Securing Information Flow in Loosely-Coupled Systems

Linas Žvirblis

Kongens Lyngby 2011
IMM-M.Sc.-2011-41

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk

IMM-M.Sc.-2011-41

Summary

Information-flow control is an important element in computer system security, and there has been significant work done in the field by Denning, Volpano, and others. However, most of the work deals with information-flow control inside a single monolithic application. Wide adoption of the Web service architecture and related technologies effectively solved the problem of universal standard of interconnection of independent systems into larger scale system, but largely ignored the problem of information-flow control. This thesis suggests an approach, which allows for information-flow control techniques of the decentralised label model to be applied to distributed loosely-coupled systems based on Web services. The resulting system design is compatible with existing Web service-based systems, and allows for integration of components that do not natively support information-flow control.

Contents

Chapter 1

Introduction

Information-flow control is an important element in computer system security, and there has been significant work done in the field by Denning [11], Volpano [34], and others. Most of the work deals with information-flow control inside a single monolithic application. However, nowadays a concept of a single independent application is fading away in favour of distributed loosely-coupled systems. In such systems separate components have little to no knowledge about each other. Such components can be created using different technologies by different vendors, and be under control of separate independent entities. In fact, this is often the case because systems using services, provided by entities such as Amazon [36], Facebook [23], Google [24], and others, as an integral part of the system functionality, are in no way a rarity. This makes the definition of a system component by itself rather fuzzy.

Wide adoption of the Web service architecture [5] and related technologies effectively solved the problem of universal standard of interconnection of independent systems into larger scale systems, but largely ignored the problem of information-flow control. In real-world old-school monolithic systems information-flow control is often overlooked and is not really considered a necessity. Which is true up to a point, because in monolithic systems information never leaves the boundaries of the system, and is controlled by the same entity. Proper use of object-oriented programming paradigm also tends to cover most of information-flow control use-cases by employing encapsulation. Software de-

velopers are usually not even aware that they are dealing with information-flow control when they declare fields in classes as being *public*, *private*, *protected*, etc. And in special cases, specialized information-flow control technologies, such as Jif (Java and information-flow) [32] can be employed. The problem is that this does not translate to the Web service architecture.

Technologies such as Java API for XML Web Services (JAX-WS) [6] allow for almost completely transparent development and integration of Web service-based systems by providing a bridge between standard programming language constructs and Web services. While this makes development of such systems considerably easier, it does mask certain implications of doing so. It is important to realize that data leaving the system through a Web service does not retain expected information-flow properties. Even if the receiving part is willing to maintain these properties, it is by no means a straightforward task, because the Web service architecture provides no standard means to transfer information-flow control rules to a client.

This thesis suggests an approach which allows information-flow control techniques to be used in distributed loosely-coupled systems, composed of semi-independent components. Information-flow control meta-data is exported via standard Web service interfaces. A secure information-flow control-aware distributed system built on Web service architecture is demonstrated, and problems involved in development are identified. Web services are used as a basis for the implementation because of clear definition of interfaces for interaction between system components. It is achieved by integrating secure information-flow control model in a Web service orchestration language in order to be able to perform static analysis on information flow.

Claim is made that such an approach is sufficient to ensure secure data flow inside a Web service-based system without requiring invasive changes to the Web service architecture, and that this would allow for a fairly transparent implementation that would allow for interconnection of existing web services, and provide required security properties with minimal overhead.

Chapter 2

Challenge

2.1 Information-flow control

We will use a very simplified model of an online shop as an example. The Shop sells products manufactured by the Company and delivers them to customers to the address of their residence. The system is composed of a Shop Service that provides an interface for the customer to order desired products, a Company Service that keeps a database of all customers and their related data, and a Postal Service that keeps a database of all people, and their addresses.

To produce useful results, the three entities have to cooperate and exchange data. The client only interacts with the Shop Service, and as far as the client is aware, that is the only entity in the system it will be exchanging data with. But that point of view is misleading, because once the data reaches the Shop Service, it is being exchanged with two more entities, namely the Company Service and the Postal Service, without the client being aware of this. As the transaction between the Shop Service and the client may involve sensitive data, such as credit card data or social security number, it may be undesirable for the client that this data is exchanged with other entities without its consent. The question is, how can this information be controlled, and data leaks prevented?

The standard answer would be to employ information-flow control technologies.

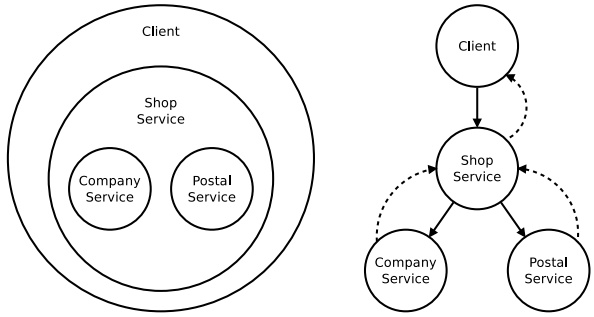

Figure 2.1: World awareness and information flow in the system

The problem is that information-flow can only be enforced in a monolithic system where all system components are under control of the same entity. This does not hold in loosely-coupled systems that may be composed of diverse components under control of independent entities. There are several challenges involved here.

Let us assume that the entities are willing to cooperate in enforcing information-flow control. Here we run into a problem that separate entities may implement their system components in different technologies, such as Java and .NET just to name two of the most popular ones. Some, or even all, of the technologies in use may not even have any support for information-flow control.

Now let us relax our assumptions, and say that all components run on systems that do support information-flow control, and actively employ it. Even then we run into a problem of how to exchange information between system components under control of different entities? It is very likely that the entities in question are exchanging information by some means of remote procedure calls or remote method invocation. It is also very likely that the technology in question does not provide means to exchange the information-flow meta-data associated with the data exchanged. This could possibly be hand-coded and exchanged as data, but may also require invasive changes to the system, which is generally not desirable.

It quickly becomes evident that such a solution may require too extensive changes to existing system components to be acceptable. It must also not be platform-dependant, because that would defeat the purpose of having distributed loosely-coupled systems. This is only possible if the solution can be constructed in such a way that would allow for interconnection of information-flow control-aware and unaware systems in a fairly generic way. Another im-

portant issue arising from the previous statement is that if the system can be composed of components not supporting information-flow control, where is an information-flow policy being enforced?

Here we demonstrate an approach that can be applied to existing system components without the need to modify them, or even be aware of information-flow control being in place. It also supports heterogeneous classification levels within messages exchanged, allowing for fine-grained classification policies.

2.2 Enforcing information-flow control policy

A key problem in implementing such a system is identifying the point of policy enforcement, assuming that at least some system components are not aware of information-flow control. Taking a closer look at the information flow in our example system reveals that only the Shop Service is a component communicating, and aware of all other components in the system. As this is a central component, through which all information is flowing, it makes for a good starting point for investigating information-flow control enforcement.

This property gives us two possibilities. First, we assume that the Shop Service is the only component aware of the information-flow control in the system, and it is solely up to it to enforce the policy. Second, we assume that system components are running their own information-flow control implementations and we want to bridge the implementations by allowing information-flow meta-data exchange. In this section both approaches are discussed.

If the assumption is that there is only one component in the system that is aware of information-flow control, it is clear that information-flow meta-data cannot originate from other components, and must be self-contained within the enforcing component. The fact that the information-flow meta-data cannot be coming from outside means that all meta-data must be present beforehand for validation. Since system components do not implement any information-flow control, it is not possible to perform information-flow control at run-time inside them. It would still be possible to perform run-time checks inside the enforcing component, but because the information-flow meta-data can be considered to be static, it is much more efficient to perform the checks at compile-time. This suggests static validation as means to enforce the policy.

Another case is if we assume that the system components are in fact running their own implementations of information-flow control. In order for them to be able to exchange the information-flow control meta-data, a format of the meta-

data and a protocol of meta-data exchange need to be defined. This also implies that the system components should be running their own implementations of the same information-flow control model. This is necessary because all system components have to be able to parse the meta-data in order to perform useful work.

In theory it is possible that the system components may be using incompatible information-flow control models, and employing conversion mechanism to export the meta-data in an appropriate format. This is a very platform-specific approach, and would likely to be too cumbersome to implement, but it is not explicitly forbidden. We just assume that any component exporting the meta-data in a required format is allowed to be part of the system, regardless of how this meta-data is generated.

An obvious question to ask is how the separate components within a loosely-coupled system can be trusted to enforce their own information-flow control policy? And a short answer is that they cannot. It is therefore crucial to prevent the components from obtaining sensitive information in the first place, or if this is not possible, just assume that the information can be leaked, and design the system accordingly. Assuring mutual trust between system components is a problem of an approach known as *design by contract* or *programming by contract* [15], but here the concept of mutual distrust is employed instead.

This thesis describes a hybrid solution that allows for both information-flow control-aware and unaware components to be integrated into a loosely-coupled system. It is based on the decentralised label model [32] and the concept of mutual distrust, rather than mutual trust.

Chapter 3

Background

In this chapter we discuss existing technologies and means, and how they can be used to achieve our goals. Namely, we look into what an information-flow control model is, how it works, and how can it be used. Also we discuss the concept of a loosely-coupled system, and look at specific implementations. We also familiarize ourselves with related work in the field of information-flow control-aware systems.

3.1 Information-flow control model

An information-flow control model is a mathematical model that allows for tracking and verifying the flow of information within a system. It is similar in concept to traditional access control models. There are several information-flow control models available, but many are too limited or too restrictive to be used in practice [29]. The decentralised label model addresses these limitations, and aims to be usable in actual implementations [32].

3.1.1 Decentralised label model

The decentralized label model, is a label model for control of information flow in systems with mutual distrust and decentralized authority. The model allows users to declassify information in a decentralized way, and provides support for fine-grained data sharing. It supports static program analysis of information flow, so that programs can be certified to permit only acceptable information flows, while largely avoiding the overhead of run-time checking [32].

The model is based on a notion of labels that allow individual owners of information to express their own policies. A reader policy allows the owner of the policy to specify which principals the owner permits to read a given piece of information. A reader policy is written $o \rightarrow r$, where the principal o is the owner of the policy, and the principal r is the specified reader [8]. A reader policy expresses privacy requirements. A writer policy written $o \leftarrow w$ allows the owner to specify which principals may have influenced ("written") the value of a given piece of information [8]. A writer policy expresses integrity requirements. Owners themselves are also principals: identifiers representing users and other authority entities such as groups or roles [31].

The model allows principals to control the flow of their information, and declassify their own data without requiring a mutually-trusted entity to perform declassification. However, a principal is only allowed to weaken the policies that it has itself provided, and thus may not endanger the data that it does not own [32].

These properties allow for the model to be implemented in distributed systems, where security policies cannot be decided by any central authority. Instead, individual participants in the system must be able to define and control their own security policies. The system will then enforce behaviour that is in accordance with all of the security policies that have been defined [32], resulting in a behaviour that resembles collaboration much more than traditional mandatory access control models [3].

3.1.2 Static analysis

Static program analysis offers static compile-time techniques for predicting safe and computable approximations to the set of values or behaviours arising dynamically at run-time when executing a program on a computer [33]. Traditionally, the main application of these techniques is in optimizing compilers in order to avoid redundant computations, and check validity of the code. More

recent applications are validation of software for absence of malicious behaviour, and information-flow control. There is no single technique that is the program analysis technique, but rather it is a wide range of different techniques that take similar approach to solve these problems.

Static analysis is a sister approach to model checking, because they both are used to achieve the same goals. The difference is that model checking requires running code, while static analysis just requires to compile it. Model checking also requires a working model of the environment, and environments are often messy and hard to specify [12]. Whereas static analysis operates directly on the code, making it somewhat more versatile approach. However, it is not uncommon to combine both approaches.

The biggest strength of static analysis is that it does not require any changes to the checked code (that being either source code, bytecode, or even binaries), and can be used to validate programs, written in "unsafe" languages such as C or assembly, for desired properties without imposing any limitation on run-time environments, thus eliminating run-time check overhead.

The previous property can be employed in order to add transparent information-flow control to systems that do not provide any information-flow control support. Minimizing the amount of required modifications to existing systems is a big advantage in deploying new technologies in the real world.

3.1.3 Jif security-type language

Jif [32] is an implementation of a security-typed language know as JFlow [30]. It is an extension of Java programming language to support information-flow control by adding the decentralised label model as an integral part of the language. It adds static analysis of information flow for improved security assurance. The primary goal is to prevent confidential and/or untrusted information from being used improperly [8].

An important difference between Jif and other work on static checking of information flow is the focus on a usable programming model. Despite a long history, static information flow analysis has not been widely accepted as a security technique. One major reason is that previous models of static flow analysis were too limited or too restrictive to be used in practice [29].

Jif aims to overcome these limitations allowing real-world applications to be written to incorporate information-flow control. It extends Java by adding labels that express restrictions on how information may be used [8] by putting the

definition of statically-checked properties of the program inside the program itself. This may appear to contradict the definition of static analysis by requiring support inside the language itself, but it is important to realize that information-flow control policy is something that matters to people, not computers, and thus cannot be inferred automatically. There is no "correct" policy if the goals are not defined.

Labels are specified in a similar way to scope and type parameters of a variable, and resemble Java annotations, which should be reasonably familiar to Java programmers.

```
1 private int {Alice → Bob} x;
```

Listing 3.1: Example of a label in Jif

Method declarations are also labelled in a familiar fashion. In the following example we can see that method *dummyMethod* returns a boolean value labelled $Alice \leftarrow \top$, takes two parameters labelled $Alice \leftarrow \top$, and begin label is also $Alice \leftarrow \top$. This means that principal *Alice* is the sole owner of the data, as the writer set consists of a single *top* principal. There is no reason why the labels cannot be heterogeneous, and may in fact differ depending on the actual goals of a global information-flow control policy within the system.

```
1 public boolean{Alice ← ⊤} validate{Alice ← ⊤}(String{Alice ← ⊤} par1,
      int{Alice ← ⊤} par2) {
2   return true;
3 }
```

Listing 3.2: Example of a method annotated with labels in Jif

If a Jif program type-checks, the compiler translates it into Java code that can be compiled with a standard Java compiler. The program can then be executed with a standard Java virtual machine. Although enforcement is mostly done at compile-time, Jif does also allow for some enforcement to take place at run-time. Therefore, Jif programs in general require the Jif runtime library [8].

Jif supports various kinds of polymorphism to make it possible to write reusable code that is not tied to any specific security policy [8]. For example, polymorphic labels for method parameters are supported. It also treats labels and principals as first-class objects, allowing for use of dynamic run-time labels and principals.

Jif does not support the Java thread model for concurrent programming in order to avoid leaking data trough timing channels [8]. Unfortunately this means that it only supports single-threaded applications. Jif also does not deal with covert channels, because detecting those is a difficult problem that is yet to be solved.

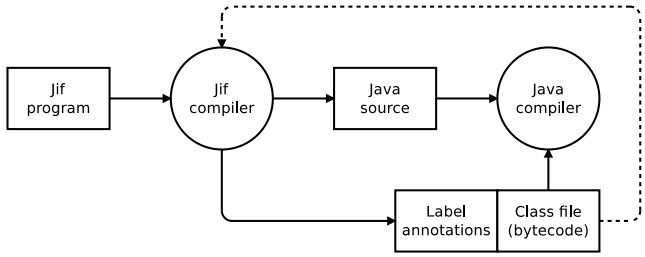

Figure 3.1: Jif compiler [30]

Despite several limitations imposed over traditional Java programming model, Jif provides a large subset of features found in Java, and is definitely suitable for implementing many real-world applications while taking advantage of information-flow control features of the decentralised label model.

3.2 Loosely-coupled systems

A system is defined as "a whole compounded of several parts or members" [28]. Traditionally many computer software systems were monolithic systems, as in composed of components that can only produce useful work when composed together. Such systems are called tightly-coupled systems. However, nowadays many systems are being designed based on completely different concepts, and resemble a collection of independent systems more than a single unit. They are called distributed loosely-coupled systems.

3.2.1 Definition of a loosely-coupled system

An important remark that needs to be made here is that terms *loosely-coupled system* and *distributed system* are synonymous, but do not mean exactly the same. According to the Web services glossary [21], coupling is the dependency between interacting systems. This dependency can be decomposed into real a dependency and an artificial dependency:

1. A real dependency is the set of features or services that a system consumes from other systems. The real dependency always exists and cannot be reduced.

	Tightly-coupled	Loosely-coupled
Interaction	Synchronous	Asynchronous
Messaging style	RPC	Document
Message paths	Hard coded	Routed
Technology mix	Homogeneous	Heterogeneous
Data types	Dependent	Independent
Syntactic definition	By convention	Published schema
Bindings	Fixed and early	Delayed
Semantic adaptation	By re-coding	Via transformation
Software objective	Reuse, efficiency	Broad applicability
Consequences	Anticipated	Unexpected

Table 3.1: Tight versus loose coupling [27]

2. An artificial dependency is the set of factors that a system has to comply with in order to consume the features or services provided by other systems. Typical artificial dependency factors are language dependency, platform dependency, API dependency, etc. Artificial dependency always exists, but it or its cost can be reduced.

Loose coupling describes the configuration in which artificial dependency has been reduced to the minimum [21]. Whereas Sun Microsystems defines the term distributed computing (remote object invocation, etc.) to refer to programs that make calls to other address spaces, possibly on another machine [37] without discussing the coupling. In his book on Web services and loose-coupling Doug Kaye [27] provides a brief summary of properties of tightly and loosely coupled systems. As can be seen in the table 3.1, tightly-coupled and loosely-coupled systems differ in both technologies and goals.

Here we will say that all loosely-coupled systems are distributed, but distributed systems can be both loosely-coupled and tightly-coupled. For example, systems based on Java RMI [40] are distributed tightly-coupled systems, and systems based on Web services [5] are distributed loosely-coupled systems. This distinction is very important when talking about information-flow control in distributed systems, because tight coupling implies much more centralised system design model, which is much closer to monolithic system design than that of loosely-coupled systems. It has been proven that information-flow control is possible in distributed tightly-coupled systems by f.ex. implementing information-flow control-aware distributed poker game [2], while information-flow control in loosely-coupled systems remains an ongoing problem.

3.2.2 Web services

Web services provide standard means of interoperating between different software applications, running on a variety of platforms and/or frameworks [5]. Unlike other RPC or remote-invocation architectures, Web services provide a universal, platform independent way of exchanging data between loosely-coupled system components. The official definition of a Web service according to W3C is as follows:

> A Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards [5].

Web service architecture was the first truly implementation-agnostic architecture for message exchange over the network. Because of vendor-agnostic nature, it gained popularity among major IT software infrastructure vendors, and quickly gained lead in competition with other remote-invocation technologies that were tied either to a vendor, platform, operating system, or were simply too complex to implement. Nowadays Web services undoubtedly have the leading position among similar technologies.

While Web service architecture essentially solved the problem of interconnection of distributed system components, it does not have any built-in support for information-flow control. However, extensible nature of underlying XML-based technologies make it possible to adapt the architecture to accommodate the new goals without breaking compatibility with existing implementations.

3.2.3 Business Process Execution Language

Business Process Execution Language (BPEL), short for Web Services Business Process Execution Language (WS-BPEL) is an OASIS standard language for specifying business process behaviour based on Web services [26]. BPEL is an orchestration language, not a choreography language. It means that it does not define a protocol for peer-to-peer interaction, but rather only specifies message exchange sequence between system components.

The basic concepts of BPEL can be applied in one of two ways, *abstract* or *executable*. A BPEL Abstract Process is a partially specified process that is not intended to be executed and that must be explicitly declared as "abstract". Whereas Executable Processes are fully specified and thus can be executed [26]. The difference is that an Abstract Process may hide some of the specific details of operations, removing information required for execution, but essentially remains human-readable, and can be used as a template for implementation, or to describe observable behaviour of Executable Processes [26].

BPEL is XML-based language, and while it does not by itself specify any graphical representation of the process, there exists a mapping between BPEL and Business Process Model and Notation (BPMN), which is a graphical representation for specifying business processes [39]. BPMN has become the de facto standard in graphically representing BPEL processes, and is widely used by many BPEL designer tools and implementation such as OpenESB [13] and their NetBeans-based BPEL designer, or BPEL Designer for Eclipse [18], and many others. This provides a way to present BPEL process in human-readable form, and aids in integration of large and complex systems. It is therefore BPEL is often used as a glue technology in integration of loosely-coupled systems based on Web services.

BPEL supports extensibility by allowing namespace-qualified attributes to appear on any BPEL element and by allowing elements from other namespaces to appear within BPEL defined elements. This is allowed in the XML Schema specifications for BPEL [26]. This means that it can be extended to include additional features not present in the standard. Extensions are either mandatory or optional. In the case of mandatory extensions not being supported by a BPEL implementation, the process definition must be rejected. Optional extensions not supported by a BPEL implementation must be ignored [26].

This gives us two possibilities in extending BPEL to provide support for information-flow control. First is to make extension mandatory and to make them an integral part of the business process. However, this requires modification to the implementing runtime environment, and may be undesirable, as the BPEL document becomes implementation-specific. Another option is to make extensions optional and to add information-flow control meta-data transparently. This allows it to be parsed by supporting tools, but does not interfere with normal execution of the process on unmodified runtime.

3.3 Related work

Despite a relative unpopularity of information-flow control-aware systems in real-world deployments, there exist several frameworks for Web application development that support the decentralised label model.

3.3.1 SIF framework

SIF (Servlet Information Flow) is a software framework for building web applications, using language-based information-flow control to enforce security [9]. SIF is built using the Java Servlet framework, much like Apache Struts [17] or Spring by SpringSource (a division of VMware) [1]. But SIF applications are written in Jif language, and the user interface of a web application is presented as HTML with forms as a way for a user to provide input [9]. SIF provides a way to include cascading style sheets (CSS), and static JavaScript code in the output HTML, allowing for rich user interfaces with only small limitations on how this can be achieved.

SIF is based on the assumption that web applications are insecure, and possibly buggy, so the information-flow control is enforced on server-side [9]. SIF does not deal with network attacks such as man-in-the-middle, eavesdropping, and does not involve cryptography. It also does not provide any security against denial-of-service attacks.

All principles of programming in Jif also apply in SIF, but it is presented in a web application-oriented fashion. SIF requires each input field on a page to have an associated security label to be enforced on the input when submitted [9]. However, SIF does not protect against the user copying sensitive information from the output web page, and pasting into a non-sensitive input field, which is not possible in general, so the user should be prevented from seeing information they are not trusted to see [9].

SIF is a promising technology that has real potential to bring information-flow control to web applications. However, it aims to solve a different problem, that does not involve distributed loosely-coupled systems. It is still important as a proof that Java servet API can be extended to support information-flow control, because Java Web service implementation is also based on servlets. Therefore it can be claimed that Web services can also be implemented in an information-flow control-aware technology.

3.3.2 Swift framework

Swift is also a framework for development of web applications based on Jif, much like SIF, but takes a different approach, which is referred to as *secure by construction* by the authors [7]. However, it is a more high-level framework somewhat comparable to Ruby on Rails [22].

One of the major problems in designing web applications is deciding how much code, and what functionality is to be implemented on client-side, and what on server-side. Usually this is an offset between responsiveness and security. Running more code on the client-side increases responsiveness of the application, and reduces the load on a server, but may also have security implications if a client is trusted with too much functionality. On the other hand, verifying every possible step on the server-side will most likely lead to the application appearing to be highly unresponsive, or slow as perceived by the user.

Swift abstracts many details that a programmer usually has to deal with in traditional web application frameworks. It aims to automatically partition application code to client-side code and server-side code while providing assurance that the resulting placement is secure and efficient [7]. It also hides the complexity of underlying HTTP message exchanges from an application developer, allowing for more focus on the functionality of the application, and more rapid application development.

Swift applications are written in Jif. Jif code is then compiled to client-side Java code, and server-side Java code through and intermediate representation known as WebIL [7]. Google Web Toolkit (GWT) is then used to compile the client-side Java code to JavaScript that can be run in a browser on the client.

It is slightly incorrect to say that Swift is a web application framework, because it offers more functionality than that usually associated with web application frameworks. It offers a complete platform for developing applications where client-side code and client-side GUI code are treated as integral part of a single application. It is not really designed to interoperate with other systems, and thus not designed to be used in loosely-coupled systems. However, making it self-sustaining and independent on external components does make it attractive when implementing tightly-coupled applications that do run on the web.

Chapter 4

Case study

The goal of this case study is to investigate if and how existing technologies can be utilised to create a new type of system. In other words, we want to see if desired properties of existing information-flow control-aware systems can be applied to Web service-based systems. Here we take an evolutionary approach where the same system is re-implemented in different technologies while trying to to keep them as consistent and close in functionality as possible. It is important that the implementations do not diverge too much, because otherwise it may not be possible to conclude that information-flow control can actually be applied to existing systems without changing their behaviour. If this is in fact the case, it would seriously limit applicability of the solution, and would make deployment of such systems difficult in loosely-coupled environments when it is not always

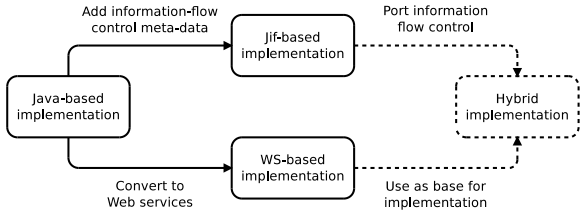

Figure 4.1: Evolutionary approach to the final implementation

possible to modify behaviour of all system components, most likely because not all of them are under control of the same entity.

The first implementation is done in plain Java with no information-flow control in place. It is used as a reference when checking consistency of the behaviour and the results produced by different implementations. Later, the Java implementation is transformed into two other implementations. It is converted to Jif code to add information-flow control capabilities, and Web service-based implementation to produce a loosely-coupled system with separated components. The final goal is to merge these two implementations in order to produce an information-flow control-aware loosely-coupled system.

4.1 Online shop system

Let us return to the online shop example discussed earlier. It is a pretty trivial system that only involves several entities, but at the same time it represents many real-world systems rather well. We will make an assumption that if a concept can be applied to our small example system, it can also be applied to a real-world solution. Of course it is true that this assumption cannot always hold, so any possible limitations are discussed separately.

To summarize the design of the system we say that the Shop sells products manufactured by the Company and delivers them to customers to the address of their residence, which is verified with the Post. The system is composed of a Shop Service that provides an interface for a client to order desired products in the Shop, a Company Service that keeps a database of all customers of the Company and their related data, and a Postal Service that allows access to a database of the Post which contains addresses of all people.

As can be seen in figure 4.2, a client initiates the process by placing an order for a desired product in the Shop via the Shop Service. We trust the client to provide his identity as part of the request made. The Shop Service relays personal information of the client to the Company via the Company Service in order to verify if the client is actually a customer in the Company. The Shop Service also relays this information to the Postal Service in order to verify if the address supplied by the client is valid. If both succeed, and actual order is placed via the Company Service and a receipt is delivered to the client.

In order to simulate a real-world system, we make an assumption that the components (or the services) of the system may not have been designed as integral part of the final system, so they are not using homogeneous data types.

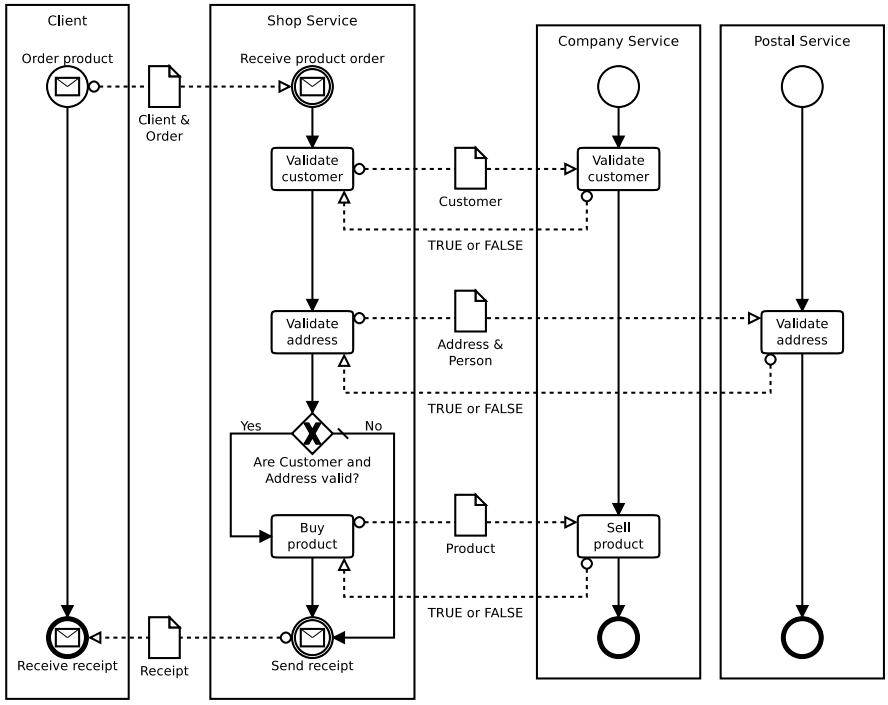

Figure 4.2: Business process of the online shop system

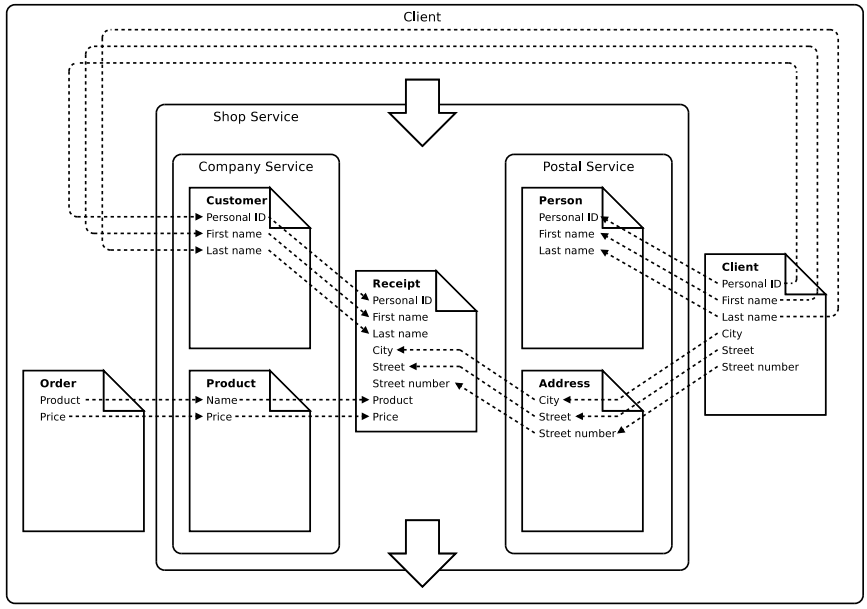

Figure 4.3: Information flow inside the system

However, we make sure that the data types are compatible, so that we could avoid data conversions in the system. Data conversions can be seen as data exchange with yet another system component, and does not add any additional value to the example system.

Actual information flow in the system is described in figure 4.3. The borders around the services are not "transparent", meaning that the client is only aware of the Shop Service, but not of the Company Service or the Postal Service. Data objects within the borders signify that they "belong" to the system component. In reality such separation is a bit artificial, because the Shop Service basically has to be aware of all the data objects, because it communicates with every entity in the system. Therefore the "origin" of a data object is somewhat irrelevant concept, it is the origin of the data that matters here.

4.2 Web service-based implementation

Web service-based implementation is a very straightforward port of the plain Java implementation, because in Java creating a Web service is just a matter of

annotating appropriate class as a Web service, and its methods as Web service methods. It then auto-generates all of the boilerplate code needed to export the methods through Web service interfaces, and generates WSDL definitions and related XSD type definitions.

We do this for several reasons. First of all, auto-generation provides the closest possible mapping between the Java code and WSDL and XSD definitions. Keeping implementations consistent is important for reasons stated earlier. Secondly, we convert a tightly-coupled monolithic system into a loosely-coupled system, which is otherwise identical in its functionality and in the results it produces.

It has been shown that information-flow control can be added to existing monolithic tightly-coupled systems, so having an equivalent implementation of a loosely-coupled system is important, because results produced by both systems have to be comparable in order to make any conclusions about the success or failure to add information-flow control to an existing loosely-coupled system.

4.3 Jif-based implementation

Because Jif is basically a superset of Java language, any Java application should be possible to port to Jif. This is not a completely true statement, because Jif does have some limitations, such as being single-threaded, and does not have nearly as rich of a class library as Java. We ignore such limitations for now, and claim that it does not affect our example implementation, because Jif is the closest thing Java code can be mapped to. In fact, we demonstrate that addition of information-flow control labels is the largest change to Java code required to convert it to proper Jif code, and that it can be done in a quite straightforward way.

To get an initial system running in Jif, we simply label everything with the same label. In other words, we only have one principal, and that principal is owner of all the data in the system. Of course, this is equivalent to not having any information-flow control at all, and serves simply as a way to familiarize oneself with the specialities of Jif. For example, Jif handles exceptions slightly differently than Java. In Jif, run-time exceptions such as *NullPointerException* and *IndexOutOfBoundsException* must be caught and handled [29], because run-time exceptions can possibly leak information and can be used as covert channels. Other than the addition of labels, and other minor differences, Jif code is basically Java, as can be seen in the code example 4.1.

The biggest challenge is actually getting the information-flow control policy

```
1  public class CompanyService {
2      public boolean validateCustomer(Customer customer) {
3          return true;
4      }
5  }
6
7  public class CompanyService {
8      public boolean{Company<-*} validateCustomer{Company<-*}(
           Customer{Company<-*} customer) {
9          return true;
10     }
11 }
```

Listing 4.1: Plain Java code (1-5) and equivalent Jif code (7-11) with labels

right. This is because there is no "right" policy for the system, as it highly depends on the set goal. As discussed earlier, the simplest form of policy is labelling all data as owned by a single principal, which by itself is a perfectly valid policy, but achieves absolutely nothing.

In our example we want to protect personal data of a client by defining a policy that only allows data, that is absolutely necessary to perform useful work, to be disclosed to particular services. We achieve this by defining two principals in the system: *Post* and *Company*, each used for labelling data originating from the Postal Service and the Company Service respectively. There is no principal for either the Shop Service or the client, because data flowing trough the Shop Service and reaching the client has to have integrity of both *Post* and *Company*. In other words, the process needs to be started with permissions of both *Post* and *Company* principals. It is not possible to define principals that have permissions of other principals, therefore the concept of *Shop* or *Client* principals is redundant, even though it may seem not obvious at the first look.

The information that needs to be exposed to both the Company Service and the Postal Service, it assigned a label of $\{Company \leftarrow \top \sqcap Post \leftarrow \top\}$. The information that is only to be exposed to one of them, is labelled either $\{Company \leftarrow \top\}$ or $\{Post \leftarrow \top\}$. Data types and labels are discussed in detail in the following section.

4.4 Data types

This section lists definitions of the data types in the system, and their labels according to the decentralised label model. The definitions are purely fictional

and may not reflect real world very closely. They are used only for demonstration purposes.

The data formats are not homogeneous, as in every entity uses its own classes, but the contained information is intentionally compatible, so it can be mapped between classes without running into a format conversion overhead. Definitions are provided in a syntax that resembles that of Jif.

4.4.1 Data types used by Company Service

Customer data type is used by the Company to store the data of its customers. It consists of a unique personal ID, a first name, and a last name.

- *String* $\{Company \leftarrow \top\}$ personal ID;
- *String* $\{Company \leftarrow \top\}$ first name;
- *String* $\{Company \leftarrow \top\}$ last name;

Product data type is used by the Company to store the data of the products it sells. It consists of a product name and a price.

- *String* $\{Company \leftarrow \top\}$ name;
- *Integer* $\{Company \leftarrow \top\}$ price;

4.4.2 Data types used by Postal Service

Person is a data type used to uniquely identify any particular person. It consists of a unique personal ID, a first name, and a last name.

- *String* $\{Post \leftarrow \top\}$ personal ID;
- *String* $\{Post \leftarrow \top\}$ first name;
- *String* $\{Post \leftarrow \top\}$ last name;

Address is a data type defining a postal address. It consists of a city of residence, a street name, and a number.

- *String* $\{Post \leftarrow \top\}$ city;
- *String* $\{Post \leftarrow \top\}$ street;
- *Integer* $\{Post \leftarrow \top\}$ street number;

4.4.3 Data types used by client

Client contains data identifying a particular client in the system. It is how the client identifies himself in the system. In a real system, this may contain some sort of proof of identity like a cryptographic key, but for the sake of simplicity, it just contains a unique personal ID, a first name, a last name, a city, a street, and a street number.

- *String* $\{Company \leftarrow \top \sqcap Post \leftarrow \top\}$ personal ID;
- *String* $\{Company \leftarrow \top \sqcap Post \leftarrow \top\}$ first name;
- *String* $\{Company \leftarrow \top \sqcap Post \leftarrow \top\}$ last name;
- *String* $\{Post \leftarrow \top\}$ city;
- *String* $\{Post \leftarrow \top\}$ street;
- *Integer* $\{Post \leftarrow \top\}$ street number;

Order is how the client places an order in the Shop. It consists of a product name, and a price, that the client is willing to pay.

- *String* $\{Company \leftarrow \top\}$ product;
- *Integer* $\{Company \leftarrow \top\}$ price;

4.4.4 Data types used by Shop Service

Receipt defines an object returned to the client upon successful completion of an order. It consists of a unique personal ID, a first name, a last name, a city of residence, a street name, a number, a product name, and a price. In other words, it is an aggregations of all previously described types. It does not matter if the data is labelled as owned by *Company* or *Post*, because the client runs with the integrity of both, and can thus access it.

- *String* $\{Company \leftarrow \top\}$ personal ID;
- *String* $\{Company \leftarrow \top\}$ first name;
- *String* $\{Company \leftarrow \top\}$ last name;
- *String* $\{Post \leftarrow \top\}$ city;
- *String* $\{Post \leftarrow \top\}$ street;
- *Integer* $\{Post \leftarrow \top\}$ street number;
- *String* $\{Company \leftarrow \top\}$ product;
- *Integer* $\{Company \leftarrow \top\}$ price;

4.5 Results

During this case study an example online shop system was designed and implemented in three different technologies: plain Java without any information-flow control, Jif with information-flow control, and Java-based Web services without information flow control. It was shown that information-flow control can be applied to existing solutions, and that monolithic tightly-coupled systems can be converted to distributed loosely-coupled systems in a straightforward manner, while retaining identical functionality.

It is important to have a reference implementation when designing a new type of system to be able to verify the correctness of a new design and implementation in a similar fashion as it is done with unit testing. It is a well known fact that unit testing is not a definite answer for verifying correctness of an implementation, but it is a time-tested well understood approach that proves to be "good enough" in most real-world applications. So we claim that taking a unit testing-like approach to designing a new type of system is as reliable as unit testing itself.

In the next chapter we discuss the design and implementation of the hybrid solution of a distributed loosely-coupled information-flow control-aware system, and how reference implementation can be used to check validity of the design and implementation.

Chapter 5

Design and implementation

In this chapter we discuss the design and implementation of an information-flow control-aware distributed loosely-coupled system. We discuss how the decentralised label model can be applied to Web service-based systems, and the challenges involved.

5.1 Decentralised label model in loosely-coupled systems

Jif is an extension of Java language to support the decentralised label model. But unlike many theoretical proposals, it actually provides a usable implementation of a complete software development kit, consisting of the language itself, a compiler, and a runtime environment. This means that real-world applications can be written in Jif to take advantage of information-flow control based on the decentralised label model. Moreover, due to similarity to Java, existing programs often can be ported to Jif, as it was demonstrated in the case study.

In this section information-flow control meta-data usually means Jif labels, but the more general, although longer, term is preferred, because it is not always

the case. In a more general sense, it may also include additional data structures, and language constructs such as declassification or endorsement operations.

We treat the Jif language and the example system implemented in Jif as references for all proposals and solutions discussed in this chapter. This is often expressed as direct mapping between Jif and the implementing technology and language, and the other way round. Because an implementation of the decentralised label model in loosely-coupled systems does not yet exist, being able to perform such a mapping between Jif and other technologies is important when verifying the solution and comparing it to Jif itself.

5.2 Adding information-flow control meta-data

To be able to perform information-flow control in a system, the data has to carry information-flow control meta-data. This section discusses different approaches considered for adding meta-data to the data, their advantages, disadvantages, and the final solution.

5.2.1 Adding meta-data at run-time

Web services do not provide a standard way to exchange meta-data associated with messages, but there is no limitation as to what kind of data can be exchanged. This means that bean classes can be extended to include whatever required meta-data within the exchanged information itself. Doing this by hand is a laborious task because it would require redesigning and reimplementing the way program handles data. However, the impact of the changes may be marginalised by employing automatic code generation. While this is may not significantly reduce the amount of work by absolute value, it does shift the focus to one specific entity – the code generator – rather than scattering the changes all over the code. This allows for a fairly generic solution that would allow to extend and modify the application at a later point without spending a considerable amount of time implementing meta-data exchange in new components to be consistent with the rest of the application. This section describes how this could be achieved in a Java-based implementation. The meta-data is assumed to be in Jif label format.

Since Java version 1.5, it provides support for annotations. Annotation types are specialized interfaces used to annotate declarations such as packages, variables, methods, classes, etc. Such annotations are not permitted to affect the semantics

of programs in the Java programming language in any way. However, they provide useful input to various tools [20], and can be seen as code pre-processor directives.

One of the most commonly used annotation in Java is the *@Override* annotation. It informs the compiler that the method in question is meant to override a method declared in a superclass, or in an interface that the class is implementing. The annotation is generally not required, but it helps preventing errors when an overriding method fails to correctly override a method in a superclass.

```
1 @Override
2 public boolean overridingMethod() {
3   return true;
4 }
```

Listing 5.1: Example of an anotation in Java

A more interesting example is the *@WebService* annotation, which is used to expose a plain Java class as a Web service, and the *@WebMethod* annotation to expose a particular method as a method of the Web service. As can be seen in the example, there is nothing unusual about the code except for the annotations. While in fact all of the boilerplate code required to expose it as a Web service is auto-generated.

```
1 @WebService()
2 public class HelloWorld {
3
4   @WebMethod()
5   public String helloWorld() {
6     return "Hello, World!";
7   }
8 }
```

Listing 5.2: Example of a Web service in Java

The annotations are processed by the Annotation Processing Tool, which is available as a command-line tool since Java 1.5 and as part of standard Java distribution since 1.6. It provides hooks for plugging into the processing process and also allows implementing custom annotations. We employ this property to customize the way Web services are generated in Java to add information-flow meta-data.

The goal can be achieved using a relatively simple trick. To be able to customize the code before it is processed by the standard Java Web service annotation processor, we create clones of *@WebService* and *@WebMethod* annotations and add additional parameters, in this case – information-flow control labels.

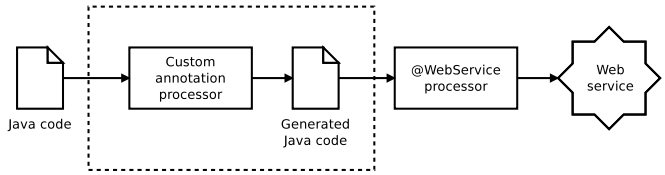

Figure 5.1: Plugging into Java Web service annotation processing

```
1  @Retention(value = RetentionPolicy.RUNTIME)
2  @Target(value = {ElementType.TYPE})
3  public @interface LabelledWebService {
4    // Original fields from @WebService annotation.
5    ...
6    // Additional field for information-flow control label.
7    public String label() default "{}";
8  }
9
10 @Retention(value = RetentionPolicy.RUNTIME)
11 @Target(value = {ElementType.METHOD})
12 public @interface LabelledWebMethod {
13   // Original fields from @WebMethod annotation.
14   ...
15   // Additional field for information-flow control label.
16   public String label() default "{}";
17 }
```

Listing 5.3: Defining custom annotations in Java

This allows for a simple substitution of standard annotations with custom ones, like in the following example. Please note that the example is just a proof of concept and is not supposed to follow any information-control model in particular.

In this way, the class will not be picked up by the standard processor, but rather our own custom processor that will in turn generate modified code with proper Web service annotations, that in turn will be processed into a Web service.

The idea is that an additional parameter is added to the Web service methods, which is used to pass the information-flow control meta-data. The original code is not overwritten, but is reused. The generated code acts as a wrapper of the original code, which checks the labels, and if the checks succeed, it call the original method.

This is a fairly powerful approach that allows for flexible solutions with minimal changes to the Java classes themselves. In fact, the initial conversion of the

```
1 @LabelledWebService(label = "{Hello <- *}")
2 public class HelloWorld {
3
4   @LabelledWebMethod(label = "{Hello <- *}")
5   public String helloWorld() {
6     return "Hello, World!";
7   }
8 }
```

Listing 5.4: Replacing standard annotations with custom ones

```
1 @WebService()
2 public class LabelledHelloWorld {
3
4   @WebMethod()
5   public String labelledHelloWorld(AccessCredentials ac) {
6
7     // Check if labels satisfy the policy.
8     if (ac.compatibleWith("{Hello <- *}")) {
9       // If yes, call the original method.
10      return HelloWorld.helloWorld();
11    } else {
12      // If not, throw exception.
13      throw new LabellingException();
14    }
15  }
16 }
```

Listing 5.5: Generated code with standard annotations

code is just a search-and-replace operation followed by manually adding the labels. The latter cannot be automated anyway, because it depends on particular policy that we want to enforce. Moreover, the code generator can be updated without modifying the rest of the static (non-generated) code. Unfortunately this approach also has strong disadvantages.

First of all, it changes the signatures (and possibly names) of classes and methods. This breaks the client, and may require extensive re-writes for system to continue functioning. It essentially puts the burden of implementation on the client. This is bad because there is likely to be more than one client, and each one of them has to implement a compatible information-flow control system. This is a laborious task and involves a lot of duplicated effort on each client.

Secondly, it relies on run-time checks. It means that it involves run-time overhead, which is likely not be a big problem taking in mind the computing power available nowadays. However it also means that the checks are being performed during information exchange, so failure in policy checks may lead to information

leaks. The leaks can be hard to debug and fix, because the code that performs label validation is auto-generated, and the code generator may propagate bugs to entire system.

5.2.2 Adding meta-data at compile-time

The original goal is to allow the interconnection of system components that support information-flow control and the ones that do not. We also want to do it in a platform-agnostic way to take advantage of interoperability offered by the Web service architecture. Previously described approach goes against these principles, and therefore it is of limited use in this case.

Since programmatic run-time check-based approach proves to be inadequate, the question is if the meta-data can be added statically, and preferably avoiding changes to the code? And the answer is yes, it can be included in WSDL definition, or more specifically the XML schema inside WSDL. W3C recommendation for XML schema [14] [35] [4] specifies a way to add a *documentation* element to elements defined in XSD, which consists of human-readable and machine-readable sections. The *appinfo* element can be used to provide information for tools, style-sheets and other applications [14]. We use this to add information-flow meta-data to type and method definitions.

But before adding labels, we need to investigate how Web-service interfaces are exported via WSDL and XSD definitions. Web service architecture is platform-independent, so every programming technology, that supports Web services, must provide means to map program code to WSDL and XSD constructs. Let us take a look at how it is done in Java.

As can be seen in table 5.1, the mapping is not straightforward. For example, not all Java classes and constructs have mappings to WSDL and some Java classes and constructs have multiple mappings to WSDL [25]. However, all Java types are mapped to XSD *elements* in one way or another. Moreover, methods are *messages*, which are also *elements* defined in XSD. This means that both methods and types can be labelled inside XSD definitions in a consistent manner.

The only requirement for the *appinfo* element is that its contents are valid XML [35]. Unfortunately Jif labels can contain characters that may interfere with XML validation. Therefore we define a simple mapping between Jif labels and their XML equivalents as shown in table 5.2.

Hooking into generation of WSDL and XSD definitions has several strong advan-

Java construct	WSDL and XML construct
Service Endpoint	Interface wsdl:portType
Method	wsdl:operation
Parameters	wsdl:input, wsdl:message, wsdl:part
Return	wsdl:output, wsdl:message, wsdl:part
Throws	wsdl:fault, wsdl:message, wsdl:part
Primitive types	xsd and soapenc simple types
Java beans	xsd:complexType
Java bean properties	Nested xsd:elements of xsd:complexType
Arrays	JAX-RPC defined array xsd:complexType
User defined exceptions	xsd:complexType

Table 5.1: Mapping from Java to WSDL and XML constructs [25]

Symbol	Jif syntax	XML equivalent
$\top$	*	top
$\bot$	_	bottom
p, q	p,q	p,q
$o \rightarrow r$	o:r *or* o->r	o reader r
$o \leftarrow w$	o<-r	o writer w
$o \rightarrow r \sqcup o^1 \rightarrow r^1$	o->r;o1->r1	o reader r join o1 reader r1
$o \leftarrow w \sqcup o^1 \leftarrow w^1$	o<-w;o1<-w1	o writer w join o1 writer w1
$o \rightarrow r \sqcap o^1 \rightarrow r^1$	o->r meet o1->r1	o reader r meet o1 reader r1
$o \leftarrow w \sqcap o^1 \leftarrow w^1$	o<-w meet o1<-w1	o writer w meet o1 writer w1
{c;d}	{c;d}	<label>c;d</label>
	authority (c,d)	<authority>c;d</authority>

Table 5.2: Mapping Jif labels [8] to XML equivalents

```
1 <xs:element name="elementName" type="tns:typeName">
2   <xs:annotation>
3     <xs:appinfo>
4       <label>...</label>
5     </xs:appinfo>
6   </xs:annotation>
7 </xs:element>
```

Listing 5.6: Any element can be annotated with *appinfo*

tages. It allows for a fairly transparent implementation that would not require changes to the client in order to continue functioning. It also exports all the information-flow control meta-data at compile-time, so it does not introduce run-time overhead, and also makes all the meta-data available for static checking before any information exchange is actually performed.

One shortcoming is that while Java runtime does provide hooks for plugging into WSDL generation, it does not cover all aspects of it. For example, it is not possible to hook into generation of XSD. Meaning that it is not possible to add meta-data to type definitions. Working around this may require changes to internal classes of Java runtime, which would make the solution completely not portable, and would likely break existing applications. It may also rely on obscure solutions like parsing SOAP message at run-time in order to add the required meta-data, which may introduce significant run-time bottle-necks. Both of these hacks are obscure enough for them to be unacceptable in many real-world applications.

5.2.3 Adding meta-data in implementation-independent way

So far we have discussed ways of adding the information-flow control meta-data to Java-based systems, but the original goal is to enable this functionality independently of the platform of the implementation. We also want to be able to integrate components that do not support information-flow control. This is not possible if the solution relies on platform-specific features, such as code generation in Java.

This does not contradict the approach, based on adding the meta-data to WSDL and XSD definitions, discussed earlier, but it has to be extended to cover both cases: when a system component supports information-flow control and want to export the meta-data via Web services, and when it does not provide such support.

The first case is covered by the compile-time, and partially run-time code generation, approach discussed earlier. It may prove challenging in that it is likely to require some work-arounds, but assuming that the component already implements information-flow control, exporting this information in a specific format is definitely a reasonably doable task.

The second case is more problematic because if the component in question does not support information-flow control, it cannot supply the required meta-data. Implementing such support in the component itself is likely to require major rewrites or even reimplementation in a different technology to be possible. This is not desirable in absolute majority of cases.

However, the fact that the meta-data can be added to standard WSDL and XSD definitions, lets us exploit one interesting feature of the Web service architecture: the WSDL and XSD definitions used by the consuming entity do not have to be the same definitions that are exported by a service. Rather, they just have to be *compatible*, meaning that they map to the same methods and data types. This property is actually utilised in pretty much every Web service-based system, where WSDL definitions are cached on the consuming side, instead of reading them from the service every time a remote method call is performed.

What this means to us, is that the information-flow control meta-data can be added on the consuming side, because annotating the type definitions with the *appinfo* element cannot invalidate the WSDL or XSD definitions [35]. As can be seen in figure 5.2, adding the meta-data on the consuming component or using the meta-data exported by the web service itself produces equivalent results. In both cases the meta data is made available for checking and is cached locally at the consuming component.

This Web service-consuming component could be the client itself, but it is rarely the case that a client would communicate directly with all the components in the system. It is more often that the client communicates with a single intermediate component that in turn executes a complex business process to deliver a result. We assume that all information is in fact performed though such an intermediate component, and designate it as the policy-enforcing component.

This intermediate component in our example is the Shop Service, and we know that the previous assumption does hold in the example system. In that case, the Shop Service has to be the only component in the system that does support information-flow control. This is because it can enforce the policy without requiring cooperation from the other system components.

Let us look at a simple example. In our example system the Company Service is not intended to receive the city of residence of the customer. However, this

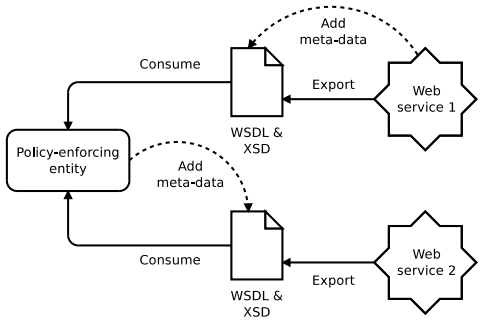

Figure 5.2: Adding meta-data to WSDL and XSD definitions

information could be passed on in a field meant for the name of the customer, because both are *string* data, and in traditional systems that would succeed. However, in an information-flow control-aware system, this would conflict with the policy and would fail. Since the assignment is performed at the policy-enforcing component, and can be verified statically before the actual exchange takes place, the data leak would be detected and the process would not be allowed to run. The receiver of the data (the Company Service) would not be aware of any of this, because it would never receive any data that is not policy-compliant.

5.3 Information-flow control inside a BPEL process

Web Service Business Process Execution Language (WS-BPEL or BPEL) is an XML-based Web service orchestration language. XML-based languages are not widely widespread among application developers as their main implementation language, but BPEL does not even try to be one. It has a very specific goal, which is to allow defining business processes in Web service-based systems, and does it well. Therefore is often used as a technology for integrating Web service-based components into a system.

Being an XML-based language is an advantage is our case, because this allows for direct use of the information-flow control meta-data exported via WSDL interfaces, and for a consistent representation of the meta-data across BPEL, WSDL, and XSD definitions. It also means that a policy validator can extract all the required meta-data just by parsing the linked XML files.

BPEL does not natively support any sort of information-flow control, so the required constructs are also missing in the language, but because BPEL is an extensible language the missing constructs can be added [26]. Any element that can be defined in XSD can be used as an extension element in BPEL.

The extension elements can be defined to be treated by the BPEL parser in two ways: either require support in the BPEL runtime, or simply ignore them. Requiring run-time support is needed when extension elements are an integral part of the process, and make the process non-executable on standard runtime. If they are merely used as meta-data for an external tool, then requiring run-time support is not necessary.

BPEL also provides a way to extend some of the standard constructs with additional elements. For example the *assign* operation defines an *extensionAssignOperation* element that can contain any custom-defined extension element. We use this to allow for declassification and endorsement of the labels during assign operations. See the example on how a custom element can be defined in an external XSD file, and used inside an *assign operation*. Elements for *endorse* and *declassify* are equivalent in definition, but differ in meaning.

```
1 <xsd:complexType name="endorse">
2   <xsd:sequence>
3     <xsd:element name="fromLabel" type="xsd:string"></xsd:element>
4     <xsd:element name="toLabel" type="tns:string"></xsd:element>
5   </xsd:sequence>
6 </xsd:complexType>
7
8   <xsd:element name="endorse" type="tns:endorse"></xsd:element>
```

Listing 5.7: Defining a custom *endorse* element

```
1 <assign name="AssingWithEndorse">
2   <copy>
3     <from variable="var1"/>
4     <to variable="var2"/>
5   </copy>
6   <extensionAssignOperation>
7     <b4j:endorse>
8       <b4j:fromLabel>C writer top</b4j:fromLabel>
9       <b4j:toLabel>C writer top meet P writer top</b4j:toLabel>
10    </b4j:endorse>
11  </extensionAssignOperation>
12 </assign>
```

Listing 5.8: Endorsement as part of assignment

A limitation of this approach is that in BPEL a single *assign* operation can have multiple *copy* operations. This cannot be done here, because it would

require a way to track which *copy* element corresponds to which *endorse* or *declassify* element. This would considerably complicate the task, and would not be consistent with Jif. Instead, we say that if an *assign* operation involves endorsement or declassification, it can only operate on a single pair of variables. If this is not the case, the policy validator has to treat this situation as an error. In practise this should not be a problem, because *assign* operations with multiple *copy* elements can be split into separate *assign* operations, without any difference in functionality, except that such assignments can only be performed in sequence, not in parallel. If the system has to perform a large enough amount of declassifications or endorsements, for it to be a problem, it may indicate a larger problem in the design of a system or the information-flow control policy.

Another more missing piece is assigning labels to variables. There is no explicitly defined way to extend a variable definition, but a *variable* element can contain any XML data, most commonly used for the *documentation* element. Because *documentation* is a rarely used in variable definitions, we replace it with our *label* extension element.

```
1 <xsd:element name="label" type="xsd:string"></xsd:element>
```

Listing 5.9: Defining a *label* extension element

```
1 <variable name="var1" xmlns:tns="..." messageType="tns:varType">
2   <b4j:label>Company writer top</b4j:label>
3 </variable>
```

Listing 5.10: Assigning a label to a variable in BPEL

The described approach has one strong advantage – it does not interfere with the execution of the process, and does not change its meaning. It can still be executed on standard a runtime, and produces the same results. The alternative of replacing elements, rather then extending them, would give a stronger control over the process and would allow for simpler syntax validation, but it would inevitably break the process on a standard runtime. Requiring a custom runtime is not a portable approach, and would likely impair the adoption of the solution quite a bit.

BPEL exports the client-facing interfaces via WSDL definitions as Web services, so the same approach of assigning information-flow control labels to WSDL and XSD definitions, as discussed earlier, can also be used here, thus completing the tool-kit with labelled methods and data types.

5.4 Mapping between Jif and XML-based languages

It is important to have a reference when designing a new type of a system, but it only makes sense if they are comparable. To be able to compare XML-based and Jif-based implementations we need to define how equivalent functionality is mapped between them.

5.4.1 Mapping between Jif and BPEL

Mapping between different programming languages is a fairly difficult task that is best illustrated by an example of a compiler. A compiler translates high-level program code to assembly commands. A good compiler does not change the meaning of the program, but it is a fact that different compilers, or even the same compiler with different options, produce different set of assembly commands given the same input.

Mapping between two high-level programming languages is a very similar, but possibly even more challenging task. It deals with the same problems of mapping between different ways to express the same thing, but must also take into account that not all high-level programming languages support the same subset of expressions. This means that what is a single statement in one programming language may have to be mapped to a sequence of statements in another. The opposite is even more difficult, because a converter has to locate these possible sequences among all possible combinations of statements.

However, we need to be able to map between Jif and BPEL, because that is how the information-flow control concepts of Jif can be applied to BPEL. As can be seen in table 5.3, not all BPEL constructs can be mapped directly to Jif, and not all Jif constructs can be mapped to BPEL. We solve the problem of mapping information-flow control-specific constructs by adding custom extension elements to BPEL, but that does not affect the problem in any significant way.

One of the biggest limitations of Jif is that it is single-threaded, whereas BPEL is heavily multi-threaded. Therefore it is not possible to map the parallel-acting BPEL constructs directly. One possible solution is to perform the actions sequentially in Jif, but that is likely to be very complex, especially when dealing with timeouts, and message listeners.

BPEL contruct	Jif construct
Actions	
Empty	Empty statement
Invoke	Call method
Receive	Method being called
Reply	Return statement
Assign	Assignments to variables
Validate	N/A
Control	
If	If statement
Pick	If statement *or* case statement
While	While statement
For Each	For statement
Repeat Until	Do while statement
Wait	N/A *or* Thread sleep in Java
Sequence	N/A
Scope	N/A *or* Declassification
Flow	N/A *or* Threads in Java
Faults	
Exit	System exit
Throw	Throw statement
Rethrow	Throw statement
Compensate	N/A *or* Complex exception handling
Information-flow control	
N/A *or* Extension	Declassify
N/A *or* Extension	Endorse
N/A *or* Extension	Labels

Table 5.3: Mapping BPEL to Jif

Another problem is that types are handled differently in BPEL (and Web services in general) than they are in Jif and Java. For example what is defined as two separate types in Jif or Java, can be mapped to a single *message type* definition in WSDL. This is a general problem encountered by all platforms that provide support for Web service architecture. In fact, there are solutions such as Java Architecture for XML Binding (JAXB) that do exactly that. It is largely a solved problem, and adding Jif support to existing XML binding frameworks is definitely a doable task. Unfortunately, these XML binding frameworks do nothing about mapping between different programming languages.

As a rule, each implementation of BPEL runtime provides their own BPEL parser and compiler. For example, Apache ODE (Orchestration Director Engine) [16] project provides BOM (BPEL Object Model) parser and compiler; Petals Link provides EasiestDemo [38] – an open source BPEL to Java generator; Eclipse hosts B2J (BPEL to Java) [19] sub-project in very early stages of development; and probably many others. However, all these project deal with an inherently different problem. Their goal is to allow execution of BPEL processes, either by compiling them to executables directly, or producing executable Java code, which is far from just being a map of language constructs. Adapting any of these solution is no easier than writing one form scratch, which would involve a tremendous amounts of effort.

The problem is further complicated by the fact that BPEL allows for writing expressions not only in XPath, but basically any scripting language including, but not limited to JavaScript. Mapping these expression requires parsing or interpreting them, which can be particularly problematic if they return data based on runtime criteria.

We claim that this problem is far too complex to be solved in a generic way. Instead, we limit the constructs we want to map to a very specific subset that is enough to demonstrate the concept. We discard any constructs that cannot be mapped, and also the ones that are not used in the example system. This leaves *invoke*, *receive*, *reply*, *assign*, *if*, *declassify*, *endorse*, and labels. BPEL construct *sequence* is present in every BPEL process, but because we omit all parallel-acting constructs, the *sequence* element has no meaning beyond being a generic container for other elements.

5.4.2 Mapping between Jif and XSD

Earlier we discussed how Jif labels can be represented in XML-based language, and how these labels can be included in XSD. This section explains how exactly the labels are mapped. Let us take a look at a simple Web service class as it

would be defined in Java. We have a single method that takes a parameter, and returns a boolean value. All these components are assigned an information-flow control label.

```
1 public class CompanyService {
2   public boolean{returnLabel} validateCustomer{beginLabel}(Customer
        {parameterLabel} customer) : {endLabel} {
3     return true;
4   }
5 }
```

Listing 5.11: A simple Web service class with Jif labels

What we refer to as *returnLabel* is a label that will be assigned to an object returned by the method. The *beginLabel* is an upper bound on the program-counter label of the caller, which is associated with very statement in the code, and a lower bound on the side effects of the method [8]. The *endLabel* specifies the program-counter label at the point of termination of the method, and is an upper bound on the information that may be learned by observing whether the method terminates normally [8]. And *parameterLabel* is a label assigned to the method parameter. Most commonly all of these labels are the same, but that is not a requirement. The requirement is that they are consistent in a way that they do not make the information flow impossible.

```
1  public class Customer {
2
3    private String{dataLabel} personalId;
4
5    public String{dataLabel} getPersonalId() {
6      return personalId;
7    }
8
9    public void setPersonalId{dataLabel}(String{dataLabel} personalId
         ) {
10     this.personalId = personalId;
11   }
12 }
```

Listing 5.12: A simple bean class with Jif labels

The same applies to bean classes that will be used in information exchanges. We assign the same *dataLabel* to fields, getters, and setters, because they are just syntactic sugar. In theory it is possible to have different labels, because getters and setters are just regular methods that may perform some additional operations. These, however, would not map to Web services, because Web services do not have a notion of a getter or setter, so any additional code contained within them would be lost.

Now let us take a look at how the code maps to WSDL and XSD definitions. WSDL definition is the same as it would appear in Java-based systems, because all information-flow control meta-data is contained within the imported XSD type definitions. It is possible to have the meta-data inside XSD, because in Web services everything is a message and every message has a type definition. For example *validateCustomer* is a method name and does not have a type by itself, whereas in Web services it is a message, just like everything else. We use this property to consistently assign labels to everything within XSD. The equivalent labels are marked the same, so it is pretty self-explanatory. Please refer to listing 5.14 for an example.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <definitions name="CompanyService" ...>
3     <types>...</types>
4     <message name="validateCustomer">
5         <part name="parameters"
6           element="tns:validateCustomer"/>
7     </message>
8     <message name="validateCustomerResponse">
9         <part name="parameters"
10          element="tns:validateCustomerResponse"/>
11    </message>
12    <portType name="Company">
13        <operation name="validateCustomer">
14            <input message="tns:validateCustomer"/>
15            <output message="tns:validateCustomerResponse"/>
16        </operation>
17    </portType>
18    <binding name="CompanyPortBinding" type="tns:Company">
19        <soap:binding transport=... style="document"/>
20        <operation name="validateCustomer">...</operation>
21    </binding>
22    <service name="CompanyService">...</service>
23 </definitions>
```

Listing 5.13: A simple Web service defined in WSDL

Element *validateCustomerResponse* has no equivalent in Jif or Java, because it is not the same as the returned object. The returned object is referenced by *validateCustomerResponse* type, and there we can see that it is actually a simple boolean type. The end-label is assigned to the response element simply as a convenience, because there is no other place where the end-label could be assigned in a consistent manner, and there is also no other label that would make sense to be assigned to the response element.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <xs:schema ...>
3     <xs:element name="validateCustomer"
4           type="tns:validateCustomer">
5         <xs:annotation>
6             <xs:appinfo>
7                 <label>beginLabel</label>
8             </xs:appinfo>
9         </xs:annotation>
10    </xs:element>
11    <xs:element name="validateCustomerResponse"
12          type="tns:validateCustomerResponse">
13        <xs:annotation>
14            <xs:appinfo>
15                <label>endLabel</label>
16            </xs:appinfo>
17        </xs:annotation>
18    </xs:element>
19    <xs:complexType name="validateCustomer">
20        <xs:sequence>
21            <xs:element name="customer" type="tns:customer">
22                <xs:annotation>
23                    <xs:appinfo>
24                        <label>parameterLabel</label>
25                    </xs:appinfo>
26                </xs:annotation>
27            </xs:element>
28        </xs:sequence>
29    </xs:complexType>
30    <xs:complexType name="validateCustomerResponse">
31        <xs:sequence>
32            <xs:element name="return" type="xs:boolean">
33                <xs:annotation>
34                    <xs:appinfo>
35                        <label>returnLabel</label>
36                    </xs:appinfo>
37                </xs:annotation>
38            </xs:element>
39        </xs:sequence>
40    </xs:complexType>
41    <xs:complexType name="customer">
42        <xs:sequence>
43            <xs:element name="personalId" type="xs:string">
44                <xs:annotation>
45                    <xs:appinfo>
46                        <label>dataLabel</label>
47                    </xs:appinfo>
48                </xs:annotation>
49            </xs:element>
50        </xs:sequence>
51    </xs:complexType>
52 </xs:schema>
```

Listing 5.14: XSD type definitions with Jif labels

5.5 Policy validator

In the previous sections we discussed ways to add the information-flow control meta-data to WSDL and XSD definitions, how to integrate system components by employing BPEL, how to add information-flow policy control-aware constructs to BPEL language, and how to map between BPEL and Jif. The only thing missing is the policy validator itself. This section describes how information-flow control is actually performed, and architecture of the validator.

The basic idea is that the validator should be able to statically verify the information-flow control policy before a system is deployed, effectively preventing information leaks by preventing systems with policy violations from running at all. This approach also avoids requiring support in the runtime environment, what greatly increases portability of the solution, and is likely to make it more acceptable in real-world applications.

Implementing a validator for the decentralised label model is definitely a doable task, proven by the fact that such an implementation already exists in Jif, but it is hardly an easy task. Therefore it would be nice if it would be possible to simply plug an existing policy-validating tool into the validator, and let it do the job. And looking at the only implementation of the decentralised model is a good starting point. The problem is that the validator for the decentralised label model, as found in Jif compiler, expects Jif code as input. So to be able to utilize it, we must first find a way to express the information-flow control-related in Jif language. For this we need a Jif code generator.

The idea behind such a Jif code generator is that it should be able to extract relevant information-flow control-related data from a BPEL process, and express this data in Jif language. For reasons discussed earlier, it is not an easy task to accurately map between different programming technologies, so a slightly different approach has to be taken here. Instead we treat the generated code as a model of the system, and the code generator as a modelling tool. And we also need to accept the limitations of a modelling approach, because the accuracy of the results is highly dependent on how well the model represents reality.

An obvious issue with this approach is that not all required code can be auto-generated. The most obvious part being the client code – the initiator of the process. However, the client-facing interfaces are exported via Web services by the BPEL runtime, so a large portion of the client code can actually be generated, as it is the case with many Web service-base systems. The missing part – actual logic – needs to be treated as if it was a unit test. That is, calling every method of the service with mock objects, instead of the actual data. This

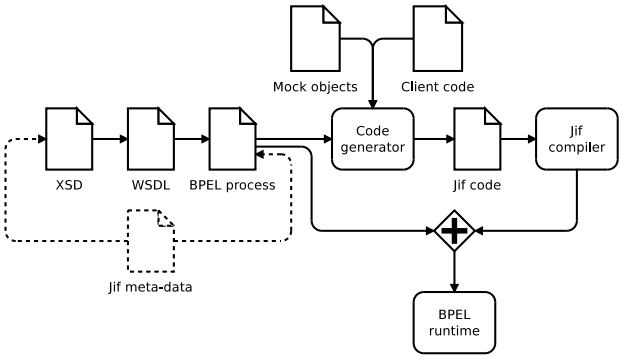

Figure 5.3: Architecture of information-flow policy validator

could be automated for primitive and simple types such as *integer* or *string* by sending random values. In other cases *null* values could be sent. However, this gets more complicated with more complicated data structures, so we just assume the code is manually written, as it is the case with unit testing nowadays.

The same approach is used for modelling the called services, such as the Company Service and the Postal Service in our example. The difference is that the server side code also needs to be generated in addition to client code. Yet again, it can be modelled without actually knowing what manipulations are made on the data inside the component. As long as pre-conditions (begin label) and post-conditions (end label) are satisfied, mock objects can be used, because the policy is content-agnostic.

Even though we are talking about generating Jif code, it does not imply that the client does in fact support information-flow control. It could very well be implemented in plain Java and contain no support for the decentralised label model. The process would still run, because of the previously described approach that allows connecting system components that do not natively support any information-flow control. This also includes the client.

All client-related data types and method definitions of the service are labelled in accordance to the scheme described earlier, and serve the same purpose as if they were originating form the client itself. This approach does not allow the client to set its own information-flow control policy, but since it is available to the client before engaging in actual data exchange, it can be verified, and if it does not meet the policy set by the client, avoid performing the data exchange.

During the validation stage, the model of the system, in a form of Jif code,

is compiled by Jif compiler, which includes verification of the information flow within the system. We claim that if the model of the system passes the validation, so should the system itself. This claim is extremely difficult to prove, because only one counter-example is needed to disprove it. Any mismatch between the model and the actual system may result in information leaks, or other policy violations. Probability of this happening can be reduced by improving quality of the code generator, and reducing the need to perform manual modifications to the generated code.

5.6 Implementing the system

In theory any Java applications could be converted to Jif application, but there are some challenges. Understanding the way information flows trough the application is an important first step that is hard to make. It introduces new concepts to object-oriented programming that may seem counter-intuitive, and makes one rethink his or her programming habits.

One such concept is the program-counter label – a label associated with every statement of the code. It takes time to get accustomed with the effects of it, because it is an "invisible" label – it is not assigned by a developer, but is always there.

```
1 boolean{P<-*} var1 = false;
2 boolean{Q<-*} var2 = false;
3
4 if (var1) {
5   var2 = true;
6 }
```

Listing 5.15: This code is invalid in Jif

Let us take a look at an example. If we ignore the labels, it is a perfectly valid Java code. But it is not valid in Jif, because of how the program-counter works. When the execution reaches the *if* statement, the program-counter label for the entire block becomes that of the *var1* variable. This makes the assignment to the *var1* variable impossible, because the labels of *var1* and *var2* are not compatible. This gets more complicated with more nested blocks.

Another difference is that input and output streams are also affected by the program-counter. For example the equivalent of Java *System.out* output stream cannot be referenced directly, but needs to be obtained form a *Runtime* class, and is only writeable by a special caller principal, which corresponds to a current

user running the application.

Luckily Jif provides means to explicitly declassify information via *declassify* and *endorse* operations. However, a high number of declassifications within a system may indicate that the system is not really compatible with the idea of information-flow control. Porting such an application to Jif may require big changes in its architecture, meaning that the more complex the system is, the less likely it is to be ported to Jif because of exponentially increasing amount of work required.

The Jif implementation of the example system was not a straightforward port from Java, but rather a circular process. In the first design the main information flow was from the leaf services towards the client. This version of the system proved to be largely incompatible with Jif due to numerous declassifications required to run the application, and was abandoned in favour of a simpler design, which changed the direction of information flows within the system, making data flow from the client towards the leaf services.

The new design greatly reduced the amount of declassifications up to a point that conflicting operations had to be introduced intentionally just to demonstrate the concept. A properly designed Jif application differs form an equivalent Java application quite minimally, in a sense that explicitly setting the labels and performing declassifications becomes redundant, and can be omitted in many cases.

Applying information-flow control principles in Web service-based systems is by any means not a straightforward task, and especially so if compatibility with existing systems is to be maintained. The task consists of several related problems: adding information-flow control meta-data, exchanging it among the system components, and verifying the information-flow control policy. Several different approaches were tried for adding the meta-data, and it was shown that it can be done in a fairly universal way without relying on complex run-time solutions, and retaining compatibility with existing Web service-based systems.

Verifying the information-flow is a more complicated problem. Systems supporting information-flow control, such as Jif, are often based on static analysis of the code to verify that the actual flow matches the one defined by the policy. This is not possible in loosely-coupled system, so an alternative approach needs to be found. The proposed approach involves producing a model of a system, and verifying the model, instead of the system itself. This is a viable approach, but testing its effectiveness would require implementing a more complete tool-kit, and performing extensive testing. Neither of the two are small tasks.

Chapter 6

Evaluation and discussion

In this chapter we discuss the process and the results of this thesis with focus on challenges encountered, and limitations of the solution. We suggest several alternative approaches, and set guidelines for future work.

6.1 Hooking directly into the internal Jif API

The current approach involves a code generation step. In reality it is more useful for demonstrating the concept than anything else. The generated code is human-readable, and is also a valid Jif application that can be compiled and executed, allowing a developer to verify its correctness. However, it adds little to no value to automatic information-flow validation.

The idea behind the code generator is that it converts the information-flow control meta-data to a format, that is accepted by our validator – the Jif compiler. But the compiler also converts the code into its internal representation prior to performing any useful work. By hooking up directly into Jif compiler API [8] we could completely omit the code generation step, and call validator methods directly.

By reducing the number of unnecessary intermediate representations we can

improve reliability, because generating a human-readable code is error-prone, as there is no type-checking – all output is just text. Just putting a semicolon in a wrong place, for example because some data that was assumed to be there was actually *null*, would render the code invalid. This is much more reliable when dealing with object representation of the code tokens, because it eliminates simple mistakes such as putting textual data, where integer is required, and similar. This would allow for detection of mistakes earlier.

Code generation also has a disadvantage that output of the generator needs to be fed to the validator by means of pipes, sockets, or whatever other inter-process communication technology, which unavoidably adds a layer of complexity and another point of failure.

We must also remember that such inter-process communication is by itself prone to information leaks, unless it is done with proper information-flow control-aware technologies. It makes it hard to claim that the system is secure in regard to information flow if the subsystem that performs the validation is itself not secure. It is therefore excluding any error-prone unnecessary steps from the system is a good idea.

6.2 Run-time policy validation

In our approach we do not require run-time support for information-flow control in any of the system components. And this is a very optimistic approach, because some policy violations can only be detected at run-time. For example *NullPointerException* or *IndexOutOfBoundsException* occurring in the system can leak information via covert channels. These have to be caught and handled in Jif, so such leaks are prevented. But if the actual run-time does not perform such checks, these leaks cannot be prevented.

The only way to avoid this problem is actually implementing information-flow control in Web service-based systems. Here SIF, a web application framework based on Jif, shows a lot of potential. Since the implementation of Web services in Java and SIF are largely built on servlets, in theory it is possible to implement a Web service framework in a similar manner to that of SIF.

Such framework would inevitably face a problem of interoperability with existing implementations. Not being able to communicate with other Web service implementations would greatly reduce the possibility of the solution being widely adopted, and would make it only usable for very specific tasks.

Moreover, this approach does not involve BPEL, so it does not allow taking advantage of XML-based technologies to full potential. In reality it is not a problem of functionality, because all that can be done in BPEL can also be done in a traditional programming language such as Java. However, we do lose the consistency of information-flow control meta-data definitions, and we lose the ability to visualise complex systems, like it is possible with BPEL to BPMN mapping. This is especially useful when dealing with information-flow control, because a developer always has to have information flow in mind, and a visual reference can greatly increase productivity, and reduce possibility of mistakes.

Another reason why BPEL is sometimes preferred as an integration technology in large loosely-coupled systems is that doing system integration in a programming language such as Java makes it tempting to add additional logic, such as format conversion, to the integrating component which complicates the architecture of the solution, and makes it difficult to update the system. Whereas with BPEL it may be just a matter of reconnecting the lines in an editor, if talking very generally.

It is important to also be able to use BPEL in information-flow control-aware loosely-coupled Web service-based system, because it is complex enterprise systems that would benefit the most from information-flow control, and it is the same complex enterprise systems that are also very likely to be taking advantage of BPEL. Unfortunately, adding this functionality to BPEL runtime may have a disadvantage of reduced portability of BPEL code, and is not a small task.

6.3 Client-side policy enforcement

When talking about information-flow validation, we designated a specific component for this purpose. In our example system it is the component that the client communicates directly with. The client has to trust this component to actually enforce the information-flow control policy.

There are different methods of establishing trust, such as digital signatures or certificates. However, none of them can actually ensure that the implementation is actually correct, so the client has absolutely no reason to trust this particular component any more than another. In reality the client can only trust itself.

In ideal case it is the client itself that should be able to verify the information flow. However, this is not a straightforward task, because the client would essentially need to have the source code (or compiled code) of the system component to be able to verify its functionality. This is most often not the case in Web

service-based systems.

Requiring the source code of all system components is definitely not an option. From a functionality point of view it is a valid approach. Client would be able to verify the components, and establish trust. But this is only going to work where all applications are open-source. The reality is that many vendors would find it unacceptable to distribute the source code of their applications because of legal, political, and business considerations.

Requiring the compiled code is a bit more realistic, because it is difficult enough to reverse-engineer a compiled binary that the vendors would be willing to provide it to the client. And compiled code does not really prevent the client from verifying it by means of static analysis. However, this also only works in a perfect world. It makes no sense to think of a system component as a single compiled binary. In reality they are likely to be complex compilations of binaries, configuration files, bytecode files, etc. Moreover, not all technologies even have a concept of compiled code.

What is needed here is a way to propagate all information-flow meta-data to the client along with a proof that the received meta-data matches the actual behaviour of the component. In our case propagating meta-data is just a matter of allowing the client to download all labelled WSDL and XSD definitions from all system components. But providing a proof of validity is a complex cryptographic task that is likely to involve some sort of code signing technique and may require development of a new cryptographic protocol.

6.4 Propagation of the meta-data

Previously discussed meta-data propagation problem is actually more important than it may appear at the first glance. In our example system we have a chain of client, validation/integration service, and leaf services. The central component is able to enforce the information-flow control policy in the system because it is in a crossroad between all information flows. But in reality the leaf services may be not leaf at all. They may be exchanging information with any number of entities, and those in turn can be doing the same.

In this case our designated validator component is no longer a central component, and can only influence the data flowing through it. It is completely unaware of other components in the system, that it does not communicate with. Here we run into the same problem as with meta-data propagation to the client, but on larger scale. Therefore it can be claimed that solving one of them is

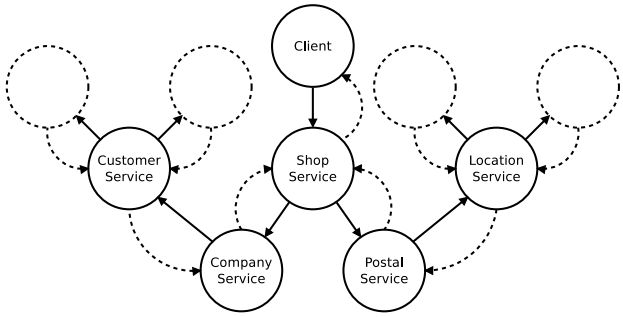

Figure 6.1: Complex system with long chain of components

equivalent to solving both of them.

One possible approach here is to pass on the meta-data from leaf components via intermediate ones while adding their meta-data to the set, repeating the procedure as it travels through all the intermediate nodes. This would also require adding verifiable identity information to the meta-data. Integrity of such a solution would need to be verifiable by cryptographic means, which is commonly addressed problem in internet protocols.

Another option is to aggregate all the meta-data directly at the validator component. Meaning that the validator has to be aware of all the system components, even those it does not communicate directly, and obtain the meta-data directly from every one of them. Technologies such as UDDI (Universal Description Discovery and Integration) [10] could be utilised in discovery of the components, even though UDDI in particular has not been widely adopted and has since been abandoned by it designers.

Either option is viable and may be preferred in certain situations. But it is clear that the issue needs to be resolved for the approach to be considered complete, and become acceptable in real-world applications. It is an important part of the system design, and cannot be considered implementation detail. This is one of the most important issues identified, as it may benefit not only our approach, but in a more general sense could be applied to any similar system, where integrity and trust are essential.

Chapter 7

Conclusions

This thesis aims to tackle a large problem of applying information-flow control techniques to distributed loosely-coupled Web service-based systems. The problem is actually a complex combination of more or less closely related problems, that need to be addressed individually before the general problem can be resolved. These can broken into the following:

- *Adding information-flow control meta-data to Web service-based systems.* It was show that it can be done in a fairly universal manner while retaining compatibility with existing implementations of Web services. Some platforms may allow to do this easier than others, but it can be considered to be implementation detail, and left to platform-specific implementation.
- *Exchanging information-flow control meta-data.* We have shown that making the meta-data available to other system components is not a complex task, and can be achieved with existing technologies. Verifying that the meta-data actually corresponds to actual behaviour of the application is a much more complex problem that involves cryptographic means, such as digital signatures or similar. No universal solution was proposed.
- *Validating information-flow control policy.* This is a complex problem consisting of identifying the point of policy enforcement, and actually enforcing it. We have shown that the most sensible point of policy enforcement

is at the point where all information flows trough. Unfortunately it is not always possible to identify such a point, because even though all system components are known, they themselves may be composed of other components, including external ones. It is also desirable that the policy enforcement could be possible at the client-side, because it is usually the client that is concerned about privacy and integrity of its data. We have proposed an approach where a model of a system is verified instead of the actual system. This is a viable approach, but testing its effectiveness would require implementing a more complete tool-kit, and performing extensive testing. Neither of the two are small tasks.

- *Run-time policy validation.* It became evident that static compile-time policy validation approach is too limiting in some cases, and run-time validation needs to be implemented to overcome these limitations. However, this requires major changes to nearly all run-time components of such a system, and may greatly reduce portability of the solution by making it platform-specific. It is also a major task in sense of effort required to implement it.

Even though the thesis did not resolve all of the problems, identifying them is a first step towards an actual implementation of a distributed loosely-coupled information-flow control-aware system. There are some principle problems, like protocols for information-flow control meta-data exchange, that need to be resolved. But often it is more of a problem of providing a usable implementation rather than defining what needs to be implemented.

In conclusion, information-flow control-aware systems such as Jif, SIF, and Swift provide good confidence that an implementation of an information-flow control-aware distributed loosely-coupled system is definitely doable, and provide a base for more complex and usable systems, that hopefully will become the norm in the future.

Appendix A

XML code

This appendix contains the source code of the example system as it is implemented in BPEL. WSDL and XSD definitions are also included, because they are integral part of a BPEL implementation. Files contained here are the versions with information-flow control extensions.

WSDL and XSD definitions of only the Shop Service (and not Company service and Postal Service), because they all are defined following the same principals, and are equivalent in every way.

A.1 Business process

A.1.1 BPEL definition

This is the business process definition in BPEL with information-flow control extensions. Variables are assigned empty labels for demonstration purposes only. Endorsement is performed as part of *assign* operation.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <process
3     name="ShopBpelModule"
```

```
4     targetNamespace="http://enterprise.netbeans.org/bpel/
          ShopBpelModule/shopBpelModule"
5     xmlns:tns="http://enterprise.netbeans.org/bpel/ShopBpelModule/
          shopBpelModule"
6     xmlns:xs="http://www.w3.org/2001/XMLSchema"
7     xmlns:xsd="http://www.w3.org/2001/XMLSchema"
8     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
9     xmlns="http://docs.oasis-open.org/wsbpel/2.0/process/executable
          "
10    xmlns:sxt="http://www.sun.com/wsbpel/2.0/process/executable/
          SUNExtension/Trace"
11    xmlns:sxed="http://www.sun.com/wsbpel/2.0/process/executable/
          SUNExtension/Editor"
12    xmlns:sxeh="http://www.sun.com/wsbpel/2.0/process/executable/
          SUNExtension/ErrorHandling"
13    xmlns:sxed2="http://www.sun.com/wsbpel/2.0/process/executable/
          SUNExtension/Editor2"
14    xmlns:b4j="http://bpel4jif.bpel.s094758.dtu.dk/"
15    xsi:schemaLocation="http://bpel4jif.bpel.s094758.dtu.dk/
          BPEL4Jif.xsd">
16    <extensions>
17        <extension namespace="http://bpel4jif.bpel.s094758.dtu.dk/"
              mustUnderstand="no"/>
18    </extensions>
19    <import namespace="http://shopservice.shop.s094758.dtu.dk/"
          location="ShopService.wsdl" importType="http://schemas.
          xmlsoap.org/wsdl/"/>
20    <import namespace="http://postalservice.postal.s094758.dtu.dk/"
           location="PostalService.wsdl" importType="http://schemas.
          xmlsoap.org/wsdl/"/>
21    <import namespace="http://companyservice.company.s094758.dtu.dk
          /" location="CompanyService.wsdl" importType="http://
          schemas.xmlsoap.org/wsdl/"/>
22    <partnerLinks>
23        <partnerLink name="PostalPartnerLink" xmlns:tns="http://
              postalservice.postal.s094758.dtu.dk/" partnerLinkType="
              tns:PostalService" partnerRole="PostalRole"/>
24        <partnerLink name="CompanyPartnerLink" xmlns:tns="http://
              companyservice.company.s094758.dtu.dk/" partnerLinkType
              ="tns:CompanyService" partnerRole="CompanyRole"/>
25        <partnerLink name="ShopPartnerLink" xmlns:tns="http://
              shopservice.shop.s094758.dtu.dk/" partnerLinkType="
              tns:ShopService" myRole="ShopRole"/>
26    </partnerLinks>
27    <variables>
28        <variable name="didBuyProduct" xmlns:tns="http://
              companyservice.company.s094758.dtu.dk/" messageType="
              tns:buyProductResponse">
29            <b4j:label></b4j:label>
30        </variable>
31        <variable name="Product" xmlns:tns="http://companyservice.
              company.s094758.dtu.dk/" messageType="tns:buyProduct">
32            <b4j:label></b4j:label>
33        </variable>
```

```
34          <variable name="isValidAddress" xmlns:tns="http://
                postalservice.postal.s094758.dtu.dk/" messageType="
                tns:validateAddressResponse">
35              <b4j:label></b4j:label>
36          </variable>
37          <variable name="AddressAndPerson" xmlns:tns="http://
                postalservice.postal.s094758.dtu.dk/" messageType="
                tns:validateAddress">
38              <b4j:label></b4j:label>
39          </variable>
40          <variable name="isValidCustomer" xmlns:tns="http://
                companyservice.company.s094758.dtu.dk/" messageType="
                tns:validateCustomerResponse">
41              <b4j:label></b4j:label>
42          </variable>
43          <variable name="Customer" xmlns:tns="http://companyservice.
                company.s094758.dtu.dk/" messageType="
                tns:validateCustomer">
44              <b4j:label></b4j:label>
45          </variable>
46          <variable name="Receipt" xmlns:tns="http://shopservice.shop
                .s094758.dtu.dk/" messageType="tns:orderProductResponse
                ">
47              <b4j:label></b4j:label>
48          </variable>
49          <variable name="ClientAndOrder" xmlns:tns="http://
                shopservice.shop.s094758.dtu.dk/" messageType="
                tns:orderProduct">
50              <b4j:label></b4j:label>
51          </variable>
52          <variable name="isValidCustomer2" xmlns:tns="http://
                enterprise.netbeans.org/bpel/ShopBpelModule/
                shopBpelModule" type="xsd:boolean">
53              <b4j:label></b4j:label>
54          </variable>
55          <variable name="isValidAddress2" xmlns:tns="http://
                enterprise.netbeans.org/bpel/ShopBpelModule/
                shopBpelModule" type="xsd:boolean">
56              <b4j:label></b4j:label>
57          </variable>
58      </variables>
59      <sequence>
60          <receive name="ReceiveProductOrder" createInstance="yes"
                partnerLink="ShopPartnerLink" operation="orderProduct"
                xmlns:tns="http://shopservice.shop.s094758.dtu.dk/"
                portType="tns:Shop" variable="ClientAndOrder"/>
61          <assign name="AssignClientToCustomer">
62              <copy>
63                  <from>$ClientAndOrder.parameters/client/firstName</
                        from>
64                  <to>$Customer.parameters/customer/firstName</to>
65              </copy>
66              <copy>
67                  <from>$ClientAndOrder.parameters/client/lastName</
                        from>
```

```
68                  <to>$Customer.parameters/customer/lastName</to>
69              </copy>
70              <copy>
71                  <from>$ClientAndOrder.parameters/client/personalId<
                        /from>
72                  <to>$Customer.parameters/customer/personalId</to>
73              </copy>
74          </assign>
75          <invoke name="ValidateCustomer" partnerLink="
                CompanyPartnerLink" operation="validateCustomer"
                xmlns:tns="http://companyservice.company.s094758.dtu.dk
                /" portType="tns:Company" inputVariable="Customer"
                outputVariable="isValidCustomer"/>
76          <assign name="AssignClientToAddressAndPerson">
77              <copy>
78                  <from>$ClientAndOrder.parameters/client/firstName</
                        from>
79                  <to>$AddressAndPerson.parameters/person/firstName</
                        to>
80              </copy>
81              <copy>
82                  <from>$ClientAndOrder.parameters/client/lastName</
                        from>
83                  <to>$AddressAndPerson.parameters/person/lastName</
                        to>
84              </copy>
85              <copy>
86                  <from>$ClientAndOrder.parameters/client/personalId<
                        /from>
87                  <to>$AddressAndPerson.parameters/person/personalId<
                        /to>
88              </copy>
89              <copy>
90                  <from>$ClientAndOrder.parameters/client/street</
                        from>
91                  <to>$AddressAndPerson.parameters/address/street</to
                        >
92              </copy>
93              <copy>
94                  <from>$ClientAndOrder.parameters/client/
                        streetNumber</from>
95                  <to>$AddressAndPerson.parameters/address/
                        streetNumber</to>
96              </copy>
97              <copy>
98                  <from>$ClientAndOrder.parameters/client/city</from>
99                  <to>$AddressAndPerson.parameters/address/city</to>
100             </copy>
101         </assign>
102         <invoke name="ValidateAddress" partnerLink="
                PostalPartnerLink" operation="validateAddress"
                xmlns:tns="http://postalservice.postal.s094758.dtu.dk/"
                 portType="tns:Postal" inputVariable="AddressAndPerson"
                 outputVariable="isValidAddress"/>
103         <assign name="AssignOrderToProduct">
```

```
104             <copy>
105                 <from>$ClientAndOrder.parameters/order/price</from>
106                 <to>$Product.parameters/product/price</to>
107             </copy>
108             <copy>
109                 <from>$ClientAndOrder.parameters/order/product</
                        from>
110                 <to>$Product.parameters/product/name</to>
111             </copy>
112         </assign>
113         <assign name="EndorseIsValidAddress">
114             <copy>
115                 <from>$isValidAddress.parameters/return</from>
116                 <to variable="isValidAddress2"/>
117             </copy>
118             <extensionAssignOperation>
119                 <b4j:endorse>
120                     <b4j:fromLabel>Company writer top</
                            b4j:fromLabel>
121                     <b4j:toLabel>Company writer top meet Post
                            writer top</b4j:toLabel>
122                 </b4j:endorse>
123             </extensionAssignOperation>
124         </assign>
125         <assign name="EndorseIsValidCustomer">
126             <copy>
127                 <from>$isValidCustomer.parameters/return</from>
128                 <to variable="isValidCustomer2"/>
129             </copy>
130             <extensionAssignOperation>
131                 <b4j:endorse>
132                     <b4j:fromLabel>Company writer top</
                            b4j:fromLabel>
133                     <b4j:toLabel>Company writer top meet Post
                            writer top</b4j:toLabel>
134                 </b4j:endorse>
135             </extensionAssignOperation>
136         </assign>
137         <if name="IfValidAddressAndCustomer">
138             <condition>$isValidCustomer2 and $isValidAddress2</
                    condition>
139             <invoke name="BuyProduct" partnerLink="
                    CompanyPartnerLink" operation="buyProduct"
                    xmlns:tns="http://companyservice.company.s094758.
                    dtu.dk/" portType="tns:Company" inputVariable="
                    Product" outputVariable="didBuyProduct"/>
140         </if>
141         <assign name="PopulateReceipt">
142             <copy>
143                 <from>$Product.parameters/product/name</from>
144                 <to>$Receipt.parameters/return/product</to>
145             </copy>
146             <copy>
147                 <from>$Product.parameters/product/price</from>
148                 <to>$Receipt.parameters/return/price</to>
```

```
149             </copy>
150             <copy>
151                 <from>$Customer.parameters/customer/firstName</from
                        >
152                 <to>$Receipt.parameters/return/firstName</to>
153             </copy>
154             <copy>
155                 <from>$Customer.parameters/customer/lastName</from>
156                 <to>$Receipt.parameters/return/lastName</to>
157             </copy>
158             <copy>
159                 <from>$Customer.parameters/customer/personalId</
                        from>
160                 <to>$Receipt.parameters/return/personalId</to>
161             </copy>
162             <copy>
163                 <from>$AddressAndPerson.parameters/address/city</
                        from>
164                 <to>$Receipt.parameters/return/city</to>
165             </copy>
166             <copy>
167                 <from>$AddressAndPerson.parameters/address/street</
                        from>
168                 <to>$Receipt.parameters/return/street</to>
169             </copy>
170             <copy>
171                 <from>$AddressAndPerson.parameters/address/
                        streetNumber</from>
172                 <to>$Receipt.parameters/return/streetNumber</to>
173             </copy>
174         </assign>
175         <reply name="SendReceipt" partnerLink="ShopPartnerLink"
              operation="orderProduct" xmlns:tns="http://shopservice.
              shop.s094758.dtu.dk/" portType="tns:Shop" variable="
              Receipt"/>
176     </sequence>
177 </process>
```

Listing A.1: Business process in BPEL with information-flow control extensions

A.1.2 BPEL extension definition

This XSD file defines the extension elements used in the BPEL process. Any element that can be defined in XSD can be used as an extension in BPEL.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2
3 <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
4     targetNamespace="http://bpel4jif.bpel.s094758.dtu.dk/"
5     xmlns:tns="http://bpel4jif.bpel.s094758.dtu.dk/"
6     elementFormDefault="qualified">
```

```
7
8       <xsd:complexType name="declassify">
9           <xsd:sequence>
10              <xsd:element name="fromLabel" type="xsd:string"></
                    xsd:element>
11              <xsd:element name="toLabel" type="tns:string"></
                    xsd:element>
12          </xsd:sequence>
13      </xsd:complexType>
14
15      <xsd:complexType name="endorse">
16          <xsd:sequence>
17              <xsd:element name="fromLabel" type="xsd:string"></
                    xsd:element>
18              <xsd:element name="toLabel" type="tns:string"></
                    xsd:element>
19          </xsd:sequence>
20      </xsd:complexType>
21
22      <xsd:element name="declassify" type="tns:declassify"></
            xsd:element>
23      <xsd:element name="endorse" type="tns:endorse"></xsd:element>
24      <xsd:element name="label" type="xsd:string"></xsd:element>
25  </xsd:schema>
```

Listing A.2: Definitions of BPEL extension elements

A.2 Definitions of Shop Service

A.2.1 WSDL definitions of Shop Service

This WSDL file is used to expose the Shop Service implemented in BPEL to the client. It is a standard WSDL file, except for the addition of partner link definitions needed for BPEL, because the information-flow control meta-data resides in an imported XSD file.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <definitions
3       xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-
             wss-wssecurity-utility-1.0.xsd"
4       xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
5       xmlns:tns="http://shopservice.shop.s094758.dtu.dk/"
6       xmlns:xsd="http://www.w3.org/2001/XMLSchema"
7       xmlns:plnk="http://docs.oasis-open.org/wsbpel/2.0/plnktype"
8       xmlns="http://schemas.xmlsoap.org/wsdl/"
9       targetNamespace="http://shopservice.shop.s094758.dtu.dk/"
10      name="ShopService">
11      <types>
```

```
12          <xsd:schema>
13              <xsd:import namespace="http://shopservice.shop.s094758.
                    dtu.dk/" schemaLocation="ShopService.xsd"/>
14          </xsd:schema>
15      </types>
16      <message name="orderProduct">
17          <part name="parameters" element="tns:orderProduct"/>
18      </message>
19      <message name="orderProductResponse">
20          <part name="parameters" element="tns:orderProductResponse"/
                >
21      </message>
22      <portType name="Shop">
23          <operation name="orderProduct">
24              <input message="tns:orderProduct"/>
25              <output message="tns:orderProductResponse"/>
26          </operation>
27      </portType>
28      <binding name="ShopPortBinding" type="tns:Shop">
29          <soap:binding transport="http://schemas.xmlsoap.org/soap/
                http" style="document"/>
30          <operation name="orderProduct">
31              <soap:operation soapAction=""/>
32              <input>
33                  <soap:body use="literal"/>
34              </input>
35              <output>
36                  <soap:body use="literal"/>
37              </output>
38          </operation>
39      </binding>
40      <service name="ShopService">
41          <port name="ShopPort" binding="tns:ShopPortBinding">
42              <soap:address location="http://localhost:8282/
                    ShopService/ShopService"/>
43          </port>
44      </service>
45      <plnk:partnerLinkType name="ShopService">
46          <plnk:role name="ShopRole" portType="tns:Shop"/>
47      </plnk:partnerLinkType>
48  </definitions>
```

Listing A.3: WSDL definitions of the Shop Service

A.2.2 XSD definitions of Shop Service

This XSD defines types used in the Shop Service. This is where the information-flow control meta-data resides. Elements that correspond to methods calls contain a begin label of a method and response element contains an end label of a method.

Please note that the end label and the label of a returned element are not the same. The *orderProductResponse* element contains the end label, and *orderProductResponse* type refers to a *return* element that contains the label of a returned object.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema
    xmlns:tns="http://shopservice.shop.s094758.dtu.dk/"
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    version="1.0"
    targetNamespace="http://shopservice.shop.s094758.dtu.dk/">

    <xs:element name="orderProduct" type="tns:orderProduct">
        <xs:annotation>
            <xs:appinfo>
                <label>Company writer top meet Post writer top</
                    label>
                <authority>Company, Post</authority>
            </xs:appinfo>
        </xs:annotation>
    </xs:element>

    <xs:element name="orderProductResponse" type="
        tns:orderProductResponse">
        <xs:annotation>
            <xs:appinfo>
                <label>Company writer top meet Post writer top</
                    label>
            </xs:appinfo>
        </xs:annotation>
    </xs:element>

    <xs:complexType name="orderProduct">
        <xs:sequence>
            <xs:element name="client" type="tns:client" minOccurs="
                0">
                <xs:annotation>
                    <xs:appinfo>
                        <label>Company writer top meet Post writer
                            top</label>
                    </xs:appinfo>
                </xs:annotation>
            </xs:element>
            <xs:element name="order" type="tns:order" minOccurs="0"
                >
                <xs:annotation>
                    <xs:appinfo>
                        <label>Company writer top meet Post writer
                            top</label>
                    </xs:appinfo>
                </xs:annotation>
            </xs:element>
        </xs:sequence>
    </xs:complexType>

```

```
44      <xs:complexType name="client">
45          <xs:sequence>
46              <xs:element name="city" type="xs:string" minOccurs="0">
47                  <xs:annotation>
48                      <xs:appinfo>
49                          <label>Post writer top</label>
50                      </xs:appinfo>
51                  </xs:annotation>
52              </xs:element>
53              <xs:element name="firstName" type="xs:string" minOccurs
                    ="0">
54                  <xs:annotation>
55                      <xs:appinfo>
56                          <label>Company writer top meet Post writer
                                top</label>
57                      </xs:appinfo>
58                  </xs:annotation>
59              </xs:element>
60              <xs:element name="lastName" type="xs:string" minOccurs=
                    "0">
61                  <xs:annotation>
62                      <xs:appinfo>
63                          <label>Company writer top meet Post writer
                                top</label>
64                      </xs:appinfo>
65                  </xs:annotation>
66              </xs:element>
67              <xs:element name="personalId" type="xs:string"
                    minOccurs="0">
68                  <xs:annotation>
69                      <xs:appinfo>
70                          <label>Company writer top meet Post writer
                                top</label>
71                      </xs:appinfo>
72                  </xs:annotation>
73              </xs:element>
74              <xs:element name="street" type="xs:string" minOccurs="0
                    ">
75                  <xs:annotation>
76                      <xs:appinfo>
77                          <label>Post writer top</label>
78                      </xs:appinfo>
79                  </xs:annotation>
80              </xs:element>
81              <xs:element name="streetNumber" type="xs:int">
82                  <xs:annotation>
83                      <xs:appinfo>
84                          <label>Post writer top</label>
85                      </xs:appinfo>
86                  </xs:annotation>
87              </xs:element>
88          </xs:sequence>
89      </xs:complexType>
90
91      <xs:complexType name="order">
```

```
92          <xs:sequence>
93              <xs:element name="price" type="xs:int">
94                  <xs:annotation>
95                      <xs:appinfo>
96                          <label>Company writer top</label>
97                      </xs:appinfo>
98                  </xs:annotation>
99              </xs:element>
100             <xs:element name="product" type="xs:string" minOccurs="
                    0">
101                 <xs:annotation>
102                     <xs:appinfo>
103                         <label>Company writer top</label>
104                     </xs:appinfo>
105                 </xs:annotation>
106             </xs:element>
107         </xs:sequence>
108     </xs:complexType>
109
110     <xs:complexType name="orderProductResponse">
111         <xs:sequence>
112             <xs:element name="return" type="tns:receipt" minOccurs=
                    "0">
113                 <xs:annotation>
114                     <xs:appinfo>
115                         <label>Company writer top</label>
116                     </xs:appinfo>
117                 </xs:annotation>
118             </xs:element>
119         </xs:sequence>
120     </xs:complexType>
121
122     <xs:complexType name="receipt">
123         <xs:sequence>
124             <xs:element name="city" type="xs:string" minOccurs="0">
125                 <xs:annotation>
126                     <xs:appinfo>
127                         <label>Post writer top</label>
128                     </xs:appinfo>
129                 </xs:annotation>
130             </xs:element>
131             <xs:element name="firstName" type="xs:string" minOccurs
                    ="0">
132                 <xs:annotation>
133                     <xs:appinfo>
134                         <label>Company writer top</label>
135                     </xs:appinfo>
136                 </xs:annotation>
137             </xs:element>
138             <xs:element name="lastName" type="xs:string" minOccurs=
                    "0">
139                 <xs:annotation>
140                     <xs:appinfo>
141                         <label>Company writer top</label>
142                     </xs:appinfo>
```

```
143                 </xs:annotation>
144             </xs:element>
145             <xs:element name="personalId" type="xs:string"
                    minOccurs="0">
146                 <xs:annotation>
147                     <xs:appinfo>
148                         <label>Company writer top</label>
149                     </xs:appinfo>
150                 </xs:annotation>
151             </xs:element>
152             <xs:element name="price" type="xs:int">
153                 <xs:annotation>
154                     <xs:appinfo>
155                         <label>Company writer top</label>
156                     </xs:appinfo>
157                 </xs:annotation>
158             </xs:element>
159             <xs:element name="product" type="xs:string" minOccurs="
                    0">
160                 <xs:annotation>
161                     <xs:appinfo>
162                         <label>Company writer top</label>
163                     </xs:appinfo>
164                 </xs:annotation>
165             </xs:element>
166             <xs:element name="street" type="xs:string" minOccurs="0
                    ">
167                 <xs:annotation>
168                     <xs:appinfo>
169                         <label>Post writer top</label>
170                     </xs:appinfo>
171                 </xs:annotation>
172             </xs:element>
173             <xs:element name="streetNumber" type="xs:int">
174                 <xs:annotation>
175                     <xs:appinfo>
176                         <label>Post writer top</label>
177                     </xs:appinfo>
178                 </xs:annotation>
179             </xs:element>
180         </xs:sequence>
181     </xs:complexType>
182 </xs:schema>
```

Listing A.4: XSD type definitions of the Shop Service with labels

Appendix B

Jif code

This appendix contains Jif code of the example system. This implementation is used as a reference. Please note that stripping any information-flow control-related information from Jif code produces valid Java code.

B.1 Business process

This is a reference implementation of the business process written in Jif.

```
1  package dk.dtu.s094758.shop;
2
3  import dk.dtu.s094758.company.CompanyService;
4  import dk.dtu.s094758.company.Customer;
5  import dk.dtu.s094758.company.Product;
6  import dk.dtu.s094758.postal.PostalService;
7  import dk.dtu.s094758.postal.Address;
8  import dk.dtu.s094758.postal.Person;
9
10 public class ShopService authority (Company, Post) {
11
12     public Receipt{Company<-* meet Post<-*} orderProduct{Company<-*
           meet Post<-*}(Client{Company<-* meet Post<-*} client,
         Order{Company<-* meet Post<-*} order) where authority (
         Company, Post) {
```

```
13
14          CompanyService companyService = new CompanyService();
15
16          Customer customer = new Customer();
17
18          String customerPersonalId = null;
19          String customerFirstName = null;
20          String customerLastName = null;
21
22          try {
23              customerPersonalId = client.getPersonalId();
24              customerFirstName = client.getFirstName();
25              customerLastName = client.getLastName();
26          } catch (NullPointerException ex) {
27          }
28
29          try {
30              customer.setPersonalId(customerPersonalId);
31              customer.setFirstName(customerFirstName);
32              customer.setLastName(customerLastName);
33          } catch (NullPointerException ex) {
34          }
35
36          boolean isValidCustomer = companyService.validateCustomer(
                customer);
37
38          PostalService postalService = new PostalService();
39
40          Address address = new Address();
41
42          String addressCity = null;
43          String addressStreet = null;
44          int addressStreetNumber = -1;
45
46          try {
47              addressCity = client.getCity();
48              addressStreet = client.getStreet();
49              addressStreetNumber = client.getStreetNumber();
50          } catch (NullPointerException ex) {
51          }
52
53          try {
54              address.setCity(addressCity);
55              address.setStreet(addressStreet);
56              address.setStreetNumber(addressStreetNumber);
57          } catch (NullPointerException ex) {
58          }
59
60          Person person = new Person();
61
62          String personFirstName = null;
63          String personLastName = null;
64          String personPersonalId = null;
65
66          try {
```

```
67              personFirstName = client.getFirstName();
68              personLastName = client.getLastName();
69              personPersonalId = client.getPersonalId();
70          } catch (NullPointerException ex) {
71          }
72
73          try {
74              person.setFirstName(personFirstName);
75              person.setLastName(personLastName);
76              person.setPersonalId(personPersonalId);
77          } catch (NullPointerException ex) {
78          }
79
80          boolean isValidAddress = postalService.validateAddress(
                address, person);
81
82          Product product = new Product();
83
84          String productName = null;
85          int price = -1;
86
87          try {
88              productName = order.getProduct();
89              price = order.getPrice();
90          } catch (NullPointerException ex) {
91          }
92
93          try {
94              product.setName(productName);
95              product.setPrice(price);
96          } catch (NullPointerException ex) {
97          }
98
99          boolean boughtProduct = false;
100
101         boolean isValidAddress2 = endorse(isValidAddress, {Company
                <-* meet Post<-*});
102         boolean isValidCustomer2 = endorse(isValidCustomer, {
                Company<-* meet Post<-*});
103
104         if (isValidAddress2 && isValidCustomer2) {
105             boughtProduct = companyService.buyProduct(product);
106         }
107
108         Receipt receipt = new Receipt();
109
110         String receiptPersonalId = null;
111         String receiptFirstName = null;
112         String receiptLastName = null;
113
114         String receiptCity = null;
115         String receiptStreet = null;
116         int receiptStreetNumber = -1;
117
118         String receiptProduct = null;
```

```
119         int receiptPrice = -1;
120
121         try {
122             receiptPersonalId = customer.getPersonalId();
123             receiptFirstName = customer.getFirstName();
124             receiptLastName = customer.getLastName();
125
126             receiptCity = address.getCity();
127             receiptStreet = address.getStreet();
128             receiptStreetNumber = address.getStreetNumber();
129
130             receiptProduct = product.getName();
131             receiptPrice = product.getPrice();
132         } catch (NullPointerException ex) {
133         }
134
135         try {
136             receipt.setPersonalId(receiptPersonalId);
137             receipt.setFirstName(receiptFirstName);
138             receipt.setLastName(receiptLastName);
139
140             receipt.setCity(receiptCity);
141             receipt.setStreet(receiptStreet);
142             receipt.setStreetNumber(receiptStreetNumber);
143
144             receipt.setProduct(receiptProduct);
145             receipt.setPrice(receiptPrice);
146         } catch (NullPointerException ex) {
147         }
148
149         return receipt;
150     }
151 }
```

Listing B.1: Business process as implemented in Jif

B.2 Main class

The role of a client in Jif-based implementation is performed by the *main* class, which initiates the process. The process needs to be started with the authority of both *Company* and *Post* principals.

There is also a caller principal p defined, which is the principal that corresponds to the user that runs the application. It is a system principal, which is a default owner of runtime-related objects such as output stream, so it needs to be taken into account when, for example, outputting to the user interface.

```
1 package dk.dtu.s094758;
2
```

```
3  import java.io.PrintStream;
4  import jif.runtime.Runtime;
5  import dk.dtu.s094758.shop.Client;
6  import dk.dtu.s094758.shop.Order;
7  import dk.dtu.s094758.company.CompanyService;
8  import dk.dtu.s094758.company.Customer;
9  import dk.dtu.s094758.company.Product;
10 import dk.dtu.s094758.shop.ShopService;
11 import dk.dtu.s094758.shop.Receipt;
12 import dk.dtu.s094758.postal.PostalService;
13 import dk.dtu.s094758.postal.Person;
14 import dk.dtu.s094758.postal.Address;
15
16 public class App authority (Company, Post) {
17
18     public static final void main{Company<-* meet Post<-* meet p
           <-*}(principal{} p, String[] args) : {Company<-* meet Post
           <-*} throws (SecurityException, IllegalArgumentException)
           where authority (Company, Post), caller(p) {
19
20         PrintStream[{}] out = null;
21
22         try {
23             Runtime[p] runtime = Runtime[p].getRuntime();
24             out = runtime==null?null:runtime.stdout(new label {});
25         } catch (SecurityException ex) {
26         }
27
28         PrintStream[{}] out1 = endorse(out, {p->*} to {p->*;Company
               <-* meet Post<-* meet p<-*});
29         PrintStream[{}] out2 = declassify(out1, {Company<-* meet
               Post<-*});
30
31       ShopService shopService = new ShopService();
32
33         Client client = new Client();
34
35         String clientPersonalId = "jens123";
36         String clientCity = "Koebenhavn";
37         String clientStreet = "Terrasserne";
38         int clientStreetNumber = 8;
39         String clientFirstName = "Jens";
40         String clientLastName = "Jensen";
41
42         client.setPersonalId(clientPersonalId);
43         client.setCity(clientCity);
44         client.setStreet(clientStreet);
45         client.setStreetNumber(clientStreetNumber);
46         client.setFirstName(clientFirstName);
47         client.setLastName(clientLastName);
48
49         Order order = new Order();
50
51         order.setProduct("ProductXXL");
52         order.setPrice(1200);
```

```
53
54          Receipt receipt = shopService.orderProduct(client, order);
55
56          String personalId = null;
57          String firstName = null;
58          String lastName = null;
59          String city = null;
60          String street = null;
61          int streetNumber = -1;
62          String name = null;
63          int price = -1;
64
65          try {
66              personalId = receipt.getPersonalId();
67              firstName = receipt.getFirstName();
68              lastName = receipt.getLastName();
69              city = receipt.getCity();
70              street = receipt.getStreet();
71              streetNumber = receipt.getStreetNumber();
72              name = receipt.getProduct();
73              price = receipt.getPrice();
74
75          } catch (NullPointerException ex) {
76          }
77
78          if (out2 == null) throw new IllegalArgumentException("Null
                output");
79
80          out2.println(personalId);
81          out2.println(firstName);
82          out2.println(lastName);
83          out2.println(city);
84          out2.println(street);
85          out2.println(streetNumber);
86          out2.println(name);
87          out2.println(price);
88      }
89  }
```

Listing B.2: Main class

B.3 Bean object

This is a bean class that contains the data of a client. It illustrates how labels are assigned to data types that are used in information exchanges. Please note that some data is labelled as accessible only to the *Post* principal, and some to both the *Post* and the *Company* principles. Any mix of labels is allowed as long as it is usable in the context of an application.

```
1 package dk.dtu.s094758.shop;
```

```
2
3 public class Client {
4
5     private String{Company<-* meet Post<-*} personalId;
6     private String{Company<-* meet Post<-*} firstName;
7     private String{Company<-* meet Post<-*} lastName;
8     private String{Post<-*} city;
9     private String{Post<-*} street;
10    private int{Post<-*} streetNumber;
11
12    public String{Post<-*} getCity() {
13        return city;
14    }
15
16    public void setCity{Post<-*}(String{Post<-*} city) {
17        this.city = city;
18    }
19
20    public String{Company<-* meet Post<-*} getFirstName() {
21        return firstName;
22    }
23
24    public void setFirstName{Company<-* meet Post<-*}(String{
          Company<-* meet Post<-*} firstName) {
25        this.firstName = firstName;
26    }
27
28    public String{Company<-* meet Post<-*} getLastName() {
29        return lastName;
30    }
31
32    public void setLastName{Company<-* meet Post<-*}(String{Company
          <-* meet Post<-*} lastName) {
33        this.lastName = lastName;
34    }
35
36    public String{Company<-* meet Post<-*} getPersonalId() {
37        return personalId;
38    }
39
40    public void setPersonalId{Company<-* meet Post<-*}(String{
          Company<-* meet Post<-*} personalId) {
41        this.personalId = personalId;
42    }
43
44    public String{Post<-*} getStreet() {
45        return street;
46    }
47
48    public void setStreet{Post<-*}(String{Post<-*} street) {
49        this.street = street;
50    }
51
52    public int{Post<-*} getStreetNumber() {
53        return streetNumber;
```

```
54      }
55
56      public void setStreetNumber{Post<-*}(int{Post<-*} streetNumber)
            {
57          this.streetNumber = streetNumber;
58      }
59  }
```

Listing B.3: Labelled bean class

Appendix C

Experimental code

This appendix contains experimental code that was written as proof-of-concept, but is not part of the final solution.

C.1 Annotation processor

This is a Java annotation processor that parses a custom-annotated Web service class, and generated additional code according to annotations. It uses Java reflection API to extract the information from original code; the source code is never parsed.

It is a proof-of-concept of how meta-data could be exchanged via Web services in a fairly transparent way form the point of view of a developer.

```
1 package dk.dtu.s094758.lib.processor;
2
3 import dk.dtu.s094758.lib.exception.ClassificationException;
4 import dk.dtu.s094758.lib.credentials.ClassificationLevel;
5 import dk.dtu.s094758.lib.credentials.AccessCredentials;
6 import dk.dtu.s094758.lib.annotation.ClassifiedWebMethod;
7 import dk.dtu.s094758.lib.annotation.ClassifiedWebService;
8 import dk.dtu.s094758.lib.model.ClassifiedVariable;
9 import dk.dtu.s094758.lib.ws.SupportsClassification;
```

```
10 import java.io.IOException;
11 import java.io.Writer;
12 import java.lang.reflect.InvocationTargetException;
13 import java.lang.reflect.Method;
14 import java.util.Iterator;
15 import java.util.List;
16 import java.util.Map;
17 import java.util.Map.Entry;
18 import java.util.Set;
19 import java.util.logging.Level;
20 import java.util.logging.Logger;
21 import javax.annotation.Generated;
22 import javax.annotation.processing.AbstractProcessor;
23 import javax.annotation.processing.Filer;
24 import javax.annotation.processing.Messager;
25 import javax.annotation.processing.ProcessingEnvironment;
26 import javax.annotation.processing.RoundEnvironment;
27 import javax.annotation.processing.SupportedAnnotationTypes;
28 import javax.annotation.processing.SupportedSourceVersion;
29 import javax.jws.WebMethod;
30 import javax.jws.WebService;
31 import javax.lang.model.SourceVersion;
32 import javax.lang.model.element.Element;
33 import javax.lang.model.element.ExecutableElement;
34 import javax.lang.model.element.Modifier;
35 import javax.lang.model.element.Name;
36 import javax.lang.model.element.PackageElement;
37 import javax.lang.model.element.TypeElement;
38 import javax.lang.model.element.VariableElement;
39 import javax.lang.model.type.TypeMirror;
40 import javax.lang.model.util.ElementFilter;
41 import javax.lang.model.util.Elements;
42 import javax.tools.JavaFileObject;
43
44 /**
45  *
46  * @author linas
47  */
48 @SupportedAnnotationTypes(value = {"dk.dtu.s094758.lib.annotation.
        ClassifiedWebService"})
49 @SupportedSourceVersion(SourceVersion.RELEASE_6)
50 public class WebServiceClassificationProcessor extends
        AbstractProcessor {
51
52     private Filer filer;
53     private Messager messager;
54     private Elements elementUtils;
55
56     @Override
57     public void init(ProcessingEnvironment processingEnv) {
58         super.init(processingEnv);
59         filer = processingEnv.getFiler();
60         messager = processingEnv.getMessager();
61         elementUtils = processingEnv.getElementUtils();
62     }
```

```
63
64      @Override
65      public boolean process(Set<? extends TypeElement> annotations,
            RoundEnvironment roundEnvironment) {
66
67          System.out.println("Processing classification annotations
                ...");
68
69          Set<? extends Element> annotatedWebServices =
                roundEnvironment.getElementsAnnotatedWith(
                ClassifiedWebService.class);
70
71          for (Element annotatedWebService : annotatedWebServices) {
72
73              PackageElement packageElement = elementUtils.
                    getPackageOf(annotatedWebService);
74
75              Name packageName = packageElement.getQualifiedName();
76              String classifiedPackageName = packageName + ".
                    classified";
77
78              Name className = annotatedWebService.getSimpleName();
79              Element fullClassName = annotatedWebService;
80
81              JavaFileObject javaFile = null;
82              try {
83                  javaFile = filer.createSourceFile(
                        classifiedPackageName + "." + className);
84              } catch (IOException ex) {
85                  Logger.getLogger(WebServiceClassificationProcessor.
                        class.getName()).log(Level.SEVERE, null, ex);
86              }
87
88              Writer writer = null;
89              try {
90                  writer = javaFile.openWriter();
91
92                  writer.write("package " + classifiedPackageName + "
                        ;\n");
93                  writer.write("\n");
94
95                  ClassifiedWebService classifiedWebService =
                        annotatedWebService.getAnnotation(
                        ClassifiedWebService.class);
96                  String targetNamespace = classifiedWebService.
                        targetNamespace();
97                  String serviceName = classifiedWebService.
                        serviceName();
98                  String portName = classifiedWebService.portName();
99                  String wsdlLocation = classifiedWebService.
                        wsdlLocation();
100
101                 writer.write("@" + Generated.class.getName() + "(\"
                        " + packageName + "." + className + "\")\n");
102                 writer.write("@" + SupportsClassification.class.
```

```
                    getName() + "()\n");
103             writer.write("@" + WebService.class.getName() + "(
                    targetNamespace = \"" + targetNamespace + "\",
                    serviceName = \"" + serviceName + "\", portName
                     = \"" + portName + "\", wsdlLocation = \"" +
                    wsdlLocation + "\")\n");
104             writer.write("public class " + className + "
                    extends " + fullClassName + " {\n");
105             writer.write("\n");
106
107             for (ExecutableElement method : ElementFilter.
                    methodsIn(annotatedWebService.
                    getEnclosedElements())) {
108
109                 Name methodName = method.getSimpleName();
110                 TypeMirror returnType = method.getReturnType();
111
112                 String modifiersString = buildModifiersString(
                        method.getModifiers());
113                 String methodParametersString =
                        buildMethodParametersString(method.
                        getParameters());
114                 String callParametersString =
                        buildCallParametersString(method.
                        getParameters());
115
116                 ClassifiedWebMethod classifiedWebMethod =
                        method.getAnnotation(ClassifiedWebMethod.
                        class);
117                 String operationName = classifiedWebMethod.
                        operationName();
118                 String action = classifiedWebMethod.action();
119                 Boolean exclude = classifiedWebMethod.exclude()
                        ;
120                 ClassificationLevel classificationLevel =
                        classifiedWebMethod.classificationLevel();
121
122                 writer.write("    @" + Generated.class.getName
                        () + "(\"" + packageName + "." + className
                        + "." + methodName + "()\")\n");
123                 writer.write("    @" + WebMethod.class.getName
                        () + "(operationName = \"" + operationName
                        + "\", action = \"" + action + "\", exclude
                         = " + exclude + ")\n");
124
125                 writer.write("    " + modifiersString + " " +
                        returnType + " " + methodName + "(" +
                        methodParametersString + ") throws " +
                        ClassificationException.class.getName() + "
                         {\n");
126                 writer.write("\n");
127
128                 writer.write("        " + ClassificationLevel.
                        class.getName() + " classificationLevel =
                        accessCredentials.getClassificationLevel()
```

```
                    ;\n");
129                 writer.write("\n");
130
131                 writer.write("        if (classificationLevel.
                        getNumericValue() < " + ClassificationLevel
                        .class.getName() + "." +
                        classificationLevel.toString() + ".
                        getNumericValue()) {\n");
132                 writer.write("            throw new " +
                        ClassificationException.class.getName() + "
                        ();\n");
133                 writer.write("        }\n");
134                 writer.write("\n");
135
136                 writer.write("        " + returnType + "
                        returnData = super." + methodName + "(" +
                        callParametersString + ");\n");
137                 writer.write("\n");
138
139                 writer.write("        Class<? extends " +
                        returnType + "> returnClass = returnData.
                        getClass();\n");
140                 writer.write("        " + Method.class.getName
                        () + "[] methods = returnClass.getMethods()
                        ;\n");
141                 writer.write("        " + Map.class.getName() +
                         "<" + String.class.getName() + ", " +
                        ClassifiedVariable.class.getName() + ">
                        classifiedVariableMap = " +
                        WebServiceClassificationUtils.class.getName
                        () + ".buildClassifiedVariableMap(methods)
                        ;\n");
142                 writer.write("\n");
143                 writer.write("        for (" + Iterator.class.
                        getName() + "<" + Entry.class.
                        getCanonicalName() + "<" + String.class.
                        getName() + ", " + ClassifiedVariable.class
                        .getName() + ">> iterator =
                        classifiedVariableMap.entrySet().iterator()
                        ; iterator.hasNext();) {\n");
144                 writer.write("\n");
145                 writer.write("            " + Entry.class.
                        getCanonicalName() + "<" + String.class.
                        getName() + ", " + ClassifiedVariable.class
                        .getName() + "> entry = iterator.next();\n"
                        );
146                 writer.write("            " +
                        ClassifiedVariable.class.getName() + "
                        classifiedVariable = entry.getValue();\n");
147                 writer.write("\n");
148                 writer.write("            if (
                        classifiedVariable.getClassifiedGetter().
                        getClassificationLevel().getNumericValue()\
                        n");
149                 writer.write("                    >
```

```
                    classifiedVariable.getClassifiedSetter().
                    getClassificationLevel().getNumericValue())
                     {\n");
150                 writer.write("\n");
151                 writer.write("                    throw new " +
                    ClassificationException.class.getName() + "
                    ();\n");
152                 writer.write("                }\n");
153                 writer.write("\n");
154                 writer.write("                if (
                    classifiedVariable.getClassifiedGetter().
                    getClassificationLevel().getNumericValue()\
                    n");
155                 writer.write("                        > " +
                    ClassificationLevel.class.getName() + "." +
                     classificationLevel.toString() + ".
                    getNumericValue()) {\n");
156                 writer.write("\n");
157                 writer.write("                    " + Method.class.
                    getName() + " method = classifiedVariable.
                    getClassifiedSetter().getMethod();\n");
158                 writer.write("                    " + Object.class.
                    getName() + " nullObject = null;\n");
159                 writer.write("\n");
160                 writer.write("                    try {\n");
161                 writer.write("                        method.invoke
                    (returnData, nullObject);\n");
162                 writer.write("                    } catch (" +
                    IllegalAccessException.class.getName() + "
                    ex) {\n");
163                 writer.write("                        throw new " +
                     ClassificationException.class.getName() +
                    "();\n");
164                 writer.write("                    } catch (" +
                    IllegalArgumentException.class.getName() +
                    " ex) {\n");
165                 writer.write("                        throw new " +
                     ClassificationException.class.getName() +
                    "();\n");
166                 writer.write("                    } catch (" +
                    InvocationTargetException.class.getName() +
                     " ex) {\n");
167                 writer.write("                        throw new " +
                     ClassificationException.class.getName() +
                    "();\n");
168                 writer.write("                    }\n");
169                 writer.write("                }\n");
170                 writer.write("            }\n");
171                 writer.write("\n");
172
173                 writer.write("            return returnData;\n");
174                 writer.write("        }\n");
175             }
176
177             writer.write("}\n");
```

```
178
179                 writer.flush();
180                 writer.close();
181
182             } catch (IOException ex) {
183                 Logger.getLogger(WebServiceClassificationProcessor.
                        class.getName()).log(Level.SEVERE, null, ex);
184             }
185         }
186
187         return true;
188     }
189
190     private String buildModifiersString(Set<Modifier> modifiers) {
191
192         StringBuilder modifierStringBuilder = new StringBuilder();
193
194         if (!modifiers.isEmpty()) {
195
196             for (Modifier modifier : modifiers) {
197                 modifierStringBuilder.append(modifier.toString());
198                 modifierStringBuilder.append(" ");
199             }
200
201             modifierStringBuilder.deleteCharAt(
                    modifierStringBuilder.length() - 1);
202         }
203
204         return modifierStringBuilder.toString();
205     }
206
207     private String buildMethodParametersString(List<? extends
            VariableElement> parameters) {
208
209         StringBuilder parameterStringBuilder = new StringBuilder();
210
211         if (!parameters.isEmpty()) {
212
213             for (VariableElement parameter : parameters) {
214
215                 String parameterName = parameter.getSimpleName().
                        toString();
216                 String parameterType = parameter.asType().toString
                        ();
217
218                 parameterStringBuilder.append(parameterType);
219                 parameterStringBuilder.append(" ");
220                 parameterStringBuilder.append(parameterName);
221                 parameterStringBuilder.append(", ");
222             }
223         }
224
225         parameterStringBuilder.append(AccessCredentials.class.
                getName());
226         parameterStringBuilder.append(" ");
```

```
227         parameterStringBuilder.append("accessCredentials");
228
229         return parameterStringBuilder.toString();
230     }
231
232     private String buildCallParametersString(List<? extends
            VariableElement> parameters) {
233
234         StringBuilder parameterStringBuilder = new StringBuilder();
235
236         if (!parameters.isEmpty()) {
237
238             for (VariableElement parameter : parameters) {
239
240                 String parameterName = parameter.getSimpleName().
                        toString();
241
242                 parameterStringBuilder.append(parameterName);
243                 parameterStringBuilder.append(", ");
244             }
245
246             parameterStringBuilder.delete(parameterStringBuilder.
                    length() - 2, parameterStringBuilder.length());
247         }
248
249         return parameterStringBuilder.toString();
250     }
251 }
```

Listing C.1: Annotation processor

Bibliography

[1] SpringSource a division of VMware. Enterprise Java development tools. http://www.springsource.com/developer/spring.

[2] Aslan Askarov and Andrei Sabelfeld. Secure implementation of cryptographic protocols: A case study of mutual distrust. In *In ESORICS*. Springer-Verlag, 2005.

[3] David E Bell and Leonard LaPadula. Secure computer system: Unified exposition and multics interpretation. *Technical Report*, 44(5):134, 1976.

[4] Paul V. Biron and Ashok Malhotra. XML schema part 2: Datatypes, W3C recommendation. http://www.w3.org/TR/xmlschema-2/, October 2004.

[5] David Booth, Hugo Haas, Francis McCabe, Eric Newcomer, Michael Champion, Chris Ferris, and David Orchard. Web services architecture, W3C working group note 11. http://www.w3.org/TR/ws-arch/, February 2004.

[6] Roberto Chinnici, Marc Hadley, and Rajiv Mordani. *The Java API for XML-Based Web Services (JAX-WS) 2.0*. Sun Microsystems Inc., 4150 Network Circle Santa Clara, CA 95054 USA, final release edition, April 2006.

[7] Stephen Chong, Jed Liu, Andrew C. Myers, Xin Qi, K. Vikram, Lantian Zheng, and Xin Zheng. Secure web applications via automatic partitioning. *SIGOPS Oper. Syst. Rev.*, 41:31–44, October 2007.

[8] Stephen Chong, Andrew C. Myers, K. Vikram, and Lantian Zheng. Jif reference manual. http://www.cs.cornell.edu/jif/doc/jif-3.3.0/manual.html, February 2009.

[9] Stephen Chong, K. Vikram, and Andrew C. Myers. Sif: enforcing confidentiality and integrity in web applications. In *Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium*, pages 1:1–1:16, Berkeley, CA, USA, 2007. USENIX Association.

[10] Luc Clement, Andrew Hately, Claus von Riegen, Tony Rogers, Tom Bellwood, Steve Capell, Luc Clement, John Colgrave, Matthew J. Dovey, Daniel Feygin, Andrew Hately, Rob Kochman, Paul Macias, Mirek Novotny, Massimo Paolucci, Claus von Riegen, Tony Rogers, Katia Sycara, Pete Wenzeland, and Zhe Wu. *UDDI Version 3.0.2, UDDI Spec Technical Committee Draft*. UDDI Spec TC, OASIS, 10 2004.

[11] Dorothy E. Denning and Peter J. Denning. Certification of programs for secure information flow. *Commun. ACM*, 20:504–513, July 1977.

[12] Dawson Engler. *Static analysis versus model checking for bug finding*, pages 1–1. Springer-Verlag, London, UK, 2005.

[13] eZ Systems AS. OpenESB community. http://openesb-community.org/.

[14] David C. Fallside and Priscilla Walmsley. XML schema part 0: Primer, W3C recommendation. http://www.w3.org/TR/xmlschema-0/, October 2004.

[15] Robert Bruce Findler and Matthias Felleisen. Contracts for higher-order functions. *SIGPLAN Not.*, 37:48–59, September 2002.

[16] The Apache Software Foundation. Apache ode (orchestration director engine). http://ode. apache.org/.

[17] The Apache Software Foundation. Apache struts. http://struts. apache.org/.

[18] The Eclipse Foundation. BPEL designer project. http://www.eclipse. org/bpel/.

[19] The Eclipse Foundation. BPEL to Java (B2J) subproject. http://www.eclipse.org/stp/b2j/.

[20] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification, Third Edition*. Addison-Wesley Longman, Amsterdam, 3 edition, June 2005.

[21] Hugo Haas and Allen Brown. Web services glossary, W3C working group note 11. http://www.w3.org/TR/ws-gloss/, February 2004.

[22] David Heinemeier Hansson and Rails core team. Ruby on rails. http://rubyonrails.org/.

[23] Facebook Inc. Facebook developers documentation. http://developers.facebook.com/docs/.

[24] Google Inc. Google code apis & tools. http://code.google.com/more/.

[25] IBM iSeries Information Center. *Eserver iSeries, Web services, WebSphere Application Server - Express Version 5.1*. IBM, 2 edition, August 2005.

[26] Diane Jordan, John Evdemon, Alexandre Alves, Assaf Arkin, Sid Askary, Charlton Barreto, Ben Bloch, Francisco Curbera, Mark Ford, Yaron Goland, Alejandro Guízar, Neelakantan Kartha, Canyang Kevin Liu, Rania Khalaf, Dieter König, Mike Marin, Vinkesh Mehta, Satish Thatte, Danny van der Rijn, Prasad Yendluri, and Alex Yiu. *Web Services Business Process Execution Language Version 2.0, OASIS Standard*. OASIS Web Services Business Process Execution Language (WSBPEL) Technical Committee, April 2007.

[27] D. Kaye. *Loosely Coupled: The Missing Pieces of Web Services*. RDS Press, 2003.

[28] Henry George Liddell and Robert Scott. *A Greek-English Lexicon*. Clarendon Press, Oxford, 1940.

[29] Andrew C. Myers. JFlow: practical mostly-static information flow control. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '99, pages 228–241, New York, NY, USA, 1999. ACM.

[30] Andrew C. Myers. *Mostly-Static Decentralized Information Flow Control*. PhD thesis, Massachusetts Institute of Technology, January 1999.

[31] Andrew C. Myers and Barbara Liskov. Complete, safe information flow with decentralized labels. In *19th IEEE Symposium on Research in Security and Privacy (RSP)*, Oakland, California, May 1998.

[32] Andrew C. Myers and Barbara Liskov. Protecting privacy using the decentralized label model. *ACM Trans. Softw. Eng. Methodol.*, 9:410–442, October 2000.

[33] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[34] Geoffrey Smith and Dennis Volpano. Secure information flow in a multi-threaded imperative language. In *IN PROC. ACM SYMP. ON PRINCIPLES OF PROGRAMMING LANGUAGES*, pages 355–364, 1998.

[35] Henry S. Thompson, David Beech, Murray Maloney, and Noah Mendelsohn. XML schema part 1: Structures, W3C recommendation. http://www.w3.org/TR/xmlschema-1/, October 2004.

[36] Jinesh Varia. Overview of amazon web services, December 2010.

[37] Jim Waldo, Geoff Wyant, Ann Wollrath, and Sam Kendall. A note on distributed computing. Technical report, Sun Microsystems Laboratories, 2550 Garcia Avenue Mountain View, CA 94043, 1994.

[38] Petals Link (EBM Websourcing). EasiestDemo - open source BPEL to Java generator. http://research.petalslink.org/display/easiestdemo/.

[39] Stephen A. White. *Using BPMN to Model a BPEL Process*. IBM Corp., United States of America, April 2005.

[40] Ann Wollrath, Roger Riggs, and Jim Waldo. Distributed object model for the Java system. In *Proceedings of the USENIX 1996 Conference on Object-Oriented Technologies*, Toronto, Ontario, Canada, June 1996. Sun Microsystems, Inc.