users quite a lot of freedom to browse the site before being forced to log in. Not only is this more convenient for the user but it also allows third-party search engines to index the site freely. Rather than forcing users to log on when it becomes strictly necessary (e.g. when about to make a purchase), commercial sites often *encourage* users to log on earlier in order to receive personalised recommendations. A realistic policy has to cope with all of these eventualities. Since the authentication step can happen (almost) at any time, the effect is to multiply up the number of states in the final machine. However, the effect of this on the policy document itself is mitigated using the metaprogramming system– the authentication steps can be wrapped up in a function in a similar way to the example functions `browse` and `sequence`.

As already indicated, a realistic e-commerce site is likely to be much larger than the toy example described here. Much of this size difference is likely to stem from the size of the product catalogue: the in the example here there were only three product URIs. However the effect of this expansion can be mitigated by grouping all the product URIs together and treating them as a single automaton input. This avoids having to maintain a manual list of all the products in the policy specification and reduces the number of states in the policy automaton.

A realistic site is likely to have a set of sub-applications dealing with issues such as (i) managing account details; (ii) creating a wish-list; and (iii) writing reviews of products. It is envisaged that each of these could be represented by small a automaton in a similar manner to the "checkout" procedure in the example of this Chapter.

In summary, a realistic site will certainly require a larger, more complicated policy specification than that presented here. However, provided that chunks of the application's URI-space can be grouped together (e.g. an entire product catalogue) and provided that good use is made of the metaprogramming system, it should still be possible to have a reasonably concise policy covering a large application.[9]

6.7 Related Work

Our work directly exploits five strands of research; the main novelty is in their combination. The strands are: (i) model-based approaches to software development and analysis; (ii) describing interfaces using automata; (iii) enforcing security policies dynamically on web-applications; (iv) meta-programming techniques; and (v) dynamic user interface transformation. Each of these will be discussed in the following sections. Additionally, Graunke et al [76] proposed a system for automatically restructuring programs for the web; this will also be discussed.

6.7.1 Model-based approaches to software development and analysis

A model is a finite abstraction which is intended to capture some of the essential properties of a system. Described earlier in Section 2.6, model-checking [42] –

[9] One may argue that, if the resulting policy is too big, then the application is fundamentally too complex anyway.

the process of automatically verifying properties of such a model – is gaining in popularity as a software verification method. However, creating a useful model of a software system is a difficult task. Software systems often have large numbers of states and an abstraction must be carefully chosen to minimise the resulting number of states in the model (in the worst case the number of states is proportional to the time taken to verify) while still remaining faithful enough to be able to verify the properties of interest.

Producing models from software

There have been many attempts to automatically translate source languages into models, a process known as *model-extraction*. Systems have been built for languages including Java [83], Ada [56] and C [89]. Compiler techniques such as partial evaluation and program slicing are commonly used to reduce the complexity of the models created by removing aspects of the application behaviour unrelated to the property of interest.

Systems vary by how much the user is involved in the model extraction and checking process. Two systems shall be described: Bandera and SLAM. Bandera [44] is a generic software model-checking tool for Java code. Bandera contains an abstraction library which the user must use to select appropriate abstractions for individual program variables. For example a Java `Vector` object (a dynamic array) may be modelled by various lengths of finite buffer. The simpler the model chosen the faster the model-checking process will be. However, if the model-checker fails to verify the desired property it might be necessary to refine the model using some longer length buffers at the cost of a slower model-checking process.

In contrast to Bandera which attempts to verify whole Java programs, the SLAM [17] project at Microsoft aims to mechanically check that library interfaces are correctly used by C programs. SLAM operates by first translating a C program and a library-related safety property into a finite-state C program with the same control-flow but only boolean variables, related to the property being checked. The program has a special error state which corresponds to a violation of the safety property (like that triggered by the SWIL command `error`). Once the abstract program has been created, a model-checker determines if the error

state can be reached through any possible path through the program. If an error path is found it is related back to the original C code. Symbolic computation and a theorem prover are used to determine whether this path corresponds to an actual error (i.e. a bug) or whether the path is an artifact of the abstraction process. If the latter than the model is refined and SLAM is executed again. Therefore using SLAM is an iterative process which continues until one of three things happen (i) the automatic refinement step fails (requiring user input); (ii) an actual error is discovered; or (iii) the code is pronounced safe, with respect to the specified safety property.

Producing software from models

Rather than starting with complicated software and attempting to extract a suitable model from it, some researchers have approached the problem from the other direction and attempted to translate models into implementations. Early work in this area includes a system [112] which allows PROMELA specifications to be transformed directly into a C program, with the addition of a scheduler, random number generator (to handle non-determinism), timeout and I/O mechanisms.

Model-Carrying Code

Model-Carrying Code [149] (MCC), inspired by earlier work on Proof-Carrying Code (PCC) [119] is a proposal where the code producer pairs the implementation (in binary form) with a high-level model of its "security-relevant" behaviour. The code consumer can statically check (at install-time) the model against its security requirements (perhaps through model-checking) and then execute the code while dynamically checking it against the model in the manner of Schneider et al [142].

A similar approach was taken by Madhavapeddy et al [114] who proposed a special-purpose policy language called Stateful Syscall Policy Language (SSPL) which described a high-level model of the acceptable system calls made by a Unix process. Models written in SSPL could be analysed statically and were enforced dynamically through Unix system call monitoring and interception.

Relevance to this work

SWIL policies are abstract models written in a language similar to PROMELA, the input language of the SPIN model checker. There are a few key differences be-

tween PROMELA and SWIL notably that SWIL has built-in types and functions for processing HTTP requests and SWIL programs are essentially deterministic, with all non-determinism pushed into calls to native functions. SWIL is designed to be a half-way house between a modelling language like PROMELA used to create abstract, non-deterministic models and an implementation language like O'Caml which is used to create concrete, deterministic code. No attempt is made here to generate SWIL code automatically; rather we take the view that manually specifying the policy is both useful for documentation and often inevitable, especially when the source code to applications is not available.

The system proposed here is closest in spirit to Model-Carrying Code: SWIL policies model "security-relevant" behaviour insofar as they relate to forceful-browsing attacks. Policies are statically analysed before ultimately being dynamically enforced. The primary difference is the context of use; MCC is intended to bridge a gap of understanding between distrusting code producers and consumers who may have conflicting goals. In contrast, the goals of web-application code producers and SWIL policy writers are assumed to be the same; neither wishes an application to be vulnerable to forceful-browsing attacks.

6.7.2 Describing Interfaces with Automata

Alfero and Henzinger propose describing interfaces as automata [46] in order to capture the temporal behaviour of interfaces. They use automata both to specify the *input assumptions* and the *output guarantees* of an interface. They define interface compatibility optimistically: interfaces are said to be compatible if there is a possible environment in which they can work successfully together.

In a similar theme to Interface Automata, typestate checking [156] is

> a compile time program analysis technique which enhances program reliability by detecting type-correct applications of operations which are non-sensical in their current context

A typestate analysis associates each program object with an automaton which describes the order in which operations can be invoked. The state of the automaton is the *context* mentioned in the quote above. For example a file object supporting

operations **open**, **read** and **close** might be described using the regular expression

$$\textbf{open}\ \ \textbf{read}^{\star}\ \ \textbf{close}$$

i.e. the file must be opened first before it may be read and it must be closed eventually. Typestate analysis can be used to check a variety of interface rules, from security properties to detecting when data structures can be successfully freed as an alternative to garbage collection.

Interface Automata and typestate checking have much in common with this work. However there are several major differences. Firstly, interface automata and typestate checking are compile-time procedures which aim to check the compatibility of software modules statically. In contrast, although SWIL policies may be generated and analysed statically they are nonetheless intended to be enforced at run-time against an unmodified application. Unlike the other approaches SWIL policies are not checked against the application source-code and so may be applied even when source-code is not present. Additionally multiple SWIL policies may be in force at any one time, each running in parallel. Finally, SWIL policies are not simply passive monitors of application behaviour but can be used to dynamically transform interfaces, by removing links and forms associated with future error states in the SWIL model.

6.7.3 Sanctum AppShield

Sanctum AppShield was first mentioned in Section 5.6.2 and its ability to protect web-form parameters was discussed. AppShield is also claimed to protect applications from forceful-browsing attacks, through a component called a *Dynamic Policy Recognition Engine* which monitors live HTTP traffic and infers a policy dynamically. While it has a low installation overhead and does not require expert users to write a policy, a purely dynamic approach like this is inherently much less powerful than the manually specified approach specified here. Without an up-front policy specification, the system must either be taught, by example, which sequences of requests are valid or it must be conservative and dynamically track all links and forms generated by the application across every session[10]. The former

[10]Consider that a user may bookmark a URI in one session and recall the bookmark from a second session.

approach (training the system) suffers from problems if the examples presented to the system do not cover all valid application behaviour. The latter approach (tracking all generated forms) may be *too* permissive and allow attacks like the travel-booking example of Section 6.1.1.

The systems presented in this thesis all support policy abstraction: the ability to keep a concrete security policy document separate from main application code. Two of the advantages of having a concrete abstract policy are (i) the policy is a valuable form of interface documentation (and can be viewed as an interface transformation); and (ii) the policy can be analysed statically enabling the verification of important properties. These benefits are not conferred by an automatic, hidden-policy, Sanctum-like solution. In essence the system presented here combines static and dynamic checking in a novel way.

6.7.4 Meta-programming

The SWIL Meta-programming System used an O'Caml embedding of SWIL to generate and combine SWIL program fragments into more complicated SWIL programs. Although not the main focus of this chapter, this section describes a number of other approaches to meta-programming and compares them to the SWIL Meta-programming System.

A traditional programming language is said to have two *stages*: compile-time and run-time. Multi-stage programming [159] (also known as generative programming or meta-programming) adds extra stages. At a minimum a multi-stage system would have: generation-time, compile-time and run-time where the code run at generation-time creates further code which is subsequently compiled at compile-time.

Multi-stage systems are surprisingly common. Systems tend to vary in two aspects: firstly they vary in how they manipulate generated code (some using strings while others may use syntax trees) and they vary in how many guarantees they are prepared to make about the generated code (i.e. whether it definitely typechecks). The parser generator `yacc` generates C code by bashing together strings containing fragments of syntax. There is no guarantee that output generated in this way will successfully parse. Described in this chapter, SMS produces SWIL code by bashing together trees of concrete syntax (known as *program frag-*

ments). Although guaranteed to parse, output generated in this way may fail to typecheck or may fail the well-formed program tests, resulting in an O'Caml exception being thrown. Recent research into meta-programming has resulted in systems like Meta-ML (described below) which can guarantee that well-typed generator programs generate well-typed output programs[11]. A couple of interesting meta-programming systems are described in the following few sections.

Meta-ML/ MetaOCaml

Based on the ML language family, Meta-ML [160] and now MetaOCaml [3] allow programs to have an arbitrary number of stages. Meta-ML adds extra syntax for staged computations: `< e >` is a staged computation which will evaluate the expression `e` in the next stage. Brackets may be nested: `< < e > >` indicates an expression will be evaluated in the next stage but one. Staged computations have types which indicate both their result (when finally evaluated) and their context. A staged computation can be immediately evaluated using the operator `run` which behaves like a type-safe `eval` operator. As mentioned in the previous section above, Meta-ML and MetaOCaml both perform static type checking guaranteeing that code will not be generated which contains a type error.

If the SWIL Meta-programming System were implemented in MetaOCaml then we could use Taha's observation [159] that a staged interpreter is a program translator. The SWIL interpreter combined with a generated SWIL AST could be used to directly generate native O'Caml code in a type-safe manner.

C++

C++ [157] created by Bjarne Stroustroup is an extension of C with accidental [170] support for meta-programming using *templates*. Templates were intended for generic programming; creating classes and functions parameterised over types. For example a class which implements a linked list might be written

```
template<class T>
  class List {
    T *head;
```

[11] A difficult property when variable bindings can be generated, possibly shadowing existing bindings of different type.

```
    List<T> *rest;
    ...
  }
```

where the `T` is a type variable. A linked list of integers would have type `List<int>`.

However as well as providing type parameters to templates C++ allows templates to accept integer value parameters. A template may be defined recursively as follows:

```
template<int X>
  class c {
    static const int result = X * c< X-1 >
    ...
  }
```

When the program is compiled, the compiler will expand the nested templates completely (possibly with a maximum recursion depth). C++ is therefore a three-stage programming language where template evaluation generates code which is compiled and then executed.

Meta-programming using templates has been exploited by several projects. The MTL library [155] uses templates to generate fixed-size kernels for linear algebra routines while the Blitz++ library [169] generates specialised algorithms, unrolling loops where possible for fast vector and matrix computation.

FreshML/ Fresh O'Caml

Although not strictly meta-programming languages, FreshML [153] and Fresh O'Caml [152] are derivatives of the ML family with extra facilities for matching and generating syntax. Originally ML (which stands for Meta-Language) was intended for manipulating logic expressions and accordingly supports sophisticated pattern matching and datatypes useful for this purpose. However when performing operations over syntax trees it is often difficult to handle variable binding properly. Syntax trees should be treated as equivalent if they differ only in the names of bound variables. Furthermore care must be taken when generating new names to manage *freshness* correctly i.e. to respect the naming and scoping rules of the

language and not accidentally confuse names that should be distinct. FreshML and Fresh O'Caml are specially designed to make syntax manipulation tasks easy and safe. It would be promising to try implementing future versions of the SWIL meta-programming system using these tools.

6.7.5 User Interface Transformation

Dynamically changing (or transforming) interfaces has been studied in several different contexts. Firstly, many applications deal with and must be able to display dynamically-generated data. Therefore interfaces must transform themselves at run-time to cope with this generated data. Secondly, in the highly mobile world of UbiComp, applications may be called upon to adapt and run on devices ranging from large wall-sized displays to small PDAs. Thirdly some application transformations are motivated by security. Each type of transformation is described in the following sections.

Pre-programming transformations

User interfaces typically comprise a static part known at application-design time and a dynamic part which depends on data only available at run-time. Traditional user interface builders (e.g. Cardelli's direct manipulation interface builder [30]) focus on the static part. As part of the influential MASTERMIND project, Castells et al [38] propose a declarative language which uses constraints to specify how the application should respond to dynamically changing data.

Rather than using a traditional user interface builder program, an alternative is to generate the whole interface from scratch. For example one approach, taken by Engelson et al [58], is to automatically generate interfaces from the datastructures used by applications; for example a structure might become a table and a string value might correspond to a text box. This has the advantage that if the code is updated (perhaps by dynamically loading a module) then the interface can change to match.

Transformations for UbiComp

Ubiquitous Computing applications pose a difficult challenge for user interface designers. Pier and Landay [128] outline how future applications will be expected to be highly mobile and able to cope with a wide range of display sizes, from wall-

sized displays down to small PDA devices. They propose the idea of a *location-independent interface*: an abstract entity which may take multiple physical forms; anything from a stylus-based interface on one device to a voice recognition system on another.

One attempt at making a location-independent interface was made by Todd and Glinert who proposed a system called POLYGLOT [165]. In POLYGLOT interfaces written in an abstract form (similar to HTML form controls) are rendered on specific devices by *presenter* objects connected using a CORBA [123] middleware to the back-end applications. Presenters are device-specific; they choose the style of the interface and can render it using whatever facilities are present on the device. Presenters can be connected and disconnect at will; multiple presenters can control the same application simultaneously.

Perhaps due to the popularity of the devices, adapting interfaces to work on PDAs has been studied in detail. One approach is to transcode images contained within web-pages, e.g. by reducing the quality and size of JPEGs with a custom web-server [39, 82]. Alternatively HTML pages themselves may be parsed and summarised such that they better bit onto PDA screens [29].

Transformations for Security

Some interfaces adapt their appearance in response to security requirements. When untrusted Java applets spawn top-level windows they are specially marked to warn the user that they are not local applications. The Trusted Computing Group initiative [166][12]places each application into one of two groups: a low-security group and a high-security group. The high-security applications are trusted to access resources like encryption keys used to secure digital content. High-security applications needing to ask the user for sensitive details (e.g. passwords) are able to establish a *trusted path* to the screen and keyboard. The aim is to prevent low-security applications from being able to fool the user by forging convincing-looking password dialogs or simply record key-presses. Whereas Java applets are marked with words like “untrusted” to warn the user, a TPCA high-security application will be marked with some special marking known only to the

[12]Also known as “Trusted computing”, “trustworthy computing”, “safer computing”, “TPCA”, “Palladium” and even “treacherous computing” depending on who you talk to.

user to provide some reassurance that the application is genuine. Effectively the marking would be a shared secret between the application and the user.

User Interface transformations are common on existing web-sites. Some bulletin-boards transform the appearance of links, specially marking those which are written by users. This prevents users unknowingly clicking on links of the form:

```
<a href="http://malicious.com/">http://good.com/</a>
```

which causes the text `http://good.com/` to be displayed on the page but when the link is clicked the actual URI fetched is `http://malicious.com/`. When combined with client-side Javascript code capable of hiding the real URI bar (containing the give-away URI `http://www.malicious.com/`) and HTML designs which mimic the appearance of the browser, a user might be tricked into thinking that the page loaded from `http://www.malicious.com/` really is `http://good.com/`.

Finally, it is important that any security-related interface transformations are never relied upon in isolation. A user of a program with a visible security marking will inevitably grow to trust that feature, increasing the potential for disaster should that feature be subverted. A case in point is the typical web-browser "padlock" icon displayed on the screen when the browser is using HTTPS i.e. HTTP over SSL. The padlock is intended to convey the fact to the user that the site can be trusted; that its identity was confirmed and all communication is encrypted. However the SSL specification [53] allows ciphers to be renegotiated at any time and the NULL cipher is always supported[13]. It is possible for an HTTPS server to request the NULL encryption cipher at any point — with the result that the data is not protected but the padlock icon is displayed.

Comparison with other UI transformation schemes

The dynamic response transformation described in Section 6.5.1 exists in a different niche to most of the interface transformations presented in this section. It is not intended to be exclusively used as part of the application design process; rather it is intended both to be used to help design new applications as well as to be externally imposed to fix pre-existing applications.

[13]It is needed at the beginning before another cipher has been negotiated.

The transformation system proposed in this chapter is built on top of HTML transformations, similar to those needed to make web-applications usable on small screen devices like PDAs. However, rather than remove *unnecessary* features (which do not fit on screen) we remove *illegal* features which would lead to a security policy violation.

The security related transformations are similar in spirit to the transformation scheme proposed in this chapter. However there are a number of key differences: (i) the system described here abstracts policy from the application code rather than entangling the two together; (ii) the transformation system does not need to be foolproof – even if a bad link is selected the request will still be blocked by the application-level firewall; and (iii) the policy here is dynamic and may be adjusted according to demand (e.g. it is possible to disable temporarily a part of an application while maintenance is being performed on it[14]).

6.7.6 Building web-applications using continuations

Several people have suggested making use of continuations to structure web programs [76, 131]. For example, Graunke et al [76] propose a system where, whenever an application wishes to interact with a user it stores its current execution context as a continuation—encrypted and protected with a MAC (to prevent tampering)—on the client-side in a form. Users are free to save the continuation-carrying forms to disk and revisit them at any arbitrary time in the future. Users can effectively *backtrack* to any interaction point as if they were running the whole application within a *timetravel debugger* [26]. In the system proposed by Graunke et al even application mutable state is stored on the client-side within cookies.

Such a scheme works well provided applications are fundamentally free of server-side side-effects; i.e. they must not rely on any mutable state on the server. Once some server state has been introduced (e.g. in the form of an e-commerce transaction log) then the user needs to be prevented from backtracking to a point before the state was last updated, otherwise they might see a stale view of the application and become confused. The strength of the work presented in this chapter is that nomatter which languages – recall a web-application might consist

[14] Although a realistic system would likely need some mechanism for telling users *why* the interface elements are disabled and for how long the restriction might last.

of multiple components written in different languages – are used to create the actual application, control over backtracking can be implemented using a SWIL policy and server-state can be protected.

6.8 Summary

This chapter presented SWIL, a language intended to protect applications from *forceful-browsing* attacks. SWIL specifies the acceptable order of web-application interface function invocations as that accepted by a deterministic finite state automaton. A translation from SWIL into PROMELA was presented allowing SWIL policies to be analysed statically by the SPIN model-checker before being dynamically interpreted by an application-level firewall. More implementation details are forthcoming in the following chapter.

To facilitate creating complex SWIL policies, an embedding of SWIL into O'Caml was created, called the SWIL Meta-programming System. This system allows the generation, manipulation and combination of SWIL *program fragments* forming complete SWIL programs.

As well as simply blocking requests in violation of the installed policy, the system proposed here is able to perform dynamic interface *transformation*. Links and forms contained within application HTML responses may be analysed and those which are determined to inevitably lead to a future policy error state are removed.

Finally, to demonstrate the utility of the system described in this chapter, an extended case study was presented based on a hypothetical e-commerce web-site. Suitable policy to protect the application was created and a number of useful properties statically checked with SPIN.

CHAPTER 7

Implementation

This chapter describes the implementation of the application-level policy systems described in this thesis. The policies described here fall into two categories: policies governing application *mobility* and application *interfaces*. Chapter 3 and Chapter 4 described the Unified Spatial Model (USM) and the Mobility Restriction Policy Language respectively. Chapter 5 and Chapter 6 described SPDL-2 and SWIL, two policy languages for securing vulnerable application interfaces. Mirroring this split, this chapter is also divided into two parts: Section 7.1 addresses mobility and Section 7.2 addresses interface policies. This chapter also includes details of a new policy mechanism known as *mobility enforcement policies*. These policies are written as simple Java classes and embody a particular movement strategy (e.g. "the agent should always run in the same room as person X"). Object instances of the classes are associated with individual *sentient mobile applications* – applications which automatically sense their environment and react accordingly. Like the other policy systems described in this thesis, these new policies are abstract specifications which may be placed in a library and shared between multiple clients.

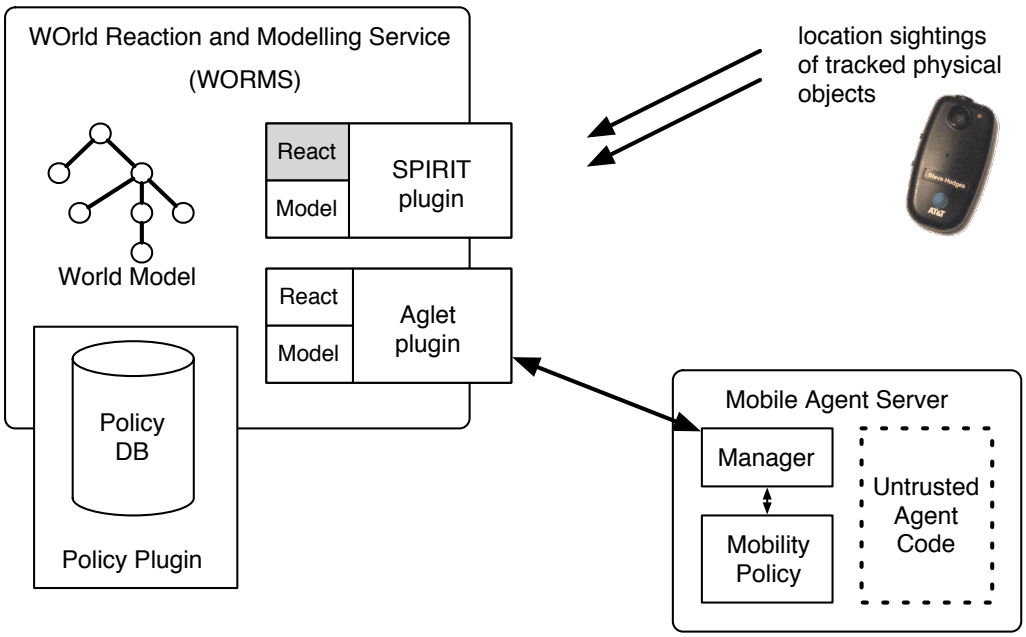


Figure 7.1: The structure of the system designed to implement mobility security policies. Note in particular the react interface of the SPIRIT plugin is greyed out (i.e. disabled); this is explained later in the text.

7.1 Mobility

Earlier Chapter 3 described a spatial model capable of simultaneously representing the locations of physical entities such as people and virtual entities such as mobile agents. The model was used as the basis for a new kind of policy: the mobility restriction policies described in Chapter 4.

This section describes a Java-based implementation framework comprising of (i) a spatial middleware system maintaining a world model in the style of Chapter 3; (ii) client and server-side support for implementing the security policies of Chapter 4; and (iii) a new kind of mobile agent policy known as *mobility enforcement policy* used to *cause* an agent to move in response to external stimuli. The structure of the framework is depicted graphically in Figure 7.1.

The implementation is described in two parts. First, Section 7.1.1 begins the description of the spatial middleware system known as the WOrld Reaction and Modelling Service (WORMS). The model supports a plug-in architecture. Plug-in modules are used to connect the middleware to external location systems and

to help implement mobility restriction policies. The middleware is designed to be location-system agnostic; adding support for a new location-system simply involves writing a new plug-in module. As well as passively monitoring the state of the world, the middleware is capable of taking action to programmatically create, destroy and move certain kinds of entities. This facility is used by a mobility policy plug-in to enforce the mobility restriction policies of Chapter 4.

Section 7.1.2 continues the discussion with a description of the software running inside the mobile agent servers. This code is divided into three parts, drawn separately on the diagram in Figure 7.1:

1. a *Manager* module which communicates directly with the WORMS Aglet plug-in;

2. a *Mobility Policy* module which implements the new mobility enforcement policies; and

3. *Untrusted Agent Code* comprising the rest of the agent code written by the user.

Recall from Chapter 4 that the mobility restriction policies were designed to influence the movements of mobile agents by either (i) blocking migration requests; or (ii) taking corrective action (e.g. killing or jailing an agent) afterwards if the policy violation cannot be avoided (i.e. it was caused by physical movement). The *Manager* module is responsible for co-ordinating the client-side part of both of these functions. It actively filters migration requests, checking for potential policy violations and also responds to commands (e.g. jail, kill) received from the Aglet plugin.

The policies of Chapter 4 were intended to *restrict* the movements of agents; nothing was said about how to *cause* agents to move in the first place. It was assumed that mobile agents would contain code to decide when to migrate and would call the appropriate migration APIs. This chapter introduces a new kind of policy known as an *mobility enforcement policy* which describes using the Unified Spatial Model *when* an agent should move and *where* an agent should move to. An application using these policies is factored into two parts: (i) a mobility policy object; and (ii) the rest of the code, which would contain no mobility-related

behaviour.

Mobility enforcement policies are *proactive* policies and may be considered as the dual of the *reactive* mobility restriction polices described previously. Enforcement policies are written abstractly as Java classes and each mobile agent is associated with a concrete object instance. Structuring applications in this way ensures that mobility behaviour is kept separate from the main agent code, allowing patterns of behaviour to be packaged into code libraries facilitating the sharing of mobility policies between applications.

The rest of this section is structured as follows: Section 7.1.1 describes the WORMS system in more detail together with information about several plug-in modules. Section 7.1.2 describes the code running in the mobile agent servers followed by a description of the mobility enforcement policies in Section 7.1.3. A number of issues arose during the development of this prototype system, these are discussed in Section 7.1.4.

7.1.1 Policy-Enforcing Spatial Middleware

This section describes the implementation of a spatial middleware system called the WOrld Modelling and Reaction Service (WORMS) which implements the spatial model of Chapter 3. The Java-based WORMS system bridges the gap between real-life sensor and reaction systems and spatially-aware mobile agents. The service constitutes a core and a set of plug-in modules. The core maintains the world model, making sure the entities in the model are well-sorted. Clients can register themselves to receive update events whenever parts of the world-model change i.e. events are generated whenever entities are created, destroyed or moved.

Plugins exist for two purposes: (i) to connect the WORMS system to individual location systems; and (ii) to impose policy on the system. The system uses a publish/subscribe mechanism; policy modules register themselves with the core, receiving events whenever any updates to the world model happen. The policy modules continually monitor for any policy violations. If any breach is observed then they take the appropriate corrective action (e.g. in the policies described in Chapter 4 this might involve killing, creating or jailing an agent).

Location plugins expose two interfaces: a passive *modelling* interface and an

active *reaction* interface. The modelling interface is used by modules which monitor the outside world (the "real-world") and feed updates into the world model. The reaction interface is used to effect changes in the outside world being modelled. For example, it is through the reaction interface that mobile agents may be frozen or destroyed.

As mentioned above, the Service is not specialised to one particular location system but rather is designed to be location-technology agnostic. Each particular location technology is associated with a plugin module which implements either or both of the interfaces (modelling and reaction). Currently the system has two such plugins: a *SPIRIT* plugin and an *Aglet* plugin.

SPIRIT Plugin

The SPIRIT [11] system first mentioned in Section 3.3.4 is a piece of location-monitoring middleware developed originally at AT&T Laboratories Cambridge. Its primary function is to take a stream of raw location events received from a network of ceiling-mounted sensors and combine these with a spatial database to produce and disseminate high-level location events concerning people and objects within the lab. Objects to be tracked are equipped with Active-Bat [173] devices — radio-triggered ultrasound-emitting location tags described earlier in Section 2.2.2. The clients of the system are location-aware applications, for example the "active map" program which displays a top-down view of an office complete with the positions and orientations of people and equipment, updated in real time. Figure 7.2 shows an active bat on the left and a screenshot of the "active map" program on the right. Note that the map is showing a zoomed-in view of an office in the Laboratory for Communication engineering where 3 people are present, represented by usernames of `kjm25`, `jpw20` and `ja316`. Desks, computer terminals and phones are also visible.

SPIRIT is a *spatial indexing system*. Primitives in SPIRIT are spatial regions (shaped volumes of space) which are allowed to move freely with respect to one-another. SPIRIT generates events whenever a region overlaps, ceases to overlap, contains another or ceases to contain another region. The functionality of the SPIRIT system is best explained by means of an example. The "desktop teleporting" [134] application is a commonly-used program which automatically moves a

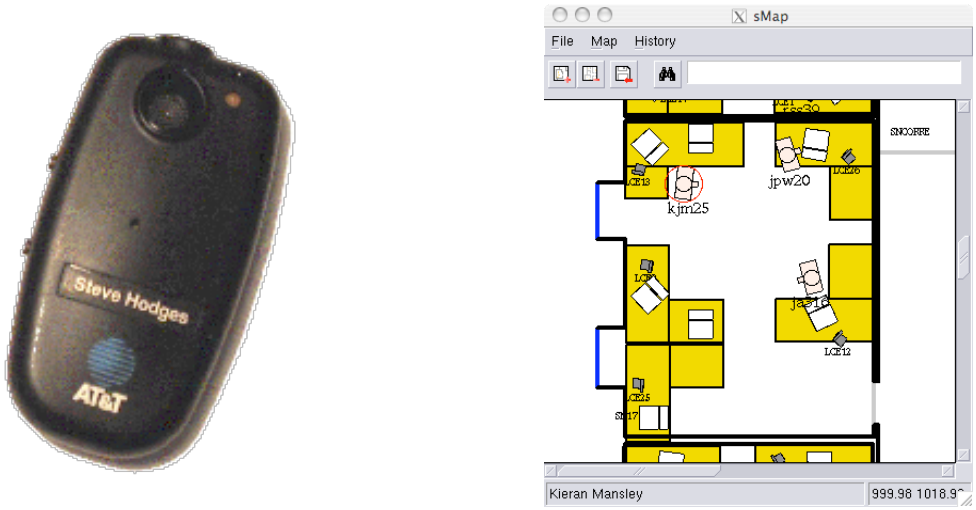


Figure 7.2: On the left: an example active bat. On the right: a screenshot of the "active map" program.

user's desktop to the terminal nearest their current physical location. The teleporting application exploits SPIRIT in the following way: every computer monitor is associated with a region of space placed in front. Roughly speaking, the region of space corresponds to the space from which the monitor would normally be used. Therefore a smaller monitor would be associated with a smaller region than a bigger monitor. Similarly, users are associated with a volume of space representing the region in which an object has to be located for them to interact effectively with it. When a person turns to face a monitor the two regions overlap, generating an event which causes the teleport application to display the user's desktop on the screen. When the user moves away, another SPIRIT event is generated which removes the teleported desktop.

The SPIRIT plugin[1]provides a mapping between the spatial regions understood by SPIRIT and specific entities in the entity hierarchy. An example of such a mapping is shown in Figure 7.3. The bottom half of the figure shows the output of the active map application monitoring two rooms — "Room 9" and "Room

[1]The low-level interface between the native SPIRIT C++ libraries and Java was written by Alastair Beresford using the Java Native Interface (JNI) facility. This small piece of code generates a continuous stream of location events which are filtered and processed by the rest of the plugin code.

10". The SPIRIT plugin has associated these rooms, two people (userid `kjm25` and `acr31`) and a pair of workstations with entities inside the Service. Once the mapping is established, the plugin listens for high-level containment events from SPIRIT (such as "user `kjm25` has left Room 9") and makes the corresponding changes in the world model. Note in the earlier diagram in Figure 7.1 how the "React" part of the SPIRIT plugin was greyed out. This is because SPIRIT is an entirely passive system; it is not possible to command SPIRIT to physically move objects around. It is possible to imagine a modified SPIRIT system which can influence its environment. This might be used, for example, to command robots to move around the building.

Aglet Plugin

The Aglet system – a Java-based mobile agent system from IBM – was first described in Section 4.7.1. Aglets (called *agents* in the terminology of Section 2.1) execute in a "context" – represented by an instance of the class `AgletContext` (called *locations* in the terminology of Section 2.1). A single computer can run multiple Aglet VMs (called *mobile agent servers* in the terminology of Section 2.1), each associated with a unique TCP port number. Each Aglet VM can contain multiple contexts. Individual Aglet contexts are named globally by URIs of the form `atp://host:port/context` where the URI scheme `atp` indicates the "Aglet Transport Protocol". Aglets exploit the conventional Java event model: events are generated when agents are created, destroyed or move between contexts. Interested parties can register themselves (as event *listeners*) with the Aglet system and receive callbacks when the events are generated.

Like the SPIRIT plugin described above, the Aglet plugin is responsible for maintaining a mapping from hostnames, context names and Aglet names to entities in the world model. Such a mapping for a fragment of a world model is displayed in Figure 7.4. The diagram depicts a path of entities from the root to an agent called "music player". The first two entities in the path ("World" and "Charlie's Office") represent physical objects and are therefore ignored by the Aglet plugin. The remaining four entities have some meaning in the Aglet universe: the "laptop" entity is a piece of hardware with a hostname ("laptop"); the context "main-123" corresponds to an Aglet process with process ID (PID) 123;

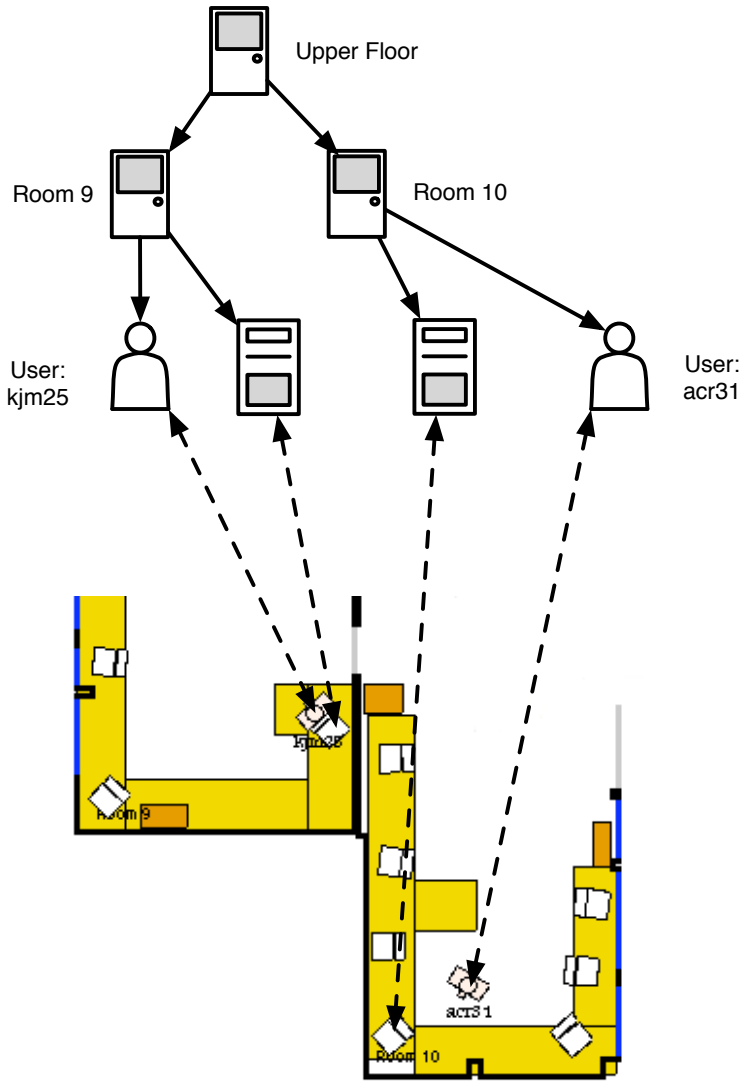


Figure 7.3: An example of the mapping between SPIRIT objects as displayed by the program `smap` (bottom) and entities in the WORMS world model (top).

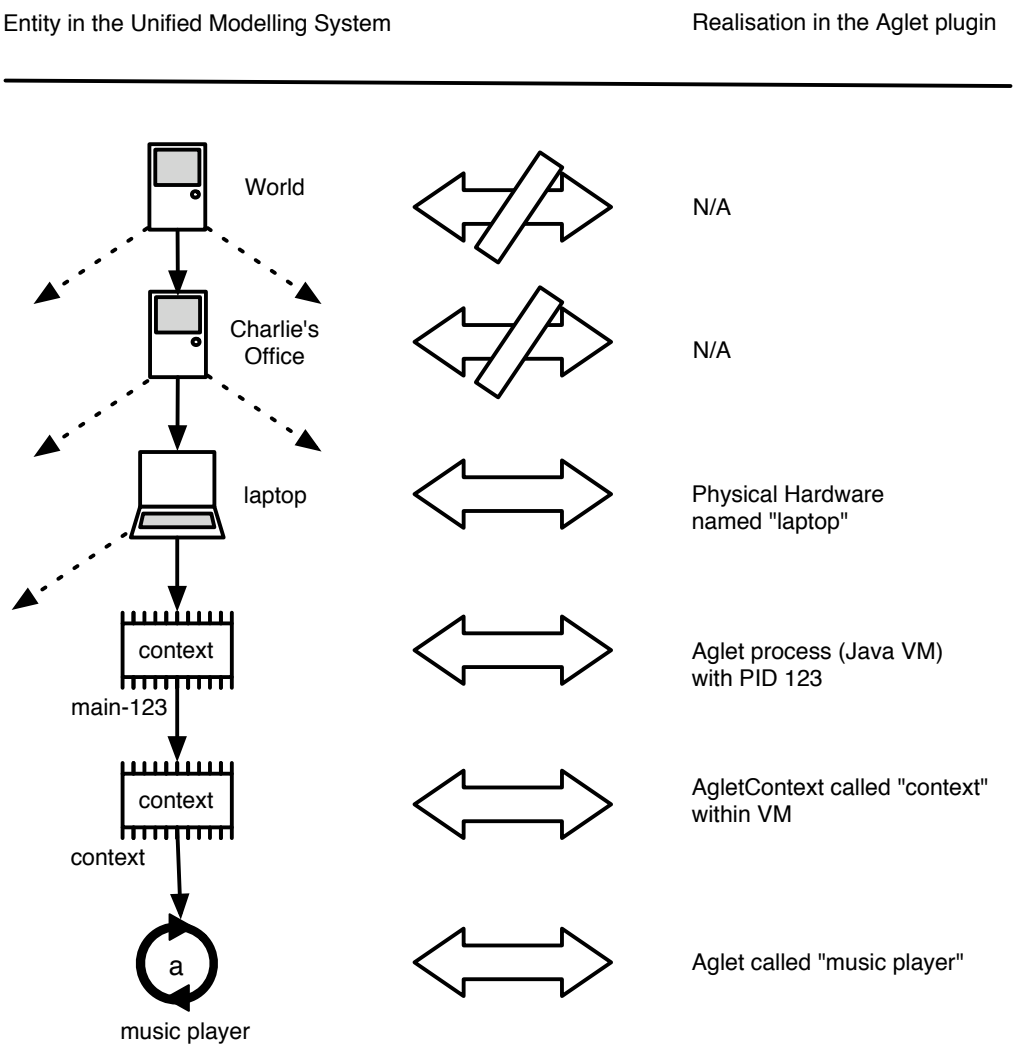


Figure 7.4: Diagram showing how entities along a path in the world model are realised by the Aglet plugin.

the context "context" corresponds to an `AgletContext` within the java VM called "context" while the entity "music player" corresponds to a running Aglet called "music player".

Unlike the SPIRIT plugin described earlier, the Aglet plugin supports the *react* interface, to control the runtime behaviour of agents. Ideally it would be possible to completely control (migrate, jail or kill) untrusted agents running in the system without their consent. However, in the prototype described here there are a number of limitations inherited from Java which make this impossible. Therefore in the current implementation, the Aglet plugin sends control messages to the individual agents and trusts them to respond correctly to the control messages. More details and discussion of the limitations inherent to the design of the prototype system are presented later in Section 7.1.4.

Mobility Restriction Policy Plugin

Recall the mobility restriction policies introduced in Chapter 4 which consisted of 4-tuples:

$$\langle location, formula, times, onfail \rangle$$

where $location$ is a path expression (see Section 3.1.3) designating a set of specific entities where the assertion given by $formula$ should hold. If, with respect to the time period described by $times$, the assertion becomes violated (e.g. by the physical movement of an object) then the system will attempt to execute the command described in the field $onfail$.

In any realistic system, policies written by different users are likely to come into conflict with each other. Section 4.5 outlined one possible conflict resolution strategy in which each entity is associated with an *owner*, who has a say in the outcome of any $onfail$ action (e.g. the killing or jailing of an agent) involving that entity.

Therefore the mobility restriction policy plugin must maintain a database of users, entities, active policies and keep track of which user owns each entity. The plugin performs two functions: (i) it receives migration requests from deployed mobile agents and either confirms or rejects the requests; and (ii) it observes other movement (e.g. physical movements) and determines when a policy violation has occurred and what corrective action should be taken (if any). These two functions

are now explained with an example.

Example

Consider a simple office environment containing two people: Alice and Bob, both of whom have one office and one PC each. An example world model is displayed graphically in the upper part of Figure 7.5. The world model is similar to the example originally used in Section 4.3. Note that dashed boxes and arrows are used to indicate the owners of each entity. There are three owners in the diagram: Alice, Bob and a user called "The Boss" who is not physically present in the model. In the example there are two policies installed in the WORMS system, one belonging to Alice and one to Bob. Alice's policy is as follows:

$$\begin{array}{lrl} \langle & \mathit{location} & = \quad \texttt{World}, \\ & \mathit{formula} & = \quad \Box\neg\texttt{music player}[\mathbf{T}] \mid (\Diamond(\text{Alice}[\mathbf{T}] \\ & & \quad \mid \Diamond\texttt{music player}[\mathbf{T}] \mid \mathbf{T})), \\ & \mathit{times} & = \quad \mathit{Always}(10\ \mathit{seconds}), \\ & \mathit{onfail} & = \quad \mathit{Kill}\ \texttt{World//}\bullet\bullet\bullet\texttt{//music player} \quad \rangle \end{array} \tag{7.1}$$

"for all time, either the `music player` is absent (not running) or it is running in a place next to me. If this remains false for 10 seconds, kill the `music player` (wherever it is)"

Note that in the upper part of Figure 7.5 the formula in the policy is holding because Alice and her agent are both within her office. The second active policy written by Bob is the same as that given back in Chapter 4 i.e.:

$$\begin{array}{lrl} \langle & \mathit{location} & = \quad \texttt{World/*}, \\ & \mathit{formula} & = \quad \Box\neg\texttt{Bob}[\mathbf{T}] \vee (\Diamond\texttt{Bob}[\mathbf{T}] \wedge \Box\neg\texttt{audio}[\neg\mathbf{0}]), \\ & \mathit{times} & = \quad \mathit{Always}(3\ \mathit{seconds}), \\ & \mathit{onfail} & = \quad \mathit{Jail}\ \texttt{/}\bullet\bullet\bullet\texttt{/audio/*} \quad \rangle \end{array} \tag{7.2}$$

"if ever I'm in an office with a music playing agent, jail the agent if it has not left within 3 seconds"

Note that in the upper part of Figure 7.5 the formula in the policy is holding because Alice and her agent are in one office and Bob is in another.

Now, consider what happens when Alice walks into Bob's office. First of all the location system senses Alice's new location and sends this information to all registered applications, including the WORMS service. The WORMS service updates the authoritative copy of the world model resulting in the configuration displayed in the lower part of Figure 7.5. Note that Alice's policy is now being violated but Bob's is not. Since the WORMS service itself acts as a location system, it sends location update events to all registered clients: specifically all the registered mobile agent servers running on all the PCs. Observe that Alice's policy had a $times$ field containing $Always(10\ seconds)$; this indicates that Alice's agent has 10 seconds to take an action which fixes the policy violation before the $onfail$ clause is executed. The WORMS system sets an internal timer for 10 seconds and waits for the agent to respond, if it can.

There are now several possibilities, including:

1. Alice walks back into her office, her policy is nolonger in violation and the timer is cancelled;

2. the music playing agent quits (or crashes), it is removed from the model, the policy is nolonger in violation and the timer is cancelled; or

3. the music playing agent attempts to migrate to a PC in Bob's office.

The last case is particularly interesting and worth considering in detail. Figure 7.6 shows the sequence of events starting with the initial message from the location system to the WORMS service triggered when Alice switches room. We have already considered up to the time marked with the asterisk, just after the WORMS service has set a timer and sent an update message to all the registered clients. We assume that the music player is a "follow-me" music player and therefore it decides to follow Alice by migrating into Bob's office. It sends a request to the WORMS system asking for permission to migrate there. The centralised WORMS system considers all agent migration requests. The WORMS system computes that, should it allow the agent to move, it would satisfy Alice's policy but place

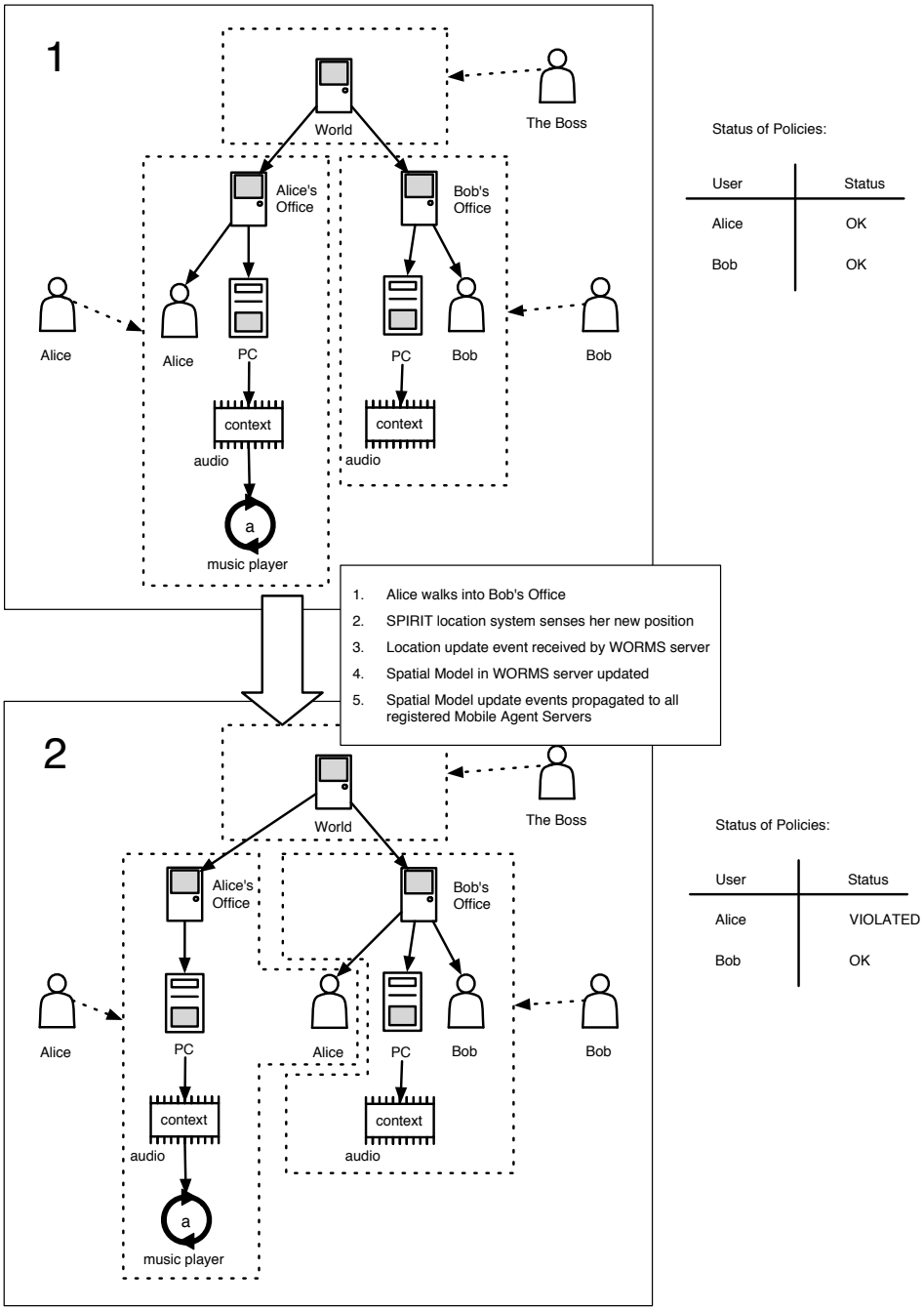


Figure 7.5: [*1*]: initial state of the WORMS system; the dashed arrows and dashed boxes indicate the owners of each entity. The table on the right indicates that two users' policies are both satisfied. [*2*]: the state of the WORMS system after Alice has been sensed moving into Bob's Office. Note her music playing agent has not moved and her policy is now being Violated.

Bob's in violation and therefore it must apply the conflict resolution policy to decide which policy should take priority.[2]

The default conflict resolution strategy was outlined back in Section 4.5. Each user is considered to "vote"[3] on a proposed change to the world model. A user is said to either be *for* the change, *against* against the change or said to *abstain* in the following circumstances:

for: *either* the change results in fewer of the users policies being violated *or* it results in no change to the number of the users policies being violated and the request happens to come from an agent they own;

against: if the change results in more of the users policies being violated; and

abstain: if the change results in no change to the number of the users policies being violated and the request comes from an agent they do not own.

A migration is considered to consist of two phases: the first when the agent leaves its old context and the second when it arrives at its new context. The request is only allowed if both phases are allowed. For the case of Alice's music player attempting to migrate into Bob's office, each phase will be considered separately.

Consider the first phase of the migration; i.e. when Alice's agent leaves Bob's office. Alice's single policy is violated if the agent stays where it is and holds if the agent disappears; therefore Alice votes for the change. Consider Bob: his policy is not violated before or after this change and since the agent does not belong to him he abstains. Since there is one vote for and one abstention there is no need to deliberate any further and the change is accepted.

Consider the second phase of the migration; i.e. when Alice's agent arrives in Bob's office. Alice's single policy holds if the agent is absent and holds if the agent is in Bob's office; therefore since the number of policies in violation is unchanged she votes for the change, because the request comes from her agent. Bob's policy, however, is not violated before the change and would be violated afterwards; therefore he votes against the change. For this phase of the migration

[2]The conflict resolution is needed because Bob's policy would be violated by the proposed move. If Bob's policy was agnostic then the conflict resolution would not be needed.

[3]Of course no actual user polling takes place, just an examination of their policies

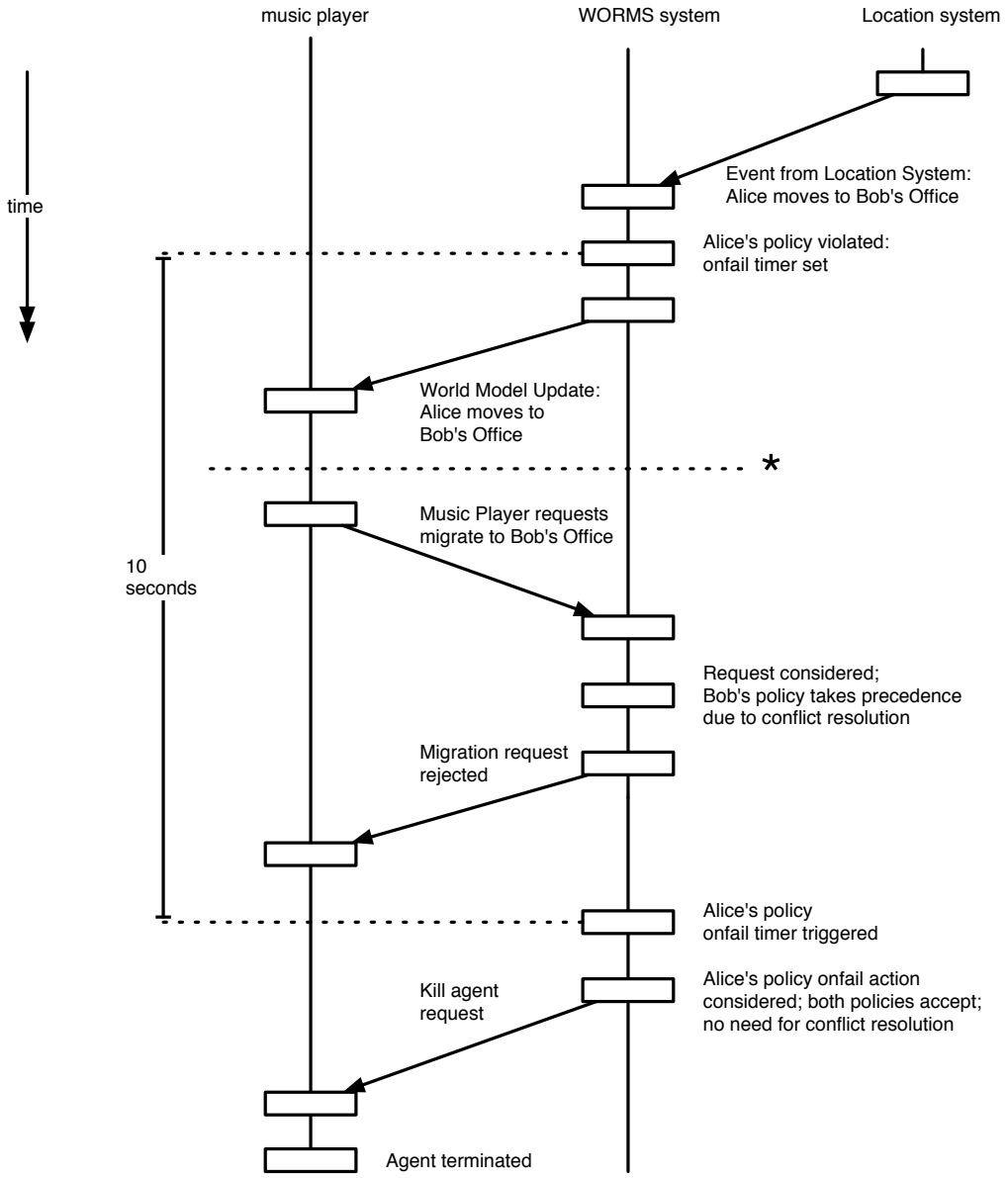


Figure 7.6: Chart showing the sequence of events which happen when Alice walks into Bob's office and remains there. Time runs vertically downwards. There are three components involved, from left-to-right: the music playing agent running on a PC in Alice's office; the WORMS system running on a centralised server; and the location system running on a centralised server.

there is one vote for (from Alice) and one vote against (from Bob) and therefore we must consider the owners of the relevant spaces.

Recall that the conflict resolution strategy adds weights to votes depending on the owners of the space where the proposed change will occur. Since the proposed change is Alice's agent arriving on Bob's PC the WORMS system examines who owns the entities on the path between the root and Bob's PC's `audio` context. There are two relevant owners: "The Boss" who owns the root entity and Bob who owns his office and the equipment inside. Since "The Boss" has not entered any policies into the system he is ignored. Bob, of course, votes against the migration. Alice's votes are ignored because she does not own any of the relevant entities. Therefore the conflict resolution system decides that Bob's wishes take precedence and the migration request is to be blocked.

Returning to the chart in Figure 7.6, a message is sent back to the agent from the WORMS system telling it that its migration request has been rejected. At this point we assume the agent continues to monitor Alice's location and if she moves somewhere else it will attempt another migration. Assuming Alice stays where she is, a short time later the 10 second timer expires and the $onfail$ action is triggered and the WORMS system considers executing the command to kill the music player agent. Recall that a kill consists of only one phase: removing the entity from the model. The WORMS system evaluates this change in the same way as the previous migration proposal and concludes that it will be accepted—this is due to the fact that Alice's policy will cease to be in violation and Bob's policy holds regardless. Therefore the WORMS system issues a command to kill the music player entity. This request is translated by the Aglet plugin into a request to the mobile agent server on Alice's PC to terminate the music playing aglet. The last step in the chart of Figure 7.6 shows this message being received and the resulting agent termination.

7.1.2 Mobile Agent Servers

This section describes the code running inside the client-side mobile agent servers. In the diagram of Figure 7.1 three modules were displayed: (i) the *Manager*; (ii) the *Mobility Policy*; and (iii) the *Untrusted Agent Code*. Each of these shall be described in turn.

The *Manager* module talks directly to the Aglet plugin. On startup its first task is to register entities in the model which correspond with instances of `AgletContext` in the local virtual machine. Whenever an agent wishes to migrate, the Manager sends the request to the Aglet plugin. The installed policy in the middleware service is queried to determine whether the request should be accepted or not. The Aglet plugin can also send requests to the Manager, telling it to destroy or jail an agent in response to a policy violation (recall that policy violations can result from unrestrained physical movement).

The policies in Chapter 4 are mobility *restriction* policies designed to limit the physical movements of mobile agents. The *Mobility Policy* in Figure 7.1 refers to another kind of policy: the *mobility enforcement policy*. These policies are written directly as Java classes and a number of example policies are described in the following section.

The rest of the code is labelled *Untrusted Agent Code* in Figure 7.1 and comprises the application-specific part of the mobile agent code. This code is standard Java Aglet code except it has access to additional APIs (notably the Manager module) and is unable to use the standard Aglet migration commands directly; all migration requests must be filtered through the Manager first.

7.1.3 Mobility Enforcement Policy

The *mobility enforcement policies* are the dual of the mobility *restriction* policies outlined in Chapter 4. The enforcement policies decide *when* an agent should move and *where* the agent should move to. The policies are represented as Java classes using a few simple APIs as exhibited by the following pseudo-code fragment:

```
public class SimplePolicy
       implements EntityChangeListener{
  ...
  public SimplePolicy(WORMS spatial_model,
                      Manager manager,
                      Entity me) {
    spatial_model.addCallback(this);
    ...
```

```
  }
  public entityDeparted(who, where){ ... }
  public entityArrived(who, where){ ... }
  ...
}
```

where `spatial_model` represents a reference to the WORMS system and the `addCallback` method registers the `SimplePolicy` instance as an `EntityChangeListener`. When the world model changes, the system invokes the `entityDeparted` and `entityArrived` methods when entities leave a place and arrive at a new place respectively. Note these correspond to the two primitive events described earlier in Section 3.2. Note also that although this API causes the agent to receive *all* change events in the model an API could be added to request only events within a particular subtree of the model to enhance scalability.

Since the policies are abstract, a library of common patterns can be created to encourage code sharing between applications. The following two sections provide examples of how such common mobility policies might look.

Example: The Follow-Me Policy

The "Follow-Me" policy instructs the application to physically follow a particular user around. Whenever the user leaves a room, the application stops running. Whenever the user enters a new room, the application migrates to a host in the new room and continues where it left off. This mobility policy is useful in a number of contexts, for example

- migrating a user's desktop to a terminal near their current physical location (recall the teleporting example in Section 7.1.1) to ensure that user applications are always easily accessible; and
- ensuring that a music playing program is always playing music that the user can physically hear, by migrating to a computer in the same room.

Java-like pseudocode for the Follow-Me policy may be written as follows:

```
public class FollowMe{
```

```
  ...
  public FollowMe(WORMS spatial_model,
                  Manager manager,
                  Entity me) {
    spatial_model.addCallback(this);
    ...
  }

  Entity findNearbyComputer(Entity x){
    // returns an entity near "x"
    // capable of running the
    // application
  }

  public entityDeparted(who, where){
    if (who.equals(me)){
      manager.stop();
    }
  }
  public entityArrived(who, where){
    if (who.equals(me)){
      Entity x=findNearbyComputer(where);
      manager.migrate(x);
    }
  }
}
```

In the constructor the `FollowMe` object registers itself with the WORMS. When entities change location in the spatial model the two callback methods `entityDeparted` and `entityArrived` are executed on the policy object. Note that in the Follow-Me policy only events relating to the user being tracked are relevant and all other events are silently ignored. When the tracked user leaves his/her current room, the `entityDeparted` method calls the `manager.stop()` method which requests that the application stop execut-

ing. When the tracked user enters another room, the `entityArrived` method attempts to migrate the application to a new computer, located inside the user's new room.

Example: The When-No-one's-Here Policy

This policy is useful for running long-term non-interactive tasks on otherwise idle computers. Whenever a user enters a room with one of these applications, the application is paused to prevent any possible degradation of the quality of service to the user. Only when the last person leaves the room is it safe for all such background tasks to be restarted.

Java-like pseudocode for this policy may be written as follows:

```
public class WhenNooneHere{
...
  public WhenNooneHere(WORMS spatial_model,
                       Manager manager,
                       Entity where) {
  spatial_model.addCallback(this);
  ...
  }

  int countPeopleHere() {
    // return number of people in
    // same room as me
  }

  public entityDeparted(who, where){
    if ( (manager.amStopped()) &&
         (countPeopleHere() == 0) )
      manager.start();
  }
  public entityArrived(who, where){
    if ( (manager.amRunning()) &&
         (countPeopleHere() > 0) )
```

```
      manager.stop();
  }
}
```

The utility function `countPeopleHere` returns the number of people in the same room as the application. The `entityDeparted` and `entityArrived` methods use this information to decide whether or not to let the application execute.

7.1.4 Issues

In the prototype system described in this section there are several outstanding issues which should to be addressed before the system is widely deployed. The first issue concerns the size of the *trusted computing base* (TCB). In computer security terminology, the TCB is defined to be the set of hardware and software components which enforce the security policy. In this implementation, the TCB comprises

1. the location system;
2. the servers running both the WORMS middleware system and the mobile agent servers;
3. the network connecting the hardware components together;
4. the code running in the WORMS middleware system;
5. the Java virtual machine; and
6. the Manager module in the mobile agent servers.

Some of these dependencies are hard to avoid; for example it is difficult not to rely on the physical security of server machines and difficult not to rely on the software (Java VM, WORMS middleware, Aglet libraries and the Manager module) being written in a secure fashion, free of application-level vulnerabilities. However, the two remaining dependencies, namely the location system and the network, are particularly problematic.

The design of the Active Bat system, described earlier in Section 2.2.2, was focused on achieving maximum performance in terms of location accuracy and update latency in a shared, trusted environment. In particular it was not designed for high integrity and availability in the security sense (integrity and availability were first discussed in Sections 2.5.2 and 2.5.3 respectively) and there are a number of possible attacks. A simple method to prevent an object being tracked (assuming the Active Bat is physically attached to the object) is to cover the ultrasound emitters on the top of the bat, preventing any signal getting through. Another possible attack relies on the fact that the ultrasound signals are narrowband – it is possible to prevent any sightings in a room by flooding the room with ultrasound continuously[4]. Neither the radio channel nor the ultrasound channel use any kind of integrity checking and therefore it is possible for a more sophisticated attacker to forge false sightings, completely manipulating the system. These problems can only be solved properly by redesigning the location system with measures to ensure integrity and availability. A move to wide-band (e.g. spread spectrum) signals would increase the difficulty of jamming the signals and cryptography could be used to guarantee the integrity of messages.

As well as relying on the Active Bat, the network connecting the components together is also part of the TCB. In the prototype system, the network is a shared ethernet network. The components can use standard cryptography to guarantee the secrecy and authenticity of messages but, due to the properties of the underlying ethernet network. the system cannot guarantee the messages are ever delivered. In the same way an attacker can prevent Active Bat sightings by flooding the ultrasound channel with noise, an attacker can prevent messages being delivered by flooding the network with data. There are two approaches which would ameliorate the situation. The first approach is to redesign the network so that it can make some reliability guarantees – a form of Quality of Service (QoS). Alternatively the set of policies allowed in the system could be changed. Rather than assuming that the movements of entities which lead to a policy violation are observed, we could instead assume that we need *proof* that the policy has not been violated, otherwise corrective action is taken. Such proof could take the form of a message generated

[4]Jangling keys or using a hoover are both proven techniques for jamming Active Bat ultrasound signals.

and signed by the location system saying "entity X was seen in entity Y at time T". The absence of any such guarantee within a certain time period could result in corrective action being taken, such as the termination of an active agent. Applications could be constructed in a robust fashion so that loosing an agent does not unduly affect the task being performed. In such a scenario, unnecessarily killing agents might be an acceptable solution.

There are a number of additional issues stemming from the choice of implementation language: Java. Java has the following (mis-)features:

- the inability to introspect the call stack of a running thread;
- the inability to kill (sometimes known as *cancel*) a running thread; and
- the security model for governing access to resources.

The first feature – the inability to introspect the call stack of a running thread – leads Java to only support weak mobility (see Section 2.1). All agents must support `stop()` and `start()` methods which provoke them to voluntarily serialise and deserialise their own execution state. It is not possible to freeze another thread and read its method call stack. The second feature – the inability to cancel a running thread – is a deliberate limitation to prevent threads being terminated while still holding locked resources, which would then become permanently locked. The consequence is that agents must respond to commands to shut themselves down and an aberrant agent cannot be forcibly killed, without restarting the virtual machine. The last feature concerns the standard Java security model for controlling access to resources. Java APIs grant permissions to resources by withholding or transmitting references to otherwise hidden objects. Once a reference is given it cannot be taken back or cancelled. If a music playing agent is frozen and the system wishes to prevent the agent from playing sound, there are two implementational possibilities. Either the agent should voluntarily give up the reference to the sound playing object or the system should interpose a proxy object which checks access permissions for every call. Both options have drawbacks; the first relies on cooperative agents while the second alters the semantics of APIs by adding extra failure modes (e.g. the command to play a sound might fail during the `play()` call but after the sound device is opened successfully).

7.2 Interfaces

The previous section dealt with enforcing *mobility* policies on applications while this section focuses on imposing policy on application *interfaces*. The policies are implemented by means of a web-based firewall called SPECTRE, the design of which is described in Section 7.2.1. SPECTRE is a sophisticated and modular firewall which allows policy modules for SPDL-2 (described in Section 7.2.2) and SWIL (described in Section 7.2.4) to be loaded dynamically. Although the implementation of SPECTRE is a research prototype, encouraging performance results are provided which demonstrate that the approach described here is feasible.

The rest of this section is structured as follows: Section 7.2.1 describes the structure of SPECTRE followed by details of the SPDL-2 implementation in Section 7.2.2. SPDL-2 policies are stateless whereas SWIL policies are inherently stateful and require state tracking. Section 7.2.3 describes how SPECTRE can be used to track sessions and Section 7.2.4 describes the implementation of SWIL policies. Finally, some results and discussion are presented in Section 7.2.5.

7.2.1 SPECTRE: A Policy-Enforcing Firewall

SPECTRE is an application-level firewall used to enforce interface policies dynamically. SPECTRE is written in O'Caml on top of a custom HTTP processing library. It receives HTTP requests intended for the back-end application server and parses the headers, including parameters and cookies. Once the request has been parsed, the firewall reads the URI and decides which module the request should be directed to. In the diagram of Figure 7.7, requests are initially directed towards both module `m1` and module `m3` in turn. Each module may choose one of the following three actions: (i) accept the request for processing; (ii) reject the message as invalid; or (iii) opt not to process the request. A rejection from a module terminates the processing of the request. If the request is accepted for further processing then it may be modified before being passed on to further modules. For example, in Figure 7.7, if module `m1` accepts a request it may be modified before being passed on to module `m2`. A module may opt not to process a request e.g. in Figure 7.7 the module `m3` may opt not to process a request in which case it will be offered to module `m1`. If no module accepts the request then the request is rejected, even if no module explicitly rejected it.

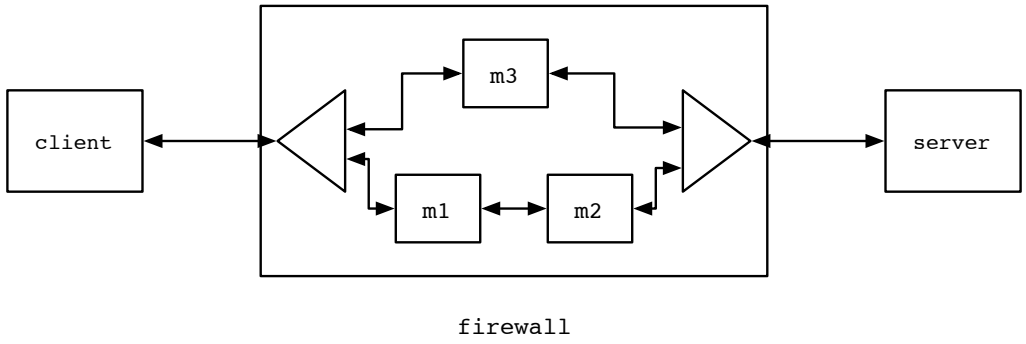


Figure 7.7: Structure of the SPECTRE Policy-Enforcing Firewall with three modules loaded: `m1`, `m2` and `m3`.

The firewall is intended to be *failsafe* – if a request is not explicitly accepted with an installed policy module then it will be rejected (i.e. the firewall operates a *default-deny* base policy). By forbidding all URIs that do not match those explicitly in the installed policies the firewall prevents an attacker from using obscure, non-standard URI encoding techniques to circumvent the protection (thus avoiding attacks of the kind recently used on Cisco's Intrusion Detection System [57]).

Once requests have been successfully processed by all relevant modules the request is forwarded to the back-end application server. HTTP response messages received are again filtered through the policy modules which perform any necessary transformations before the response is finally returned to the original user.

7.2.2 SPDL-2

The *SPDL-2 Compiler* takes an SPDL-2 specification (as described in Section 5.4.1) and compiles it into a policy module for execution on the firewall. Validation rules and constraints are compiled into both O'Caml for dynamic execution and Javascript which can be embedded into forms and executed on clients. Generating client-side form validation Javascript code is not always desirable – many applications already have their own, custom Javascript for this purpose – and therefore Javascript generation can be turned off on a `<policy>`-wide basis (by setting the `javascript` attribute of the `<policy>` tag to '`N`'). The SPDL-2 DTD may be found in Appendix A.

The policy module operates as follows: Once a valid URI has been detected in the HTTP-Request, the module examines the names of all parameters and cookies

present. If unexpected data is present, or expected data is absent then the error is logged, the request is blocked and a message returned to the user indicating the request was rejected. The next step is the checking of parameter type and length constraints. Assuming those tests pass, any required message authentication code is verified and the validation and transformation code is applied.

Transformations should be *total* functions on strings. A badly-written transformation which generates an exception will cause the request to be aborted and an error message to be returned to the client. Additionally, observe that if the policy document requires the insertion of client-side Javascript or if the policy indicates that it might contain a link to a MAC-protected URI then the firewall must process the HTTP response returned from the web-server. The HTML is parsed and rewritten in order for the appropriate validation code (pre-generated by the policy compiler) to be added to forms (see Section 7.2.2). Message authentication codes are also inserted to protect form fields and URI-parameters from malicious client-side tampering (see Section 7.2.2).

Client-side Form Validation

Whenever requested by the policy document, the module inserts Javascript code to perform client-side validation checks. (Recall that the insertion of Javascript is only intended to enhance usability – data is always rechecked by server-side code to avoid the kind of Form Modification attacks described in Section 5.2.1). The inserted code checks most of the SPDL-2 constraints: types, lengths and all custom constraints written in the validation language. The resulting program is inserted into the `onSubmit` attribute of a `<form>` tag unless such an attribute is already present – any already present validation code is considered to take priority over the generated code.

Message Authentication Codes

Recall that SPDL-2 policies allow URI parameters to be marked indicating that they must only receive data which is accompanied by a *Message Authentication Code* (MAC) [143] generated by the firewall. As HTML is sent to the client, the firewall annotates the appropriate parts with MACs; these MACs are checked when the form data is submitted. This mechanism is used to prevent the Form Modification attacks described in Section 5.2.1.

The implementation uses the HMAC [20] algorithm to generate MACs by securely combining the protected data with a secret key and a timestamp. In this way an attacker is unable to generate new MACs without first finding out the secret key. A major danger still facing this system is avoiding *replay attacks* [158] where clients replay messages already annotated with MACs in unexpected contexts. Two steps are taken to avoid such attacks:

1. A time-stamp is included in the MAC to guarantee message *freshness*.

2. Rather than generating separate MACs for each individual protected field, a single MAC is generated for all protected client-side state bundled together. This protects the integrity of the whole message, protecting against cut-and-splice attacks (in which MAC-annotated fields are swapped into other messages).

Both steps are SPDL-2 mechanisms which require storing no session state in the application-level firewall. However these mechanisms are not foolproof. For example, in the case study originally presented in Section 5.5 a MAC is generated for *both* the `productID` *and* `Price` fields. Although users can replay such messages this results in multiple purchases of the same product for the *correct* price. The intention is that the MAC prevents the `Price` and `productID` being modified independently.

In this example the problem of multiple identical purchases can be partially rectified by using a SWIL policy in addition to a SPDL-2 one. Such a policy could take many different forms. At one extreme, a very simple SWIL policy could prevent the user submitting two consecutive requests to the same sensitive URI by accident. A more complicated policy could use a native function call to check the request against a database containing previous requests, to prevent duplicates. Note this second case may require large amounts of storage space; however applications like the hypothetical e-commerce application would probably already store the required information anyway (e.g. in a transaction log).

SPDL-2 requires that HTML pages fetched from URIs that are contained in a single `<policy>` block may only contain links to MAC-protected URIs[5] which

[5] A MAC-protected URI has at least one `<parameter>` with its `MAC` attribute set to '`Y`'.

are found in the same `<policy>` block (or in a nested `<policy>` block). By forcing the application's URIs to be partitioned in this way an important performance optimisation is facilitated which is discussed in Section 7.2.5 as well as encouraging the policy designer to structure their policy for ease of future maintenance.

7.2.3 Tracking Session State

The SPDL-2 policies described in Section 7.2.2 may be considered *stateless policies* because they can always be implemented by a stateless firewall. In contrast, SWIL policies which describe the acceptable order of URI invocations have an implicit notion of state and therefore cannot be implemented by a stateless firewall[6]. This section describes a firewall module which provides the necessary state handling.

Since HTTP is itself a stateless protocol, it is necessary to manually create a session on top of the HTTP protocol using one of the techniques discussed in Section 2.4.5. Once a session is established, state associated with the policy can be associated with it. When processing requests and responses, the session information can be retrieved from the HTTP messages and any associated state (stored within the firewall) can be modified.

The session state tracking module creates a session over HTTP using client-side state via specially-named cookies. The module does not impose any policy of its own in the sense that it never *rejects* any requests or responses; it only *transforms* responses by adding the special session tracking cookies. Note that the session established by the session tracking module is completely orthogonal to any sessions established by the application itself. Each such session would use a different cookie name.

The session tracking module operates in the following way. Upon receiving a request the session tracking module examines all cookies within the HTTP message. If the specially-named session tracking cookie is absent (indicating the need for a new session) then a new session identifier is created and the cookie added to the request. This modified request is then passed to other policy modules which

[6]Unless the policy was so simple it only had a single state, in which case it could be implemented on a stateless firewall.

can use the session cookie to store state associated with the session. When the session tracking module receives the response back from the application (possibly after being transformed by other policy modules) the session cookie is added again to the response. This scheme relies on the user to have cookies enabled in their browser; if a user does not have cookies enabled then every request will cause a new session to start.

Session lifetimes

Sessions are not eternal; the maximum lifetime of an idle session is a configurable parameter of the tracking module. The time of the last request is stored with every session and those older than the maximum lifetime are culled. If a user attempts to restart a deleted session then a new session is created for them and they must restart their interaction with the application from the beginning. This is a similar tactic used by many web-applications and by other systems such as NAT routers. The exact choice of maximum lifetime value depends on the nature of the application and reflects a trade-off between convenience and security. A long lifetime is convenient for a user who can return to the application after a lengthy period of absence while a short lifetime guards against a user forgetting to logout, leaving a terminal and allowing a malicious user to continue their session.

Auditing

Tracking user sessions in an application-level firewall allows the firewall to perform *auditing* in which it records each user's interactions with the application. The audit log can be used to answer the question, "how did the user get here" by displaying the user's path through the application. This is especially important since a user will likely not be able to simply bookmark their current place in the application and expect it to work indefinitely; once their session times out they will have to re-enter the application from the beginning.

Many web-applications already individually contain code to display "bread-crumbs" – visual hints of the path taken by the user through the application. This function may be offloaded to the application-level firewall and the code shared across multiple applications. Auditing is also very useful in the case of error conditions. A user can be told how they managed to trigger the error so that hopefully they will not repeat their mistake again.

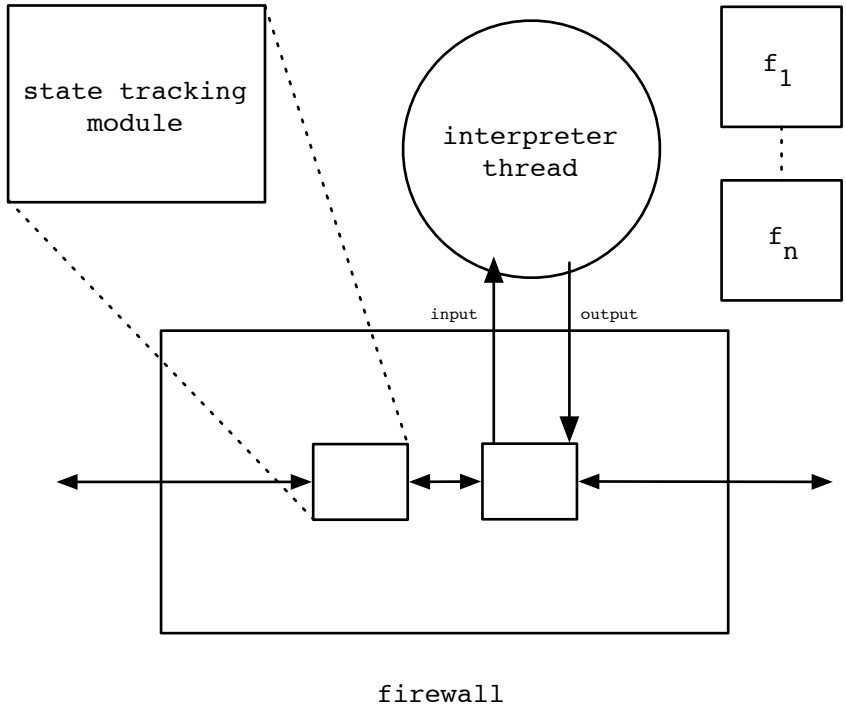


Figure 7.8: A configuration of the application-level firewall with two modules arranged in a pipeline. The first module is the state tracking module from Section 7.2.3 while the second is the SWIL module from Section 7.2.4.

7.2.4 SWIL

The SWIL language, described in Chapter 6 allows the acceptable order of URI invocations to be defined as a sequence accepted by a deterministic finite state automaton. The implementation consists of a firewall policy module stub which communicates with a SWIL interpreter running in a separate O'Caml thread. The diagram in Figure 7.8 shows a typical configuration of the firewall. Notice that the session tracking module must be on the data-path before the SWIL interpreter since the SWIL interpreter relies on having access to user session state information. The SWIL policy module is connected to the interpreter thread by a pair of synchronous O'Caml channels represented by arrows and external functions (mapped to O'Caml functions) are represented by a set of square boxes. More details of the SWIL interpreter functionality were given back in Section 6.5.

On receiving a request the policy module creates a SWIL request object: a

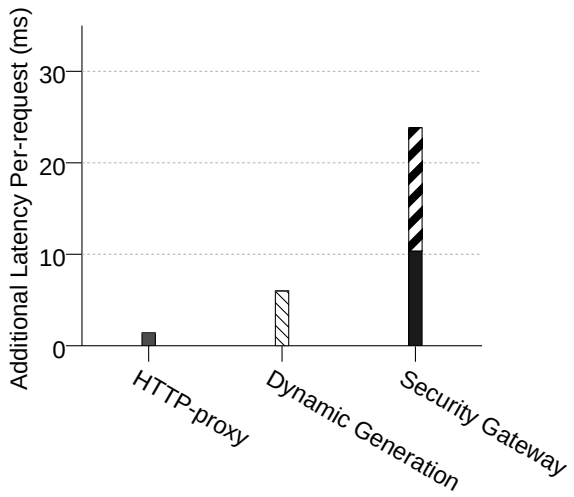


Figure 7.9: A comparison of the latency of the system with latencies incurred in common types of HTTP processing. Note the third bar has been subdivided into two parts; the lower bar represents latency due to message buffering while the upper bar represents latency due to parsing and rewriting HTML.

record consisting of every field mentioned by a call to `GetField` in the SWIL program. This data is sent via the synchronous channel to the interpreter thread which responds by sending back a boolean value on the output channel: true if the request was accepted and false if the machine has entered the error state.

If the module has been configured to perform dynamic response transformation (described in Section 6.5.1) then, on receiving a response message the module parses the HTML document to extract all the links and forms. These are then communicated to the interpreter thread which decides if any should be deleted or greyed out. The HTML is then rewritten with all the offending links and forms modified.

7.2.5 Raw Performance

In this section basic performance results of the prototype SPECTRE system are presented focusing mainly on measurements of latency and throughput.

Figure 7.9 shows the worst-case latency of the firewall prototype and compares it to the latency of other common types of HTTP processing. The results were measured by fetching the homepage of the Laboratory for Communication

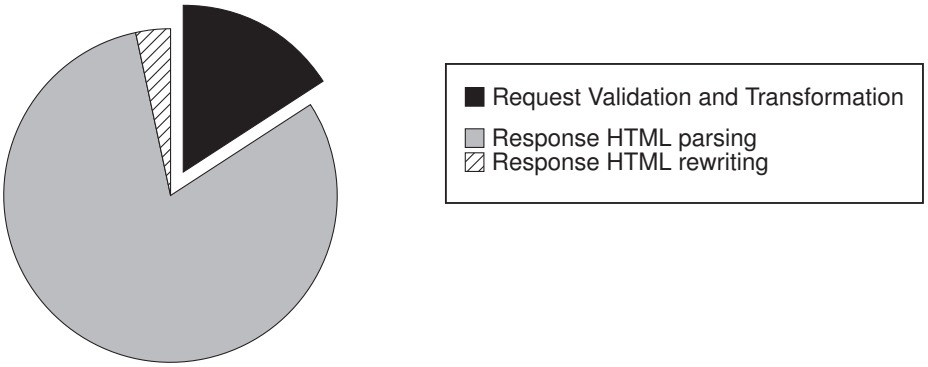


Figure 7.10: A breakdown showing the relative cost of HTTP processing stages within the application-level firewall.

Engineering (University of Cambridge)[7] augmented with the web-form described in the case-study of Section 5.5. The leftmost bar shows the latency added by a Squid [8] proxy cache when fetching a statically compiled version of the page; the middle bar shows the added latency of dynamically generating the page using PHP and a MySQL [5] back-end; the rightmost bar shows the latency of using the firewall to enforce the security policy found in Appendix B. The final bar is divided into two sections: the (lower) solid black section represents the latency due to buffering the HTTP messages; the (upper) striped section shows the latency due to parsing the HTTP messages and annotating the HTML with MACs. Figure 7.10 shows the relative cost of processing HTTP requests and HTTP responses in the case of the example web-form. Note that the total processing time is dominated by the HTML parsing stage.

The latency of the system appears large compared with the latencies incurred in proxy caching and dynamic page generation. To some extent this is due to the fact that the naïve implementation is a proof-of-concept prototype which is completely unoptimised, based upon an inefficient non-pipelined HTTP 1.0 library. However, it is recognised that the complexity of the application-level tasks performed by the firewall will necessarily incur more latency than the lower level manipulation performed by proxies such as Squid. Future potential optimisations include (*i*) using a specialised HTML parser to concentrate only on relevant parts of HTML syntax (currently a general HTML parser is used which performs a great

[7] `http://www-lce.eng.cam.ac.uk/`

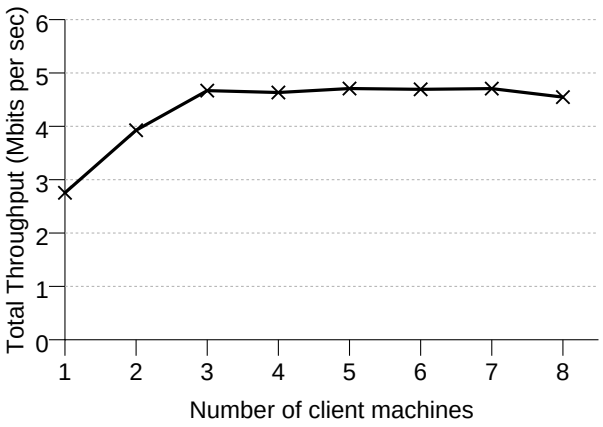


Figure 7.11: Total throughput of a single firewall as the number of concurrently connected clients varies.

deal of unnecessary work); (*ii*) reducing latency by streaming the HTTP messages and processing them on-the-fly whenever possible; (*iii*) writing speed critical parts of the firewall directly in C or implementing in hardware.

Figure 7.11 shows how the total throughput of a single firewall varies as the number of concurrently connected clients increases. Each client is running a single threaded application continuously requesting the test URI. The machines are all connected on a fast 100Mbit/s switched network and the measurements were taken running the firewall on a dual Intel P-III 500 MHz during an off-peak time when the network was idle. The throughput quickly reaches a maximum value as the CPUs become saturated. Note that it took 3 clients to saturate 2 CPUs probably because of lack of support for persistent connections in the HTTP library used. Again, it is claimed that optimising the code for performance and running the filter on a higher spec machine would yield a significantly higher maximum throughput.

In the case of policies like SPDL-2 which are inherently stateless one may increase throughput linearly simply by deploying multiple replicas and using a load balancing scheme[8] to distribute work between them (see Figure 7.12). (Note that stateful systems do not scale linearly in this way since, ultimately, the centralised state becomes a bottleneck across the cluster.)

[8]This assumes that load balancing can be achieved cheaply, e.g. by round-robin DNS.

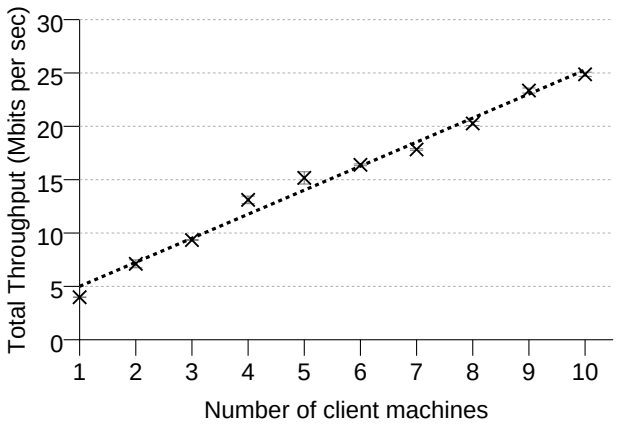


Figure 7.12: Total throughput of a cluster of firewalls as the number of concurrently connected clients varies.

The measurements presented are *worst case* in the sense that the test policy and the test HTML were both complicated and required extensive processing within the firewall. Both SPDL-2 and SWIL have features which are particularly expensive. SPDL-2 has a number of expensive features like MAC annotation and insertion of client-side Javascript. In the first version of SPDL [146] turning on the MAC facility for *any* URI would have necessitated processing the HTML output of *every* page. By requiring that URIs within a single `<policy>` block do not reference MAC-protected URIs outside that block (see Section 5.4.1), the new system allows the developer to specify the set of application URIs which make use of MAC-protected client-side state and, critically, *those which do not*. URIs with no MAC or Javascript requirements can be processed much more efficiently by the firewall as the HTTP response from the web-server can be streamed to the client directly: it does not have to be parsed or re-written. Given that processing the HTTP response is by far the dominant performance cost incurred by the firewall (see Figure 7.10) significant speedups are achieved.

Similarly SWIL policies comprise both an expensive part: response transformation (see Section 6.5.1) and a cheap part: request filtering. Only the request filtering is necessary for security; the response transformation exists solely to improve the user interface. Therefore if performance is a problem the expensive response transformation can be disabled.

Furthermore, it is believed that many HTML pages are simpler than the contrived example and have fewer security constraints (i.e. shorter pages without form parameters), leading to better *average case* performance. For example consider that many of the HTTP messages would contain graphics and hence would not require any processing at all. A performance-optimised firewall could examine the content-type header of HTTP responses, using streaming instead of buffering if no HTML processing is required.

In summary, the performance figures presented in this section relate to an unoptimised prototype under worst-case conditions. The system is designed to be scalable and optimisable and therefore the techniques represented are claimed to be applicable in practice.

7.3 Summary

This chapter described the implementations of the two important types of policy described in this thesis: mobility policy and interface policy. Mobility policy consisted of both restriction policy and enforcement policy, limiting and directing the movements of agents within an environment permeated with location-sensors. The mobility policy system comprised two main elements: (i) a new form of spatial middleware designed to be independent of the underlying choice of location-system and policy type; and (ii) client-side support code based upon the Java Aglet mobile agent system. The system was implemented and a number of issues discussed.

The interface policies were implemented through a modular application-level firewall called SPECTRE. SPECTRE supports both stateless policies like SPDL-2 policies as well as stateful policies such as SWIL policies. SPECTRE was written in O'Caml on top of a custom HTTP processing library. The SPECTRE system was implemented and a number of encouraging performance results were presented.

CHAPTER 8

Conclusions and Further Work

Chapter 1 stated that:

> The thesis of this work is that abstracting application-level security policy (which we defined to be policy written to govern both the *mobility* and *interface behaviour* of systems) is an effective technique for guarding against application-level vulnerabilities which would otherwise plague future ubiquitous computing systems.

It is time to critically evaluate this statement. We proceed by breaking down this thesis statement into two sub-claims:

1. Abstracting application-level security policy is an effective technique for guarding against application-level vulnerabilities; and

2. The application-level vulnerabilities referred to in (1) above would otherwise plague future ubiquitous computing systems.

We begin by addressing claim number 1: that the policy systems presented in this thesis are an "*effective technique for guarding against application-level vulnerabilities*". Each of the policy systems presented in this thesis in Chapters 4 through 6 were presented along with realistic examples, many of which are based

on actual vulnerabilities found "in the wild" and reported in the IT press. As each policy language was presented it was demonstrated exactly how it could be used to prevent particular vulnerabilities. Details of a prototype implementation complete with encouraging performance results was presented in Chapter 7. Since the technique of abstracting application-level policy was shown to be workable both in theory and in practice we claim the technique is *effective* and that sub-claim 1 holds.

Furthermore we argue that *abstracting* security policy is a important concept in itself. Keeping policy specifications separate from implementation code allows the policies to be easily analysed (c.f. the use of SPIN in Section 6) and also serve as valuable de-facto documentation. In many cases software is built from a mixture of "off the shelf" and custom components, some only available in binary form. It may not be possible or feasible in such systems to modify the source code directly; in these cases abstracting security policy is the only option.

Sub-claim number 2 above claims that the application-level vulnerabilities addressed by Chapters 5 through 6 would otherwise "*plague future ubiquitous computing systems*". This claim is justified in two further sub-claims:

1. Future ubiquitous computing systems will be constructed from large numbers of physically and logically mobile processes (agents) which will make existing protection mechanisms – network and transport-level firewalls which together form the *Maginot Line* of Internet security – completely redundant.

2. The application-level vulnerabilities addressed by the policy languages presented in this thesis will be present in future systems irrespective of the exact choice of underlying protocols.

Consider the first claim that existing protection mechanisms will become redundant. This claim is justified with arguments from Chapter 1, specifically that

- code will become more mobile as the gulf between processing speed of individual nodes and latency of inter-node communication widens;

- more devices will be physically mobile as manufacturing processes improve and embedded devices become networked;

- higher-level protocols (like Web Services) tunnel over other protocols, enabling them to penetrate network-level firewalls;
- the bulk of code in a system – and hence the majority of bugs – is at the application-level.

Section 1.6 argued that current firewalls are inadequate because (i) increasing mobility and protocol tunnelling allows code to bypass the points at which the firewall rules are applied; and (ii) increasingly vulnerabilities will be discovered and exploited at the application-level, where network and transport-level firewalls have no effect.

Consider the second claim that the application-level vulnerabilities addressed by policy languages described in this thesis are independent of the exact choice of underlying protocol. This claim shall now be justified separately for the mobility and interface policy languages.

The mobility policy language in Chapter 4 was implemented on top of Java Aglets but only relied upon two features which are present in all existing mobile agent systems. The first feature is the ability to have multiple named locations co-existing on the same host, either as part of the same agent server or by running multiple servers on different ports. The second feature is the ability to interpose a policy checking module between mobile agents and their running environment. We claim that these abilities are core features that will be present in all future mobile agent systems. The enforcement mechanism described in Chapter 7 was built in a centralised fashion: one server machine maintained the authoritative model of the world and the database of policies. This server, which is a client of the SPIRIT location service which is also centralised, was used to authorise all agent migration requests and to take corrective action against agents which were in violation of policies. This centralised security policy enforcement service is similar to existing access control mechanisms such as NIS under Unix and Active Directory under Windows. None of these systems are designed to be used over the open Internet but rather within individual organisations or companies. Further work would be needed to investigate ways of sensibly distributing the enforcement mechanism if operation over the Internet was required.

The interface policy languages SPDL-2 in Chapter 5 and SWIL in Chapter 6 are both languages designed to protect vulnerable web-applications i.e. applications running over HTTP. However, the vulnerabilities they serve to protect against – specifically validating requests with respect to type specifications and temporal automata – are general concepts which apply to any RPC-based distributed system. Although the concrete details of the protocols may change (e.g. to Web-Services, XML-RPC or even CORBA), provided the same types of applications are in use, the vulnerabilities will remain and languages like SPDL-2 and SWIL will still be useful. To justify this claim, we now describe the applicability of these techniques first to CORBA and then to Web-Services.

CORBA [123] allows application interfaces to be specified in an Interface Definition Language (IDL) which is then compiled into client and server code. CORBA IDL has many built-in types including `int`, `string`, `bool` and `float`. It also has support for record types, arrays and disjoint union types. CORBA has the concept of a request `Context` which is similar to the HTTP concept of a cookie. CORBA's built-in types negate the need for the simpler SPDL-2 typechecking. However it lacks anything as sophisticated as the arbitrary validation and transformation expressions of SPDL-2 and has no support for any kind of stateful interface checking. Support for a variant of SPDL-2 and SWIL could be added to a CORBA IDL compiler and used to generate "proxy classes" to enforce these policies dynamically.

"Web-Services" is the name given to a suite of protocols based around exchanging XML documents. The Web Service Description Language (WSDL) [40] allows the specification of application interfaces in a similar fashion to CORBA IDL. The XML documents exchanged are typed (e.g. by XML-Schemas [59]) and, like in the case of CORBA, this typing subsumes the simpler SPDL-2 typechecking. However, also like CORBA, there are no arbitrary validation or transformation expressions and there is no support for any kind of stateful interface checking—therefore SPDL-2 and SWIL are still useful to applications written in a Web-Services world.

8.1 Contributions

In summary this thesis provided the following contributions:

1. the Unified Spatial Model (USM): a spatial model capable of representing simultaneously the physical locations of objects such as people and the virtual locations of mobile agents. The USM is the foundation of the Mobility Restriction Policy Language (MRPL) – a language which allows policies to be written to limit the movements of mobile agents in an environment augmented with location sensors;

2. the language SPDL-2 which allows the expression of application-level interface security policies in which per-request and response validation and transformation rules can be written to protect web-applications from many kinds of common security vulnerabilities; and

3. the language SWIL which allows the flow of control across an application interface to be described as a sequence of operations accepted by a finite state automaton. The automaton can be both analysed statically with a model-checker and dynamically checked with an application-level firewall.

Together this model and these policy languages allow abstract application-level policy specifications to be written to protect against many important kinds of vulnerability. These application-level policy systems have been implemented; implementation details and performance results were provided.

8.2 Further Work

The systems presented in this thesis were intended to investigate ways of imposing policy to protect vulnerable future ubiquitous computer systems. Simple prototypes were constructed and analysed. In the real world things are more complex and much more effort would need to be expended to make the systems robust enough to deploy. Unfortunately this would require considerably more implementation time than is available during a PhD.

However there are a number of opportunities for future research based on this thesis:

- Using a location system as part of a security mechanism places a lot of emphasis on the reliability and integrity of the location system itself. It would be interesting to consider what could be done to make such a system

more dependable, perhaps by creating a whole new location system or by changing the types of policy which can be expressed (e.g. rather than saying "delete the data on this device if the device is located *outside* the building" we might say "delete the data on this device if we have not been seen *inside* the building for 10 minutes").

- Spatial models like that described in Chapter 3 could be used to model non-physical entities like network configurations (e.g. a secure network might be a separate entity from an insecure network) as well as more physical entities (e.g. the model could include door locks and other physical access controls).

- Although we argue that maintaining separate security policy specifications is a good thing, it would nevertheless be useful to have a means of generating automatically at least part of a policy for an application. Perhaps the simpler part of an SPDL-2 policy could be generated leaving the more complicated part to human policy designers.

- The prototype implementation described in Chapter 7 had encouraging performance characteristics. However we recognise that some applications demand extremely high performance. A major overhead of the current mechanism is the need to parse HTTP messages and HTML responses. These overheads could be eliminated if the policy could be automatically "woven" (in the sense of Aspect-Oriented Programming) into the application source code using an automatic tool. The resulting program would have the security code inserted where the HTTP and HTML messages are first generated, obviating the need to parse the messages later.

- The mobility restriction policies make use of a central server with an authoritative model of the world. It would be useful to investigate to what extent this can be distributed and combined with a distributed location-sensing system.

Bibliography

[1] The <bigwig> project. More information available from http://www.brics.dk/bigwig/.

[2] The Bluetooth Specification version 1.1. Available from http://www.bluetooth.com/.

[3] The Meta-O'Caml project. More information available from http://www.metaocaml.org/.

[4] μitron4.0 specification version 4.00.00. ITRON Committee, TRON Association. Edited by Hiroaki Takada. More information available from http://www.ertl.jp/ITRON/SPEC/mitron4-e.html.

[5] *MySQL database server*. More information available from http://www.mysql.com/.

[6] The OpenBSD project. More information available from http://www.openbsd.org/.

[7] PF: The OpenBSD Packet Filter. More information available from http://www.openbsd.org/faq/pf/.

[8] Squid web proxy cache. More information available from http://www.squid-cache.org/.

[9] The Open Web Application Security Project Top Ten. Available from http://www.owasp.org/.

[10] Vxworks®. Wind River Systems. More information available from `http://www.windriver.com/`.

[11] ADLY, N., STEGGLES, P., AND HARTER, A. SPIRIT: a Resource Database for Mobile Users. In *Proceedings of ACM CHI'97 Workshop on Ubiquitous Computing, Atlanta, Georgia* (March 1997).

[12] AMADIO, R. M., AND PRASAD, S. Localities and failures (extended abstract). In *FSTTCS* (1994), pp. 205–216.

[13] ANDERSON, R. J. *Security Engineering*. John Wiley & Sons Inc, April 2001.

[14] ANDERSON, R. J., AND NEEDHAM, R. M. Programming Satan's Computer. *Computer Science Today - Recent Trends and Developments 1000*, 426–440.

[15] ARCE, I. The rise of the gadgets. *IEEE Security and Privacy 1*, 5 (September 2003), 78–81.

[16] BAHL, P., BALACHANDRAN, A., AND PADMANABHAN, V. Enhancements to the RADAR User Location and Tracking System. Tech. Rep. MSR-TR-2000-12, Microsoft Research, February 2000.

[17] BALL, T., AND RAJAMANI, S. K. Automatically validating temporal safety properties of interfaces. In *Proceedings of the 8th international SPIN workshop on Model checking of software* (2001), Springer-Verlag New York, Inc., pp. 103–122.

[18] BARTEL, M., BOYER, J., FOX, B., LAMACCHIA, B., AND SIMON, E. Xml-signature syntax and processing. W3C Recommendation. 12 February 2002. Available from `http://www.w3.org/TR/2002/REC-xmldsig-core-20020212/`.

[19] BBC NEWS. First mobile phone virus created. Available from `http://news.bbc.co.uk/1/hi/technology/3809855.stm`.

[20] BELLARE, M., CANETTI, R., AND KRAWCZYK, H. Keying hash functions for message authentication. In *Advances of Cryptology – Crypto '96 Proceedings* (1996), vol. 1109 of *LNCS*, Springer-Verlag.

[21] BERESFORD, A. R., AND STAJANO, F. Location privacy in pervasive computing. *IEEE Pervasive Computing 2*, 1 (2003), 46–55.

[22] BERNERS-LEE, T., FIELDING, R., AND FRYSTYK, H. RFC 1945: Hypertext Transfer Protocol — HTTP/1.0, May 1996. Status: INFORMATIONAL. Available from `ftp://ftp.internic.net/rfc/rfc1945.txt`.

[23] BETTINI, L., AND NICOLA, R. D. Translating strong mobility into weak mobility. In *Proceedings of the 5th International Conference on Mobile Agents* (2002), Springer-Verlag, pp. 182–197.

[24] BETTINI, L., NICOLA, R. D., PUGLIESE, R., AND FERRARI, G. L. Interactive mobile agents in x-klaim. In *Proceedings of the 7th Workshop on Enabling Technologies* (1998), IEEE Computer Society, pp. 110–117.

[25] BIC, L. F., FUKUDA, M., AND DILLENCOURT, M. B. Distributed computing using autonomous objects. *IEEE Computer 29*, 8 (August 1996), 55–61.

[26] BOOTH, S. P., AND JONES, S. B. Walk backwards to happiness - debugging by time travel. In *Automated and Algorithmic Debugging* (1997), pp. 171–183.

[27] BORRIELLO, G., AND WANT, R. Embedded computation meets the world wide web. *Communications of the ACM 43*, 5 (2000), 59–66.

[28] BRABRAND, C., MÖLLER, A., RICKY, M., AND SCHWARTZBACH, M. I. Powerforms: Declarative client-side form field validation. *World Wide Web 3*, 4 (2000), 205–214.

[29] BUYUKKOKTEN, O., GARCIA-MOLINA, H., AND PAEPCKE, A. Seeing the whole in parts: Text summarization for web browsing on handheld devices. *Tenth International World Wide Web Conference* (5 2001).

[30] CARDELLI, L. Building user interfaces by direct manipulation. In *Proceedings of the 1st annual ACM SIGGRAPH symposium on User Interface Software* (1988), ACM Press, pp. 152–166.

[31] CARDELLI, L. Obliq: A language with distributed scope. Tech. Rep. SRC-RR-122, Digital Equipment Corporation, June 1994. Available from `http://www.hpl.hp.com/techreports/Compaq-DEC/SRC-RR-122.html`.

[32] CARDELLI, L. A language with distributed scope. In *Conference Record of POPL '95: 22nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, Calif.* (New York, NY, 1995), pp. 286–297.

[33] CARDELLI, L. Semistructured computation. In *Revised Papers from the 7th International Workshop on Database Programming Languages* (2000), Springer-Verlag, pp. 1–16.

[34] CARDELLI, L., AND GORDON, A. D. Mobile Ambients. In *Proceedings of Foundations of Software Science and Computation Structures (FoSSaCS)*, M. Nivat, Ed., vol. 1378. Springer-Verlag, Berlin, Germany, 1998, pp. 140–155.

[35] CARDELLI, L., AND GORDON, A. D. Anytime, anywhere: modal logics for mobile ambients. In *Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (2000), ACM Press, pp. 365–377.

[36] CARDELLI, L., AND GORDON, A. D. Mobile ambients. *Theoretical Computer Science 240*, 1 (2000), 177–213.

[37] CASTAGNA, G., AND VITEK, J. Seal: A framework for secure mobile computations. In *Internet Programming Languages*, H. Bal, B. Belkhouche, and L. Cardelli, Eds., no. 1686 in Lecture Notes in Computer Science. Springer, 1999, pp. 47–77.

[38] CASTELLS, P., SZEKELY, P., AND SALCHER, E. Declarative models of presentation. In *Proceedings of the 2nd international conference on Intelligent user interfaces* (1997), ACM Press, pp. 137–144.

[39] CHANDRA, S., ELLIS, C. S., AND VAHDAT, A. Differentiated multimedia web services using quality aware transcoding. In *INFOCOM - Nineteenth Annual Joint Conference Of The IEEE Computer And Communications Societies* (Tel Aviv, Israel, 2000).

[40] CHRISTENSEN, E., CURBERA, F., MEREDITH, G., AND WEERAWARANA, S. Web services description language (wsdl) 1.1. W3C Note. 15 March 2001. Available from `http://www.w3.org/TR/2001/NOTE-wsdl-20010315`.

[41] CLAESSENS, J., PRENEEL, B., AND VANDEWALLE, J. (How) can mobile agents do secure electronic transactions on untrusted hosts? A survey of the security issues and the current solutions. *ACM Transactions on Internet Technology 3*, 1 (2003), 28–48.

[42] CLARKE, E., GRUMBERG, O., AND PELED, D. *Model Checking*. Cambridge, MA: MIT Press, 1999.

[43] CONCHON, S., AND FESSANT, F. L. Jocaml: Mobile Agents for Objective-Caml. In *First International Symposium on Agent Systems and Applications (ASA'99)/Third International Symposium on Mobile Agents (MA'99)* (Palm Springs, CA, USA, 1999).

[44] CORBETT, J. C., DWYER, M. B., HATCLIFF, J., AND ROBY. Bandera: a source-level interface for model checking java programs. In *Proceedings of the 22nd international conference on Software engineering* (2000), ACM Press, pp. 762–765.

[45] DAYAL, U., HANSON, E. N., AND WIDOM, J. Active database systems. In *Modern Database Systems*. 1995, pp. 434–456.

[46] DE ALFARO, L., AND HENZINGER, T. A. Interface automata. In *Proceedings of the 8th European software engineering conference held jointly with*

9th ACM SIGSOFT international symposium on Foundations of software engineering (2001), ACM Press, pp. 109–120.

[47] DE IPINA, D. L., AND LO, S.-L. LocALE: a Location-Aware Lifecycle Environment for Ubiquitous Computing. In *15th International Conference on Information Networking (ICOIN'01)* (1 2001).

[48] DE IPIA, D. L., AND LAI LO, S. Sentient Computing for Everyone. In *Third IFIP WG 6.1 International Working Conference on Distributed Applications and Interoperable Systems (DAIS'2001)* (09 2001), pp. 41–54.

[49] DE NICOLA, R., FERRARI, G. L., AND PUGLIESE, R. Klaim: A kernel language for agents interaction and mobility. *IEEE Trans. Softw. Eng. 24*, 5 (1998), 315–330.

[50] DENNING, D. E., AND DENNING, P. J. Certification of programs for secure information flow. *Communications of the ACM 20*, 7 (July 1977), 504–513.

[51] DETLEFS, D. L., LEINO, K. R. M., NELSON, G., AND SAXE, J. B. Extended static checking. Tech. Rep. 159, Compaq SRC, December 1998.

[52] DEY, A. K., AND D.ABOWD, G. Towards a better understanding of context and context-awareness. In *Proceedings of the CHI 2000 Workshop on "The What, Who, Where, When, Why and How of Context-Awareness"* (2000).

[53] DIERKS, T., AND ALLEN, C. RFC 2246: The TLS protocol version 1, Jan. 1999. Status: PROPOSED STANDARD. Available from `ftp://ftp.internic.net/rfc/rfc2246.txt`.

[54] DOUGLIS, F., AND OUSTERHOUT, J. K. Transparent process migration: Design alternatives and the sprite implementation. *Software - Practice and Experience 21*, 8 (1991), 757–785.

[55] DUMITRAS, T., KERNER, S., AND MARCULESCU, R. Towards on-chip fault-tolerant communication. In *Proc. Asia & South Pacific Design Automation Conf. (ASP-DAC)* (January 2003).

[56] DWYER, M. B., AND PASAREANU, C. S. Filter-based model checking of partial systems. In *Proceedings of the 6th ACM SIGSOFT international symposium on Foundations of software engineering* (1998), ACM Press, pp. 189–202.

[57] EEYE DIGITAL SECURITY. %u-encoding IDS bypass vulnerability. Advisory AD20010705, `http://www.eeye.com/html/Research/Advisories/AD20010705.html`.

[58] ENGELSON, V., FRITZSON, D., AND FRITZSON, P. Automatic generation of user interfaces from data structure specifications and object-oriented application models. In *Proceedings ECOOP '96* (Linz, Austria, 1996), P. Cointe, Ed., Springer-Verlag, pp. 114–141.

[59] FALLSIDE, D. C., AND WALMSLEY, P. Xml schema part 0: Primer secnd edition, October 2004. W3C Recommendation. Available from `http://www.w3c.org/TR/2004/REC-xmlschema-0-20041028/`.

[60] FEDERAL COMMUNICATIONS COMMISSION. Enhanced 911. More information available from `http://www.fcc.gov/911/enhanced/`.

[61] FERRARI, G., MONTANGERO, C., SEMINI, L., AND SEMPRINI, S. Mark, a reasoning kit for mobility. *Automated Software Eng. 9*, 2 (2002), 137–150.

[62] FERRARI, G., MONTANGERO, C., SEMINI, L., AND SEMPRINI, S. The Mob_{adtl} model and method to design network aware applications. Tech. Rep. TR-03-08, University of Pisa, Computer Science Dept, June 2003. Available from `ftp://ftp.di.unipi.it/pub/techreports/TR-03-08.ps.Z`.

[63] FEYNMAN, R. P. *"Surely you're joking, Mr. Feynman!": Adventures of a curious character (Reprint Edition)*. W. W. Norton and Company, April 1997.

[64] FIELDING, R., GETTYS, J., MOGUL, J., FRYSTYK, H., MASINTER, L., LEACH, P., AND BERNERS-LEE, T. RFC 2616: Hypertext Transfer Pro-

tocol — HTTP/1.1, June 1999. Status: DRAFT STANDARD. Available from `ftp://ftp.internic.net/rfc/rfc2616.txt`.

[65] FONTANA, R., RICHLEY, E., AND BARNEY, J. Commercialization of an ultra wideband precision asset location system. In *IEEE Conference on Ultra Wideband Systems and Technologies* (November 2003).

[66] FOR THE ADVANCEMENT OF STRUCTURED INFORMATION STANDARDS), O. O. Xacml language proposal, January 2002. version 0.8, available from `http://www.oasis-open.org/committees/xacml`.

[67] FOSTER, I. T. The anatomy of the grid: Enabling scalable virtual organizations. In *Proceedings of the 7th International Euro-Par Conference Manchester on Parallel Processing* (2001), Springer-Verlag, pp. 1–4.

[68] FOURNET, C., AND GONTHIER, G. The reflexive chemical abstract machine and the join-calculus. In *Proceedings of the 23rd ACM Symposium on Principles of Programming Languages (POPL'96)* (Jan 1996).

[69] FOURNET, C., GONTHIER, G., LÉVY, J.-J., MARANGET, L., AND RÉMY, D. A calculus of mobile agents. In *Proceedings of the 7th International Conference on Concurrency Theory (CONCUR'96)* (1996), Springer-Verlag, pp. 406–421.

[70] FOURNET, C., LEVY, J.-J., AND SCHMITT, A. An Asynchronous, Distributed Implementation of Mobile Ambients. In *IFIP TCS* (2000), pp. 348–364.

[71] FRANKLIN, S., AND GRAESSER, A. Is it an agent, or just a program?: A taxonomy for autonomous agents. In *Intelligent Agents III, Agent Theories, Architectures, and Languages, ECAI '96 Workshop (ATAL), Budapest, Hungary, August 12-13, 1996, Proceedings* (1997), J. P. Müller, M. Wooldridge, and N. R. Jennings, Eds., vol. 1193 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 21–35.

[72] GIACALONE, A., MISHRA, P., AND PRASAD, S. Facile: a symmetric integration of concurrent and functional programming. *Int. J. Parallel Program. 18*, 2 (1990), 121–160.

[73] GLIGOR, V. D. Characteristics of role-based access control. In *ACM Workshop on Role-Based Access Control* (1995).

[74] GONG, L., MUELLER, M., PRAFULLCHANDRA, H., AND SCHEMERS, R. Going Beyond the Sandbox: An Overview of the New Security Architecture in the Java Development Kit 1.2. In *USENIX Symposium on Internet Technologies and Systems* (Monterey, CA, 1997), pp. 103–112.

[75] GOTTSCHALK, K., GRAHAM, S., KREGER, H., AND SNELL, J. Introduction to web services architecture. *IBM Systems Journal 41*, 2 (2002).

[76] GRAUNKE, P., FINDER, R. B., KRISHNAMURTHI, S., AND FELLEISEN, M. Automatically restructuring programs for the web.

[77] GRAUNKE, P. T., FINDLER, R. B., KRISHNAMURTHI, S., AND FELLEISEN, M. Modeling web interactions. In *Proceeedings of ESOP 2003* (April 2003), P. Degano, Ed., vol. 2618 of *Lecture Notes in Computer Science*, Springer Verlag, pp. 238–252.

[78] GROSS, N. The earth will don an electronic skin. BusinessWeek, Aug. 30, 1999. Available from `http://www.businessweek.com/1999/99_35/b3644024.htm`.

[79] GSM ASSOCIATION. SE.23 Permanent Reference Document on Location Based Services. Available from `http://www.gsmworld.com/technology/applications/location.shtml`.

[80] HALLS, D. A. *Applying Mobile Code to Distributed Systems*. PhD thesis, University of Cambridge, 1997.

[81] HALLS, D. A., AND ROONEY, S. G. Controlling the Tempest: Adaptive management in advanced ATM control architectures. *IEEE Journal on Selected Areas in Communications 16*, 3 (1998), 414–423.

[82] HAN, R., BHAGWAT, P., LAMAIRE, R., MUMMERT, T., PERRET, V., AND RUBAS, J. Dynamic adaptation in an image transcoding proxy for mobile WWW browsing. *IEEE Personal Communication 5*, 6 (1998), 8–17.

[83] HAVELUND, K., AND PRESSBURGER, T. Model checking java programs using java pathfinder. *International Journal on Software Tools for Technology Transfer, STTT 2*, 4 (April 2000).

[84] HIGHTOWER, J., AND BORRIELLO, G. A survey and taxonomy of location sensing systems for ubiquitous computing. UW CSE 01-08-03, University of Washington, Department of Computer Science and Engineering, Seattle, WA, August 2001.

[85] HIGHTOWER, J., BRUMITT, B., AND BORRIELLO, G. The location stack: A layered model for location in ubiquitous computing. In *Proceedings of the 4th IEEE Workshop on Mobile Computing Systems & Applications (WMCSA 2002)* (Callicoon, NY, June 2002), IEEE Computer Society Press, pp. 22–28.

[86] HOFMANN-WELLENHOF, B., LICHTENEGGER, H., AND COLLINS, J. *Global Positioning System: Theory and Practice*, 5th ed. Springer-Verlag, March 2001.

[87] HOHLFELD, M., AND YEE, B. How to migrate agents, Aug 1998. Available from `http://www.bennetyee.org/ucsd-pages/pub/migrate.ps`.

[88] HOLZMANN, G. J. *The SPIN Model Checker : Primer and Reference Manual title*. Pearson Educational, September 2003.

[89] HOLZMANN, G. J., AND SMITH, M. H. An automated verification method for distributed systems software based on model extraction. *IEEE Trans. Softw. Eng. 28*, 4 (2002), 364–377.

[90] HONDA, K., AND TOKORO, M. An object calculus for asynchronous communication. *Lecture Notes in Computer Science 512* (1991), 133–147.

[91] HOPPER, A. 1999 Sentient Computing. *Phil. Trans. R. Soc. Lond. 358*, 1 (2000), 2349–2358.

[92] HULL, R., NEAVES, P., AND BEDFORD-ROBERTS, J. Towards situated computing. In *Proceedings of the 1st IEEE International Symposium on Wearable Computers* (1997), IEEE Computer Society, p. 146.

[93] IEEE. *Telecommunications and Information Exchange Between Systems - Local and Metropolitan Area Network - Part 11: Wireless LAN Medium Access Control and Physical Layer Specifications*. 1999.

[94] IMAMURA, T., DILLAWAY, B., AND SIMON, E. Xml encryption syntax and processing. W3C Recommendation. 10 December 2002. Available from `http://www.w3.org/TR/2002/REC-xmlenc-core-20021210/`.

[95] INTERNET SECURITY SYSTEMS (ISS). Form tampering vulnerabilities in several web-based shopping cart applications. ISS alert. Available from `http://xforce.iss.net/alerts/advise42.php`.

[96] IOANNIDIS, Y. E., AND SELLIS, T. K. Conflict resolution of rules assigning values to virtual attributes. In *Proceedings of the 1989 ACM SIGMOD international conference on Management of data* (1989).

[97] JACOBSEN, K., AND JOHANSEN, D. Mobile Software on Mobile Hardware – Experiences with TACOMA on PDAs. Tech. rep., Department of Computer Science, University of Tromsø, Norway, 12 1997.

[98] JACOBSEN, K., AND JOHANSEN, D. Ubiquitous Devices United: Enabling Distributed Computing Through Mobile Code. In *Proceedings of the Symposium on Applied Computing (ACM SAC'99)* (February 1999).

[99] JOHANSEN, D. Mobile agents: Right concept wrong approach. In *Mobile Data Management. Proceedings of the 5th IEEE International Conference on Mobile Data Management (MDM 2004)* (2004), IEEE Computer Society, pp. 300–301.

[100] JOHANSEN, D., LAUVSET, K. J., VAN RENESSE, R., SCHNEIDER, F. B., SUDMANN, N. P., AND JACOBSEN, K. A TACOMA retrospective. *Softw. Pract. Exper. 32*, 6 (2002), 605–619.

[101] JOHANSEN, D., VAN RENESSE, R., AND SCHNEIDER, F. B. An Introduction to the TACOMA Distributed System Version 1.0. Tech. Rep. 95-23, Department of Computer Science, University of Tromsø, Norway, 6 1995.

[102] KAHN, J. M., KATZ, R. H., AND PISTER, K. S. J. Next century challenges: Mobile networking for "smart dust". In *International Conference on Mobile Computing and Networking (MOBICOM)* (1999), pp. 271–278.

[103] KATSIRI, E., AND MYCROFT, A. Knowledge representation and scalable abstract reasoning for sentient computing using first-order logic. In *Challenges and Novel Applications for Automated Reasoning Workshop* (Miami, July 2003).

[104] KERCKHOFFS, A. La cryptographie militaire. 5–38. Available from `http://www.cl.cam.ac.uk/users/fapp2/kerckhoffs/`.

[105] KUANG, H., BIC, L. F., AND DILLENCOURT, M. B. Iterative grid-based computing using mobile agents. In *Proceedings of the 2002 Intternational Conference on Parallel Processing* (Los Alamitos, Calif., Aug. 2002), T. S. Abdelrahman, Ed., The International Association for Computers and Communications (IACC), IEEE Computer Society, pp. 109–117.

[106] LANGE, D. B., AND OSHIMA, M. *Programming and Deploying Java Mobile Agents with Aglets*. Addison Wesley, 1998.

[107] LANGE, D. B., AND OSHIMA, M. Seven good reasons for mobile agents. *Communications of the ACM 42*, 3 (March 1999), 88–89.

[108] LEROY, X. *The Objective Caml System Release 3.0*. INRIA, Rocquencourt, France, 2000.

[109] LEYDEN, J. Mosquitos smartphone 'trojan' there by design, Aug 2004. Available from `http://www.theregister.co.uk/2004/08/11/mosquitos_malware_myth/`.

[110] LICKLIDER, J. Man-computer symbiosis. *IRE Transactions on Human Factors in Electronics HFE-1* (March 1960), 4–11.

[111] LOBO, J., BHATIA, R., AND NAQVI, S. A. A policy description language. In *AAAI/IAAI* (1999), pp. 291–298.

[112] LOEFFLER, S., AND SERHOUCHNI, A. Creating implementations from PROMELA models. In *Proceedings of the SPIN96 DIMACS Workshop*, pp. 72–80.

[113] LOREK, L. New e-rip-off maneuver: Swapping price tags. ZD-Net. 5th March, 2001. Available from `http://www.zdnet.com/intweek/stories/news/0,4164,2692337,00.html`.

[114] MADHAVAPEDDY, A., MYCROFT, A., SCOTT, D., AND SHARP, R. The case for abstracting security policies. In *Proceedings of the 2003 International Conference on Security and Management (SAM'03)*.

[115] MILNER, R. *A Calculus of Communicating Systems*. Springer-Verlag New York, Inc., 1982.

[116] MILNER, R. *Communicating and Mobile Systems: The π Calculus*. Cambridge University Press, May 1999.

[117] MILNER, R., TOFTE, M., HARPER, R., AND MACQUEEN, D. *The Definition of Standard ML (Revised)*. MIT Press, 1997.

[118] MONTANGERO, C., AND SEMINI, L. Distributed states logic. In *Proceedings of the Ninth International Symposium on Temporal Representation and Reasoning (TIME'02)* (July 2002).

[119] NECULA, G. C. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Langauges (POPL '97)* (Paris, Jan. 1997), pp. 106–119.

[120] NEEDHAM, R. M., AND MULLENDER, S. J. Names. Available from `http://wwwhome.cs.utwente.nl/~sape/gos/chap12.pdf`.

[121] NELSON, B. J. *Remote Procedure Call*. PhD thesis, Carnegie-Mellon University, 1981. Published as CMU technical report CMU-CS-81-119.

[122] NICOLA, R. D., FERRARI, G. L., AND PUGLIESE, R. Programming access control: The klaim experience. In *Proceedings of the 11th International Conference on Concurrency Theory* (2000), Springer-Verlag, pp. 48–65.

[123] OBJECT MANAGEMENT GROUP. *Common Object Request Broker Architecture (CORBA/IIOP)*, 1995. More information available from `http://www.corba.org/`.

[124] ODLYZKO, A. M. Privacy, economics, and price discrimination on the internet. In *ICEC2003: Fifth International Conference on Electronic Commerce*, ACM, pp. 355–366.

[125] OPEN GIS CONSORTIUM, INC. OpenGIS Simple Features Specification for SQL version 1.1, 1998. Available from `http://www.opengis.org/docs/99-049.pdf`.

[126] PAULSON, L., AND NIPKOW, T. Isabelle. Available from `http://www.cl.cam.ac.uk/Research/HVG/Isabelle`.

[127] PESCATORE, J. Web services: Application-level firewalls required. Gartner, Inc. Report. 7th March, 2002. Available from `http://www4.gartner.com/DisplayDocument?id=353429`.

[128] PIER, K., AND LANDAY, J. Issues for location-independent interfaces. Tech. Rep. ISTL92-4, Xerox Palo Alto Research Center, December 1992.

[129] PIERCE, B. C., AND TURNER, D. N. Pict: A programming language based on the pi-calculus. In *Proof, Language and Interaction: Essays in Honour of Robin Milner* (2000), G. Plotkin, C. Stirling, and M. Tofte, Eds., MIT Press.

[130] PRIYANTHA, N. B., CHAKRABORTY, A., AND BALAKRISHNAN, H. The cricket location-support system. In *Proceedings of the 6th annual inter-*

national conference on Mobile computing and networking (2000), ACM Press, pp. 32–43.

[131] QUEINNEC, C. The influence of browsers on evaluators or, continuations to program web servers. In *International Conference on Functional Programming* (2000), pp. 23–33.

[132] REAVIS, J. Layer seven: The future of vulnerabilities. SPI Dynamics White Paper. Available from `http://www.spidynamics.com/`.

[133] REPPY, J. H. *Concurrent Programming with Events—The Concurrent ML Manual*, version 0.9.8 ed. AT&T Bell Lab., Feb 1993.

[134] RICHARDSON, T. Teleporting: Mobile X sessions. *The X Resource 13*, 1 (1995), 133–140.

[135] RIELY, J., AND HENNESSY, M. A typed language for distributed mobile processes (extended abstract). In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (1998), ACM Press, pp. 378–390.

[136] RIVEST, R. RFC 1321: The MD5 message-digest algorithm, Apr. 1992. Status: INFORMATIONAL. Available from `ftp://ftp.internic.net/rfc/rfc1321.txt`.

[137] SALTZER, J., REED, D., AND CLARK, D. End-to-end arguments in system design. *ACM Transactions in Computer Systems 2*, 4 (November 1984), 277–288.

[138] SAMARATI, P., AND DI VIMERCATI, S. D. C. Access control: Policies, models, and mechanisms. In *FOSAD* (2001), vol. 2171 of *Lecture Notes in Computer Science*, Springer, pp. 137–196.

[139] SAMET, H. The quadtree and related hierarchical data structures. *ACM Comput. Surv. 16*, 2 (1984), 187–260.

[140] SANCTUM INC. AppShield™ white paper. March 2001. Available from `http://www.sanctuminc.com/`.

[141] SCHELDERUP, K., AND OLNES, J. Mobile agent security - issues and directions. In *Proceedings of the 6th International Conference on Intelligence and Services in Networks* (1999), Springer-Verlag, pp. 155–167.

[142] SCHNEIDER, F. B. Enforceable security policies. *ACM Trans. Inf. Syst. Secur. 3*, 1 (2000), 30–50.

[143] SCHNEIER, B. *Applied cryptography: protocols, algorithms, and source-code in C*. John Wiley & Sons, New York, 1994.

[144] SCOTT, D., BERESFORD, A., AND MYCROFT, A. Spatial policies for sentient mobile applications. In *Proceedings of the 4th IEEE International Workshop on Policies for Distributed Systems and Networks* (2003), IEEE Computer Society, p. 147.

[145] SCOTT, D., BERESFORD, A., AND MYCROFT, A. Spatial security policies for mobile agents in a sentient computing environment. In *FASE* (2003), vol. 2621 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 102–117.

[146] SCOTT, D., AND SHARP, R. Abstracting Application-Level Web Security. In *The Eleventh International World Wide Web Conference Proceedings* (May 2002), pp. 396–407.

[147] SCOTT, D., AND SHARP, R. Developing secure web applications. *IEEE Internet Computing 6*, 6 (2002), 38–45.

[148] SCOTT, D., AND SHARP, R. Specifying and enforcing application-level web security policies. *IEEE Transactions on Knowledge and Data Engineering 15*, 4 (Jul/Aug 2003), 771–783.

[149] SEKAR, R., VENKATAKRISHNAN, V., BASU, S., BHATKAR, S., AND DUVARNEY, D. C. Model carrying code: A practical approach for safe execution of untrusted applications. In *19th ACM Symposium on Operating Systems Principles (SOSP)* (2003).

[150] SEMICONDUCTOR INDUSTRY ASSOCIATION (SIA). Industry Facts and Figures Page. Available from `http://www.sia-online.org/pre_facts.cfm`.

[151] SHEKHAR, S., CHAWLA, S., RAVADA, S., FETTERER, A., LIO, X., AND LIU, C. Spatial databases: Accomplishments and research needs. *IEEE Transactions on Knowledge and Data Engineering 11*, 1 (Jan/Feb 1999), 45–55.

[152] SHINWELL, M. R., AND PITTS, A. M. Fresh objective caml user manual, Sept 2003. Available from `http://www.freshml.org/foc/docs/foc-manual.pdf`.

[153] SHINWELL, M. R., PITTS, A. M., AND GABBAY, M. J. FreshML: Programming with binders made simple. In *Eighth ACM SIGPLAN International Conference on Functional Programming (ICFP 2003), Uppsala, Sweden* (Aug 2003), ACM Press, pp. 263–274.

[154] SHIRKY, C. Web services and context horizons. *IEEE Computer 35*, 9 (September 2002).

[155] SLEK, J. G., AND LUMSDAINE, A. The matrix template library: A generic programming approach to high performance numerical linear algebra. In *ISCOPE* (dec 1998), D. Caromel, R. R. Oldehoeft, and M. Tholburn, Eds., vol. 1505 of *Lecture Notes in Computer Science*, Springer-Verlag.

[156] STROM, R. E., AND YELLIN, D. M. Extending typestate checking using conditional liveness analysis. *IEEE Transactions on Software Engineering (TSE) 19*, 1 (Jan 1993), 478–485.

[157] STROUSTRUP, B. *The C++ Programming Language*, 3rd ed. Addison-Wesley Pub Co.

[158] SYVERSON, P. A taxonomy of replay attacks. In *Computer Security Foundations Workshop VII* (1994), IEEE Computer Society Press.

[159] TAHA, W. A gentle introduction to multi-stage programming. In *Domain-Specific Program Generation* (2004), C. Lengauer, D. Batory, C. Consel, and M. Odersky, Eds., vol. 3016 of *Lecture Notes in Computer Science*, Springer-Verlag.

[160] TAHA, W., AND SHEARD, T. Multi-stage programming with explicit annotations. In *Proceedings of the 1997 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation* (1997), ACM Press, pp. 203–217.

[161] TARDO, J., AND VALENTE, L. Mobile agent security and Telescript. In *IEEE CompCon '96* (1996), pp. 58–63.

[162] TENNENHOUSE, D. Proactive computing. *Communications of the ACM 43*, 5 (2000), 43–50.

[163] THIEMANN, P. An embedded domain-specific language for type-safe server-side web-scripting, May 2003. Accepted for publication in ACM Transactions on Internet Technology. Available from `http://www.informatik.uni-freiburg.de/~thiemann/papers/wash-cgi.ps.gz`.

[164] THOMSEN, B. Facile antigua release, july 1994. Posted to USENET in `comp.compilers`. Copy available from `http://compilers.iecc.com/comparch/article/94-07-092`.

[165] TODD, L., AND GLINERT, E. P. Polyglot: An architecture and prototype for distributed multi-interface computing. Available from `http://www.rpi.edu/~toddr/Archives/1997/a01g-pgmi/`.

[166] TRUSTED COMPUTING GROUP. TPM specification version 1.2. Available from `https://www.trustedcomputinggroup.org/home`.

[167] UNYAPOTH, A. *Nomadic π-Calculi: Expressing and Verifying Communication Infrastructure for Mobile Computation.* PhD thesis, University of Cambridge Computer Laboratory, March 2001.

[168] VAN DIGGELEN, F., AND ABRAHAM, C. Indoor GPS: The No-Chip Challenge. Available from `http://www.gpsworld.com/gpsworld/article/articleDetail.jsp?id=3053`.

[169] VELDHUIZEN, T. L. Arrays in Blitz++. In *ISCOPE* (dec 1998), D. Caromel, R. R. Oldehoeft, and M. Tholburn, Eds., vol. 1505, Springer-Verlag, pp. 223–230.

[170] VELDHUIZEN, T. L. C++ templates as partial evaluation. In *Proceedings of Partial Evaluation and Semantics-Based Program Manipulation (PEPM'99)* (jan 1999).

[171] WALL, L., AND SCHWARTZ, R. L. *Programming Perl*. O'Reilly & Associates, Inc., Sebastopol, CA, 1992.

[172] WANT, R., HOPPER, A., FALCAO, V., AND GIBBONS, J. The active badge location system. Tech. Rep. 92.1, ORL, 24a Trumpington Street, Cambridge CB2 1QA, 1992.

[173] WARD, A. *Sensor-driven Computing*. PhD thesis, University of Cambridge, August 1998.

[174] WARD, A., JONES, A., AND HOPPER, A. A New Location Technique for the Active Office. *IEEE Personal Communications 4*, 5 (October 1997), 42–47.

[175] WEISER, M. The Computer for the 21st Century. *Scientific American* (9 1991).

[176] WHITE, J. E. Telescript technology: The foundation for the electronic marketplace. Tech. rep., General Magic, Inc., 2465 Latham Street, Mountain View, CA 94040, 1994.

[177] WOJCIECHOWSKI, P. T. *Nomadic Pict: Language and Infrastructure Design for Mobile Computation*. PhD thesis, University of Cambridge Computer Laboratory, March 2000.

[178] WORLD-WIDE WEB CONSORTIUM. XML Path Language (XPath) Specification, November 1999. Available from `http://www.w3.org/TR/xpath/`.

[179] WORLD-WIDE WEB CONSORTIUM. Hypertext markup language activity statement, October 2003. Available from `http://www.w3.org/MarkUp/Activity`.

[180] YANNOPOULOS, A., STAVROULAS, Y., AND VARVARIGOU, T. A. Moving e-commerce with pivots: Private information viewing offering total safety. *IEEE Transactions on Knowledge and Data Engineering 16*, 6 (June 2004), 641–652.

[181] YAU, S. S., AND KARIM, F. A context-sensitive middleware-based approach to dynamically integrating mobile devices into computational infrastructures. *Parallel and Distributed Computing 64*, 2 (Feb 2004), 301–317.

APPENDIX A

SPDL-2 DTD

```
<!ELEMENT site (policy*)>
  <!ATTLIST site name                CDATA #REQUIRED>
  <!ATTLIST site policy_author_name  CDATA #REQUIRED>
  <!ATTLIST site policy_author_email CDATA #REQUIRED>
  <!ATTLIST site description         CDATA "unspecified">
  <!ATTLIST site base_uri            CDATA #REQUIRED>


  <!ELEMENT policy (parameter*, uri*, cookie*, policy*)>
    <!ATTLIST policy name            CDATA #REQUIRED>
    <!ATTLIST policy description     CDATA "unspecified">
    <!ATTLIST policy javascript      (Y | N) "N">


    <!ELEMENT parameter (validation*, transformation*)>
      <!ATTLIST parameter method     (GET | POST |
                                      GETandPOST)
                                      "GETandPOST">
      <!ATTLIST parameter name       CDATA #REQUIRED>
```

```
  <!ATTLIST parameter maxlength  CDATA #REQUIRED>
  <!ATTLIST parameter minlength  CDATA   "0">
  <!ATTLIST parameter required   (Y | N) "N">
  <!ATTLIST parameter MAC        (Y | N) "Y">
  <!ATTLIST parameter type       (int
                                 | float
                                 | bool
                                 | string) #REQUIRED>


<!ELEMENT cookie (validation*, transformation*)>
  <!ATTLIST cookie name          CDATA #REQUIRED>
  <!ATTLIST cookie maxlength     CDATA #REQUIRED>
  <!ATTLIST cookie minlength     CDATA   "0">
  <!ATTLIST cookie required      (Y | N) "N">
  <!ATTLIST cookie MAC           (Y | N) "Y">
  <!ATTLIST cookie type          (int
                                 | float
                                 | bool
                                 | string) #REQUIRED>


<!ELEMENT uri (parameter*)>
  <!ATTLIST uri prefix           CDATA #REQUIRED>


  <!ELEMENT transformation       (#PCDATA)>
    <!ATTLIST transformation htmlencode (Y | N) "Y">

  <!ELEMENT validation           (#PCDATA)>
```

APPENDIX B

Case Study SPDL-2

```
...
<policy ...>

  <uri prefix="Buy.asp"
       description="Do CreditCard transaction"
       javascript="Y">

     <parameter name="price" method="POST" maxlength="10"
                minlength="1" required="Y" type="float" >
       <validation> this gt 0.0 </validation>
     </parameter>

     <parameter name="productID" method="POST"
                maxlength="10" minlength="1"
                required="Y" type="int" />


     <parameter name="surname" method="POST"
                maxlength="30" minlength="2"
```

```
            required="Y" MAC="N" type="string">
    <transformation>
       Transform.EscapeSingle
           (Transform.EscapeDouble(this))
    </transformation>
</parameter>

<parameter name="CCnumber" method="POST"
           maxlength="16" minlength="16"
           MAC="N" required="Y" type="int">
    <validation>
       let
           fun first(s:string):string =
               String.mid(s,1,1)
           fun rest(s:string):string  =
               String.mid(s,2,String.length(s)-1)

           fun double(s:string,a:bool):string =
              if s="" then ""
                else
                  (if a then first(s)
                   else String.fromInt
                   (Int.fromString(first(s))*2))
                  ++ (double (rest (s), not a))

           fun sum(s:string):int =
              if s="" then 0
                       else (Int.fromString (first(s)))
                            + (sum (rest(s)))

        in sum(double(this,false)) % 10 = 0
       end
    </validation>
</parameter>
```

```
    <parameter name="expires" method="POST"
               maxlength="5" minlength="5"
               MAC="N" required="Y" type="string">
        <validation> String.format(this,"\d\d/\d\d") and
                     Int.fromString(mid(s,1,2))<=12 and
                     Int.fromString(mid(s,1,2))> 0
        </validation>
    </parameter>
  </uri>
</policy>
```