Binary code analysis for application integration

Niek Oost

September 2, 2008

Abstract

Application software for business usage is everywhere; almost any company in the world depends on the application software it uses, it has become a major asset for these companies.

Due to multiple (sometimes untangible) factors, application functionality is rebuild for new applications, in stead of reused. This practice leads to the waste of time and money, both in the development of the functionality as well as the debugging efforts that are required.

In our view, existing application functionality should be used as long as it is possible. We present a method for analyzing applications, finding reusable functionality in the applications and make the found functionality available as a Web Service.

This thesis presents an overview of technologies for remodelling applications as well as an overview of technologies to control application execution externally. A comparative analysis in each of these directions is presented, together with a design for an application that integrates the technologies for Web Services mining and implementation.

Contents

Chapter 1

Introduction

'He didn't tell me straight away of course. Oh no. First he wiped a couple of windows and charged me a fiver. Then he told me.'

Arthur Dent in The Hitch Hikers Guide to the Galaxy

Since the introduction of the personal computer, an incredible number of applications have been created for it. Many of them are very innovative, but in a lot of cases the same functionality is reinvented. A lot of work is being done over and over again, and this could be avoided.

1.1 Research context

Most business applications have been in use for many years and during their lifetimes they are extensively tested and improved. These applications have become major assets in the companies that use them and they shape the business of their owners. Each time business requirements undergo major changes it gets harder to keep these 'dinosaurs' up to date. Most of the time this is due to the financial aspects of maintaining legacy code, but often architectural aspects and changed user requirements play a major role too. The following scenario illustrates this case:

EXAMPLE 1.1.1 (LANZAROTE) *Imagine a hotel in Lanzarote that has a booking system on a computer at the main desk. It has functionality for reserving rooms, overviews of used capacity, cost of rooms, etc. The software has once been written by the neighbor of the owner of the hotel. This system has been used successfully for many years and it works perfectly for the current situation at the hotel.*
To expand their customer base, the hotel manager has decided to work together with a big organization that sells vacations over the internet. This organization demands access to the hotel's booking system to automate the booking process.
The hotel manager wants to fulfill the demands, but he also wants to keep using the current system. Unfortunately the hotel never received the source code of the booking system and the programmer seems to have vanished.

With current technology the hotel manager has to replace the system with something else that will probably work quite differently. This means that the personnel has to learn to work with this new application. It also implies that all data that is encompassed by the current system needs to be migrated to the new application. This might be very hard to accomplish, since proprietary data storage formats are used very often in this type of system. These problems are common when choosing application migration in a company.
Current application binaries can contain useful functionality that is not used to its full potential when it is locked inside a single application. When new applications are developed, many functions that have been build and tested for earlier systems are being built again, which is expensive in both money and time.
Most business applications have been in use for many years and they are extensively tested and updated in that period, which (hopefully) increased their quality as well as their value. For a variety of reasons, e.g. the end date of an existing contract, maintenance on applications is stopped. Another reason might be that the original implementor stopped its activities and the source code of the application has not been made available.
When parts of the functionality of an application has been rendered obsolete due to changed business requirements, in many cases the decision is made to replace the entire application. In our view this practice is not necessary, since it is (at least theoretically) possible to reuse certain functions of an application binary in other systems. In this way the effort that has been put in the creation of qualitatively good functionality is not wasted.
To prevent a company from having to bear the downsides of application migration, we are proposing a solution. Our solution can automatically discover reusable functionality in application binaries and make the discovered functionality available as a Web Service. A *Web Service* is a component that can be accessed via multiple formats and protocols through some network. A technique we call *"Web Services mining"* is developed to find potentially interesting functionality to provide as operations of a Web Service. After analysis, a subset of the found candidates can be selected to install as a Web Service.

1.2 An introduction to Web Services technology

Application integration is a topic that has received a lot of attention throughout the years. An enormous number of papers has been written on the subject and many standards have been developed in this field. Application vendors have either modified their products to adhere to (some of) these standards or they rely on third party

technology to let their products integrate with other products.
A set of standards that receives a lot of attention nowadays is Web Services technology. It is maintained by the World Wide Web Committee (W3C) and it is supported by all major vendors. The technology builds on other standardized technologies (such as XML [17]) while it is not tied to a certain platform. Our research focusses on Web Services technology since it is a simple, yet powerful technology that is built around the idea of operations executed by some entity.
In their "Web Services glossary" [16] the W3C provides the following definitions for *Web Service* and *service* where 'Web Service' is a concrete technology and 'service' an abstract concept.

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

2 DEFINITION (SERVICE) *A service is an abstract resource that represents a capability of performing tasks that form a coherent functionality from the point of view of providers entities and requesters entities. To be used, a service must be realized by a concrete provider agent.*

In these definitions we can find the notion of a single element (a Web Service), somewhere in a network, that provides several functions (operations) to the outside world. Other elements in that world can send requests to such a service and the service will do its best to execute the requests.
Now we will continue with providing the reader with a technological overview of Web Services with respect to our research, followed by a discussion on how to design Web Services.
Web Services are described by the Web Service Description Language, in our case, version 2.0 (WSDL2.0). This standard [19] enables the separation of the description of the abstract functionality offered by a service from concrete details of a service implementation. The conceptual model of WSDL 2.0 can be described as a set of components, see figure 1.1.
In the component model (figure 1.1), the `Description` component is a container for WSDL 2.0 components and type system components. The WSDL 2.0 components are `Interface`, `Bindings` and `Services`. The type system components are element declarations and type definitions.

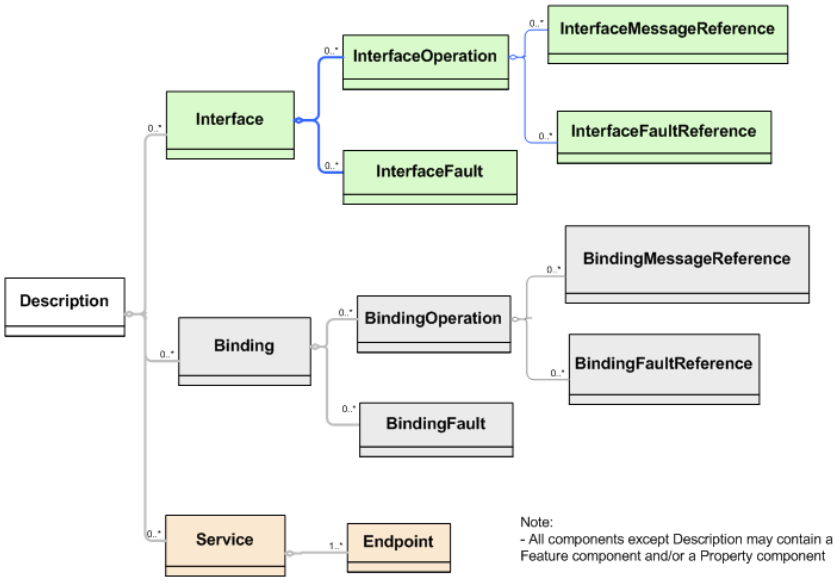


Figure 1.1: *WSDL 2.0 component model*

The *Interface* component describes sequences of messages that a service sends and/or receives. Related messages are grouped into operations. Operations are sequences of input and output messages. An interface is a set of operations.
An *InterfaceOperation* component describes an operation that a given interface supports. An operation is an interaction with the service consisting of a set of messages exchanged between the service and other parties involved in the interaction. An *InterfaceMessageReference* component defines the content of a message exchanged in an operation. An *InterfaceFaultReference* component associates a defined type, specified by an *InterfaceFault* component, to a fault message exchanged in an operation.
An *InterfaceFault* is an event that occurs during execution of a message exchange that disrupts the normal flow of messages. A fault is raised when a party is unable to communicate an error condition inside the normal message flow.
A *Binding* component describes a concrete message format and transmission protocol which may be used to define an endpoint. A binding component defines the implementation details necessary to access the service. A *BindingFault* component describes a concrete binding of a particular fault within an interface to a particular concrete message format. The *BindingOperation* component describes the concrete message format(s) and protocol interaction(s) associated with a particular interface operation for a given endpoint. A *BindingMessageReference* component describes a concrete binding of a particular message participating in an operation to a particular concrete message format. A *BindingFaultReference* component describes a concrete binding of a particular fault

participating in an operation to a particular concrete message format.
A *Service* component describes a set of endpoints at which a particular deployed implementation of the service is provided. The endpoints thus are in effect alternate places at which the service is provided. An *Endpoint* component defines the particulars of a specific endpoint at which a given service is available.
Web services rely on some kind of transport, SOAP [18] and HTTP are common. A SOAP message is an XML document containing the following elements:

- An Envelope element that indicates that the XML document is a SOAP message.
- An optional Header element
- A Body element that contains the call and response information
- An optional Fault element

1.3 Modeling Web Services after application binaries

Desktop applications that are used by companies nowadays generally provide a Graphical User Interface (GUI). All functionality used on a daily basis is incorporated in that GUI, which makes it the authoritative source for functionality at a high level of abstraction. All functionality is grouped in a more or less logical way. Dialogs are used to group related functionality and information together, while style elements (like raised or line borders) on a dialog provide clues toward an even more cohesive set of information items.
In our view this implies that the GUI of an application is the single most important source that provides information about the functionality and the grouping of that functionality for an application. And therefore the most important provider of information for Web Services mining.
Our research is aimed at business applications that are forms-based information systems that manage concrete entities. In our research the notion of a *candidate Web Service operation* (or candidate) is very important. We want to detect application functionality that is strongly related so we can present it as a single Web Service operation to the outside world. Of course our system should not generate candidates that are conceptually not sound or candidates that take too long to execute.
Designing Web Services requires to make choices within the constraints imposed by Web Services technology. Therefore we need to think about some rules of thumb for designing Web Services, for instance in the field of granularity. It is not trivial to decide what the best granularity for Web Services operations is. Should we implement operations that tell us exactly how many products are sold by a company and a function

Primitive	Description
Add entity	Adds an entity to the storage.
Modify entity	Modifies an entity already existing in the storage.
List entities	Generates a list of entities of a certain type existing in the storage.
Remove entity	Removes an existing entity.
Remove multiple entities	Removes multiple existing entities.

Table 1.1: *Primitives for Web Service operations*

for each category of products? Should we design operations that gives us all statistical information at once? What is a good measure for result set size (in bytes)? What is the maximum in number of messages contained by an interface? How much time is a Web Service allowed to take before it reacts?

All these questions help us understand the type of functionality to present as a Web Service. More information can also be gained by looking at the standards surrounding Web Services.

In our approach an application is a storage for entities. This is mainly because our approach is based on user interface analysis for the discovery of Web Services and our sense that it is much more valuable to aim at those types of applications. They provide functionality at a higher level of abstraction than an application like a calculator.

When analyzing applications as storage for entities, there are several primitives for the operations:

In our research the possibilities for modeling Web Services after application binaries is explored. In short, a model is created based on the analysis of an application binary. This model is further analyzed and converted into a more convenient model that is used to find candidate Web Service operations. After the candidates have been selected, an application is executed by an execution engine that waits for incoming requests to be forwarded to the application binary.

In the context of our research we are referring to the application analyzer as the Web Service Miner (WSMiner) and to the execution engine as Web Service Mining Executer (WSMExecuter). The input of the WSMiner is an application binary (Binary) and the output is called an WSMProfile. The WSMProfile is the input to the WSMExecuter, it is used to start up the Web Service and generate a WSDL file for it. The set of these elements is called the WSMToolkit. This process is shown in the flow-chart of figure 1.2.

It is not necessary to create two separate applications, but it seems a good idea to have a stand alone executer for situations where multiple instances of the same application binary need to provide the same Web Service operations. Such a situation can occur in a company where multiple users have the same application on their desktop computers. They manage their own information and these sets of information need to be integrated

by an overarching Web Service. In such a case it is easier to design the profiles centrally.

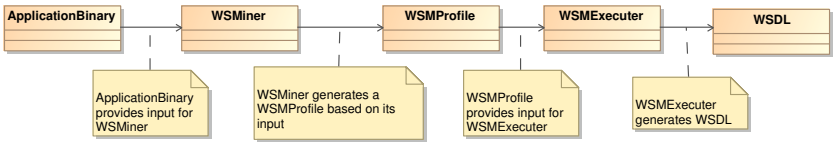


Figure 1.2: *WSM Toolkit*

1.4 Example

To clarify the workings of our system we provide a small example together with expected results:

EXAMPLE 1.4.1 (A SIMPLE APPLICATION : DAILYNOTE) *An application accepts notes about the activity of an employee. Its interface consists of an input box for text, together with buttons for adding a note or canceling the addition of a note (see figure 1.3).*

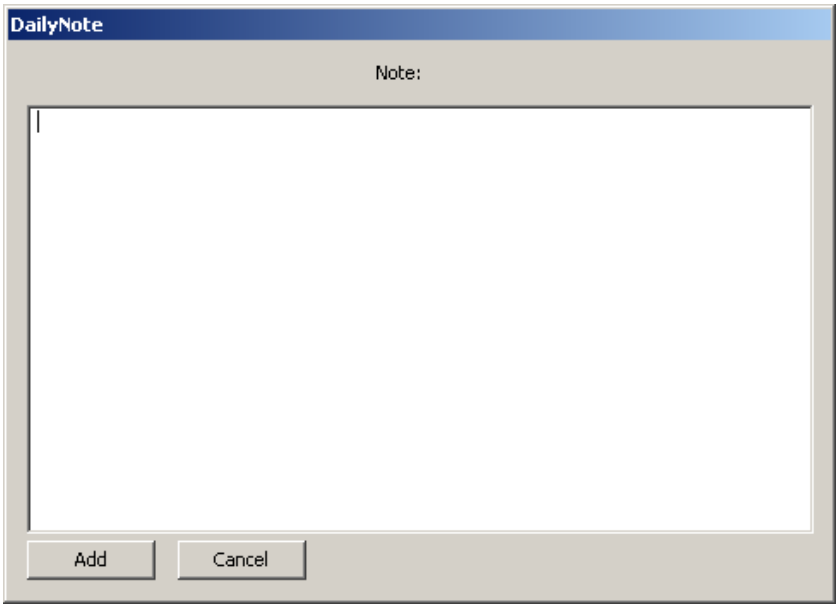


Figure 1.3: *DailyNote*

To generate a Web Service for the application, the WSMiner loads the DailyNote executable and analyzes its structure. Based on the interface of DailyNote a WSMProfile is

generated. It contains one candidate : `AddNote`, a Web Service operation that accepts a string and returns nothing.
The WSMExecuter is started with the WSMProfile, it immediately generates a WSDL file for the service it is about to start. The WSMExecuter then creates a thread which listens at a certain port for incoming requests. When an operation is invoked, the message is forwarded to the DailyNote instance that has already been started. The WSMExecuter puts the received text into the textbox of the application and sends a `ButtonClick event` to the ADD button.
After DailyNote has been analyzed by the WSM Toolkit, we would expect a certain output. For this example we have handcrafted the example WSDL file of figure A.2 in appendix A. A SOAP request to this Web Service can be found in figure A.1.

1.5 Research Questions and contributions

When business requirements change, the applications supporting the business have to change as well, otherwise they will become obsolete. In many situations application refactoring is not an option, which leads to either the development of new applications, or organizations using work arounds for the problems.
In this thesis, we investigate a solution to improve this situation. We consider the following research question and sub-questions:

1. How can a binary application be adapted to Web Services technology?
   (a) How to find Web Service operation candidates in an application binary?
   (b) How to adapt an application binary for Web Services technology?

This thesis is organized as follows. In chapter 2, we focus on *model recovery from application binaries*. In chapter 3, we explore the possibilities for adapting binaries to Web Services technology. Chapter 4 discusses an example scenario in which our technology should provide a solution. The chapter introduces the use cases that our system is built for and requirements for our system.
Chapter 5 introduces the models used in our system and the conversions between those models. Chapter 6 provides the design of our proposed solution, while chapter 7 is concerned with the implementation of our prototype. Finally, chapter 8 contains a summary conclusion and suggestions for future work.

Chapter 2

Application Model Recovery

Be patient, for the world is broad and wide.

Edwin A. Abbott

Application analysis has been a popular subject of research for several decades. Performance profiling [10], performance evaluation and bug detection are important examples. These techniques all start with discovering (part of) the structure, function and operation of the application, also called reverse engineering.
For their specific purposes a single technological approach (disassembling) is enough to retrieve the required information. In our case, multiple technologies have to be combined to retrieve a sufficiently rich model of the application under scrutiny. Our approach combines binary analysis with user interface analysis and a knowledge base at a much higher level of abstraction than byte code.
In this chapter an overview of solutions in the different fields of research is provided.

2.1 Application analysis

Application analysis can indicate a multitude of options: source versus binary code analysis, dynamic versus static analysis, black box versus white box analysis. In this section we provide an overview of the technologies and a choice for our system.
Without any knowledge of the workings of an application, it is virtually only possible to provide remote access to the main message loop of an application. In such a case we could only give users of a Web Service a 'key stroke injection' operation which would forward the key stroke to the application. While this could be used for some form of application integration through Web Services access, it is not the desired level of interaction for our purposes. Especially since it implies that extensive knowledge about the application is needed at the calling party (not to mention the painstakingly slow service responses).
Before application functionality can be exported, it is necessary to know what functionality is provided by an application. A thorough understanding of the application internals is needed to provide higher level constructs to construct Web Service operation prototypes for a specific application. Because source code is usually not available, a

reliable analysis result can only be gathered from the *binary code*, the result of compiling source code. The binary code is the most authoritative source of information about program content and behavior, since it incorporates every decision in source code as well as behavior introduced by compilers and linkers.
To make sure that our technique can be used in as many situations as possible, we assume that we have to deal with *stripped binary code*: application binary code without symbolic information available. If symbolic information is available we neglect it since the symbolic information can be erroneous or incomplete. Harris and Miller discuss the analysis of stripped binary code in [8].
There are two types of binary analysis: dynamic and static. While the static version analyzes an executable file without actually executing it, the dynamic version is focussed on analyzing an application during execution.
Dynamic analysis is performed by executing an application and monitoring its behavior. During the analysis, any instruction that is executed can be inspected, together with the contents of processor registers, stack and memory. This is usually done by instrumenting the instructions of interest with an analysis function. When the code is executed by a tool, using a framework for dynamic analysis, the analysis function is called by the framework, providing processor state information to the analysis function. Such frameworks are called dynamic binary instrumentation (DBI) frameworks. These frameworks can instrument any instruction executed by an application. The main technique is to take a block of instructions and instrument them before execution (Just-In-Time principle). Examples of binary instrumentation tools are: PIN, Dyninst, Valgrind and DynamoRIO.
To overcome the fact that dynamic analysis can only analyze a single path of execution, Pan et al. [13] discus execution path modification during dynamic analysis. The paper describes a technique that provides three functionalities : (1) checkpointing the execution state, (2) resuming execution at a checkpoint and (3) starting execution at an arbitrary point in the program with a specific architectural state. In our view these functionalities are more suitable for modifying application behavior than analyzing the behavior, therefore we discuss this technique in more detail in chapter 3.
As argued by the creators of TTanalyze [1] we could also use a *hardware emulator* to run a test application within an installation of an operating system. By analyzing everything that happens during execution we could retrieve the most important activities. In our view, this method implies a lot of overhead due to setting up a hardware emulator with an operating system. For our situation a smaller, easier solution can be constructed.
We could also treat an application as a black box and use *device drivers* to monitor activity associated with the application. The drawback of this technique is that it requires an operator to monitor application behavior during usage. If a complete set of Web Service operations has to be found, an operator has to actually invoke every single function of

the application. This is not the easiest of methods from the operator's (or consultant's) point of view, not to mention the costs associated with this type of analysis.
Static analysis is the analysis of the source or the compiled code of an application without executing it. It consists of analyzing code and extracting structures in the code at different levels of granularity. This is a very intensive process which might be even more complicated when the distributor of the application has deliberately *obfuscated* the code; reordering and rewriting the binary code in such a way that it is very hard to recover the original structures, while the application execution semantics stay the same. A big advantage of the static method is that it is able to analyze any possible path of execution of an application, in contrast with dynamic analysis, which can only analyze a single path of execution at a time. Static analysis can cover a complete program.
The first step of static analysis is usually *disassembling* the binary code, a process that consists of transforming the binary code into corresponding assembler instructions. These instructions are then grouped in such a way that information about the application structure, like control flow and data flow, is easily accessible.
There are no known frameworks for complete static binary analysis publicly available, at least our research has not come up with anything useful. Therefore we can only take advantage of byte code decoders that can be used in a static analyzer built by ourselves. Since we need a complete picture of an application and with the given drawbacks of dynamic analysis, we have to use static analysis. Therefore we require that an application that is to be analyzed should not be obfuscated in such a way that our static analyzer is unable to decode it properly.

2.2 Byte code and processor instructions

Designing a static analyzer requires knowledge on the subject of the analysis: byte code. In this section we will describe several observations about byte code. Several concepts are defined: *sequence of instructions*, *path of execution* and *loop of instructions*
A processor is designed to execute a certain set of instructions, encoded in a processor architecture specific byte code. There are some general concepts that are implemented by any processor architecture. In general, any processor has instructions for changing the path of execution, these are called branching instructions. These branching instructions can either be unconditional or conditional. An *unconditional branch* always changes the path of execution. A *conditonal branch* depends on the state of a certain value (in general boolean values, called flags, stored in a dedicated register) to decide on which instruction will be executed next.
We define three relations between instructions:

- `target`: an instruction i is a target of instruction j if a processor might execute

instruction i directly after j when the application is processed.

- `follows`: an instruction i `follows` an instruction j if and only if i is a `target` of j.
- `precedes`: an instruction i `precedes` an instruction j if and only if j is a `target` of i.

For the design of our static analyzer we observe several things:

1. an application has a single entry point, the first instruction that is executed when the application is started by the operating system, this is called the `AddressOfEntry`.
2. except for the AddressOfEntry, every instruction i is a target of some other instruction j.
3. an unconditional branch instruction b can have more than one possible target instruction that is executed after b, just one of those target instructions will actually be executed after b.
4. conditional branches have two or more target instructions of which only one will be executed based on some condition. These targets are the fall-through address and the address(es) pointed to by the branching target operand.
5. every instruction can be the target of multiple other instructions, hence loops can exist.
6. a CALL instruction can be seen as a short descriptor for the execution of a larger set of instructions.
7. instructions can take up multiple bytes, so not every location in the byte code depicts a single instruction.

To recover all possible paths of execution of an application, we need to find every branching instruction, thus unconditional and conditional branches and their target addresses. An unconditional branch has only one branching target address, whereas a conditional branch has a branching target as well as a fall-through target address. Although call-instructions are also perceived as branching instructions from the processor-viewpoint, we do not consider them as branching instructions in our system. The reason for this choice is that the Execution Path Model should distinguish between functions and 'normal' execution paths.
We define two relationships between instructions and operands:

- `affects`: An instruction i `affects` operand o if the execution of i can possibly change the value of o.
- `dependson`: An instruction i `dependson` operand o if i can not be executed without the availability of the value indicated by o.

We define the following:

- A *sequence of instructions* is a list of instructions $i_0, i_1, \ldots, i_n, \ldots$ that has a specified value for i_n, for each integer $n \in \mathbb{N}$. Each instruction i_n precedes the instruction i_{n+1}.
- A *path of execution* is an arbitrary length sequence of instructions that starts at the first instruction of an application and ends at a last instruction.
- A *loop of instructions* is a sequence of instructions $(i_0, i_1, \ldots, i_n, \ldots)$ where $i_{n+1} = i_0$.
- A *path* is a list of instructions that are not necessarily related as in a `follows` or `precedes` relationship, but they are ordered in execution order.

With these definitions in place we can describe how the structure of an the paths of execution can be recovered correctly.

2.3 Execution Path Model

The model is composed from the input byte code. Each instruction is decoded and then added at the right place in the set of sequences. The 'right place' depends on the location in the set of sequences of the instruction preceding the instruction that is added. When there is no preceding instruction as is the case with the first instruction, a new sequence is created. When an instruction logically follows the last instruction of a sequence, the instruction is added to the back of the sequence. In all other situations the sequence is being split after the source instruction and before the target instruction and the relations between the sequences are updated accordingly. In figures 2.1 and 2.2 all situations for linking two instructions in the set of sequences are described. In table 2.1 these situations are repeated with information about the effect of the newly added link on the set of sequences and loops. A = means that the number of items stays the same, a + means an increase with one, a ++ implies an increase with two.
The set of sequences and their relationships is one of the main elements of the Execution Path Model (EPM), another important element is the loop. A loop of instructions is a sequence of instructions that contains a branch to the first instruction of the sequence. Since we have a sequence model present, we can easily find the loops in the application

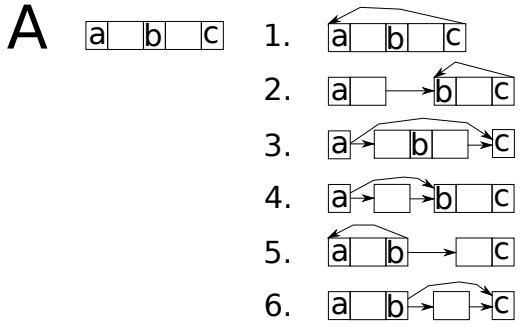
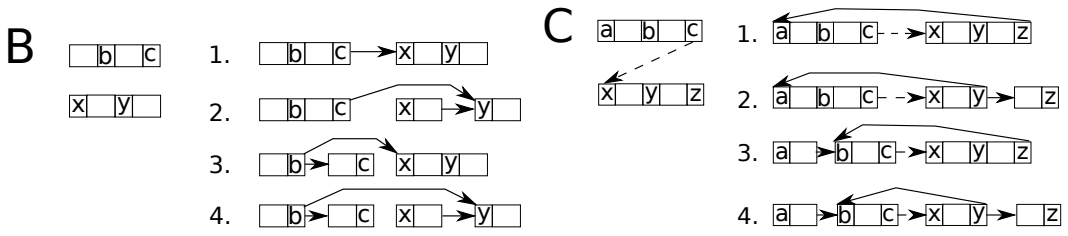


Figure 2.1: *Linking instructions in the same sequence*



(a) Two unrelated sequences (b) Two related sequences

Figure 2.2: *Linking instructions in distinct sequences*

by analyzing the sequence model. In this case a loop is a list of sequences where the first sequence is the target of the last sequence in the list.

3 DEFINITION (EXECUTION PATH MODEL (EPM)) *A set of sequences of instructions, the relationships between these sequences, the loops of sequences and the instructions and operands that make up these sequences together with the multitude of paths of execution.*

In figure 2.3 a class diagram for the Execution Path Model is provided. In this diagram we can find an `Instruction` class that has a composition relation with the `Operand` class and an association with itself. The `Sequence` class is associated with the Instruction class and itself (circular associations are allowed) in the model. The `Loop` class is associated with the Sequence class and the `Function` class is associated with the Loop class and the Sequence class. The EPM root element that aggregates the other classes of the EPM is not shown to keep the diagram easy to read.

Sit.	Description	Seq.	Loops
A1	Linking the end of a sequence to its own beginning	=	+
A2	Linking the end of a sequence to an instruction that is not the first instruction	+	+
A3	Linking the first instruction to the last	++	=
A4	Linking the first instruction to an instruction that is not the last instruction of the sequence	++	=
A5	Linking an instruction that is last nor first instruction of a sequence to the first instruction	+	=
A6	Linking an instruction that is last nor first instruction of a sequence to the last instruction	++	=
B1	Linking the last instruction of a sequence to the first instruction of an other sequence	=	=
B2	Linking the last instruction of a sequence to an instruction in another sequence of which it is not the first instruction	+	=
B3	Linking an instruction which is not the last instruction of a certain sequence to the first instruction of another sequence	+	=
B4	Linking an instruction which is not the last instruction of a certain sequence to an instruction of another sequence which is not the first instruction of that sequence	++	=
C1	Link the end of a sequence to the beginning of a preceding sequence	=	+
C2	Link an instruction which is not the last instruction of a sequence, to the beginning of a preceding sequence	+	+
C3	Link the end of a sequence to the middle of a preceding sequence	+	+
C4	Link an instruction which is not the last instruction of a sequence, to the middle of a preceding sequence	++	+

Table 2.1: *Creation of loops and sequences*

2.4 Analyzing targets

Our system analyzes all possible paths of execution of an application, at least when the instructions are located in the application binary. During analysis an EPM is built by adding instructions to the EPM and updating relations between the instructions.

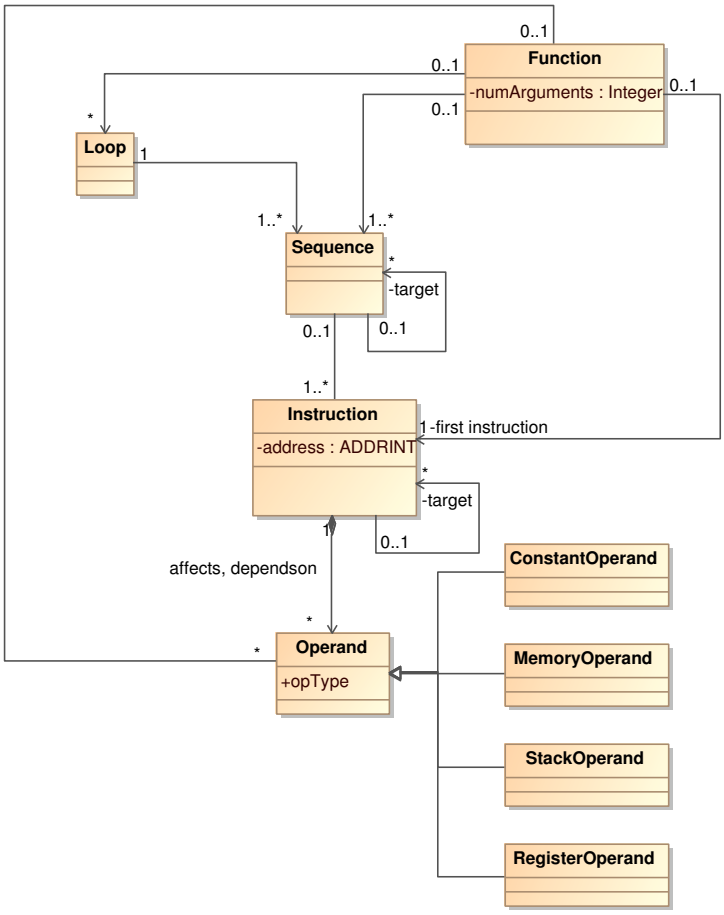


Figure 2.3: *Execution Path Model (EPM)*

When a branching instruction is encountered, there are two possible situations : the target is static (indicated by a constant value) or dynamic (depending on registers, memory or stack). When a static target is found it is obvious how to proceed, but the dynamic case is much harder. In the following paragraphs several examples of dynamic branching are discussed followed by a solution called *backtracking*. In short, backtracking is a method to retrieve a certain (dynamic) value needed by an instruction, by visiting preceding instructions.

The concepts of *operand descriptor* and *operand (descriptor) instantiation* are introduced to distinguish between the information provided by decoding a single instruction and the effect of the execution of multiple instructions. A descriptor informs us about the information needed to get to the right value, whereas an instantiation is (an element of) the set of possible values for the descriptor that can actually be encountered during

execution of the application.

We start with a short introduction into assembler codes. An overview of the assembler codes used in this chapter is provided in table 2.2. A complete overview of instructions accepted by a certain processor can be found in the manuals provided by the processor manufacturer. For the Intel IA-32 processor architecture, the manual can be found in the "Intel IA-32 Architecture Software Developer's Manual" [4].

The operands of the instructions can be (depending on the specific semantics of the instruction) *constant*, *memory* or *register*. It is also possible that a certain operand value is used as a pointer to a certain memory location. For instance `DWORD PTR [0x1001020]` should not be seen as the value `0x1001020` but as a pointer to the value of memory location `0x1001020`.

Code	Description
`MOV X, Y`	Move the value of operand Y into the location defined by operand X.
`CALL X`	Call the function depicted by operand X. This can be an address relative to the current instruction or a direct address.
`JMP X`	Jump to the address depicted by operand X. This can be an address relative to the current instruction or a direct address.
`JB X`	Jump to the address depicted by operand X if the right flags in the EFLAGS register are set. Otherwise go to the address directly following the instruction (fall-through address).
`PUSH X`	Put the value of operand X at the top of the stack.
`POP X`	Get the value of the top of the stack and put it into register X.
`SUB X,Y`	Subtract the value depicted by operand Y from the value of operand X and store the result in operand X.
`ADD X, Y`	Add the value depicted by operand Y to the value of operand X and store the result in operand X.
`INC X`	Increment the value of operand X with 1 and store the result in operand X.
`CMP X, Y`	Compare the values of operands X and Y and set the flags in the EFLAGS register accordingly.

Table 2.2: *Assembler codes*

Simple case

In figure 2.4 a very **simple case** is provided. When the analyzer tries to find the target(s) of the CALL instruction at line 2, it needs to find the most recent instruction that affects register EDI. In this case it should find the MOV instruction at line 1, which is *completely satisfied* (it does not depend on other values than the constant memory reference to 0x1001020). After finding the fully satisfied value, the analyzer processes the instruction at line 2 with the operand EDI replaced by a value of 0x1001020.

```
1           MOV EDI, DWORD PTR [0x1001020]
2           CALL EDI
```

Figure 2.4: *A simple backtracking example*

Multiple paths of execution

Most times, **multiple paths of execution** will lead to an instruction with a dynamic target. When the value for the dynamic target is affected by instructions in each of the preceding paths of the instruction, the analyzer has to deal with multiple sets of values to calculate the dynamic target value. Each path yields different instantiations of the required operands and these different sets of values need to be combined in a sensible way. In figure 2.5 an example is provided where the value of EDI at `addr2` has to be retrieved. The value can be 0x1003333 or 0x1008888 depending on the path of execution before reaching `addr2`. Our analyzer has to mark both addresses 0x1003333 and 0x1008888 as the start of a function. Multiple sets of value instantiation are also encountered when dealing with functions that are called from multiple locations.

```
1           ...
2           MOV EDI, 0x1003333
3   addr2:  CALL EDI
4           ...
5   addr1:  MOV EDI, 0x1008888
6           JMP addr2
7           ...
```

Figure 2.5: *Multiple possible values for EDI*

Stack values

Another complicated situation occurs when a **value on the stack** has to be retrieved. In such a case PUSH and POP instruction influence the stack as well as intermediate function calls. In figure 2.6 several PUSH and POP instructions precede the CALL instruction on line 6. The CALL depends on the value of register EAX, which depends on the stack pointer register ESP and a constant 0x08. In this case the second value from the top of the stack is required (every element on the stack consists of 4 bytes on 32-bit systems), so the value pushed at line 1 should be found. During backtracking, the analyzer increases the value of ESP by four when a POP is encountered and it decreases the value of ESP with four when a PUSH is found on the preceding path of execution.

```
1           PUSH 0x1001234
2           PUSH 0x1001230
3           POP EBX
4           PUSH 0x1001122
5           MOV EAX, DWORD PTR [ESP + 0x08]
6           CALL EAX
```

Figure 2.6: *Retrieving a stack value*

.

Function arguments

In general, **function arguments** are passed by pushing them onto the stack. When an instruction that is part of a function wants to retrieve the passed value, it has to look for a memory location relative to the value of the stack pointer. Due to our choice to treat functions apart from sequences and loops, we need a mechanism to find the values at these memory locations by searching for the instructions that call the function, followed by an analysis of the stack from such a function call location. The example in figure 2.7 shows that the instruction at line 6 needs the value of EAX, the instruction at line 6 provides EAX, but it depends on some value relative to the stack pointer. Since the instruction at line 5 is the first instruction of the function, the instruction locations where this function is called need to be analyzed. In this case `0x1001230` is returned.

Multiple operands

When an operand of an instruction depends on a memory location, it is possible that the values of **multiple operands** are needed to calculate a single memory address. A

```
1           PUSH 0x1001234
2           PUSH 0x1001230
3           CALL lab1
4           ...
5   lab1:   MOV EAX, DWORD PTR [ESP + 0x04]
6           CALL EAX
```

Figure 2.7: *Finding argument values from within a function*

.

memory address is calculated by taking a *segment*, *base*, *index* and *offset* value and combining them. Each of these values can be delivered by registers. Thus a set of values from each path of execution has to be retrieved. When multiple paths of execution are providing the required values, multiple sets of values should be retrieved. In figure 2.8 two sets of values should be found, one containing 0x10 for EBX and 0x1 for ECX and one set containing 0x1F for EBX and 0x2 for ECX. In such a case, the analyzer should not return two sets, one containing the two possible values for EBX (0x10 and 0x1F) and one containing values for ECX (0x1 and 0x2), but a set (EBX = 0x10, ECX = 0x1) and a set (EBX = 0x1F and ECX = 0x2).

```
1           MOV EBX, 0x10
2           MOV ECX, 0x1
3           JMP lab1
4           ...
5   lab1:   MOV EAX, DWORD PTR EBX[ECX]
6           CALL EAX
7           ...
8           MOV ECX, 0x2
9           MOV EBX, 0x1F
10          JMP lab1
```

Figure 2.8: *Finding multiple sets of values*

.

Local variables

Local variables in functions are often placed beyond the top of the stack, with a negative index relative to the stack pointer. In figure 2.9 an example is provided. A copy of the

original ESP is made on line 1, then ESP is decreased with 0x10, effectively reserving 16 bytes (4 DWORD) for local variables. Then the values of the first two DWORD are set. During backtracking from line 8 to find the value of `[ESP - 0x4]`, lines 7 and 5 will be used. In principle the term *relative constant* could help: the value of ESP is not actually needed to get to the value of DWORD PTR [ESP - 0x4], so we don't have to track back to the start of the function.

```
1           CALL lab1
2           ...
3   lab1:   MOV EBP, ESP
4           SUB ESP, 0x10
5           MOV DWORD PTR [ESP - 0x4], 0
6           MOV DWORD PTR [ESP - 0x8], 10
7           INC [ESP + 0x4]
8           MOV EAX, DWORD PTR [ESP - 0x4]
```

Figure 2.9: *Local variables on the stack*

.

Loops

The last case of retrieving values involves **loops**. In 2.4 a loop is presented which walks through a list of function pointers, calling them one by one. The example starts with fetching a value from the stack into the ESI register, followed by an unconditional jump to lab0. At lab0, the code compares ESI with another value on the stack. If ESI is smaller than the stack value it is being compared with, then the execution continues at lab1 otherwise the code will continue at lab2 (effectively leaving the loop). At lab1 the address at the memory location pointed to by ESI is loaded into register EAX. At line 6, the function pointed to by EAX is called. Finally ESI is incremented with 0x4, which effectively lets the function fetch the next address in the list.

Our analyzer takes the backtracking approach in which it tries to find values for EAX. It is not possible to predict that for instance `JB lab1` has an influence on the value(s) of EAX, therefore we introduce called *loop analysis*. This technique takes the guards on conditional branches into account. A conditional branch depends on the EFLAGS register, which is affected by instructions like TEST, CMP and XOR. So when a conditional branch is found, the most recent preceding instruction affecting EFLAGS has to be analyzed.

```
1           MOV ESI , DWORD PTR [ESP + 0x4]
2           JMP lab0
3           ...
4   lab1:   MOV EAX, DWORD PTR [ESI]
5           ...
6           CALL EAX
7           ADD ESI, 0x4
8   lab0:   CMP ESI, DWORD PTR [ESP+0x8]
9           JB lab1
10  lab2:   ...
```

Figure 2.10: *A loop over a list of pointers*

Id	Situation
1	Register value in the same sequence
2	Register value in multiple paths of execution
3	Stack value
4	Stack value as function argument
5	Stack value as local variable
6	Value changed in loops

Table 2.3: *Backtracking for operands on the IA-32 architecture*

Function calls

When a function is called in the path leading toward an instruction, it can affect stack as well as register values and memory locations. Therefore, we have to look at the effect of functions. They depend on input arguments, so these need to be known before analysis of the function is executed. Therefore we introduces the `Function` class in the EPM.

Summary

In the previous paragraphs we have introduced several situations in which the value of a dynamic operand had to be calculated based on preceding instructions. These situations are summarized in table 2.3. In the following paragraphs an algorithm for retrieving dynamic values is presented.
Dynamic values depend on the path(s) of execution leading toward the locations where the dynamic values are used. Since it is possible to have loops in a path of execution as well as multiple calls to a certain function, it is not possible to keep a single state of the processor for each instruction location. To solve this problem we have two options:

(1) keeping track of a set of processor state-objects for every instruction, when a branch is encountered, the processor state-objects are duplicated for every possible branch. (2) using backtracking (lazy evaluation) to calculate values, only when it is strictly necessary.
Since our analyzer is mainly concerned with retrieving all possible paths of execution, only static and dynamic values used by call, unconditional and conditional jump instructions are needed. It seems like a lot of overhead to calculate processor state objects for every instruction when only a small portion of those values will be used. Therefore we choose backtracking to calculate target addresses.

2.5 Backtracking

In our EPM we have two relationships between instructions and operands that guide our backtracking: `dependson` and `affects`. The `dependson` relationship implies that an instruction can only be executed when the values of the operands that are in the `dependson` relationship with this instruction are available. The `affects` relationship implies that the instruction will modify the values of the operands with which it shares the `affects` relationship.
To calculate the target of a branch, we only have to backtrack for the values of the operands that have a `dependson` relationship with the branching instruction. The most recent instructions that have an `affects` relationship with the required operands are analyzed to yield the values we need, these are called the source instructions. In the process we might have to backtrack the operands that have a `dependson` relationship with these source instructions as well.
During backtracking the analyzer walks the paths of execution that lead to an instruction I in reversed direction. When the analyzer encounters an instruction J that affects an operand on which I depends, then the analyzer tries to retrieve possible values for the operands that have a `dependson` relationship with J. The analyzer stops the backtracking procedure when all operands needed are found. See figure 2.11 for pseudo code.
When all operands are completely satisfied, a forward tracking procedure is started which creates sets of values that satisfy each instruction on the path P found by the backtracker. Finally a set of input values for instruction I is available and the system can calculate target addresses.
After backtracking has completed, a process called tracking forward is started over the path found by the backtracking procedure. It simply takes the first instruction, composes a `ValueInstanceModel` for the different values and takes that model to the next instruction of the path and let that instruction compose its output based on the

```
1  BacktrackValuesInASequence(Instruction I, Sequence S) {
2
3    RequiredOperands  = I.dependson
4
5    while RequiredOperands not Empty {
6      I = Preceding instruction of I
7      if I affects some operand of RequiredOperands {
8        Add I to Path
9        Remove Operands affected by I from RequiredOperands
10       If RequiredOperands is Empty {
11         return TrackForward(Path)
12       }
13       Add I.dependson to RequiredOperands
14     }
15
16
17     if RequiredOperands is Empty {
18       return TrackForward(Path)
19     }
20
21     PreviousSequences = S.previous;
22     for each s = element of PreviousSequences {
23       Add BackTrack(s, RequiredOperands) to ValueModel
24     }
25     return ValueModel
26 }
```

Figure 2.11: *Pseudo code for the backtracking procedure*

input model. Until the last instruction is removed from the path. See figure 2.12.

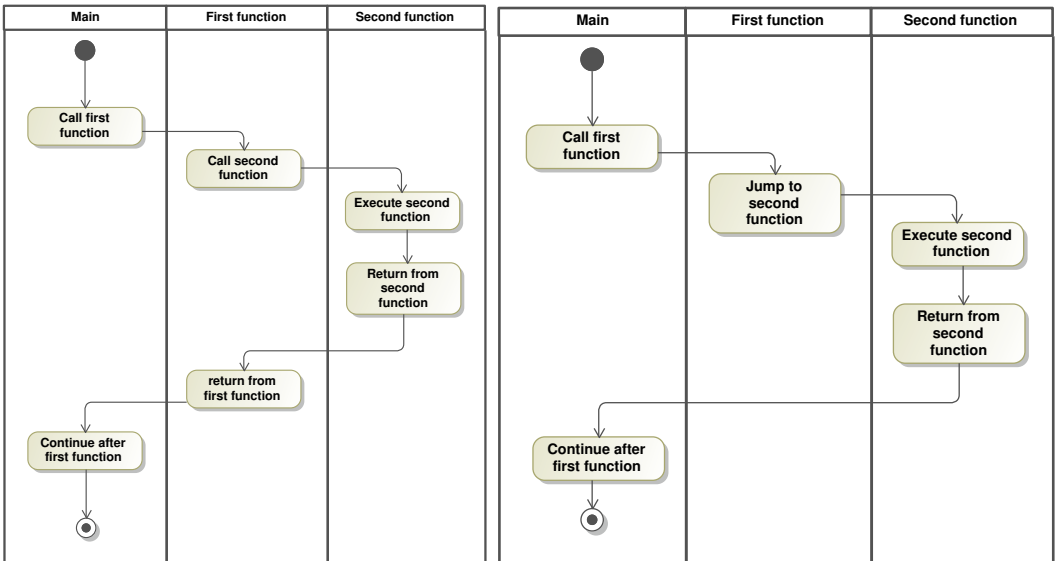
During backtracking we will encounter situations in which a certain instruction becomes a target for another instruction. In such a case the branches and calls following the targeted instruction might be affected as well, in the sense that the number of different paths leading to them increases, effectively providing more opportunities for backtracking to find values. Therefore we have to recalculate targets after the path of execution for a certain instruction has changed.

In certain situations external library functions are executed by a branch to the first instruction. This is done for speed, since a CALL will save the address of the next instruc-

```
1   For each element of path
2           calculate valuemodelinstance recursively.
```

Figure 2.12: *Pseudo code for the forward tracking procedure*

tion (relative to the address of the CALL) on the stack, while a branch just changes the IP register. This technique is only applicable when code is already part of a call and one wants to save the bother of an extra RET and possibly copying return values. So a jump to another function is executed which returns and this directly ends the current call. In figure 2.13 the difference between the two techniques are displayed.



(a) Nested Calls (b) Combined function return

Figure 2.13: *Function nesting*

2.6 Library semantics: a knowledge base

Static analysis is the best approach to application analysis for our purposes (see section 2.1). It can be used to create an EPM, which provides an insight into the workings of the application, but it is not enough to facilitate the mining of Web Service operation candidates. Knowledge about the calling structure of an application is simply not enough to get insight in the higher level purpose of an application. Therefore we need more knowledge about the purpose of the external functions called by an application. Operating systems provide functionality in a lot of areas, such that programmers can be ”creative” at a higher level of abstraction. Ideally the functionality is divided into

categories aimed at specific tasks, such as networking or user interfaces. These functionalities can also be provided by other libraries. These libraries give us information at even a higher level of abstraction than the calls they forward to the operating system.

Semantic information about these functions is needed. For instance a call to a function "CreateWindow" has no meaning by itself, therefore a method to introduce such semantic information for Web Service generation should be implemented. This information is stored in the Operation Model (OM). A knowledge base will contain the "semantic profiles", the conversion rules from function calls to semantic elements.

The knowledge base contains function-models that are aware of argument types and values. With these types and values, as well as the actual instantiation of these arguments (the values actually encountered in an application), specific semantic elements are generated. For instance a function that creates a button would be enabled to generate a *button* element in the User Interface Model (this model is discussed in section 2.8) with the coordinates it actually receives in the application. An other concept that should be captured is the creation of interrelated elements. For instance the creation of a window (`CreateWindow`) returns a handle and such a handle is used to indicate that a certain call to `CreateButton` creates a button for the specified window.

During our research, we can not neglect the importance of function libraries other than the functionality provided by an operating system. For instance Trolltech's Qt® [11] is a commercial cross-platform library that provides functionality in many areas. The same is true for the open source project WxWidgets™ [14]. They capture concepts at a cross-platform level.

Other libraries are more domain specific, for instance financial or statistical libraries. They provide functionality closer to a certain domain. For instance currency conversion will never be provided by an operating system, but a function for currency conversion provides information at a higher level of abstraction than a general conversion from one number to another.

These libraries provide functionality at a different level of granularity than an operating system. By analyzing them as well as operating system functions, we can provide semantics at a higher level or a more specific level of abstraction. Therefore we also implement a higher level semantic analysis for these libraries (mainly consisting of manually matching semantic elements with certain calling structures). Semantics in this thesis are aimed at Web Service operation generation. Therefore, only a small set of functions has to be implemented in the knowledge base.

For instance knowing which buttons and labels are created is not enough to extract knowledge about the relationships between these controls. The relationships are most definitely important for guiding users into the right naming schemes for operation arguments. Therefore we introduce user interface model recovery in section 2.8.

2.7 Operation Model

The knowledge base described in the previous section contains information at a higher level of abstraction than byte code. To prevent cluttering up the EPM with information at a higher level of abstraction than the other elements, we introduce the Operation Model (OM). This model is generated based on the EPM of an application. Semantic information from the knowledge base is extracted, based on the function calls in the EPM, creating a strong basis for Web Service operation mining. It is discussed in more detail in chapter 5.

2.8 User Interface Model recovery

As we established in chapter 1, this thesis is concerned with applications that provide their users with a Graphical User Interface (GUI). We also established that the graphical user interface should be seen as the most important provider of information for Web Service mining (section 1.3). In this section a method for recovering a User Interface Model (UIM) is described.
When an application provides a GUI, multiple methods are available to the application programmer for constructing it. One of the methods consists of building windows and their controls in programming code. Step by step the windows/screens/dialogs are built up, used and finally destroyed when they are not needed any more. A second method consists of distributing a *precompiled resource* with the application binary which represents a whole dialog. The resource is loaded when needed and unloaded when it is no longer in use.
Building dialogs and their controls in program code can be done in many ways. On Microsoft Windows ™ we have seen examples of using the standard libraries for that operating system as well as cross-platform libraries like Qt® and WxWidgets®. These libraries all have different methods for building up a user interface and they use different naming schema's and different sets of properties for each control.
Our abstract representation of the GUI should not gather too many details about the interface. In principle any component can be represented by the *form controls* defined by the XForms specification [20]. We will use the XForms specification as the basis for our own user interface model.
`Triggers` are elements that trigger a certain activity, such as a click on an 'ok'-button or on a menuitem. An `input` defines a component that accepts input of a certain data type, for instance a checkbox (boolean) or a text input. An `output` is a control that outputs information which is not intended to be changed by a user, such as an icon.
In figure 2.14 a model is presented to capture an abstract representation of an applica-

tion's graphical user interface. A user interface consists of dialogs or windows (from the perspective of the user) that contain controls like buttons, text areas and labels. These controls all have some attributes in common, such as *location* and *boundaries*. Therefore a general `Control` class is introduced, which is a more abstract version of all other window controls. We abstract away the difference between a text `Input` and a `Textarea`, since we are not interrested in rebuilding an interface. We also abstracted away the difference between a `Submit` and a `Trigger`. The control classes can be grouped together in a `Dialog` class, this is a container for Components and it should not be confused with the Dialog component in user interface libraries.
We distinguish `output` from a `label`, because the purpose of a label is informational in our model, while the output component implements a method for returning information to a user.
This forms the first version of the UIM as reasoned from the application's point of view. To make the model more useful for our purpose we have to incorporate a mechanism to extend the abstraction level of the model in such a way that it can be used for the generation of Web Service operation candidates.

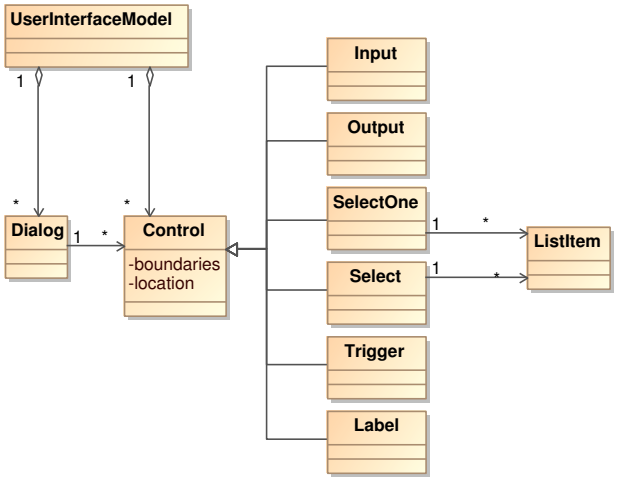


Figure 2.14: *User Interface Model*

Extending the UIM

A point of information for developing a Web Service is the combination of a field name with the field type. In user interfaces, this is generally represented by a label component placed near an input component for which it is intended to provide extra information. There is no association defined between the two. If we want to generate Web Service

operation candidates, we have to find a method to associate the non-label controls with the right labels.
In [6] a method for the recovery of user interfaces with a focus on finding the correct combinations of label and input components is described. The method requires a set of facts (the GUI components, their coordinates and their types) and then it uses conflict resolution to find the optimal set of combinations of components and labels. From that set of combinations, we could generate the operation candidates.
To incorporate this information in the UIM, we have to extend the UIM a bit, of which the result is shown in figure 2.15. Three classes are added to the original UIM : the `CaptionedControl` associates a Control with a Label (the model allows a Label associated with another Label, but in practice it will not be used). The `Phrase` groups multiple Controls together and the `Section` which groups Phrase and CaptionedControls together.
With the extended UIM in place, we are able to easily generate data types for Web Service operations. A Section corresponds to an operation candidate as we will see in chapter 5.

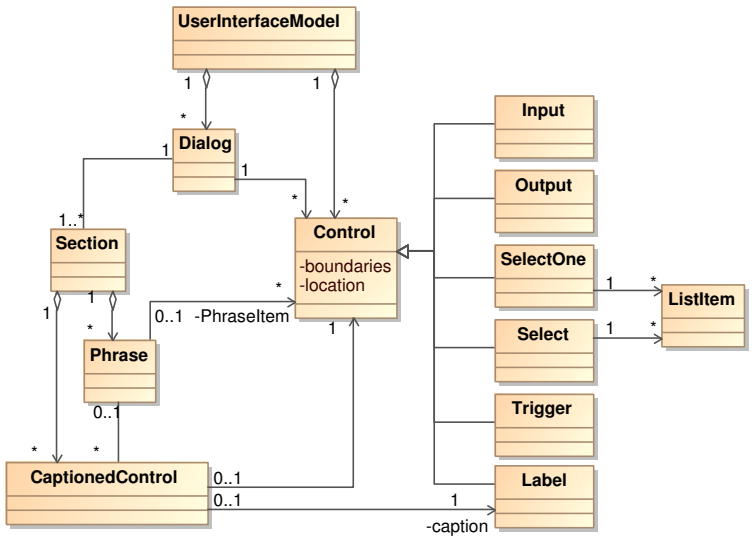


Figure 2.15: *User Interface Model - extended*

2.9 Putting it all together for integration

During our research we keep in mind that we are trying to provide functionality as Web Services operations, therefore we need to focus on situations where a certain

output is generated based on a certain input. These can be data centric functions, like storing and retrieving data. But we can also come across execution functions, such as a Sum-operator that calculates the sum of several values. Or a data conversion service which converts an amount in Euros to an amount in United States Dollars.
For our project we want to select functionality that can be reused through Web Services, without being too fine grained or maybe too coarse grained.
For instance the addition of a record and its contents to a database through a graphical user interface is a set of keyboard and mouse events, window create/destroy (or show/hide) events and several checks on constraints. Also the interaction with a database (in another process) is part of such a function.
While the combination of binary analysis and user interface remodeling provides a good source of information, the information found can be quite overwhelming.

Chapter 3

Controlling applications

How does the sea become the king of all streams?
Because it lies lower than they!
Hence it is the king of all streams.

Lao Tzu

Being able to analyze application binaries is one thing, but putting the information gathered to use is a whole different story. Depending on the requirements on the behavior of our tool, several possible solutions exist for externally influencing application behavior. These possibilities range from injecting messages into the application's message queue(s) to modifying the original application binary. In this chapter we provide the requirements on the influencing part as well as an overview of possible solutions.

3.1 Controlling application behavior

When the analysis generates a WSMProfile, we need to provide a method to implement the Web Service. This involves receiving invocations on the Web Service operations and forwarding them to the application. When an invocation results in output, it needs to be returned to the caller of the Web Service operation. A mediator between the application and the Web Service endpoint is needed, this system should interact with a running instance of the analyzed application.
A method for controlling application execution is needed. For this purpose, several techniques are available, these are listed in the following sections. In the last section of this chapter an overview of the techniques is provided.

3.2 API hooking

Operating systems provide services to applications. On Microsoft Windows™ the services are provided by dynamic linkable libraries (DLL's) that are loaded into the private address space of an application. On other operating systems, similar techniques are used.

Events issued by user interface components, keyboard, mouse and other sources are presented to an application as messages. To act on the messages, every process has an event loop which waits for these messages.
If our system has to control an application by entering messages, it could use API hooking to control the message retrieval function of the operating system. With this technique, the message retrieval function, mentioned earlier, is replaced by a new version that injects these messages into the application. The function is replaced by taking advantage of the following (at least on Microsoft Windows™): any byte in the private address space of an application can be directly overwritten by processes that have access to the private address space. When this is actually done, it is called API hooking.

3.3 Detours

Detours [15] is a library for instrumenting arbitrary Win32 functions on x86-machines. At runtime, Detours replaces the first couple of instructions of a function with a unconditional jump to a detour function. This Detour function consists of the code one wants to insert and a return to the caller of the detoured function. If necessary, it is possible to execute the original function as well. For this purpose, the Detours library provides a *trampoline function* which is capable of executing the original function which has been 'detoured'.

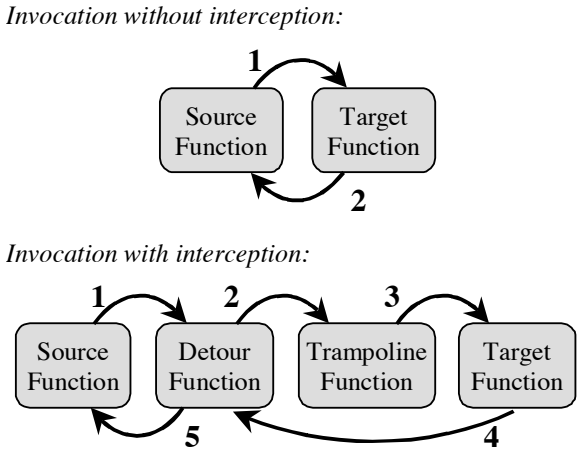


Figure 3.1: *Invocation with and without detours.*

In our system we could use Detours to instrument the equivalent of WIN32 `GetMessage` in the main message loop. We could replace it with a function that injects messages to execute the functions requested by Web Service operations.

Detours are inserted at execution time, which implies that they do not change the original application binary. The library is capable of changing binaries, but that is only additional functionality, not the main bit. A problem is that the library is only available for Win32, but the concept is quite simple and it should be no problem to achieve a similar technique on other platforms.

3.4 Breakpoint Trapping

Our WSMExecuter could insert breakpoints at crucial places in an application binary. When we provide a debugging exception handler, the exception handler can invoke the required functions. A major drawback of this technology is that debugging exceptions suspend all application threads. An other thing is that a second operating system process must catch the exception. This implies a high performance penalty.
Breakpoint trapping and Detours are more of a reactive type of interception. They wait for an event to happen and then they come into action. In our system we want to direct the application such that it executes functions as we intend them. Therefore Detours and Breakpoint trapping are less useful for us.

3.5 Process Injection

Process Injection is a method that ”injects” (sends) messages into (to) a process to which it has access. By sending keyboard and mouse messages, the remote process mimics a user interaction. This technique is non intrusive : nothing of the application code is modified at all (not even at runtime). A drawback is that not every process is allowed to inject messages into other processes. It is possible to circumvent this drawback by starting the required process from the injecting process, effectively becoming the owner of the target (application) process.

3.6 Loadable Module

Modern operating systems allow extensions to the kernel from other parties than the operating system manufacturer. They provide a system for modules to be loaded into kernel space, (on Microsoft Windows, these loadable modules are called ”Device Drivers”). By creating a *module* and loading it, it is automatically placed in kernel space. When functions of the device driver are executed, they have the highest level of access to other elements, like processes and memory. With the privileges gained at that level, we can completely control the execution of an application, ideal for our purposes.

A drawback of this technique is that a bug in the device driver can make a whole operating system in-stable. Another drawback is that it is necessary to switch between user space and kernel space, which implies a large performance penalty. Also two different pieces of software have to be written that are synchronized.

3.7 Dynamic instrumentation

Dynamic instrumentation is a method for monitoring and changing behavior of an application dynamically at multiple levels of granularity (instruction, basic block, routine, etc.). DBI frameworks (see section 2.1) make it quite easy to implement dynamic instrumentation solutions. The technique imposes quite some overhead, since instrumenting at a certain level of granularity generally implies monitoring of anything at that level of granularity. An interesting aspect of this technology is that the original executable is not altered at all. Examples of DBI frameworks are Dyninst, PIN and Valgrind.
In "Controlling Program Execution through Binary Instrumentation" [12] a method is described for altering the execution path of an application through binary instrumentation, specifically for PIN. Although there is a large performance penalty for the functionality, it is an interesting technique for our project.

3.8 Binary rewriting and editing binary files

Binary rewriting involves changing an application's byte code. By changing functions of an application or changing some library or operating system function, we can gain access to specific pieces of functionality that interests us.
By using this technique it is possible to interact with another application that sends instructions. A big drawback is that this technique requires that an application file is altered, but it has a better performance than DBI frameworks.

3.9 Requirements on application control

To select the best option available, we need to check for requirements on our system. The technology that is best suited is then selected. A first requirement is that we will not change (part of) the byte code of operating system functions. Our system should not break down when an OS-patch changes functionality that we had instrumented earlier. There is also no need for such a strategy. Replacing a whole function at once is no problem.

A second requirement is that the file that contains the application binary is not modified, the application should still be working in stand alone mode and some applications put their storage relative to the application file. When the application runs stand alone, it should contain all data that has been entered manually as well as the data entered via Web Service interaction. It would be a bad idea to have two different data sets for the same application.
For our situation, `Process Injection` is the best solution. It is no problem to become the owner of a process, since there is no reason why our WSMToolkit should not be allowed to start an application in a sub-process.

Chapter 4

Scenarios

The question is not what you look at, but what you see.

Henry David Thoreau

In the previous chapters the basis for our research is introduced. In the current chapter we are providing a scenario in which our proposed solution can be used. It is the introduction to the models presented in chapter 5 and the concrete design of our solution in chapter 6. Finally our scenario will be used as the test case for our prototype implementation as described in chapter 7.

4.1 A potential case

At Evolved, a small production company, the only account manager employed uses Windows Address Book (WAB) as the main tool for storing customer contact information. Due to an increase in the number of customers, a new account-manager is employed. The new account-manager needs access to the customer information recorded by the first person as well as a way to add new customer information. Management does not want a situation in which information has to be synchronized between applications all the time, so they want a centralized solution. They are afraid to convert the internal WAB-data to some other format, so ideally WAB is used as a central storage for customer information. They want to use a single-user application as a centralized multi-user application.
Joe is hired to implement a solution for this situation. He has some experience with the WSM Toolkit and he knows that it will provide the best solution for the situation at hand. He installs the toolkit on the desktop computer of the first account manager. He starts the WSMiner and opens wab.exe with it. WSMiner is analyzing for several minutes and it shows a set of potential Web Service operations. Joe selects the ones he really needs for the situation at Evolved and he modifies some of the parameters for the Web Service operations. Finally he exports the set of selected operations to a WSMProfile.
Now he starts testing the accuracy of the WSMProfile. He starts the WSMExecuter with the WSMProfile just generated. The WSMExecuter starts the Web Service and it

generates a WSDL file. Joe also generates an application that interacts with the Web Service, based on the WSDL file. He loads the WSDL file in his test-application and tries the Web Service. It seems to work fine, so he distributes the new application and WSDL file to the computer of the new account manager. He also installs it at the computer of the first account manager.
Joe created a useable integration of data in under an hour and management is thrilled with the result.

4.2 Expected results

The automatic Web Service mining of `wab.exe` is the goal of our prototype implementation of the WSM Toolkit. By using the general concepts developed in this thesis, the prototype should at least be able to do so. In this section we are looking at the expected results of the automatic analysis of `wab.exe` so we can compare these with the real results and give a qualitative comparison of them.
Many functions of `wab.exe` that are accessible through its interface are not interesting for Web Service access. For instance the *help* and *about* menu-items should not be considered as good examples of Web Service operations. They are both static elements that do not provide any information regarding the main purpose of the application.
Better options are the creation of new groups, contacts and directories. Also the removal of contacts and the possibility to change them are good examples, and last but not least: the search for contacts.

Chapter 5

Models

Such a life, with all vision limited to a Point, and all motion to a Straight Line, seemed to me inexpressibly dreary

Edwin A. Abbot

Our system has to convert an application binary into a concrete WSDL file together with a description on how to process requests to the Web Service. This task is divided into several subtasks that involve building an integrated application model, converting this model into a Web Service description model and creating execution profiles as well as a WSDL file. In this chapter the models and rules for conversion from one model to another are described.

5.1 The Execution Path Model

During static analysis of application byte code, the bytes are decode into instructions and paths of execution in the application are recovered. To facilitate easy access to the information recovered, we introduced a model called the *Execution Path Model (EPM)* in chapter 2. It contains relationships between instructions and possible paths of execution.

In short it contains the processor instructions that make up the application and their operands. The instructions are grouped together in sequences that can be part of loops and functions. A visual representation can be found in figure 5.1, this is a copy of the model in chapter 2, figure 2.3.

In general the static analyzer has a main loop that analyzes instructions up to a branch. At such a branch the analyzer calculates and stores each branch target. We have to take into account that at each branch we might have multiple targets to analyze, due to the operand values that point to the 'next' instruction. Therefore we store source and target to generate links.

Functions consist of a single first instruction that starts a sequence that can be analyzed up to the RET instructions.

Often internal branching behavior and calling behavior of a function depend on the inputs it receives. Therefore it is impossible to fully analyze such a function without

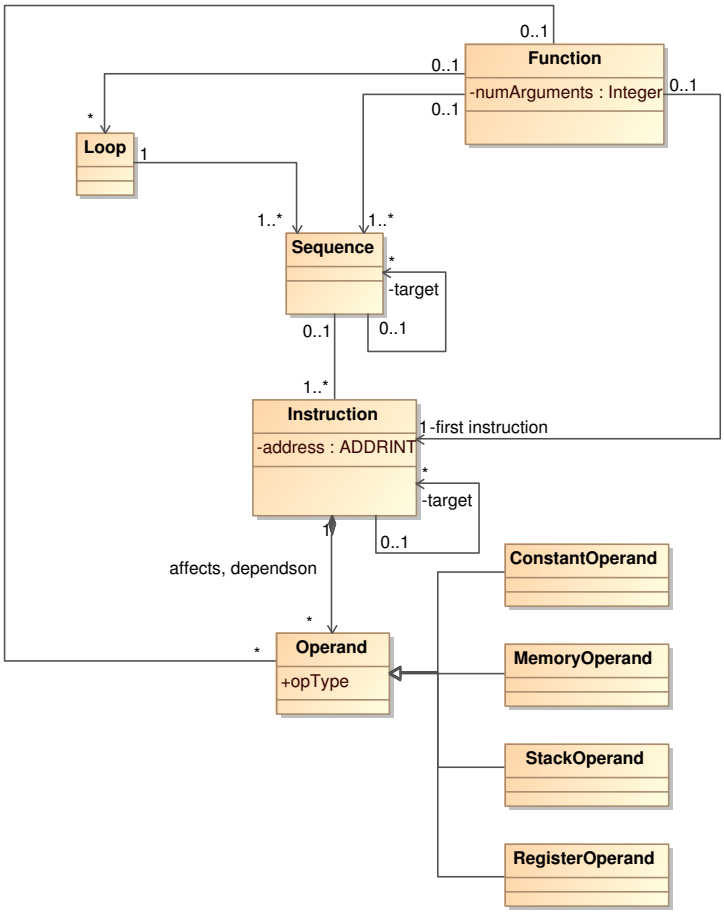


Figure 5.1: *Execution Path Model (EPM)*

knowledge of the situations in which it is executed. Therefore we record callers.
In the end our analyzer has traversed every possible path of execution of the application. It created an Execution Path Model in the process, which is the input for the next analysis step.
In the next section the OM is discussed as well as a method to extract an OM from an EPM.

Generating an EPM

The addition of a link between two instructions (implying that the target instruction can directly follow the source instruction during execution) to the Execution Path Model initiates a lot of activity. This is due to the fact that the `Sequence` and `Loop` objects contained by the Execution Path Model and their relations need to be updated.

Function	Description
addInstruction	Adds an instruction to the end of the sequence
hasNoTargets	returns true if the sequence has no target sequences
isFirst	Returns true if the instruction address is the first in the sequence
isLast	Returns true if the instruction address is the last in the sequence
containsAddress	Returns true if the instruction address is part of the sequence
getAddresses	Accessor for the internal ordered list of instruction addresses
getTargets	Returns pointers to the sequences directly following the sequence
splitAfter	Splits a sequence after a certain instruction into two sequences, the newly created sequence that formed the tail of the sequence is added as target to the original sequence
splitBefore	Splits a sequence before a certain instruction into two sequences, the newly created sequence that formed the tail of the sequence is added as target to the original sequence

Table 5.1: *Functions for the Sequence component*

A `Sequence` object provides several functions to the EPM to update its state. They are listed in table 5.1.

A `Function` object provides additional information about an instruction that is the first instruction of a function and a set of instructions (and the sequences and loops defined upon them) that form the function. To be able to retrieve function argument values that are pushed onto the stack before a certain function is called, the Function object also records the addresses of instructions that point to the function. A Function contains `affects` relationships with memory and registers. These relationships are dynamic and depend on the caller of the instruction. This is due to the fact that for instance the allocation of memory affects memory depending on the size of the memory that should be allocated, as indicated by an argument.

A `Loop` instance is a collection of Sequence instances that form a loop. A Sequence can be part of multiple loops, for instance an outer and an inner loop sharing a Sequence object.

Converting EPM to OM

The generation of an OM based on an EPM is quite trivial. The process starts with the first instruction of the application. From that instruction on every path of execution is traversed while recording the interesting function calls. Loops as well as splits and joins in the execution paths are used to distinguish between the different windows

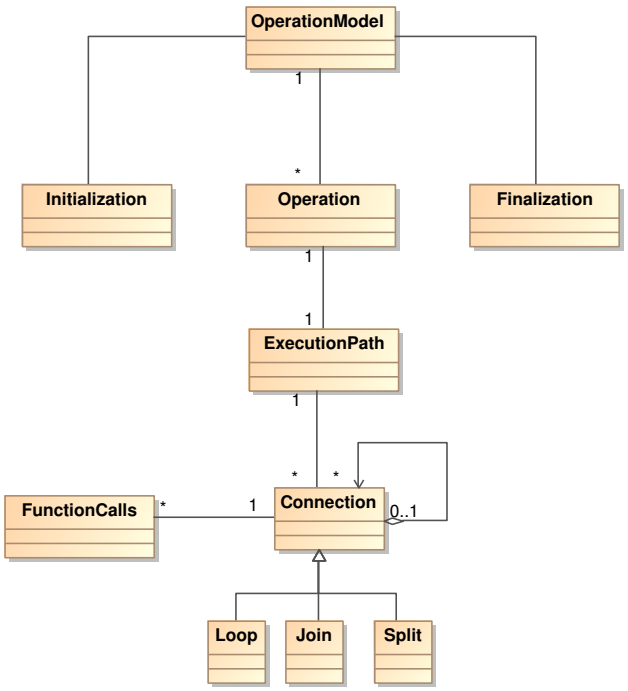


Figure 5.2: *Operation Model (OM)*

that can be generated during application execution. Afterward the different commands that lead to the generation of windows are detected, such that it is possible to generate events in the WSMExecuter on the invocation of a Web Service operation.

5.2 The Operation Model

In the step described in section 2.3, a binary application is analyzed statically. The static analysis yields an EPM, which is converted into an Operation Model (OM), the source for the generation of a WSMprofile.
The OM bridges the gap between the EPM and actual Web Service operation descriptions. The EPM provides a way to recover the paths of execution supported by an application. The OM describes the application at a higher level of abstraction than the EPM, the analyzer uses a knowledge base for the library functions encountered.
The knowledge base provides information about the library functions that are used by the application. The specific library functions used and in which order they are used, form a semantic model for the application, which helps in the creation of a UIM and ultimately with the creation of a WSM.
The OM contains function calls, their arguments and results, their location in the byte

code and, most important, their relation to other function calls. For instance when a window is created by a library function, a handle to the window is returned. This handle is used to add controls to the window. By keeping track of the handles, the OM can be built easily such that later-on it is possible to extract a UIM.

The OM records the byte locations of the instructions that call the functions it contains. With these byte locations, the application can easily be instrumented. While the application will not be controlled by instrumentation, we need it to find out if everything is working as expected.

5.3 The User Interface Model

The User Interface Model (UIM) groups together user interface components. The components and the relations between them define how possible Web Services might be constructed. Without the UIM it is impossible to mine Web Service operations in a sensible way. Our model is loosely based on the XForms specification, this is due to the fact that the XForms specification is developed to provide an abstract model for user interfaces.

The XForms Model contains *form controls*, the elements that empower a user to input data as well as a means to output information to a user. Several elements to group these form controls together are *switch*, *repeat* and *group*. The grouping elements are themselves contained in an *instance* element that is part of the XForms *model*. We are not interested in *submission methods* or *bind* elements, because our model is only intended as an intermediary form of application information, not for implementation/execution.

In XForms the form controls consist of a user interface control as well as a label. This is the property that we are looking for (next to the abstract model), since we want an easy method to express a relationship between a control and a label, which could be used to generate the names for Web Service operation arguments.

The grouping property is equally important, since it can guide us in generating complete and meaningful Web Service operations: each grouping is a potential Web Service operation.

The UIM describes the user interface of an application in an abstract model. The model is based on the XForms specification, since XForms has a similar goal as our model. Since the controls (or widgets) imply a certain data model with constraints on the inputs, it is not enough to only use the `instance` element of the model, but we will also need a set of `bind` elements.

Generating an UIM

Converting an OM to a UIM requires that we first traverse every path of execution that is considered an Operation. By using the knowledge base, the effect of a function on the UIM is added.
After OM to UIM conversion, the UIM is traversed to find the CaptionedControls as well as the Phrase components. These are then used to generate `types` for the Web Service. In figure 5.3 the extended UIM is provided.
We are also faced with the fact that dialogs can be strongly related. For instance entering information about a person into a customer relationship management system might require filling in multiple forms that are interrelated. Or dialogs with an informational popup after pushing ”ok” or ”cancel”. Such a case is not a part of MORE so we extend the technique at that point.
A whole other problem consists of single dialog applications. These have all their inputs on one dialog making it very hard to make a sensible grouping. In such a case we can only rely on a visual interpretation that can take style properties into account, for instance a border grouping four input widgets and their captions together.
Another aspect of the user interface is provided by the menu. In general, the most important functions of an application are accessible through the menu of a window. By analyzing the menu and its event handlers, a better idea of the application functionality for user interaction is gained.

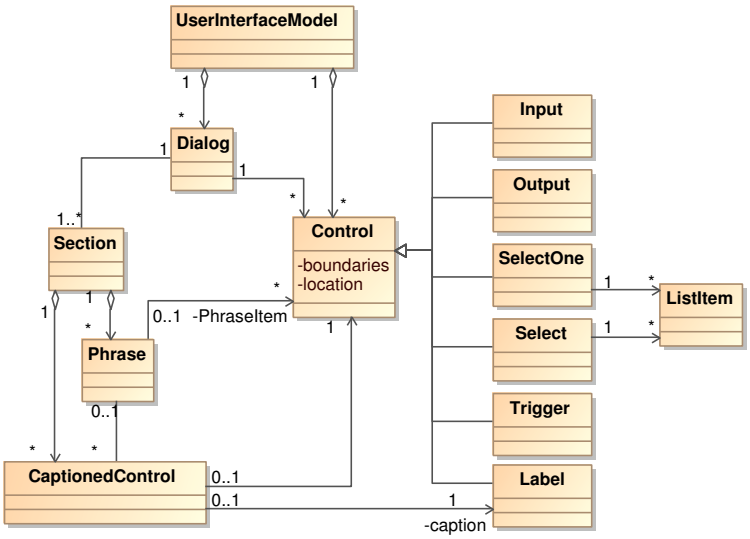


Figure 5.3: *User Interface Model - extended*

5.4 The Web Service Model

Detecting possible Web Service operations is the main goal of analyzing applications. In our solution, several models are generated to contain certain conceptually related aspects of an application. The analysis should result in a WSMProfile that contains any information necessary to implement Web Service operations. The Web Service Model (WSM) (see figure 5.5) is the intermediary model that captures the essence of an application to generate the WSMProfile. It is tightly coupled to WSDL, UIM and OM.
At a high level, the WSDL-specification consists of a container (the `description` component) for two categories of components: WSDL 2.0 components and type system components. The WSDL 2.0 components consist of *interfaces*, *bindings* and *services*. Of these, the bindings and services are mostly run-time related, therefore the responsibility for correctly generating them lies with the WSMExcuter. Interfaces, on the other hand, should be contained in the Web Services model. They are closely related to application structure.
Type system components describe the constraints on a message's content and as such they are closely related to the application. The WSMiner should provide the contents of this part of the WSDL specification through the WSM.
The *type definitions* property of the main *description* component is required. Because the WSDL 2.0 definition only defines the use of XML Schema (although a number of alternate schema languages can be used), we constrain ourselves to XML Schema as well. Therefore Web Service operations can be expressed as a complexType element containing all the relevant fields and names.
When a resource is used, we still need to look around the usage of the resource to find the initialization information for the components.
The WSDL 2.0 InfosetModel (see figure 5.4) provides a nice starting point for our internal model which will contain the Web Service operation candidates as well as knowledge on how to implement these operations.
The UIM contains CaptionedControls, these form the base for Web Service operation candidates. The label of a CaptionedControl is used to generate a unique field-name. The control is used to determine the data type for the field. The Section is used to generate the integrated types for each operation, it simple generates a ComplexType containing every field.
The WSM has to point in the right direction to control execution of a program. Therefore we have to take user interface features together with instruction locations. As we saw earlier, it is best to stay as close to the original application execution as possible. We can't bypass user interface components altogether since they can influence the values received form components. Therefore we have selected an execution technology in chapter 3 that controls an application just like a macro-recorder/executer would do.

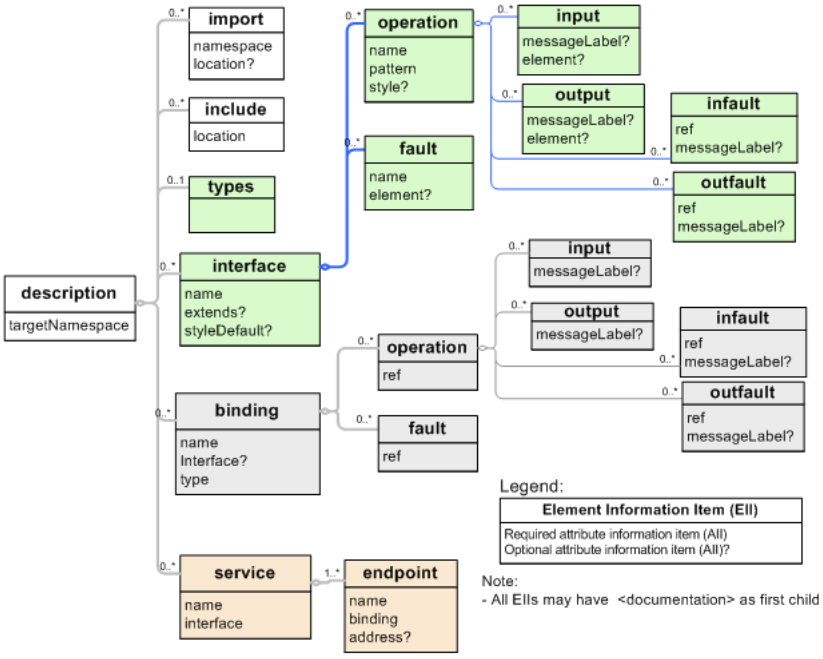


Figure 5.4: *WSDL 2.0 infoset model*

This implies that we need to record at least two features of an application: the commands used to show a certain screen and a way to modify the contents of the controls on screen. These features will be recorded in to the operation model. The operation model will contain every aspect of building up the user interface components and the method for entering information into a component.

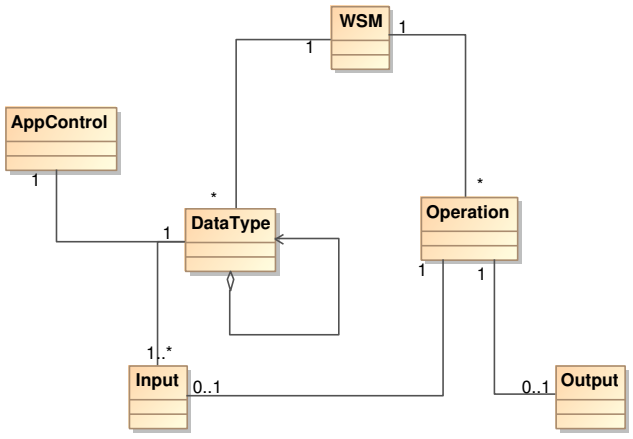


Figure 5.5: *Web Service Model (WSM)*

Chapter 6

Design of the proposed solution

Day by day, and almost minute by minute the past was brought up to date. In this way every prediction made [...] could be shown by documentary evidence to have been correct.

1984- George Orwell

In chapter 2 we established that a static disassembler is necessary to draw a complete picture of application behavior. The behavior is analyzed to get Web Service operation candidates. In this chapter we are providing a design for our solution. In this chapter the implicit requirements from the earlier chapters are brought together and made explicit.

6.1 Stakeholders

Several stakeholders are involved in both the design, maintenance and usage of our system. To create a system suitable for all stakeholders, let us consider them:

Operators

An operator in our system is a person who uses the WSMToolkit to generate a WSM-Profile with the WSMiner. He or she loads an application to analyze and tweaks the results before actual usage. After the analysis, the operator is also responsible for the correct installation of a Web Service endpoint by the means of using the WSMExecuter. Operators are mainly concerned about the usability of the system and the correctness of results.

Web Service users

After installment of a Web Service by the operator, the people who use such a Web Service (directly or by the means of a Web Services enabled application) are the *Web Service users*. These people are concerned with easy access to the Web Service and the right naming schema's. Performance is also one of their concerns.

WSM Implementers

The implementers of the WSMToolset are responsible for the correct implementation of the application software. They are concerned with the maintainability of the software as well as the extensibility.

6.2 Key design drivers

Designing a system that is generic enough to capture the essence of any type of desktop application, but at the same time versatile enough to recover minute details about these applications is not an easy task. Many details need to be taken into account and at many points, alternative solutions must be considered.
We define a small number of main principles to which our design shall adhere: the key design drivers. For our system, the following key drivers are defined: extensibility, maintainability and portability.

- To meet the demands of abstracting away the details of many libraries with their own peculiarities calls for *extensibility*. During its lifetime, the WSMToolkit will be extended with knowledge about new libraries, changed library semantics and more.

- During the extension of the functionality of the application, the source code will have to change often. To make sure that the updating of source code goes smoothly, *maintainability* is important to consider.

- As said before, our system is aimed at multiple platforms, therefore *portability* of its source code is a design driver.

6.3 User requirements

Users of our system (the operators) need certain functionality to perform the tasks for which they use the system. The user requirements for our system are summarized in table 6.1.
First of all the system should provide a user with a means of selecting a certain application for analysis (UR01). Since automatic analysis can not be expected to be completely correct and it might generate operation and variable names that are not entirely descriptive, the system has a method for altering the analysis results to better suit the situation (UR02). When analysis has completed, the results (with possible alterations) can be stored for future reuse and improvements (UR03).

No.	Requirement Description
UR01	The WSMToolset shall provide a means to select external files for analysis
UR02	The WSMToolset shall provide a means to modify names of operations and arguments in the analysis results
UR03	The WSMToolset shall have a method for storing analysis results
UR04	The WSMToolset shall provide a means to start the Web Service that resulted from the analysis
UR05	The WSMToolset shall not bind the analysis results to a specific installment of an application
UR06	The WSMToolset shall provide an extensible knowledge base for library semantics
UR07	The WSMToolset shall provide a help system

Table 6.1: *User Requirements for the WSMToolkit*

When analysis has been completed, a user probably wants to start the Web Service for testing and, later on, for actual usage (UR04). An analysis profile can be shared with others to make sure that any application of the same type provides the same Web Service operations (UR05).
The knowledge base of the system is extensible, such that operators can extend or correct the behavior of the analyzer (UR06). For a correct usage of a system, the users have to be informed about the exact workings of functionality of our system. therefore we provide a help system (UR07).

6.4 System Requirements

Requirements on the system are more focussed on the internal workings of the WSMToolset than the user requirements, they also provide more specific requirements on the functionality stated in the user requirements. In table 6.2 an overview of the system requirements is provided.
The WSMToolset has a means of finding the internal structure of an application by disassembling its byte code (SR01). This information is used to find Web Service operation candidates through the analysis of user interface features (SR02). As stated in UR03 the system shall be able to export information (SR03). To improve analysis results, the WSMToolset contains information about the functions provided by operating systems and external libraries at a high level of abstraction (SR04). Due to UR05, the WSMToolset consists of two separate applications: the WSMiner which analyzes application binaries (SR05) and a WSMExecuter (SR06). The WSMExecuter forwards requests for

No.	Requirement Description
SR01	The WSMToolset shall disassemble an application file statically
SR02	The WSMToolset shall analyze user interface features to obtain Web Service operation candidates
SR03	The WSMToolset shall export relevant information for reuse
SR04	The WSMToolset shall replace external functions with pre-compiled function templates, contained by the knowledge base
SR05	The WSMToolset shall provide a separate analysis application
SR06	The WSMToolset shall provide a separate execution environment
SR07	The WSMToolset shall be able to set up an Web Service endpoint
SR08	The WSMToolset shall be able to inject requests for its Web Service operations into an instance of the application it has mined
SR09	The WSMToolset shall be able to retrieve results from injected requests

Table 6.2: *System Requirements for the WSMToolkit*

a Web Service operation that are received on its own endpoint (SR07) to an application binary (SR08). The results of an operation are retrieved and send back if necessary (SR09).

6.5 WSMToolkit

Based on the system requirements presented in the previous section, the system has been split into several packages. There are packages for the **WSMiner** and the **WSMExecuter** as well as a package for the handling of the **WSMProfile**. The WSMProfile package is introduced because both applications in the WSMToolkit have to deal with the WSMProfile and its different instances (as a file or as a model in memory). In figure 6.1 the partitioning and some higher level constructs are shown.

The WSMToolkit also employs a **Decoder** package. This is due to the fact that our system uses third party byte decoders for each processor architecture. To keep the WSMiner design as clean as possible (see the "maintenance" design driver), the peculiarities of each decoder is hidden by the Decoder package. The same reason leads to the SOAPEngine package in our design: third party implementations only require a wrapper.

Furthermore, the models described in chapter 5 are each represented by their own package. These will not be further discussed in this chapter, since chapter 5 is already extensive and complete.

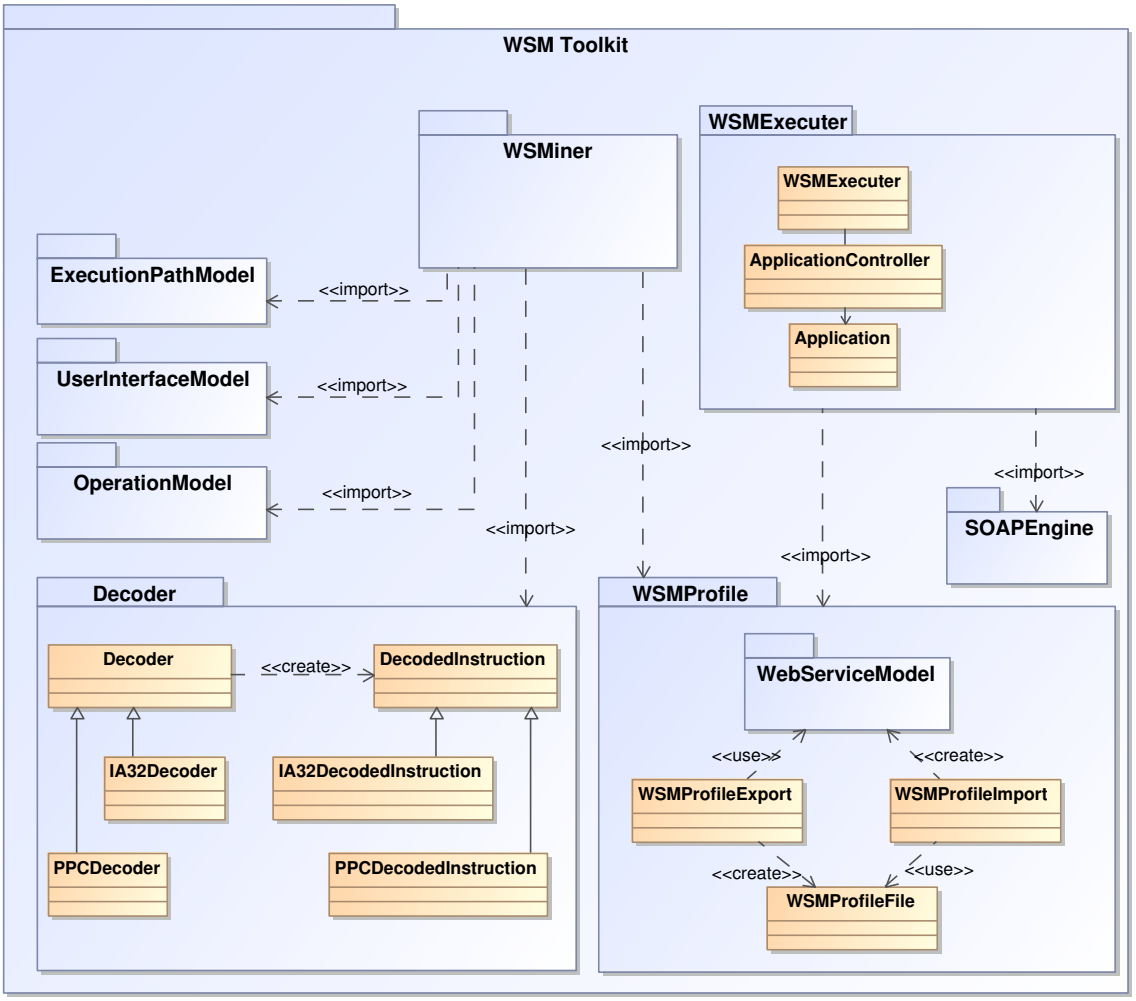


Figure 6.1: *Partitioning into packages*

6.6 WSMiner

The WSMiner is the part of the WSMToolkit that analyzes application binaries and extracts a set of Web Service operations that can be invoked on the application binary by the means of the WSMExecuter. A discussion about the analysis process can be found in chapters 2 and 5. We provide a concrete design for the WSMiner in this section. Its requirements can be found in table 6.3.

The WSMiner accepts an input file that represents the application binary (RM01) in a specific format intended for a certain platform (RM02). Static binary analysis (RM03) is used for analyzing the byte code of the application and building a model (EPM). Multiple processor architectures are accepted (RM04) by the means of the `Decoder`

No.	Requirement Description
RM01	The WSMiner shall input an application binary file
RM02	The WSMiner shall accept multiple file formats
RM03	The WSMiner shall use static binary analysis
RM04	The WSMiner shall accept byte code for multiple processor architectures
RM05	The WSMiner shall have an extendable knowledge base for function profiles
RM06	The WSMiner shall use handcrafted function profiles for library functions
RM07	The WSMiner shall recover a user interface model for an application
RM08	The WSMiner shall generate Web Service operation candidates
RM09	The WSMiner shall provide functions for changing the Web Service operation candidates
RM10	The WSMiner shall output a WSMProfile for use by a WSMExecuter
RM11	The WSMiner shall decode byte code correctly
RM12	The WSMiner shall distinguish between a (un)conditional branch to a function and an actual call to that same function
RM13	The WSMiner shall distinguish between loop analysis and sequence analysis

Table 6.3: *Functional Requirements for the WSMiner*

package.

User interface artifacts can be used in many ways by an application, in general by the use of external libraries. Therefore the system provides an extendable knowledge base (RM05) that provides a higher level of abstraction (than byte code) of the found functionality. The functionality is captured by function profiles (since generation of user interface components is done function-wise, generally) that are handcrafted (RM06). There does not already exist a higher level semantics knowledge base that we can use.

With the information provided by the knowledge base, our application generates an Operation Model (OM) which is an intermediate model for the generation of a User interface Model (UIM) as we have seen in chapter 5. This is done by traversing the execution paths and using the function profiles.

The function profiles are used to recover a user interface model (RM07) for an application. The UIM is used to find Web Service operation candidates (RM08). The WSMiner provides a user with the means to alter these operation candidates (RM09).

In the end the operation candidates are exported as a WSMProfile (RM10) for the actual creation of a Web Service for the application.

Of-course the WSMiner consists of many components that work together to provide acceptable results. The different requirements mentioned before are implemented by a multitude of components that provide specific functionality.

Components of the WSMiner

The input application binary (RM01) can be of multiple file formats (RM02) it therefore is parsed by a `Parser` component that is a generalization of more specific parsers like `PECOFFParser` or `ELFParser`.
The static analysis (RM03) is performed by an `Analyzer`. Since we will accept byte code for multiple processor architectures (RM04), the `Analyzer` uses a byte code `Decoder` that is a generalization of more specific decoders like `IA32Decoder` and `PPCDecoder`. Classes concerned with decoding byte code are placed in the Decoder package.
The knowledge base (RM05) is captured by the `KnowledgeBase` component. This component is a composition of `KBLibrary` components that capture knowledge about libraries. The `KBLibrary` components are compositions of `KBFunctionTemplate` components that capture specific information about a certain function provided by a certain library. To make the Knowledge base extendable (RM05), the knowledge base provides a file format that can be imported.
A User Interface Model (UIM) is built by the WSMiner to have a higher semantic abstraction of the application. The KnowledgeBase is able to generate elements of the UIM.
When the WSMiner is started, it loads an application binary file into a parser for the file format (on Microsoft Windows this would be PE/COFF format). With the information provided by the parser, the address of first instruction of the application is identified. That address is provided to a static analyzer which generates an EPM for the input file. After static analysis the system generates an operation model (OM) of the application. With the OM it finds out how the user interface for the application is generated (by library calls or loading a dialog resource) and it starts building a user interface model (UIM).
Finally the different models are put together as the Web Service model. With the (almost) complete information about runtime behavior of the application as well as the user interface abstraction in place, candidate Web Service operations are generated. A selection procedure (possibly guided by the user) filters the results to get a set of operations that are to be provided as a Web Service. Finally a WSMProfile is generated.
In figure 6.2 the design of the WSMiner is presented. In the following sections the different components are described extensively.

Disassembler

The `Disassembler` in our design is a component that provides services to disassemble an executable file and generates an `ExecutionPathModel` component based on its contents. It provides a function *loadFile* to load an executable file and a

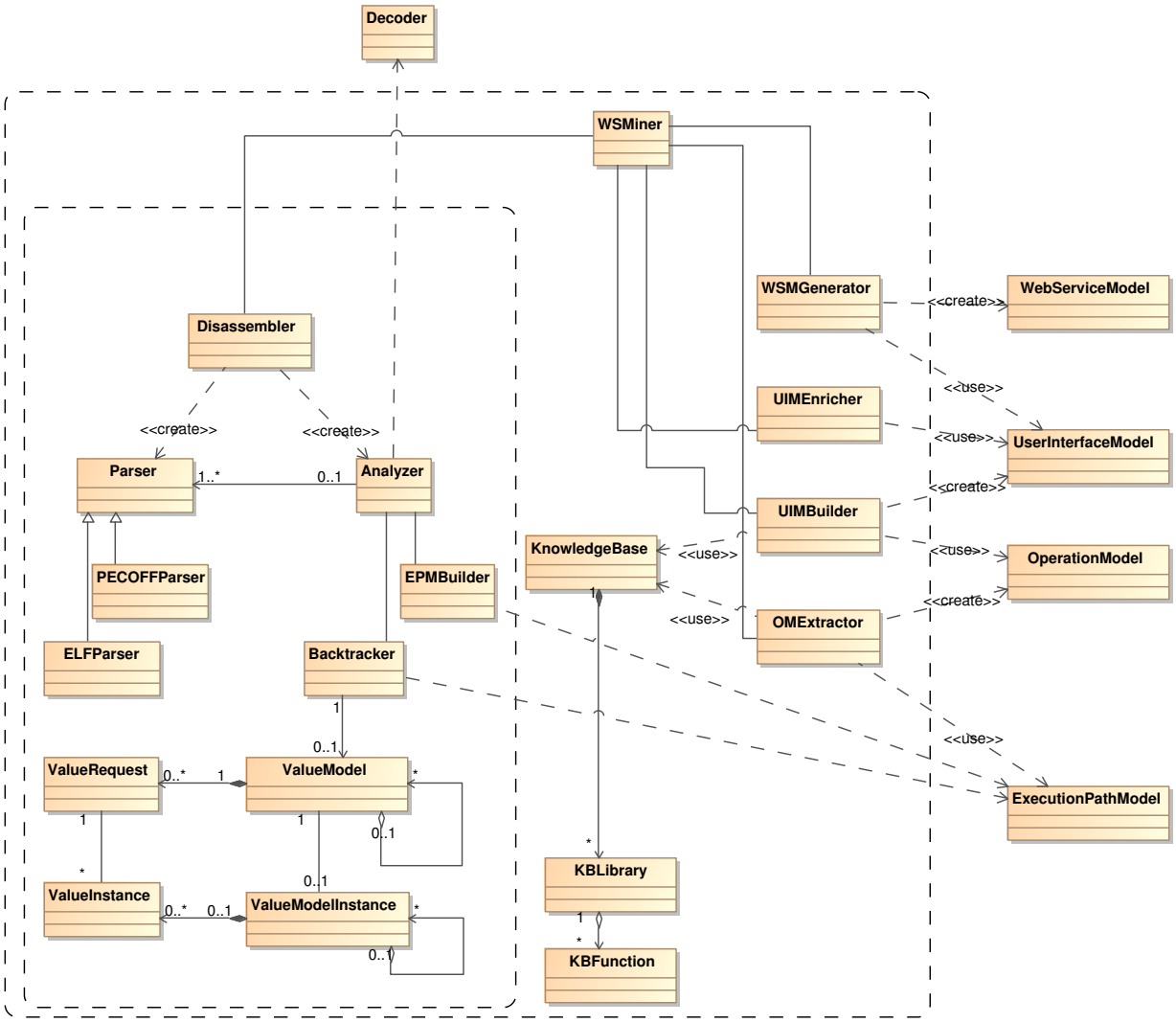


Figure 6.2: *WMiner*

function *getExecutionPathModel* to retrieve the `ExecutionPathModel` after analysis.
For loading and parsing an executable file, the `Disassembler` uses the services of a `Parser` component. This component provides the **AddressOfEntryPoint** as well as every byte of the file and the sections it is made up of.
The `Analyzer` component provides the actual disassembler functionality that builds up the `ExecutionPathModel`. Since the `ExecutionPathModel` is to be used later on to generate an `OperationModel`, we want the creation of the model to be independent of the model itself.
For this purpose we can use the **Builder** pattern described by Gamma et al. [7] or the **Whole-Part** pattern described by Buschmann et al. [2]. The Builder pattern is more

appropriate for our situation.The Analyzer can be viewed as the **Director** in the Builder pattern. The `ExecutionPathModel` component can be seen as a **ConcreteBuilder** in the Builder pattern.
During the creation of the EPM for a certain application, sequences are created and split, together with the build up of loops. The loops need to be informed of the changes to the sequences they contain. Therefore the Publisher-Subscriber pattern is applied to their relationship.

Implementing backtracking

The requirements on backtracking are presented in table 6.4. To guide backtracking and storing information about instructions and the values retrieved, several classes are used. The first class is `ValueModel`, this class contains the descriptors for the values that need to be retrieved. The second class is `ValueModelInstance`, this is an instantiation of the ValueModel in the sense that it contains possible values for the elements of the ValueModel. If a ValueModel contains only one element, the ValueModelInstance is a set of values for that particular element. If a ValueModel contains a set of elements, the ValueModelInstance contains a set of ValueModelInstance objects that all represent the retrieved values for the ValueModel for a particular path of execution preceding the instruction requesting the backtracking. Finally the `ValueInstance` class represents an actual value for the requested Operand.
In our system, the operand 'indicator' is represented by a `ValueRequest` component, a possible value for such an Operand is represented by a `ValueInstance`. When an instruction depends on multiple operands, the collection of the operands is called a `ValueModel` and a concrete collection of values for the ValueModel is a `ValueModelInstance`.
Preparing functions for backtracking can dramatically improve the speed and quality of backtracking attempts. Since our main analysis loop first traverses 'normal' paths of execution before it analyzes function calls, we can expect a function structure to be complete before call targets are analyzed within a function. When we are preparing a function, the main activity consists of replacing the qualifying operands with a concrete binding to a certain storage location.
Operands are either static values or they point to storage locations with dynamic content. The addressable storage locations are stack, registers and memory. When a certain operand value is needed, a backtracking procedure for the operand value is started. In the simplest version, the backtracking procedure goes back to find an instruction that affects the operand.
The concept of *relatively constant* introduces the idea that certain values don't need to be backtracked until every uncertainty has been resolved. For instance segment register

ID	Description
RBT01	The Backtracker shall retrieve static values
RBT02	The Backtracker shall retrieve dynamic values in the same sequence
RBT03	The Backtracker shall retrieve a set of values in the same sequence when multiple ValueRequest objects are requested
RBT04	The Backtracker shall retrieve a set of sets of values when multiple sequences provide information
RBT05	The Backtracker shall performed backtracking from the requested location in reverse order on the path(s) of execution leading toward the requested location
RBT06	The Backtracker shall recover function argument values
RBT07	The Backtracker shall recover local memory values
RBT10	The Backtracker shall recover directly fulfilled (within the same sequence or its preceeding sequences)
RBT11	The Backtracker shall recover loops
RBT12	The Backtracker shall make a difference between an instruction as a normal instruction and an instruction as a loop guard
RBT13	The Backtracker shall provide affects and dependson relations with operands for functions

Table 6.4: *Requirements on backtracking*

values don't change during the execution of a function, so it is not necessary to keep searching for its value. When a memory location relative to the segment is changed, we only need to find the instruction that changes the memory location with that same segment value, the exact value is not needed to understand behavior.

Information concerning the structure of applications has to be retrieved piecewise. Since the processor instruction for the IA32 processor are variable length instructions and almost any value depicts a certain instruction, it is hard, if not impossible, to retrieve the EPM in one pass. We chose for an approach that retrieves paths in bursts of sequences with constant branching targets followed by calculating dynamic targets and call targets. This approach implies that not all information can be complete at any time. For instance backtracking over a sequence that contains an unprocessed call-target can not be performed correctly any time. Also the addition of links can introduce new paths for backtracking and even loops.

The biggest challenge consists of the recovery of memory values. These depend on multiple registers (segment, base, index) and a constant (displacement).

When a target address has to be retrieved for a CALL instruction, the system has to

be aware of the `dependson` and `affects` relationships of such a call. We are only interested in the CALL target and not in the information needed to actually execute the CALL. In such a case we can neglect the `dependson` relationship with `ESP`, `EIP` and `SS`. The same is true for the affects relationship: the CALL instruction itself will return the `ESP` register to the state before the call was issued. Of-course the `EIP` is changed to the instruction following the CALL. To make sure the backtracker is not retrieving too much information, the `ESP`, `EIP` and `SS` registers will be suppressed for the backtracking ValueModel.
The same kind of suppression can be applied to some special cases of instructions. For instance `XOR EBX, EBX` depends on EBX, but we don't need its value because the result will always be zero (0). Of course when we are checking loop-guards, the value is important to determine the resulting EFLAGS-values. This implies that the filtering of operands can not be implemented in the instruction object, therefore we place it in the ValueRequest.
When during backtracking a function call is encountered, the effect of the function is calculated, in such a case, memory changes and register impacts can be compiled. This might be overhead in many cases, but it has to be done to retrieve reliable results.
For functions we register the affects and arguments. The affects can be memory or registers, possibly depending on the input arguments. The arguments are either values or pointers, depending on how they are pushed onto the stack.
As loops are consisting of one or more sequences that might be split into pieces, a mechanism is needed to keep the relations between loop and sequence objects synchronized. The *publisher-subscriber* pattern is exactly what we need.

Static analysis revisited

Our static analyzer is based on an algorithm that analyzes instructions one by one in execution order. The analyzer starts with the first instruction of the application and reasons from that point on. Whenever an unconditional branch is encountered, the target address is scheduled for analysis, where a backtracking procedure for the target address might be executed. When a conditional branch is found, all its target instructions are scheduled for analysis.
The analyzer keeps analyzing until no new addresses to analyze are found. When an instruction outside the address space of the application is found, the analyzer looks up information about the address in our knowledge base. It will not further analyze that address directly, later on when more information is needed, the analysis process might be started for the particular address. In figure 6.3 we present the main loop of our analysis system.
The tool creates a sequence at the first instruction of an application. Then it adds follow-

```
1  FetchInstruction from InstructionStack
2  CompileEffects for Instruction
3  if Instruction not RET | JMP
4          Store NextInstruction into InstructionStack
5  if Instruction is JMP  | Jcc
6          Store TargetInstruction into InstructionStack
7  if Instruction is CALL:
8          Store TargetInstruction into FunctionEntrySet
```

Figure 6.3: *Pseudo code for the main analysis loop*

ing instructions to that sequence until a branch to more than one target is encountered. In such a case the sequence that contains the branch is split and new sequences are created for each target and the links between sequences are set accordingly.

The static analyzer tracks back for possible values for operands used by an instruction. The simplest case is when an operand is satisfied in the local non-looping sequence by a constant, in such a case there is only one possible value for the operand. When the operand is satisfied in the local sequence, but the sequence is looping, there might be multiple values for the operand. In such a case a special loop-backtracking algorithm is started. This algorithm tries to find the constraints on the loop-guards such that it can provide a smaller set of possible values.

The last part of the first analysis phase is the modeling of functions. In this context a function is a set of instructions of which the first instruction is the target of a CALL instruction and all instructions following that instruction, including other function calls.

OMExtractor

The static analysis results in a model of sequences, loops and functions and the relations between these elements (the Execution Path Model). The Operation Model (OM) presents a higher level picture of the structure of the binary application, which is very useful in the detection of Web Service operation candidates. The `OMExtractor` generates an OperationModel based on the ExecutionPathModel.

To create an OM from an EPM, the KnowledgeBase is used to detect constructs at higher levels of abstraction than in the EPM. As described in chapter 5, the process of conversion is started with the first instruction of the application and the Sequence to which it belongs. It then traverses the targets of the first Sequence, until every Sequence has been visited.

```
1  MainAnalysis(insaddr: AddressOfEntryPoint) {
2          ins = Decode(insaddr);
3          nextaddr = insaddr + ins.size;
4          if(unconditional_branch(ins)) {
5                  foreach(addr = {targets}) {
6                          sequence.Add(target)
7                  }
8          } else if(conditional_branch(ins)) {
9                  foreach(addr = {nextaddr, targets}) {
10                         seqnew = CreateSequence(addr);
11                         LinkToSequence(sequence, seqnew);
12                 }
13         } else {
14                 sequence.Add(ins);
15         }
16 }
```

Figure 6.4: *Main Analysis*

UIMBuilder and UIMEnricher

Chapter 5 outlines the process of model conversion. To get from a lower level of abstraction to a higher level of abstraction, every instruction that involves user interface elements is added to the UIM. This is done by the UIMBuilder. It simply traverses the operations in the OperationModel and filters out the functions that are concerned with user interface elements.

The second step consists of finding caption-relations between controls and labels. This process is executed by the UIMEnricher. It traverses the knowledge in the UIM and extends that knowledge with the caption relations between labels and controls.

WSMGenerator

Before we can export a WSMProfile, it is necessary to generate a WSM from the UIM. This process is executed by the WSMGenerator, a component that analyzes the UIM. The resulting WSM can be converted to a WSMProfile by the WSMProfile package.

6.7 WSMProfile

The last step of the WSMiner execution consists of the generation of the WSMProfile. In this section the format of the filetype is described based on the requirements. First of all the WSMProfile should reflect the WSDL20 InfosetModel. This model provides everything that is necessary for describing a Web Service. In addition to this information, the file format should indicate how every operation has to be implemented by the executor. Where the application binary needs to be instrumented and how the application can be instructed to get to the right state.
The WSMProfile package provides export and import classes to convert between a WebServiceModel object and a WSMProfileFile.

6.8 WSMExecuter

The WSMExecuter accepts a WSMProfile and uses that profile to set up a Web Service endpoint and the means to forward requests to the application. Because endpoints are dynamically assigned in our situation, the WSMExecuter generates a WSDL file for the Web Service during initialization. The WSDL file can then be published to make sure that the service can be found and used.
To forward requests to the application for which a WSMProfile has been generated, the WSMExecuter uses an `ApplicationController`. This component is able to start an application (represented by a `Application` component) process and inject messages and intercept results. Depending on the operating system, the component uses a certain technique for application control (see chapter 3).
The WSMExecuter depends heavily on the services of the `SOAPEngine` package. This package provides the whole SOAP and WSDL side of the WSMExecuter, it can be provided by something like Apache Axis CPP.

6.9 SOAPEngine

The SOAPEngine package consists of wrapper classes for an external library like Apache AXIS. It provides services for generating WSDL files, setting up a request listener and parsing SOAP messages.

Chapter 7

Prototype implementation

Quit the wrong stuff.
Stick with the right stuff.
Have the guts to do one or the other.

Seth Godin - The Dip

A prototype was implemented based on the design of the previous chapter, together with a basic mining client. The prototype was built using Microsoft Visual C++ and Intel XED2. We used Apache Axis CPP to implement the WSMExecuter.

7.1 Parsing application files

Applications are stored in files with a certain format. Next to the binary code, a file also contains information about how and where it should be loaded into memory and where the first instruction to execute is located. The implementation of a prototype for our solution is aimed at Microsoft Windows applications. Therefore the `Parser` component is created for the PE/COFF format which is required on that platform. Microsoft provides a format specification [5], even more important are the header-structures that are provided in `winnt.h`. With these two sources of information we are able to correctly load PE/COFF files, find their sections and identify the starting byte in the code section.
When loading an application file, we immediately load the whole file into memory. The PE/COFF format contains a "MS-DOS compatible exe header" which has a reference to the PE Header. This header consists of a file header and an optional header.
Following the headers we can find the sections in the file. For instance the `.text` section, which contains the processor instructions. After analyzing the sections, our parser starts analyzing the data directories in the optional header. These contain references to the import table (for importing references to external functionality), the resource table, etc. Each data directory is fully contained by a section, so when we know which section contains the data directory, we can use its VirtualAddress and PointerToRawData to calculate the position in memory.

After everything has been loaded, the decoding of the executable byte code can be started, as is explained in the next section.

7.2 Analyzing an application

Static analysis consists of decoding byte code from the first instruction of an application until the complete structure has been traversed. In our prototype we used XED2 (X86 Encoder Decoder, version 2) as our IA-32 instruction set decoder. XED2 [3] is a free product (although not open source) that is created by Intel as part of the PIN dynamic instrumentation suite [9].
By entering a pointer to the first byte of an instruction, XED2 provides us with a data structure containing information about the decoded instruction, for instance its length in bytes, type of instruction and its operands.
When a target resides in another file (library), the analyzer has to create a new parser which provides the information needed to investigate the behaviour of such a target. It traverses the directories, just as an operating system would do, to locate the first occurence of the required library file.
The decoding of instructions is a small portion of the analysis procedure. The largest part consists of storing instructions at the right place in the `ExecutionPathModel` based on the information gained from decoding.
We implemented two nested loops: one for analyzing instructions and their static targets and a second one for dynamic branch-traversal. The first loop is the inner loop of the second one. The inner loop retrieves the static targets of the instruction and creates links from the current instruction to those static targets. During this analysis loop, dynamic targets as well as call targets are are stored, such that they can be analyzed later on.

Names

An important part of the static analyzer is recovering function names in imported libraries. Without these names it is impossible to retrieve addresses of the functions that are imported by our subject. It is also hard to use the `KnowledgeBase` if we don't know the names of the functions.
The central part of the PE/COFF format for import information is the import lookup table. The import directory table contains an import directory table for each imported library. This table stores a pointer to the import lookup table of a library which in turn contains all functions imported by the subject. If we wanted to calculate the location of the name of a function that is called by `CALL DWORD PTR [0x10011E8]` in

`calc.exe` we have to execute the steps listed in figure 7.1.

```
1  thunk RVA = 0x10011E8 - optionalHeader.ImageBase
2  Scan the Import address table to find the largest value that is
3         smaller than our thunkRVA
4  Calculate the difference (11E8 - 11BC = 2C)
5  Use the import lookup table RVA associated with this import
6         directory table entry to lookup a DLL import lookup table,
7         which contains a reference to the
8         first element (0x12DC8 - E00 = 0x11FC8)
9  This value needs to be added to the difference calculated
10        earlier (2C) yielding 0x11ff4
11 At that location we can find a reference to the Hint-name table
12       containing a hint value and the string "\_cexit".
```

Figure 7.1: *Retrieving the name of an imported function*

Implementing dependson and affects

A single instruction depends on the values of one or more registers, memory locations and constants. To make it possible to easily find the instructions that change a certain value, the `affects` relationship has been introduced in chapter 2. In XED2 this concept has also been implemented by the `_rw` attribute of the operands of an instruction, it is of type `xed_operand_action_enum_t` (table 7.1). When this value indicates that an operand might be read, we add it to the *dependson* part, if it indicates that an operand might be written, we add it to the *affects* part of an instruction. If an operand is both read as well as written, the operand is added to both the *dependson* and the *affects* sets.

Constructing an Instruction object

Instruction objects are the basic elements that contain the information needed to build an ExecutionPathModel. An `Instruction` object is instantiated based on the information of the `DecodeInstruction` retrieved from the application file. First of all the address of the logical next instruction is recorded. When the instruction is a RET or a JMP, there is no logical next instruction, but otherwise it is available. The system makes a difference between branches and calls.

Value	Meaning
XED_OPERAND_ACTION_INVALID	
XED_OPERAND_ACTION_RW	Read and written (must write).
XED_OPERAND_ACTION_R	Read-only.
XED_OPERAND_ACTION_W	Write-only (must write).
XED_OPERAND_ACTION_RCW	Read and conditionlly written (may write).
XED_OPERAND_ACTION_CW	Conditionlly written (may write).
XED_OPERAND_ACTION_CRW	Conditionlly read, always written (must write).
XED_OPERAND_ACTION_CR	Conditional read.
XED_OPERAND_ACTION_LAST	

Table 7.1: *enumeration xed_operand_action_enum_t*

Operand instantiation

For registers it is very easy to represent instantiations, since every register has only one instance in single processor environments. Therefore when an instruction depends on a register, an instruction that affects that same register has a direct relationship.
For the stack it is a bit harder to make an easy mapping. We could implement a virtual stack and monitor it when we PUSH or POP values. When composing dependson and affects information for a whole function, the analyzer detects that a function call which pushes a certain register at its start and pops the value of that register back at the end of the function will be identified as a function that has no effect on the register we are analyzing.

Function Models

Function models inform the backtracker about the effect of an application. Specific conventions are used when a compiler generates code for a function. For instance which register is used for returning values, the order of input arguments on the stack (left-to-right or right-to-left). When implementing a knowledge base we can use these conventions to easily create function models for function profiles.
On Microsoft Windows most functions comply to the WINAPI (also known as `__stdcall`) conventions (instead of FAR PASCAL) which implies the following: (1) arguments are placed on the stack from right to left, (2) stack-cleanup is performed by the called function and (3) name is prepended with an underscore and appended with an at-sign followed by the number of bytes of stackspace required. Furthermore the return value is placed in the EAX register. The naming conventions are only applied to newly compiled code, the Windows API has more readable naming schemes.
A function can affect registers, stack and memory. To guide backtracking we have to

implement certain standard notations for function behavior. We have chosen to only allow elements generated by the Backtracker as semantic elements. In this way it easy to extend the set of permissible elements in source code: just add a new element to the Backtracker component and some code to use the element by the backtracking procedure.

One is the change of memory, for instance `init mem x with 0` or `between addr x and addr y make every value zero`. Another is the calculation on the return value based on input arguments. For instance true or false as possible return values, others are 0 or length of input string. For library functions, we implement a knowledge base with function-templates, other functions will be generated on the fly.

7.3 Windows Resources

Icons, dialogs, strings and menus are all embedded as resources in application binaries or in the libraries they import. For instance `wab.exe` has its resources stored in `wab32res.dll`. To learn more about an application, the resource data has to be analyzed. This yields an overview of which controls are placed on a window and how they relate spatially as well as menu structures.

In principle, the dialog resource can be seen as a set of control-creation-codes executed at once. This is due to the fact that a dialog resource is loaded with the CreateWindow function and that function generates all controls needed at once.

To effectively recover UIM information from a dialog resource, our analyzer loads the dialog resource and displays it. At display time, the elements on the dialog can be traversed (using `EnumChildWindows`) and the information gathered can be added to the UIM. In figure 7.2 the process of loading a resource (in this case the Dialog resource with index 114) from a DLL file (in this case wab32res.dll). The `DialogProc` function registers a handle to the window, such that the main analyzer process can call EnumChildWindows to recover every control on the dialog.

EnumChildWindows requires a pointer to a callback function that is automatically called for every ChildWindow encountered. By generating controls in that callback function, the UIM is generated.

Menus can be analyzed in memory, by loading the menu from resource with a call to `LoadMenu`. After the menu has been loaded, the different menuitems can be traversed, to retrieve their labels and the id of the menu item. The ids are the ids of the commands that need to be executed by a the application when the menu item is selected. The menu id is also the AppControl initialization for a certain operation.

```
1  HMODULE hExe;
2  HGLOBAL hResLoad;
3  HRSRC hRes;
4
5  hExe = LoadLibraryW(L"wab32res.dll");
6  if(hExe != NULL) {
7    hRes = FindResource(hExe, MAKEINTRESOURCE(114), RT_DIALOG);
8    if(hRes != NULL) {
9      hResLoad = LoadResource(hExe, hRes);
10     ::DialogBoxIndirectParam(hInst,  (LPCDLGTEMPLATE)hResLoad,
11                  0 , DialogProc, 0);
12   }
13 }
```

Figure 7.2: *Loading and analyzing a resource*

DialogBox	Create a modal dialog
CreateDialog	Create a mode-less dialogbox
EndDialog	destroy a modal dialog box
DestroyWindow	destroy a mode-less dialog box
CreateWindowEx	
ShowWindow	

Table 7.2: *Building blocks for forms*

7.4 UIM

When a programmer creates a dialog in program code (not with a user interface editor), he or she uses function calls like the ones in figure 7.3.

```
1  // Create button
2  button1 = CreateWindow("Button", "Submit", WS_CHILD |
3                         WS_VISIBLE | BS_DEFPUSHBUTTON ,
4                         200,300, 50,60, hwnd,
5                         (HMENU)1, i, NULL);
```

Figure 7.3: *Generating a button*

Due to our focus on providing Web Service operations that drive our application, our

UIM specifically provides a means for describing groups of related components. The components are input or output values and their labels. The values can have constraints.

7.5 Implementing a KnowledgeBase

Higher level knowledge about functions is stored in a KnowledgeBase. Currently, we have only implemented functions necessary to find user interface artefacts. We have implemented the knowledge base for instance for the functions listed in table 7.2. In the listing in figure 7.3 an example call to the CreateWindow function is provided. The knowledgebase entry for `CreateWindow` is able to generate a UIM::Button control from this function.

For instance it is aware of the 11 arguments of the function. It is programmed to recover the string pointed to by the first argument and act upon its value ("Button" implies that the analyzer has to deal with a Button control), the second value is a pointer to the text on the button, the third argument shows that the type of button is BS_DEFPUSHBUTTON (this parameter could also indicate that the button is a checkbox or for instance a radio button). The boundaries of the control on its parent is indicated by arguments 3 to 6 (zero based index). Argument 7 is the parent of the button. Argument 8 to 10 are not interesting for us.

7.6 Web service mining

After static binary analysis of the application has been completed, the model (EPM) created by that step is analyzed further to understand its structure (OM).

The Win32 operating system automatically creates a message queue for each thread. The application has to provide a message loop which retrieves messages from the thread's message queue and dispatches them to the appropriate window procedures. A thread must create at least one window before starting its message loop.

To find the message loop for a 'normal' Win32 application, we have to look for function calls to `GetMessageW` or `PeekMessageW`. When the return value (and thus EAX) of GetMessageW is 0 (ZERO), the WM_QUIT message is encountered, indicating that the application needs to exit. To find our messageloop, we can take advantage of this fact: a jump-if-not-zero after a `test eax, eax` will give away the start of the message loop.

7.7 WSMExecuter

Although the WSMExecuter is largely depending on Apache Axis, it implements one extremely important task: controlling an application to call its functions and retrieve its return values. The `ApplicationController` component is responsible for the correct execution of application functionality. When a request is received, the ApplicationController starts up the application that has been mined. It issues messages to navigate the application to the right dialog and enters the information received from the request. Finally the dialog is submitted and the results that might be gained are recorded.
If the Web Service operation needs to return information, the ApplicationController instructs the SOAPEngine to send a reply.
Our prototype has been developed for Microsoft Windows XP. The first task it has to accomplish is to start an application such that the ApplicationController has full access to the application features. It turns out that this extremely simple to accomplish: use `CreateProcess` to start the external application in a proces controlled by the ApplicationController. CreateProcess provides information that can be used to retrieve a handle to the main window of the application.
The `EnumWindows` function iterates over the set of all windows that are currently opened by any running application. This function needs a handle to a call back function that is called for any element of the set of windows. It provides a handle (HWND) to the window to the registered callback function. With the HWND it is possible to retrieve the associated process id, which can be matched to the id of the process just started by the ApplicationController. With the HWND we have access to all user interface components of the application, including menus and the controls on the different windows.
When Axis receives a request for our application, it is forwarded to the ApplicationController. The first thing that has to be done, is to change the state of the application such that the information received can be entered into the forms or the required information can be retrieved from it. With the `AppControl` element of the WSM, we have all the information needed to get to the right state. For instance when a menu item has been analyzed, a unique *messageId* is retrieved. When this messageId is being sent to the window of the application, it is just like a user clicked the menuitem. We can do this by calling `::SendMessage(hWnd, WM_COMMAND, messageid, 0)`.
Secondly, the ApplicationController has to modify the contents of controls to represent filling in the form. For a edit control we can use the `SendMessage` function when we have retrieved the `hwnd` pointing to the control. See figure 7.4.

```
1  wchar_t value[] = L"The value to enter";
2  SendMessage(hWnd, WM_SETTEXT, 0, (LPARAM) value);
```

Figure 7.4: *Modifying a control's content : the Edit control*

Chapter 8

Conclusion and outlook

Forethought we may have, undoubtedly, but not foresight

Napoleon Bonaparte

8.1 Summary and Contribution

This thesis investigates the possibilities for automatic analysis of application binaries and the automatic extension of such binaries with Web Service access. Existing technologies for application analysis and model recovery are used and extended where necessary. The different technologies are combined into a design for the system.
The system depends heavily on a static binary analyzer developed by ourselves. The analyzer generates a model containing all paths of execution for a certain application: the Execution Path Model (EPM). Using an intermediary model, the Operation Model (OM), the EPM is converted into a model for the user interface of an application: the User Interface Model (UIM).
The UIM is analyzed in order to match user interface controls with captions. The combination of a caption and a control provides us with enough information to generate a data field with a correct name and data type. A set of these data fields combines into an input type for a Web Service operation.
After analysis, a set of Web Service operation candidates is captured in a Web Service Model (WSM). This model provides information about possible operations, how to inject them into an application and how to retrieve return values after injection of an operation.
The prototype is not perfect, but it does a pretty good job in the case of `wab.exe`. The analyzer and the injector are implemented in C++. It is built for Microsoft Windows and on that platform the injector simply sets up a Web Service endpoint with Apache Axis CPP. When a request for an operation has been received, the analyzed application is started in a process owned by the WSMExecuter. From that moment on messages are sent to the application according to the received operation request.
The remainder of this chapter is used for the evaluation of the process as well as the results. We also provide some side notes on issues not addressed in this paper.

Information security issues

With the introduction of Web Services, it becomes increasingly important to take information security into account. Since a Web Service's reach can easily be extended from the local network to the world wide web, it seems not very unlikely that a mistake in granting access to certain data will be made.
When a Web Service in the local area network of an organization provides access to customer information, this is probably the right situation. But when someone decides to grant access to such a Web Service through a proxy to the outside network, it is highly probable that someone's privacy is going to be violated.
One might argue that Web Services are designed by humans and that a designer has to take the privacy issues into account. But when Web Services are automatically mined, the mining application should assist the designer in making the right decisions.

Licensing

Most business applications are provided with strict licenses. A license might not allow the analysis of an application binary or the execution of the application inside another process. Although our system does not modify application binaries, it might still violate the license.

Future work

Due to time constraints we were not able to incorporate all our initial ideas in this research:

- *Retrieval of validation rules from binary code*
  When input has been received through user interface controls, an application might validate the received input. To improve the data types generated by our system, it would be a good idea to retrieve those validation rules.

- *Bypassing the code by direct data-model-manipulation*
  In stead of entering data to an application through a user interface, better performance can be gained when data is immediately entered into the underlying data model of the application.

Appendix A

DailyNote - expected analysis results

This chapter provides the expected analysis results from the DailyNote example in chapter 1.

```
1  <S:Envelope xmlns:S="http://www.w3.org/2003/05/soap-envelope">
2   <S:Header>
3   </S:Header>
4   <S:Body>
5     <m:addNote xmlns:m="http://namespaces.example.com/dailynote">
6       <note>Pete cleared his desk today.</note>
7     </m:addNote>
8   </S:Body>
9  </S:Envelope>
```

Figure A.1: *A request to the WSDL for DailyNote*

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <wsdl:description
3        xmlns:wsdl="http://www.w3.org/2005/08/wsdl"
4        xmlns:dn="http://namespaces.example.com/dailynote"
5        xmlns:wsoap= "http://www.w3.org/2005/08/wsdl/soap"
6        xmlns:soap="http://www.w3.org/2003/05/soap-envelope">
7    <wsdl:documentation>
8      Send us your DailyNote.
9    </wsdl:documentation>
10
11   <wsdl:types>
12    <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
13           xmlns="http://namespaces.example.com/dailynote">
14     <xs:element name="note" type="xs:string"/>
15    </xs:schema>
16   </wsdl:types>
17
18   <wsdl:interface name="dailyNote">
19    <wsdl:operation name="addNote"
20         pattern="http://www.w3.org/2005/08/wsdl/in-only
21         style="http://www.w3.org/2005/08/wsdl/style/iri">
22     <input messageLabel="In" element="dn:note" />
23    </wsdl:operation>
24
25    <wsdl:binding name="dailyNoteSOAPBinding"
26       interface="dn:dailyNote"
27       type="http://www.w3.org/2005/08/wsdl/soap"
28       wsoap:protocol=
29        "http://www.w3.org/2003/05/soap/bindings/HTTP">
30     <wsdl:operation ref="dn:addNote"
31       wsoap:mep=
32          "http://www.w3.org/2003/05/soap/mep/soap-response"/>
33    </wsdl:binding>
34    <wsdl:service name="dailyNoteService"
35        interface="dn:dailyNote" >
36     <wsdl:endpoint name="dailyNoteEndpoint"
37        binding="dn:dailyNoteSOAPBinding"
38        address ="http://example.com/dailyNote"/>
39    </wsdl:service>
40   </wsdl:interface>
41 </wsdl:description>
```

Figure A.2: *A possible WSDL for DailyNote*

Bibliography

[1] Ulrich Bayer, Christopher Kruegel, and Engin Kirda. TTAnalyze: A tool for analyzing malware.

[2] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. Pattern Oriented Software Architecture Volume 1: A system of patterns. Wiley, Februari 2006.

[3] Intel Corporation. Xed2 manual.
http://rogue.colorado.edu/Pin/docs/20751/Xed/html/.

[4] Intel Corporation. Intel ia-32 architecture software developer's manual, 2003.

[5] Microsoft Corporation. Microsoft portable executable and common object file format specification revision 8.1.
http://www.microsoft.com, Februari 2008.

[6] Yves Gaeremynck, Lawrence D. Bergman, and Tessa Lau. More for less: model recovery from visual interfaces for multi-device application design. In Proc. of the international conference on Intelligent user interfaces, Jan 2003, pages 69–76. ACM Press, 2003.

[7] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design patterns: elements of reusable object-oriented software. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[8] Laune C. Harris and Barton P. Miller. Practical analysis of stripped binary code. SIGARCH Comput. Archit. News, 33(5):63–68, 2005.

[9] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. SIGPLAN Not., 40(6):190–200, 2005.

[10] Barton P. Miller, Mark D. Callaghan, Joanthan M. Cargille, Jeffrey K. Hollingsworth, R. Bruce Irvin, Karen L. Karavanic, Krishna Kunchithapadam, and Tia Newhall. The paradyn parallel performance measurement tool. IEEE Computer, 28(11):37–46, 1995.

[11] Daniel Molkentin. The Book of Qt 4: The Art of Building Qt Applications. No Starch Press, San Francisco, CA, USA, 2007.

[12] Heidi Pan and Krste Asanovic. Controlling program execution through binary instrumentation, 2005.

[13] Heidi Pan, Krste Asanović, Robert Cohn, and Chi-Keung Luk. Controlling program execution through binary instrumentation. SIGARCH Comput. Archit. News, 33(5):45–50, 2005.

[14] Julian Smart, Kevin Hock, and Stefan Csomor. Cross-Platform GUI Programming with wxWidgets (Bruce Perens Open Source). Prentice Hall PTR, Upper Saddle River, NJ, USA, 2005.

[15] Redmond Wa, Galen Hunt, Galen Hunt, Doug Brubacher, and Doug Brubacher. Detours: Binary interception of win32 functions. In In Proceedings of the 3rd USENIX Windows NT Symposium, pages 135–143, 1999.

[16] World Wide Web Consortium (W3C). Web services glossary.
http://www.w3.org/TR/ws-gloss/, February 2004.

[17] World Wide Web Consortium (W3C). Extensible markup language (xml) 1.0 (fourth edition).
http://www.w3.org/TR/xml/, August 2006.

[18] World Wide Web Consortium (W3C). Simple object access protocol (soap) 1.1.
http://www.w3.org/TR/soap/, 2007.

[19] World Wide Web Consortium (W3C). Web services description language (wsdl) version 2.0 part 1: Core language.
http://www.w3.org/TR/wsdl20/, June 2007.

[20] World Wide Web Consortium (W3C). Xforms 1.1 - w3c candidate recommendation 29 november 2007.
http://www.w3.org/TR/xforms11/, 2007.

Index