박 사 학 위 논 문

Doctoral Thesis

손실이 있는 인터페이스 어댑터 체인의 수학적 분석

Formal Analysis Framework
for Lossy Interface Adapter Chaining

정 유철 (鄭 有喆 Chung, Yoo Chul)
전산학과
Department of Computer Science

KAIST

2010

손실이 있는 인터페이스 어댑터 체인의 수학적 분석

Formal Analysis Framework for Lossy Interface Adapter Chaining

Formal Analysis Framework for Lossy Interface Adapter Chaining

Advisor : Professor Dongman Lee

by

Chung, Yoo Chul
Department of Computer Science
KAIST

A thesis submitted to the faculty of the KAIST in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science

Daejeon, Korea
2009. 11. 04.
Approved by

Professor Dongman Lee
Advisor

손실이 있는 인터페이스 어댑터 체인의 수학적 분석

정 유철

위 논문은 한국과학기술원 박사학위논문으로 학위논문심사 위원회에서 심사 통과하였음.

2009년 11월 24일

심사위원장 이동만 (인)

심사위원 이영희 (인)

심사위원 현순주 (인)

심사위원 고인영 (인)

심사위원 김명준 (인)

DCS 20055182 정 유철. Chung, Yoo Chul. Formal Analysis Framework for Lossy Interface Adapter Chaining. 손실이 있는 인터페이스 어댑터 체인의 수학적 분석. Department of Computer Science. 2010. 112p. Advisor Prof. Dongman Lee. Text in English.

Abstract

In an ideal ubiquitous computing environment, computing services would be embedded in physical objects all around us, which would all work together seamlessly. As computing could be embedded in all sorts of physical objects, a myriad of different interfaces could arise for different computing services with similar functionality. However, software in ubiquitous computing environments must still be able to access computing services in the environment even when the desired service has an unfamiliar interface, otherwise this will be an obstacle to the seamless operation of a ubiquitous computing environment.

Interface adapters can provide a solution to this problem by transforming interfaces as necessary. And chaining them together enables much more flexibility without incurring a prohibitive development cost in creating all of the required interface adapters for direct interface adaptation. Unfortunately, an interface adapter is likely to be imperfect, so interface adaptation would often incur adaptation loss. This is even more of an issue when interface adapters are chained. To properly consider loss in the construction of interface adapter chains, a mathematical framework is required to analyze such chains.

We develop a number of mathematical frameworks to analyze the loss in interface adapter chaining, each building upon another. For each mathematical framework, we define algebraic objects and operations that express the loss and how it is affected by each interface adapter. These in turn are used to develop algorithms and prove complexity results for problems relevant to lossy interface adapter chaining.

Contents

List of Tables

List of Figures

List of Algorithms

1. Introduction

In an ideal ubiquitous computing environment, computing services would be embedded in physical objects and the physical environment all around us, which would all work together seamlessly to provide better service from the physical world and intuitive access to the online world [21, 35, 40, 60, 67, 74]. Among other things, this implies that all sorts of physical objects should embed computing capability for using the object itself and to let other computing services use the object whenever this is deemed best for the user.

For seamless operation, a wide variety of computing services embedded in a diverse set of physical objects must be able to work with each other. However, unless the day comes when a single regulatory body can mandate the static standardization of all computing services, a realistic environment will always include computing services from diverse manufacturers based on multiple standards, internal and external [13, 25, 65, 72]. This in turn results in a myriad of different interfaces being created for services with similar functionality. Thus computing services in ubiquitous computing environments must be able to work with other computing services with different interfaces than may have been known when the services are developed.

The obvious way to solve this sort of problem is to rigorously standardize service types and their interfaces. However, even creating a draft standard of a well-established service such as printing can take over 15 months [59]: standardization of new kinds of services with less widespread deployment can take even longer, or even not at all, and it can be expected that many such services will arise in ubiquitous computing environments. Until such standardization occurs, a service must still be able to interoperate seamlessly with other services. Even *with* standardization, standards can change over time, and unless

all services in every physical object is updated accordingly, services would still end up being loosely coupled.

In order to access a different interface than a client was written for without rewriting the client, interface adapters could be used to convert invocations in one interface to another [4, 16, 22, 52, 54, 56, 58, 77]. However, it is unlikely that interface adaptation can be done perfectly, since interfaces are usually developed independently of each other with no regard for compatibility. Adaptation loss will usually result as certain methods cannot be adapted by the interface adapter, and the problem is only worse when adapters are chained [26, 27, 31, 34, 55, 71]. Even analyzing how much loss results from an interface adapter chain is not a trivial problem that can be modeled as a shortest path problem.

In ubiquitous computing environments, it is especially important that the loss during interface adaptation be considered since it cannot be waived on demand. Much of the existing work on interface adaptation focuses on the development of entire software systems which integrate existing software components [5, 16, 37, 54, 56, 58], and there is an expectation that perfect adaptation can be achieved at least in principle. This is not an expectation that can carry over to ubiquitous computing environments. For one thing, perfect adaptation may be fundamentally impossible for a service that represents hardware, e.g. there would be no way for a speaker to play back pure video. Another issue is that interface adapters be cannot created or modified on demand since a developer must manually intervene, which is not an issue for the development of static software systems.

In order to analyze the adaptation loss incurred by interface adapter chains and to develop algorithms which can construct them, a theoretical framework is required which can express and analyze lossy interface adapter chaining. However, no previous work has studied how to rigorously analyze the loss in interface adapter chains, so either lossy interface adapters were impossible to

use, or it was impossible to properly evaluate how much loss may be incurred when chaining lossy interface adapters.

We develop an algebraic framework that analyzes loss in interface adapter chaining, where the concepts and algebraic operations for expressing and analyzing lossy dependencies are defined. This framework should provide the basic mathematical building blocks necessary when creating algorithms for lossy interface adaptation. In fact, several such frameworks will be created, simpler ones being the bases for more complex ones but each still being potentially useful for particular application domains. We will also develop several algorithms and prove the computational complexity for several problems related to lossy interface adapter chaining.

1.1 Terminology

A *service* is a concrete entity which performs tasks for external entities. Similar concepts include components and distributed objects.

An *interface* specifies the set of methods that a service provides. Software that knows how to use a specific interface should be able to use any service that conforms to this interface. Similar concepts include classes and protocols.

A *method* is the basic unit of communication by which external entities can request a service to perform a specific task. Similar terms and concepts include operation, function, procedure, and message.

An *interface adapter* converts a source interface to a target interface. It allows the use of a service that conforms to the source interface by software written to use the target interface.

When we mention *loss* in relation to interface adaptation, this means that one or more methods specified in an interface cannot be used. There should be no loss when a directly accessing a service, but loss may be inevitable when it must be accessed using a different interface through interface adaptation.

1.2 Motivation

In this dissertation, we take the approach that services which provide similar functionality can be accessed through different interfaces than that provided by the service itself through the use of pre-existing interface adapters. Each interface is accessed through methods, and an interface adapter can provide an alternative interface by implementing external methods using the methods available in the original interface.

There can be various approaches to creating the interface adapters themselves, from manual development of an adapter to semi-automatic generation through semantic or code analysis. While manual development of interface adapters is probably the most reliable method, the mathematical frameworks described in this dissertation does not preclude the use of alternative methods [4, 32, 37, 52, 54, 77], and the generation of interface adapters themselves is outside the scope of this dissertation, as our mathematical framework assumes a fixed set of interfaces and pre-existing interface adapters.

As a concrete example, we will describe how the web service XWebCheckOut could be accessed using the Google Checkout API, an example we base on one from [52]. In figure 1.1, we can see how XWebCheckOut has a different interface from that of Google Checkout. For a network client that only knows how to use the Google Checkout API, it would need an adapter which can convert the source interface for XWebCheckOut to the target interface for Google Checkout.

A developer could implement methods for the Google Checkout interface by using methods available in the XWebCheckOut interface. For example, the Place-Order method in the Google Checkout interface could be implemented using the AddOrder and UpdateOrder methods in the XWebCheckOut interface. Doing this for each method in the Google Checkout interface will result in an interface adapter that adapts the XWebCheckOut interface to the Google Checkout interface.

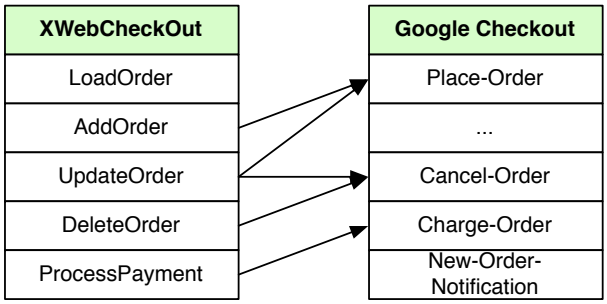

Figure 1.1: Example of service interface adaptation.

However, interface adaptation might not be perfect as some methods in the target interface simply cannot be implemented using only methods in the source interface, resulting in lossy interface adaptation. We can see this in figure 1.1, where there is no feasible way to implement the NEW-ORDER-NOTIFICATION method using only methods available from the XWebCheckOut interface, assuming that the method cannot be implemented independently of XWebCheckOut.

Since a network client is using the target interface to access a service, the loss in the target interface is of more interest than the inability to provide access to the full range of functionality provided in the source interface. For example, with a network client that only knows how to use the Google Checkout API, it is more relevant that an interface adapter may not be able to provide the NEW-ORDER-NOTIFICATION method in the Google Checkout interface, rather than that the functionality provided by the LOADORDER method in the XWebCheckOut interface is missing.

If we require that all available interfaces for similar services must be adapted between each other with only a single adapter in between, then the number of interface adapters required is in the order of n^2. Developing all the required adapters can be impractical, so interface adapters can be *chained* to adapt a source interface to one interface, this interface adapted to another interface, and so on until we get a target interface that a network client knows how to

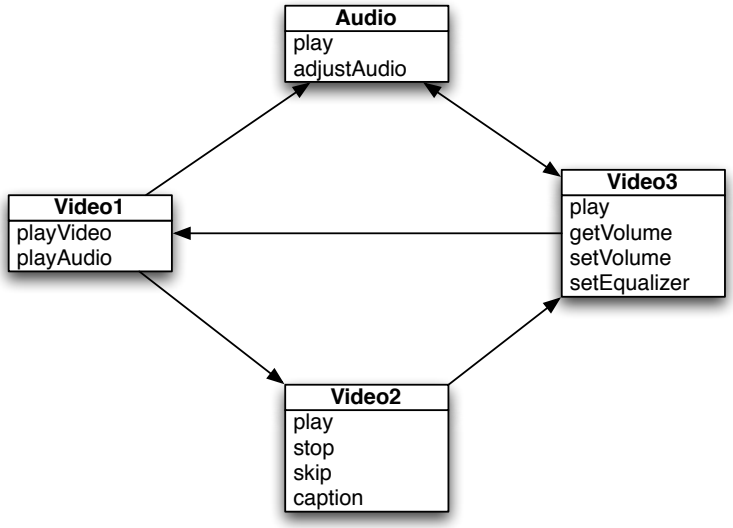

Figure 1.2: Multiple interfaces related by interface adapters.

use [26, 34, 55, 71]. In the best case, we can even get away with only n adapters given n interfaces.

However, different chains of interface adapters result in different loss from the interface adaptation, so we need a way to analyze the chaining of lossy interface adapters. We will look at another example in figure 1.2, where there are four interfaces and six interface adapters, each of the latter represented by an arrow from the source interface to the target interface it converts from and to.

Each interface may have the following characteristics:

- *Video1* can play both video and audio files.
- *Video2* can only play video files, but can stop playback, skip over a fixed amount of time, and select captions.
- *Video3* can only play video files, but can get and set the volume and set the equalizer for the audio output.

Adapter	Target	Source	Implements	Using
Video1toVideo2	*Video2*	*Video1*	*play*	*playVideo*
Video2toVideo3	*Video3*	*Video2*	*play*	*play*
Video1toAudio	*Audio*	*Video1*	*play*	*playAudio*
AudiotoVideo3	*Video3*	*Audio*	*setVolume*	*adjustAudio*
			setEqualizer	*adjustAudio*
Video3toAudio	*Audio*	*Video3*	*adjustAudio*	*setVolume* *setEqualizer*
Video3toVideo1	*Video1*	*Video3*	*playVideo*	*play*

Table 1.1: Interface adapters for figure 1.2.

- *Audio* can only play video files, but can set audio properties, which are the volume and the equalizer.

Each interface adapter may be implemented as in table 1.1, which specifies the methods in the source interface used to implement methods in the target interface.[1] Methods in a target interface not mentioned are not adapted to. Figure 1.3 graphically shows the methods in a source interface that the interface adapters use to implement methods in a target interface.

A service with interface *Video1* may be available, and we may want to access it using a client that only understands interface *Video3*. There is no interface adapter which directly converts interface *Video1* to *Video3*, but there are interface adapter chains which can indirectly do the conversion. Chaining *Video1toVideo2* with *Video2toVideo3* or chaining *Video1toAudio* with *AudiotoVideo3* can convert interface *Video1* to *Video3*.

Given multiple possible interface adapter chains, we would want to use the

[1]While it may seem odd to have an adapter *AudiotoVideo3* from *Audio* to *Video3* which cannot adapt a playback method, it can still be useful when someone wishes to reduce loud noises from an audio device with interface *Audio* using a remote control that only understands the interface *Video3*.

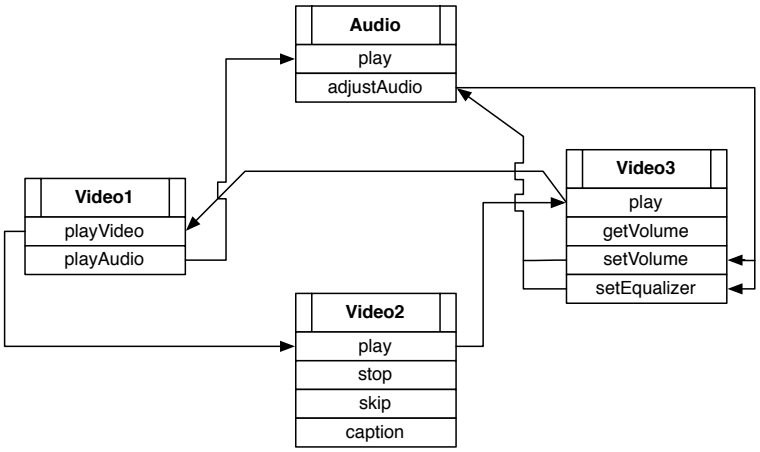

Figure 1.3: Method dependencies for the interface adapters in figure 1.2.

best interface adapter chain that can provide the most methods in the target interface. Associating a cost with an adapter depending on how well it adapts the methods in its target interface and using minimum-cost path algorithms such as Dijkstra's algorithm [14] would be an obvious approach to choose the best interface adapter chain. However, this naive approach would not work as we will see from the example in figure 1.2.

Video1toAudio and *AudiotoVideo3* can adapt one out of two methods in *Audio* and *Video3*, respectively. In contrast, *Video1toVideo2* and *Video2to-Video3* can adapt one out of four methods in *Video2* and *Video3*. One might think that the *Video1toAudio* and *AudiotoVideo3* chain would be better than the *Video1toVideo2* and *Video2toVideo3* chain simply by looking at how lossy each interface adapter is, but one would be wrong.

AudiotoVideo3 requires the *adjustAudio* method in *Audio* to implement the *setVolume* and *setEqualizer* methods in *Video3*. However, *Video1toAudio* cannot implement the *adjustAudio* method in *Audio*, so the *Video1toAudio* and *AudiotoVideo3* chain ends up with no available methods for *Video3*. In contrast, the *Video1toVideo2* and *Video2toVideo3* chain can provide the method *play* for *Video3*. A single number for each interface adapter cannot express

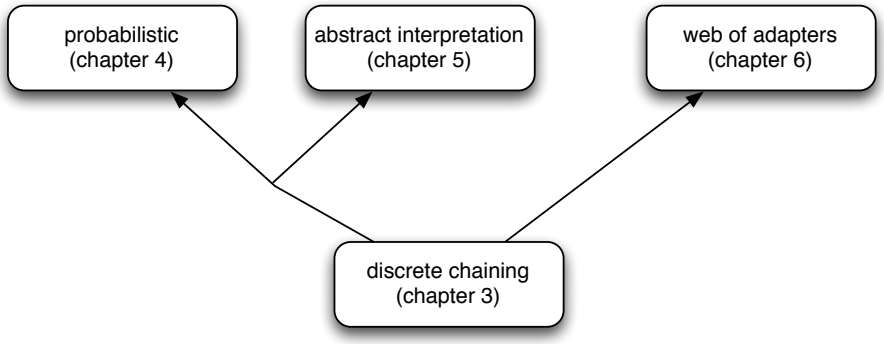

Figure 1.4: Roadmap of frameworks for analyzing lossy interface chaining.

such dependencies properly, so we need a more precise approach to analyze the loss in interface adapter chains.

In the rest of this dissertation, we will discuss how to mathematically analyze the loss incurred from the chaining of interface adapters. We will also assume that interface adapters are implemented as transparently as possible: while an interface adapter may not be able to provide all of the methods in the target interface, the methods it will provide will work just as if they were invoked directly on a service having the target interface.

1.3 Contributions

Analyzing the loss during interface adapter chaining is not as simple as reducing the interfaces and adapters to a weighted graph and then finding the shortest path [14, 18, 29], which is why a rigorous mathematical framework is required. Multiple mathematical frameworks are developed, each building upon another. This is not only so that a complex mathematical framework can be built upon a simpler and easier to understand framework, but it is also the case that each framework can be practical by themselves depending on the system architecture and available resources. This section briefly explains each

framework, which comprise the contributions of this dissertation, and how they will be built upon each other as in figure 1.4.

There are some common results that are expected from each framework. One is a set of rigorous mathematical definitions and operations that form the framework itself. These will be used to express loss incurred by interface adapters. The mathematical operations allow these to be analyzed in an algebraic manner, which can be used to compute how well interface adaptation can be done. Another result that is expected from each framework are either polynomial-time algorithms for solving relevant problems or proofs of the computational complexity of these problems. Such problems include finding the optimal combination of interface adapters or how to use them during actual operation, e.g. which methods with which interface adapters should be invoked during interface adaptation.

The basic framework described in chapter 3, upon which the others are built, analyzes boolean dependencies for single interface adapter chains, i.e. each individual method dependency is assumed to be completely fulfilled or completely missing. This is applicable when interface adapters are singly linked together during interface adaptation with no partial dependencies to worry about. Based on this basic framework, there are two paths for building more complex frameworks.

One path is to refine the analysis of individual method dependencies by allowing that they need not be complete. This allows cases where not all arguments given to a method can be handled due to interface adaptation. This inspires two other approaches, a probabilistic approach [20, 28, 46, 76] and another based on abstract interpretation [3, 9, 10, 66]. The probabilistic approach described in chapter 4 would be appropriate when the individual partial dependencies are fuzzy, whereas the approach based on abstract interpretation as described in chapter 5 would be appropriate when the partial dependency can be decomposed into sharp abstract domains, i.e. arguments can be divided

into unambiguous abstract domains.

The other path is to handle a combination of method dependencies which form a directed acyclic graph as described in chapter 6. This is a case where interface adapters are not just combined into a single chain, but rather in which they can be combined as a directed acyclic graph. This allows an interface adaptation system to adapt an interface with the least loss possible without having to choose between interface adapter chains with different losses.

1.4 Organization

The remainder of this dissertation is organized as follows. In chapter 2, we review related work on interface adaptation. Subsequent chapters contain the main work of this dissertation, where chapter 3 describes the most basic framework based on a discrete approach, chapter 4 describes a probabilistic framework, chapter 5 describes a framework based on abstract interpretation, and chapter 6 describes a DAG-based framework. In chapter 7, we discuss the pros and cons of each of these theoretical frameworks and other issues relevant to applying them to a real interface adaptation system. Chapter 8 concludes this dissertation.

2. Related work

The idea of using interface adapters is a well-known idea that has even been described as a notable design pattern [22]. While some work attempt to avoid the need for interface adaptation altogether by using language constructs that enable a certain amount of leeway in interface changes [5, 41], there is a limit to how much change such approaches can tolerate. Using interface adapters, on the other hand, does not have this limitation. We survey work related to interface adaptation in this chapter.

2.1 Generating adapters

This section surveys related work on generating interface adapters. It should be noted that all of them require significant human effort, even if some provide tools to identify certain problems, or specification languages that are much easier to use compared to coding an interface adapter directly in a general-purpose programming language.

2.1.1 Putilo and Atlee

Putilo and Atlee [56] describes a language called Nimble that allows developers to declare how the parameters to a procedure call can be transformed at run-time, which allows a procedure to evolve without having to modify callee code. It uses a simple pattern-based language to specify how the actual parameters that a procedure receives should be transformed. While it is focused on rearranging or restructuring the parameters, it also has a limited ability to specify algebraic operations such as addition or multiplication on the parameters. It

has no support for renaming procedures, however.

A mapping specification written in Nimble is translated into another language which specifies the exact steps in transforming the parameters, which in turn is used by a compiler specific to the programming language and execution environment to create actual executable code for the interface adapter. The linker is modified so that the adapter is called instead of the procedure, and the adapter will invoke the procedure instead after it transforms the parameters.

2.1.2 Reussner

Reussner [58] uses parameterized contracts, which is a generalization of classical contracts [48], to adjust the protocol of a component to match what is expected by its environment. Having pre-conditions and post-conditions set to requires-interfaces and provides-interfaces, it parameterizes the pre-condition with the post-condition of the component and vice versa. The contracts are analyzed using finite state machines to check compatibility and construct new contracts to adapt to a new execution environment.

They created the CoConut/J tool suite to implement their adaptation system. It can use specifications written as message sequence charts to specify contracts when code for a component is not yet available, or it can use annotations in the code itself to specify contracts. Their system does not handle signature mismatches where message names or parameters might be different from what is expected.

2.1.3 Benatallah et al.

Benatallah et al. [4] categorizes common mismatch patterns for web service interfaces. Based on these common mismatch patterns, they suggest various code templates that can be instantiated based on developer-provided parameters that can form part of an interface adapter. Categorizing mismatch pat-

terns makes it easier to identify incompatibilities between interfaces, which the provided templates can then be used to significantly reduce the effort required to develop an interface adapter. A developer must identify the mismatches and fill in the missing details not covered by the templates, however.

Common mismatch patterns are divided between those on the operational level and on the protocol level. Mismatches on the operational level are differences in operation names, parameter types, or return types, which basically form the signature of an operation. Mismatches on the protocol level are differences in expected behavior such as message ordering mismatch, extra message mismatch, missing message mismatch, etc.

2.1.4 Dumas et al.

Dumas et al. [16] proposes the use of visual elements to construct interface adapters. The visual elements connect two interfaces that are specified as a flowchart, and they specify adaptive behavior such as message transforms or message aggregation, although there is no support for message reordering. Each visual element is mapped to an operator in an interface transformation algebra so that their behaviors are formally defined.

They provide an interface mapping tool to graphically edit how interfaces should be transformed into other interfaces. The output of the interface mapping tool is used by a service mediation engine which maps the output to a finite state machine and uses this to adapt an interface to another.

2.1.5 Motahari Nezhad et al.

Motahari Nezhad et al. [52] categorizes common mismatch patterns for web service interfaces and provides tools for identifying them. Interface-level mismatches are identified by attempting to match messages by name and type and then identifying mismatches in the signatures. Message matching is done by

exactly matching a name, but it can also be done by matching the structures of the interfaces that are specified with XML schemas [57]. A developer needs to match messages that have not been matched, unmatch those that have been mistakenly identified, and fix mismatch details incorrectly inferred.

Once interface-level matches are identified, they identify protocol-level mismatches by roughly simulating the adapter. A mismatch tree is constructed where children mark the next state of a protocol interaction, with alternate children representing alternate interactions. The mismatch tree is used to identify possible deadlocks and ways to avoid them, supported by interface-based inference, log-based inference, and developer input.

They implemented their approach inside IBM WID, which is an Eclipse-based IDE for development of composite applications based on SCA [64]. They included a mismatch tree editor and interface mapping editor to allow developers to inspect and modify the matchings and mismatches identified.

2.1.6 Kongdenfha et al.

Kongdenfha et al. [37] uses aspect-oriented programming [33] to extend the work of Benatallah et al. [4]. They also use mismatch patterns to identify mismatches, and templates are provided to reduce the development effort for constructing adapters. But instead of only generating standalone adapters, they also support aspect-oriented adapters that extend the service itself instead of running separately. They extend a BPEL engine with an aspect manager, which can insert adaptation aspects into running web services as needed.

2.2 Formalizing interface adaptation

This section surveys related work on formalizing interface adaptation. None of the work mentioned in this section allow for loss during interface adaptation. Without loss to consider, signature mismatches are trivial for humans to solve

but too hard for computers to even identify, so much of the focus on formalizing interface adaptation has been on the behavior of interfaces.

2.2.1 Yellin and Strom

Yellin and Strom [77] describe a formal approach to specifying a protocol for an interface, which explicitly specify the message order as required by the interface. The protocol specification defines a finite-state machine, where messages form the edges. They use this as the basis to analyze whether two interacting components have behavioral problems such as messages arriving out of sequence or a deadlock, i.e. checking whether two interacting components have protocol compatibility.

They also provide a way to bind two incompatible software components together using adapters. An adapter is modeled as a finite-state machine which specifies how messages are transformed. The correctness of the adapter can then be verified by analyzing the adapter specification along with the protocol specifications of the two interacting components. An adapter specification can be difficult to write, however, so interface mappings can be written to specify how two interfaces map to each other. They provide an algorithm to either convert an interface mapping into an adapter specification that satisfies protocol compatibility with the interacting components, or to conclude that no such adapter can exist.

2.2.2 Spitznagel and Garlan

Spitznagel and Garlan [68] describe a formal approach to specifying the protocol of an adapter using FSP [45]. The protocol of a base connection between software components is specified at a high level in FSP, and the behavior of the whole system when an adapter is applied to the base connection can be derived by combining the specifications. The use of FSP to specify protocols

allows tools to analyze whether a component behaves as expected when an adapter is applied, and they show examples of this with adapters that add retry or failover behavior. Their approach allows adapter specifications for orthogonal functionality to be composed easily, which makes it easier to analyze the behavior of a component when many adapters are applied.

2.2.3 Bracciali et al.

Bracciali et al. [6] uses a subset of π-calculus [49] to formally express and reason about component adaptation. Component interfaces are expressed using interaction patterns, which describe the input and output actions for a protocol at a high level. It also has a limited ability express the data that is exchanged at an abstract level. Adapters are specified by mapping action signatures and sequences. This specification can be used to derive concrete adapters in a way to avoid deadlocks and satisfy all action correspondences as expected by the mapping. The use of π-calculus enables the automatic verification of many properties for interacting systems, which includes the compatibility of component protocols.

2.2.4 Poizat et al.

Poizat et al. [54] specify interfaces using signatures and labelled transition systems [45], through which behavior is modeled by a process algebra. Mappings are expressed with regular expressions of synchronous vectors, which express not only synchronization between processes on the same event names, but more general correspondences between the events of the process involved. Using interface specifications and mappings, they have an algorithm which can generate the concrete steps required by an adapter to ensure correct interaction, with or without message ordering.

Using their adapter generation algorithm, they suggest a methodology for

incrementally building a software system from components, where an adapter is generated each time a software component is added to ensure correct interaction between components. Their claim is that an incremental approach can avoid the problems of a global approach, where every change in any component might force global analysis of the entire system.

2.3 Combining adapters

This section surveys related work where multiple interface adapters are combined together to achieve interface adaptation. While a few mention the possibility of lossy interface adapters, almost all of the work mentioned in this section are focused on lossless interface adapter chaining, and only Kim et al. [34] actually attempts to analyze loss.

2.3.1 Keller and Hölzle

Keller and Hölzle [32] use binary component adaptation to adapt the interface to Java classes. Their work requires that all Java classes and delta files are distributed as a single package, where adaptation allows developers to maintain binary compatibility even when the interface evolves, avoiding the need to modify source code.

The changes needed to adapt an interface are described in an adaptation specification written in a Java-like language. The adaptation specification is then compiled into a delta file, which contains bytecode that implements the changes and specifies where in a class file the bytecode should be inserted. The delta file is not executed directly when adapting an interface, but instead specifies how Java classes should be modified. This is analogous to having an interface adapter integrated directly with a service rather than running it separately. The rewriting of Java classes is achieved at load-time, and the necessary support is integrated with the Java virtual machine [43].

Delta files can be chained together so that independent functionality or non-overlapping methods and fields can be modified in multiple delta files developed separately. While there is no discussion of chaining version-to-version delta files to provide compatibility between a large gap in versions, their work can support such approaches. However, the developer must manually select the delta files and determine their position in the chain.

2.3.2 Hallberg

Hallberg [27] suggests a methodology to maintain permanent backwards compatibility for modules in Haskell [53], although the approach can be applied to other programming languages as well. Modules would no longer be identified only by name, instead being identified by name *and* version, and Haskell code would always import specific versions of modules. The version of a module will be incremented whenever there is a change in the interface, but modules that are compatible with previous versions of the interface will also be maintained in perpetuity. This methodology would allow code to always be compatible with a module even when the module continues to evolve.

To ensure that a module can always be used even with previous versions of the interface, it is suggested that an adapter be created that allows a specific version of a module to be used with the interface of the previous version. Chaining these adapters together will allow the latest version of a module to be used through any of the older versions of the interface.

2.3.3 Kaminski et al.

Kaminski et al. [31] suggests using a chain of adapters for web services to maintain permanent backward compatibility even as the interface of a web service evolves. With every change in the interface, an adapter is constructed so that the new version of the web service can be used with the old version

of the interface. Further changes are handled not by updating the adapters, but by chaining adapters for consecutive versions of the web service. Clients do not access a web service directly; instead, clients indirectly access the web service through an adapter that provides an interface for a specific version. This allows clients to continue access a web service through an older version of the interface, negating the need to update code within clients.

2.3.4 Vayssiére

Vayssiére [71] supports the interface adaptation of proxy objects for Jini [1]. The goal is to enable clients to use services even when they have different interfaces than expected. It provides an adapter service which hooks into the lookup service, so that a client can use a proxy object without having to be aware that any adaptation occurs.

Adapters are registered with the adapter service, which in turn registers all acyclic chains of interface adapters that can adapt to a given type to the lookup service. Registering all possible chains can result in an exponential number of registrations in the lookup service (they claim that the maximum number of registrations would be limited, but they ignore that there can be an exponential number of paths in a reasonable graph). There is no discussion of which chain of adapters should be used to adapt an interface, simply specifying that all chains matching the expected input and output types should be returned.

2.3.5 Gschwind

Gschwind [26] allows components to be accessed through a foreign interface and implements an interface adaptation system for Enterprise JavaBeans [47]. It implements a centralized adapter repository that stores adapters, along with weights that mark the priority of an adapter. Clients query an adaptation component to obtain an interface adapter chain, which is used to convert the

interface of an object into another. The adaptation component, which runs on the client side, retrieves the required adapters from the adapter repository. An adapter is publicly distributed as a single archive that includes the adapter code and description.

The adaptation repository uses Dijkstra's algorithm [14] to construct the shortest interface adapter chain that adapts a source interface into a target interface. While there is support for marking an adapter as lossy or not, it does not have the capability to properly analyze and compare the loss of interface adapter chains.

2.3.6 Ponnekanti and Fox

Ponnekanti and Fox [55] suggests using interface adapter chaining for network services to handle the different interfaces available for similar types of services. They provide a way to query all services whose interfaces can be adapted to a known interface. They also support lossy adapters, but the support is limited to detecting whether a particular method and specific parameters can be handled at runtime. They do not provide a method to analyze the loss of an interface adapter chain, so they are unable to choose a chain with less loss when alternatives are available. Adapter chains are constructed through a rule system based on the source and target interfaces of adapters, which is similar to constructing a path in a graph through a blind search algorithm. They also support the composition of services in addition to transforming a single interface to another, which is accomplished by constructing a tree of interface adapters.

2.3.7 Kim et al.

Kim et al. [34] describes an ad hoc scheme for analyzing the loss in interface adapter chains. The scheme is based on boolean matrixes which specify the

methods required in a source interface to implement a method in a target interface. A mapping product is defined on these matrixes which computes the loss incurred when interface adapters are chained. The mathematical model they use is not rigorously constructed, however. They also only consider the adaptation of methods as a whole and do not handle the case where methods could handle certain arguments but not others.

Using their scheme, they describe an algorithm based on uniform cost search [61] which can construct an interface adapter chain with the least adaptation loss. Despite having exponential time in the worst case, their experiments with small random graphs indicate that the algorithm might work quickly in practice. They also show that interface adapter chains constructed with the algorithm have significantly less loss than those constructed through a depth-first search, which ignores loss.

3. Discrete chains

In this chapter, we begin by constructing the most basic framework for analyzing lossy interface adapter chaining. It is assumed that a method in a target interface can be implemented as long as all of the prerequisite methods in the source interface are available. It ignores the possibility that there is no way an argument to a method in the target interface can be converted to an argument to a method in the source interface, so that an adapted method would always work properly if it works at all.

3.1 Mathematical basics

We can start formalizing the problem of lossy interface adaptation by defining an *interface adapter graph*. This is a directed graph where interfaces are nodes and adapters are edges. If there are interfaces I_1 and I_2 with an adapter A that adapts source interface I_1 to target interface I_2, then I_1 and I_2 would be nodes in the interface adapter graph while A would be a directed edge from I_1 to I_2.

We do not assume that there can be at most one adapter which can adapt one interface to another. This reflects the fact that there can be multiple adapters from different developers, similar to how there can be multiple device drivers available for a graphics card. It also simplifies some of the arguments, although they would still hold even with such a restriction with only minor changes in the proofs.

We will be using a range convention for the index notation used to express matrixes and vectors [11].

3.1.1 Method dependencies

The next step is to formally describe each adapter, i.e. each edge in the interface adapter graph, in a way that would be useful for analyzing loss. We should be able to figure out which methods in the target interface can be provided by an interface adapter given the methods available in the source interface. We do this by defining a *method dependency matrix*, a boolean matrix which describes how an interface adapter implements methods in the target interface using available methods in the source interface.

The method dependency matrix a_{ji} for an adapter A, where a_{ji} represents either the matrix itself or a single component in the matrix depending on the context, is defined by how the adapter depends on the availability of a method in the source interface in order to implement a method in the target interface. a_{ji} is true if and only if method j in the target interface can be implemented only if method i in the source interface is available. We denote the method dependency matrix associated with an adapter A as $depend(A)$.

We also define a *method availability vector* p_i for an interface, where each component p_i is true if and only if method i is available. This boolean vector is not intrinsic to an interface, unlike the method dependency matrix which *is* intrinsic to an interface adapter. Instead, it is used to represent the loss in interface adaptation such that method i in the target interface can be used only if p_i is true. For a fully functional service that implements all methods specified in its interface, the components of its method availability vector should all be true. We denote the number of true components in method availability vector p_i as $\|p_i\|$, which is equivalent to the Manhattan norm [69] when true and false components are replaced by 1 and 0, respectively.

Given method availability vector p_i for a source interface and the method dependency matrix a_{ji} for an interface adapter, we can derive the method availability vector q_j for the target interface. A method j in the target interface can only be implemented if all of the methods it depends on are available in

the source interface. So if q_j is to be true for fixed j, then all p_i must be true when a_{ji} is true:

$$q_j = \bigwedge_i (a_{ji} \rightarrow p_i) = \bigwedge_i (\neg a_{ji} \vee p_i) \tag{3.1}$$

However, equation (3.1) is incomplete in that it does not properly distinguish between methods which can always be implemented and methods which cannot be implemented given the source interface. For example, a method that returns the value of π does not need anything from the source interface, whereas there would be no way to implement a video playback method given only a source interface specialized exclusively for audio playback. For both cases, all a_{ji} are false for a specific method j, and equation (3.1) would give the wrong result for the latter case.

This can be worked around by defining a *dummy method* that is never available for every interface. We arbitrarily call this "method 1", so that p_1 will always be false for any method availability vector. It is easy to see that extending the definition of the method dependency matrix with the following rules is consistent with our definitions and equations for the method dependency matrix and method availability vector:

- a_{11} is true, while a_{1i} is set to false for all $i \neq 1$.
- If method j can always be implemented in the target interface, set a_{ji} to false for all i.
- If method j can never be implemented given the source interface, set a_{j1} to true, while a_{ji} is set to false for all $i \neq 1$.
- If method j depends on the availability of actual methods in the source interface, then a_{j1} is false.

For succinctness, we denote a method availability vector for interface I which represents that all methods are available, i.e. when the component for the dummy method is false while all the other components are true, by $\mathbf{1}'_I$.

We also define the operator $\otimes$ for a method dependency matrix as applied to a method availability vector to represent the operation in equation (3.1), or in other words:

$$a_{ji} \otimes p_i \equiv \bigwedge_i (\neg a_{ji} \vee p_i) \tag{3.2}$$

It is easy to see that a square boolean matrix where the diagonals are true and the rest of the components are false is an identity matrix for the adaptation operator $\otimes$. We denote an identity matrix with n rows as $\mathbf{I}_n$.

3.1.2 Adapter composition

To analyze the chaining of lossy interface adapters, we are also interested in how to derive a composite method dependency matrix from the composition of two method dependency matrixes, which would be equivalent to describing the chaining of two interface adapters as if they were a single interface adapter.

Given interfaces I_1, I_2, and I_3, let the corresponding method availability vectors be p_i, q_j, and r_k. In addition, let there be interface adapters A_1 and A_2, where A_1 converts I_1 to I_2 and A_2 converts I_2 to I_3, with corresponding method dependency matrixes a_{ji} and b_{kj}, respectively. We would like to know how to derive the method dependency matrix that would correspond to an interface adapter equivalent to A_1 and A_2 chained together.

From equation (3.1) and our assumptions:

$$\begin{aligned}
r_k &= \bigwedge_j (\neg b_{kj} \vee q_j) \\
&= \bigwedge_j \left(\neg b_{kj} \vee \bigwedge_i (\neg a_{ji} \vee p_i) \right) \\
&= \bigwedge_j \bigwedge_i (\neg b_{kj} \vee \neg a_{ji} \vee p_i) \\
&= \bigwedge_i \bigwedge_j (\neg b_{kj} \vee \neg a_{ji} \vee p_i)
\end{aligned}$$

$$= \bigwedge_i \left(\bigwedge_j (\neg b_{kj} \vee \neg a_{ji}) \vee p_i \right)$$
$$= \bigwedge_i \left(\neg \bigvee_j (b_{kj} \wedge a_{ji}) \vee p_i \right)$$

We reuse the operator $\otimes$ to represent the composition of two method dependency matrixes, and by comparing the above with equation (3.1), we can define it as:

$$b_{kj} \otimes a_{ji} = \bigvee_j (b_{kj} \wedge a_{ji}) \tag{3.3}$$

$\mathbf{I}_n$ from section 3.1.1 is also an identity matrix for the method dependency matrix composition operator $\otimes$.

The $\otimes$ operator is "associative"[1] when applied to method dependency matrixes and a method availability vector, i.e. $b_{kj} \otimes (a_{ji} \otimes p_i) = (b_{kj} \otimes a_{ji}) \otimes p_i$, which shows that in terms of loss, chaining adapters and then applying it to a source interface is equivalent to applying each adapter one by one to the source interface:

$$\begin{aligned}
b_{kj} \otimes (a_{ji} \otimes p_i) &= \bigwedge_j \left(\neg b_{kj} \vee \bigwedge_i (\neg a_{ji} \vee p_i) \right) \\
&= \bigwedge_j \bigwedge_i (\neg b_{kj} \vee \neg a_{ji} \vee p_i) \\
&= \bigwedge_i \bigwedge_j (\neg b_{kj} \vee \neg a_{ji} \vee p_i) \\
&= \bigwedge_i \bigwedge_j (\neg (b_{kj} \wedge a_{ji}) \vee p_i) \\
&= \bigwedge_i \left(\bigwedge_j \neg (b_{kj} \wedge a_{ji}) \vee p_i \right)
\end{aligned}$$

[1] It is not technically associative in this context as the $\otimes$ operator as applied to method dependency matrixes is not really the same as the $\otimes$ operator as applied to a method dependency matrix and a method availability vector, similarly to how $\times$ for numbers is different from $\times$ for sets.

$$
\begin{aligned}
&= \bigwedge_i \left(\neg \bigvee_j (b_{kj} \wedge a_{ji}) \vee p_i \right) \\
&= (b_{kj} \otimes a_{ji}) \otimes p_i
\end{aligned}
$$

Likewise, method dependency matrix composition is associative:

$$
\begin{aligned}
c_{lk} \otimes (b_{kj} \otimes a_{ji}) &= \bigvee_k \left(c_{lk} \wedge \bigvee_j (b_{kj} \wedge a_{ji}) \right) \\
&= \bigvee_k \bigvee_j (c_{lk} \wedge b_{kj} \wedge a_{ji}) \\
&= \bigvee_j \bigvee_k (c_{lk} \wedge b_{kj} \wedge a_{ji}) \\
&= \bigvee_j \left(\bigvee_k (c_{lk} \wedge b_{kj}) \wedge a_{ji} \right) \\
&= (c_{lk} \otimes b_{kj}) \otimes a_{ji}
\end{aligned}
$$

However, method dependency matrix composition is *not* commutative, as can be easily seen by considering the composition of method dependency matrixes that are not square matrixes.

We can also formalize the somewhat vague intuition that a longer interface adapter chain is worse in terms of loss. If A_1 and A_2 are interface adapters, where A_1 converts I_1 to I_2 and A_2 converts I_2 to I_3, with $a_{ji} = \mathit{depend}(A_1)$ and $b_{kj} = \mathit{depend}(A_2)$ in which a_{11} and b_{11} are both true as in section 3.1.1, then for $p_k = b_{kj} \otimes \mathbf{1}'_{I_2}$ and $p'_k = b_{kj} \otimes a_{ji} \otimes \mathbf{1}'_{I_1}$:

$$
p_k = (\neg b_{k1} \vee f) \wedge \bigwedge_{j \neq 1} (\neg b_{kj} \vee t) = \neg b_{k1}
$$

$$
\begin{aligned}
p'_k &= \bigwedge_j \left(\neg b_{kj} \vee \left((\neg a_{j1} \vee f) \wedge \bigwedge_{i \neq 1} (\neg a_{ji} \vee t) \right) \right) \\
&= \bigwedge_j (\neg b_{kj} \vee \neg a_{j1}) \\
&= \neg b_{k1} \wedge \bigwedge_{j \neq 1} (\neg b_{kj} \vee a_{j1})
\end{aligned}
$$

$$\therefore\ p'_k \rightarrow p_k \tag{3.4}$$

With I_1 and I_2 being the source interfaces for the interface adapters that a_{ji} and b_{kj} represent, respectively, we can also infer from equation (3.4) that

$$\|b_{kj} \otimes \mathbf{1}'_{I_2}\| \geq \|b_{kj} \otimes a_{ji} \otimes \mathbf{1}'_{I_1}\| \tag{3.5}$$

which, along with the associativity of method dependency matrix composition, formalizes the notion that extending an interface adapter chain is worse in terms of loss.

The definitions of the method dependency matrix and the method availability vector in section 3.1.1, along with the associativity rules proven in this section, provide a succinct way to mathematically express and analyze the chaining of lossy interface adapters.

3.1.3 An example

As an example, we apply the mathematical framework to the interfaces and adapters in figure 1.2. We will denote interfaces *Video1*, *Video2*, *Video3*, and *Audio* as I_1, I_2, I_3, and I_4, respectively, while A_1, A_2, A_3, A_4, A_5, and A_6 denote the interface adapters *Video1toVideo2*, *Video2toVideo3*, *Video1toAudio*, *AudiotoVideo3*, *Video3toAudio*, and *Video3toVideo1*, respectively. We also index each method in the order they appear in figure 1.2 along with an extra dummy method with index 1, and let $a^k_{ji} = \mathit{depend}(A_k)$. Figure 1.2 is already an interface adapter graph, which is simplified and labeled in figure 3.1.

Some method dependency matrixes would be:

$$a^1_{ji} = \begin{pmatrix} t & f & f \\ f & t & f \\ t & f & f \\ t & f & f \\ t & f & f \end{pmatrix}$$

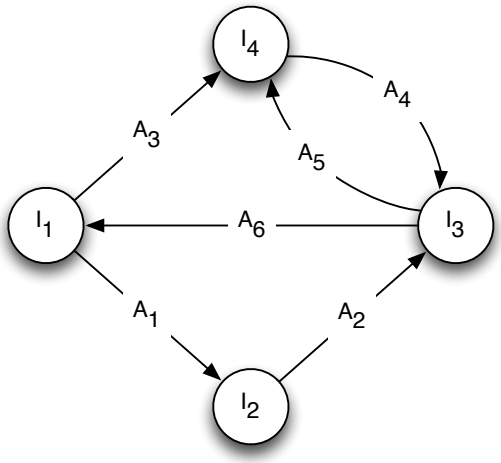

Figure 3.1: Interface adapter graph for figure 1.2.

$$a^2_{ji} = \begin{pmatrix} t & f & f & f & f \\ f & t & f & f & f \\ t & f & f & f & f \\ t & f & f & f & f \\ t & f & f & f & f \end{pmatrix}$$

$$a^5_{ji} = \begin{pmatrix} t & f & f & f & f \\ t & f & f & f & f \\ f & f & f & t & t \end{pmatrix}$$

Given a fully functional service which conforms to interface *Video1*, we would expect that only the *play* method would be available for interface *Video3* after going through the adapter chain A_1 and A_2, which can be verified by computing the method availability vector $a^2_{kj} \otimes a^1_{ji} \otimes \mathbf{1}'_{I_1}$:

$$a^2_{kj} \otimes a^1_{ji} \otimes \mathbf{1}'_{I_1} = [f, t, f, f, f]$$

One can also verify the following by hand, which would be expected from the associativity of $\otimes$. Associativity can be very useful in developing algorithms analyzing chains of lossy interface adapters, since fragments of an in-

terface adapter chain can be assembled independently and still give the same method dependency matrix for the whole chain.

$$\begin{aligned} & a^5_{lk} \otimes a^2_{kj} \otimes a^1_{ji} \otimes \mathbf{1}'_{I_1} \\ = \; & a^5_{lk} \otimes (a^2_{kj} \otimes (a^1_{ji} \otimes \mathbf{1}'_{I_1})) \\ = \; & ((a^5_{lk} \otimes a^2_{kj}) \otimes a^1_{ji}) \otimes \mathbf{1}'_{I_1} \\ = \; & (a^5_{lk} \otimes a^2_{kj}) \otimes (a^1_{ji} \otimes \mathbf{1}'_{I_1}) \\ = \; & [f, f, f] \end{aligned}$$

We can also verify the following, which is consistent with equations (3.4) and (3.5), and is in line with the intuition that extending an adapter chain can only be worse in terms of loss, although this does not mean that a longer adapter chain is always worse than a shorter adapter chain.

$$\begin{aligned} a^5_{lk} \otimes \mathbf{1}'_{I_3} &= [f, f, t] \\ a^5_{lk} \otimes a^2_{kj} \otimes \mathbf{1}'_{I_2} &= [f, f, f] \end{aligned}$$

3.2 Optimal adapter chaining

One of the things that could be hoped from the mathematical framework in section 3.1 is that it could help with the development of an efficient algorithm for obtaining an optimal interface adapter chain from an actual service to a target interface that incurs the least loss in terms of functionality. Unfortunately, the problem is NP-complete, as will be shown in this section, dashing hopes for such an algorithm.

First, we must formally describe the problem, which we will call CHAIN. Let us have an interface adapter graph $(\{I_i\}, \{A_i\})$, where $\{I_i\}$ is the set of interfaces and $\{A_i\}$ is the set of interface adapters. Let a^k be the method dependency matrix associated with adapter A_k. Let $S \in \{I_i\}$ be the source interface and $T \in \{I_i\}$ be the target interface. Then the problem is whether

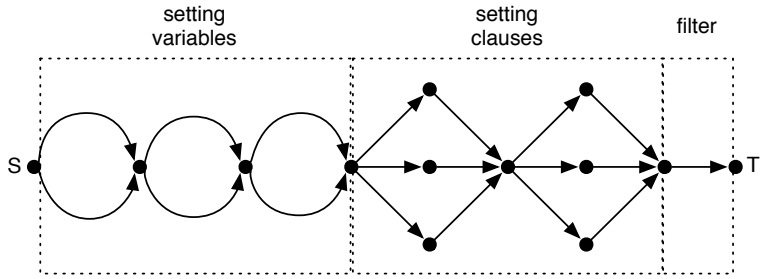

Figure 3.2: Boolean expression reduced to an interface adapter graph.

there is an interface adapter chain $[A_{P(1)}, A_{P(2)}, \ldots, A_{P(m)}]$ such that the source of $A_{P(1)}$ is S, the target of $A_{P(m)}$ is T, and $\|v^T\| = \|a^{P(m)} \otimes \cdots \otimes a^{P(2)} \otimes a^{P(1)} \otimes \mathbf{1}'_S\|$ is at least as large as some parameter N.

Informally, this is an optimization problem which tries to maximize the number of methods that can be used in a fixed target interface, obtained by applying an interface adapter chain on a fully-functional service which conforms to the source interface. We show that the problem is NP-complete by reducing 3SAT [8] to CHAIN.

Based on the conjunctive normal form of a boolean expression E with exactly 3 literals in each clause, we will construct an interface adapter graph G in three parts and the corresponding method dependency matrixes. One part will model the setting of each variable to true or false, another part will model the value of each clause once the variable values are set, and the last part will serve as a filter so that E is satisfiable if and only if there is a chain in G such that $\|v^T\|$ equals the number of clauses in E.

Figure 3.2 shows what a reduction from an instance of 3SAT to an instance of CHAIN would generally look like.

3.2.1 Representing values

We will represent values of literals and clauses using the method availability vector for each interface, where all but one of the nodes in the constructed

interface adapter graph will contain the same set of methods. At certain points in the interface adapter graph, a true or false component in the method availability vector would directly map to the value of a literal or a clause.

For almost all nodes, including the source, the set of methods will be fixed with one dummy method, one method for each clause, and one method for each literal, so almost all method dependency matrixes will be square matrixes. As the method dependency matrixes will have large parts in common with the identity matrix, we will only be mentioning how each matrix differs from the identity matrix.

Each method will be labeled as follows:

- The dummy method will be labeled d.
- For each clause c_i, the method will be labeled c_i.
- For each variable v_i, the method for the variable itself will be labeled l_i, while the method for the negation of the variable will be labeled $\overline{l_i}$.

There is a single method dependency matrix used in the filter part of the graph that will not be a square matrix.

3.2.2 Handling literals

The basic approach of this part of the graph, which we will call the *variable handling subgraph*, is to set the value for each variable depending on which adapters are chosen to be included in the chain. For each variable v_1, v_2, $\ldots$, v_v, we define nodes V_1, V_2, $\ldots$, V_v, and we let $V_0 = S$. Between each V_{i-1} and V_i, we define two adapters which will leave everything about the method availability vector unchanged from one node to the next except for the components corresponding to the literals for v_i. One will make the variable effectively true, while the other will make the variable effectively false.

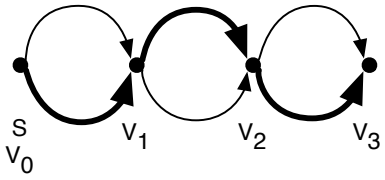

Figure 3.3: Choosing a variable assignment.

For each V_i for $i > 0$, we will define a *positive literal adapter* A_{l_i} with method dependency matrix a^{l_i} and a *negative literal adapter* $A_{\overline{l_i}}$ with method dependency matrix $a^{\overline{l_i}}$. For the positive literal adapter, $a^{l_i}_{l_i j}$ is false for all j, $a^{l_i}_{\overline{l_i} d}$ is true, and $a^{l_i}_{\overline{l_i} j}$ is false for all j other than d. Similarly for the negative literal adapter, $a^{\overline{l_i}}_{\overline{l_i} j}$ is false for all j, $a^{\overline{l_i}}_{l_i d}$ is true, and $a^{\overline{l_i}}_{l_i j}$ is false for all j other than d.

It should then be easy to see that for a method availability vector p_i with a false p_d, all components of $a^{l_i} \otimes p_i$ should be the same as p_i except for the components corresponding to l_i and $\overline{l_i}$, which will be true and false, respectively. Likewise, all components of $a^{\overline{l_i}} \otimes p_i$ should be the same as p_i except for the components corresponding to l_i and $\overline{l_i}$, which will be false and true, respectively.

The rest of the interface adapter graph will be the descendant of V_v, so any adapter chain from S to T *must* go through all of V_0, V_1, ..., V_v in order, and for every variable one and only one of the positive literal adapter or the negative literal adapter must be chosen as in figure 3.3 due to the structure of the variable handling subgraph. This is equivalent to choosing a variable assignment, and at V_v, the method availability vector p_i will be such that for each variable v_i, p_{l_i} and $p_{\overline{l_i}}$ will have opposite values, so that it would be the same as setting the value of v_i to p_{l_i}.

3.2.3 Handling clauses

Based on the variable assignment that is taken care of by the variable handling subgraph in section 3.2.2, this part of the graph, which we will call the *clause handling subgraph*, is responsible for determining the value of each clause.

In order to model disjunction, not only do we define a node C_i for each clause c_i, we also define three subnodes C_{ij}, for j from 1 to 3, for each of the literals in the clause. These nodes are separate from those defined in section 3.2.2. The idea is that if any of the literals are true, then at least one of the nodes will end up with a method availability vector marking the clause as true, so we can use this to mark the same for C_i itself. We also set $C_0 = V_v$ for convenience of notation, and c will be the number of clauses.

For each clause c_i, there are edges from C_{i-1} to each of the subnodes C_{ij}, and in turn there are edges from each subnode C_{ij} to C_i. So there will be three alternate paths from C_{i-1} to C_i.

For edge (C_{i-1}, C_{ij}), if l corresponds to the literal for C_{ij}, the method dependency matrix a for the edge is defined by setting $a_{c_i l}$ to true and $a_{c_i k}$ to false for all k other than l. Then it should be easy to see that $a \otimes p$ is the same as the method availability vector p except for the component p_{c_i}, which would be true if and only if p_l is also true. For edge (C_{ij}, C_i), the corresponding method dependency matrix is simply the identity matrix.

If clause c_i is true, then one of the literals must be true. Then the path through the subnode C_{ij} for the true literal will result in a true component for the clause in the method availability vector at C_i. If the clause is not true, then the same component will be false no matter the path taken, since it will be false for all subnodes C_{ij}.

T will be the descendant of C_c, and since the source is in the variable handling subgraph, which is only connected to the clause handling subgraph by C_0, any interface adapter chain from S to T *must* go through each of the nodes C_0, C_1, ..., C_c in order as in figure 3.4. And if all clauses are true

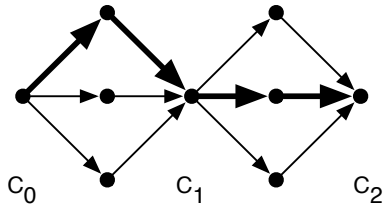

Figure 3.4: Disjunction through alternate paths.

with the variable assignment done in the variable handling subgraph, which is equivalent to choosing which adapters to include from the subgraph, only then will there be a path from C_0 to C_c which will result in true components for all clauses in the method availability vector at C_c.

3.2.4 Filtering

The last part of the constructed interface adapter graph is the filtering part, which discards all methods corresponding to literals from the method availability vector so that only the dummy method and methods corresponding to clauses remain.

The filtering subgraph is made up of only two nodes and a single edge. One of the nodes is the target T, and its interface only contains the dummy method and all the methods corresponding to clauses. The other node is C_c from section 3.2.3. The $(c+1) \times (2v+c+1)$ method dependency matrix a_{ji} for the edge from C_c to T defined as follows accomplishes the filtering:

- For all clauses c_i, $a_{c_i c_i}$ is true.
- For the dummy method, a_{dd} is true.
- All other components are false.

3.2.5 Analysis of the reduction

The constructed interface adapter graph has $v + 4c + 2$ nodes and $2v + 6c + 1$ edges, where v is the number of variables and c is the number of clauses. Also, each method dependency matrix has at most $(1+c+2v)^2$ components, so the reduction of a candidate for 3SAT to a candidate for CHAIN can be done in polynomial time. So we just need to verify that there is a positive answer for CHAIN with $N = c$ if and only if there is a positive answer for 3SAT.

If the boolean expression is satisfiable, then there is a variable assignment that makes it true. Consider the following interface adapter chain. In the variable handling subgraph, include edges that correspond to the variable assignment. In the clause handling subgraph, there is guaranteed to be a path where all components corresponding to clauses in the method availability vector at the target end up being true, given the path in the variable handling subgraph, so use this path in the chain. Then $\|v^T\|$ will be exactly c.

Conversely, suppose there is an adapter chain such that $\|v^T\| = c$. Then assigning values to variables according to the path through the variable handling subgraph results in a satisfying variable assignment for the boolean expression. This is because the clause handling subgraph and the fact that $\|v^T\| = c$ together imply that all clauses are true for the derived variable assignment. And given an arbitrary interface adapter chain and an optimal chain, it is easy to verify whether the arbitrary adapter chain is not optimal, so CHAIN is NP-complete.

3.3 A greedy algorithm

As shown in section 3.2, the problem of finding an optimal interface adapter chain that would make available the most methods in the target interface is an NP-complete problem. Short of developing a polynomial-time algorithm for an NP-complete problem, practical systems will have to use a heuristic algorithm

or an exponential-time algorithm with reasonable performance in practice.

Algorithm 1 is a greedy algorithm that finds an optimal interface adapter chain between a given source interface and a target interface. Given an interface adapter graph G, it works by looking at every possible acyclic adapter chain with an arbitrary source that results in the target interface t in order of increasing loss, taking advantage of equation (3.5), until we find a chain that starts with the desired source interface s. In this context, loss means the number of methods unavailable in the target interface given a fully functional service with the source interface, which is computed in algorithm 2, so the algorithm is guaranteed to find the optimal interface adapter chain. In the worst case, however, the algorithm takes exponential time since there can be an exponential number of acyclic chains in an interface adapter graph.

While algorithm 1 may take exponential time in the worst case, results with a similar algorithm from [34] based on a small randomly generated interface adapter graph suggest that the greedy algorithm has acceptable performance in practice.

Algorithm 1 can easily be extended to support the selection of an optimal source interface with weights associated with methods expressing their importance as in algorithm 3. This can be done by checking that the starting point of an interface adapter chain is included in a set of possible source interfaces, instead of just comparing it to a single source interface, and summing the weights for the available methods in the target interface as in algorithm 4 and using equation (3.4), instead of just counting the methods.

Unlike algorithm 1, which would find an interface adapter chain after a single service was presumably found by a service discovery process, algorithm 3 can be used in the service discovery process itself to search for the best service, not just in terms of what is required from the service, but also considering how well the client could use the service. And by weighting the methods in the target interface, it can take into account the importance of each method.

Algorithm 1 A greedy algorithm for interface adapter chaining.

procedure GREEDY-CHAIN($G = (V, E)$, s, t)
  $C \leftarrow \{[]\}$ ▷ chains to extend
  $M = \emptyset$ ▷ discarded chains
  $D \leftarrow \{[] \mapsto \mathbf{I}_{\dim(\mathbf{1}'_t)}\}$ ▷ method dependency matrixes
  while $C \neq \emptyset$ **do**
    $c \leftarrow$ element of C minimizing LOSS(c, D)
    if $c \neq [] \wedge \mathit{source}(c[1]) = s$ **then**
      return c
    else if no acyclic chain not in $C \cup M$ extends c **then**
      $C \leftarrow C - \{c\}$
      $M \leftarrow M \cup \{c\}$
    else
      if $c = []$ **then**
        $B \leftarrow \{[e] \mid e \in E, \mathit{target}(e) = t\}$
      else
        $B \leftarrow \{e : c \mid e \in E, \mathit{target}(e) = \mathit{source}(c[1])\}$
      end if
      remove cyclic chains from B
      $C \leftarrow C \cup B$
      $D \leftarrow D \cup \{e : c \mapsto D[c] \otimes \mathit{depend}(e) \mid e : c \in B\}$
    end if
  end while
end procedure

Algorithm 2 Computing the loss of an interface adapter chain.

function LOSS(c, D)

$s \leftarrow source(c[1])$

$t \leftarrow target(c[|c|])$

return $\dim(\mathbf{1}'_t) - \|D[c] \otimes \mathbf{1}'_s\|$

end function

By having sufficiently large weights for essential methods compared to those of non-essential methods, algorithm 3 can also guarantee that an adapter chain which makes all essential methods available will always be preferred over those which do not.

Algorithm 3 Greedy discovery for weighted interface adapter chaining.

procedure GREEDY-CHAIN($G = (V, E)$, S, t, W)
  $C \leftarrow \{[]\}$ ▷ chains to extend
  $M = \emptyset$ ▷ discarded chains
  $D \leftarrow \{[] \mapsto \mathbf{I}_{\dim(\mathbf{1}'_t)}\}$ ▷ method dependency matrixes
  while $C \neq \emptyset$ **do**
    $c \leftarrow$ element of C maximizing WEIGHT(c, D, W)
    if $c \neq [] \wedge \mathit{source}(c[1]) \in S$ **then**
      return $(\mathit{source}(c[1]), c)$
    else if no acyclic chain not in $C \cup M$ extends c **then**
      $C \leftarrow C - \{c\}$
      $M \leftarrow M \cup \{c\}$
    else
      if $c = []$ **then**
        $B \leftarrow \{[e] \,|\, e \in E, \mathit{target}(e) = t\}$
      else
        $B \leftarrow \{e : c \,|\, e \in E, \mathit{target}(e) = \mathit{source}(c[1])\}$
      end if
      remove cyclic chains from B
      $C \leftarrow C \cup B$
      $D \leftarrow D \cup \{e : c \mapsto D[c] \otimes \mathit{depend}(e) \,|\, e : c \in B\}$
    end if
  end while
end procedure

Algorithm 4 Computing the weight of an interface adapter chain.

function WEIGHT(c, D, $W = w_i$)

$s \leftarrow source(c[1])$

$t \leftarrow target(c[|c|])$

$p_i \leftarrow D[c] \otimes \mathbf{1}'_s$

return $\sum_{p_i} w_i$

end function

4. Probabilistic chains

Chapter 3 took the approach of assuming that a method in a target interface could be implemented as long as all the prerequisite methods in the source interface were available. This also implies that any argument given to the method in the target interface can always be converted into arguments that the methods in the source interface can handle, as well as it always being possible for the results to be converted to a form appropriate for the target interface. This assumption is not always true: for a trivial example, negative numbers for a square root function cannot be handled if either the source interface or target interface are unaware of imaginary numbers.

This chapter describes a probabilistic approach to handling the *partial* adaptation of methods, where the loss may occur not just due to missing functionality or methods, but also due to an interface adapter being unable to handle all arguments given for a method in a target interface. The approach extends the work described in chapter 3.

4.1 Mathematical basics

As in section 3.1, we can define an interface adapter graph, which is a directed graph where interfaces are nodes and adapters are edges. A range convention for the index notation used to express matrixes and vectors will also be in effect [11].

$V_{m,I}(a)$	Method m of interface I can properly handle argument a.
$V_{m,I}$	Method m of interface I can properly handle its argument.
$C^{A}_{m \to m'}$	Interface adapter A can successfully convert an argument for method m in the target interface to an argument for method m' in the source interface and convert back the result.

Table 4.1: Probabilistic events.

4.1.1 Method dependencies

We develop a probabilistic approach by starting off with the most general form of expressing the probabilities and adding assumptions until we have a probabilistic formula that is practical. Without additional assumptions, the probabilities can only be expressed in a way that is useless for analyzing real systems. The additional assumptions allow us to express the desired probabilities in a way that they can be feasibly computed from a set of values that can be measured in practice.

We first describe the notation for expressing certain probabilistic events in table 4.1. These events denote whether a method can handle a given argument, or whether an interface adapter can convert an argument for a method in a target interface to an argument for a method in the source interface and successfully convert back the result. We assume that a method only accepts a single argument: this is not a problem since methods with multiple arguments can simply be modeled as a method accepting a single tuple with multiple components [70]. If a method does not need an argument, we treat it as receiving a dummy argument anyway.

Let us say that we wish to adapt methods in source interface I_S into method j in target interface I_T. The most general form for expressing the probability that a method could handle an argument is to sum the probabilities for every possible argument, where we must consider the probability of the method receiving a specific argument and then the probability that the method can handle it:

$$P(V_{j,I_T}) = \sum_a P(V_{j,I_T}(a))P(A = a) \tag{4.1}$$

The most general form for expressing the probability requires that we know the probability distribution of arguments, which is not feasible except for the simplest of argument domains. For example, the probability distribution for a simple integer argument may require 2^{32} or 2^{64} probabilities to be expressed for the typical computer architecture, and even measuring such a probability distribution may not be feasible in the first place. It is also not feasible that we already know the probabilities for how a method can handle each and every possible argument.

For this reason, we make the assumption that the probabilities do *not* depend on the specific arguments. Given this assumption, we can now express $P(V_{j,I_T})$ in terms of whether an argument can be converted and whether it can be handled. More specifically, this means that for *all* methods in the source interface that the interface adapter A requires to implement a method in the target interface, it must be the case that the argument can be converted *and* the method in the source interface can handle the converted argument. Using the method dependency matrix defined in section 3.1.1, $P(V_{j,I_T})$ can be expressed as:

$$P(V_{j,I_T}) = P\left(\bigcap_{a_{ji}} \left(V_{i,I_S} \cap C^A_{j\to i}\right)\right) \tag{4.2}$$

This is still too unwieldy an expression to be practical, since it is unclear

how dependencies in the events for different methods in the source interface affect the overall probability. It would also be unclear how to measure the probabilities beforehand without trying out every possible argument and configuration of interface adapter chains, something that is clearly not feasible. Therefore we make an additional assumption that the events for separate methods in the source interface are independent.

With the additional assumption, $P(V_{j,I_T})$ can be expressed as:

$$P(V_{j,I_T}) = \prod_{a_{ji}} P(V_{i,I_S} \cap C^A_{j \to i}) \tag{4.3}$$

However, equation (4.3) is still not appropriate for practical use. The reason is that it entangles the work done by the interface adapter and whether the method in the source interface can handle the converted argument. Basically, the probabilities intrinsic to the interface adapter and the source interface are entangled. If the source interface itself is the result of adaptation through an interface adapter chain, then we have the problem of a configuration-dependent event being entangled with a configuration-independent event, and there is no simple way to derive the required probabilities.

Thus we make one final additional assumption that the probability an interface adapter can successfully convert arguments and results is independent from the probability that a method in the source interface can handle an argument. This allows us to express $P(V_{j,I_T})$ as:

$$P(V_{j,I_T}) = \prod_{a_{ji}} P(V_{i,I_S}) P(C^A_{j \to i}) \tag{4.4}$$

Equation (4.4) is finally in a form that can be used practically. The probability that an interface adapter A can successfully convert an argument for method j in the target interface to an argument for method i in the source interface, $P(C^A_{j \to i})$, is a value that is intrinsic to an interface adapter. In principal, it could be measured empirically or obtained through analysis of the

interface adapter code. The probability that method m_i in source interface I_S can handle an argument, $P(V_{i,I_S})$, is also a value that can be obtained, either through analytical or empirical means if the interface of a service is being adapted, or through a recursive application of equation (4.4).

We now have the basis for describing a framework similar to the one developed for the discrete chain approach. As in chapter 3, we define a method availability vector and a method dependency matrix, but in addition we also define a *conversion probability matrix*.

As before, the method availability vector p_i expresses how well a method is supported in an interface, and it is not intrinsic to an interface but rather represents the loss from interface adaptation. Unlike in section 3.1.1, however, where the method availability vector is a boolean vector merely expressing whether a method is available or not, the components for a method availability vector in the probabilistic approach are probabilities. p_i is defined as the probability that method i can handle an argument it receives, i.e. $p_i = P(V_{i,I})$.

The method dependency matrix is the same as defined in section 3.1.1 and is used in equation (4.4). Unlike for the discrete chain approach, however, the method dependency matrix does not suffice to describe the relevant information for an interface adapter. We also require a set of probabilities $P(C^A_{j\to i})$ for how well an interface adapter converts an argument for a method in the target interface to that for the relevant method in the source interface. The conversion probability matrix t_{ji} is defined in terms of these probabilities, where $t_{ji} = P(C^A_{j\to i})$.

Given method availability vector p_i, method dependency matrix a_{ji}, and conversion probability matrix t_{ji}, we can now define the adaptation operator $\otimes$. Instead of just the method dependency matrix being applied to the method availability vector, the conversion probability matrix must also be applied in conjunction with the method dependency matrix:

$$(a_{ji}, t_{ji}) \otimes p_i = \prod_{a_{ji}} t_{ji}\, p_i \tag{4.5}$$

We will also call a tuple of a method dependency matrix and a conversion probability matrix such as (a_{ji}, t_{ji}) a *probabilistic adaptation factor*. We will denote the probabilistic adaptation factor for an interface adapter A as $depend(A)$.

It should be emphasized that equation (4.5) is only rigorously correct given the following three assumptions. However, the three assumptions make it possible to feasibly compute $P(V_{i,I})$ from values that can be feasibly measured or estimated a priori in a rigorously sound manner, instead of having to define an ad hoc computational framework where definitions are vague in their operational meaning. It is still an open question of how closely real systems would fit these assumptions.

- The probabilities do *not* depend on the specific arguments.
- The events for separate methods in the source interface are independent.
- The probability that an interface adapter can successfully convert arguments and results is independent from the probability that a method in the source interface can handle an argument.

As in section 3.1.1, it should be noted that equation (4.5) is incomplete in that it is ambiguous what the result should be when no a_{ji} is true. If this is the case, it could be that the method in the target interface can always be implemented regardless of availability of methods in the source interface, or it could be that the method cannot be implemented no matter what.

The workaround is simple: as in section 3.1.1, a dummy method is defined for each interface, where the method dependency matrixes follow the same rules. For the conversion probability matrix, setting t_{j1} to zero for all j would yield the expected results, given the usual convention that an empty product

has a value of one [38].[1] We will denote a method availability vector for interface I in which all methods are available and can handle all arguments by $\mathbf{1}'_I$, where all components have value one except for the component corresponding to the dummy method, which has value zero.

4.1.2 Adapter composition

As in section 3.1.2, we would like to be able to derive a composite probabilistic adaptation factor from the composition of two probabilistic adaptation factors, which would be equivalent to describing the chaining of two interface adapters as if they were a single interface adapter.

Given interfaces I_1, I_2, and I_3, let the corresponding method availability vectors be p_i, q_j, and r_k. In addition, let there be interface adapters A_1 and A_2, where A_1 converts I_1 to I_2 and A_2 converts I_2 to I_3, with corresponding probabilistic adaptation factors (a_{ji}, t_{ji}) and (b_{kj}, u_{kj}), respectively. We would like to know how to derive the probabilistic adaptation factor (c_{ki}, v_{ki}) that would correspond to an interface adapter equivalent to A_1 and A_2 chained together.

c_{ki} is obviously derived the same way as done in section 3.1.2. As for v_{ki}, from equation (4.4) and our assumptions:

$$\begin{aligned} r_k &= \prod_{b_{kj}} u_{kj}\, q_j \\ &= \prod_{b_{kj}} \left(u_{kj} \prod_{a_{ji}} t_{ji}\, p_i \right) \\ &= \prod_{b_{kj}} \prod_{a_{ji}} u_{kj}\, t_{ji}\, p_i \end{aligned}$$

[1] The values for t_{1i} do not matter except for $i = 1$, so they can be arbitrarily set to zero.

$$= \prod_{b_{kj} \wedge a_{ji}} u_{kj}\, t_{ji}\, p_i \tag{4.6}$$

We want the above to be equivalent to the following:

$$\begin{aligned} r_k &= \prod_{c_{ki}} v_{ki}\, p_i \\ &= \prod_{\bigvee_j (b_{kj} \wedge a_{ji})} v_{ki}\, p_i \end{aligned} \tag{4.7}$$

The composition operator is derived by carefully considering the terms in equations (4.6) and (4.7), based on collecting the terms for fixed i.

If we collect the terms in equation (4.6) with fixed i, we have (4.8). It should be emphasized that (4.8) is *not* identical to (4.6): the former is a product over varying j with *both* i and k fixed, while the latter is a product over varying i and j with only k fixed. Also note that if $b_{kj} \wedge a_{ji}$ are all false for varying j, then no terms affect the result of (4.6). This would be equivalent to (4.8) having a value of one, which is expected from an empty product.

$$\prod_{b_{kj} \wedge a_{ji}} u_{kj}\, t_{ji}\, p_i \tag{4.8}$$

On the other hand, consider the term in equation (4.7) with fixed i. If $\bigvee_j (b_{kj} \wedge a_{ji})$ is false, i.e. $b_{kj} \wedge a_{ji}$ are all false for varying j, then the term is excluded from the product and is equivalent to multiplying by one, instead. If it is true, on the other hand, then $v_{ki}\, p_i$ is the term that corresponds to the fixed i. So if we set $v_{ki}\, p_i$ according to (4.9),[2] then equations (4.7) and (4.6) end up having the exact same values.

$$v_{ki} = \prod_{b_{kj} \wedge a_{ji}} u_{kj}\, t_{ji} \tag{4.9}$$

[2]Remember that only k is fixed in (4.6) and (4.7), but both k and i are fixed in (4.9).

From this, we can conclude that the composition operator $\otimes$ for two probabilistic adaptation factors should be defined as:

$$(b_{kj}, u_{kj}) \otimes (a_{ji}, t_{ji}) = (b_{kj} \otimes a_{kj}, \prod_{b_{kj} \wedge a_{ji}} u_{kj}\, t_{ji}) \tag{4.10}$$

The $\otimes$ operator is "associative" when applied to a probabilistic adaptation factors and a method availability vector, i.e. $(b_{kj}, u_{kj}) \otimes ((a_{ji}, t_{ji}) \otimes p_i) = ((b_{kj}, u_{kj}) \otimes (a_{ji}, t_{ji})) \otimes p_i$.[3]

$$\begin{aligned}
(b_{kj}, u_{kj}) \otimes ((a_{ji}, t_{ji}) \otimes p_i) &= (b_{kj}, u_{kj}) \otimes \prod_{a_{ji}} t_{ji}\, p_i \\
&= \prod_{b_{kj}} u_{kj} \prod_{a_{ji}} t_{ji}\, p_i \\
&= \prod_{b_{kj}} \prod_{a_{ji}} u_{kj}\, t_{ji}\, p_i \\
&= \prod_{b_{kj} \wedge a_{ji}} u_{kj}\, t_{ji}\, p_i \\
&= \prod_{\bigvee_j (b_{kj} \wedge a_{ji})} \prod_{b_{kj} \wedge a_{ji}} u_{kj}\, t_{ji}\, p_i \\
&= \prod_{b_{kj} \otimes a_{ji}} \left(\prod_{b_{kj} \wedge a_{ji}} u_{kj}\, t_{ji} \right) p_i \\
&= (b_{kj} \otimes a_{ji}, \prod_{b_{kj} \wedge a_{ji}} u_{kj}\, t_{ji}) \otimes p_i \\
&= ((b_{kj}, u_{kj}) \otimes (a_{ji}, t_{ji})) \otimes p_i
\end{aligned}$$

Likewise, probabilistic adaptation factor composition is associative, where the following derivation depends on the fact that $b_{kj} \otimes a_{ji} = \bigvee_j (b_{kj} \wedge a_{ji})$ must be true if $b_{kj} \wedge a_{ji}$ is true:

[3] It is technically not associative in this context since the $\otimes$ operator in $(b_{kj}, u_{kj}) \otimes (a_{ji}, t_{ji})$ is not the same as the $\otimes$ operator in $(a_{ji}, t_{ji}) \otimes p_i$.

$$
\begin{aligned}
&(c_{lk}, v_{lk}) \otimes ((b_{kj}, u_{kj}) \otimes (a_{ji}, t_{ji})) \\
&= (c_{lk}, v_{lk}) \otimes (b_{kj} \otimes a_{kj}, \prod_{b_{kj} \wedge a_{ji}} u_{kj}\, t_{ji}) \\
&= (c_{lk} \otimes b_{kj} \otimes a_{kj}, \prod_{c_{lk} \wedge (b_{kj} \otimes a_{kj})} v_{lk} \prod_{b_{kj} \wedge a_{ji}} u_{kj}\, t_{ji}) \\
&= (c_{lk} \otimes b_{kj} \otimes a_{kj}, \prod_{c_{lk} \wedge b_{kj} \wedge a_{ji} \wedge (b_{kj} \otimes a_{kj})} v_{lk}\, u_{kj}\, t_{ji}) \\
&= (c_{lk} \otimes b_{kj} \otimes a_{kj}, \prod_{c_{lk} \wedge b_{kj} \wedge a_{ji}} v_{lk}\, u_{kj}\, t_{ji}) \\
&= (c_{lk} \otimes b_{kj} \otimes a_{ji}, \prod_{(c_{lk} \otimes b_{kj}) \wedge c_{lk} \wedge b_{kj} \wedge a_{ji}} v_{lk}\, u_{kj}\, t_{ji} \\
&= (c_{lk} \otimes b_{kj} \otimes a_{ji}, \prod_{(c_{lk} \otimes b_{kj}) \wedge a_{ji}} \left(\prod_{c_{lk} \wedge b_{kj}} v_{lk}\, u_{kj} \right) t_{ji} \\
&= (c_{lk} \otimes b_{kj}, \prod_{c_{lk} \wedge b_{kj}} v_{lk}\, u_{kj}) \otimes (a_{ji}, t_{ji}) \\
&= ((c_{lk}, v_{lk}) \otimes (b_{kj}, u_{kj})) \otimes (a_{ji}, t_{ji})
\end{aligned}
$$

However, probabilistic adaptation factor composition is *not* commutative, as can be easily seen by considering the composition of probabilistic adaptation factors whose components are not square matrixes.

We can also show a monotonicity property similar to equation (3.4), which formalizes the notion that extending an interface adapter chain results in worse adaptation loss. If A_1 and A_2 are interface adapters, where A_1 converts I_1 to I_2 and A_2 converts I_2 to I_3, with $(a_{ji}, t_{ji}) = \mathit{depend}(A_1)$ and $(b_{kj}, u_{kj}) = \mathit{depend}(A_2)$ where they follow the rules for the dummy method in sections 3.1.1 and 4.1.1, then let $p_k = (b_{kj}, u_{kj}) \otimes \mathbf{1}'_{I_2}$ and $p'_k = (b_{kj}, u_{kj}) \otimes (a_{ji}, t_{ji}) \otimes \mathbf{1}'_{I_1}$. We have:

$$p_1 = p'_1 = 0$$

$$p_k = \prod_{j \neq 1 \wedge b_{kj}} u_{kj} \tag{4.11}$$

$$p'_k = \prod_{i \neq 1 \wedge b_{kj} \wedge a_{ji}} u_{kj}\, t_{ji} = \prod_{b_{kj}} \prod_{i \neq 1 \wedge a_{ji}} u_{kj}\, t_{ji} = \prod_{b_{kj}} \left(u_{kj} \prod_{i \neq 1 \wedge a_{ji}} t_{ji} \right) \tag{4.12}$$

If method k can never be implemented given the source interface, then b_{k1} will be true, and given that u_{k1} will be zero, p'_k will also have to be zero. Otherwise, b_{k1} will be false, so we can do a term by term comparison of equations (4.11) and (4.12), taking advantage of the fact that u_{kj} and t_{ji} are probabilities so that they are greater than or equal to zero and lesser than or equal to one:

$$0 \leq \prod_{i \neq 1 \wedge a_{ji}} t_{ji} \leq 1$$

$$u_{kj} \prod_{i \neq 1 \wedge a_{ji}} t_{ji} \leq u_{kj}$$

$$\therefore\ p'_k \leq p_k \tag{4.13}$$

The definitions of the method dependency matrix and the method availability vector in section 4.1.1, along with the associativity rules proven in this section, provide a succinct way to mathematically express and analyze the chaining of lossy interface adapters using a probabilistic approach.

4.1.3 An example

As an example, we apply the probabilistic approach to the interfaces and adapters in figure 1.2. As in section 3.1.3, we will denote interfaces *Video1*, *Video2*, *Video3*, and *Audio* as I_1, I_2, I_3, and I_4, respectively, while A_1, A_2, A_3, A_4, A_5, and A_6 denote the interface adapters *Video1toVideo2*, *Video2toVideo3*,

Video1toAudio, *AudiotoVideo3*, *Video3toAudio*, and *Video3toVideo1*, respectively. We also index each method in the order they appear in figure 1.2 along with an extra dummy method with index 1, and let $(a^k_{ji}, t^k_{ji}) = depend(A_k)$.

The method dependency matrixes are the same as with the discrete approach of chapter 3, and we repeat those shown in section 3.1.3 here:

$$a^1_{ji} = \begin{pmatrix} t & f & f \\ f & t & f \\ t & f & f \\ t & f & f \\ t & f & f \end{pmatrix}$$

$$a^2_{ji} = \begin{pmatrix} t & f & f & f & f \\ f & t & f & f & f \\ t & f & f & f & f \\ t & f & f & f & f \\ t & f & f & f & f \end{pmatrix}$$

$$a^5_{ji} = \begin{pmatrix} t & f & f & f & f \\ t & f & f & f & f \\ f & f & f & t & t \end{pmatrix}$$

Figure 1.2 does not specify the conversion probabilities of arguments by the interface adapters. In a real system, the probabilities must be obtained experimentally by testing many of the arguments or a priori by a developer reasoning out the probabilities. Here we let the probabilities for our example be those specified in table 4.2, in which case the corresponding conversion

A	j	i	t_{ji}
A_1	2	2	1.0
A_2	2	2	0.9
A_3	2	3	1.0
A_4	4	3	0.8
A_4	5	3	0.6
A_5	3	4	1.0
A_5	3	5	0.9
A_6	2	2	0.8

Table 4.2: Example conversion probabilities for figure 1.2.

probability matrixes are:

$$t^1_{ji} = \begin{pmatrix} 0.0 & 0.0 & 0.0 \\ 0.0 & 1.0 & 0.0 \\ 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 \end{pmatrix}$$

$$t^2_{ji} = \begin{pmatrix} 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.9 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \end{pmatrix}$$

$$t^5_{ji} = \begin{pmatrix} 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 1.0 & 0.9 \end{pmatrix}$$

From section 3.1.3, we know that the *play* method would be available for interface *Video3* after going through the adapter chain A_1 and A_2 given a fully functional service which conforms to interface *Video1*, and we can infer that

play can handle its argument with roughly probability 0.9 by computing the probabilistic adaptation factor $(a^2_{kj}, t^2_{kj}) \otimes (a^1_{ji}, t^1_{ji}) \otimes \mathbf{1}'_{I_1}$:

$$(a^2_{kj}, t^2_{kj}) \otimes (a^1_{ji}, t^1_{ji}) \otimes \mathbf{1}'_{I_1} = ([f, t, f, f, f], [0.0, 0.9, 0.0, 0.0, 0.0])$$

One can also verify the following by hand, which would be expected from the associativity of $\otimes$:

$$\begin{aligned}
&(a^5_{lk}, t^5_{lk}) \otimes (a^2_{kj}, t^2_{kj}) \otimes (a^1_{ji}, t^1_{ji}) \otimes \mathbf{1}'_{I_1} \\
&= (a^5_{lk}, t^5_{lk}) \otimes ((a^2_{kj}, t^2_{kj}) \otimes ((a^1_{ji}, t^1_{ji}) \otimes \mathbf{1}'_{I_1})) \\
&= (((a^5_{lk}, t^5_{lk}) \otimes (a^2_{kj}, t^2_{kj})) \otimes (a^1_{ji}, t^1_{ji})) \otimes \mathbf{1}'_{I_1} \\
&= ((a^5_{lk}, t^5_{lk}) \otimes (a^2_{kj}, t^2_{kj})) \otimes ((a^1_{ji}, t^1_{ji}) \otimes \mathbf{1}'_{I_1}) \\
&= ([f, f, f], [0.0, 0.0, 0.0])
\end{aligned}$$

We can also verify the following, which is consistent with equations (4.13), and is in line with the intuition that extending an adapter chain can only be worse in terms of loss, although this does not mean that a longer adapter chain is always worse than a shorter adapter chain.

$$\begin{aligned}
(a^5_{lk}, t^5_{lk}) \otimes \mathbf{1}'_{I_3} &= ([f, f, t], [0.0, 0.0, 0.9]) \\
(a^5_{lk}, t^5_{lk}) \otimes (a^2_{kj}, t^2_{lk}) \otimes \mathbf{1}'_{I_2} &= ([f, f, f], [0.0, 0.0, 0.0])
\end{aligned}$$

4.2 Optimal adapter chaining

Like the optimal adapter chaining problem with the discrete chain approach, the optimal adapter chaining problem with the probabilistic approach is NP-complete as well. This is intuitively the case since the probabilistic approach should be able to encompass the discrete approach, and we show this formally in this section.

We first formally define the optimal adapter chaining problem in the probabilistic approach, which we will call PROB-CHAIN. Let us have an interface

adapter graph $(\{I_i\}, \{A_i\})$, where $\{I_i\}$ is the set of interfaces and $\{A_i\}$ is the set of interface adapters. Let f^k be the probabilistic adaptation factor associated with adapter A_k. Let $S \in \{I_i\}$ be the source interface and $T \in \{I_i\}$ be the target interface. Let $\{r_m\}$ be the relative invocation probabilities for the methods in the target interface such that $\sum_m r_m = 1$. Then the problem is whether there is an interface adapter chain $[A_{P(1)}, A_{P(2)}, \ldots, A_{P(n)}]$ such that the source of $A_{P(1)}$ is S, the target of $A_{P(n)}$ is T, and $\sum_m r_m \, v_m^T$ is at least as large as some probability X, where $v^T = f^{P(n)} \otimes \cdots \otimes f^{P(2)} \otimes f^{P(1)} \otimes \mathbf{1}'_S$.

Informally, this is an optimization problem which tries to maximize the probability that an argument can be handled by a method in a fixed target interface, obtained by applying an interface adapter chain on a fully-functional service which conforms to the source interface. $\{r_m\}$ would express how often methods are invoked relative to each other.

We next show how the probabilistic approach can encompass the discrete approach of chapter 3. Let there be a method availability vector p_i and a method dependency matrix a_{ji} as expressed in the discrete approach. We construct corresponding method availability vector p'_i, method dependency matrix a'_{ji}, and conversion probability matrix t'_{ji} as expressed in the probabilistic approach as follows. If p_i is true, then set p'_i to one, else set p'_i to zero. a'_{ji} is just the same as a_{ji}. And set all t'_{ji} to one. Then we have:

$$a_{ji} \otimes p_i = \bigwedge_j (a_{ji} \rightarrow p_i) = \bigwedge_{a_{ji}} p_i$$

$$(a'_{ji}, t'_{ji}) \otimes p'_i = \prod_{a'_{ji}} t'_{ji} \, p'_i = \prod_{a_{ji}} p'_i$$

and it is easy to see that a component of $a_{ji} \otimes p_i$ is true if and only if the corresponding component of $(a'_{ji}, t'_{ji}) \otimes p'_i$ is one, and that a component of $a_{ji} \otimes p_i$ is false if and only if the corresponding component of $(a'_{ji}, t'_{ji}) \otimes p'_i$ is zero.

This shows how an interface adapter graph for the discrete approach can be

converted to one for the probabilistic approach in a way that the adaptation operators in both approaches basically have the same behavior. Since all the mathematics for both approaches follow from the definition of the adaptation operators, we have just shown that the probabilistic approach can encompass the discrete approach.

Finally, we show that PROB-CHAIN can be used to solve CHAIN from section 3.2. Given M methods in the target interface, use the method described above to convert an input for CHAIN to an input for PROB-CHAIN, where we also set all r_m to $\frac{1}{M}$. Then $\sum_m r_m v_m^T$ will be $\frac{n}{M}$, where n is the number of methods available from the interface adapter chain, so PROB-CHAIN with X set to $\frac{N}{M}$ will solve CHAIN. Since CHAIN is NP-complete and it is easy to verify if an alternate chain results in smaller $\sum_m r_m v_m^T$, PROB-CHAIN must also be NP-complete.

4.3 A greedy algorithm

As shown in section 4.2, the problem of finding an optimal interface adapter chain maximizing the probability of an argument being handled by a method in the target interface is an NP-complete problem. Short of developing a polynomial-time algorithm for an NP-complete problem, practical systems will have to use a heuristic algorithm or an exponential-time algorithm with reasonable performance in practice.

For the optimal interface chaining problem in the probabilistic approach, the monotonicity property as expressed in equation (4.13) allows us to use a very similar algorithm to the one in section 3.3. Algorithm 5 is a greedy algorithm that finds an optimal interface adapter chain between a given source interface and a target interface. Given an interface adapter graph G, it works by looking at every possible acyclic adapter chain with an arbitrary source that results in the target interface t in order of increasing loss, taking advantage

of equation (4.13), until we find a chain that starts with the desired source interface s.

In this context, loss means the probability that a method in the target interface *cannot* handle an argument given a fully functional service with the source interface, which is computed in algorithm 6, so the algorithm is guaranteed to find the optimal interface adapter chain. In the worst case, however, the algorithm takes exponential time since there can be an exponential number of acyclic chains in an interface adapter graph.

Just as algorithm 1 can be extended to algorithm 3 to support behavior similar to service discovery by checking whether the current source is among a potential set of source interfaces instead of just checking against one, algorithm 5 can be similarly extended.

Algorithm 5 A probabilistic greedy algorithm for interface adapter chaining.

procedure PROB-GREEDY-CHAIN($G = (V, E)$, s, t, $\{r_m\}$)
  $C \leftarrow \{[]\}$ ▷ chains to extend
  $M = \emptyset$ ▷ discarded chains
  $D \leftarrow \{[] \mapsto \mathbf{I}_{\dim(\mathbf{1}'_t)}\}$ ▷ method dependency matrixes
  while $C \neq \emptyset$ **do**
    $c \leftarrow$ element of C minimizing PROB-LOSS($c, D, \{r_m\}$)
    if $c \neq [] \wedge \mathit{source}(c[1]) = s$ **then**
      return c
    else if no acyclic chain not in $C \cup M$ extends c **then**
      $C \leftarrow C - \{c\}$
      $M \leftarrow M \cup \{c\}$
    else
      if $c = []$ **then**
        $B \leftarrow \{[e] \mid e \in E, \mathit{target}(e) = t\}$
      else
        $B \leftarrow \{e : c \mid e \in E, \mathit{target}(e) = \mathit{source}(c[1])\}$
      end if
      remove cyclic chains from B
      $C \leftarrow C \cup B$
      $D \leftarrow D \cup \{e : c \mapsto D[c] \otimes \mathit{depend}(e) \mid e : c \in B\}$
    end if
  end while
end procedure

Algorithm 6 Computing the probabilistic loss of an interface adapter chain.

function PROB-LOSS(c, D, $\{r_m\}$)

$s \leftarrow \mathit{source}(c[1])$

$v \leftarrow D[c] \otimes \mathbf{1}'_s$

return $1 - \sum_m r_m \, v_m$

end function

5. Abstract interpretation

Chapter 3 took the approach of treating a method as a single unit that is either available or not available, while chapter 4 loosened this by treating the availability of a method probabilistically. Another approach to handling the partial adaptation of methods is to use a simple form of abstract interpretation [9, 10, 12, 30], which is the subject of this chapter.

Dividing arguments up into abstract domains, the abstract interpretation approach looks at which abstract domains can be handled by a method instead of looking at the likelihood. Interface adapters are "executed abstractly" to derive which abstract domains can be handled by a method in the target interface.

5.1 Mathematical basics

As in section 3.1, we can define an interface adapter graph, which is a directed graph where interfaces are nodes and adapters are edges. We also assume that a method accepts only a single argument: multiple arguments can be modeled as a tuple with multiple components [70], while no argument can be modeled by a dummy argument.

For each method in an interface, its argument domain is divided up into disjoint domains that are each represented by abstract values. For example, integer arguments could be mapped to abstract values d_+, d_-, and d_0. These domains and abstract values must be fixed for each method in an interface, and they must not be different for different interface adapters. The chaining of interface adapters cannot be analyzed otherwise.

There is a special abstract value denoted by $\perp$ distinct from any other

abstract value, which is used when a method is unable to handle any other abstract value, i.e. when a method cannot handle any arguments. Including $\perp$ in the set of all possible abstract values for a method can be considered lifting the set [50], and we call this lifted set the *abstract argument domain* for the method.

5.1.1 Method dependencies

To represent how methods in a source interface are used to implement a method in a target interface, including which arguments must be used for the methods in the source interface to handle an argument for the method in the target interface, we use a function-based approach. Functions are represented with a set-theoretic approach [51], where a function is a set of pairs of arguments and values.

Let there be an interface adapter A with source interface I_S and target interface I_T. Let D_i be the abstract argument domain of method i in I_S, and let D'_j be the abstract argument domain of method j in I_T. If I_S has n methods and I_T has n' methods, then we can define the *abstract dependency function* h_A for A:

$$h_A : D_1 \times D_2 \times \cdots \times D_n \rightarrow 2^{D'_1} \times 2^{D'_2} \times \cdots \times 2^{D'_{n'}}$$

where the jth component in a result tuple is the set of all possible abstract values that can be handled by method j in the target interface given the specified abstract values accepted by the methods in the source interface. If a method is unavailable in the source interface or it is unable to handle any arguments, then its corresponding abstract value in the argument to h_A would be $\perp$. All components in the result tuple include $\perp$ so that chained interface adapters can be analyzed even when certain methods end up being unavailable.

We also define a method availability vector p for the abstract interpretation approach. It is simply a tuple of sets with type $2^{D_1} \times 2^{D_2} \times \cdots \times 2^{D_n}$, assuming there are n methods and D_i is the abstract argument domain for method i.

It represents the abstract values that each method is able to handle, and it is not intrinsic to an interface or interface adapter. Instead, it expresses how well interface adaptation is done.

In the following, we will use the shorthand $\cup$ for the "union" of two tuples, in which the result is a tuple with the same number of components, and each component is the union of the corresponding components in the arguments. In other words, $[S_1, \ldots, S_n] \cup [S'_1, \ldots, S'_n] = [S_1 \cup S'_1, \ldots, S_n \cup S'_n]$. The "indexed union" of tuples is a straightforward extension of the shorthand. We will also denote the Cartesian product of the components of a tuple of sets with the prefix $\times$. In other words, $\times[S_1, \ldots, S_n] = S_1 \times \cdots \times S_n$.

If an interface adapter A with abstract dependency function h_A were to be used to convert an source interface I_S to target interface I_T, where the method availability vector for I_S is p, then it is easy to see that each component of the resulting method availability vector q should be:

$$q = \bigcup_{x \in (\times p)} h_A(x) \tag{5.1}$$

Basically, each component of q should be the union of all possible abstract values for the corresponding method as adapted from p through h_A.

It would be more convenient to simply apply a function to a method availability vector, and this can easily be done by defining an *abstract adaptation function* f_A corresponding to an abstract dependency function h_A based on equation (5.1), where D_1, ..., D_n are the abstract argument domains for the source interface of the adapter:

$$f_A = \{(X, \bigcup_{x \in X} h_A(x)) \mid X \subseteq D_1 \times \cdots \times D_n\} \tag{5.2}$$

Obviously, constructing an abstract dependency function for an interface adapter is easier than constructing an abstract adaptation function, so it would be expected that the abstract dependency function is constructed first, and then the abstract adaptation function is constructed from this, after which the

abstract adaptation function would be used to analyze interface adapter chains. We will denote the abstract adaptation function of an interface adapter A by $depend(A)$.

Now that we have defined the abstract adaptation function, defining the adaptation operator is simple: it is simply function application. In fact, we will not use special notation to represent the operator and will just use the standard notation:

$$f_A(p)$$

Unlike in chapter 3 or chapter 4, there is no need to define something like a dummy method since an adaptation dependency function is general enough to encompass cases which are ambiguous in the other approaches. If a method in a target interface can always be implemented, then the appropriate abstract value results can be specified for an argument of $[\bot, \ldots, \bot]$, whereas if a method can never be implemented, then $[\{\bot\}, \ldots, \{\bot\}]$ can be returned as the result.

We denote a method availability vector for an interface I to a fully functional service by $\mathbf{1}_I$, where each set in the tuple includes the entire abstract argument domain for the corresponding method.

5.1.2 Adapter composition

As in section 3.1.2, we would like to be able to derive a composite abstract adaptation function from the composition of two abstract adaptation functions, which would be equivalent to describing the chaining of two interface adapters as if they were a single interface adapter.

For the abstract interpretation approach, this is very straightforward because the abstract adaptation function is a function: the composition is just function composition. And function composition is well known to be associative [73], although not commutative in general. In fact, we will use the

standard notation for abstract adaptation function composition:

$$f_{A'} \circ f_A$$

We can also show a monotonicity property similar to equation (3.4), which formalizes the notion that extending an interface adapter chain results in worse adaptation loss. If A_1 and A_2 are interface adapters, where A_1 converts I_1 to I_2 and A_2 converts I_2 to I_3, with $f_{A_1} = \mathit{depend}(A_1)$ and $f_{A_2} = \mathit{depend}(A_2)$, and h_{A_1} and h_{A_2} are the adaptation dependency functions associated with f_{A_1} and f_{A_2}, respectively, then:

$$f_{A_2}(z) = \bigcup_{x \in (\times z)} h_{A_2}(x) \subseteq \bigcup_{x \in (\times \mathbf{1}_{I_2})} h_{A_2}(x) = f_{A_2}(\mathbf{1}_{I_2})$$

$$\therefore\ (f_{A_2} \circ f_{A_1})(\mathbf{1}_{I_1}) \subseteq f_{A_2}(\mathbf{1}_{I_2}) \tag{5.3}$$

with the shorthand that the "subset" relationship for tuples denotes each corresponding component satisfying the subset relationship, i.e. $[S_1, \ldots, S_n] \subseteq [S'_1, \ldots, S'_n]$ denotes $S_1 \subseteq S'_1 \wedge \cdots \wedge S_n \subseteq S'_n$.

The definitions of the abstract adaptation function, the abstract dependency function, and the method availability vector in section 5.1.1, along with the associativity and monotonicity rules proven in this section, provide a succinct way to mathematically express the chaining of lossy interface adapters using an abstract interpretation approach.

5.1.3 An example

As an example, we apply the abstract interpretation approach to the interfaces and adapters in figure 1.2. The abstract argument domains for each method in each interface must be determined manually based on a human-level understading of the interfaces, not only considering a natural division of arguments but also anticipating divisions relevant to potential interface adapters. Table 5.1 contains an example of how the abstract argument domains could be defined for figure 1.2.

Interface	Method	Abstract values
Video1	*playVideo*	MOV, AVI, MKV
	playAudio	MP3, OGG, WAV
Video2	*play*	INDEO, MP4, THEORA, DIVX
	stop	DUMMY
	skip	INTEGER
	caption	LANGUAGE
Video3	*play*	MOV, AVI, MKV, RM
	getVolume	DUMMY
	setVolume	INTEGER
	setEqualizer	EQSPEC
Audio	*play*	AU, WAV, OGG
	adjustAudio	VOLUME, EQUALIZER, MIXED

Table 5.1: Example abstract argument domains for figure 1.2.

For interface adapter *Video1toVideo2*, the *playAudio* method in *Video1* is not required to implement any of the methods, while *stop*, *skip*, and *caption* methods in *Video2* cannot be implemented using the methods of *Video1*. Let us say that *playVideo* of *Video1* can handle MOV files, which can contain video encoded in MP4, AVI files, which can contain video encoded in INDEO and DIVX, and MKV files, which can contain video encoded in MP4, DIVX, and THEORA. Then we would have an abstract dependency function with tuples as specified in table 5.2. The abstract dependency function has 16 elements, while the abstract adaptation function has $2^4 \times 2^4 = 256$ elements, which we do not show here.

From the adaptation dependency function in table 5.2, it is easy to see that with the interface adapter *Video1toVideo2*, an argument for *play* in *Video2* that corresponds to abstract value MP4 can be handled if *playVideo* in *Video1* can handle an argument that corresponds to abstract values MOV or MKV. If f

$$((\bot, \bot), (\{\bot\}, \{\bot\}, \{\bot\}, \{\bot\}))$$
$$((\bot, \text{MP3}), (\{\bot\}, \{\bot\}, \{\bot\}, \{\bot\}))$$
$$((\bot, \text{OGG}, (\{\bot\}, \{\bot\}, \{\bot\}, \{\bot\}))$$
$$((\bot, \text{WAV}), (\{\bot\}, \{\bot\}, \{\bot\}, \{\bot\}))$$
$$((\text{MOV}, \bot), (\{\bot, \text{MP4}\}, \{\bot\}, \{\bot\}, \{\bot\}))$$
$$((\text{MOV}, \text{MP3}), (\{\bot, \text{MP4}\}, \{\bot\}, \{\bot\}, \{\bot\}))$$
$$((\text{MOV}, \text{OGG}), (\{\bot, \text{MP4}\}, \{\bot\}, \{\bot\}, \{\bot\}))$$
$$((\text{MOV}, \text{WAV}), (\{\bot, \text{MP4}\}, \{\bot\}, \{\bot\}, \{\bot\}))$$
$$((\text{AVI}, \bot), (\{\bot, \text{INDEO}, \text{DIVX}\}, \{\bot\}, \{\bot\}, \{\bot\}))$$
$$((\text{AVI}, \text{MP3}), (\{\bot, \text{INDEO}, \text{DIVX}\}, \{\bot\}, \{\bot\}, \{\bot\}))$$
$$((\text{AVI}, \text{OGG}), (\{\bot, \text{INDEO}, \text{DIVX}\}, \{\bot\}, \{\bot\}, \{\bot\}))$$
$$((\text{AVI}, \text{WAV}), (\{\bot, \text{INDEO}, \text{DIVX}\}, \{\bot\}, \{\bot\}, \{\bot\}))$$
$$((\text{MKV}, \bot), (\{\bot, \text{MP4}, \text{DIVX}, \text{THEORA}\}, \{\bot\}, \{\bot\}, \{\bot\}))$$
$$((\text{MKV}, \text{MP3}), (\{\bot, \text{MP4}, \text{DIVX}, \text{THEORA}\}, \{\bot\}, \{\bot\}, \{\bot\}))$$
$$((\text{MKV}, \text{OGG}), (\{\bot, \text{MP4}, \text{DIVX}, \text{THEORA}\}, \{\bot\}, \{\bot\}, \{\bot\}))$$
$$((\text{MKV}, \text{WAV}), (\{\bot, \text{MP4}, \text{DIVX}, \text{THEORA}\}, \{\bot\}, \{\bot\}, \{\bot\}))$$

Table 5.2: Elements in an example abstract dependency function.

is the corresponding abstract adaptation function and the method availability vector is $p = [\{\perp, \text{MOV}, \text{MKV}\}, \{\perp, \text{MP3}\}]$, then

$$f(p) = [\{\perp, \text{MP4}, \text{DIVX}, \text{THEORA}\}, \{\perp\}, \{\perp\}, \{\perp\}]$$

which shows that with interface adaptation to *Video2*, *play* would be able to handle arguments corresponding to MP4, DIVX, and THEORA, while the *stop*, *skip*, and *caption* methods would not be available.

For the other interface adapters, the adapters *Video2toVideo3*, *Video3toAudio*, and *Video3toVideo1* have 40 elements in their abstract dependency functions and 2048 elements in their abstract adaptation functions, while the adapters *Video1toAudio* and *AudiotoVideo3* have 16 elements in their abstract dependency functions and 256 elements in their abstract adaptation functions.

5.2 Complexity

The abstract interpretation approach can easily encompass the discrete approach of chapter 3 by having each method accept only a single abstract value besides $\perp$. However, this does not mean that the optimal adapter chaining problem, where the number of accepted abstract values is maximized, is NP-complete. The reason is that the reduction of a method dependency matrix in the discrete approach to an abstract adaptation function in the abstract interpretation approach can require exponential time. In fact, simply storing the abstract adaptation function can require an exponential amount of memory.

In the discrete and probabilistic approaches of chapters 3 and 4, representing a method dependency matrix or a probabilistic adaptation factor requires $O(m^2)$ space, where m is the maximum number of methods in an interface. Applying the adaptation or composition operators would require $O(m^2)$ or $O(m^3)$ time. However, since the abstract interpretation approach requires that functions be represented as a set of tuples to retain generality, it requires

$O(d^m)$ space to represent an abstract dependency function and $O(2^{dm})$ space to represent an abstract adaptation function. Recall that an abstract argument domain must include $\perp$ in addition to a separate abstract value, so d is necessarily greater than or equal to 2, thus this is a truly exponential bound.

In fact, the exponential space complexity is a lower bound, not just an upper bound. With exactly m methods in a source interface and with d_1, d_2, ..., d_m abstract values in the abstract argument domains for each method, the number of tuples in the abstract dependency function is exactly $\prod_{i=1}^{m} d_i$, and the number of tuples in the abstract adaptation function is exactly $\prod_{i=1}^{m} 2^{d_i}$. We can see this in the example of section 5.1.3.

With complex interfaces that have many methods and non-trivial abstract argument domains, the exponential space complexity of the abstract interpretation approach makes it unlikely to be used in a system not dedicated to analyzing lossy interface chains which lacks the correspondingly exponential amount of memory. It is an open question whether real world interfaces will be trivial enough such that the abstract interpretation approach can be used more generally.

5.3 A greedy algorithm

While the exponential complexity of the abstract interpretation approach might make it unfeasible to obtain an optimal adapter chain on demand in a resource-constrained interactive system and virtually impossible to analyze complex interface adapters, it may be a slow but useful tool for software architecture analysis to derive an optimal adapter chain with simple interface adapters. So we describe an algorithm which can construct an optimal interface adapter chain in algorithm 7, which works thanks to the monotonicity property in equation (5.3).

Just as algorithm 1 can be extended to algorithm 3 to support behavior

Algorithm 7 Adapter chaining algorithm with abstract interpretation.

procedure PROB-GREEDY-CHAIN($G = (V, E)$, s, t)
  $C \leftarrow \{[]\}$ ▷ chains to extend
  $M = \emptyset$ ▷ discarded chains
  $D \leftarrow \{[] \mapsto \mathbf{I}_{\dim(\mathbf{1}_T)}\}$ ▷ method dependency matrixes
  while $C \neq \emptyset$ **do**
    $c \leftarrow$ element of C maximizing COUNT-ABSTRACT(c, D)
    if $c \neq [] \wedge \mathit{source}(c[1]) = s$ **then**
      return c
    else if no acyclic chain not in $C \cup M$ extends c **then**
      $C \leftarrow C - \{c\}$
      $M \leftarrow M \cup \{c\}$
    else
      if $c = []$ **then**
        $B \leftarrow \{[e] \mid e \in E, \mathit{target}(e) = t\}$
      else
        $B \leftarrow \{e : c \mid e \in E, \mathit{target}(e) = \mathit{source}(c[1])\}$
      end if
      remove cyclic chains from B
      $C \leftarrow C \cup B$
      $D \leftarrow D \cup \{e : c \mapsto D[c] \circ \mathit{depend}(e) \mid e : c \in B\}$
    end if
  end while
end procedure

Algorithm 8 Computing number of accepted abstract values.

function COUNT-ABSTRACT(c, D)

$s \leftarrow source(c[1])$

$v \leftarrow D[c](\mathbf{1}_s)$

return sum of count of abstract values in each component of v

end function

similar to service discovery by checking whether the current source is among a potential set of source interfaces instead of just checking against one, algorithm 7 can be similarly extended. Abstract values that represent arguments can also be similarly weighted to prioritize what needs to be adapted.

6. Web of interface adapters

The approaches described so far use only a single interface adapter to adapt a given target interface from a source interface, and interface adapters can be combined only as a singly-linked chain. These approaches force us to choose among imperfect chains of interface adapters, where one chain might be able to adapt certain methods in the source interface but cannot adapt other methods covered by another chain, and vice versa. However, using a *web of interface adapters*, a directed acyclic interface adapter graph where different adapters can be used to adapt different methods in an interface, can cover all methods that can possibly be adapted, incurring minimum loss.

6.1 Mathematical basics

The mathematical basics for this chapter extends the discrete approach described in section 3.1. As before, a method availability vector expresses the result of interface adaptation, while a method dependency matrix describes the adaptation behavior of a single interface adapter.

However, we will not use the method dependency matrix directly. Instead, we define a *method dependency set*, which is a set of method dependency matrixes with the same source interface and target interface. It is used to express the alternate paths through which a method may be adapted. The method dependency set for an actual interface adapter would be the singleton set with the method dependency matrix for the adapter, and we denote the method dependency set associated with an adapter A as $depend(A)$.

Given a method dependency set S and method availability vector p_i, the adaptation operator $\otimes$ for a method dependency set applied to a method

availability vector is defined as:

$$S \otimes p_i = \bigvee_{a_{ji} \in S} (a_{ji} \otimes p_i) \tag{6.1}$$

Basically, a method in the target interface is available if any of the alternate interface adapters can provide it from the methods in the source interface. We also follow the rules for a dummy method as in section 3.1.1.

Given two method dependency sets S and S', the composition operator $\otimes$ can be defined as:

$$S' \otimes S = \{b_{kj} \otimes a_{ji} \,|\, b_{kj} \in S' \wedge a_{ji} \in S\} \tag{6.2}$$

If two method dependency sets have the same source and target interfaces, then an equivalent method dependency set that merges them together is simply their union. We can define a merge operator $\oplus$, which is obviously commutative and associative:

$$S' \oplus S = S' \cup S \tag{6.3}$$

A similar monotonicity property to equation (3.4) also holds. If A_1 and A_2 are interface adapters, where A_1 converts I_1 to I_2 and A_2 converts I_2 to I_3, with $S = \mathit{depend}(A_1)$ and $S' = \mathit{depend}(A_2)$, then for $p_k = S' \otimes \mathbf{1}'_{I_2}$ and $p'_k = S' \otimes S \otimes \mathbf{1}'_{I_1}$, using equations (6.1) and (3.4):

$$p'_k \rightarrow p_k \tag{6.4}$$

6.2 Web of lossy adapters

Algorithm 9 can construct a web of interface adapters that can cover all possible methods in a target interface given a fully functional source interface, which we will refer to as a *maximally covering* web of interface adapters. It is based on unit propagation for Horn formulae [15], targeted towards building a web of interface adapters. It works in two phases, where it first computes all

methods in all interfaces that can be adapted given the source interface, and then extracts only the subgraph relevant for the target interface. Algorithms 10 and 11 are subalgorithms responsible for setup and subgraph extraction, respectively.

Algorithm 9 Constructing maximally covering web of interface adapters.

```
function MAXIMAL-COVER(G, s, t)
    (Q, D, S, M, C) ← COVER-SETUP(G, s)
    while Q is not empty do
        extract (I, i) from Q
        for (A = (I, I'), j) ∈ M[I][i] do
            if C[A][j] > 0 then
                C[A][j] ← C[A][j] − 1
                if C[A][j] = 0 then                    ▷ adaptation viable
                    D[I'][j] ← D[I'][j] ∪ {A}
                    if not S[I'][j] then
                        S[I'][j] ← true
                        insert (I', j) into Q          ▷ trigger new dependent
                    end if
                end if
            end if
        end for
    end while
    return (COVER-SUBGRAPH(D, t), D)
end function
```

Simply constructing a web of interface adapters is not the goal by itself, of course. The real goal is to use the interface adapters to adapt methods from a source interface into those of a target interface. Choosing which adapters should be used for which methods is more complex than in the case for a single chain, where there is no choice at all. Algorithm 12 is an abstract algorithm for

Algorithm 10 Setup for constructing maximal covering.

function COVER-SETUP($G = (V, E)$, s)
  $Q \leftarrow$ empty queue
  for $I \in V$ and method i of I **do**
    $D[I][i] \leftarrow \emptyset$ ▷ list of viable adapters
    $S[I][i] \leftarrow \mathit{false}$ ▷ whether satisfiable
    $M[I][i] \leftarrow \emptyset$
    for $A = (I, I') \in E$ **do**
      $M[I][i] \leftarrow M[I][i] \cup \{(A, j) \mid \mathit{depend}(A)_{ji}\}$ ▷ dependents
    end for
  end for
  for $A = (I_1, I_2) \in E$ and method j of I_2 **do**
    $C[A][j] \leftarrow |\{i \mid \mathit{depend}(A)_{ji}\}|$ ▷ unsatisfied dependency count
  end for
  for each method i of s **do** ▷ start with source interface
    $S[s][i] = \mathit{true}$
    insert (s, i) into Q
  end for
  return (Q, D, S, M, C)
end function

Algorithm 11 Extract subgraph comprising web of interface adapters.

```
function COVER-SUBGRAPH(D, t)
    V' ← ∅, E' ← ∅
    Q ← empty queue, Q' ← ∅
    for method i of t do
        insert (t, i) in Q and Q'
    end for
    while Q is not empty do
        extract (I', j) from Q
        V' ← V' ∪ {I'}
        E' ← E' ∪ D[I'][j]
        for A = (I, I') ∈ D[I'][j] do
            for i such that depend(A)_ji do
                if (I, i) ∉ Q' then
                    insert (I, i) into Q and Q'
                end if
            end for
        end for
    end while
    return (V', E')
end function
```

determining which interface adapters should be invoked when adapting each method. It needs more information than just the web of interface adapters, which is provided by the value D also returned in algorithm 9.

The interface adapters used to adapt a given method are specified by algorithm 12; the concrete steps involved in actually adapting a method are left to how interface adaptation is actually done, whether it be direct invocation by the interface adapter, call substitution after constructing the call graph, or composition of interface adapters specified in a high-level language. The exact criterion for selecting an adapter in algorithm 12 also does not affect the correctness of the algorithm.

Algorithm 12 Adapting a specific method in the target interface.

procedure ADAPT-METHOD(D, s, t, m)
  if $D[t][m] = \emptyset$ **then**
    adaptation not possible
  else
    RECURSIVE-ADAPT(D, s, t, m, $\{t\}$)
  end if
end procedure
procedure RECURSIVE-ADAPT(D, s, I, m, V)
  if $I = s$ **then**
    return
  end if
  select $A = (I', I) \in D[I][m]$ where $I' \notin V$
  for method i in I' where $depend(A)_{mi}$ **do**
    RECURSIVE-ADAPT(D, s, I', i, $V \cup \{I'\}$)
  end for
  adapt method m in interface I using A
end procedure

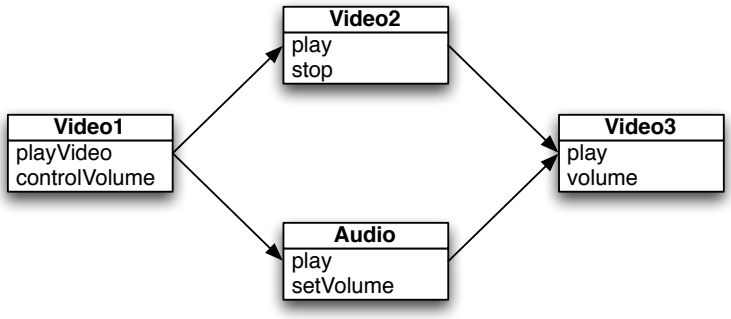

Figure 6.1: An example interface adapter graph.

Algorithm 9 constructs a maximally covering web of adapters, but it completely ignores the number of interface adapters it incorporates in the web. It could end up constructing a web with hundreds of interface adapters when less than a dozen would do. However, trying to minimize the number of incorporated interface adapters is an NP-complete problem as we will show in section 6.4. Invoking the minimum number of interface adapters while actually adapting a method also turns out to be NP-complete.

6.3 An example

As an example, we apply the web of adapters approach to the interface adapter graph in figure 6.1. *Video1*, *Video2*, and *Video3* are interfaces capable of playing back video, while *Audio* is an interface capable of playing back only audio. *Video1*, *Video3*, and *Audio* support volume control, but *Video2* does not. The interface adapter from *Video1* to *Audio* does not even attempt to adapt the playback method.

Let us say that we have a fully functional service conforming to interface *Video1*, and we wish to access it through interface *Video3*. With a singly-linked chain approach, there can only be an interface adapter chain from *Video1* to *Video3* either via *Video2* or via *Audio*. If the chain goes through *Video2*, then

the *play* method in *Video3* can be provided but not the *volume* method, while if the chain goes through *Audio*, then the *volume* method can be provided but not the *play* method.

With the web of adapters approach, on the other hand, *both* methods in *Video3* can be provided by using both chains of interface adapters. The *play* method of *Video3* can be adapted from *play* of *Video2* which in turn is adapted from *playVideo* of *Video1*, and the *volume* method of *Video3* can be adapted from *setVolume* of *Audio* which in turn is adapted from *controlVolume* of *Video1*. It should be noted that if the interface adapters store state, then using different adapters for different methods might cause problems, although this should not be an issue for the example of figure 6.1.

If a_{ji}, b_{kj}, c_{ji}, d_{kj} are the method dependency matrixes for the interface adapters from *Video1* to *Video2*, from *Video2* to *Video3*, from *Video1* to *Audio*, and from *Audio* to *Video3*, respectively, then they would be:

$$a_{ji} = \begin{pmatrix} t & f & f \\ f & t & f \\ t & f & f \end{pmatrix}$$

$$b_{kj} = \begin{pmatrix} t & f & f \\ f & t & f \\ t & f & f \end{pmatrix}$$

$$c_{ji} = \begin{pmatrix} t & f & f \\ t & f & f \\ f & f & t \end{pmatrix}$$

$$d_{kj} = \begin{pmatrix} t & f & f \\ t & f & f \\ f & f & t \end{pmatrix}$$

and the respective method dependency sets A, B, C, and D would be $\{a_{ji}\}$, $\{b_{kj}\}$, $\{c_{ji}\}$, and $\{d_{kj}\}$. We can then compute the loss from the web of adapters

approach as applied to figure 6.1:

$$((B \otimes A) \oplus (D \otimes C)) \otimes \mathbf{1}'_{Video1} = [f, t, t]$$

which agrees with our reasoning that all actual methods in *Video3* can be adapted from *Video1* using the web of adapters approach.

6.4 Minimizing number of adapters

While algorithm 9 can construct a maximally covering web of interface adapters in polynomial time ($O(m^2)$ being a loose time bound with a straightforward implementation, where m is the total number of methods), it is unlikely there will be a polynomial-time algorithm for finding a maximally covering web of adapters with the minimum number of interface adapters. This is because the problem is NP-complete, which we will prove with a reduction from one-in-three 3SAT [24].

We formally define MINWEB as the problem of whether there is a web of interface adapters in an interface adapter graph from a given source interface to a given target interface such that it is maximally covering and has at most K interface adapters. Given a candidate boolean expression for one-in-three 3SAT with c clauses and v variables, we will reduce it to a candidate interface adapter graph for MINWEB such that the boolean expression is an instance of one-in-three 3SAT if and only if there is a maximally covering web of interface adapters with at most $v + 2c$ adapters.

For each variable, we create an interface with methods corresponding to all the literals, two for each variable. For each clause, we create an interface with only a single method. We also separately create a source interface with methods corresponding to the possible literals and a target interface with methods corresponding to the clauses.

Starting from the source interface, we connect the interfaces corresponding to variables serially. Between each of these interfaces, we define two adapters,

one which makes the method corresponding the successor variable true and the other which makes it false, by making the method corresponding to the positive literal available and the method corresponding to the negative literal unavailable in one adapter and the opposite in the other adapter. Other literals are left alone.

From the sink node of the variable handling subgraph, we create three adapters to each of the interfaces corresponding to the clauses. Each adapter corresponds to a literal in the clause, and the sole method in the interface is available only if the method corresponding to the literal is available. And from each interface corresponding to a clause, there is a single adapter to the target interface for the entire graph which makes the method corresponding to the clause available only if the sole method in the clause interface is available.

For the graph constructed this way, the entire graph is obviously maximally covering with $2v+4c$ adapters and all methods available at the target interface. If the original boolean expression is an instance of one-in-three 3SAT, then a satisfying assignment can specify a singly-linked path through the variable interfaces, followed by each true literal specifying the adapters to pass through to each clause interface, followed by the adapters to the target interface, and the resulting directed acyclic graph is a maximally covering web of adapters with $v+2c$ adapters, since all methods will be available at the target interface.

Conversely, if there is a maximally covering web of adapters with $v + 2c$ adapters, then $2c$ adapters connect to the clause interfaces since all clause interfaces must be included. The remaining v adapters must be a singly-linked path through the variable interfaces, and the selection of adapters for each variable interface specifies a variable assignment which satisfies the original boolean expression with only one true literal in each clause. Therefore MIN-WEB is NP-complete, and we can also conclude that minimizing the number of required adapters to adapt a single method is also NP-complete by removing the other methods in the target interface.

7. Discussion

We have described various approaches to analyzing the loss in interface adapter chaining. We first began with a discrete approach in chapter 3 where method dependencies are simply boolean. We then moved on to probabilistic and abstract interpretation approaches in chapter 4 and chapter 5, respectively, where method dependencies are no longer boolean but partial. Another direction was taken in chapter 6, where instead of just chaining interface adapters we allowed them to form directed acyclic graphs. Each framework has their strengths and weaknesses in terms of simplicity, performance, precision, flexibility, etc., making them suitable for different application domains.

While the discrete approach is much more precise than naively assigning a cost to an interface adapter, it considers a method as an indivisible unit and does not take into account methods that may only be partially adapted, i.e. a method which cannot accept all valid arguments. However, it is much more simpler to infer the necessary method dependencies. In fact, this is akin to what is required in many dependency analysis problems [17, 23, 44, 63], and code analysis techniques can be used to automatically extract the required method dependencies [7, 62].

The discrete approach only allows for singly linked chains of interface adapters. While this would often not be able to minimize loss as much as the web of adapters approach would be able to, the fact that only a single interface adapter is used to adapt a source interface to a subsequent target interface makes it much simpler to use interface adapters which maintain state. Interface adapters which must adapt the protocol and not just the method signatures require state, so it is easier to use the discrete approach along with approaches where the interface adapter adjusts protocol behavior [16, 37, 52, 54, 77].

Experimental results also suggest that the discrete approach is reasonably fast enough such that it can be used for adapting interfaces on demand [42]. This all suggests that the discrete approach is best applicable when most methods are perfectly adapted if they are adapted at all, when nothing more than code analysis is desired for extracting method dependencies, when interface adapters must maintain state to also adapt behavior, and a reasonably fast response for interface adaptation is required.

The probabilistic approach is more precise than the discrete approach, being able to model partially adapted methods. It also only allows singly linked chains of interface adapters, so it should be able to incorporate interface adapters with state with little problem. The general structure is similar to the discrete approach, having similar computational complexity, so it should generally be slower than the discrete approach by only a constant factor.

However, the probabilistic approach assumes the independence assumptions in section 4.1.1 hold, and it is an open question how closely the inferred probabilities would fit the actual behavior of non-trivial interfaces and interface adapters in the real world. Obtaining the necessary probabilities is also not as simple as getting the method dependencies in the discrete approach. It is likely that some sort of time-consuming program tracing [2, 36, 39, 75] would be required to obtain the probabilities.

This would suggest that the probabilistic approach would be best suitable when partial method adaptation occurs such that the independence assumptions of section 4.1.1 are reasonably valid, when interface adapters must maintain state to also adapt behavior, and a reasonably fast response for interface adaptation is required. It has the drawback that quite a bit of effort would be required to obtain the necessary probabilities, however.

The abstract interpretation approach can be the most precise, not having to rely on questionable assumptions as with the probabilistic approach. However, it does rely on abstract argument domains being properly defined

for every method in every interface. An improperly set up abstract argument domain would require a great deal of effort to update related data structures appropriately, and sometimes it might not be easy to define an abstract argument domain which would work well with every conceivable interface and interface adapter.

A much more serious problem with the abstract interpretation approach is its complexity. It has exponential space complexity, which means a prohibitive amount of memory may be required. Even worse, it requires exponential effort to set up the necessary values, which for even moderately complex interface adapters might be infeasible to do automatically by computer, much less by a human developer. And the necessary functions have to be constructed by a human developer unless sophisticated program analyses can be developed that could automatically define abstract argument domains and infer how interface adapter code will adapt them.

The exponential complexity of the abstract interpretation approach suggests that it should be used for offline analysis of simple interface adapter chains. The better precision may be useful when determining if a set of interface adapters shipped with a deployment of a ubiquitous computing environment can satisfactorily support seamless operation. However, the abstract interpretation approach may still be practical as a subsystem if the number of methods and the size of abstract argument domains are small: it is an open question of whether they would be small enough in real world systems.

The web of adapters approach has the same precision as the discrete approach, but the adaptation loss can be reduced to the absolute minimum possible. By using alternate interface adapters as needed, this minimum can be even smaller than the minimum loss that can be achieved in the discrete approach. However, allowing alternate interface adapters also makes it difficult to use interface adapters which maintain state. It also makes the implementation of interface adapters more complex, since they must be able to consider

Framework	Setup	Precision	Space	Statefulness
Discrete	simple	boolean	polynomial	compatible
Probabilistic	moderate	fuzzy	polynomial	compatible
Abstract	complex	sharp	exponential	compatible
Web	simple	boolean	polynomial	stateless

Table 7.1: Comparison of mathematical frameworks.

alternates when other interface adapters must be invoked.

The algorithms for the web of adapters approach should be reasonably fast. This suggests that it is best suited when all interface adapters are stateless in a system where on demand interface adaptation is required. However, the increased complexity required in the implementation of interface adapters may make the approach unattractive for an interface adaptation system.

Table 7.1 organizes the strengths and weaknesses of each approach discussed above in a succinct manner.

7.1 A case study

While we have shown examples loosely based on figure 1.2 for each of the mathematical frameworks in their corresponding chapters, in this section we will describe at a high level how the frameworks can be applied to a set of interfaces based on actual web services. Figure 7.1 shows six interfaces for actual payment processing web services, along with eight hypothetical interface adapters represented as arrows from source interfaces to target interfaces. We will discuss the situation where an application only knows how to use the interface for PayPal, but the only payment processing service actually available conforms to the interface for Moneybookers.

For the interfaces of figure 7.1, XWebCheckOut and Google Checkout are not able to support recurring payments, while the interfaces can. Among the

Figure 7.1: Example interface adapter graph with payment interfaces.

```
class MoneybookersToXWebCheckOutAdapter extends XWebCheckOut {
   private Moneybookers source;

   ...

   public void PROCESSPAYMENTLSPG(Order order) {
       source.AuthorizePayment(convertOrder(order));
       source.ExecuteTransfer(convertOrder(order));
   }
}
```

Figure 7.2: Example code snippet for interface adapter.

remaining interfaces, CheddarGetter is *only* capable of supporting recurring payments and cannot support one-time payments. PayPal, Moneybookers, and Amazon FPS are all capable of supporting both one-time payments and recurring payments. Amazon FPS also supports multi-use payment tokens, which is something that can be purchased once and used multiple times for getting multiple products at different times. It should be apparent that perfect interface adaptation is not possible among these interfaces.

With the exception of the web of adapters approach, a Java code snippet of the interface adapter from Moneybookers to XWebCheckOut could look something like what is shown in figure 7.2. In the code snippet, the interface adapter uses the AuthorizePayment and ExecuteTransfer methods of Moneybookers to implement the PROCESSPAYMENTLSPG method of XWebCheckOut.

We will discuss how the different approaches to analyzing lossy interface adapter chaining can be applied to this example, which should illustrate their differences.

Discrete approach

As we discussed for the discrete approach, it is very easy to infer the method dependencies simply by inspecting the code for an interface adapter. For example, we can immediately see that providing the PROCESSPAYMENT method of XWebCheckOut depends on the AuthorizePayment and ExecuteTransfer methods being available in the code snippet of figure 7.2. In fact, this process can be completely automated by analyzing source or binary code for the interface adapter, so deriving the method dependency matrix for an interface adapter requires very little effort from a developer.[1]

Once the method dependency matrixes are derived, we can apply section 3.1 to analyze how well an interface adapter chain can adapt the interface for Moneybookers to PayPal. For instance, the interface adapter chain that goes through Moneybookers, XWebCheckOut, Google Checkout, and PayPal could provide the 5 methods DoAuthorization, DoDirectPayment, DoExpressCheckoutPayment, DoNonReferencedCredit, and DoReferenceTransaction. The interface adapter chain through Moneybookers, Amazon FPS, CheddarGetter, and PayPal could provide the 3 methods CreateRecurringPaymentsProfile, GetRecurringPaymentsProfileDetails, and UpdateRecurringPaymentsProfile, as shown in figure 7.3.

Using algorithm 1 of section 4.3 can give us the optimal interface adapter chain that maximizes the number of methods that can be provided in the interface for PayPal from the interface for Moneybookers. In this example, the optimal interface adapter chain could go through Moneybookers, Amazon FPS, Google Checkout, and PayPal, which can provide the 7 methods DoAuthorization, DoDirectPayment, DoExpressCheckoutPayment, DoNonReferencedCredit, DoReferenceTransaction, DoVoid, and RefundTransaction.

[1] Things become hard again when reflection [19] is used, but this could be dealt with by conservatively assuming that all methods in the source interface are necessary on the rare occasion reflection is used by an interface adapter.

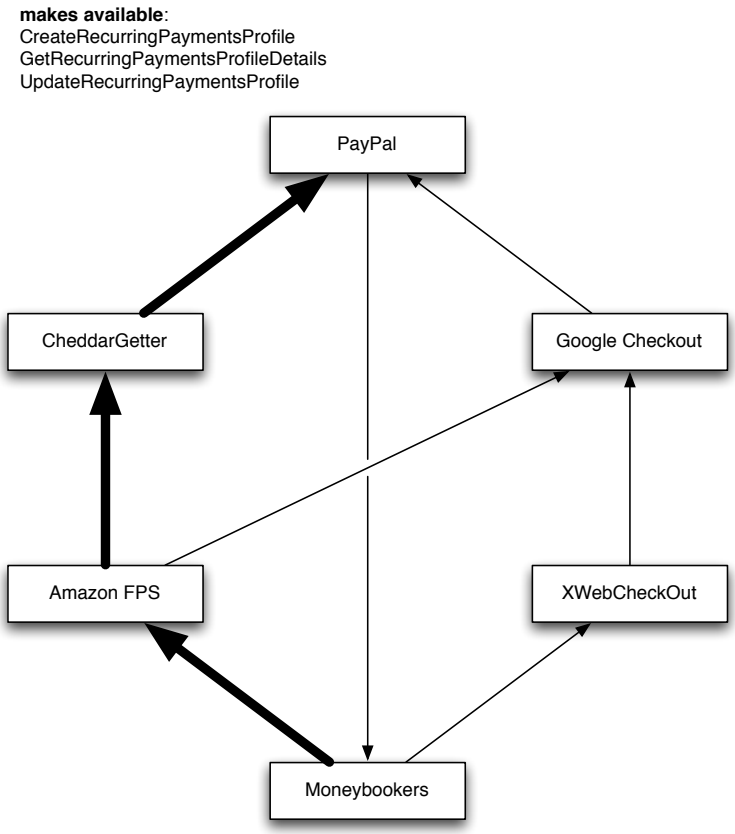

Figure 7.3: Example of loss analysis with discrete approach.

Probabilistic approach

Compared to the discrete approach, it is significantly harder to fill in the probabilities that are required by the conversion probability matrixes of the interface adapters. Either a developer must estimate all of the probabilities, e.g. estimating the probability to be 0.66 when converting an argument for Pay to AuthorizePayment because Amazon FPS supports one-time, recurring, and multi-use payments while Moneybookers only supports one-time and recurring payments, or some sort of program tracing should be done with a large random sample of arguments to measure the probability that the interface adapter converts an argument properly. The former requires significant developer effort, while the latter is very time-consuming: either way, they are harder than what the discrete approach would require.

But once the conversion probability matrixes are obtained, the probabilistic approach can infer results about partial adaptation of methods, something that the discrete approach is entirely unable to do. For instance, applying section 4.1 to the interface adapter chain through Moneybookers, Amazon FPS, CheddarGetter, and PayPal could show that the 3 methods CreateRecurringPaymentsProfile, GetRecurringPaymentsProfileDetails, and UpdateRecurringPaymentsProfile can run properly with probability 0.95, 1.0, and 1.0, respectively as shown in figure 7.4, because of slight incompatibilities in the creation of recurring payments. In contrast, the discrete approach would not be able to handle such slight incompatibilities at all, making the probabilistic approach more precise. The greedy algorithm of section 5.3 can be used to maximize the probability that a method of PayPal runs properly, i.e. minimizing the probability that an error occurs when a method is invoked on some argument.

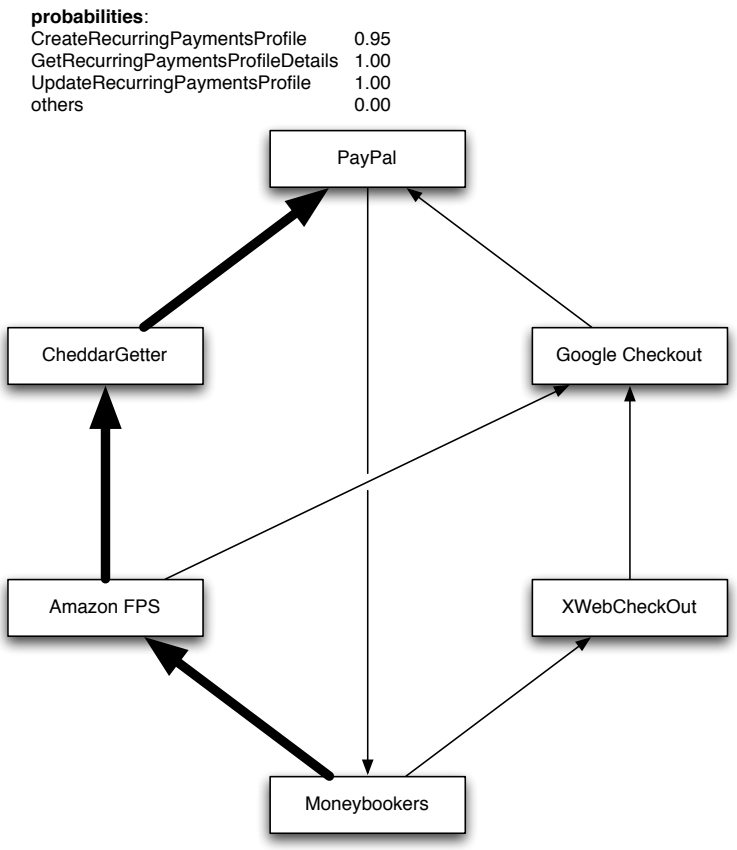

Figure 7.4: Example of loss analysis with probabilistic approach.

Abstract interpretation approach

Defining the abstract argument domains for every interface must be done by a developer and cannot be automated. For instance, it is beyond the capabilities of current computers to understand that arguments to the Pay method of Amazon FPS can be generally divided into three types of arguments, corresponding to one-time, recurring, and multi-use payments. This must be done by a human developer who can understand this from the specification and implementation of the interface.

With such understanding, the developer could infer that the CancelToken can accept arguments corresponding to recurring and multi-use payments, which could be represented with the abstract values RECUR and MULTIUSE, that the Pay method can accept arguments corresponding to one-time, recurring, and multi-use payments, which could be represented with the abstract values ONCE, RECUR, and MULTIUSE, while the other methods require only one abstract value besides $\perp$.

However, preparing the abstract argument domains is trivial compared to the preparation of the abstract dependency function. For the interface adapter from Amazon FPS to CheddarGetter, 768 entries must be filled manually for the abstract dependency function. It cannot be constructed automatically since this requires a human level understanding of the interface. This is an extremely onerous task for a developer, but other interface adapters are even worse. For the interface adapter from PayPal to Moneybookers, over 16 million entries must be filled manually, which is completely infeasible. Even if the abstract dependency function can be created, the abstract adaptation function would require 281,474,976,710,656 entries: even with a very optimistic estimate of one byte per entry, this is still more than two hundred terabytes.[2]

[2]If there were an interface adapter from OpenGL to DirectX, the abstract dependency function would require more entries than there are atoms in the Milky Way Galaxy; for the abstract adaptation function, more entries than there are atoms in the observable universe.

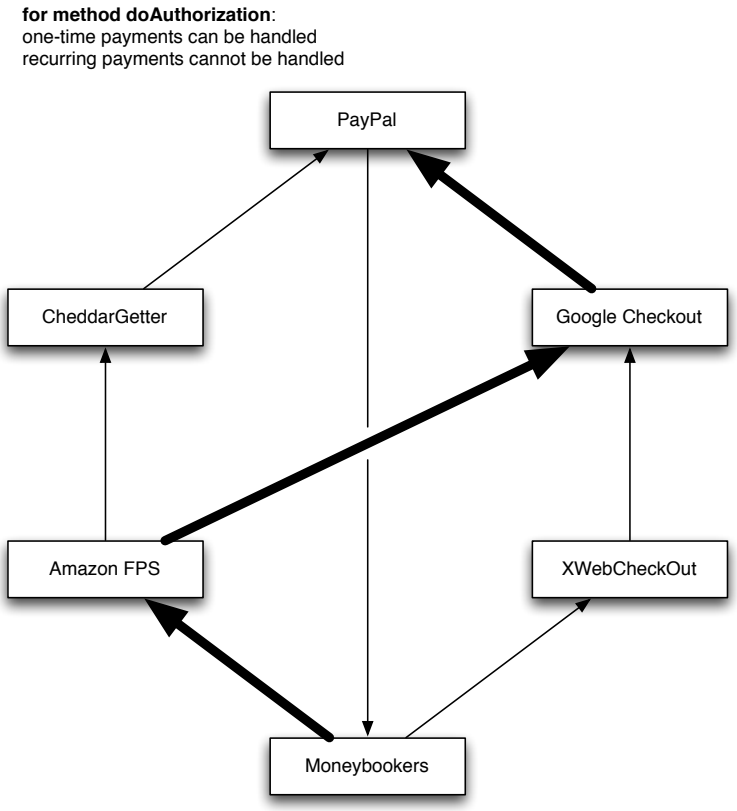

Figure 7.5: Example of loss analysis with abstract interpretation approach.

If the abstract adaptation functions could somehow be constructed despite these enormous resource requirements, then they might be able to infer precise results such as the doAuthorization method in PayPal being able to handle arguments corresponding to one-time payments but not arguments corresponding to recurring payments when an interface adapter chain through Moneybookers, Amazon FPS, Google Checkout, and PayPal is used as shown in figure 7.5. Given its precision, the abstract interpretation approach can be useful when interfaces and interface adapters are as simple as the in example in section 5.1.3, but the approach is infeasible when they are even as moderately complex as the ones in this case study.

Web of adapters approach

With the web of adapters approach, even implementing interface adapters cannot be as straightforward as is done in figure 7.2. This is because the adapters used to recursively adapt methods such as AuthorizePayment and ExecuteTransfer must be selected based on information returned by algorithm 9, so interface adapter implementation must be done quite differently. However, this is not in any way an insurmountable problem, although the specific details of how this problem is handled would strongly depend on the general approach to implementing interface adapters. On the other hand, the web of adapters approach has the same precision as the discrete approach, so preparing the method dependency sets should be just as easy.

Using the web of adapters approach, algorithm 9 constructs a maximally covering web of adapters that includes all interface adapters except for the one from PayPal to Moneybookers as shown in figure 7.6. In contrast to the discrete approach, where we must sacrifice providing either the method DoAuthorization or the method CreateRecurringPaymentsProfile, the web of adapters approach allows us to provide *both* methods, and in fact it could provide the 10 methods DoAuthorization, DoDirectPayment, DoExpressCheckoutPayment, DoNonReferencedCredit, DoReferenceTransaction, DoVoid, RefundTransaction, CreateRecurringPaymentsProfile, GetRecurringPaymentsProfileDetails, and finally UpdateRecurringPaymentsProfile. This is less loss than is possible with the discrete approach.

However, if any of the interface adapters must maintain state to function properly, then the web of adapters approach should not be used. Otherwise, the use of different adapters for different methods can result in unexpected behavior when a method depends on state being properly maintained by another method, the latter which could be adapted by another adapter.

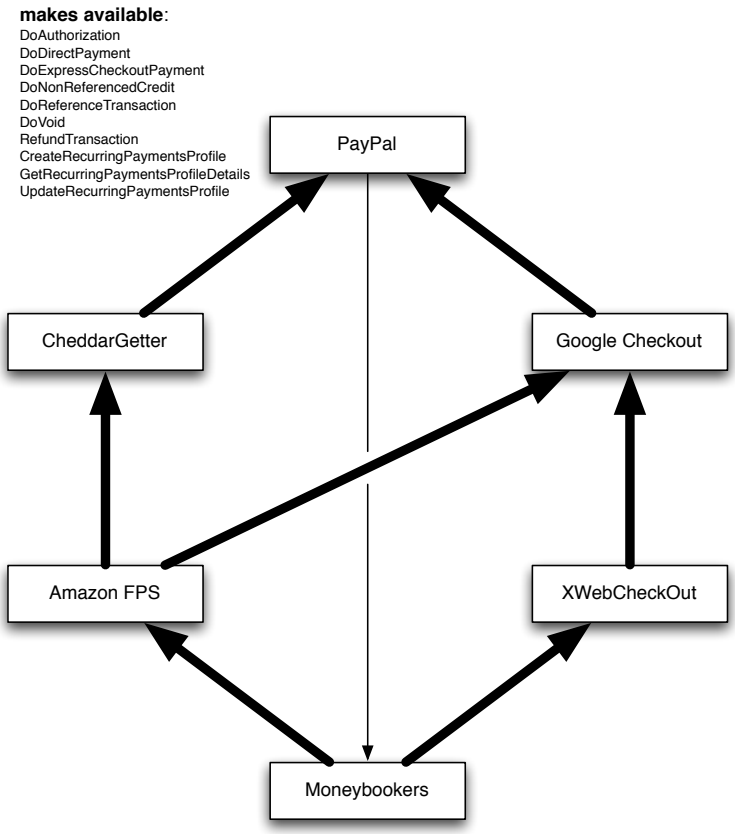

Figure 7.6: Web of interface adapters for figure 7.1.

8. Conclusions

As ubiquitous computing environments become more widespread, a myriad of different interfaces will be defined for the diverse set of services that will be developed. Standardization will be unable to keep up with the rapid proliferation of services, however, so different interfaces will be specified for similar services. It is not reasonable to expect that software for ubiquitous computing environment will be able to handle all possible interfaces that may be required to use a service which serves its demands, not to mention interfaces that are not even in existence yet, so some form of interface adaptation is required.

Interface adaptation can be achieved by using interface adapters, where an intermediate entity can transform an unknown interface to a known one. This approach allows interfaces to be adapted without rewriting software, which is an option not available when adaptation must occur dynamically. Creating an interface adapter requires human effort, however, and the quadratic amount of effort to create all interface adapters necessary for directly transforming between interfaces makes interface chaining an attractive option.

Unfortunately, it will often be the case that interface adaptation cannot be done perfectly. This is because interfaces are rarely written with compatibility with other interfaces in mind. Adaptation loss is to be expected, and even more so when interface adapters are chained, so interface adaptation should consider the loss incurred. This is especially so in ubiquitous computing environments, where it is infeasible to generate code on demand that might possibly fill in the imperfections or when there is are fundamental incompatibilities due to limitations in the service.

We have described several mathematical frameworks that can analyze such adaptation loss incurred by interface adapter chaining. They define a way

to represent the adaptation loss using method availability vectors, a way to represent the adaptation behavior of interface adapters, and the mathematical operations necessary for analyzing loss. These are then used to construct algorithms or prove the computational complexity for relevant problems such as constructing an optimal adapter chain which incurs the minimum loss.

The various mathematical frameworks differ in their precision and graph structure. The discrete, probabilistic, and abstract interpretation approaches allow for singly-linked chains of interface adapters and differ by their precision, with the probabilistic and abstract interpretation approaches being built upon the discrete approach. The web of adapters approach expands the discrete approach, allowing interface adapters to form a directed acyclic graph, which allows more methods to be adapted than can be accomplished by the discrete approach.

The various approaches have their separate strengths and weaknesses, however, which make them suitable for different application domains. The discrete approach is relatively simple to use and has a relatively fast algorithm for constructing optimal interface adapter chains, which makes it suitable for interface adaptation systems that require quick responses. The probabilistic approach would be useful if the discrete approach is not precise enough, while the web of adapters approach can be used when interface adapters are stateless. The increased precision of the abstract interpretation approach comes at the cost of exponential complexity, which would make it more suitable for offline analysis of interface adaptation.

Our work in creating mathematical frameworks with which lossy interface adapter chaining can be analyzed will be a rigorous and sound foundation upon which interface adaptation systems can be built, where lossy interface adapters are properly handled so that adaptation loss can be minimized.

Directions for future research

This dissertation has focused on the mathematics of analyzing lossy interface adapter chaining. However, there are several issues when trying to apply any of the proposed mathematical frameworks to an actual implementation of an interface adaptation system.

One is the derivation of dependencies from source code or binary software modules, which would be required to create method dependency matrixes or conversion probability matrixes. While it is an easy problem with the discrete approach when the most straightforward method of implementing interface adapters is used, it may become an issue if more complex approaches to interface adapter implementation are required. It is a much more significant issue with the probabilistic approach: while testing an interface adapter with a large random sample of arguments should give the required probabilities, the details of exactly how this can be done still needs to be worked out.

Another issue when applying the mathematics to an actual implementation is the optimization of data structures and operations. Having focused on an elegant formulation of the lossy interface adapter chaining problem, not so much attention has been paid to making the data structures as efficient as possible. In particular, actual interface adapters will likely require only a small number of methods in a source interface for implementing a specific method in a target interface, which suggests that a sparse matrix representation can be much more efficient in practice than a naive implementation of the mathematical structures in this dissertation.

One last issue relevant to implementation is the efficacy of the probabilistic approach. It depends on three independence assumptions, so it is an open question whether the probabilistic approach can give reasonable results for real systems.

There are also potential areas of further research on the purely theoretical side. Related to the question about the efficacy of the probabilistic approach,

there may be an alternate probabilistic formulation that can also be feasibly applied in practice, but requires weaker and more realistic assumptions about the behavior of interfaces and interface adapters. Extending the web of adapters approach so that it can incorporate interface adapters which maintain state is also an avenue of further research.

The abstract interpretation approach uses a set-theoretic approach to representing functions in order to be as general as possible, but this is the cause of its exponential space complexity. Investigating the existence of special-case function representations for abstract dependency functions which could represent almost all realistic cases would be a worthwhile endeavor for reducing the exponential space complexity of the abstract interpretation approach.

Finally, for the problems we have shown to be NP-complete, it would be of interest if there were polynomial-time approximation algorithms for constructing optimal interface adapter chains or for minimizing the number of adapters required in a maximally covering web of interface adapters.

요 약 문

손실이 있는 인터페이스 어댑터 체인의 수학적 분석

유비쿼터스 컴퓨팅 환경에서는 물리적 환경에 존재하는 여러 가지 객체에 컴퓨팅 서비스가 내장되어 서로 문제없이 연계하여 동작될 것이다. 다양한 종류의 물리적 객체에 컴퓨팅 서비스가 내장될 수 있기 때문에 기능적으로 비슷한 서비스들을 이용하기 위한 다양한 인터페이스들이 만들어질 것이다. 하지만 유비쿼터스 컴퓨팅 환경에서는 인터페이스가 다르더라도 서비스를 이용할 수 있어야 문제없이 서비스들이 연계되어 동작할 수 있다.

이 문제는 인터페이스를 필요에 따라 변환할 수 있는 인터페이스 어댑터를 이용하여 해결할 수 있다. 여러 개의 인터페이스 어댑터를 연결하면 직접 변환을 위한 모든 인터페이스 어댑터를 개발자가 개발해야 하는 과중된 노력을 피할 수 있다. 하지만 인터페이스 어댑터가 완벽하게 인터페이스를 변환하지 못할 경우가 많을 것이며, 인터페이스 어댑터들을 체인으로 연결해 사용할 경우는 이 문제가 더욱 중요해질 것이다. 인터페이스 어댑터 체인을 구성할 때 변환 손실을 제대로 고려하기 위해서는 이러한 체인의 손실을 분석할 수 있는 수학적 바탕이 필요하다.

이 논문에서는 인터페이스 어댑터 체인의 손실을 분석할 수 있는 수학적 바탕을 제시한다. 인터페이스 어댑터 체인의 손실 표현 및 각각 인터페이스 어댑터가 손실에 끼치는 영향을 유추하기 위한 수학적 객체 및 연산을 정의한다. 이를 이용하여 손실이 있는 인터페이스 어댑터 체인과 관련된 문제를 위한 알고리즘의 작성 및 복잡도 증명을 한다.

References

[1] Ken Arnold, editor. *The Jini Specifications*. Addison-Wesley, 2nd edition, December 2000. ISBN 978-0-201-72617-6.

[2] Thomas Ball and James R. Larus. Optimally profiling and tracing programs. *ACM Transactions on Programming Languages and Systems*, 16(4):1319–1360, July 1994. doi: 10.1145/183432.183527. URL http://doi.acm.org/10.1145/183432.183527.

[3] Roberto Barbuti, Roberto Giacobazzi, and Giorgio Levi. A general framework for semantics-based bottom-up abstract interpretation of logic programs. *ACM Transactions on Programming Languages and Systems*, 15 (1):133–181, January 1993. doi: 10.1145/151646.151650. URL http://doi.acm.org/10.1145/151646.151650.

[4] Boualem Benatallah, Fabio Casati, Daniela Grigori, Hamid R. Motahari Nezhad, and Farouk Toumani. Developing adapters for web services integration. In *Proceedings of the 17th International Conference on Advanced Information Systems Engineering*, volume 3520 of *Lecture Notes in Computer Science*, pages 415–429, Porto, Portugal, June 2005. Springer-Verlag. ISBN 978-3-540-26095-0. doi: 10.1007/11431855_29.

[5] J. Bosch. Superimposition: a component adaptation technique. *Information and Software Technology*, 41(5):257–273, March 1999. doi: 10.1016/S0950-5849(99)00007-5.

[6] Andrea Bracciali, Antonio Brogi, and Carlos Canal. A formal approach to component adaptation. *Journal of Systems and Software*, 74(1):45–54, January 2005. doi: 10.1016/j.jss.2003.05.007.

[7] A. Brown, G. Kar, and A. Keller. An active approach to characterizing dynamic dependencies for problem determination in a distributed environment. In *Proceedings of the 2001 IEEE/IFIP International Symposium on Integrated Network Management*, pages 377–390, May 2001. ISBN 0-7803-6719-7. doi: 10.1109/INM.2001.918054.

[8] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, pages 151–158. ACM Press, 1971. doi: 10.1145/800157.805047.

[9] Patrick Cousot. Program analysis: The abstract interpretation perspective. *ACM Computing Surveys*, 28(4es), December 1996. doi: 10.1145/242224.242433. URL http://doi.acm.org/10.1145/242224.242433.

[10] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 238–252. ACM Press, January 1977. doi: 10.1145/512950.512973. URL http://doi.acm.org/10.1145/512950.512973.

[11] M. Crampin and F. A. E. Pirani. *Applicable Differential Geometry*, chapter 0, pages 5–7. Number 59 in London Mathematical Society Lecture Note Series. Cambridge University Press, March 1987. doi: 10.2277/0521231906.

[12] Dennis Dams, Rob Gerth, and Orna Grumberg. Abstract interpretation of reactive systems. *ACM Transactions on Programming Languages and Systems*, 19(2):253–291, March 1997. doi: 10.1145/244795.244800. URL http://doi.acm.org/10.1145/244795.244800.

[13] Rogério de Lemos, Cristina Gacek, and Alexander Romanovsky. Architectural mismatch tolerance. In Rogério de Lemos, Cristina Gacek, and

Alexander Romanovsky, editors, *Architecting Dependable Systems*, volume 2677 of *Lecture Notes in Computer Science*, pages 175–194. Springer-Verlag, July 2003. ISBN 978-3-540-40727-0. doi: 10.1007/3-540-45177-3_8. URL http://www.springerlink.com/content/x312284r52h5342w.

[14] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, June 1959. doi: 10.1007/BF01386390.

[15] William F. Dowling and Jean H. Gallier. Linear-time algorithms for testing the satisfiability of propositional Horn formulae. *Journal of Logic Programming*, 1(3):267–284, October 1984.

[16] Marlon Dumas, Murray Spork, and Kenneth Wang. Adapt or perish: Algebra and visual notation for service interface adaptation. In *Proceedings of the 4th International Conference on Business Process Management*, pages 65–80. Springer-Verlag, September 2006. ISBN 978-3-540-38901-9. doi: 10.1007/11841760_6.

[17] Stuart I. Feldman. Make — a program for maintaining computer programs. *Software Practice and Experience*, 9(4):255–265, April 1979. doi: 10.1002/spe.4380090402. URL http://www3.interscience.wiley.com/journal/113444488/abstract.

[18] Robert W. Floyd. Algorithm 97: Shortest path. *Communications of the ACM*, 5(6):345, June 1962. doi: 10.1145/367766.368168. URL http://doi.acm.org/10.1145/367766.368168.

[19] Ira R. Forman and Nate Forman. *Java Reflection in Action*. Manning Publications, 2004. ISBN 978-1932394184.

[20] I. M. Forsythe, P. Milligan, and P. P. Sage. Probabilistic program analysis for parallelizing compilers. In *Proceedings of the 6th International*

Conference on High Performance Computing for Computational Science, pages 610–622. Springer-Verlag, June 2005. doi: 10.1007/11403937_46.

[21] Armando Fox, Brad Johanson Pat Hanrahan, and Terry Winograd. Integrating information appliances into an interactive workspace. *IEEE Computer Graphics and Applications*, 20(3):54–65, May 2000. doi: 10.1109/38.844373.

[22] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, November 1994. ISBN 0-201-63361-2.

[23] J. Gao, G. Kar, and P. Kemarii. Approaches to building self healing systems using dependency analysis. In *Proceedings of the 2004 IEEE/IFIP Network Operations and Management Symposium*, volume 1, pages 119–132, April 2004. ISBN 0-7803-8230-7. doi: 10.1109/NOMS.2004.1317649.

[24] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, January 1979. ISBN 0-7167-1045-5.

[25] David Garlan, Robert Allen, and John Ockerbloom. Architectural mismatch: Why reuse is still so hard. *IEEE Software*, 26(4):66–69, July 2009. doi: 10.1109/MS.2009.86.

[26] Thomas Gschwind. Type based adaptation: An adaptation approach for dynamic distributed systems. In *Proceedings of the Third International Workshop on Software Engineering and Middleware*, volume 2596 of *Lecture Notes in Computer Science*, pages 130–143, May 2002. ISBN 978-3-540-07549-3. doi: 10.1007/3-540-38093-0_9. URL http://www.springerlink.com/content/x4718jhu072m2m37/.

[27] Sven Moritz Hallberg. Eternal compatibility in theory. *The Monad.Reader*, 2, May 2005. URL http://www.haskell.org/tmrwiki/EternalCompatibilityInTheory. No longer online, available from the Internet Archive Wayback Machine.

[28] Yuan-Shin Hwang, Peng-Sheng Chen, Jenq Kuen Lee, and Roy Dz-Ching Ju. Probabilistic points-to analysis. In *Proceedings of the 14th International Workshop on Languages and Compilers for Parallel Computing*, pages 290–305. Springer-Verlag, 2001. doi: 10.1007/3-540-35767-X_19.

[29] Donald B. Johnson. Efficient algorithms for shortest paths in sparse networks. *Journal of the ACM*, 24(1):1–13, January 1977. doi: 10.1145/321992.321993. URL http://doi.acm.org/10.1145/321992.321993.

[30] N. D. Jones and F. Nielson. Abstract interpretation: A semantics-based tool for program analysis. In *Handbook of Logic in Computer Science*, volume 4, pages 527–636. Oxford University Press, 1995.

[31] Piotr Kaminski, Marin Litoiu, and Hausi Müller. A design technique for evolving web services. In *Proceedings of the 2006 Conference of the Center for Advanced Studies on Collaborative Research*, Toronto, Ontario, Canada, October 2006. ACM Press. doi: 10.1145/1188966.1188997.

[32] Ralph Keller and Urs Hölzle. Binary component adaptation. In *Proceedings of the 12th European Conference on Object-Oriented Programming*, volume 1445 of *Lecture Notes on Computer Science*, pages 307–329. Springer-Verlag, July 1998. ISBN 978-3-540-64737-9. doi: 10.1007/BFb0054097.

[33] Gregor Kiczales, John Irwin, John Lamping, Jean-Marc Loingtier, Cristina Videira Lopes, Chris Maeda, and Anurag Mendhekar. Aspect-oriented programming. In *Proceedings of the European Conference on*

Object-Oriented Programming, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer-Verlag, June 1997. ISBN 978-3-540-63089-0. doi: 10.1007/BFb0053381.

[34] Byoungoh Kim, Kyungmin Lee, and Dongman Lee. An adapter chaining scheme for service continuity in ubiquitous environments with adapter evaluation. In *Proceedings of the Sixth IEEE International Conference on Pervasive Computing and Communications*, pages 537–542. IEEE Computer Society Press, March 2008. doi: 10.1109/PERCOM.2008.70.

[35] Tim Kindberg and Armando Fox. System software for ubiquitous computing. *IEEE Pervasive Computing*, 1(1):70–81, January 2002.

[36] Andreas Knüpfer and Wolfgang E. Nagel. Construction and compression of complete call graphs for post-mortem program trace analysis. In *Proceedings of the 2005 International Conference on Parallel Processing*, pages 165–172. IEEE Computer Society Press, June 2005. ISBN 0-7695-2380-3. doi: 10.1109/ICPP.2005.28.

[37] Woralak Kongdenfha, Hamid Reza Motahari-Nezhad, Boualem Benatallah, Fabio Casati, and Régis Saint-Paul. Mismatch patterns and adaptation aspects: A foundation for rapid development of web service adapters. *IEEE Transactions on Services Computing*, 2(2):94–107, April 2009. doi: 10.1109/TSC.2009.12.

[38] Serge Lang. *Algebra*, volume 211 of *Graduate texts in mathematics*, page 9. Springer-Verlag, revised third edition, 2002.

[39] James R. Larus. Abstract execution: A technique for efficiently tracing programs. *Software Practice and Experience*, 20(12):1241–1258, December 1990. doi: 10.1002/spe.4380201205.

[40] Dongman Lee, Seunghyun Han, Insuk Park, Saehoon Kang, Kyungmin Lee, Soon J. Hyun, Young-Hee Lee, and Geehyuk Lee. A group-aware middleware for ubiquitous computing environments. In *Proceedings of the 14th International Conference on Artificial Reality and Telexistence*, pages 291–298, December 2004.

[41] Keunwoo Lee, Anthony LaMarca, and Craig Chambers. HydroJ: Object-oriented pattern matching for evolvable distributed systems. In *Proceedings of the 2003 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications*, pages 205–223. ACM Press, October 2003. ISBN 1-58113-712-5. doi: 10.1145/949305.949324. URL http://doi.acm.org/10.1145/949305.949324.

[42] Kyungmin Lee, Byoungoh Kim, Yoo Chung, and Dongman Lee. Loss-minimizing interface adapter chaining. Submitted to the Journal of Systems and Software, October 2009.

[43] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. The Java Series. Addison-Wesley, 1997. ISBN 0-201-63452-X.

[44] Joseph P. Loyal and Susan A. Mathisen. Using dependence analysis to support the software maintenance process. In *Proceedings of the 1993 Conference on Software Maintenance*, pages 282–291, September 1997. ISBN 0-8186-4600-4. doi: 10.1109/ICSM.1993.366934.

[45] Jeff Magee and Jeff Kramer. *Concurrency: State Models and Java Programs*. John Wiley & Sons, 2nd edition, July 2006. ISBN 978-0-470-09355-9.

[46] David Victor Mason. *Probabilistic program analysis for software component reliability*. PhD thesis, University of Waterloo, 2002.

[47] Vlada Matena, Sanjeev Krishnan, Linda DeMichiel, and Beth Stearns. *Applying Enterprise JavaBeans: Component-Based Development for the J2EE Platform*. Addison-Wesley, second edition, May 2003. ISBN 0-201-91466-2.

[48] Bertrand Meyer. *Object-Oriented Software Construction*, page 342. Prentice-Hall, 2nd edition, 1997. ISBN 0-13-629155-4.

[49] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes. *Information and Computation*, 100(1):1–77, September 1992. doi: 10.1016/0890-5401(92)90008-4.

[50] John C. Mitchell. *Foundations for Programming Languages*, chapter 5, pages 308–312. Foundations of Computing. The MIT Press, 1996. ISBN 0-262-13321-0.

[51] Yiannis N. Moschovakis. *Notes on Set Theory*, chapter 4, page 39. Springer-Verlag, 1994. ISBN 0-387-94180-0.

[52] Hamid Reza Motahari Nezhad, Boualem Benatallah, Axel Martens, Francisco Curbera, and Fabio Casati. Semi-automated adaptation of service interactions. In *Proceedings of the 16th International Conference on World Wide Web*, pages 993–1002. ACM Press, May 2007. ISBN 978-1-59593-654-7. doi: 10.1145/1242572.1242706. URL http://doi.acm.org/10.1145/1242572.1242706.

[53] Simon L. Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 2003. ISBN 0-521-826144. URL http://haskell.org/onlinereport/.

[54] Pascal Poizat, Gwen Salaün, and Massimo Tivoli. An adaptation-based approach to incrementally build component systems. *Theoretical Computer Science*, 182:155–170, June 2007. doi: 10.1016/j.entcs.2006.09.037.

[55] Shankar R. Ponnekanti and Armando Fox. Application-service interoperation without standardized service interfaces. In *Proceedings of the First IEEE International Conference on Pervasive Computing and Communications*. IEEE Computer Society Press, March 2003. doi: 10.1109/PERCOM.2003.1192724.

[56] James M. Purtilo and Joanne M. Atlee. Module reuse by interface adaptation. *Software Practice and Experience*, 21(6):539–556, June 1991. doi: 10.1002/spe.4380210602.

[57] Erhard Rahm and Philip A. Bernstein. A survey of approaches to automatic schema matching. *The VLDB Journal*, 10(4):334–350, December 2001. doi: 10.1007/s007780100057.

[58] Ralf H. Reussner. Automatic component protocol adaptation with the CoConut/J tool suite. *Future Generation Computer Systems*, 19(5):627–639, July 2003. doi: 10.1016/S0167-739X(02)00173-5.

[59] Tim Rohaly. Report on the fourth Jini community meeting, 2000. URL http://www.javaworld.com/javaworld/javaone00/j1-00-jinicomm.html. No longer online, available from the Internet Archive Wayback Machine.

[60] Manuel Román, Christopher Hess, Renato Cerqueira, Anand Ranganathan, Roy H. Campbell, and Klara Nahrstedt. A middleware infrastructure for active spaces. *IEEE Pervasive Computing*, 1(4):74–83, October 2002.

[61] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*, chapter 3, page 75. Prentice-Hall, second edition, 2003. ISBN 0-13-790395-2.

[62] Barbara G. Ryder. Constructing the call graph of a program. *IEEE Transactions on Software Engineering*, 5(3):216–226, May 1979.

[63] Neeraj Sangal, Ev Jordan, Vineet Sinha, and Daniel Jackson. Using dependency models to manage complex software architecture. In *Proceedings of the 2005 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications*, pages 167–176. ACM Press, 2005. ISBN 1-59593-031-0. doi: 10.1145/1094811.1094824. URL http://doi.acm.org/10.1145/1094811.1094824.

[64] SCA. Service component architecture, November 2006. URL https://www.ibm.com/developerworks/library/specification/ws-sca/.

[65] Mary Shaw. Architectural issues in software reuse: It's not just the functionality, it's the packaging. *ACM SIGSOFT Software Engineering Notes*, 20(SI):3–6, August 1995. doi: 10.1145/223427.211783. URL http://doi.acm.org/10.1145/223427.211783.

[66] Harald Søndergaard. An application of abstract interpretation of logic programs: Occur check reduction. In *Proceedings of the European Symposium on Programming*, pages 327–338. Springer-Verlag, March 1986. ISBN 978-3-540-16442-5. doi: 10.1007/3-540-16442-1_25.

[67] João Pedro Sousa and David Garlan. Aura: An architectural framework for user mobility in ubiquitous computing environments. In *Proceedings of the 3rd IEEE/IFIP Conference on Software Architecture: System Design, Development and Maintenance*, pages 29–43, August 2002. ISBN 1-4020-7176-0.

[68] Bridget Spitznagel and David Garlan. A compositional formalization of connector wrappers. In *Proceedings of the 25th International Conference on Software Engineering*, pages 374–384. IEEE Computer Society Press, May 2003. ISBN 0-7695-1877-X. doi: 10.1109/ICSE.2003.1201216.

[69] Gilbert W. Stewart. *Matrix Algorithms: Basic Decompositions*, chapter 1, page 46. Society for Industrial and Applied Mathematics, 1998. ISBN 978-0-898714-14-2.

[70] Jeffrey D. Ullman. *Elements of ML Programming*, pages 50–51. Prentice-Hall, 1998. ISBN 0-13-080391-X.

[71] Julien Vayssière. Transparent dissemination of adapters in Jini. In *Proceedings of the Third International Symposium on Distributed Objects and Applications*, pages 95–104, September 2001. ISBN 0-7695-1300-X. doi: 10.1109/DOA.2001.954075.

[72] Steve Vinoski. The more things change ... *IEEE Internet Computing*, 8 (1):87–89, January 2004. doi: 10.1109/MIC.2004.1260709.

[73] Seth Warner. *Modern Algebra*, chapter 1, pages 29–35. Dover, 1990. ISBN 0-486-66341-8.

[74] Mark Weiser. Some computer science issues in ubiquitous computing. *Communications of the ACM*, 36(7):75–84, July 1993. doi: 10.1145/159544.159617. URL http://doi.acm.org/10.1145/159544.159617.

[75] John Whaley. A portable sampling-based profiler for Java virtual machines. In *Proceedings of the ACM 2000 Java Grande Conference*, pages 78–87. ACM Press, June 2000. ISBN 1-58113-288-3.

[76] S. Yacoub, B. Cukic, and H. H. Ammar. A scenario-based reliability analysis approach for component-based software. *IEEE Transactions on Reliability*, 53(4):465–480, December 2004. doi: 10.1109/TR.2004.838034.

[77] Daniel M. Yellin and Robert E. Strom. Protocol specifications and component adaptors. *ACM Transactions on Programming Languages and Systems*, 19(12):292–333, March 1997. doi: 10.1145/244795.244801. URL http://doi.acm.org/10.1145/244795.244801.

Acknowledgments

I would like to thank my thesis advisor Professor Dongman Lee for his indispensable help with the research that has culminated in this dissertation. He has helped me through my setbacks and made possible my triumphs during the years of my doctoral research.

I would also like to express my gratitude to the other members of my thesis advisory committee: Professor Younghee Lee, Professor Soon Joo Hyun, and Professor In-Young Ko from KAIST, and Doctor Myung-Joon Kim from ETRI. They have helped made this dissertation a more valuable work.

I would like to thank all the members of the Collaborative Distributed Systems and Networks Laboratory at KAIST, with whom there have been many stimulating discussions. In particular, I would like to thank Byoungoh Kim and Kyungmin Lee, whose work on interface adaptation became the starting point of the research that comprised my thesis.

Last but not least, I am eternally grateful to my parents and sister, without whom I would not be here today.