TOWARDS WEB SERVICE TAGGING BY SIMILARITY DETECTION

by

DOUGLAS MARTIN

A thesis submitted to the School of Computing

in conformity with the requirements for

the degree of Master of Science

Queen's University

Kingston, Ontario, Canada

October, 2011

Copyright © Douglas Martin, 2011

Abstract

The web of the future will require automated tagging of equivalent or similar services in support of service discovery and the selection of appropriate alternatives in case of failure. Code similarity detection tools, or clone detectors, provide a mature and scalable method of identifying these kinds of similarities and can be used to assist in this problem. However, they require a set of units to be compared; something to which the most popular description language, WSDL (Web Service Description Language), does not lend itself. First, each WSDL description can contain more than one operation description, which does not provide the granularity we need to compare services on the operation level. Secondly, these operation descriptions are mixed together throughout the file, often sharing some common elements. This thesis describes a technique for extracting the elements of each operation description and consolidating them into a self-contained unit using TXL, a source transformation language. These units, referred to as Web Service Cells or WSCells (pronounced "wizzles"), can then be used by similarity detectors to search for similarities. We describe a modified architecture to the NICAD clone detector to support the creation of WSCells, and the implementation of a special WSDL extractor we used to emulate this modification in its absence.

Co-Authorship

Parts of this thesis, specifically parts of Chapter 2 and Chapter 6, have been published in a book chapter with James R. Cordy for The Smart Internet: Current Research and Future Applications [19]. Other parts, specifically Chapter 4, have been published in the proceedings of the ICSE 5th International Workshop on Software Clones (IWSC 2011) [20]. For both of these publications, I was the primary author and conducted the research under the supervision of James R. Cordy.

Parts of this work will also appear in a paper in the proceedings of the 13th IEEE International Symposium on Web Systems Evolution (WSE 2011) [13], which was co-authored with Scott Grant, James R. Cordy and David Skillicorn. This was a joint research effort involving this work and the research of Scott Grant. The paper was written by both Scott and myself, however, only my involvement is used in this thesis.

Acknowledgements

I would like to thank my supervisor, Jim Cordy, for guiding through this new experience we call “grad school.” I’d also like to thank my lab mate, and neighbour, Scott Grant for his invaluable input throughout this work, but also for the distractions like the endless YouTube videos and the addictive Flash games. I’d like to extend that gratitude to all of my lab mates – Matthew Stephan, Andrew Stevenson, Gehan Selim, Manar Alalfi, Asil Almonaies, Widd Gama and briefly Chancel Roy (creator of the NICAD clone detector, used in this thesis) – but also members of the Modeling and Software Engineering group (MASE) who feel just as much as lab mates as anyone else – Eric Rapos, Karolina Zurowska, Eyrak Paen, Nick Chausse and Mark Fischer.

A huge thank you goes to IBM Centres for Advanced Studies (CAS) in Markham, Ontario for their full support of this research and beyond with an IBM CAS Fellowship. I had the great opportunity of working at the Markham lab in the summer of 2010 where I met a lot of friendly people.

Lastly, I would like to thank my family – my mother Judy, father Brian and sister Ashley (not to mention my extended family of aunts and uncles) – for their support. And to the many friends I’ve made along the way, notably Marie Matheson and Cyrus Boadway for putting up with me during the summer working at IBM CAS, Andrew Brown for playing cribbage with me every day at lunch, and the many, many others that could fill pages. Together you have made this one of the most enjoyable experiences I have ever had.

Thank you all.

Table of Contents

Page

List of Figures

List of Tables

1 – Introduction

The World Wide Web as we know it has existed for nearly two decades. Beginning as nothing more than a set of interconnected static webpages, it has evolved into something much more. From e-commerce sites, like Amazon and eBay, to social sites, like Facebook and Twitter, the web has become a platform for rich internet applications. Many of these web applications provide APIs to call their services that can be used to create new applications and compose new and more complex services. This growing number of services makes it difficult for developers and users to find the right services to meet their needs. Tagging and categorization provide a means to find services, and while doing so by hand may work for new services, automation will be needed to tag the vast array of services that already exist. This thesis describes a first step towards such an automation using similarity detection tools.

1.1 Motivation

Automated tagging and categorization involves identifying similarities between services and service operations, so similarity detection tools, or clone detectors, are something we may be able to utilize. There has been much research in finding duplicated (or slightly modified) code fragments in software systems and as a result there are a number of tools and techniques at our disposal [24]. Given a repository of web services, a clone detector should be able to identify groups of similar web service operations, which we can then tag using domain specific ontologies. But there is a challenge that must be overcome. The problem lies in how web services are described and made publically available.

Web Service Description Language (WSDL) [5, 6], the language used to describe a large number of web services, is structured in such a way that makes it difficult for a clone detector to identify and extract units to compare. A WSDL document contains descriptions of all the operations that the web service offers, however, each of these operations is separated into pieces that are scattered throughout the file. This poor locality of operations makes it challenging to select fragments to compare because each piece is only part of the whole operation description.

Using entire WSDL descriptions might work, but does not yield the desired level of granularity. This problem led to the idea of reorganizing WSDL descriptions so that all pieces related to an operation are grouped into a self-contained blocks.

1.2 Contribution

This thesis makes two new contributions - a new class of code clones, called *contextual clones*, and a new clone detection architecture to support it. First, we propose an extension to the architecture of the NICAD clone detector [11, 22] to include a phase at the beginning that allows TXL transformations to be applied to the original source that would not be possible after the extraction of potential clones (NICAD's term for units to compare).

Second, as evidence to support the need of the proposed modification, we describe a new strategy for finding clones by modifying source code fragments to add contextual information found outside of them. We illustrate this technique for WSDL and show why it is necessary. We provide an implementation of a special WSDL language plugin written in TXL [9, 10], a source transformation language used by NICAD. Using this, we emulate the proposed preprocessing phase by performing transformations *during* the extraction of potential clones to show that this yields more meaningful results than the naïve approach.

1.3 Chapter Overview

We introduce our new clone detection strategy in detail beginning in Chapter 2 with the background information needed to understand it. This chapter starts with a description of web services with an emphasis on RPC-based services and WSDL. It continues with an overview of source transformation systems, particularly TXL and focusing on the aspects of TXL that are used in the contextualization described later. Then we will look at clone detection and some terminology before going on to describe NICAD, the clone detector we used to search for similar services. The chapter finishes with a survey of research related to finding similar web services, but that use a different approach from the one presented here.

In Chapter 3, we give a more detailed overview of our approach, using an example to illustrate the need for contextualization. Chapter 4 discusses the problem we face when attempting to use clone detection on WSDL descriptions and proposes a new strategy to solve it. It goes on to describe a change to the NICAD architecture that would support this strategy, and describes how we can emulate it for testing. Chapter 5 gives a detailed look at the TXL transformation rules used in a special WSDL extractor for NICAD that contextualizes WSDL operations descriptions into something we call Web Service Cells, or WSCells (pronounced “wizzles”). Chapter 6 provides an empirical evaluation using two datasets of WSDL documents to show the benefits of contextualization. Finally, we close in Chapter 7 with a summary of our work and what it means for the future, as well as further enhancements that might help give more meaningful results.

2 – Background

Our research with web services uses solutions designed for a well-known problem – clone detection – and a method for manipulating code – source transformation – to solve a relatively new problem – finding and tagging similar services. This chapter gives an overview of each of these topics, while focusing more on the aspects of each that are important to understanding this thesis. We start by discussing web service architectures and their description languages, particularly Web Service Description Language (WSDL), the subject of this research. The following section highlights source transformation systems, which we use to contextualize WSDL operations. Then we discuss clone detection and define the terminology used throughout this thesis. We focus on NICAD, the clone detector used for this research and how it works. Finally, we briefly describe other research related to finding similar web service operations in WSDL descriptions and how these approaches differ from the one presented here.

2.1 Web Services

The World Wide Web Consortium (W3C) defines a web service as “a software system designed to support interoperable machine-to-machine interaction” [16]. Put simply, a web service is an application running on a server that has an interface allowing it to be called by a client.

2.1.1 Web Service Architectures

There are 2 main web service architectures – RPC-based and RESTful. RPC-based architectures [4] centre on an exposed set of operations or procedures (the RPC stands for Remote Procedure Call). A client invokes the operation by passing a set of parameters, most commonly using SOAP (Simple Object Access Protocol) [14], a framework for exchanging XML messages over computer networks. The web service then invokes the specified operation with the given parameters on the server and returns the results in the same way the client sent the input. In this architecture the client is responsible for maintaining the state of the application and its resources.

RESTful (REpresentational State Transfer) architectures [21] offload this responsibility to the server. Rather than calling an operation and passing the state to the service every time, RESTful architectures are resource-based, meaning the application state is stored on the server in the form of resources. Resources are any conceivable concept that is addressable, such as a photo, a place, a person, or an event. Being addressable means that each resource has a URI (Uniform Resource Identifier) associated with it. For instance, the capital of Canada may be represented as "`http://maps.example.com/NorthAmerica/Canada/Ontario/Ottawa`" in a mapping service. Clients of strictly RESTful web services manipulate these resources using HTTP's native methods like `GET`, `SET`, `DELETE` and `POST`. For example, calling the address given above for Ottawa with the `HTTP/GET` method might return a map of Ottawa; calling "`http://example.com/ user/doug`" with the `HTTP/SET` along with some appropriate parameters in the HTTP body might create a user with the username "doug" (access restrictions may apply).

Not all web services fall strictly into one of these architectures, however. There are some hybrid architectures that may, for instance, be RESTful in that the server stores the application state and stores resources, but clients invoke operations using only the `HTTP/POST` method instead of `SET` and `DELETE`. These services are not quite RPC-based because the server stores the state, but they are also not strictly RESTful because they don't make use of the HTTP methods. Many web services today, that claim to be RESTful, are in fact using this hybrid architecture.

2.1.2 Description Languages

In order to be used by client applications, web services must provide some sort of interface description to application developers. The interface description language depends on the architecture of the web service being described.

The easiest method of describing web services, no matter the architecture, is using prose, because an English description of the interface will likely already exist from the design process. The problem with this method is that there is no standard; anything goes. This can result in very poor documentation, making the service difficult to use. It also means the service is difficult to discover, and code stubs cannot be generated automatically to invoke the service's operations.

This is where standardized description languages come in. These are XML-based languages that allow the service's operation and their parameters to be specified in a single document that can be published or broadcast on the web for application developers to discover. Once obtained, they can be used to automatically generate code snippets for client applications to use to invoke the service, without having to know the specific address or protocol being used.

There are 2 main description languages – Web Application Description Language (WADL) and Web Service Description Language (WSDL). WADL [15] is the description language used to define RESTful web services; however it is not widely used today. Most RESTful or hybrid application developers opt to using prose. WSDL [5, 6] is the description language used to define RPC-based services but, with WSDL 2.0 [5], is also capable of describing RESTful services as well. It is for this reason that the W3C recommended language for defining web service interfaces is WSDL 2.0. We discuss WSDL in further detail in the following section.

2.1.3 Web Service Description Language (WSDL)

Web Service Description Language, or WSDL, is the most common way of defining RPC-based web services. There are two major versions – WSDL 1.1 [6] and WSDL 2.0 [5]. Only WSDL 2.0 is a W3C recommended standard, because it can describe both RPC-based and RESTful architectures, but WSDL 1.1 remains widely used. For this reason, only WSDL 1.1 will be used in this thesis and any mention of WSDL refers to version 1.1 unless otherwise specified.

We use a simple hotel reservation service shown in **Figure 2.1** as an illustrative example throughout the remainder of this section. This service consists of one operation, `ReserveRoom`, defined in the `<portTypes>` section, which takes an element of type `CreditCard` and an element of type `Room` as parameters. These two element types are defined in the `<types>` element.

```
<?xml version="1.0"?>
<definitions name="HotelReservationService"
             targetNamespace="http://myhotel.com/reservationservice.wsdl"
             xmlns:tns1="http://myhotel.com/reservationservice.xsd"
             xmlns:xsd1="http://myhotel.com/reservationservice.xsd"
             xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
             xmlns="http://schemas.xmlsoap.org/wsdl/">
     <types>
          <schema targetNamespace="http://myhotel.com/reservationservice.xsd"
                  xmlns="http://www.w3.org/2000/10/XMLSchema">
               <complexType name="Room">
                    <sequence>
                         <element name="roomID" type="xsd:int"/>
                         <element name="numBeds" type="xsd:int"/>
                         <element name="isSmoking" type="xsd:boolean"/>
                    </sequence>
               </complexType>
               <complexType name="CreditCard">
                    <sequence>
                         <element name="ccNumber" type="xsd:int"/>
                         <element name="cardHolder" type="xsd:string"/>
                         <element name="expiryDate" type="xsd:date"/>
                    </sequence>
               </complexType>
               <element name="ReserveRoomRequest">
                    <complexType>
                         <sequence>
                              <element name="ccard" type="tns1:CreditCard"/>
                              <element name="room" type="tns1:Room"/>
                         </sequence>
                    </complexType>
               </element>
               <element name="ReserveRoomResponse">
                    <complexType>
                         <sequence />
                    </complexType>
               </element>
               <element name="RoomNotAvailableException">
                    <complexType>
                         <sequence />
                    </complexType>
               </element>
          </schema>
     </types>

     <message name="ReserveRoomRequest">
          <part name="body" element="xsd1:ReserveRoomRequest"/>
     </message>
     <message name="ReserveRoomResponse">
          <part name="body" element="xsd1:ReserveRoomResponse"/>
     </message>
     <message name="RoomNotAvailableException">
          <part name="body" element="xsd1:RoomNotAvailableException"/>
     </message>

     <portType name="HotelReservationServicePortType">
          <operation name="ReserveRoom">
               <input message="tns1:ReserveRoomRequest"/>
               <output message="tns1:ReserveRoomResponse"/>
               <fault message="tns1:RoomNotAvailableException"/>
          </operation>
     </portType>
```

Figure 2.1 – *Example WSDL description.*

An example WSDL description of a simple hotel reservation service with a single operation called "ReserveRoom." The <binding> and <service> elements have been excluded for space reasons.

A WSDL description defines one or more operations provided by a particular web service. These descriptions are broken up into pieces and grouped based on what aspect of the operation they define. These pieces, when linked together (**Figure 2.2**), form the description of the operation. The pieces are organized into 5 groups: types, messages, port types, bindings, and services.

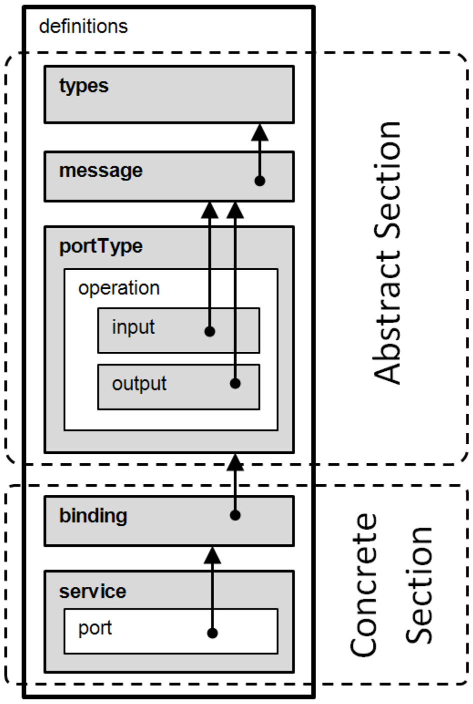

Figure 2.2 – *Structure of a WSDL description.*

A WSDL description of a web service consists of an Abstract section (<types>, <message>, <portType>) and a Concrete section (<binding>, <service>). The Abstract section contains descriptions of the operations, while the Concrete section specifies the address of the service and the message protocol. In this work, we do not consider the Concrete section because it is specific to each web service and is not useful for discovering similar operations.

Types

The `<types>` element contains type definitions for exchanging data between a requesting client and the web service using the XML Schema language [2, 27]. It may contain one or more of these schema, and they can be defined locally (i.e., embedded in the types section), or externally by referencing an ".xsd" file. The schema may define `<complexType>` elements, which are essentially objects that contain other elements. For example, in our simple hotel reservation service, we define a type called `Room` containing two integers to represent the room ID and number of beds, and a boolean to represent whether the room is smoking or non-smoking. The schema also defines elements that are referenced by parts in the messages section.

Messages

`<message>` elements define the data elements corresponding to the input, output and faults of each operation. They contain one or more `<part>` elements representing parameters of the operation. They can be typed where each part represents a parameter, but often they simply refer to an element in the `<types>` element that contains the parameters. This is the case with our example. For instance, the message named `ReserveRoomRequest` contains a part named *body*, which refers to an element in the `<types>` element also named `ReserveRoomRequest`. This element contains the operation's two input parameters, payment and room.

Port Types

The `<portTypes>` element contains one or more `<operation>` elements representing the operations that make up the web service. Each of these operations may contain an `<input>` and/or `<output>` element depending on what kind of communication takes place (e.g. request-response, notification, etc.). It may also contain any number of `<fault>` elements representing errors that may occur. These inputs, outputs and faults refer to messages defined elsewhere in the file. Our simple hotel reservation example contains one operation, `ReserveRoom`, which has one input, one output and one fault, each referring to a message defined previously.

Bindings

`<binding>` elements define a message format and protocol for a `<portType>` element. Often this protocol is Simple Object Access Protocol (SOAP) [14].

Services

`<service>` elements define a group of ports. Ports define an endpoint and specify an address for a binding.

These 5 groups can be split into 2 categories. Types, messages and port types are abstract, while bindings and services are concrete. Our tool only takes the abstract groups into consideration, since the concrete groups are specific to the implementation and not necessary for comparing operations.

2.2 Source Transformation

Source transformation is the transformation of one piece of source code to another, be it within the same language or between two different languages. Source transformation systems automate this process by allowing users to specify transformation rules and apply them to some given source code. Two of these systems – Extensible Stylesheet Language Transformations (XSLT) and TXL – are discussed in this section. XSLT is source transformation system designed specifically for XML-based documents, while TXL is more general and allows grammars to be written for any language. It may seem logical to use XLST for transforming WSDL since it is an XML-based language; however, TXL is already integrated with NICAD, our chosen clone detector. TXL also allows us to create a grammar specifically for WSDL, and not just XML in general, making it easier to manipulate.

2.2.1 Extensible Stylesheet Language Transformations (XSLT)

Extensible Stylesheet Language Transformations, or XSLT [7, 18], is a source transformation system designed specifically for transforming XML documents. All modern internet browsers – including Mozilla Firefox, Microsoft Internet Explorer, and Google Chrome – support XLST, so no special application is necessary. The main use of XSLT is in transforming XML data into HTML documents, which can then be viewed in a human readable format. In this way, XSLT can be thought of as the XML equivalent of CSS (Cascading Style Sheets) for HTML, but they are much more powerful. Unlike CSS, which is essentially a list of attributes to be used as a guideline in displaying an HTML document, XSLT is full transformation system in that it can transform an XML document into a new document without modifying the source. **Figure 2.3** shows an example XSLT transformation (a) to filter a list of graduate students (b) into a list of MSc students (c).

An XSLT stylesheet consists of one or more templates – rules that are applied to the source document. Each template has a `match` attribute, which contains an XPath query [8] specifying the element on which to apply it.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template match="/">
    <MScStudents>
        <xsl:for-each select="studentList/student">
            <xsl:if test="degreeType = 'MSc'">
                <student>
                    <name><xsl:value-of select="name"/></name>
                    <email><xsl:value-of select="email"/></email>
                </student>
            </xsl:if>
        </xsl:for-each>
    </MScStudents>
</xsl:template>

</xsl:stylesheet>
```

***a)** Example XSLT template to filter MSc students from a list of graduate students.*

```
<studentList>
    <student>
        <name>Doug Martin</name>
        <degreeType>MSc</degreeType>
        <email>doug@cs.queensu.ca</email>
    </student>
    <student>
        <name>Scott Grant</name>
        <degreeType>PhD</degreeType>
        <email>scott@cs.queensu.ca</email>
    </student>
    <student>
        <name>Eric Rapos</name>
        <degreeType>MSc</degreeType>
        <email>eric@cs.queensu.ca</email>
    </student>
</studentList>
```

***b)** Example XML document containing a list of graduate students.*

```
<MScStudents>
    <student>
        <name>Doug Martin</name>
        <email>doug@cs.queensu.ca</email>
    </student>
    <student>
        <name>Eric Rapos</name>
        <email>eric@cs.queensu.ca</email>
    </student>
</MScStudents>
```

***c)** Result of applying the XSLT in a) to the XML document in b).*

Figure 2.3 – *Example XSLT transformation.*

The XSLT in (a), when applied to an XML list of graduate students (b), filters out the MSc students and places them in a new XML list (c).

The content of a template is an outline of the desired output, with special tags for manipulating the source document. `<xsl:value-of>` is used to retrieve the value of a node specified in the `select` attribute (an XPath query). `<xsl:for-each>` is used to apply a piece of the template to all nodes matching an XPath query. `<xsl:if>` and `<xsl:choose>` (used with `<xsl:when>` and `<xsl:otherwise>`) are conditional tags for applying a piece of the template to nodes upon matching the expression in the `test` attribute.

As mentioned, an XSLT stylesheet can contain more than one template. The template matching the root element is applied first (`match="/"`) while other templates are applied within that template using `<xsl:apply-templates>`. The template that matches the selected element in the `select` attribute is applied to that element and, when complete, the calling template continues. For example, consider the XSLT shown in **Figure 2.3a**. The `<xsl:for-each>` block could be placed in its own template matching "`studentList`" and replaced with "`<xsl:apply-templates select='studentList'>`" as shown in **Figure 2.4**. Both stylesheets are valid and produce the same results (**Figure 2.3c**).

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template match="/">
    <MScStudents>
        <xsl:apply-templates select="studentList"/>
    </MScStudents>
</xsl:template>

<xsl:template match="studentList">
    <xsl:for-each select="student">
        <xsl:if test="degreeType = 'MSc'">
            <student>
                <name><xsl:value-of select="name"/></name>
                <email><xsl:value-of select="email"/></email>
            </student>
        </xsl:if>
    </xsl:for-each>
</xsl:template>
</xsl:stylesheet>
```

Figure 2.4 – *Alternative XSLT to Figure 2.3a.*

This XSLT completes the same transformation as that of Figure 2.3, but uses multiple templates to accomplish it.

2.2.2 TXL

Unlike XSLT, which can only transform XML documents, TXL [9, 10] operates on any language and as such, requires a user-defined grammar as input with which to parse input. Source documents are parsed (using the provided grammar) into a tree, which is then modified based on transformation rules. This new tree is then unparsed back into the provided grammar and outputted as a new transformed document.

A TXL program consists of two parts – a grammar and a set of transformation rules. A TXL grammar specifies the syntax of a language using a notation similar to BNF (Backus–Naur Form). **Figure 2.5** shows a TXL grammar for the XML schema of the student list example from earlier (see **Figure 2.3**).

A TXL grammar begins with a `program` definition, a special nonterminal defining the structure of a document of the source language. In the case of the example, a program is simply a student list, or a `<studentList>` element. The `program` is defined just like any other nonterminal, using a define statement of the form `define X...end define`, where `X` is the name of the nonterminal. Each `define` statement contains a list of alternative forms of the nonterminal it defines, separated by a vertical bar ("`|`"). Other nonterminals can be used inside a definition by enclosing its name in square brackets (e.g. "`[X]`"). For simple nonterminals, it may make sense to define them in the `tokens` statement. The `tokens` statement is used to define special types of input through regular expressions. For example, in the `studentList` grammar shown in **Figure 2.5**, email has been defined as a string of characters followed by the "`@`" symbol and another string of characters.

TXL contains a number of predefined nonterminals that can be used, such as `[id]` for an identifier containing alphanumeric characters, `[stringlit]` for a string contained in double quotes, and `[charlit]` for a string contained in single quotes. Modifiers can be used to express optional nonterminals (e.g. `[opt X]`) or multiple nonterminals (e.g. `[repeat X]` for 0 or more `X`, `[repeat X+]` for 1 or more `X`). For example, in **Figure 2.5**, a student is defined as a `<student>` element containing `[repeat student_attr+]`. This means that a student element contains 1 or more `student_attr`'s which are later defined as a name, email or degree type.

```
tokens
    emailaddress    "[-\i.]+@[-\i.]+"
end tokens

define program
    [student_list]
end define

define student_list
    [SPOFF] '<'studentList> [SPON]                                          [NL][IN]
        [repeat student]                                                    [EX]
    [SPOFF]'</'studentList> [SPON]                                          [NL]
end define

define student
    [SPOFF] '<'student> [SPON]                                              [NL][IN]
        [repeat student_attr+]                                              [EX]
    [SPOFF] '</'student> [SPON]                                             [NL]
end define

define student_attr
        [name]
    |   [degree_type]
    |   [email]
end define

define name
    [SPOFF] '<'name> [SPON] [repeat id] [SPOFF]'</'name> [SPON]             [NL]
end define

define degree_type
    [SPOFF] '<'degreeType> [SPON] [id] [SPOFF] '</'degreeType> [SPON]       [NL]
end define

define email
    [SPOFF] '<'email> [SPON] [emailaddress] [SPOFF] '</'email> [SPON]       [NL]
end define
```

Figure 2.5 – *TXL grammar for studentList XML data.*

This figure shows a TXL grammar for studentList XML data like that of Figure 2.3b. Using a specialized grammar such as this allows us to easily extract specific kinds of elements (e.g all <student> elements).

One of the benefits of TXL is *pretty printing*, which allows formatting to be specified in the grammar to be used when unparsing the transformed program. Without pretty printing, the output is printed on a single line; but, using a few built-in nonterminals, the author of a grammar can dictate how the output should look. `[SPON]` and `[SPOFF]` are used to turn spaces on and off, respectively; `[SP]` will explicitly place a space; and `[IN]` and `[EX]` will indent and exdent, respectively. By default, TXL inserts spaces in a way that is appropriate for a language like C; however, there are times where spaces are inappropriate. For example, when defining the XML schema in **Figure 2.5**, it is appropriate to turn spacing off while outputting a tag, thereby forcing TXL to print "`<student>`" instead of "`<  student  >`". It is also desirable to indent an element's contents, so we place a [IN] after every opening tag and an [EX] before every closing tag.

The second part of any TXL program is a set of transformation rules describing how to modify the input. These are defined with either the rule keyword or the function keyword, the difference being that a rule is applied to the output of the rule recursively until it fails, while a function is applied only once. **Figure 2.6** shows TXL rules equivalent to the XSLT from **Figure 2.3**. In it, a student list program is deconstructed into a list of [student] nonterminals. Each [student] is passed to the getMScs function, which returns a new (intitially empty) list of [student]'s with a [degree_type] containing "MSc." This new list is encapsulated in <MScStudents> tags and outputted at the newly transformed XML. Both these TXL rules and the XSLT stylesheet perform the exact same transformation on an XML list of students.

TXL is example-based, meaning a rule (or function) consists of a pattern and its replacement. Most rules take the form "replace [*target_type*] *pattern*...by *replacement*." Nonterminals in the pattern can be further pattern-matched using a deconstruct statement. If a pattern does not match at any time, the rule fails and execution continues from where it was called.

Constructing new nonterminals is done using the construct statement, which takes the form "construct *Name* [*type*]." Following that, the contents of the nonterminal are defined, which must match the grammar definition. The contents may be explicitly stated, like NewP in the main rule of **Figure 2.6**, or the result of a function call, like MScs.

TXL has a number of built-in functions, two of which are used extensively in this thesis. The append function, of the form "X [. Y]," appends two lists of nonterminals (lists can be a single nonterminal) together to form a new list. This is done at the end of the getMScs function shown in **Figure 2.6**. Here a matching student, Student, is appended to the list of already matching students MScStudents. The extract function, of the form "[^ X]", extracts all of the nonterminals of a target type from an element. For example, in **Figure 2.6**, the extract function is used to extract the [degree_type]'s from a [student].

```
% Include grammar from Figure 2.5
include "StudentList.grm"

% Redefine program to allow new <MScStudents> list
redefine program
        ...
    |   [SPOFF] '<'MScStudents> [SPON]                                        [NL][IN]
            [repeat student]                                                  [EX]
        [SPOFF] '</'MScStudents> [SPON]                                       [NL]
end redefine

% The first function/rule to be executed
function main
    replace [program]
        P [program]
    deconstruct P
        '<'studentList>
            Students [repeat student]
        '</'studentList>
    construct MScs [repeat student]
         % For each student, check if degree type is 'MSc'
         % Add matches to empty list (_)
        _ [getMScs each Students]
    construct NewP [program]
        '<'MScStudents>
            MScs
        '</'MScStudents>
    by
        NewP
end function

% If Student has degreeType of MSc, add it to the list ([repeat student]).
function getMScs Student [student]
    replace [repeat student]
        MScStudents [repeat student]
    construct DegreeType [repeat degree_type]
        _ [^ Student]
    deconstruct DegreeType
        '<'degreeType> MSc '</'degreeType>
        _ [repeat degree_type]
    by
        MScStudents [. Student]
end function
```

Figure 2.6 – *TXL transformation rules for creating a list of MSc students.*

These TXL transformation rules, when applied to an XML list of graduate students (like Figure 2.3b), create a list of MSc students from studentList XML data just like the XSLT transformation in Figure 2.3.

2.3 Clone Detection

Copying and pasting snippets of source code is a common practice in software development, but can have negative consequences later on in the product lifecycle, particularly when it comes to maintenance. This has led to the development of tools to seek out these duplicated code fragments, called *code clones* or simply *clones*, to either eliminate them or study their evolution throughout development. These tools [24], commonly referred to as clone detectors or similarity detectors, can be used to find not only exact copies of code fragments but also copies that are similar and differ by a few lines.

There are four main categories of clone detection techniques [24] – textual, lexical, syntactical, and semantic. These techniques differ in the ways they represent the source code and search that representation. Textual techniques use the original source code as is with little to no transformation or normalization done before actual clone detection is performed. Lexical techniques transform the source code into a series of tokens like those produced by a compiler. Syntactical approaches use parsers to convert the source code into a tree and find clones by looking for common subtrees. Semantic approaches often transform source code into a program dependency graph (PDG) to represent data dependencies and control flow. The problem of finding clones is then reduced to finding isomorphic subgraphs in these newly created PDGs.

There are a large number of clone detectors – NICAD [11, 22], CCFinder [17] and CloneDR [1] to name a few – that each take different approaches. We settled on NICAD for our research because we needed a clone detector that would allow us to use a language plugin for WSDL, since it is not a language commonly used, and NICAD allows custom TXL plugins (i.e. grammars, extractors, etc.) to be written for any new language. NICAD has also been shown to have high precision and recall [23].

2.3.1 Terminology

There are a number of terms used in the field of clone detection that will be used throughout this thesis. They are defined in this section for reference and visually summarized in **Figure 2.7**.

Code Fragment

A code fragment, or simply a fragment, is any continuous sequence of lines from a given source code.

Code Clone

A code clone is a code fragment for which there exists at least one other code fragment determined to be similar to it by some definition of similarity (determined by the clone detection method used).

Clone Types

There are 4 main types of clones that may be searched for by a clone detector. The first 3 types are based on similarity of the source text and build upon each other. *Type-1* clones are identical with some possible variations in formatting or comments; *Type-2* clones allow for variation in types, identifiers and literals; and *Type-3* clones further allow for inserted, modified, or deleted statements. The last type, *Type-4* clones, is based on similarity in function with variation in syntactical structure.

Clone Class

A code class is a group of code clones determined to be similar to each other under the same definition. Every code clone in the class is similar to every other code clone in the same class.

Near-miss Clones

Near-miss clones are *type-3* clones, meaning they may differ by a number of statements within a given tolerance threshold.

Potential Clone

A potential clone is term used by the NICAD clone detector for a code fragment that is extracted from the source code base to be compared in the clone analysis phase. It may be a file, a class, a function, or some arbitrary unit of source code. NICAD compares each potential clone to each other potential clone for similarity, organizes them into classes, and outputs the results.

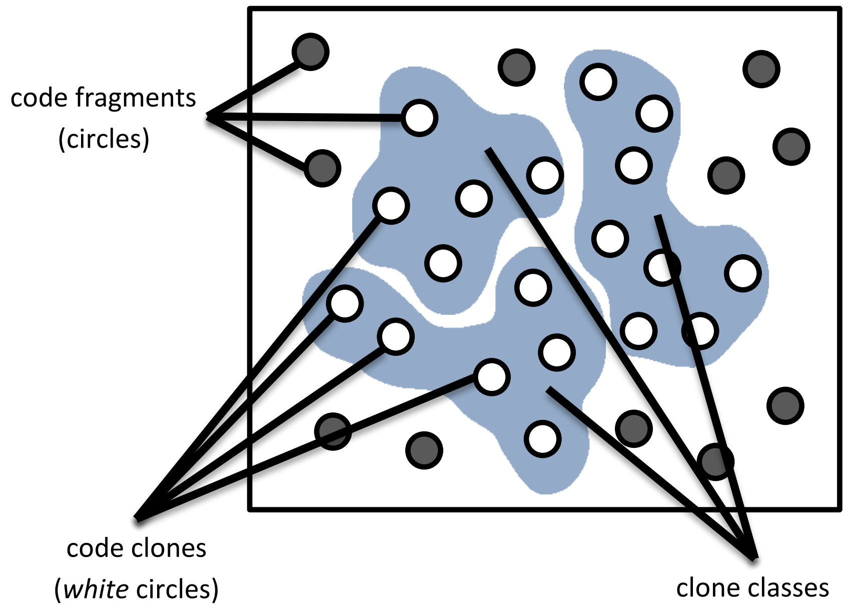

Figure 2.7 – *A visualization of clone detection terminology.*

The rectangle represents the source code base and the circles represent code fragments. The greyish areas represent clone classes (groups of similar code fragments).The circles inside them (white) are called clones because there exists at least one other code fragment similar to it.

2.3.2 NICAD

NICAD [11, 22] is a lightweight clone detector that uses textual techniques but gains the benefits of tree-based approaches of syntactical techniques by leveraging the agile parsing of TXL. This allows it to reformat potential clone fragments in a standardized way using TXL's pretty printing and perform normalization to abstract varying aspects (e.g. identifiers, expressions) so that similar code fragments with only minor differences are detected in the analysis phase.

NICAD has 3 phases – parse / extract, normalize, and compare. Each stage performs some operation on the code base or a set of potential clones (fragments to be compared) and passes the result to the next phase. These phases are shown in **Figure 2.8**.

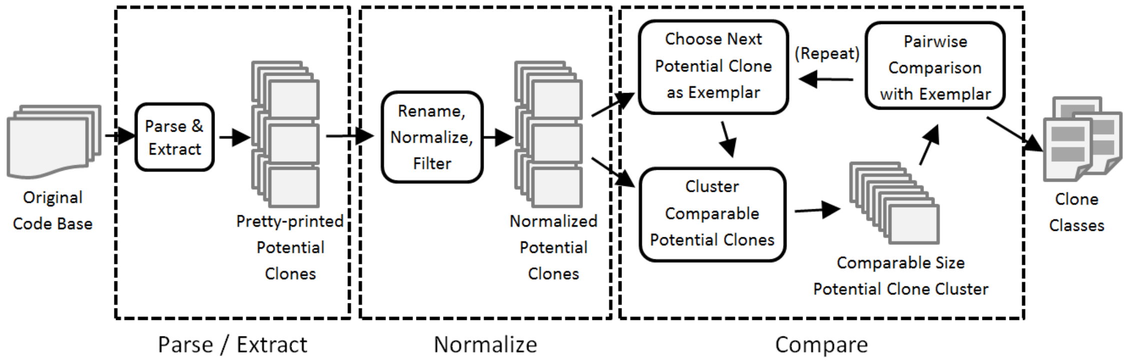
Original
Code Base
Parse & Extract
Pretty-printed Potential Clones
Rename, Normalize, Filter
Normalized Potential Clones
Choose Next Potential Clone as Exemplar
(Repeat)
Pairwise Comparison with Exemplar
Cluster Comparable Potential Clones
Comparable Size Potential Clone Cluster
Clone Classes
Parse / Extract
Normalize
Compare

Figure 2.8 – *NICAD architecture (based on diagram in [11]).*

NICAD consists of 3 phases – a parse / extract phase where potential clones are extracted and pretty printed using TXL; a normalize phase where TXL transformations can be applied to filter or rename parts of each potential clone; and a compare phase where the clone analysis is performed.

Phase 1 – Parse / Extract

The first phase is where potential clones are parsed and extracted at a particular granularity (e.g. functions, blocks) from the original source using TXL. In doing so, potential clones are reformatted in a standard way using TXL's pretty-printing such that different programmer coding styles (i.e. varying amounts of indents, newlines, spaces, and comments) can be ignored in the comparison phase. Newlines may be added to break up long lines of code for better identification of near-miss clones. For example, the header of a `for` loop (e.g. "`for (int i = 0; i < 10; i++)`") may be formatted such that each of the expressions delimited by semicolons are on separate lines. In this way, two for loop headers that differ by only one of these expressions may still be considered similar to each other when compared line-by-line.

This phase is language specific and NICAD has built in support for common languages, like C, C#, Java and Python, using two granularities, functions and blocks. If, however, the source code is written in another language or a different granularity is desired, NICAD's architecture allows new extractors, written in TXL, to be placed in the plugins directory with a name of the form *language*-*granularity*`-extract.txl`, where *language* is the name of the language (e.g. `wsdl`) and *granularity* is the name of the granularity (e.g. `functions`). When NICAD is called from the command line (e.g. `nicad functions wsdl ./wsdl-files`), it looks for a file with the name of the form above (e.g. `wsdl-functions-extractor.txl`) and uses it to parse and extract the source.

Phase 2 – Normalize

The second phase of NICAD is where the extracted code fragments can be further transformed to help find more near-miss clones. This phase is optional and not performed by default. Instead, NICAD allows the user to write their own TXL transformations and specify it as a *normalizer* in the configuration file. In addition, NICAD includes transformations for renaming identifiers (blind or consistent) and filtering (i.e. removing) or abstracting (i.e. replacing) certain syntactical structures.

Phase 3 – Compare

The final phase of NICAD is where the actual clone detection is performed on the generated potential clones. NICAD uses an efficient implementation of the Longest Common Subsequence (LCS) algorithm. Using the sequence of lines of two potential clones, it produces a new sequence of the lines containing the longest in-order sequence of lines that the two fragments have in common. This new sequence can be obtained by deleting lines from either of the potential clones. For example, consider the two sequences of elements (letters) shown in **Figure 2.9**. The sequences and their longest common subsequence are shown on the left. The diagram on the right shows which letters are part of the LCS (highlighted with boxes) and which letters are deleted (shown with lines through them).

After determining the longest common subsequence of two potential clones, NICAD calculates the percentage of unique items in each. A unique item is considered to be one that appears in the original code fragment but does not appear in the LCS. In this way, an item may be considered unique even if it exists in both code fragments but, because of the ordering, is not considered to be a common element as part of the LCS. If the percentage of unique items for both code fragments is below a given threshold (0%, 10%, 20%, and 30% are done by default), they are considered clones. A percentage of 0% for both indicates that the two code fragments are exact clones.

Two sequences:

```
A B C E F H I

A C D F G H I J
```

Longest Common Subsequence:

```
A C F H I
```

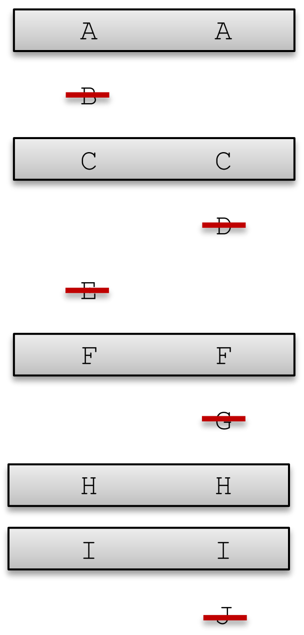

Figure 2.9 – *The longest common subsequence.*

The Longest Common Subsequence (LCS) of two sequences of characters (shown on the left). The LCS is the longest in-order sequence of characters that the two sequences have in common. By lining the sequences up (right) we can see which characters are in one sequence, but not the other.

Because of the expense of comparing every possible pair of potential clones (especially in large systems), NICAD only compares two code fragments if it is actually possible for them to be clones given their size and the threshold being considered. So, two fragments, *x* and *y*, are only considered if $size(x)$ - $size(x) * threshold \geq size(y) \leq size(x)$ - $size(x) * threshold$, where *threshold* is the current threshold being considered represented as a decimal between 0 and 1.

Once all potential clones have been compared, NICAD generates clone classes using the database of clone pairs. If `P1` and `P2` are clones, they form a clone class along with all other fragments that form a clone pair with `P2` and so on. Clone classes are outputted in an XML file or as a more viewable HTML page.

2.4 Finding Similar Services

Searching for similar services is not a new research topic. Others have tackled this problem over the past few years. The difference between these approaches and the one presented in this thesis, however, is that they all create new tools to solve the problem, whereas we attempt to leverage existing tools (clone detectors).

Dong et al. [12] developed a web service search engine called Woogle that gives the user the ability to perform a similarity search. Once a user finds a web service that is close to meeting their needs, they may search for services that are similar to it, take similar inputs, or compose with the given service. At the heart of this search is a clustering algorithm that groups the service's parameters into semantically similar concepts. This clustering algorithm uses the heuristic that parameters that occur together often tend to express the same concepts.

Syeda-Mahmood et al. [25] have explored the use of domain-independent and domain-specific ontologies when comparing service descriptions. Specifically, they looked at large company mergers or acquisitions where each company has its own set of web services that do similar things, but use different terminology. They used a number of techniques to aid in the search. First, they used word tokenization to separate multi-term parameter names (e.g. PartNumber) into its individual terms (e.g. PartNumber becomes "Part" and "Number"). Then, they used part-of-speech tagging and filtering to identify noun phrases and adjectives. Next, they expanded abbreviations (e.g. Cust becomes Customer). Finally, they used a synonym search with a thesaurus like WordNet to find synonyms for each word and assigned a similarity score based on how close the words were. They then use a matching algorithm to produce a ranked list of matching services and tested it using a domain-independent ontology, a domain-specific ontology, and none at all. What they found was that using a domain-specific ontology improved precision over a domain-independent ontology.

Stroulia et al. [26] developed a suite of methods designed to aid developers in the search for a suitable web service operation. They implemented 3 methods for this. First, when only a textual description of a service is available, they use a vector-space model to match the description with the text inside the service's `<documentation>` tags. A variation of this method uses WordNet to find synonyms (words with similar meaning), hypernyms (word parent), hyponyms (word children), and sibling senses (e.g. am, are, is) for the textual descriptions and apply scores based on how close they match. Second, when a stub of a web service is available and a structurally similar service is desired, they do structure matching. In this method, they compare data types, messages and operations of all pair-wise combinations from the source and target services. Finally, when a stub of a web service is available and a semantically similar service is desired, they do semantic structure matching. This method is an extension of the structure matching described above except instead of looking for compatible type mappings to find a syntactically similar service, they look for semantically compatible mappings to try to find a semantically similar service. This suite of methods solves the problem of service discovery based on different stages in a development process.

2.5 Summary

In this chapter, we have presented an overview of web services, their architectures, and description languages, particularly WSDL, used to describe them. We also looked at 2 source transformation tools, XSLT and TXL, and why we chose TXL over the more obvious XSLT. Next, we discussed clone detection, some terminology used throughout this thesis, and the clone detector chosen for this thesis, NICAD. Finally, we looked at approaches others have taken to find similar web services. We have also seen hints of why this may present a challenge when using clone detectors. In the next chapter, we will give an overview of our overall approach to solving this problem and how we accomplished it.

3 – Overview

While there have been a few attempts to find similar web services (see Chapter 2), the goal of this research is to leverage the efficiency and scalability of already existing clone detection (or similarity detection) tools by offloading the similarity search to them. But to do this, service descriptions, written in Web Service Description Language (WSDL) [6], must first be transformed to allow clone detection tools be used. A description of an operation in WSDL starts with an `<operation>` element, which contains elements that refer to other elements, and so on. This syntax splits the operation descriptions into delocalized pieces that make it difficult to identify suitable units to compare.

Clone detectors require contiguous sequences of code to use as potential clones in which to search for similarities. For typical programming languages, systems can be files, classes, methods, blocks or statement sequences depending on the desired level of granularity. However, the appropriate division is not clear for web services due to the segmentation of the operations. Entire WSDL descriptions can be used, but this only allows for comparison at the service level and not the desired operation level. The `<operation>` elements can be used, but they ignore other valuable information from the rest of the description and, other than common WSDL syntax, rarely contain anything more than the operation name.

This problem led us to develop a set of TXL [9, 10] transformations to *contextualize* `<operation>` elements by consolidating all of the pieces of each operation description into something we call *Web Service Cells*, or *WSCells* (pronounced "wizzles"). It also highlighted a current limitation to the NICAD clone detector [11, 22]. Currently, NICAD only allows TXL transformations to be performed after the extraction of potential clones, but like in the case of WSDL, there are times when we may wish to transform a code base before this phase.

In this thesis, we propose a new phase to the NICAD clone detector that performs optional TXL transformations to the source code base that would allow `<operation>` elements to be contextualized before the extraction of potential clones. We present WSDL as evidence to support such a phase, and show how we built a special WSDL extractor to emulate it in its absence. Finally, we performed an empirical analysis using two large sets of WSDL descriptions to see how clone detection on WSCells compares to using unprocessed `<operation>` elements.

3.1 NICAD Pre-Transformer

NICAD currently has 3 phases (see Section 2.3.2): a parse/extract phase where potential clones are extracted from the code base; a normalize phase where optional TXL transformations can be performed on the potential clones, such as renaming identifiers; and lastly the compare phase where potential clones are compared line-by-line to detect clones. The problem we discovered with this is that once potential clones are extracted, the rest of the code base is inaccessible. For example, with WSDL, this means that once we extract `<operation>` elements, other elements that make up the full description are no longer available. If we wish to contextualize `<operation>` elements, it must be done before they are extracted and passed to the normalize phase.

This illustrates the need for a phase before extraction, which we call the *NICAD Pre-Transformer*. The pre-transform phase can be thought of as a kind of preprocessor. But unlike some preprocessors, which perform simple tasks like renaming, the NICAD Pre-Transformer would perform user-defined TXL transformations capable of any sort of source transformation. In particular, it would support contextualization, which we describe later in this thesis for WSDL descriptions.

3.2 Contextualization

Since WSDL operation descriptions are split into delocalized pieces, we provide a set of TXL transformation rules to consolidate them into a self-contained unit. We call this process *contextualization* because it adds context to the minimal `<operation>` elements letting the clone detector analyze the big picture. We call the contextualized WSDL operations it produces *WSCells*.

The TXL rules for contextualizing WSDL take a single WSDL document, extract the base operation (the `<operation>` element), and insert the referenced elements into the element that references it. So for an `<input>` inside the operation, the corresponding `<message>` is found and inserted into it; for each `<part>` inside the `<message>`, the corresponding `<element>` is found and inserted; and so on, until there are no more elements left. The end result looks something like **Figure 3.1**.

```
<source file="HotelReservation.wsdl" startline="57" endline="61">
    <operation name="ReserveRoom" >
       <input message="tns1:ReserveRoomRequest">
           <message name="ReserveRoomRequest">
               <part name="body" element="xsd1:ReserveRoomRequest">
                   <element name="ReserveRoomRequest">
                       <element name="ccard" type="tns1:CreditCard”>
                           <element name="ccNumber" type="xsd:int"/>
                           <element name="cardHolder" type="xsd:string"/>
                           <element name="expiryDate" type="xsd:date"/>
                       </element>
                       <element name="room" type="tns1:Room">
                           <element name="roomID" type="xsd:int"/>
                           <element name="numBeds" type="xsd:int"/>
                           <element name="isSmoking" type="xsd:boolean"/>
                       </element>
                   </element>
               </part>
           </message>
       </input>
       <output message="tns1:ReserveRoomResponse">
           <message name="ReserveRoomResponse">
              <part name="body" element="xsd1:ReserveRoomResponse">
                   <element name="ReserveRoomResponse"/>
               </part>
           </message>
       </output>
       <fault message="tns1:RoomNotAvailableException">
           <message name="RoomNotAvailableException">
               <part name="body" element="xsd1:RoomNotAvailableException">
                   <element name="RoomNotAvailableException"/>
               </part>
           </message>
       </fault>
   </operation>
</source>
```

Figure 3.1 – *WSCell for a “ReserveRoom” operation.*

This is a fully constructed WSCell for a “ReserveRoom” operation of a simple hotel reservation web service. The service is discussed in Chapter 5 and is shown in Figure 5.1.

Once we wrote the WSDL contextualizing rules, we had to package it for NICAD. NICAD’s plugin architecture allows a TXL program that extracts potential clones to be written for any language that is not included out of the box. In the absence of the proposed NICAD Pre-Transformer, we have created a special extractor for WSDL that transforms operation descriptions into WSCells *during* the extraction phase. From there, we can simply invoke NICAD with the "`-wsdl`" flag and let it create WSCells before detecting clones.

3.3 Validation

The WSCells constructed by the special WSDL extractor are now suitable for the compare phase of NICAD (we do not use any normalizations here). Two sets of WSDL documents were used to test the consolidation method. The first set was a set of web services containing a number of identical or nearly identical service descriptions from a limited domain. The other set was obtained from the Seekda web service search engine [28]. It contained a large number of service descriptions from a very broad range of domains.

For each set, the results of NICAD on the set of WSCells were compared against the results of the set of base `<operation>` elements extracted from the `<portTypes>` element. In Chapter 6, we will look at some examples and show that using WSCells produces better results than the naïve approach.

3.4 Summary

In this chapter, we have given an overview of the work in this thesis. First, we proposed a new Pre-Transformer for the NICAD clone detector. Then, we presented WSDL as evidence for the need of such a phase, and described our contextualization method to turn WSDL descriptions into a set of contextualized operations, or WSCells. Finally, we briefly described how we plan on validating our approach with an empirical analysis of two sets of WSDL descriptions. In the next chapter, we will describe the NICAD pre-transform phase in more detail and discuss why it is needed to effectively detect contextual clones in WSDL.

4 – NICAD Pre-Transformer

As we have seen from the overview in the previous chapter, the poor locality of operation descriptions in WSDL [6] poses a unique challenge for clone detection. But by contextualizing the bare `<operation>` elements, we can create suitable units for comparison. NICAD [11, 22] allows TXL [9, 10] transformations to be performed on fragments of code, but only after they have been extracted from the code base. What we need for WSDL is a transformation before this extraction – a pre-transform phase or *pre-transformer*. In this chapter, we will show why we need to contextualize WSDL in greater detail, and introduce *contextual clones* – a new class of code clones invented for this problem. We go on to propose a modification to the NICAD architecture to support it – the pre-transformer – and describe how we emulated it for the purposes of this thesis.

4.1 Contextual Clones in WSDL

WSDL descriptions of web services highlight the need for an optional transformation step before code extraction in NICAD and other clone detectors. Its scattered syntax and semantics makes it difficult to identify appropriate units for comparison (potential clones in NICAD), and simple clone detection on WSDL operation descriptions does not provide any useful answers. This has led to the development of a new class of code clones we call *contextual clones* – clones found by embedding remote referenced information before comparison.

4.1.1 Challenge with WSDL

As we saw in Chapter 2, a WSDL description of a web service contains specifications of one or more operations that the service provides. At the heart of the service description is the `<portTypes>` element, which contains a list of operations and the inputs, outputs and faults handled by each. However, the `<input>` and `<output>` elements rarely provide any useful information by themselves (in most cases, their names are the same as the operation name with “Request” or “Response” appended to the end). Their purpose is to reference other elements of the WSDL description that provide type information and describe the parameters the operation expects and what can be expected in return.

This scattered referential form makes it difficult for a clone detector to identify appropriate units (potential clones) for comparison. Comparing entire WSDL descriptions doesn't allow us to identify similarity at the operation level; if two services share a similar operation but the remainder of the services is different enough, a clone detector may ignore it, yielding a low recall level. On the other hand, if we compare at the operation description level, we ignore the type and parameter information described elsewhere in the service description, yielding a high level of false positives.

To illustrate this, consider the two `GetStock` operations taken from two different service descriptions, shown in **Figure 4.1a**. Looking at these operations, we might conclude that they are clones; in fact, we would probably agree that these are exact clones. However, by looking at the rest of the service descriptions in which they are embedded – or contextualizing the operations – we see that the use of the word "stock" has two completely different meanings in the two services (**Figure 4.1b**). The first service uses "stock" to mean inventory, while the second uses "stock" to mean a financial stock on the stock market. Thus ignoring contextual information can lead us to falsely identify operations as clones.

With this in mind, it becomes difficult to choose appropriate units to act as potential clones. We could use the entire service description, but that would not provide the granularity to compare services at the operation level. It would also mean that two services with a single similar operation might be ignored if the remaining operations were different enough. To achieve a finer level of granularity, we could use the `<operation>` elements from the `<portTypes>` element like those shown in **Figure 4.1a**. The problem with this is that, while it gives us the name of the operation, it ignores all of the associated parameter information from the `<types>` element. All of the pieces of an operation description need to be used together to accurately compare it to other operations. But since these pieces do not occur together, the whole WSDL description must be reorganized in order to extract them as a single unit.

a) b)

```
<operation name="GetStock" >
  <input message="GetStockRequest" />
  <output message="GetStockResponse" />
</operation>
```

```
<complexType name="Stock">
  <sequence>
    <element name="Supplier" type="string"/>
    <element name="Warehouse" type="string"/>
    <element name="OnHand" type="string"/>
    <element name="OnOrder" type="string"/>
    <element name="Demand" type="string"/>
  </sequence>
</complexType >
```

```
<operation name="GetStock" >
  <input message="GetStockRequest" />
  <output message="GetStockResponse" />
</operation>
```

```
<complexType name="Stock">
  <sequence>
    <element name="date" type="string"/>
    <element name="open" type="float"/>
    <element name="high" type="float"/>
    <element name="low" type="float"/>
    <element name="close" type="float"/>
    <element name="volume" type="float"/>
  </sequence>
</complexType >
```

Figure 4.1 – *Two "GetStock" operations.*

Two "GetStock" <operation> elements look identical when compared (a), but refer to elements from two different domains (b).

4.1.2 Contextual Clones

This challenge led us to define a new class of code clones, those found by adding contextual information to a code fragment. Contextualized code is created by expanding parts of a code fragment to include information from elsewhere, giving meaning to something that may have little meaning otherwise. This information may be explicit, like a reference to an object, or implicit, like the location of a file or a non-standard definition from a dictionary. The newly expanded fragments are used as potential clones and given to a clone detector. The clones found in this way are called *contextual clones*.

Definition 1: *Contextualized Code*. **Contextualized code** is a code fragment that has been modified or expanded to include information referenced, either explicitly or implicitly by context, from elsewhere in the code or environment.

Definition 2: *Contextual Clone*. A **contextual clone** is a code clone found by comparing contextualized code fragments as potential clones. Contextual clones are rarely the result of copying and pasting, but rather indicate higher-level relationships between fragments.

Contextualized clone detection, using a textual approach, can detect near-miss clones better than tree- or graph-based methods that look for matching subtrees or subgraphs, even if they follow references. For example, consider two identical code fragments. Suppose an if-statement is added to one, leaving everything else unchanged. We still have two nearly identical fragments; however, their syntactical structure is now different. This may confuse tree-based approaches, but for text-based approaches, they simply differ by two lines (possibly less). In this way, contextual clone detection may produce a higher recall value, while keeping the same precision.

Code fragments can be contextualized using TXL, which is built into NICAD. However, it currently only supports transformations on already extracted code fragments, too late for contextualization as the other pieces are discarded after extracting one type of nonterminal. This difficulty led to the idea of an additional phase at the beginning of NICAD.

4.2 NICAD Architecture Modification

Recall the NICAD architecture from Chapter 2 (Section 2.3.2). It begins with the parse / extract phase where potential clones are extracted from the original code base to be compared. It comes packaged with a number of extractors for common programming languages at various granularities, but new ones can be written without any modification to NICAD itself.

A NICAD extractor consists of 3 parts: a TXL grammar for the desired language; TXL transformation rules to extract potential clones; and a bash script to call the TXL transformation. The TXL portion consists of rules to extract the nonterminals for the desired granularity and surround them in `<source>` tags with the filename and line numbers as attributes. The bash script finds all source language files in a given folder (parameter passed by NICAD script) and applies the TXL transformation to each. All TXL files go in the "`txl`" folder and can be named anything, while the bash script is placed in the "`scripts`" folder with a name of the form

“`Extract`-*granularity*-*language*.” When NICAD is invoked with “`nicad` *granularity language dir*,” it will look for the script of that form and use it as the extractor.

This works for most languages but, as we have seen with WSDL, some circumstances call for transformations before appropriate units can be extracted. What we propose is a new phase at the beginning, we call the pre-transform phase, where TXL transformations can be applied to the source code base *before* potential clones are extracted. **Figure 4.2** shows an updated NICAD architecture with this new phase.

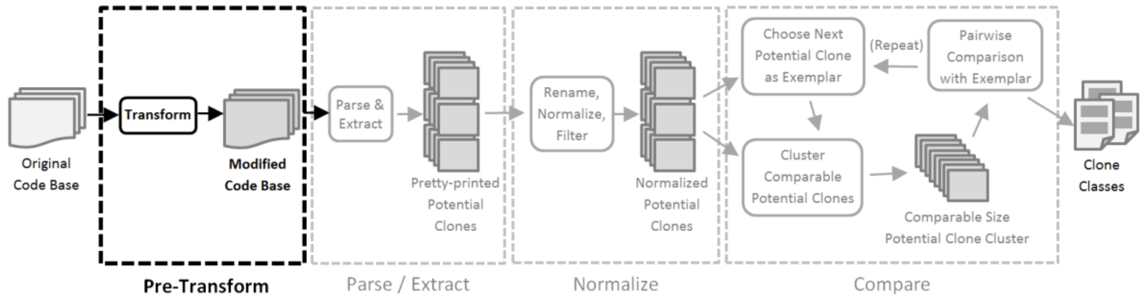
Original Code Base
Transform
Modified Code Base
Pre-Transform
Parse & Extract
Pretty-printed Potential Clones
Parse / Extract
Rename, Normalize, Filter
Normalized Potential Clones
Normalize
Choose Next Potential Clone as Exemplar
(Repeat)
Pairwise Comparison with Exemplar
Cluster Comparable Potential Clones
Comparable Size Potential Clone Cluster
Compare
Clone Classes

Figure 4.2 – *The proposed new NICAD architecture.*

The proposed new phase for NICAD (highlighted) performs an optional transformation on the code base before potential clones are extracted (based on diagram in [11]).

4.3 Implementation

This new phase, like the parse / extract phase, would allow users to write restructuring transformations to contextualize code fragments before extraction. Like other NICAD plugins, these transformations can be added to the TXL plugins folder of NICAD with a special name to indicate their role, and can then be specified in NICAD configuration files to control which ones are used in any particular NICAD run. They may be named using the convention "`PT-`*pretransformname*-*language*`.txl`", for example "`PT-contextualize-wsdl.txl`."

Since this proposed new phase requires an architectural modification to NICAD itself, for the purposes of this thesis, we have implemented a special WSDL extractor that transforms `<operation>` elements into WSCells (see Chapter 4) while they are being extracted so that the whole WSDL description is still accessible. This approximation gives us the results we want for the purposes of this thesis without the necessity of re-architecting NICAD.

While this works for WSDL, it is not what we want to do conceptually and may not work for other languages or situations where context may need to be taken from outside the file containing the base unit (e.g. `<operation>` element in WSDL). We would also like to make this new concept – contextualizing pre-transformations in clone detection – explicit in order to allow other developers and researchers to explore other applications that use this idea.

4.4 Summary

In this chapter we have proposed a new phase to the NICAD architecture called the pre-transform phase, allowing contextualizing source transformations to be applied to a source code base before potential clones are extracted. We also mentioned how we can embed transformations, which piece together operation descriptions, into a NICAD extractor plugin for WSDL to circumvent its current limitation without modifying NICAD. The following chapter will go over these transformations in detail.

5 – Contextualization

Similarity detection tools require units of code (e.g. fragments, blocks, etc.) to compare as potential clones. WSDL [6] poses a unique challenge because it is a description language, not a traditional programming language, and as such, must be given special consideration. In Chapter 4, we proposed a new pre-transform phase for the NICAD clone detector [11, 22] that would support contextualizing WSDL operations by grouping related pieces of the operation descriptions together. We also showed how we could insert TXL [9, 10] transformation rules into a NICAD extractor to emulate this for WSDL in the absence of such a phase. In this chapter, we will go over these transformation rules in detail. The result is the second contribution of this thesis – a transformation of WSDL descriptions into a set of the service's `<operation>` elements with all related pieces contained within.

5.1 Contextualized WSDL Operations (WSCells)

Before getting into the details, it is important to understand what we are trying to accomplish. Operation descriptions in WSDL are divided into pieces. Each input/output is associated with a message that contains the input and output parameters, respectively. The parameters have types that are either primitive (e.g. `string`, `int`, etc.) or user-defined. The problem is that these are all separated and co-mingled with similar pieces of other operation descriptions and may even share common elements. **Figure 5.1** contains a WSDL description of a simple hotel reservation service with one operation called `ReserveRoom`. Notice that the operations, messages and type definitions are all grouped together forming sections for each kind. A full operation description is comprised of pieces in each of these sections. What we do in the extractor is contextualize the `<operation>` elements by consolidating all of these pieces into a self-contained unit that can be compared with others composed in the same way.

```
<?xml version="1.0"?>
<definitions name="HotelReservationService"
             targetNamespace="http://myhotel.com/reservationservice.wsdl"
             xmlns:tns1="http://myhotel.com/reservationservice.xsd"
             xmlns:xsd1="http://myhotel.com/reservationservice.xsd"
             xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
             xmlns="http://schemas.xmlsoap.org/wsdl/">
     <types>
          <schema targetNamespace="http://myhotel.com/reservationservice.xsd"
                  xmlns="http://www.w3.org/2000/10/XMLSchema">
               <complexType name="Room">
                    <sequence>
                         <element name="roomID" type="xsd:int"/>
                         <element name="numBeds" type="xsd:int"/>
                         <element name="isSmoking" type="xsd:boolean"/>
                    </sequence>
               </complexType>
               <complexType name="CreditCard">
                    <sequence>
                         <element name="ccNumber" type="xsd:int"/>
                         <element name="cardHolder" type="xsd:string"/>
                         <element name="expiryDate" type="xsd:date"/>
                    </sequence>
               </complexType>
               <element name="ReserveRoomRequest">
                    <complexType>
                         <sequence>
                              <element name="ccard" type="tns1:CreditCard"/>
                              <element name="room" type="tns1:Room"/>
                         </sequence>
                    </complexType>
               </element>
               <element name="ReserveRoomResponse">
                    <complexType>
                         <sequence />
                    </complexType>
               </element>
               <element name="RoomNotAvailableException">
                    <complexType>
                         <sequence />
                    </complexType>
               </element>
          </schema>
     </types>

     <message name="ReserveRoomRequest">
          <part name="body" element="xsd1:ReserveRoomRequest"/>
     </message>
     <message name="ReserveRoomResponse">
          <part name="body" element="xsd1:ReserveRoomResponse"/>
     </message>
     <message name="RoomNotAvailableException">
          <part name="body" element="xsd1:RoomNotAvailableException"/>
     </message>

     <portType name="HotelReservationServicePortType">
          <operation name="ReserveRoom">
               <input message="tns1:ReserveRoomRequest"/>
               <output message="tns1:ReserveRoomResponse"/>
               <fault message="tns1:RoomNotAvailableException"/>
          </operation>
     </portType>
```

Figure 5.1 – *Example WSDL description of a simple hotel reservation service.*

This WSDL description of a hotel reservation service contains a single operation called "ReserveRoom," which takes a room and a credit card as input and returns an acknowledgment upon completion or a "RoomNotAvailableException" fault if the room is not available.

We refer to the newly contextualized WSDL operations as *Web Service Cells*, or *WSCells* (pronounced “wizzles”), because they are the pieces that together form a whole web service description. Although designed to be used with clone detectors to find similarities, they can also be useful for better understanding web service operations because of the way that they gather and organize pieces of WSDL to form a set of whole operation descriptions. **Figure 5.2** shows the resulting WSCell generated by the WSDL extractor for the `ReserveRoom` operation in the previous example.

```
<source file="HotelReservation.wsdl" startline="57" endline="61">
    <operation name="ReserveRoom" >
       <input message="tns1:ReserveRoomRequest">
           <message name="ReserveRoomRequest">
               <part name="body" element="xsd1:ReserveRoomRequest">
                   <element name="ReserveRoomRequest">
                       <element name="ccard" type="tns1:CreditCard”>
                           <element name="ccNumber" type="xsd:int"/>
                           <element name="cardHolder" type="xsd:string"/>
                           <element name="expiryDate" type="xsd:date"/>
                       </element>
                       <element name="room" type="tns1:Room">
                           <element name="roomID" type="xsd:int"/>
                           <element name="numBeds" type="xsd:int"/>
                           <element name="isSmoking" type="xsd:boolean"/>
                       </element>
                   </element>
               </part>
           </message>
       </input>
       <output message="tns1:ReserveRoomResponse">
           <message name="ReserveRoomResponse">
              <part name="body" element="xsd1:ReserveRoomResponse">
                   <element name="ReserveRoomResponse"/>
               </part>
           </message>
       </output>
       <fault message="tns1:RoomNotAvailableException">
           <message name="RoomNotAvailableException">
               <part name="body" element="xsd1:RoomNotAvailableException">
                   <element name="RoomNotAvailableException"/>
               </part>
           </message>
       </fault>
   </operation>
</source>
```

Figure 5.2 – *WSCell for “ReserveRoom” operation.*

This is a complete WSCell for the “ReserveRoom” operation from the simple hotel reservation service in Figure 5.1. All referenced elements have been inserted in the elements that reference them, giving a structured description of the operation.

WSCells contain all of the information related to a web service operation provided by the description except where to invoke it (found in the `<binding>` and `<service>` elements). These were excluded because they are unnecessary and could interfere with finding similarities because every service has a different URL. The location of the web service can always be found by using the information found in the `<source>` tags to trace the operation back to the original WSDL description.

5.2 TXL for WSDL

NICAD extractor plugins are written with TXL, a source transformation language discussed in Chapter 2. In our work we use it to build a WSDL extractor that parses a source WSDL file, transforms its operation descriptions into WSCells by merging the referenced parts from other sections, and extract the consolidated WSCells as the code fragments to be compared. Each TXL program has two components: a grammar, and a set of transformation rules. This section will go over each of these components for WSDL in detail.

5.2.1 WSDL Grammar

Since no TXL grammar for WSDL existed, one had to be crafted for the purposes of this work. The existing XML grammar was not appropriate as it would not allow easy referencing of specific types of WSDL elements (e.g. complex types, operations, inputs, outputs, etc.) required to build new restructured operations. In other words, using a generic XML grammar would mean having to filter specific WSDL elements in transformation rules, whereas using a WSDL-specific grammar offloads this responsibility to the TXL parser. The same can be said for other XML languages with a predefined structure; if the structure is known beforehand, it makes sense to define a new grammar.

The WSDL grammar is quite straightforward, assuming an understanding of the structure of WSDL descriptions (see Chapter 2). **Figure 5.3** shows the beginning of the grammar, mainly the TXL `program` definition; the full grammar is listed in Appendix A. A WSDL `program` consists of an XML declaration, standard of any XML-based document, and a “definitions” element, the root of every WSDL description. A definitions element contains top-level elements, which represent the various elements permitted to be inside (i.e., types, messages, port types, bindings, and services). These elements are all defined in a similar way.

```
tokens
        id
    |   "\a[-\i]*"
end tokens

% All types of documentation are treated as comments and ignored.
comment
    <!--  -->
    <xs:annotation> </xs:annotation>
    <documentation  </documentation>
    <wsdl:documentation  </wsdl:documentation>
    <xs:documentation  </xs:documentation>
    <xsd:documentation  </xsd:documentation>
end comments

% Main TXL "program"
define program
    [opt xml_def]
    [definitions]
end define

% XML declaration
define xml_def
    [SPOFF] '<?xml [attribute_list] '?> [SPON]                              [NL]
end define

% The <definitions> element is the root element of every WSDL description
define definitions
    [SPOFF] '< [opt prefix] 'definitions [opt attribute_list] '> [SPON][NL][IN]
        [repeat top_level_elements]                                         [EX]
    [SPOFF] '</ [opt prefix] 'definitions> [SPON]                           [NL]
end define

% The <definitions> element contains <types>, <message>, <portType>
% (which contain <operation> elements), <binding>, and <service> elements.
define top_level_elements
        [types]
    |   [message]
    |   [port_type]
    |   [binding]
    |   [service]
    |   [element]
end define
```

Figure 5.3 – *Excerpt of the TXL WSDL grammar.*

This is the beginning of the WSDL grammar for our extractor where the root <definitions> element is declared. The full grammar is listed in Appendix A.

5.2.2 Transformation Rules

As with any TXL program, the heart of our extractor lies in the transformation rules. Beginning with the operation description within the `<portTypes>` element and working its way down the hierarchy, each rule consolidates the elements associated with a certain kind of element (e.g., input, message, part, etc.) until it cannot go any further. The end result is an operation description, with all elements associated with it, structured in a way to reflect the hierarchy of elements.

A series of helper rules are applied to the consolidated contents of each part that remove elements that clutter up the description. Some, like leftover `<complexType>` tags, can be inferred from the structure. Others, like empty `<sequence>` elements, provide no new meaning (likely the result of automated generation of WSDL descriptions from some tool).

But before rules can be written, a few modifications to the WSDL grammar must be made specifically for this task, since the operations produced are not valid WSDL. For example, input elements are not permitted to contain message elements, so the definition must be updated to reflect that. A new source element must be defined to contain the consolidated operation and also provide information about where it came from (i.e. what is the name and path of the WSDL file). The program definition of a WSDL “program” must also be modified; no longer is it a single definitions element, but a list of source elements. These changes are shown in **Figure 5.4**.

The transformation rules are broken down into roughly one function for every type of element that then calls another function to get its contents and return a new element. Most of the functions find a referenced element using a technique where the function is called repeatedly with a new element from a list of possible elements. If the given element matches some criteria, it gets appended to a running list (initially empty). This new list then forms the contents of a new element.

```
% Define <source> element to wrap around newly constructed operations.
define source
    [SPOFF] '<'source [opt attribute_list] '> [SPON]                      [NL][IN]
        [repeat element]                                                  [EX]
    [SPOFF] '</'source> [SPON]                                            [NL]
end define

% Redefine <operation> elements so filename and begin/end line nums are known.
redefine operation
    [attr srcfilename] [attr srclinenumber]
    [SPOFF] '< [opt prefix] 'operation [opt attribute_list] [SPON] '>     [NL][IN]
        [repeat operation_scope]                                          [EX]
    [attr srclinenumber]
    [SPOFF] '</ [opt prefix] 'operation> [SPON]                           [NL]
end redefine

% Redefine a WSDL program to allow a list of <source> elements.
redefine program
        ...
    |   [repeat source]
end redefine

% Allow all types of elements to be inserted into these during contextualization.
redefine complex_type
        ...
    |   [SPOFF] '< [opt prefix] 'complexType [opt attribute_list] '> [SPON][NL][IN]
            [repeat element]                                              [EX]
        [SPOFF] '</ [opt prefix] 'complexType> [SPON]                     [NL]
end redefine

redefine simple_type
    [SPOFF] '< [opt prefix] 'simpleType [opt attribute_list] '> [SPON]    [NL][IN]
        [repeat element]                                                  [EX]
    [SPOFF] '</ [opt prefix] 'simpleType> [SPON]                          [NL]
end redefine

redefine element
        ...
    |   [SPOFF] '< [opt prefix] 'element [opt attribute_list] '> [SPON]   [NL][IN]
            [repeat element]                                              [EX]
        [SPOFF] '</ [opt prefix] 'element> [SPON]                         [NL]
end redefine
```

Figure 5.4 – *Changes to the WSDL grammar.*

These redefine statements at the beginning of the TXL transformation rules (listed in Appendix B) allow any kind of element to be inserted into some elements that would not be allowed otherwise and <source> tags to be used for NICAD.

Figure 5.5 shows the contextualization process where each box roughly translates to one TXL function. We summarize these functions here, using the service in **Figure 5.1**, from the beginning of this chapter, as an example to illustrate what is happening at each stage to produce the WSCell shown in **Figure 5.2**. A listing of all the TXL rules is available for reference for Appendix B, along with the full grammar for WSDL in Appendix A.

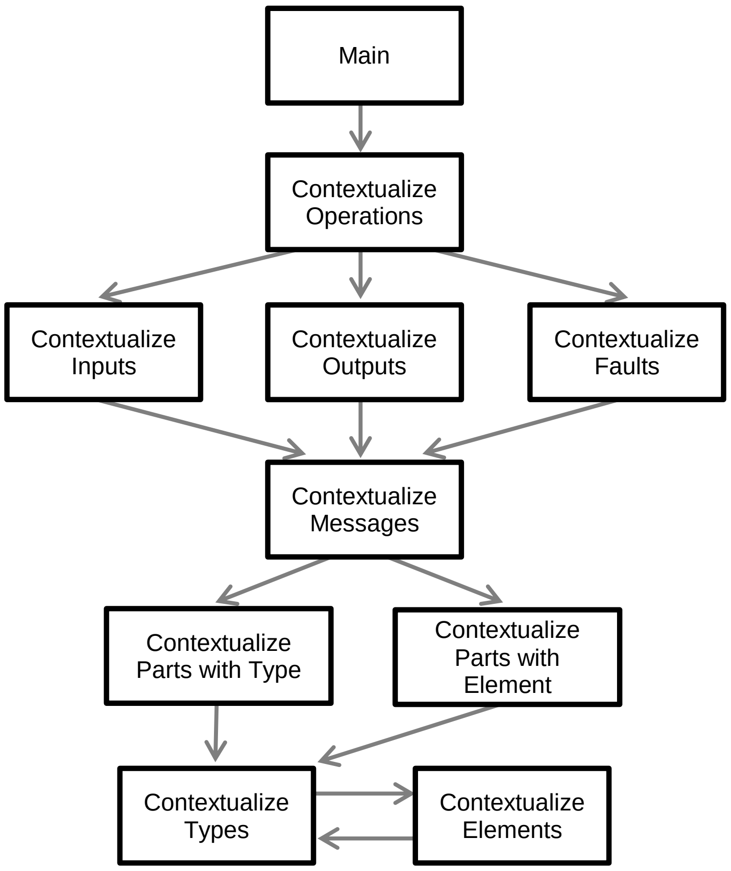

Figure 5.5 – The c*ontextualization process.*

This diagram illustrates the process of contextualizing an <operation> element into a WSCell with each box representing a stage in the process starting with the main function. Most stages consist of one function, with some (such as "Contextualize Types") representing multiple functions.

Main

Every TXL program begins with a `main` function that parses the input program and allows transformations to be performed on it. In our WSDL extractor, the `main` function deconstructs the WSDL description and exports the namespaces and schema for use in other functions. More importantly, it extracts the `<operation>` elements from the `<portTypes>` element and calls the `contextualizeOperations` function with each operation to transform the original WSDL description into a list of contextualized operations, or WSCells.

Contextualize Operations

The `contextualizeOperations` function takes an `<operation>` element and transforms it into a contextualized operation. It matches a list of elements (i.e. contextualized operations) and replaces it with an identical list with the newly formed operation appended to it.

To construct the new contextualized operation, it first extracts the `<input>`, `<output>`, and `<fault>` elements from the original operation and calls functions specific to each one (`contextualizeInputs` for input tags, etc.) to contextualize them. The newly formed elements are then joined together and inserted into an `<operation>` element with the same attributes as the original. The newly contextualized operation is surrounded by `<source>` tags that contain the filename of the WSDL description it comes from, as well as well as the beginning and end line numbers of the original operation element. This extra tag allows NICAD to trace back to the original operation after clone detection is performed. **Figure 5.6** shows the WSCell from our example hotel reservation service at this point, with the tags added at this stage highlighted.

```
<source file="HotelReservation.wsdl" startline="57" endline="61">
    <operation name="ReserveRoom" >
        ( contextualized <input> )
        ( contextualized <output> )
        ( contextualized <fault> )
   </operation>
</source>
```

Figure 5.6 – *Construction of the "ReserveRoom" WSCell (1/6).*

This is a representation of a partial "ReserveRoom" WSCell produced by the contextualizeOperations function. Tags that are added by this stage are highlighted. Placeholders are used for elements that have yet to be contextualized.

Contextualize Inputs/Outputs/Faults

The functions for consolidating an operation's inputs, outputs and faults are almost identical. In each case, the `message` attribute is extracted, which gives the name of a `<message>` element declared somewhere else in the description. It calls the `contextualizeMessages` function with each `<message>` element (extracted and exported in the main function), which finds and contextualizes the matching message. This message is then surrounded with `<input>`, `<output>` or `<fault>` tags (depending on which function is called) with the original attributes.

Once each input, output, and fault has been contextualized, they are joined together in a list by the `contextualizeOperations` function and inserted into the new contextualized operation. **Figure 5.7** shows the WSCell at this point, with the tags added by each function highlighted.

```
<source file="HotelReservation.wsdl" startline="57" endline="61">
    <operation name="ReserveRoom" >
       <input message="tns1:ReserveRoomRequest">
           ( contextualized <message> )
       </input>
       <output message="tns1:ReserveRoomResponse">
           ( contextualized <message> )
       </output>
       <fault message="tns1:RoomNotAvailableException">
           ( contextualized <message> )
       </fault>
   </operation>
</source>
```

Figure 5.7 – *Construction of the "ReserveRoom" WSCell (2/6).*

This is a representation of a partial "ReserveRoom" WSCell produced by the contextualizeInputs, contextualizeOutputs and contextualizeFaults functions. Tags that are added by this stage are highlighted. Placeholders are used for elements that have yet to be contextualized.

Contextualize Messages

The functions outlined in the previous section call the `contextualizeMessages` function with the target message name and each `<message>` element in the file. Each iteration of the `contextualizeMessages` function checks to see if the `<message>` element's name matches the target name. If it does, it calls the `contextualizePartsWithElement` and the `contextualizePartsWithType` functions (see next section) with each part contained inside the matching `<message>` element. **Figure 5.8** shows the WSCell at this point.

```
<source file="HotelReservation.wsdl" startline="57" endline="61">
    <operation name="ReserveRoom" >
       <input message="tns1:ReserveRoomRequest">
           <message name="ReserveRoomRequest">
               ( contextualized <part> )
           </message>
       </input>
       <output message="tns1:ReserveRoomResponse">
           <message name="ReserveRoomResponse">
               ( contextualized <part> )
           </message>
       </output>
       <fault message="tns1:RoomNotAvailableException">
           <message name="RoomNotAvailableException">
               ( contextualized <part> )
           </message>
       </fault>
   </operation>
</source>
```

Figure 5.8 – *Construction of the "ReserveRoom" WSCell (3/6).*

This is a representation of a partial "ReserveRoom" WSCell produced by the contextualizeMessages function. Tags that are added by this stage are highlighted. Placeholders are used for elements that have yet to be contextualized.

Contextualize Parts

`<message>` elements contain one or more `<part>` elements, each of which can be constructed in one of two ways. First, it could have an `element` attribute that points to an `<element>` element defined elsewhere in the description. Second, it could have a `type` attribute that either refers to an XML primitive type or a user-defined type (`<complexType>` or `<simpleType>`). The `contextualizeMessages` function calls 2 different functions – one for each case – with each `<part>` element. Each function will match one of the `<part>` elements and contextualize it.

The first of these functions is the `contextualizePartsWithElement`. As the name suggests, it matches `<part>` elements with an `element` attribute and contextualizes them. The attribute refers to the name of an `<element>` element in the XML schema, which has a type or contains the parameters of the operation. In this case, there will be only one `<part>` element to contextualize, so the `getMatchingSchemaElement` function is called with the target element name and each `<element>` element from the appropriate schema. Our hotel reservation service example contains only `<part>` elements with `element` attributes, so this function matches them. The resulting WSCell at this point is shown in **Figure 5.9**.

```
<source file="HotelReservation.wsdl" startline="57" endline="61">
    <operation name="ReserveRoom" >
       <input message="tns1:ReserveRoomRequest">
           <message name="ReserveRoomRequest">
               <part name="body" element="xsd1:ReserveRoomRequest">
                  ( contextualized ReserveRoomRequest <element> )
               </part>
           </message>
       </input>
       <output message="tns1:ReserveRoomResponse">
           <message name="ReserveRoomResponse">
               <part name="body" element="xsd1:ReserveRoomResponse">
                  ( contextualized ReserveRoomResponse <element> )
               </part>
           </message>
       </output>
       <fault message="tns1:RoomNotAvailableException">
           <message name="RoomNotAvailableException">
               <part name="body" element="xsd1:RoomNotAvailableException">
                  ( contextualized RoomNotAvailableException <element> )
               </part>
           </message>
       </fault>
   </operation>
</source>
```

Figure 5.9 – *Construction of the "ReserveRoom" WSCell (4/6).*

This is a representation of a partial "ReserveRoom" WSCell produced by the contextualizeParts function. Tags that are added by this stage are highlighted. Placeholders are used for elements that have yet to be contextualized.

Instead of referring to another element that contains the parameters, `<part>` elements may themselves be used as parameters. In this case, they contain a `type` attribute that may refer to a `<complexType>` or `<simpleType>` defined in an XML schema or one of the XML primitive types (e.g. `string`, `int`, etc.). Either way, this function calls the `getMatchingType` function, which returns a contextualized `<part>` if there is a match or the original `<part>` otherwise.

Put simply, the only difference between these two functions is the kind of matching element they search for (i.e. `<element>` or `<complexType>`). Some descriptions use `<part>` elements as parameters with a type, while others use them to refer to another element that contains the parameters. Both functions end up calling the `contextualizeTypes` function, described in the next section.

Get Matching Elements/Types

The rest of the TXL rules contextualize elements from the schema defined in the `<types>` element at the beginning of the description. This portion of the description defines the parameters of the operation, including the definitions of each user-defined type. It also contains `<element>` elements that are referred to by `<part>` elements, as in our hotel reservation service example. The `getMatchingSchemaElement` function finds this element and contextualizes any elements contained within. If it is empty, meaning that the message is simply an acknowledgement, it returns it as a singleton element (one that consists of a single tag); if not, it extracts and contextualizes each of the elements inside. Our example contains both cases, and the result of this function is shown in **Figure 5.10**.

```
<source file="HotelReservation.wsdl" startline="57" endline="61">
  <operation name="ReserveRoom" >
    <input message="tns1:ReserveRoomRequest">
      <message name="ReserveRoomRequest">
        <part name="body" element="xsd1:ReserveRoomRequest">
          <element name="ReserveRoomRequest">
            ( contextualized CreditCard <element> )
            ( contextualized Room <element> )
          </element>
        </part>
      </message>
       </input>
       <output message="tns1:ReserveRoomResponse">
      <message name="ReserveRoomResponse">
        <part name="body" element="xsd1:ReserveRoomResponse">
          <element name="ReserveRoomResponse" />
        </part>
      </message>
       </output>
       <fault message="tns1:RoomNotAvailableException">
      <message name="RoomNotAvailableException">
        <part name="body" element="xsd1:RoomNotAvailableException">
          <element name="RoomNotAvailableException" />
        </part>
      </message>
       </fault>
   </operation>
</source>
```

Figure 5.10 – *Construction of the “ReserveRoom” WSCell (5/6).*

This is a representation of a partial “ReserveRoom” WSCell produced by the getMatchingSchemaElement function. Tags that are added by this stage are highlighted. Placeholders are used for elements that have yet to be contextualized.

`<element>` elements can have types associated with them, which are defined using `<complexType>` elements for types that contain other `<element>` elements, or `<simpleType>` elements to place restrictions on XML primitive types. `<simpleType>` elements do not need to be contextualized because their contents do not reference any other elements. `<complexType>` elements, however, contain `<element>` elements that may have types that are defined with other `<complexType>` elements. Those types may contain elements with types defined in another `<complexType>` element, and so on.

To contextualize types, we use 2 functions that form a recursive loop. First, the `contextualizeTypes` function contextualizes an `<element>` element by applying the `contextualizeComplexTypes` function to each `<complexType>` element in order to find the one that matches the `type` attribute. That function, after finding the matching type, extracts all of the elements from the type definition and contextualizes them by calling the `contextualizeTypes` function again. This loop continues until a matching type definition cannot be found (the base case), indicating that the type may be a primitive XML type, or that the definition may be in another externally reference XML schema. **Figure 5.11** shows the final WSCell for the `ReserveRoom` operation from our example, highlighting the consolidated elements found during this stage.

This example operation requires at most 2 cycles through the loop, but this is a small example. Larger services may have dozens of nested elements, depending on the size of their data structures. None of the WSDL descriptions we tested contained recursive types (i.e. a type that contains an element with the type of an ancestor), so we chose not to handle them. However, this is permitted by the specification of XML Schema [2] and, therefore, the WSDL specification [5, 6]. With the current implementation of the extractor, shown in Appendix B, a recursive type definition would result in an infinite loop. A better implementation would recursively insert type definitions, as described above, but stop if it reaches one that it has seen before.

Another limitation to this implementation is that it only considers type definitions within the `<types>` element. Some WSDL descriptions contain references to external XML Schema, available at a given URL. A better implementation would retrieve these external definitions and include them in the contextualization, possibly by inserting them into the `<types>` element before any rules are applied.

```
<source file="HotelReservation.wsdl" startline="57" endline="61">
    <operation name="ReserveRoom" >
       <input message="tns1:ReserveRoomRequest">
           <message name="ReserveRoomRequest">
               <part name="body" element="xsd1:ReserveRoomRequest">
                   <element name="ReserveRoomRequest">
                       <element name="ccard" type="tns1:CreditCard">
                           <element name="ccNumber" type="xsd:int"/>
                           <element name="cardHolder" type="xsd:string"/>
                           <element name="expiryDate" type="xsd:date"/>
                       </element>
                       <element name="room" type="tns1:Room">
                           <element name="roomID" type="xsd:int"/>
                           <element name="numBeds" type="xsd:int"/>
                           <element name="isSmoking" type="xsd:boolean"/>
                       </element>
                   </element>
               </part>
           </message>
       </input>
       <output message="tns1:ReserveRoomResponse">
           <message name="ReserveRoomResponse">
              <part name="body" element="xsd1:ReserveRoomResponse">
                   <element name="ReserveRoomResponse"/>
               </part>
           </message>
       </output>
       <fault message="tns1:RoomNotAvailableException">
           <message name="RoomNotAvailableException">
               <part name="body" element="xsd1:RoomNotAvailableException">
                   <element name="RoomNotAvailableException"/>
               </part>
           </message>
       </fault>
   </operation>
</source>
```

Figure 5.11 – *Construction of the "ReserveRoom" WSCell (6/6).*

This is the final representation of the "ReserveRoom" WSCell after the contextualizeTypes and contextualizeComplexTypes functions have completed. Tags that are added by this stage are highlighted.

5.3 Summary

In this chapter, we described how to transform a WSDL operation description into a contextualized operation we call a WSCell using a simple hotel reservation service as an example to illustrate. First, we briefly went over the TXL grammar we wrote (available in full in Appendix A), then we described the TXL transformation rules involved and what they do to transform the example operation (a listing of these rules is available in Appendix B). In the next chapter, we will show that by using these as potential clones, we can detect more meaningful clones than we could without contextualizing.

6 – Validation

The previous chapter presented TXL [9, 10] transformation rules for restructuring WSDL [6] descriptions into a set of self-contained operation descriptions called WSCells by consolidating all the pieces of an operation description. We have used these rules to make a WSDL extractor for the NICAD clone detector [11, 22] to emulate the pre-transform phase proposed in Chapter 4. In this chapter, we present an empirical analysis to show that consolidating operations yields better results than the naïve approach (i.e. just extracting the `<operation>` elements), when using NICAD.

6.1 Procedure and Results

We tested our method on two sets of WSDL service descriptions. Set 1 is a system of over 200 web services containing more than 1,100 operations, many of them similar or duplicates. Set 2 is a collection of over 500 service descriptions containing over 7,500 operations from a wide variety of domains, obtained through a web services search engine by Seekda [28].

For each set, we compared the result of clone detection at various difference thresholds on the set of contextualized operations against the results on the set of original non-contextualized operations present in the original service descriptions. It should be noted that the purpose of this evaluation was to test our method of contextualizing operations against non-contextualized operations using clone detection. We have not tested the overall approach of using clone detection to find similar web service operations at this time (i.e. we cannot say if clone detection is better than any of the approaches mentioned in Chapter 2 at finding similar web services).

NICAD produces a set of clusters of operations that it determines to be clones (i.e. similar) for a given difference threshold or tolerance. The threshold specifies the portion of lines that are allowed to be different between two code fragments (in our case WSDL operation descriptions) in order for them to be considered clones. For example, a threshold of 0.3 means that operation descriptions are allowed to differ in 30% of their total number of lines (i.e. 3 lines in 10), while a threshold of 0 means that two operations must be exactly the same to be considered clones.

NICAD efficiently compares each potential clone (operation description) to the others looking for ones that differ by at most the number of lines allowed by the threshold described above. Based on this, it outputs XML files for each threshold with a list of `<class>` tags containing the similar operations belonging to a particular operation class.

We used our WSDL extractor for NICAD to produce and extract the WSCells from the two sets of web service descriptions. We also created another similar extractor that used the naïve approach of extracting the sparse `<operation>` elements in order to evaluate the benefits of our consolidation method over raw clone detection. **Table 6.1** and **Table 6.2** show the NICAD generated clone metrics for each set (non-contextualized and contextualized).

Difference Threshold	Set 1		Set 2	
	No Context	Context	No Context	Context
0.0	683	456	514	455
0.1	683	520	518	652
0.2	707	575	741	1116
0.3	713	618	2758	2548

Table 6.1 – *NICAD output – Number of clones.*

The number of code (operation) clones found by NICAD near-miss clone analysis of the contextualized operations of two different sets of web services.

Difference Threshold	Set 1		Set 2	
	No Context	Context	No Context	Context
0.0	169	166	386	316
0.1	169	129	389	423
0.2	172	138	543	729
0.3	171	132	1287	1504

Table 6.2 – *NICAD output – Number of clone classes.*

The number of clone classes found by NICAD near-miss clone analysis of the contextualized operations of two different sets of web services.

It may be noted that there is little or no difference in the numbers from **Table 6.1** and **Table 6.2** for thresholds 0.0 and 0.1 with no context. This is not an error, but rather due to the fact that `<operation>` elements with no context tend to be 3-5 lines long (most often an opening tag, an input tag, an output tag, and a closing tag), so the difference threshold has little to no effect. For example, the number of lines permitted to be different for operations of 4 lines is 0 with thresholds of 0.0 and 0.1, and 1 with thresholds of 0.2 and 0.3 (once rounded to the nearest whole line).

6.2 Observations

It makes little sense to calculate and compare precision and recall for each approach because they each use a different definition of clones. The approach with no context uses a traditional definition, while the approach with context uses our definition of contextual clones from Section 4.1.2. Comparing them in this way would be meaningless. Furthermore, it is unclear what makes a "correct" clone pair in the case of WSDL. For example, if two operations use the same data types as input but generate completely different output, should they be considered contextual clones? So, instead of calculating precision and recall, we make some general observations about the results.

The first thing we notice is that the number operations that are exactly the same (threshold 0.0 in **Table 6.1**) decreases when considering our contextualized operations over non-contextualized operations. The same can be said for similar operations that are within the threshold of 0.1 and in the case of Set 1, 0.2 and even 0.3. It may appear that contextualizing the operations gives worse results. However, upon deeper inspection, we see that what is actually happening is that false positives (i.e. operations found to be similar that are not) are being filtered out, yielding higher precision in the results.

One might think that if two operations were the same non-contextualized, contextualizing them (assuming that the contextualizing is done the same way for each) would just add new information that was still the same for both. But this is not the case. Consider two non-contextualized operations with the same input and output tags. They may appear to be the same, however when we look at the type definitions of the parameters, we see that they use different types, which may even have completely different names. Consider the example `GetStock` operation from Chapter 4. **Figure 6.1** shows the operation before and after contextualization for two different fragments. At the top, we see the description of the non-contextualized operations, which appear identical. However, when we contextualize them (shown below the non-contextualized operations in the figure), we see that they have very different meanings. Specifically, we see that the word “stock" has been used to mean inventory in one context and a financial stock quote in the other.

Non-Contextualized:
```
<operation name="GetStock" >
  <input message="GetStockRequest" />
  <output message="GetStockResponse" />
</operation>
```

Contextualized:
```
<operation name="GetStock" >
  <input message="GetStockRequest">
    <message name="GetStockRequest">
       <part name="parameters" element="GetStockRequest">
         <element name="GetStockRequest">
           <element name="InventoryNumber" type="xsd:int" />
         </element>
       </part>
    </message>
  </input>
  <output message="GetStockResponse">
    <message name="GetStockResponse">
       <part name="parameters" element="GetStockResponse">
         <element name=“Stock”>
           <element name=“Supplier” type=“xsd:string”/>
           <element name=“Warehouse” type=“xsd:string”/>
           <element name=“OnHand” type=“xsd:string”/>
           <element name=“OnOrder” type=“xsd:string”/>
           <element name=“Demand” type=“xsd:string”/>
        </element>
       </part>
    </message>
  </output>
</operation>
```

Non-Contextualized:
```
<operation name="GetStock" >
  <input message="GetStockRequest" />
  <output message="GetStockResponse" />
</operation>
```

Contextualized:
```
<operation name="GetStock" >
  <input message="GetStockRequest">
    <message name="GetStockRequest">
       <part name="parameters" element="GetStockRequest">
         <element name="GetStockRequest">
           <element name="symbol" type="xsd:string" />
         </element>
       </part>
    </message>
  </input>
  <output message="GetStockResponse">
    <message name="GetStockResponse">
       <part name="parameters" element="GetStockResponse">
         <element name=“Stock”>
           <element name=“date” type=“xsd:string”/>
           <element name=“open” type=“xsd:float”/>
           <element name=“high” type=“xsd:float”/>
           <element name=“low” type=“xsd:float”/>
           <element name=“close” type=“xsd:float”/>
           <element name=“volume” type=“xsd:float”/>
         </element>
       </part>
    </message>
  </output>
</operation>
```

Figure 6.1 – *Contextualization of two “GetStock” operations (first seen in Figure 4.1).*

These two “GetStock” operations appear to be exact clones when non-contextualized, but are actually from different domains, which we can see when we contextualize them.

The second thing we noticed is that by contextualizing the operations, operation classes (i.e. groups of similar operations) can be split into multiple classes. For instance, a class that contained 10 non-contextualized operations may be split into 2 classes with 5 contextualized operations in each. This tends to be the case in situations like the one described above (**Figure 6.1**) when there are operations that look similar non-contextualized, but when contextualized can be seen to have different contexts.

Finally, and most importantly, we found that contextualizing operations allowed us to find similar operations with clone detection that we would never have been able to find without. For example, consider the operation class shown in **Figure 6.2**. It contains operations, all related to charting, with completely different names that could not be recognized as related by clone detection if they were not contextualized. Some, such as `DrawYieldCurveCustom`, don't even have “chart" in their name. The reason these are recognized as being similar by NICAD is that the contextualization expands and inserts all type definitions into the operation description, and all of these operations actually contain very similar elements.

This validates our initial belief that contextualizing operation descriptions can allow us to more effectively find similar operations. It contextualizes an otherwise sparse operation description, which allows standard clone detection tools like NICAD to match based on context.

```
<operation name="GetRealChartCustom">
  <input message="GetRealChartCustomSoapIn"/>
  <output message="GetRealChartCustomSoapOut"/>
</operation>

<operation name="GetLastSaleChartCustom">
  <input message="GetLastSaleChartCustomSoapIn"/>
  <output message="GetLastSaleChartCustomSoapOut"/>
</operation>

<operation name="DrawHistoricalChartCustom">
  <input message="DrawHistoricalChartCustomSoapIn"/>
  <output message="DrawHistoricalChartCustomSoapOut"/>
</operation>

<operation name="DrawIntraDayChartCustom">
  <input message="DrawIntraDayChartCustomSoapIn"/>
  <output message="DrawIntraDayChartCustomSoapOut"/>
</operation>

<operation name="GetDelayedChartCustom">
  <input message="GetDelayedChartCustomSoapIn"/>
  <output message="GetDelayedChartCustomSoapOut"/>
</operation>
```

```
<operation name="GetTopicChartCustom">
  <input message="GetTopicChartCustomSoapIn" />
  <output message="GetTopicChartCustomSoapOut" />
</operation>

<operation name="GetTopicBinaryChartCustom">
  <input message="GetTopicBinaryChartCustomSoapIn"/>
  <output message="GetTopicBinaryChartCustomSoapOut"/>
</operation>

<operation name="DrawRateChartCustom">
  <input message="DrawRateChartCustomSoapIn"/>
  <output message="DrawRateChartCustomSoapOut"/>
</operation>

<operation name="DrawRateChartCustom">
  <input message="DrawRateChartCustomSoapIn"/>
  <output message="DrawRateChartCustomSoapOut"/>
</operation>

<operation name="DrawYieldCurveCustom">
  <input message="DrawYieldCurveCustomSoapIn"/>
  <output message="DrawYieldCurveCustomSoapOut" />
</operation>
```

Figure 6.2 – *A cluster of contextual clones.*

Here we see a clone cluster found by NICAD containing operations related to charting. This cluster was found using contextualized <operation> elements (i.e. WSCells); however, their large size prevents them from being shown in this figure.

6.3 Summary

Contextualizing WSDL operations before clone detection has been shown to give better results than not, thus providing a good example of where a pre-transform phase in NICAD would be beneficial. In the next chapter, we will summarize what was presented in this thesis and discuss possible future work.

7 – Conclusion

7.1 Summary

WSDL [6] descriptions of web services pose problems for finding similarities using clone detectors like NICAD [11, 22]. Operation descriptions are broken up into pieces and de-localized making it difficult to extract appropriate units to compare. If just one type of piece is extracted, then the others are unavailable later in the clone analysis. This led us to propose a NICAD Pre-Transformer to make transformations of the original code base possible before the extraction of potential clones. This allows operation descriptions to be reorganized into a set of consolidated units that can then be extracted and compared. In the absence of such a phase, we developed a special WSDL extractor for NICAD that performs this consolidation while extracting operations.

We refer to this process of consolidating pieces of an operation description as contextualization because it adds context to `<operation>` elements, which don't hold much meaning on their own. It has led to a new idea in clone detection we call *contextual clones* – clones found using code fragments that have been modified to add information referenced, either explicitly or implicitly, from elsewhere in the code or environment.

We call contextualized WSDL operations *Web Service Cells*, or *WSCells*, because they are like cells that together form a web service. Our NICAD extractor creates WSCells by finding and inserting a referenced element into the element that references it. So for an `<input>` inside the operation, the corresponding `<message>` is found and inserted into it; for each `<part>` inside the `<message>`, the corresponding `<element>` is found and inserted; and so on, until there are no more elements left.

Using this extractor we have demonstrated that the contextualization makes WSDL descriptions more amenable to analysis using code similarity techniques. The contextualization not only refines the results generated by clone detection to avoid identifying services that while similar on the surface actually have very different meanings, but also produces results identifying similar services that could not be found without contextualization. While we have used NICAD exclusively throughout this thesis, it is important to note that the extractor can be used with TXL [9, 10] independently of NICAD to produce WSCells that could be used with other clone detectors or even concept location techniques.

7.2 Limitations

While we have shown that WSCells produce more meaningful results with clone detection, there remain a couple of limitations. First, clone detectors like NICAD do not look for semantic relationships, so they can miss important relationships between smaller code fragments. The kinds of clones we do see are those that share user-defined types, which indicates that they may be used together. We can find clones from similar domains this way because some, such as the financial domain, use a standard XML schema for their services.

The second limitation, that is present with any approach to finding similar web service operations using WSDL, is that even if two operation descriptions are identical, they may not do the same thing. Since WSDL defines an interface, the only information available is the input and output of each operation. Besides the name and optional `<documentation>` tags, there is no information to guarantee what the operation will do when invoked.

7.3 Future Work

Now that we have developed a way to perform clone detection on WSDL descriptions, we can begin to attack the real goal of our work, the automated tagging of alternative services and operations for a given purpose or to match a given protocol or domain ontology. Using the results of our clone detection-based similarity analysis, we can further leverage our source transformation methods to tag and classify similar services and operations.

There are a number of directions we can take with this research in the future. First, there are numerous normalizations with which we can experiment in NICAD to possibly achieve better results. For example, we could remove closing tags since they provide no meaningful information for clone detection and they can even be harmful to the results. Since most WSCells share at least 5 closing tags at the end due to nesting, this can lead to false positives, particularly for smaller WSCells. Other possible normalizations include, putting attributes like `name` and `type` on their own line, removing repetitive tags (e.g. `<message>` and `<part>`), and even going as far as to remove all tags (i.e. using only attributes) while maintaining order.

We have also used WSCells with *Latent Dirichlet allocation* (LDA) [3] to detect conceptual relationships [13] using topic models. As with NICAD and clone detection, we found that WSCells produced more meaningful relationships than non-contextualized `<operation>` elements. However, LDA found different kinds of relationships than NICAD; relationships that have more to do with meaning than with shared data types. For example, LDA was able to find operations that related to phone number and addresses, even if they didn't share exactly the same types. Future work may look at how we can use these two techniques – clone detection and concept location – together to detect all kinds of related services to help developers create new services.

This work introduces a new class of code clones - those that are hidden in the original source text, but may be uncovered by contextualization similar to what we have done for WSDL. This new idea of contextual clones holds promise for other languages as well, specifically other XML-based domain specific languages. Using our consolidation technique, we may be able to find clones in textual representations of software models.

It is unclear whether contextualization would prove useful with imperative languages, like C or Java, since these languages are already well-suited for clone detection (or rather, clone detection has been designed for them). There may, however, be special cases that can utilize it. There may also be applications of the proposed NICAD Pre-Transformer (described in Chapter 4) to be explored. For example, a code base may be transformed to some normalized state in an attempt to find clones between mutated code of the same malicious software to aid in the detection of malware.

Our work brought to light an inherent difficulty to understand WSDL descriptions and need by users to understand repositories of them. WSCells provide a more comprehensible and unified description of operations that users seemed to really respond to. Future work may explore ways of detecting relationships and visualizing them.

References

[1] I.D. Baxter, A. Yahin, L. Moura, M. Sant'Anna and L. Bier, "Clone Detection Using Abstract Syntax Trees," in *Proceedings of the 14th International Conference on Software Maintenance*, *ICSM 1998*, Bethesda, MD, USA, pp. 368-377, 1998.

[2] P.V. Biron and A. Malhotra, "XML Schema Part 2: Datatypes Second Edition," *The World Wide Web Consortium*, 2004. [Online]. Available: `http://www.w3.org/TR/xmlschema-2/`. [Accessed: June 2011].

[3] D. Blei, A. Ng, and M. Jordan, "Latent dirichlet allocation," *Journal of Machine Learning Research*, vol. 3, pp. 993-1022, 2003.

[4] E. Cerami, *Web Services Essentials*. Sebastopol, CA: O'Reilly Media, 2002.

[5] R. Chinnici, J. Moreau, A. Ryman and S. Weerawarana, "Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language," *The World Wide Web Consortium*, 2007. [Online]. Available: `http://www.w3.org/TR/wsdl20/`. [Accessed: June 2011].

[6] E. Christensen, F. Curbera, G. Meredith and S. Weerawarana, "Web Services Description Language (WSDL) 1.1," *The World Wide Web Consortium*, 2001. [Online]. Available: `http://www.w3.org/TR/wsdl/`. [Accessed: June 2011].

[7] J. Clark, "XSL Transformations (XSLT) Version 1.0," *The World Wide Web Consortium*, 2001. [Online]. Available: `http://www.w3.org/TR/xslt/`. [Accessed: June 2011].

[8] J. Clark and S. DeRose, "XML Path Language (XPath) Version 1.0," *The World Wide Web Consortium*, 1999. [Online]. Available: `http://www.w3.org/TR/xpath/`. [Accessed: June 2011].

[9] J.R. Cordy, "The TXL Source Transformation Language," *Science of Computer Programming*, vol. 61, no. 3, pp. 190-210, August 2006.

[10] J.R. Cordy, I.H. Carmichael and R. Halliday, *The TXL Programming Language*. Queen's University, Kingston, November 2007.

[11] J.R. Cordy and C.K. Roy, "The NiCad Clone Detector," in *Proceedings of the Tool Demo Track of the 19th International Conference on Program Comprehension, ICPC 2011*, Kingston, ON, Canada, pp. 219-220, June 2011.

[12] X. Dong, A. Halevy, J. Madhaven, E. Nemes and J. Zhang, “Similarity Search for Web Services,” in *Proceedings of the 30th VLDB Conference, VLDB 2004*, Toronto, ON, Canada, pp. 372-383 2004.

[13] S. Grant, D. Martin, J.R. Cordy and D. Skillicorn, “Contextualized Semantic Analysis of Web Services,” in *Proceedings of the 13th International Symposium on Web Systems Evolution, WSE 2011*, Williamsburg, VA, USA, 7 pp., September 2011 (to appear).

[14] M. Gudgin, M. Hadley, N. Mendelsohn, J. Moreau, H.F. Nielsen, A. Karmarkar and Y. Lafon, “SOAP Version 1.2 Part 1: Messaging Framework (Second Edition),” *The World Wide Web Consortium*, 2007. [Online]. Available: `http://www.w3.org/TR/soap12-part1/`. [Accessed: June 2011].

[15] M. Hadley, “Web Application Description Language,” *The World Wide Web Consortium*, 2004. [Online]. Available: `http://www.w3.org/Submission/wadl/`. [Accessed: June 2011].

[16] H. Hass and A. Brown, “Web Services Glossary,” *The World Wide Web Consortium*, 2004. [Online]. Available: `http://www.w3.org/TR/ws-gloss/`. [Accessed: June 2011].

[17] T. Kamiya, S. Kusumoto and K. Inoue, “CCFinder: A Multilinguistic Token-Based Code Clone Detection System for Large Scale Source Code,” *IEEE Transactions on Software Engineering*, vol. 28, no. 7, pp. 654-670, 2002.

[18] M. Kay, “XSL Transformations (XSLT) Version 2.0,” *The World Wide Web Consortium*, 2007. [Online]. Available: `http://www.w3.org/TR/xslt20/`. [Accessed: June 2011].

[19] D. Martin and J.R. Cordy, “Towards Web Services Tagging by Similarity Detection,” in *The Smart Internet: Current Research and Future Applications, Lecture Notes in Computer Science 6400*, M. Chignell et al., Eds. New York: Springer Verlag, pp. 222-240, October 2010.

[20] D. Martin and J.R. Cordy, “Analyzing Web Service Similarity Using Contextual Clones,” in *Proceedings of the 5th International Workshop on Software Clones, IWSC 2011*, Waikiki, Hawaii, pp. 41-46, May 2011..

[21] L. Richardson and S. Ruby, *RESTful Web Services*. Sebastopol, CA: O'Reilly Media, 2007.

[22] C.K. Roy, “Detection and Analysis of Near-Miss Software Clones,” Ph.D. dissertation, Queen's University, Kingston, ON, Canada, August 2009.

[23] C.K. Roy and J.R. Cordy, “A mutation / injection-based automatic framework for evaluating code clone detection tools,” in *Proceedings of the 4th International Workshop on Mutation Analysis, Mutation 2009*, Denver, CO, USA, pp. 157-166, April 2009.

[24] C.K. Roy, J.R. Cordy and R. Koschke, “Comparison and Evaluation of Code Clone Detection Techniques and Tools: A Qualitative Approach,” *Science of Computer Programming*, vol. 74, pp. 470-495, May 2009.

[25] T. Syeda-Mahmood, G. Shah, R. Akkiraju, A. Ivan and R. Goodwin, “Searching Service Repositories by Combining Semantic and Ontological Matching,” in *Proceedings of the 3rd International Conference on Web Services*, *ICWS 2005*, Orlando, FL, USA, pp. 13-20, 2005.

[26] E. Stroulia and Y. Wang, “Structural and Semantic Matching for Assessing Web-Service Similarity,” *International Journal of Cooperative Information Systems*, vol. 14, no. 4, pp. 407-437, 2005.

[27] H.S. Thompson, D. Beech, M. Maloney and N. Mendelsohn, “XML Schema Part 1: Structures 2nd Edition,” *The World Wide Web Consortium*, 2004. [Online]. Available: `http://www.w3.org/TR/xmlschema-1/`. [Accessed: June 2011].

[28] Web Services Search Engine, `http://webservices.seekda.com/`.

Appendix A – WSDL Grammar

```
#pragma -case -w 32000

tokens
  id  |   "\a[-\i]*"
end tokens

comments
  <!--  -->
  <xs:annotation> </xs:annotation>
  <documentation  </documentation>
  <wsdl:documentation  </wsdl:documentation>
  <xs:documentation  </xs:documentation>
  <xsd:documentation  </xsd:documentation>
end comments

define program
  [opt xml_def]
  [definitions]
end define

define xml_def
  [SPOFF] '<?xml [attribute_list] '?> [SPON]                              [NL]
end define

define prefix
    'soap:
  | 'xs:
  | 'xmlns:
  | 'wsdl:
  | 'mime:
  | 'http:
  | [id] ':
end define

define attribute_list
  [repeat attribute]
end define

define attribute
    [name_attr]
  | [target_namespace_attr]
  | [type_attr]
  | [element_attr]
  | [message_attr]
  | [xmlns_attr]
  | [basic_attr]
end define

define name_attr
  [SP] [opt prefix] 'name = [string]
end define

define target_namespace_attr
  [SP] [opt prefix] 'targetNamespace = [string]
end define
```

```
define element_attr
  [SP] [opt prefix] 'element = [string]
end define

define type_attr
  [SP] [opt prefix] 'type = [string]
end define

define message_attr
  [SP] [opt prefix] 'message = [string]
end define

define xmlns_attr
  [SP] 'xmlns: [id] = [string]
end define

define basic_attr
  [SP] [opt prefix] [id] = [string]
end define

define string
    [stringlit]
  | [charlit]
end define

define definitions
  [SPOFF] '< [opt prefix] 'definitions [opt attribute_list] '> [SPON]       [NL][IN]
    [repeat top_level_elements]                                             [EX]
  [SPOFF] '</ [opt prefix] 'definitions> [SPON]                             [NL]
end define

define types
  [SPOFF] '< [opt prefix] 'types [opt attribute_list] '> [SPON]             [NL][IN]
    [repeat types_scope]                                                    [EX]
  [SPOFF] '</ [opt prefix] 'types> [SPON]                                   [NL]
end define

define schema
    [SPOFF] '< [opt prefix] 'schema [opt attribute_list] '> [SPON]          [NL][IN]
      [repeat schema_scope]                                                 [EX]
    [SPOFF] '</ [opt prefix] 'schema> [SPON]                                [NL]
  | [SPOFF] '< [opt prefix] 'schema [opt attribute_list] '/> [SPON]         [NL]
end define

define import_or_include
    [import_element]
  | [include_element]
end define

define import_element
    [SPOFF] '< [opt prefix] 'import [opt attribute_list] '/> [SPON]         [NL]
  | [SPOFF] '< [opt prefix] 'import [opt attribute_list] '>
    '</'import> [SPON]                                                      [NL]
end define

define include_element
    [SPOFF] '< [opt prefix] 'include [opt attribute_list] '/> [SPON]        [NL]
  | [SPOFF] '< [opt prefix] 'include [opt attribute_list] '>
    '</'include> [SPON]                                                     [NL]
end define

define schema_element
    [SPOFF] '< [opt prefix] 'element [opt attribute_list] '/>  [SPON]       [NL]
  | [SPOFF] '< [opt prefix] 'element [opt attribute_list] '> [SPON]         [NL][IN]
```

```
      [repeat schema_scope]                                                   [EX]
    [SPOFF] '</ [opt prefix] 'element> [SPON]                                 [NL]
end define

define any_element
    [SPOFF] '< [opt prefix] 'any [opt attribute_list] '/>  [SPON]             [NL]
end define

define element_attribute
  [SPOFF] '< [opt prefix] 'attribute [opt attribute_list] '/> [SPON]          [NL]
end define

define any_attribute
  [SPOFF] '< [opt prefix] 'anyAttribute [opt attribute_list] '/> [SPON]       [NL]
end define

define complex_type
  [SPOFF] '< [opt prefix] 'complexType [opt attribute_list] '> [SPON]         [NL][IN]
    [repeat complexType_scope]                                                [EX]
  [SPOFF] '</ [opt prefix] 'complexType> [SPON]                               [NL]
end define

define complex_content
  [SPOFF] '< [opt prefix] 'complexContent [opt attribute_list] '> [SPON]      [NL][IN]
    [repeat complexType_scope]                                                [EX]
  [SPOFF] '</ [opt prefix] 'complexContent> [SPON]                            [NL]
end define

define extension
  [SPOFF] '< [opt prefix] 'extension [opt attribute_list] '> [SPON]           [NL][IN]
    [repeat complexType_scope]
            [EX]
  [SPOFF] '</ [opt prefix] 'extension> [SPON]                                 [NL]
end define

define simple_type
  [SPOFF] '< [opt prefix] 'simpleType [opt attribute_list] '> [SPON]          [NL][IN]
    [repeat simpleType_scope]                                                 [EX]
  [SPOFF] '</ [opt prefix] 'simpleType> [SPON]                                [NL]
end define

define simple_content
  [SPOFF] '< [opt prefix] 'simpleContent [opt attribute_list] '> [SPON]       [NL][IN]
    [repeat simpleType_scope]                                                 [EX]
  [SPOFF] '</ [opt prefix] 'simpleContent> [SPON]                             [NL]
end define

define restriction
  [SPOFF] '< [opt prefix] 'restriction [opt attribute_list] '> [SPON]         [NL][IN]
    [repeat restriction_scope]                                                [EX]
  [SPOFF] '</ [opt prefix] 'restriction> [SPON]                               [NL]
end define

define enumeration
  [SPOFF] '< [opt prefix] 'enumeration [attribute_list] '/>  [SPON]           [NL]
end define

define minInclusive
  [SPOFF] '< [opt prefix] 'minInclusive [attribute_list] '/>  [SPON]          [NL]
end define

define maxInclusive
  [SPOFF] '< [opt prefix] 'maxInclusive [attribute_list] '/>  [SPON]          [NL]
end define
```

```
define pattern
    [SPOFF] '< [opt prefix] 'pattern [attribute_list] '/>  [SPON]           [NL]
end define

define whiteSpace
    [SPOFF] '< [opt prefix] 'whiteSpace [attribute_list] '/>  [SPON]        [NL]
end define

define length
    [SPOFF] '< [opt prefix] 'length [attribute_list] '/>  [SPON]            [NL]
  | [SPOFF] '< [opt prefix] 'minLength [attribute_list] '/>  [SPON]         [NL]
  | [SPOFF] '< [opt prefix] 'maxLength [attribute_list] '/>  [SPON]         [NL]
end define

define sequence_indicator
    [SPOFF] '< [opt prefix] 'sequence> [SPON]                               [NL][IN]
      [repeat schema_scope]                                                 [EX]
    [SPOFF] '</ [opt prefix] 'sequence> [SPON]                              [NL]
  | [SPOFF] '</ [opt prefix] 'sequence> [SPON]                              [NL]
end define

define all_indicator
    [SPOFF] '< [opt prefix] 'all> [SPON]                                    [NL][IN]
      [repeat schema_scope]                                                 [EX]
    [SPOFF] '</ [opt prefix] 'all> [SPON]                                   [NL]
  | [SPOFF] '</ [opt prefix] 'all> [SPON]                                   [NL]
end define

define choice_indicator
    [SPOFF] '< [opt prefix] 'choice> [SPON]                                 [NL][IN]
      [repeat schema_scope]                                                 [EX]
    [SPOFF] '</ [opt prefix] 'choice> [SPON]                                [NL]
  | [SPOFF] '</ [opt prefix] 'choice> [SPON]                                [NL]
end define

define group_indicator
  [SPOFF] '< [opt prefix] 'group> [SPON]                                    [NL][IN]
    [repeat complexType_scope]                                              [EX]
  [SPOFF] '</ [opt prefix] 'group> [SPON]                                   [NL]
end define

define message
  [SPOFF] '< [opt prefix] 'message [opt attribute_list] [SPON] '>           [NL][IN]
    [repeat message_scope]                                                  [EX]
  [SPOFF] '</ [opt prefix] 'message> [SPON]                                 [NL]
end define

define part
    [SPOFF] '< [opt prefix] 'part [opt attribute_list] '/>  [SPON]          [NL]
  | [SPOFF] '< [opt prefix] 'part [opt attribute_list] '> [SPON]
  [NL][IN]
      [repeat element]                                                      [EX]
    [SPOFF] '</ [opt prefix] 'part> [SPON]                                  [NL]
end define

define binding
    [SPOFF] '< [opt prefix] 'binding [opt attribute_list] [SPON] '>         [NL][IN]
      [repeat binding_scope]                                                [EX]
    [SPOFF] '</ [opt prefix] 'binding> [SPON]                               [NL]
  | [SPOFF] '< [opt prefix] 'binding [opt attribute_list] '/> [SPON]        [NL]
end define

define service
```

```
  [SPOFF] '< [opt prefix] 'service [opt attribute_list] [SPON] '>           [NL][IN]
    [repeat service_scope]                                                  [EX]
  [SPOFF] '</ [opt prefix] 'service> [SPON]                                 [NL]
end define

define port
    [SPOFF] '< [opt prefix] 'port [opt attribute_list] [SPON] '>            [NL][IN]
      [repeat element]                                                      [EX]
    [SPOFF] '</ [opt prefix] 'port> [SPON]                                  [NL]
  | [SPOFF] '< [opt prefix] 'port [opt attribute_list] '/> [SPON]           [NL]
end define

define port_type
  [SPOFF] '< [opt prefix] 'portType [opt attribute_list] [SPON] '>          [NL][IN]
    [repeat operation]                                                      [EX]
  [SPOFF] '</ [opt prefix] 'portType> [SPON]                                [NL]
end define

define operation
    [SPOFF] '< [opt prefix] 'operation [opt attribute_list] [SPON] '>       [NL][IN]
      [repeat operation_scope]                                              [EX]
    [SPOFF] '</ [opt prefix] 'operation> [SPON]                             [NL]
  | [SPOFF] '< [opt prefix] 'operation [opt attribute_list] '/> [SPON]      [NL]
end define

define input
    [SPOFF] '< [opt prefix] 'input [opt attribute_list] '> [SPON]           [NL][IN]
      [repeat element]                                                      [EX]
    [SPOFF] '</ [opt prefix] 'input> [SPON]                                 [NL]
  | [SPOFF] '< [opt prefix] 'input [opt attribute_list] '/> [SPON]          [NL]
end define

define output
    [SPOFF] '< [opt prefix] 'output [opt attribute_list] '> [SPON]          [NL][IN]
      [repeat element]                                                      [EX]
    [SPOFF] '</ [opt prefix] 'output> [SPON]                                [NL]
  | [SPOFF] '< [opt prefix] 'output [opt attribute_list] '/> [SPON]         [NL]
end define

define infault
    [SPOFF] '< [opt prefix] 'infault [opt attribute_list] '> [SPON]         [NL][IN]
      [repeat element]                                                      [EX]
    [SPOFF] '</ [opt prefix] 'infault> [SPON]                               [NL]
  | [SPOFF] '< [opt prefix] 'infault [opt attribute_list] '/> [SPON]        [NL]
end define

define outfault
    [SPOFF] '< [opt prefix] 'outfault [opt attribute_list] '> [SPON]        [NL][IN]
      [repeat element]                                                      [EX]
    [SPOFF] '</ [opt prefix] 'outfault> [SPON]                              [NL]
  | [SPOFF] '< [opt prefix] 'outfault [opt attribute_list] '/> [SPON]       [NL]
end define

define fault
    [SPOFF] '< [opt prefix] 'fault [opt attribute_list] '> [SPON]           [NL][IN]
      [repeat element]                                                      [EX]
    [SPOFF] '</ [opt prefix] 'fault> [SPON]                                 [NL]
  | [SPOFF] '< [opt prefix] 'fault [opt attribute_list] '/> [SPON]          [NL]
end define

define interface
    [SPOFF] '< [opt prefix] 'interface [opt attribute_list] '> [SPON]       [NL][IN]
      [repeat interface_scope]                                              [EX]
    [SPOFF] '</ [opt prefix] 'interface> [SPON]                             [NL]
  | [SPOFF] '< [opt prefix] 'interface [opt attribute_list] '/> [SPON]      [NL]
end define

define top_level_elements
    [types]
  | [service]
  | [message]
  | [port_type]
  | [binding]
  | [interface]
  | [element]
end define

define  types_scope
    [schema]
end define

define schema_scope
    [schema_element]
  | [any_element]
  | [import_or_include]
  | [complex_type]
  | [simple_type]
  | [element]
end define

define complexType_scope
    [complex_content]
  | [simple_content]
  | [element_attribute]
  | [any_attribute]
  | [extension]
  | [sequence_indicator]
  | [all_indicator]
  | [choice_indicator]
  | [group_indicator]
  | [schema_element]
  | [any_element]
end define

define simpleType_scope
    [simple_content]
  | [restriction]
  | [extension]
  | [schema_scope]
end define

define restriction_scope
    [enumeration]
  | [minInclusive]
  | [maxInclusive]
  | [pattern]
  | [whiteSpace]
  | [length]
end define

define message_scope
    [part]
  | [element]
end define

define operation_scope
    [operation]
  | [input]
  | [output]
  | [infault]
  | [outfault]
  | [fault]
  | [element]
end define

define binding_scope
    [operation]
  | [element]
end define

define service_scope
    [port]
  | [element]
end define

define interface_scope
    [operation]
  | [fault]
  | [element]
end define

define element
    [definitions]
  | [types]
  | [schema]
  | [import_element]
  | [include_element]
  | [complex_type]
  | [complex_content]
  | [extension]
  | [simple_type]
  | [simple_content]
  | [element_attribute]
  | [any_attribute]
  | [restriction]
  | [enumeration]
  | [minInclusive]
  | [maxInclusive]
  | [pattern]
  | [whiteSpace]
  | [length]
  | [schema_element]
  | [any_element]
  | [sequence_indicator]
  | [choice_indicator]
  | [all_indicator]
  | [group_indicator]
  | [service]
  | [interface]
  | [message]
  | [port_type]
  | [binding]
  | [operation]
  | [input]
  | [output]
  | [infault]
  | [outfault]
  | [fault]
  | [port]
  | [part]
  | [singleton_tag]
  | [tag]
```

```
end define

define singleton_tag
  [SPOFF] '< [opt prefix] [id] [attribute_list] '/> [SPON]                  [NL]
end define

define tag
  [begin_tag]                                                                [NL][IN]
    [repeat element]                                                         [EX]
  [end_tag]                                                                  [NL]
end define

define begin_tag
   [SPOFF] '< [opt prefix] [push id] [attribute_list] '> [SPON]

end define

define end_tag
   [SPOFF] '</ [opt prefix] [pop id] '> [SPON]
end define
```

Appendix B – WSDL Extractor

```
include "WSDL.grm"

% Define <source> element to wrap around newly constructed operations.
define source
  [SPOFF] '<'source [opt attribute_list] '> [SPON]                        [NL][IN]
    [repeat element]                                                      [EX]
  [SPOFF] '</'source> [SPON]                                              [NL]
end define

% Redefine <operation> elements so filename and begin/end line numbers are known.
redefine operation
    [attr srcfilename] [attr srclinenumber]
    [SPOFF] '< [opt prefix] 'operation [opt attribute_list] [SPON] '>     [NL][IN]
      [repeat operation_scope]                                            [EX]
    [attr srclinenumber]
    [SPOFF] '</ [opt prefix] 'operation> [SPON]                           [NL]
  | [attr srcfilename] [attr srclinenumber]
    [SPOFF] '< [opt prefix] 'operation [opt attribute_list] '/> [SPON]    [NL]
end redefine

% Redefine a WSDL program to allow a list of <source> elements.
redefine program
    ...
  | [repeat source]
end redefine

% Allow all types of elements to be inserted into these during consolidation.
redefine complex_type
    [SPOFF] '< [opt prefix] 'complexType [opt attribute_list] '> [SPON]   [NL][IN]
      [repeat element]                                                    [EX]
    [SPOFF] '</ [opt prefix] 'complexType> [SPON]                         [NL]
  |   ...
end redefine

redefine simple_type
  [SPOFF] '< [opt prefix] 'simpleType [opt attribute_list] '> [SPON]      [NL][IN]
    [repeat element]                                                      [EX]
  [SPOFF] '</ [opt prefix] 'simpleType> [SPON]                            [NL]
end redefine

redefine message
  [SPOFF] '< [opt prefix] 'message [opt attribute_list] '> [SPON]         [NL][IN]
    [repeat element]                                                      [EX]
  [SPOFF] '</ [opt prefix] 'message> [SPON]                               [NL]
end redefine

redefine element
    ...
  | [SPOFF] '< [opt prefix] 'element [opt attribute_list] '> [SPON]       [NL][IN]
      [repeat element]                                                    [EX]
    [SPOFF] '</ [opt prefix] 'element> [SPON]                             [NL]
end redefine

redefine schema
    ...
  | [xml_def]
```

```
end redefine

% The main function calls the contextualizeOperations function with each <operation>
% element. It returns a list of contextualized operations as the new "program."
function main
  replace [program]
    P [program]

  % Extract all <operation> elements from the <portTypes> element.
  construct AllPortTypes [repeat port_type]
    _ [^ P]
  construct AllOperations [repeat operation]
    _ [^ AllPortTypes]

  % Extract all schema, messages and namespace definitions and export them for use later
  % This saves having to extract the same elements over and over again.
  construct AllSchema [repeat schema]
    _ [^ P]
  construct AllMessages [repeat message]
    _ [^ P]
  deconstruct P
    _ [opt xml_def]
    '< _ [opt prefix] 'definitions DefAttrList [opt attribute_list] '>
      _ [repeat top_level_elements]
    '</ _ [opt prefix] 'definitions>
  construct DefNamespaces [repeat xmlns_attr]
    _ [^ DefAttrList]
  export Schema [repeat schema]
    AllSchema
  export Messages [repeat message]
    AllMessages
  export AllNamespaces [repeat xmlns_attr]
    DefNamespaces

  % Construct contextualized operations using the previously extracted <operation>
  % elements
  construct Operations [repeat source]
    _ [contextualizeOperations each AllOperations]
  by
    Operations
end function

% The contextualizeOperations function takes an <operation> element, contextualizes it,
% and appends it to a list.
function contextualizeOperations Operation [operation]
  replace [repeat source]
    PreviousOperations [repeat source]

  % Deconstruct the operation to get the attribute list filename and begin/end line number
  deconstruct Operation
    File [srcfilename] BeginLine [srclinenumber]
     '< _ [opt prefix] 'operation AttrList [opt attribute_list] '>
      Contents [repeat operation_scope]
    EndLine [srclinenumber] '</ _ [opt prefix] 'operation>

  % Add double quotes because these will be used as attributes in the <source> tag
  construct Filename [stringlit]
    _ [quote File]
  construct BeginLineNumber [stringlit]
    _ [quote BeginLine]
  construct EndLineNumber [stringlit]
    _ [quote EndLine]

  % Extract the inputs, outputs and faults
  construct Inputs [repeat input]
    _ [^ Contents]
  construct Outputs [repeat output]
    _ [^ Contents]
  construct Faults [repeat fault]
    _ [^ Contents]

  % Contextualize the inputs, outputs and faults
  construct NewInputs [repeat operation_scope]
    _ [contextualizeInputs each Inputs]
  construct NewOutputs [repeat operation_scope]
    _ [contextualizeOutputs each Outputs]
  construct NewFaults [repeat operation_scope]
    _ [contextualizeFaults each Faults]

  % Append the contextualized  inputs, outputs and faults to use as the contents
  % of the new contextualized operation
  construct NewOperationContents [repeat operation_scope]
    NewInputs [. NewOutputs] [. NewFaults]

  % Create the contextualized operation using the newly contextualized inputs,
  % outputs and faults
  construct NewOperation [operation]
    '<'operation AttrList '>
      NewOperationContents
    '</'operation>

  % Surround the contextualized operation with <source> tags for NICAD.
  construct SourceTag [source]
    '<'source 'file= Filename 'startline= BeginLineNumber 'endline= EndLineNumber '>
      NewOperation
    '</'source>
  by
    PreviousOperations [. SourceTag]
end function

% The contextualizeInputs function takes an <input> element, contextualizes it,
% and appends it to a list.
function contextualizeInputs Input [input]
  replace [repeat operation_scope]
    PreviousInputs [repeat operation_scope]

  % Get the messages (already extracted from the main function).
  import Messages [repeat message]

  %Get the value of the input's "message" attribute to find the matching one.
  construct AttrList [repeat attribute]
    _ [^ Input]
  construct MessageAttr [repeat message_attr]
    _ [^ AttrList]
  deconstruct MessageAttr
    _ [opt prefix] 'message = MessageName [stringlit]
    _ [repeat message_attr]
  construct MessageNameNoPrefix [stringlit]
    MessageName [deletePrefix]

  % Find the message with the name matching the message attribute and contextualize it.
  construct NewInputContents [repeat element]
    _ [contextualizeMessages MessageNameNoPrefix each Messages]

  % Create the contextualized input out of the contextualized message.
  construct NewInput [operation_scope]
    '<'input AttrList '>
      NewInputContents
    '</'input>
  by
    PreviousInputs [. NewInput]
end function

% The contextualizeOutputs function takes an <output> element, contextualizes it,
% and appends it to a list.
function contextualizeOutputs Output [output]
  replace [repeat operation_scope]
    PreviousOutputs [repeat operation_scope]

  % Get the messages (already extracted from the main function).
  import Messages [repeat message]

  %Get the value of the output's "message" attribute to find the matching one.
  construct AttrList [repeat attribute]
    _ [^ Output]
  construct MessageAttr [repeat message_attr]
    _ [^ AttrList]
  deconstruct MessageAttr
    _ [opt prefix] 'message = MessageName [stringlit]
    _ [repeat message_attr]
  construct MessageNameNoPrefix [stringlit]
    MessageName [deletePrefix]

  % Find the message with the name matching the message attribute and contextualize it.
  construct NewOutputContents [repeat element]
    _ [contextualizeMessages MessageNameNoPrefix each Messages]

  % Create the contextualized output out of the contextualized message.
  construct NewOutput [operation_scope]
    '<'output AttrList '>
      NewOutputContents
    '</'output>
  by
    PreviousOutputs [. NewOutput]
end function

% The contextualizeFaults function takes a <fault> element, contextualizes it,
% and appends it to a list.
function contextualizeFaults Fault [fault]
  replace [repeat operation_scope]
    PreviousFaults [repeat operation_scope]

  % Get the messages (already extracted from the main function).
  import Messages [repeat message]

  %Get the value of the fault's "message" attribute to find the matching one.
  construct AttrList [repeat attribute]
    _ [^ Fault]
  construct MessageAttr [repeat message_attr]
    _ [^ AttrList]
  deconstruct MessageAttr
    _ [opt prefix] 'message = MessageName [stringlit]
    _ [repeat message_attr]
  construct MessageNameNoPrefix [stringlit]
    MessageName [deletePrefix]

  % Find the message with the name matching the message attribute and contextualize it.
  construct NewFaultContents [repeat element]
    _ [contextualizeMessages MessageNameNoPrefix each Messages]

  % Create the contextualized fault out of the contextualized message.
  construct NewFault [operation_scope]
```

```
    '<'fault AttrList '>
      NewFaultContents
    '</'fault>
  by
    PreviousFaults [. NewFault]
end function

% The contextualizeMessages function takes name and a <message> element and,
% if the name matches the <message>'s name attribute, contextualizes it,
% and appends it to a list.
function contextualizeMessages Name [stringlit] Message [message]
  replace [repeat element]
    PreviousMessages [repeat element]

  % Get the message's name
  deconstruct Message
    '< _ [opt prefix] 'message AttrList [opt attribute_list] '>
      MessageContents [repeat element]
    '</ _ [opt prefix] 'message>
  construct NameAttr [repeat name_attr]
    _ [^ AttrList]
  deconstruct NameAttr
    _ [opt prefix] 'name = MessageName [stringlit]
    _ [repeat name_attr]

  % Match the <message>'s name with the name passed as a parameter.
  % If they do not match, this will fail and the caller will move on
  % to the next <message>.
  deconstruct MessageName
    Name

  % Extract and contextualize each <part> element. Parts may refer to <element>s
  % or be typed (which may be defined in the <types> element as a <complexType>
  % or <simpleType>). We need to apply different rules for each case.
  construct Parts [repeat part]
    _ [^ MessageContents]
  construct PartsWithElements [repeat element]
    _ [contextualizePartsWithElement each Parts]
  construct PartsWithTypes [repeat element]
    _ [contextualizePartsWithType each Parts]
  construct NewMessageContents [repeat element]
    PartsWithElements [. PartsWithTypes]

  % Create the contextualized message using the contextualized parts.
  construct NewMessage [element]
    '<'message AttrList '>
      NewMessageContents
    '</'message>
  by
    PreviousMessages [. NewMessage]
end function

% The contextualizePartsWithElement function takes a <part> element as a parameter and
% contextualizes it by finding the element to which it refers.
function contextualizePartsWithElement Part [part]
  replace [repeat element]
    PreviousParts [repeat element]

  % Get the value of the <part>'s "element" attribute.
  deconstruct Part
    '< _[opt prefix] 'part AttrList [opt attribute_list] '/'>
  construct ElmtAttr [repeat element_attr]
    _ [^ AttrList]
  deconstruct ElmtAttr
```

```
    _ [opt prefix] 'element = ElmtName [stringlit]
    _ [repeat element_attr]

  % Get the <element>s from the schema from the namespace matching the prefix
  % of the element name we just found.
  import AllNamespaces [repeat xmlns_attr]
  import Schema [repeat schema]
  construct Prefix [stringlit]
    ElmtName [deleteSuffix]

construct MatchingNamespace [stringlit]
    _ [getMatchingNamespaceURI Prefix each AllNamespaces]
  construct MatchingSchema [repeat schema]
    _ [getMatchingSchema MatchingNamespace each Schema]
  construct SchemaElements [repeat schema_element]
    _ [^ MatchingSchema]

  % Find the <element> with a name that matches the <part>'s element attribute
  % and contextualize it.
  construct ElementNameNoPrefix [stringlit]
    ElmtName [deletePrefix]
  construct MatchingElements [repeat element]
    _ [getMatchingSchemaElement ElementNameNoPrefix each SchemaElements]

  % Remove some unneccessary <complexType> tags that only clutter the description.
  construct ProcessedMatchingElements [repeat element]
    MatchingElements   [stripComplexTypes1] [stripComplexTypes2] [stripComplexTypes3]
                       [stripComplexTypes4] [stripComplexTypes5] [stripComplexTypes6]

  % Create the new contextualized <part> with the contextualized <element>.
  % A special function is used to check if there are any matching elements to add
  % (schema may be referenced externally, but we don't have access).
  % If there aren't then no closing tag is used.
  construct Elmt [element]
    '<'part AttrList '/'>
  construct NewPart [element]
    Elmt [createPart ProcessedMatchingElements AttrList]
  by
    PreviousParts [. NewPart]
end function


% The contextualizePartsWithTy[e function takes a <part> element as a parameter and
% contextualizes it by finding the type to which it refers, if there is one
% (it may be a primitive type).
function contextualizePartsWithType Part [part]
  replace [repeat element]
    PreviousParts [repeat element]

  % Get the value of the <part>'s "type" attribute.
  deconstruct Part
    '< _[opt prefix] 'part AttrList [opt attribute_list] '/'>
  construct TypeAttr [repeat type_attr]
    _ [^ AttrList]
  deconstruct TypeAttr
    _ [opt prefix] 'type = TypeName [stringlit]
    _ [repeat type_attr]

  % Get the <complexTypes>s from the schema from the namespace matching the prefix
  % of the type name we just found.
  import AllNamespaces [repeat xmlns_attr]
  import Schema [repeat schema]
  construct Prefix [stringlit]
    TypeName [deleteSuffix]
  construct MatchingNamespace [stringlit]
    _ [getMatchingNamespaceURI Prefix each AllNamespaces]
  construct MatchingSchema [repeat schema]
    _ [getMatchingSchema MatchingNamespace each Schema]
  construct ComplexTypes [repeat complex_type]
    _ [^ MatchingSchema]

  % Find the <complexType> with a name that matches the <part>'s type attribute
  % and contextualize it.
  construct TypeNameNoPrefix [stringlit]
    TypeName [deletePrefix]
  construct MatchingElements [repeat element]
    _ [getMatchingType TypeNameNoPrefix each ComplexTypes]

  % Remove some unneccessary <complexType> tags that only clutter the description.
  construct ProcessedMatchingElements [repeat element]
    MatchingElements   [stripComplexTypes1] [stripComplexTypes2] [stripComplexTypes3]
                       [stripComplexTypes4] [stripComplexTypes5] [stripComplexTypes6]

  % Create the new contextualized <part> with the contextualized <complexType>.
  % A special function is used to check if there are any matching elements to add
  % (schema may be referenced externally, but we don't have access).
  % If there aren't then no closing tag is used.
  construct Elmt [element]
    '<'part AttrList '/'>
  construct NewPart [element]
    Elmt [createPart ProcessedMatchingElements AttrList]
  by
    PreviousParts [. NewPart]
end function


% The getMatchingSchemaElement function takes name and a <element> and,
% if the name matches the <element>'s name attribute, contextualizes it,
% and appends it to a list.
function getMatchingSchemaElement Name [stringlit] Elmt [schema_element]
  replace [repeat element]
    PreviousElements [repeat element]

  % Get the value of the <element>'s "name" attribute.
  construct ElmtAttrList [opt attribute_list]
    _ [getSingletonElementAttrList Elmt] [getElementAttrList Elmt]
  construct NameAttr [repeat name_attr]
    _ [^ Elmt]
  construct TypeAttr [repeat type_attr]
    _ [^ Elmt]
  deconstruct NameAttr
    _ [opt prefix] 'name = ElmtName [stringlit]
    _ [repeat name_attr]

  % Match the <element>'s name with the name passed as a parameter.
  % If they do not match, this will fail and the caller will move on
  % to the next <element>.
  deconstruct Name
    ElmtName

  % Get all the <complexType> elements.
  import Schema [repeat schema]
  construct SchemaTypes [repeat complex_type]
    _ [^ Schema]

  % We want to extract the <element>s from inside the matching <element> we found
  % so that they can be contextualized. But the TXL extract function (^) returns
  % the top-level <element> first in the list of extracted <element>s so we remove it.
  construct ParentAndChildren [repeat schema_element]
    _ [^ Elmt]
  % Remove the top-level <element>.
  deconstruct ParentAndChildren
    _ [schema_element] % throw away Parent
    Children [repeat schema_element]

  % Contextualize each of the <element>s extracted from the matching <element>
  construct ComplexChildren [repeat element]
    _ [contextualizeTypes each Children]

  % Create a new <element> using the attributes of the original, but containing
  % the newly consolidated children <element>s.
  construct NewElmt [element]
    Elmt

construct ComplexElement [element]
    NewElmt [createComplexElement ComplexChildren ElmtAttrList] [stripPrefix]
  by
    PreviousElements [. ComplexElement]
end function

% The getMatchingType function is similar to the get MatchingSchemaElement,
% but for <part>s that refer to a type.
% It takes name and a <complexType> and, if the name matches the <element>'s
% type attribute, contextualizes it, and appends it to a list.
function getMatchingType Name [stringlit] Elmt [complex_type]
  replace [repeat element]
    PreviousElements [repeat element]

  % Get the value of the <complexType>'s "name" attribute.
  deconstruct Elmt
    '< _ [opt prefix] 'complexType AttrList [opt attribute_list] '>
      _ [repeat element]
    '</ _ [opt prefix] 'complexType>
  construct NameAttr [repeat name_attr]
    _ [^ Elmt]
  deconstruct NameAttr
    _ [opt prefix] 'name = ElmtName [stringlit]
    _ [repeat name_attr]

  % Match the <complexType>'s name with the name passed as a parameter.
  % If they do not match, this will fail and the caller will move on
  % to the next <complexType>.
  deconstruct Name
    ElmtName

  % Extract the <element>s from inside the matching <complexType> we found so that
  % they can be contextualized.
  construct TypeElements [repeat schema_element]
    _ [^ Elmt]

  % Contextualize each of the <element>s extracted from the matching <complexType>.
  construct ContextualizedElements [repeat element]
    _ [contextualizeTypes each TypeElements]
  by
    PreviousElements [. ContextualizedElements]
end function

% The contextualizeTypes function takes a <element> (passed by parameter) and
% contextualizes it by finding the type to which it refers, if there is one
% (it may be a primitive type).
% This function forms a recursive loop with contextualizeComplexTypes function
% (the function following this one). It calls contextualizeComplexTypes,
% which calls contextualizeTypes, and so on until no more types can be resolved.
function contextualizeTypes Elmt [schema_element]
  replace [repeat element]
    PreviousElements [repeat element]

  % Get the value of the <element>'s "type" attribute.
  construct ElmtAttrList [opt attribute_list]
    _ [getSingletonElementAttrList Elmt] [getElementAttrList Elmt]
  construct TypeAttr [repeat type_attr]
    _ [^ ElmtAttrList]
  deconstruct TypeAttr
    _ [opt prefix] 'type = TypeString [stringlit]
    _ [repeat type_attr]
  construct Prefix [stringlit]
    TypeString [deleteSuffix]

  % Get the <complexTypes>s and <simpleTypes>s from the schema in the namespace
  % matching the prefix of the type name we just found.
  import AllNamespaces [repeat xmlns_attr]
  import Schema [repeat schema]
  construct MatchingNamespace [stringlit]
    _ [getMatchingNamespaceURI Prefix each AllNamespaces]
  construct MatchingSchema [repeat schema]
    _ [getMatchingSchema MatchingNamespace each Schema]
  construct SchemaComplexTypes [repeat complex_type]
    _ [^ MatchingSchema]
  construct SchemaSimpleTypes [repeat simple_type]
    _ [^ MatchingSchema]

  % Contextualize each <element> by finding the matching <complexType> or
  % <simplexType> from the ones just extracted.
  construct ComplexTypes [repeat element]
    _ [contextualizeComplexTypes TypeAttr each SchemaComplexTypes]
  construct SimpleTypes [repeat element]
    _ [contextualizeSimpleTypes TypeAttr each SchemaSimpleTypes]

  % Create a new <element> with the contextualized elements inside.
  construct Types [repeat element]
    SimpleTypes [. ComplexTypes]
  construct NewElmt [element]
    Elmt
  construct ComplexElement [element]
    NewElmt [createComplexElement Types ElmtAttrList] [stripPrefix]
  by
    PreviousElements [. ComplexElement]
end function

% The contextualizeComplexTypes function takes a "type" attribute (a list of them that
% should only contain one) and a <complexType> and, if the type matches the <element>'s
% type attribute, contextualizes it, and appends it to a list.
% This function is forms a recursive loop with contextualizeTypes function
% (the previous function). It calls contextualizeTypes, which calls
% contextualizeComplexTypes, and so on until no more types can be resolved.
function contextualizeComplexTypes TypeAttr [repeat type_attr] ComplexType [complex_type]
  replace [repeat element]
    PreviousTypes [repeat element]

  % Get the value of the <complexType>'s "name" attribute.
  deconstruct ComplexType
    '< _ [opt prefix] 'complexType AttrList [opt attribute_list] '>
      Contents [repeat element]
    '</ _ [opt prefix] 'complexType>
  construct TypeNameAttr [repeat name_attr]
    _ [^ AttrList]
  deconstruct TypeNameAttr
    _ [opt prefix] 'name = TypeName [stringlit]
    _ [repeat name_attr]

  % Get the value of the "type" attribute from the one passed, the <complexType>
  % we're trying to find.
  deconstruct TypeAttr
    _ [opt prefix] 'type = ElmtType [stringlit]
    _ [repeat type_attr]
  construct ElmtTypeNoPrefix [stringlit]
    ElmtType [deletePrefix]

  % Match the <complexType>'s name with the type passed as a parameter.
  % If they do not match, this will fail and the caller will move on to the
  % next <complexType>.
  deconstruct TypeName
    ElmtTypeNoPrefix

  % Extract all of the <element>s from the matching <complexType> and contextualize them.
  construct SchemaElements [repeat schema_element]
    _ [^ Contents]

construct ComplexElements [repeat element]
    _ [contextualizeTypes each SchemaElements]
  by
    PreviousTypes [. ComplexElements]
end function

% The contextualizeSimpleTypes function takes a "type" attribute (a list of them
% that should only contain one) and a <simpleType> and, if the type matches the
% <element>'s type attribute, contextualizes it, and appends it to a list.
function contextualizeSimpleTypes TypeAttr [repeat type_attr] SimpleType [simple_type]
  replace [repeat element]
    PreviousTypes [repeat element]

  % Get the value of the <simpleType>'s "name" attribute.
  deconstruct SimpleType
    '< _ [opt prefix] 'simpleType AttrList [opt attribute_list] '>
      Contents [repeat element]
    '</ _ [opt prefix] 'simpleType>
  construct TypeNameAttr [repeat name_attr]
    _ [^ AttrList]
  deconstruct TypeNameAttr
    _ [opt prefix] 'name = TypeName [stringlit]
    _ [repeat name_attr]

  % Get the value of the "type" attribute from the one passed, the <complexType>
  % we're trying to find.
  deconstruct TypeAttr
    _ [opt prefix] 'type = ElmtType [stringlit]
    _ [repeat type_attr]
  construct ElmtTypeNoPrefix [stringlit]
    ElmtType [deletePrefix]

  % Match the <simpleType>'s name with the type passed as a parameter.
  % If they do not match, this will fail and the caller will move on to
  % the next <complexType>.
  deconstruct TypeName
    ElmtTypeNoPrefix

  % Create a new <element> with the contexts of the matching <simpleType> inside.
  % (Unlike <complexType>s, <simpleType>s do not contain <element>s; they contain
  % some kind of restriction like ennumeration.
  % Therefore, we don't have to contextualize any further.)
  construct Elmt [element]
    SimpleType
  construct ComplexElement [element]
    Elmt [createComplexElement Contents AttrList] [stripPrefix]
  by
    PreviousTypes [. ComplexElement]
end function

% This helper function replaces a [stringlit] of the form X:Y with Y
% (deletes the prefix).
% This is used to get the name or type of an element since they will
% often have prefixes specifying a nampespace.
function deletePrefix
  replace [stringlit]
    NameWithPrefix [stringlit]
  construct IndexOfColon [number]
    _ [index NameWithPrefix ":"] [+ 1]
  deconstruct not IndexOfColon
    1
  by
    NameWithPrefix [: IndexOfColon 999]
end function

% This helper function replaces a [stringlit] of the form X:Y with X
% (deletes the suffix).
% Prefixes are used to specify a namespace. This function is used to get
% the namespace prefix of an element's type so that it can find the schema
% in the same namespace.
function deleteSuffix
  replace [stringlit]
    NameWithSuffix [stringlit]
  construct IndexOfColon [number]
    _ [index NameWithSuffix ":"] [- 1]
  deconstruct not IndexOfColon
    1
  by
    NameWithSuffix [: 0 IndexOfColon]
end function

% This helper function is to get the attribute list of a singleton element
% (one that contains no other elements).
% This (along with getElementAttrList) is used sometimes because
% TXL will extract all attributes, even from elements inside.
% We use this becuase we want to make sure we have the parent element's attributes.
% And since, we don't know whether the element is singleton or not, we apply both;
% one fails, the other won't.
function getSingletonElementAttrList Elmt [schema_element]
  deconstruct Elmt
    '< _ [opt prefix] 'element AttrList [opt attribute_list] '/>
  replace [opt attribute_list]
    _ [opt attribute_list]
  by
    AttrList
end function

% This helper function is to get the attribute list of an element.
% This (along with getSingletonElementAttrList) is used sometimes because
% TXL will extract all attributes, even from elements inside.
% We use this becuase we want to make sure we have the parent element's attributes.
% And since, we don't know whether the
% element is singleton or not, we apply both; one fails, the other won't.
function getElementAttrList Elmt [schema_element]
  deconstruct Elmt
    '< _ [opt prefix] 'element AttrList [opt attribute_list] '>
```

```
      _ [repeat schema_scope]
    '</ _ [opt prefix] 'element>
  replace [opt attribute_list]
    _ [opt attribute_list]
  by
    AttrList
end function

% This helper function takes a list of elements and an attribute list and creates
% a new <element> with the given attribute list, containing the given elements.
% We use it so that if there are no elements in the list, we
function createComplexElement Contents [repeat element] AttrList [opt attribute_list]
  deconstruct Contents
    _ [element]
    _ [repeat element]
  replace [element]
    _ [element]
  by
    '<'element AttrList '>
      Contents
    '</'element>
end function

% This helper function takes a list of elements and an attribute list and creates
% a new <element> with the given attribute list, containing the given elements.
function createPart Contents [repeat element] AttrList [opt attribute_list]
  deconstruct Contents
    _ [element]
    _ [repeat element]
  replace [element]
    _ [element]
  by
    '<'part AttrList '>
      Contents
    '</'part>
end function

% This helper function is applied to <element>'s to remove the namespace prefix
% ([stringlit] :) from the tag so that <element>s from different namespaces
% can be matched by clone detection.
% After contextualization, namespaces don't mean anything anymore.
function stripPrefix
  replace [element]
    '< _[opt prefix] 'element AttrList [attribute_list] '/>
  by
    '<'element AttrList '/>
end function

% This helper function matches a namespace prefix to a URI used to find the
% schema in that namespace.
function getMatchingNamespaceURI Prefix [stringlit] Namespace [xmlns_attr]
  replace [stringlit]
    _ [stringlit]
  deconstruct Namespace
    'xmlns: PrefixID [id] '= NamespaceURI [stringlit]
  construct PrefixString [stringlit]
    _ [quote PrefixID]
  deconstruct PrefixString
    Prefix
  by
    NamespaceURI
end function

% This helper function is used to find the schema from a given namespace, given its URI.
```

```
function getMatchingSchema Namespace [stringlit] Schema [schema]
  replace [repeat schema]
    PreviousSchema [repeat schema]
  construct TargetNamespace [repeat target_namespace_attr]
    _ [^ Schema]
  deconstruct TargetNamespace
    _ [opt prefix] 'targetNamespace = URI [stringlit]
    _ [repeat target_namespace_attr]
  deconstruct URI
    Namespace
  by
    PreviousSchema [. Schema]
end function

% The following helper functions remove <complexType> tags, along with other tags
% used with them (like <sequence). These tags clutter up the operation description
% and are not necessary for clone detection (and may even result in poor clones).
rule stripComplexTypes1
  replace [element]
    '< _[opt prefix] 'element AttrList [opt attribute_list] '>
      '< _[opt prefix] 'complexType>
        '< _[opt prefix] 'sequence>
          Contents [repeat element]
        '</ _[opt prefix] 'sequence>
      '</ _[opt prefix] 'complexType>
    '</ _ [opt prefix] 'element>
  construct NewElmt [element]
    '<'element AttrList '>
      Contents
    '</'element>
  by
    NewElmt
end rule

rule stripComplexTypes2
  replace [element]
    '< _[opt prefix] 'element AttrList [opt attribute_list] '>
      '< _[opt prefix] 'complexType>
        '< _[opt prefix] 'sequence>
        '</ _[opt prefix] 'sequence>
      '</ _[opt prefix] 'complexType>
    '</ _ [opt prefix] 'element>
  construct NewElmt [element]
    '<'element AttrList '/>
  by
    NewElmt
end rule

rule stripComplexTypes3
  replace [element]
    '< _[opt prefix] 'element AttrList [opt attribute_list] '>
      '< _[opt prefix] 'complexType>
        '< _[opt prefix] 'sequence/>
      '</ _[opt prefix] 'complexType>
    '</ _ [opt prefix] 'element>
  construct NewElmt [element]
    '<'element AttrList '/>
  by
    NewElmt
end rule

rule stripComplexTypes4
  replace [element]
    '< _[opt prefix] 'element AttrList [opt attribute_list] '>
```

```
      '< _[opt prefix] 'complexType>
        '< _[opt prefix] 'all>
          Contents [repeat element]
        '</ _[opt prefix] 'all>
      '</ _[opt prefix] 'complexType>
    '</ _ [opt prefix] 'element>
  construct NewElmt [element]
    '<'element AttrList '>
      Contents
    '</'element>
  by
    NewElmt
end rule

rule stripComplexTypes5
  replace [element]
    '< _[opt prefix] 'element AttrList [opt attribute_list] '>
      '< _[opt prefix] 'complexType>
        '< _[opt prefix] 'all>
        '</ _[opt prefix] 'all>
      '</ _[opt prefix] 'complexType>
    '</ _ [opt prefix] 'element>
  construct NewElmt [element]
    '<'element AttrList '/>
  by
    NewElmt
end rule

rule stripComplexTypes6
  replace [element]
    '< _[opt prefix] 'element AttrList [opt attribute_list] '>
      '< _[opt prefix] 'complexType>
        '< _[opt prefix] 'all/>
      '</ _[opt prefix] 'complexType>
    '</ _ [opt prefix] 'element>
  construct NewElmt [element]
    '<'element AttrList '/>
  by
    NewElmt
end rule
```