The Domain Name System Advisor, A Model-Based Quality Assurance Framework

Thesis submitted for the degree of
Doctor of Philosophy
at the University of Leicester

by

Marwan Mohammed Mahmoud Radwan
Department of Informatics
University of Leicester

May 2017

Declaration of Authorship

I hereby certify that the thesis I am submitting is entirely my own original work except where otherwise indicated. I am aware of the University of Leicester's regulations concerning plagiarism, including those regulations concerning disciplinary actions that may result from plagiarism. Any use of the works of any other author, in any form, is properly acknowledged at their point of use.

- Student's signature:
- Name (in capitals): MARWAN MOHAMMED MAHMOUD RADWAN
- Date of submission: 7^{th} of December 2016

Abstract

The Domain Name System (DNS) has a direct and strong impact on the performance of nearly all aspects of the Internet. DNS relies on a delegation-based architecture, where resolution of names to their IP addresses require resolving the names of the servers responsible for those names.

The recursive graphs of the inter-dependencies that exist between servers associated with each zone are called *Dependency Graphs*. We constructed a *DNS Dependency Model* as a unified representation of these *Dependency Graphs*. We utilized a set of *Structural Metrics* defined over this model as indicators of external quality attributes of the DNS. We applied machine learning in order to construct *Prediction Models* of the perceived quality attributes of the DNS out of the structural metrics of the model and evaluate the accuracy of these models.

Operational Bad Smells are configuration and deployment decisions, made by zone administrators, that are not totally errant or technically incorrect and do not currently prevent the system from doing its designated functionality. Instead, they indicate weaknesses that may impose additional overhead on DNS queries, or increase system vulnerability to threats, or increase the risk of failures in the future.

We proposed the ***ISDR*** *(**I**dentification, **S**pecification, **D**etection and **R**efactoring) Method* that enables DNS administrators to identify bad smells on a high-level abstraction using a consistent taxonomy and reusable vocabulary. We developed techniques for systematic detection and recommendations of reaction mechanisms in the form of graph-based refactoring rules.

The *ISDR Method* along with the *DNS Quality Prediction Models* are used to build the *DNS Quality Assurance Framework* and the *DNS Advisor Tool*. Assessing the perceived quality attributes of the DNS at an early stage enables us to avoid the implications of defective and low-quality designs. We identify configuration changes that improve the availability, security, stability and resiliency postures of the DNS.

In the name of Allah, the Most Beneficent, the Most Merciful.

Acknowledgements

First and above all, I praise God, the Almighty, for providing me this opportunity and granting me the capability to proceed successfully.

This thesis appears in its current form due to the assistance and guidance of several people. I would like to offer my sincere thanks to all of them. In particular, I am profoundly indebted to my PhD advisor, Professor Dr. Reiko Heckel, who was very generous with his time and knowledge and assisted me in each step to complete this project. I have been extremely lucky to have a supervisor who cared so much about myself and my work, and who responded to my questions and queries so promptly. Reiko has also provided insightful discussions about each part of this research. I also thank my second supervisor Dr. Emilio Tuosto and PhD tutor, Dr. Fer-Jan de Vries, for their support and constructive discussions during the annual viva sessions.

My late father and mother; I just simply wish you were alive today to share this moment with me and the rest of the family. I wish to thank my family, especially my wife, Mai, for her sincere love, care and support throughout this entire period and for providing the much needed motivation by encouragement, and taking care of the kids. I thank her for believing in me even when I did not. Her quiet patience, unwavering love and tolerance of my occasional temper moods is a testament in itself of her unyielding devotion and love.

I also owe my affectionate gratitude to my sister Sadia, who has been continuous support to me. My father-in-law, mother-in-law, brothers, sisters, and their families always encouraged me to stand where I am today. I would like to thank my friends back in Palestine, and specially PNINA staff, for their support and restless efforts in keeping the organisation running smoothly while I am abroad.

Last but not the least, I may have slipped some names to mention here but I say thanks to everyone in Gaza, Palestine and Leicester, United Kingdom for being supportive and well-wisher to me in this period of life. God bless and guide you all.

Contents

List of Figures

List of Tables

List of Algorithms

I dedicate this thesis to

my late parents,

who showed me the way,

and my wife,
who has always supported me in my journey.

Chapter 1

Introduction

The Domain Name System (DNS) is one of the most fundamental infrastructures of today's Internet. The critical importance of the DNS raises high demands for its stability, security and resilience. The DNS is a distributed database for storing information on domain names, the primary namespace for hosts on the Internet. The name space is organised in a hierarchical structure to ensure domain name uniqueness. Each node in the DNS tree corresponds to a zone. Each zone belonging to a single administrative authority is served by multiple name servers. In addition to IP addresses, the DNS is used to look up mail servers, cryptographic keys, latitude and longitude values, and other diverse types of data. The use of the Internet is critically dependent on the reliable, trustworthy, and responsive operation of the DNS.

1.1 Motivations

The correct and error-free operation of the DNS is crucial for the reliability of most applications on the Internet. Delegation is crucial in achieving DNS name space's

scalability however there are many misconfigurations and bad deployment choices made by system administrators that may lead to data inconsistencies, vulnerable configurations or even failure of resolution. A mistake in configuring a specific DNS zone may potentially have adverse impacts on the global Internet [2, 3].

While DNS plays a critical role for the operation of Internet, DNS zone administration relies heavily on error-prone manual configurations. Operational guidelines [4–6] require that a zone have multiple authoritative name servers, and that they be distributed through diverse network topological and geographical locations to increase the reliability of that zone as well as improve overall network performance and access. These are meant to make DNS services robust against unexpected failures. Recent work [7–12] outlines the need for zone operators to understand how many inter-dependencies they may inadvertently be incurring through the deployment and sharing of DNS secondary servers.

This research is motivated by the lack of formal analysis of the DNS interdependencies stemming from the delegation-based architecture as well as operational deployment choices made by system administrators. Therefore, the need for early indicators of perceived quality attributes is recognized in order to avoid the implications of defective and low-quality configurations and deployment choices during the late stages of operation.

We use perceived quality as a mechanism at the model level to approximate the indicators of real system quality attributes. Efforts to improve risk management related to DNS security, stability and resilience must be guided by an ability to predict these characteristics and apply correction mechanisms to rectify for any degrading in these quality attributes as a result of a misconfiguration or bad deployment choice.

1.2 Problem Statement

The large body of literature on DNS operation [4, 7–9, 13–19] suggests that the area is mature, and problems are well understood. However, reality is contrary to this suggestion. The DNS ecosystem has evolved to include many players, such as DNS entities, which include trusted, untrusted, and semi-trusted ones, making it very difficult to reason about its resolution and operation. DNS is well known to have configuration issues. Jung et al. [20] found that a significantly high amount of Internet traffic is caused by miss-configured DNS servers and resolvers. These configuration issues may lead to performance degradation, crashes, and endless loops in resolvers. They may also confuse users by returning inappropriate error messages propagated by applications.

The original DNS design focused mainly on system robustness against physical failures, and neglected the impact of operational errors such as misconfiguration and bad deployment choices. Several previous measurements [13, 15, 17] showed that zones with configuration errors suffer from reduced availability and increased query delays up to an order of magnitude. DNS administrators have to decide on operational parameters and be aware of their implications for the DNS's overall system qualities.

On the deployment level, configuring the number of redundant authoritative DNS servers for a certain zone must take into consideration the operational overhead associated with querying multiple servers in parallel. Choosing servers with names under other zones provides zone redundancy but may incur security and resiliency threats to the zone. Deciding on where to physically locate the servers should ensure a certain degree of resistance against different types of failures. Peering with external organizations for secondary server hosting should take into consideration the impact of transitional trust and administrative complexity [9, 16].

The original DNS design documents [5, 6, 21–23] call for diverse placement of authoritative name servers for a zone. Bad configurations may lead to *cyclic dependencies* while bad deployment choices may lead to *diminished and false server redundancy*. It is also assumed that redundant DNS servers fail independently; previous measurements [7, 13] showed that operational deployment choices made at individual zones can introduce *excessive zone influence*. All those *bad smells* severely affect the availability, security, stability and resiliency of the overall domain name system.

1.3 Solution Outline

System administrators' operational decisions have far reaching effects on the DNS's quality attributes. They need to be soundly made to create a balance between the availability, security, stability and resilience of the system. We need to be able to direct the zone administrator to places in the zone file that contain potential design and deployment problems that may compromise availability, resiliency or security of a domain name before the changes become into production.

In order to achieve this goal, we approached the problem from a design point of view that takes into consideration the DNS zone configuration and server deployment choices rather than from the dynamic behavioural view which includes statistical and post-deployment measurements.

Since many of the misconfiguration can not be detected from the zone file or deployments directly, there is a need for a DNS model that encompasses all information related to the zone file and the server deployments in one conceptual graph. The conceptual graph representation facilitates modelling at multiple levels of detail simultaneously.

We constructed a DNS Dependency Model *(The DNS Model)* as a unified representation of these Dependency Graphs. We utilized a set of *Structural Metrics* defined over this model as indicators of external quality attributes of the domain name system. We applied some machine learning algorithms in order to construct *Prediction Models* of the perceived quality attributes of the operational system out of the structural metrics of the model. Assessing these quality attributes at an early stage of the design/deployment enables us to avoid the implications of defective and low-quality designs.

We proposed the ***ISDR*** *(**I**dentification, **S**pecification, **D**etection and **R**efactoring) Method* to identify, specify and detect misconfiguration and bad deployment choices in the form of operational bad smells. The method deals with smells on a high-level of abstraction using a consistent taxonomy and reusable vocabulary. The method also utilizes the set of structural metrics defined over the *DNS Model* to detect the smells in early stages of the DNS deployment. It also suggests graph-based refactoring rules as correction mechanisms for the bad smells. We apply and validate the method using several representative case studies. The method techniques are used as early indicators of external quality attributes in order to avoid the implications of defective and low-quality designs and deployment choices.

The method is integrated within a DNS advisory tool to flag configuration changes that might decrease the robustness or security posture of a domain name, before even the changes become into production.

We built the tool based on graph-based model transformation tools and techniques and validated our approach through empirical assessments and several case studies.

1.4 Contributions

The contributions of this research are:

1. Building a DNS Dependency Model to describe the Domain Name System operational world to detect violations of the design and deployment principles at the authoritative level.

2. Identification of a set of structural metrics *DNS Metrics Suite*, defined over the DNS model, and building prediction models for the various DNS quality attributes out of these metrics.

3. Proposing the *ISDR method* which is a model-based approach that subsumes all the steps necessary to identify, specify, formalise, detect and catalogue the DNS operational bad smells. The method deals with smells on a high-level of abstraction using a consistent taxonomy and reusable vocabulary, defined over the *DNS Model*. Graph-based refactorings are proposed as correction mechanisms for those bad smells and their priorities, conflicts and dependabilities are analysed and their quality impacts are verified.

4. Building a pre-emptive *DNS Advisor Tool* that implements the ISDR method and related model transformation techniques in order to detect and flag configuration changes that might decrease the robustness or security posture of a domain name, before even the changes become into production.

1.5 Publications

In this section some of the relevant co-authored documents are listed. Material from these publications has been used in this dissertation since it was developed in the context of this PhD research.

1. Radwan, Marwan and Reiko Heckel, "Refactoring Operational Smells within the Domain Name System", Software Technologies: Applications and Foundations (STAF-14) Conference, University of York, UK, 21-25 July 2014.

2. Radwan, Marwan, and Reiko Heckel. "Detecting and Refactoring Operational Smells within the Domain Name System." 1st Graph as Models (Gam) Workshop, ETAPS-2015, 11-12 April 2015, London, United Kingdom, arXiv preprint arXiv:1504.02615 (2015).

3. Radwan, Marwan and Reiko Heckel, "Prediction of the Domain Name System (DNS) Quality Attributes," Paper accepted for publication, The 32nd ACM Symposium on Applied Computing (SAC 32), April 3-6, 2017, Marrakesh, Morocco, http://dx.doi.org/10.1145/3019612.3019728.

1.6 Thesis Outline

The main body of this thesis consists of eight chapters followed by a final chapter drawing conclusions and giving suggestions for future work. Following this introduction, Chapter 2 discusses relevant background about the design, operation and structure of the DNS. It describes the DNS on a high level to build up the necessary background for the succeeding chapters.

Chapter 3 presents the DNS Model with its main components, features, relationships and integrity constraints. It also includes a background section on related formalisms of graph transformation concepts and definitions related to this thesis. In Chapter 4 a set of structural metrics is defined over the DNS Dependency Model. Prediction models of the perceived quality attributes of the DNS model are constructed out of the structural metrics.

Chapter 5 discusses the ISDR method which includes bad smells' identification, specification, and detection. Examples of correction mechanisms in the form of graph-based refactoring rules are also presented. Chapter 6 discusses the specifications, tools and implementation of the ISDR method techniques and presents several case studies for its validation.

Chapter 7 continues with the implementation of the advisory tool by discussing the behaviour preservation properties of the refactorings and analysing their execution priorities, conflicts and dependabilities. As prototype of the DNS Advisor is presented as a realization of the all artefacts and techniques presented in this thesis.

Chapter 8 gives a survey of related work in the subjects discussed within the different chapters of this research and relates them to our research contributions. Chapter 9 concludes the thesis with lessons learned, research limitations and directions of future work.

Chapter 2

Background

The outline of the chapter is as follows. Section 2.1 gives an overview and a background of the Domain Name System (DNS) operation including the details of the DNS query process, DNS interdependencies, DNS operational planes and introduces the concept of dependency graphs. Then, the DNS quality attributes and DNS health indicators are discussed in Section 2.2. Relevant graph transformation theoretical background and supporting tools are presented in Section 2.3. Finally models as graphs and model transformation approaches and tools are discussed in Section 2.4.

2.1 The Domain Name System

The Domain Name System (DNS) is a hierarchical distributed database [22] that can map names to some data. In most cases it is used to map a name to an Internet Protocol (IP) address. The system is mature and very successful. It was designed in the 80's to replace host-local configuration files for naming of Internet hosts. DNS queries consist of a single User Datagram Protocol (UDP) request from the client

followed by a single UDP reply from the server. The Transmission Control Protocol (TCP) is used when the response data size exceeds 512 bytes, or for tasks such as zone transfers [23]. DNS is part of the application layer of the TCP/IP reference model [24].

2.1.1 General Operation of the DNS

The DNS is a conceptually simple system that allows a string of labels (such as "www", "le", "ac", and "uk") joined by dots into a "domain name" to be looked up in a database distributed across multiple DNS servers. The dots in a domain name are important because they represent potential administrative boundaries. For example, the dot between "ac" and "uk" in the domain name "www.le.ac.uk" represents the administrative boundary between the "uk" top-level domain and Janet, the organization responsible for the "ac.uk".

The Internet's domain name space is a single large tree, read right-to-left, with progressively more specific administrative units to the left. The term *zone* is used to indicate administrative units within the DNS tree. For example, the "le.ac.uk" zone is the piece of the DNS tree including all names ending in ".le.ac.uk". Further subdivisions are common, even within a single organization, and "le.ac.uk" might have multiple zones, such as "cs.le.ac.uk", "art.le.ac.uk" and "eng.le.ac.uk".

A *zone* is a point of delegation in the DNS tree. It contains all names from a certain point downward except those which are delegated to other zones. A *delegation point* has one or more NS records in the *parent zone*, which should be matched by equivalent NS records at the root of the "delegated zone". [14].

Each name server maintains the domain name information regarding a zone in the DNS name space. Several predefined properties, or resource records (RR), can be

associated with a domain name. All of the RRs pertaining to the domain names in a zone are stored in a master file, maintained by the primary name server of that zone. Each zone also has one or more secondary name server, which periodically synchronizes its local DNS file with the master file.

Secondary name servers respond to DNS queries but are not involved in maintaining the master file. Operators of each zone determine the number of authoritative name servers and their placement and manage all changes to the zone's data content. In spite of the fact that zone administration is autonomous, coordination is required and essential to maintain the consistency, stability and resilience of the DNS hierarchy. The DNS architecture was later enhanced with DNS Security Extensions [25], [26] to provide data origin authentication.

Types of Domain Name Servers [27]:

- **Root Servers**. The name servers that serve the DNS root zone, commonly known as the root DNS servers, are a network of hundreds of servers in many countries around the world. They are configured in the DNS root zone as 13 named authorities (labeled A through M). Operators who manage DNS resolvers typically need to configure a "root hints file". This file contains the names and IP addresses of the root servers, so the software can bootstrap the DNS resolution process.

- **Top-Level Domain (TLD) Servers**. These servers are responsible for top-level domains such as com, org, net, edu, and gov, and all of the country top-level domains such as uk, ca, and ps. The company Verisign Global Registry Services maintains the TLD servers for the com top-level domain, the company Educause maintains the TLD servers for the edu top-level domain and the organisation Nominet maintains the uk country code TLD.

- **Authoritative Servers**. Every organization with publicly accessible hosts (such as Web servers and mail servers) on the Internet must provide publicly accessible DNS records that map the names of those hosts to IP addresses. An organization's authoritative server houses these DNS records for that particular organisation.

- **Recursive Resolvers or Local DNS Servers**. There is another important type of DNS server called Recursive Resolvers or local DNS servers. Resolvers make queries (recursively) on behalf of applications and (usually) cache the responses to improve DNS performance and scalability. In the case of smaller enterprises and end users, Internet service providers typically operate resolvers. In the case of larger enterprises, the resolvers are usually operated by the enterprises themselves or by large-scale DNS hosting providers.

A root, top-level domain or authoritative server responds to DNS lookup requests with one of the following responses:

- A positive response in which an answer to the question is provided;
- A negative response indicating the answer does not exist; or
- A referral providing an indication of where further information may be obtained.

Authoritative servers are typically operated by or on behalf of zone administrators. DNS registrars, and hosting providers often operate authoritative servers on behalf of their customers. The authoritative DNS infrastructure, particularly for "high value" zones such as top-level domains, is being increasingly outsourced to DNS-focused service providers.

A local DNS server does not strictly belong to the hierarchy of servers but is nevertheless central to the DNS architecture. When a host makes a DNS query, the query is sent to the local DNS server, which acts as a proxy, forwarding the query into the DNS server hierarchy.

2.1.2 DNS Query Process

The most common type of DNS lookup is for IP addresses. This is the type of lookup that occurs each time a user types a URL into a web browser. Normally, the individual application (such as the web browser) does not perform the full lookup, which involves several steps. Figure 2.1 shows the process by which an application looks up the domain name www.le.ac.uk and how it is mapped to the DNS data, control and management operational planes.

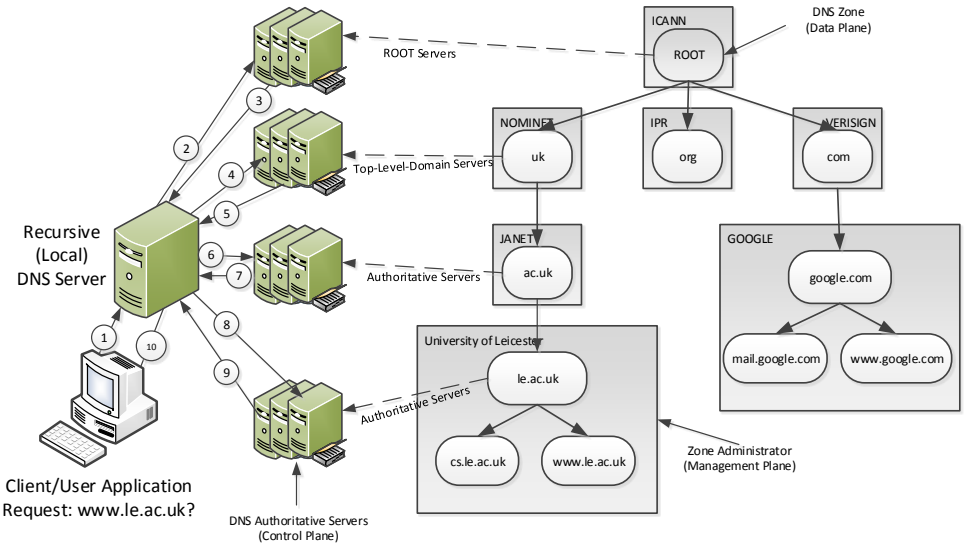


FIGURE 2.1: An illustration of the DNS resolution process.

To find the IP address of www.le.ac.uk, the client (e.g a web browser) submits a DNS query to a local (recursive) DNS server (step 1). Assuming that the corresponding

IP is not in the resolver cache, it will ask one of the root name servers for the translation (step 2). The names and IP addresses of root name servers are locally stored within each server. The root name servers will respond with a "referral", telling the resolver to query the DNS servers of the .uk top-level-domain for an answer (step 3). The resolver then repeats this process for the .uk name servers and get a referral to ask the .ac.uk authoritative name servers which in turn answers with a referral to as the le.ac.uk name servers (step 4 -7). The resolver next asks one of the le.ac.uk name servers for the translation (step 8), and gets the answer in step (9), and finally forwards the answer to the requesting client (step 10) who will use this information to connect to the web server hosting the web site www.le.ac.uk. Throughout the process, resolvers may encounter name servers hosted under other zones whose names need to be resolved before contacting them about the original request.

Any DNS lookup process may involve the operators of numerous Internet-connected networks, physical and virtual servers, support and back-office systems, and related infrastructure. The many parties and components involved in every single DNS lookup multiply the potential risks to the availability, security, stability, and resilience of the DNS. Due to the importance of the DNS for the operation of the Internet, any event that negatively impacts DNS Security, Stability, or Resiliency would have significant impact on the Internet.

2.1.3 DNS Inter-dependencies

Inter-dependencies are common in the DNS and stem from the hierarchal structure of the DNS, the DNS protocol as well as from different motivations and goals. A zone is said to depend on a name server if the name server could be involved in

the resolution of names in that zone. The dependencies among name servers that directly or indirectly affect a zone are represented as a dependency graph.

2.1.3.1 DNS Operational Planes

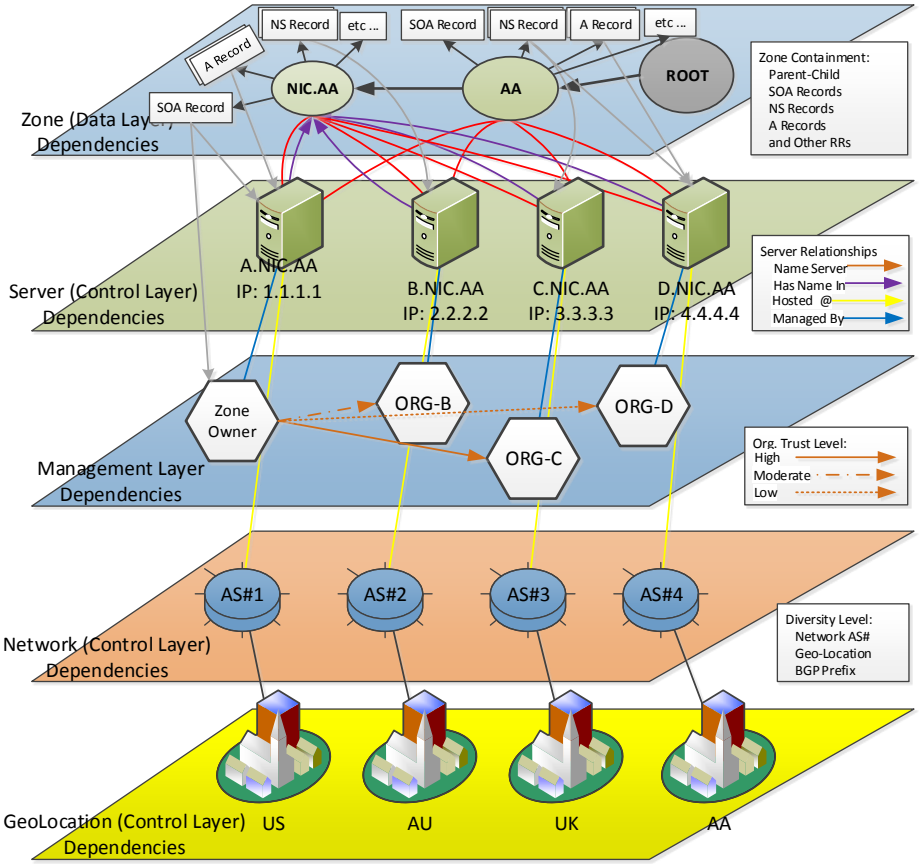


FIGURE 2.2: Interdependencies within the DNS Operational Planes.

The zone's data plane is the interconnected graph of all infrastructure resource records defined within the zone's configuration file. The interconnected graph of all authoritative name servers involved in the resolution process of a domain within a certain zone is called the zone's control plane and the interconnected graph of all administrative units (organisations) involved is called the management plane.

One reason that the DNS is so powerful is that its data plane allows administrators a great deal of flexibility: they can manage their name space however they like. However, the control and management planes' flexibility can lead to operational problems if not managed conscientiously. Figure 2.2 shows a schematic of the various interdependencies that occur within and between the three operational planes of the domain name system.

2.1.3.2 Dependency Graphs

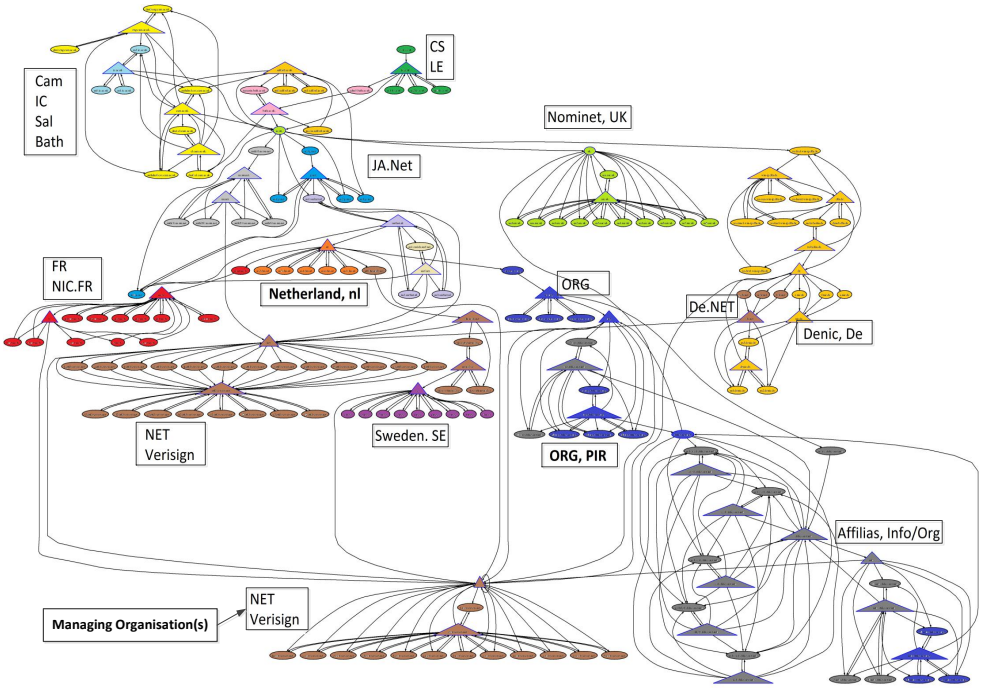


FIGURE 2.3: Name Dependency Graph of (le.ac.uk).

The recursive structures of inter-dependencies within and between the DNS operational planes are represented by dependency graph. A dependency graph deccio2010 is a directed connected graph with a distinguished node (r) which is the root zone. Each node in the graph represents a zone name, and each edge signifies that its source is directly dependent on its target for proper resolution of itself and any descendant domain names.

Figure 2.2 shows deferent dependencies that occur at the different DNS operational planes. Zone or Data-Layer dependencies include child-parent dependency between a zone and its parents as well as dependencies within the different resource records within a zone or delegated zones. Control-layer dependencies includes the various dependencies stemming from hosting zone files within secondary name server who has names under other zones. It also include network distribution dependencies as well as geographical location dependencies of those servers. Management layer dependencies includes trust relationship between the different organisations that manage the name servers of a particular zone and their interactions.

2.2 DNS Quality Attributes and DNS Health

Due to the fact that we aim at modelling the DNS system from the perspective of authoritative system administrators and zone managers, we are concerned with the DNS perceived quality as anticipated by the system administrator during the process of designing, configuring and deployment of the DNS system. Quality attributes have different definitions based on the point of view of the DNS user. For example, resilience is viewed by users as availability and viewed by providers as a combination of detection, response, resistance and recovery processes that increase overall confidence in relying on and investing in the Internet over the long-term [11].

In this research, we focus on four quality attributes of the DNS as perceived by authoritative zone managers and system administrators [10] and they are:

- *Availability* is defined as the ability of the group of authoritative name servers of a particular zone (e.g., a TLD), to answer DNS queries. For the service to be considered available at a particular moment, at least two of the delegated

name servers registered in the DNS must have successful results to each of their public-DNS registered "IP addresses" to which the name server resolves.

- *Security* is the ability of the components of the system to protect the integrity of DNS information and critical system resources.

- *Stability* is the consistency of authoritative name servers' names within the system and the consistency of system components' performance over time. That is, if authoritative servers' names within a system change with high frequency, the system is unstable and if a query takes 10 milliseconds to respond in one instance and 1000 milliseconds to respond in a second instance, resolution time is unstable which means the system is also unstable.

- *Resilience* is the ability of the system to provide and maintain an acceptable level of name resolution service in the face of faults and changes in normal operating conditions.

Given the fact that the DNS protocol offers administrators and zone operators a high level of flexibility in configuring their zone and the deployment structure for their systems, it can be anticipated that low-quality configurations and deployment choices can ripple through to many operational domain name systems. Therefore, the need for early indicators of external quality attributes is recognized in order to avoid the implications of defective and low-quality design and deployment during the late stages of system operation.

The security, stability and resilience of DNS have received significant attention over the past few years. Following the 2009, 2010 and 2012 DNS symposia [28], [29], [30], the Internet Corporation for Assigned Names and Numbers (ICANN) specified the following indicators for DNS health:

- **Availability:** The ability of DNS to be operational and accessible when required.
- **Coherency:** The ability of DNS to accurately resolve name queries; this is one of the core principles of DNS. For example, if the IP address 192.0.2.1 is resolved to www.foo.example.com, then the coherency principle implies that the name www.foo.example.com should resolve to the IP address 192.0.2.1.
- **Integrity:** The ability of DNS to guard against improper data modification or destruction; this includes ensuring information non-repudiation and authenticity.
- **Resiliency:** The ability of DNS to effectively respond and recover to a known, desired and safe state in the event of a disturbance.
- **Security:** The ability of DNS to limit or protect itself from malicious activities (e.g., unauthorized system access, fraudulent representation of identity and interception of communications).
- **Speed:** The performance of DNS with respect to response time and throughput. Note that, in addition to queries, speed applies to maintenance, administration and management operations.
- **Stability:** The ability of DNS to function in a reliable and predictable manner (e.g., protocols and standards). Stability is important because it facilitates universal acceptance and usage.
- **Vulnerability:** The likelihood that a DNS weakness can be exploited by one or more threats.

2.3 Graph Transformation

Graphs and diagrams have been used to represent a variety of problems in computer science and software engineering. They provide a simple mathematical model for representing pairs of objects connected by links [31]. More formally, a graph consists of a set of vertices V and a set of edges E, each edge having a source and a target vertex in V. Graphs can be typed, allowing for the definition of meta-models that describe how instances should be built. Additionally, in order to carry further information, it is possible to use attributes in graphs, storing values of pre-defined data types.

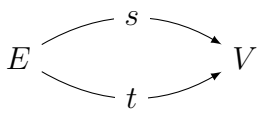
Graph Transformation Systems (GTS) have been used to model the dynamic behaviour of systems where graphs model the systems' states and their evolution is specified by graph transformation rules [32]. The conceptual (type) level of the system is represented by a type graph (model) and its instance level is represented by an instance graph. A type graph is usually visualised using a class diagram in Unified Modelling Language (UML). An instance graph is visualised by an object diagram. Graph transformation rules describe pre and post conditions of operations.

In our research, we use a type graph to describe the DNS Dependency Model. The graph transformation rules are used to suggest correction mechanisms in the form of refactorings to remedy for operational bad smells identified and detected in the instances of the model (i.e. Dependency Graphs). In the following sections, we provide fundamental definitions and basic concepts of graphs and graph transformations.

2.3.1 Basic Concepts

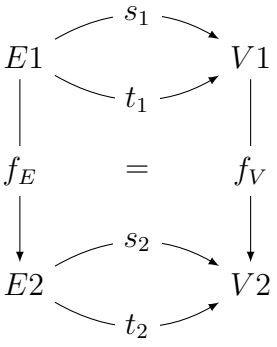
Definition 2.1. (**Graph** [33]) A graph $G = (V, E, s, t)$ consists of a set V of nodes (also called vertices), a set E of edges, and two functions $s, t : E \to V$, the source

and target functions:



A graph homomorphism is a mapping between two graphs that respects their structure. More concretely it maps adjacent vertices to adjacent vertices.

Definition 2.2. (**Graph Morphism** [33]) Given graphs G_1 , G_2 with $G_i = (V_i, E_i, s_i, t_i)$ for i = 1,2, a graph morphism $f : G_1 \to G_2$, $f = (f_V, f_E)$ consists of two functions $f_V : V_1 \to V_2$ and $f_E : E_1 \to E_2$ that preserve the source and target functions, i.e. $f_V \circ s_1 = s_2 \circ f_E$ and $f_V \circ t_1 = t_2 \circ f_E$:



A *graph morphism* f is injective (or surjective) if both functions f_V , f_E are injective (or surjective, respectively); f is called isomorphic if it is bijective, which means both injective and surjective.

In this algebraic representation, a graph is considered as a two sorted *algebra* where the sets of vertices V and edges E are the carriers, while the source $s : E \to V$ and target $t : E \to V$ are two unary operators [34]. The composition property of graph morphisms is one of the necessary ingredients to show that graphs form a category (see Corollary 2.6).

Fact 2.3. *(**Composition of Graph Morphisms** [33]) Given two graph morphisms* $f = (f_V, f_E) : G_1 \to G_2$ *and* $g = (g_V, g_E) : G_2 \to G_3$*, the composition* $g \ o \ f = (g_V \ o \ f_V, g_E \ o \ f_E) : G_1 \to G_3$ *is again a graph morphism.*

The model can be conveniently expressed as a type graph. A typed graph consists of a graph and a corresponding type graph. The type graph defines a set of types that are assigned to the nodes and edges of the graph by a typing morphism.

Definition 2.4. (**Typed Graph** [33]) A *type graph* is a distinguished graph $TG = (V_{TG}, E_{TG}, s_{TG}, t_{TG})$ where V_{TG} and E_{TG} are called the vertex and the edge type alphabets, respectively.
A tuple $(G, type)$ of a graph G together with a graph morphism type: $G \to TG$ is then called a typed graph over TG.

Definition 2.5. (**Typed Graph Morphism** [33]) Given typed graphs $G_1^T = (G_1, type_1)$ and $G_2^T = (G_2, type_2)$, a *typed graph morphism* $f : G_1^T \to G_2^T$ is a graph morphism $f : G_1 \to G_2$ such that $type_2 \ o \ f = type_1$:

$$\begin{array}{ccc} G_1 & \xrightarrow{\quad f \quad} & G_2 \\ & {\scriptstyle type_1} \searrow \quad = \quad \swarrow {\scriptstyle type_2} & \\ & TG & \end{array}$$

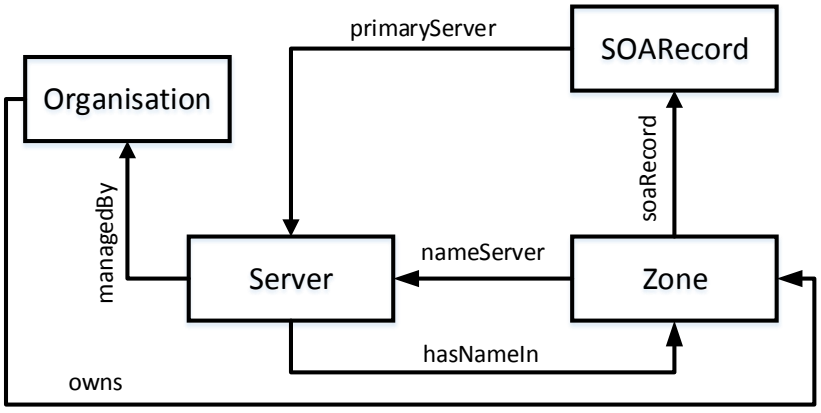


FIGURE 2.4: Typed Graph Example

Figure 2.4 shows an example of a graph with node and edge labels. A graph morphism from G_1 to G_2 is illustrated in Figure 2.5. The dashed vertical arrows represent the morphism's node and edge mapping components. This morphism is injective but not surjective.

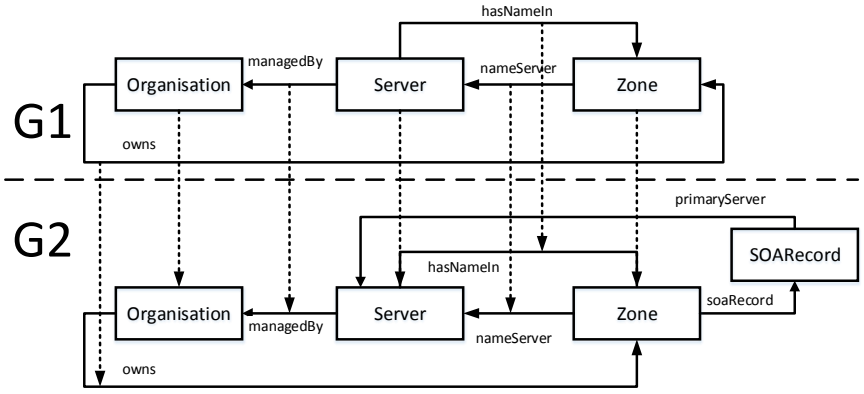


FIGURE 2.5: A Graph Morphism From G1 to G2

In order to use categorical constructs on graphs, it is necessary to show that graphs form a category.

Corollary 2.6. *(**Category of Graphs** [35])*

- *The class of all graphs (as defined in Definition 2.1) as objects and of all graph morphisms (see Definition 2.2) forms the category* ***Graphs****, with the composition given in Fact 2.3, and the identities are the pairwise identities on nodes and edges.*
- *Given a type graph TG, typed graphs over TG and typed graph morphisms (see Definition 2.5) form the category* $\boldsymbol{Graphs_{TG}}$.

Definition 2.7. (**E-graph and E-graph Morphism** [33]) An E-graph G with $G = (V_G, V_D, E_G, E_{NA}, E_{EA}, (source_j, target_j)_{j \in \{G,NA,EA\}})$ consists of the sets:

- V_G and V_D, called the graph and data nodes (or vertices), respectively;

- E_G, E_{NA}, and E_{EA} called the graph, node attribute, and edge attribute edges, respectively; and the source and target functions:
- $source_G : E_G \to V_G$, $target_G : E_G \to V_G$ for graph edges;
- $source_{NA} : E_{NA} \to V_G$, $target_{NA} : E_{NA} \to V_D$ for node attribute edges; and
- $source_{EA} : E_{EA} \to E_G$, $target_{EA} : E_{EA} \to V_D$ for edge attribute edges

Consider the E-graphs G^1 and G^2 with $G^k = (V_G^k, V_G^k, E_G^k, E_G^k, E_G^k, (source_j^k, target_j^k)_{j\in\{G,NA,EA\}})$ for $k = 1, 2$. An E-graph morphism $f : G1 \to G2$ is a tuple $(f_{V_G}, f_{V_D}, f_{E_G}, f_{E_{NA}}, f_{E_{EA}})$ with $f_{V_i} : V_i^1 \to V_i^2$ and $f_{E_j} : E_j^1 \to E_j^2$ for $i \in G, D$, $j \in G, NA, EA$ such that f commutes with all source and target functions, for example f_{V_G} o $source_G^1 = source_G^2$ o f_{E_G} .

Graph transformation has been used as a meta-language to specify and implement visual modelling techniques, like the UML [33]. In most visual modelling techniques, (typed) attributed graphs are used as a representation mechanism [36]. An attributed graph can be seen as a graph where attributes are assigned for the nodes and edges [37]. Several different concepts for typed and attributed graph transformation have been proposed (e.g. [33, 37]). These approaches followed the algebraic approach to provide formal definitions of attributed graph transformation. In [33], the authors introduced a new concept, which is called, *E-graphs*, which allows both node and edge attributions.

Definition 2.8. (**Attributed Graph and Attributed Graph Morphism** [33]) Let $DSIG = (S_D, OP_D)$ be a data signature with attribute value sorts $S'_D \subseteq S_D$. An attributed graph $AG = (G, D)$ consists of an E-graph G together with a DSIG-algebra D such that $\dot{\cup}_{s\in S'_D} D_s = V_D$.

For two attributed graphs $AG^1 = (G^1, D^1)$ and $AG^2 = (G^2, D^2)$, an attributed

graph morphism $f : AG^1 \to AG^2$ is a pair $f = (f_G, f_D)$ with an E-graph morphism $f_G : G^1 \to G^2$ and an algebra homomorphism $f_D : D^1 \to D^2$

Definition 2.9. (**Typed Attributed Graph and Typed Attributed Graph Morphism** [33]) Given a data signature $DSIG$, an attributed type graph is an attributed graph $ATG = (TG, Z)$, where Z is the final DSIG-algebra. A typed attributed graph (AG, t) over ATG consists of an attributed graph AG together with an attributed graph morphism $t : AG \to ATG$.
A typed attributed graph morphism $f : (AG^1, t^1) \to (AG^2, t^2)$ is an attributed graph morphism $f : AG^1 \to AG^2$ such that $t^2 \; o \; f = t^1$.

2.3.2 GT-Based Model Transformation

After having defined the objects of transformation as instances of type graphs satisfying constraints, model transformations can be specified in terms of graph transformation. Formally, meta models are type graphs whose instance graphs represent models. That means, the type-instance mapping of typed graphs, which has so far been used to model the relation of objects to their classes and component instances to their components, shall now be reserved for the mapping between a model and its meta model. Therefore, the object-class and component instance component mappings are defined in the meta model itself [32].

Definition 2.10. (**Graph Transformation System** [33]) A typed graph transformation system $GTS = (TG, P)$ consists of a type graph TG and a set of typed graph productions P.

A typed graph grammar $GG = (GTS, S)$ consists of a typed graph transformation system GTS and a typed start graph S. We may use the abbreviation GT system for typed graph transformation system.

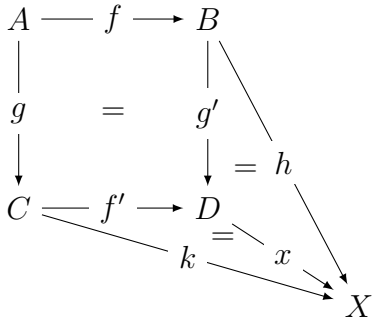
The Algebraic/Double Pushout Approach

Various graph transformation approaches have been developed. A general approach is called the algebraic approach, where an entire sub-graph can be replaced by a new sub-graph. The algebraic approach is based on pushout constructions in the category Graphs of graphs. Pushouts are used to model the gluing of graphs, which is required to apply graph transformation rules to graphs.

Definition 2.11. (**Pushout** [33]) Given morphisms $f : A \to B$ and $g : A \to C$ in a category C, a pushout (D, f', g') over f and g is defined by:

- a pushout object D and
- morphisms $f' : C \to D$ and $g' : B \to D$ with $f' \; o \; g = g' \; o \; f$

such that the following universal property is fulfilled: for all objects X and morphisms $h : B \to X$ and $k : C \to X$ with $k \; o \; g = h \; o \; f$, there is a unique morphism $x : D \to X$ such that $x \; o \; g' = h$ and $x \; o \; f' = k$:



We write $D = B +_A C$ for the pushout object D, where D is called the gluing of B and C via A.

The core of a graph transformation is a graph production $p : L \to R$ consisting of a pair of graphs L and R. L is called the left-hand side graph (LHS) and R is called

the right-hand side graph (RHS). Applying rule p to a source graph means finding a match of L in the source graph and replacing it with R, thus creating the target graph. In the DPO approach, a graph K is used. K is the common interface of L and R, i.e. their intersection. Hence, a rule is given by a span $p : L \leftarrow K \rightarrow R$.

Definition 2.12. (**Graph Production** [33]) A (typed) graph production $p = (L \xleftarrow{l} K \xrightarrow{r} R)$ consists of (typed) graphs L, K, and R, called the left-hand side, gluing graph, and the right-hand side respectively, and two injective (typed) graph morphisms l and r. Given a (typed) graph production p, the inverse production is defined by $p^{-1} = (R \xleftarrow{r} K \xrightarrow{l} L)$.

A graph transformation starts by finding a match m of L in the source graph G. Then, $m(L \backslash l(K))$ are removed from G to create an intermediate graph D. The match m has to satisfy the gluing condition (see Definition 2.14). The graph $D = (G \backslash m(L)) \cup m(l(K))$ is obtained by removing the vertices and edges of L from G that are not in the image l. In the second step, a target graph H is produced by gluing $R \backslash l(K)$ and D; that is, a pushout $D \xleftarrow{k} K \xrightarrow{r} R$.

Definition 2.13. (**Graph Transformation** [33]) Given a (typed) graph production $p = (L \xleftarrow{l} K \xrightarrow{r} R)$ and a (typed) graph G with a (typed) graph morphism $m : L \rightarrow G$, called the match, a direct (typed) graph transformation $G \overset{p,m}{\Rightarrow} H$ from G to a (typed) graph H is given by the following double-pushout (DPO) diagram, where (1) and (2) are pushouts in the category $\boldsymbol{Graphs}$ (or $\boldsymbol{Graphs_{TG}}$, respectively):

$$\begin{array}{ccccccc} L & \longleftarrow & l & \longrightarrow & K & \longrightarrow & R \\ \downarrow & & & & \downarrow & & \downarrow \\ m & & (1) & & k & & (2) \quad n \\ \downarrow & & & & \downarrow & & \downarrow \\ G & \longleftarrow & f & \longrightarrow & D & \longrightarrow & H \end{array}$$

A sequence $G_0 \Longrightarrow G_1 \Longrightarrow \ldots \Longrightarrow G_n$ of direct (typed) graph transformations is called a (typed) graph transformation and is denoted by $G_0 \overset{*}{\Longrightarrow} G_n$. For $n = 0$,

we have the identity (typed) graph transformation $G_0 \overset{id}{\Longrightarrow} G_0$. Moreover, for $n = 0$ we allow also graph isomorphisms $G_0 \cong G_0'$, because pushouts and hence also direct graph transformations are only unique up to isomorphism.

The gluing condition is a constructive approach to formulate a syntactic criterion for the applicability of a (typed) graph production.

Definition 2.14. (**Gluing Condition** [33]) Given a (typed) graph production $p = (L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R)$, a (typed) graph G, and a match $m : L \rightarrow G$ with $X = (V_X, E_X, s_X, t_X)$ for all $X \in L, K, R, G$, we can state the following definitions:

- The gluing points GP are those nodes and edges in L that are not deleted by p, i.e. $GP = l_V(V_K) \cup l_E(E_K) = l(K)$.
- The identification points IP are those nodes and edges in L that are identified by m, i.e. $IP = \{v \in V_L | \exists w \in V_L, w \neq v : m_V(v) = m_V(w)\ \} \cup \{\ e \in E_L | \exists f \in E_L, f \neq e : m_E(e) = m_E(f)\ \}$.
- The dangling points DP are those nodes in L whose images under m are the source or target of an edge in G that does not belong to $m(L)$, i.e. $DP = \{v \in V_L | \exists e \in E_G \backslash m_E(E_L) : s_G(e) = m_V(v) \text{ or } t_G(e) = m_V(v)\}$.

Production p with match m satisfy the gluing condition if all identification points and all dangling points are also gluing points, ie. $IP \cup DP \subseteq GP$.
A graph transformation is a sequence of productions applied to a graph. A set of production rules that may applied to a graph is defined as a graph transformation system. A graph grammar is basically a graph transformation system with a fixed start graph.

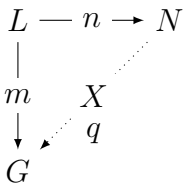
Two direct graph transformations $G \overset{p_1,m_1}{\Longrightarrow} H_1$ and $G \overset{p_2,m_2}{\Longrightarrow} H2$ are *in conflict if they are not parallel independent.* This type of conflict is called *delete-use conflict.*

A critical pair characterises the conflict situation in a minimal context.

Definition 2.15. (**Critical Pair** [33]) A critical pair for the pair of rules (p_1, p_2) is a pair of direct graph transformations $P_1 \overset{p_1,o_1}{\Longleftarrow} K \overset{p_2,o_2}{\Longrightarrow} P_2$ in conflict, such that o_1 and o_2 are jointly surjective morphisms. The context is minimal, because o_1 and o_2 are required to be jointly surjective morphism. This means that each item in K has a pre-image in L_1 or L_2, thus K can be considered as a suitable gluing of L_1 and L_2. If GTS does not contain critical pairs, it is locally confluent.

Negative Application Conditions (NACs) allow control over the applicability of rules in a Graph Transformation System. A NAC is connected to either the LHS or RHS of a production rule forming a pre or postcondition on the rule. If this pattern is found in the corresponding host graph, the production cannot be applied.

Definition 2.16. (**Negative Application Condition** [33]) A Negative Application Condition or $NAC(n)$ on L is an arbitrary morphism $n : L \to N$. A morphism $g : L \to G$ satisfies $NAC(n)$ on L i.e. $g \models NAC(n)$ if and only if does not exists and injective $q : N \to G$ such that $q \circ n = g$.



A set of $NACs$ on L is denoted by $NACL = NAC(n_i) \mid i \in I$. A morphism $g : L \to G$ satisfies NAC_L if and only if g satisfies all single $NACs$ on L i.e. $g \models NAC(n_i) \forall i \in I$.

Definition 2.17. (**Production Rule with NACs** [33]) A set of $NACs$ $NACL$ $(resp. NACR)$ on $L(resp.R)$ for a rule $p : L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R$ (with injective l and r) is called a left (resp. right) NAC on p. $NAC_p = (NAC_L, NAC_R)$ consisting of a set

of left and a set of right $NACs$ on p is called a set of $NACs$ on p. A rule (p, NAC_p) with $NACs$ is a rule with a set of $NACs$ on p.

2.3.3 Graph Transformations Tools

In this section, we introduce the transformation language and tool environment that we use in our implementation of the *DNS Dependency Model* and its corresponding model transformation techniques.

2.3.3.1 EMF

The Eclipse Modeling Framework (EMF) [38] provides a modeling and code generation framework for Eclipse applications for building tools and other applications based on structured data models. The Essential Meta-Object Facility (EMOF) is an Object Management Group (OMG) standard for model-driven engineering [39]. The type information of sets of instance models is defined in a so-called core model corresponding to metamodel in EMOF. The core or metamodel for core models is the Ecore model.

The EMF model can be seen as a type graph with attribution, inheritance and multiplicities and its instance model can be seen as a typed attributed graph [40].A $Transformation$ consists of a *RuleSet* containing the set of *Rules* for the transformation. Furthermore, it has a link to the core model its instances are typed over. If needed, a start structure can be defined as well to have a fixed starting point for the transformation available. A transformation together with a start structure forms an EMF grammar. An *in-place EMF transformation* is a rule-based modification of an EMF source model resulting in an EMF target model. Both, the EMF source and target models are typed over the same EMF core model which itself is again typed

over Ecore. The transformation rules are typed over the Transformation Model which itself is an instance of Ecore again. In our approach, we use the Henshin transformation tool [41], which has its roots in attributed graph transformations. Henshin offers a formal foundation for validation of EMF model transformations.

An EMF model is a class diagram and can be represented by an attributed type graph with inheritance and containment [42]. Graph transformation rules specify local changes on graphs, so-called graph transformations. A rule consists of a left hand side graph (LHS), a right hand side graph (RHS), as well as a mapping from LHS to RHS. Although the LHS defines the precondition for the transformation, i.e. the pattern to be found in the model, its relation to RHS formulates the actions to be performed. All object nodes and edges which occur in LHS, but not in RHS are deleted, while all elements occurring in the RHS and not in the LHS are newly created. Elements occurring in both LHS and RHS have to be there for the transformation to take place, but are not changed. Moreover, negative application conditions (NACs) can be formulated.

A NAC consists of an extension of the LHS where the structure not being part of the LHS is prohibited to occur in the model. Another (implicit) application condition for graph transformation rules is the so-called dangling condition which allows the application of a rule only if adjacent edges of nodes to be deleted occur in the LHS, thus are also scheduled for deletion. Moreover in rule graphs, abstract nodes (typed over abstract node types) may occur. When a rule is applied, its abstract nodes in the LHS are mapped to concrete nodes in the instance graph such that each concrete instance node is in the clan of the corresponding mapped abstract node.

2.3.3.2 Henshin Language and Tools

Henshin [41] provides a state-of-the-art model transformation language for the Eclipse Modeling Framework. Henshin supports both direct transformations of EMF single model instances (endogenous transformations), and translation of source model instances into a target language (exogenous transformations). The Henshin transformation language uses pattern-based rules on the lowest level, which can be structured into nested transformation units with well-defined operational semantics. Its transformation rules are supported by powerful application conditions and flexible attribute computations. They can be structured by means of transformation units that can control the order of rule applications. Henshin offers a visual syntax, sophisticated editing functionalities, execution and analysis tools. The Henshin transformation language has its roots in attributed graph transformations, which offer a formal foundation for validation of EMF model transformations. Before defining rules in Henshin, a model/metamodel should be created using the EMF Eclipse plug-in. The rules can be applied to an instance model of the model/metamodel, which can also be created using EMF tools. There are two editors to define model transformations in Henshin: i) a tree-based editor, generated by EMF itself, and ii) a graphical editor, implemented using GMF. The graphical editor shows rules in an integrated manner with the pattern to find (left-hand side, LHS), the resulting pattern (right-hand side, RHS) and application conditions. In the top of every rule, its name and parameters are specified. Inside a rule, we create Nodes, Edges and Attributes. The nodes represent the classes of the metamodel and the edges are used to specify the link between nodes. Nodes and edges are annotated with stereotypes (actions). There are a number of actions:

- preserve: the node/edge is preserved during the rule application.
- delete: delete an existing node/edge after the rule application.

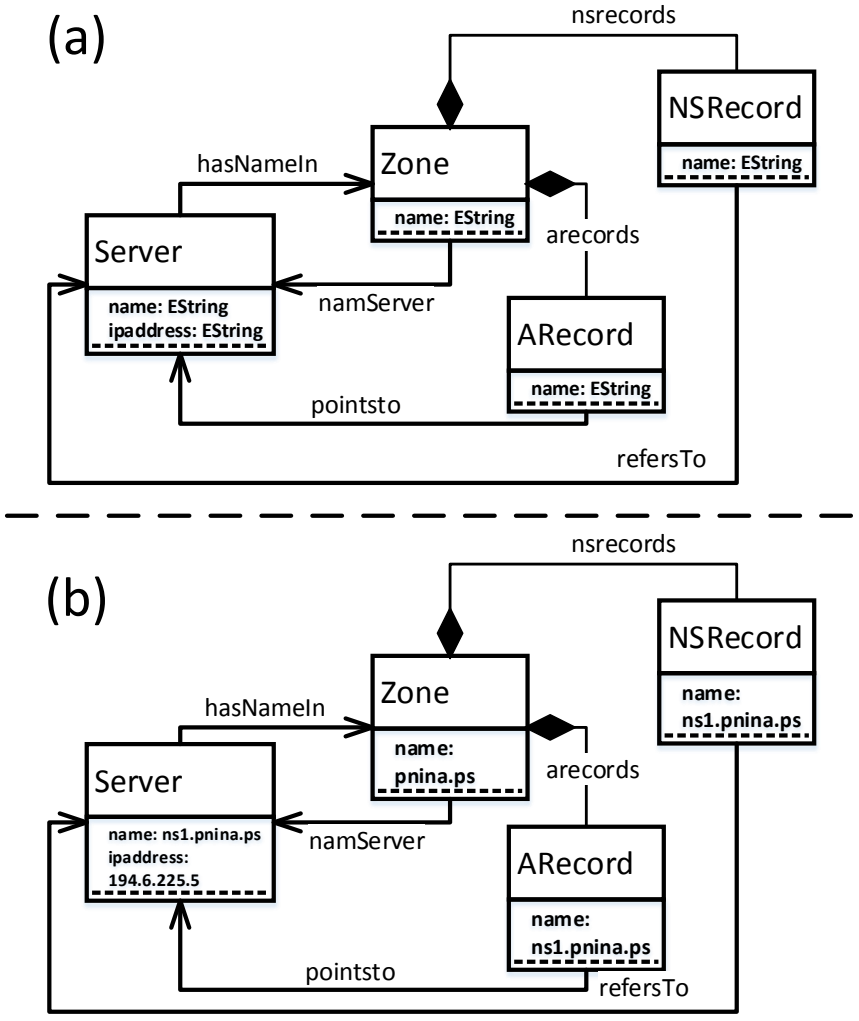


FIGURE 2.6: Attributed Typed Graph Model (a) and Its Instance (b).

- create: create a new node/edge after the rule application.
- forbid: forbid the existence of a node/edge during the rule application.

Figure 2.7 illustrates how the rule *createARecord* would be represented in Henshin. In this example, we show how a new component of type *ARecord* can be added to the model instance with the *zone* as the context where the rule will be applied and *servername* as input parameters for the rule to indicate the data elements within this record.

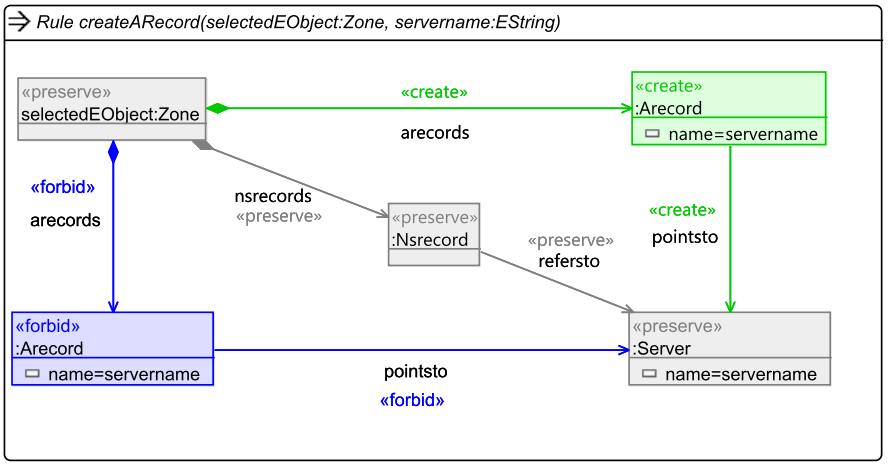


FIGURE 2.7: *createARecord* Transformation Rule

2.3.3.3 EMF Refactor

EMF Refactor [1] is an existing Eclipse project which can calculate metrics and perform refactorings on Ecore and UML models. In particular, EMF Refactor supports metrics reporting, smell detection, and refactoring for models being based on the Eclipse Modeling Framework. The following techniques can be used in a concrete specification of a new EMF model metric, smell, or refactoring:

- Model metrics can be concretely specified in Java, as OCL expressions, by Henshin pattern rules, or as a combination of existing metrics using a binary operator.
- Model smells can be concretely specified in Java, by Henshin pattern rules, or as a combination of an existing metric and a comparator like greater than (>).
- The three parts of a model refactoring can be concretely specified in Java, in Henshin (pattern rules for precondition checks; transformations for the proper model change), or as a combination of existing refactorings using the CoMReL language.

2.4 Model Transformation

Model-driven engineering (MDE) is a discipline in software engineering that relies on models as first class entities and that aims to develop, maintain and evolve software by performing model transformations. Models are system abstractions that allow developers and other stakeholders to effectively address concerns, such as answering a question about the system or effecting a change [43].

A model is useful if it helps to gain a better understanding of the system. In an engineering context, a model is useful if it helps in deciding upon the appropriate actions that need to be taken to reach and maintain the system's goal. The goal of software is to automate some tasks in the real world. Models of software requirements, structure and behaviour at different levels of abstraction help all stakeholders deciding how this goal should be accomplished and maintained [44].

Model manipulation is a central activity in many model-based software engineering activities [45] like, model translations (e.g., translating a UML class model into an ER model), model augmentations (e.g., weaving aspects into a UML class model), and model alignments (e.g., mapping a content model to its GUI view), to mention just a few. An important question concerns the source and target artifacts of the model transformation. If these artifacts are programs (i.e., source code, bytecode, or machine code), one uses the term program transformation. If the software artifacts are models, we use the term model transformation [44].

Model manipulations are usually implemented by means of model-to-model (M2M) transformations. A M2M transformation transforms a model M_a conforming to a metamodel MM_a into a model M_b conforming to a metamodel MM_b (where MM_a and MM_b can be the same or different metamodels). In particular, inspecting and

modifying models to reduce their complexity and improve their readability, maintainability and extensibility (i.e. by performing model refactoring [46]) are important issues of model development.

In order to transform models, these models need to be expressed in some modelling language (e.g., UML for design models, and programming languages for source code models). The syntax and semantics of the modelling language itself is expressed by a metamodel.

Models are usually defined using Domain-Specific Modelling Languages (DSMLs)which are themselves specified through a meta-model. A domain-specific language (DSL) is a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain [47]. The key characteristic of DSLs is their focussed expressive power. A DSML should contain useful, appropriate primitives and abstractions for a particular application domain. Domain-specific modelling languages (DSMLs) successfully separate the conceptual and technical design of a software system by modelling requirements in the DSML and adding technical elements by appropriate generator technology.

The Eclipse Modeling Framework (EMF) [38] has evolved to a de facto standard technology to define models and modelling languages. EMF provides a modeling and code generation framework for Eclipse applications based on the structured data models. The modelling approach is similar to that of MOF (Meta-Object Facility) which is the Object Management Group (OMG) standard for model-driven engineering. EMF supports essential MOF (EMOF) as part of the OMG MOF 2.0 specification [48]. Containment relations, *i.e. compositions* in UML, define an ownership relation between objects. In MOF and EMF, the hierarchical containment structure is used to implement a mapping to XML, known as XMI (XML Meta data interchange) [49].

EMF instance model is called *rooted* if there is one container which contains all other elements transitively. Although EMF instance models do not need to be rooted in general, this property is important for storing them, or more general, to define the model's extent. EMF instance models can be represented as graphs and EMF model transformations as graph transformations.

Chapter 3

DNS Dependency Model

Model-driven engineering (MDE) [50] is a software engineering discipline that uses models as the primary artifacts throughout software development processes and adopt model transformation both for their optimization as well as for model and code generation. A model is a simplified abstract view of the complex reality. Models provide abstractions, which allow developers to focus on the relevant properties of the system, and ignore unnecessary complications [51]. A model has an abstract and a concrete syntax. The abstract syntax is often defined in terms of a metamodel, which is an explicit model of the constructs and well-formedness rules needed to build specific models within a domain of interest. The concrete syntax is the (graphical or textual) representation of the model.

Although it would be ideal to represent the system with one concise model, a system description requires multiple views: each view represents a projection of the complete system that shows a particular aspect. A view requires a number of diagrams that visualise the information of that particular aspect of the system. For example, the concepts used in the object oriented software system diagrams are model elements that represent common object oriented concepts such as classes,

objects and messages, and their relationships, including associations, dependencies and generalisation [52].

The outline of the chapter is as follows. Section 3.1 gives an overview of the basic MDE concepts and definitions used throughout this chapter and beyond. Then, the approach used in modelling the domain name system (DNS) is illustrated in Section 3.2. A diagrammatic representation for the DNS Model and detailed presentation of the model components, attributes, associations and constraints are presented in Section 3.1. Model validation in terms of the 6C goal quality model [53] is presented in Section 3.4.

3.1 Basic Concepts

We start by looking at the concepts that are at the core of MDE that will be used throughout this chapter.

- **System:** A *system* may include anything: a program, a single computer system, some combination of parts of different systems, a federation of systems, each under separate control, people, an enterprise, a federation of enterprises ... etc.
- **Model:** A *model* of a system is a description or specification of that system and its environment for some certain purpose. In [54], Warmer and his colleagues state: "A model is a description of a (part of) systems written in a well-defined language. A well-defined language is a language with well-defined form (syntax), and meaning (semantics), which is suitable for automated interpretation by a computer". A *model* is often presented as a combination of drawings and text.

- **Model-Driven-Engineering**: *Model-Driven-Engineering (MDE)* is an approach to system development which increases the power of models that work. It is *model-driven* because it provides a means for using models to direct the course of understanding, design, construction, deployment, operation, maintenance and modification.

- **Viewpoint** A *viewpoint* of a system is a technique of abstraction using a selected set of concepts and structuring rules in order to focus on particular concerns within that system. Here "abstraction" is used to mean the process of suppressing selected details to establish a simplified model.

- **Model-Transformation** *Model transformation* is the process of converting one model to another model of the same system.

- **Implementation** An *implementation* is a specification, which provides all the information needed to construct a system to put it into operation.

3.2 Modelling the DNS

Efforts to improve risk management related to DNS availability, security, stability and resilience must be guided by an ability to evaluate these characteristics. We need to avoid the implications of misconfigurations and bad deployment choices made by system administrators that may lead to data inconsistencies, vulnerable configurations or even failure of resolution at an early stage of the design/deployment of the DNS.

The *DNS Dependency Model* is an attempt to describe the Domain Name System (system) operational world for a particular operational goal (purpose) of detecting violations of the design and deployment principles at the authoritative level (view). For detecting problems in the configuration and deployment of the DNS, we have to

search for certain patterns representing those problems in the instances of the model of the system (i.e., the dependency graphs). This means we have to be able to specify a problem and to query the model instance about the existence and occurrences of the specified problem.

From an operational perspective, there are three views of the domain name system:

- The *user's point of view*: from this view, the domain system is accessed through a simple procedure or OS call to a local resolver. The domain space consists of a single tree and the user can request information from any section of the tree.
- From the *resolver's point of view*, the domain system is composed of an unknown number of name servers. Each name server has one or more pieces of the whole domain tree's data, but the resolver views each of these databases as essentially static.
- From a *zone administrator's point of view*, the domain system consists of separate sets of local information called *zones*. The name server has local copies of data related to some of the zones. The name server must periodically refresh its zones' data from master copies in local files or foreign name servers managed by external organisations. The name server must concurrently process queries that arrive from users through external DNS resolvers. This is the authoritative level view that we are concerned in modelling the DNS from throughout this study.

From the perspective of the authoritative zone administrator, the DNS [22] has three major components:

- The *DOMAIN NAME SPACE* and *RESOURCE RECORDS*, which are specifications for a tree structured name space and data associated with the names.

Conceptually, each node and leaf of the domain name space tree names a set of information, and query operations are attempts to extract specific types of information from a particular set. A query names the domain name of interest from a particular part of the tree and describes the type of resource information that is desired. For example, the Internet uses some of its domain names to identify hosts; queries for address resources, return Internet host addresses (IP addresses). These components are modelled as part of the *DataLayer* within the *DNS model*.

- *NAME SERVERS* are physical or logical hosts with server programs which hold information about the domain tree's structure of a particular zone information. A name server may cache structure of a zone information or about any part of the domain tree, but in general a particular name server has complete information about a subset of the domain space, and pointers to other name servers that can be used to lead to information from any part of the domain tree. Name servers know the parts of the domain tree for which they have complete information; a name server is said to be an *AUTHORITY* for these parts of the name space. Authoritative information is organized into units called *ZONEs*, and these zones can be automatically distributed to the name servers which provide redundant service for the data in a zone. This component with its attributes and associations is modelled as part of the *ControlLayer* within the *DNS model*.

- *HOSTING ORGANISATIONS* are entities that are responsible for providing DNS hosting services. The Domain Name System requires that multiple servers exist for every delegated (*zone*). These servers are hosted and managed by providers who have multiple servers in various geographic locations that provide resilience and minimize latency for clients around the world. By operating DNS nodes closer to end users, DNS queries travel a much shorter

distance, resulting in faster Web address resolution speed. The zone manager coordinates with other peer organisations or commercial DNS hosting companies to provide secondary DNS hosting for their zone's data through different types of mutual agreements. This component with its attributes and associations is modelled as part of the *ManagementLayer* within the *DNS model*.

3.3 The DNS Dependency Model

Throughout this section, we explain how we constructed the *DNS Dependency Model* as a unified representation of DNS Dependency Graphs. We are modelling the DNS system from the prospective (view) of authoritative system administrators and zone managers. To develop the model, we use Eclipse Modelling Framework (EMF) [38] as the modelling language for our application domain. EMF (core) is a common standard for data models, many technologies and frameworks are based on. This includes server solutions, persistence frameworks, UI frameworks and support for graph-based transformation tools. The *DNS Dependency Model* is composed of the following elements:

- **Operational Entities** (e.g. resource records, zones, servers and organizations).
- **Properties** of operational entities such as (in-bailiwick which are name servers with names registered within the same zone and out-of-bailiwick name servers which are servers with names registered within third party zones).
- **Relations** between the entities (e.g. access attributes such as dependability, containment, delegation and management).

The operational DNS entities that appear in our model as shown in Figure 3.1 fall into two categories: primitive and composed entities. Composed entities have an

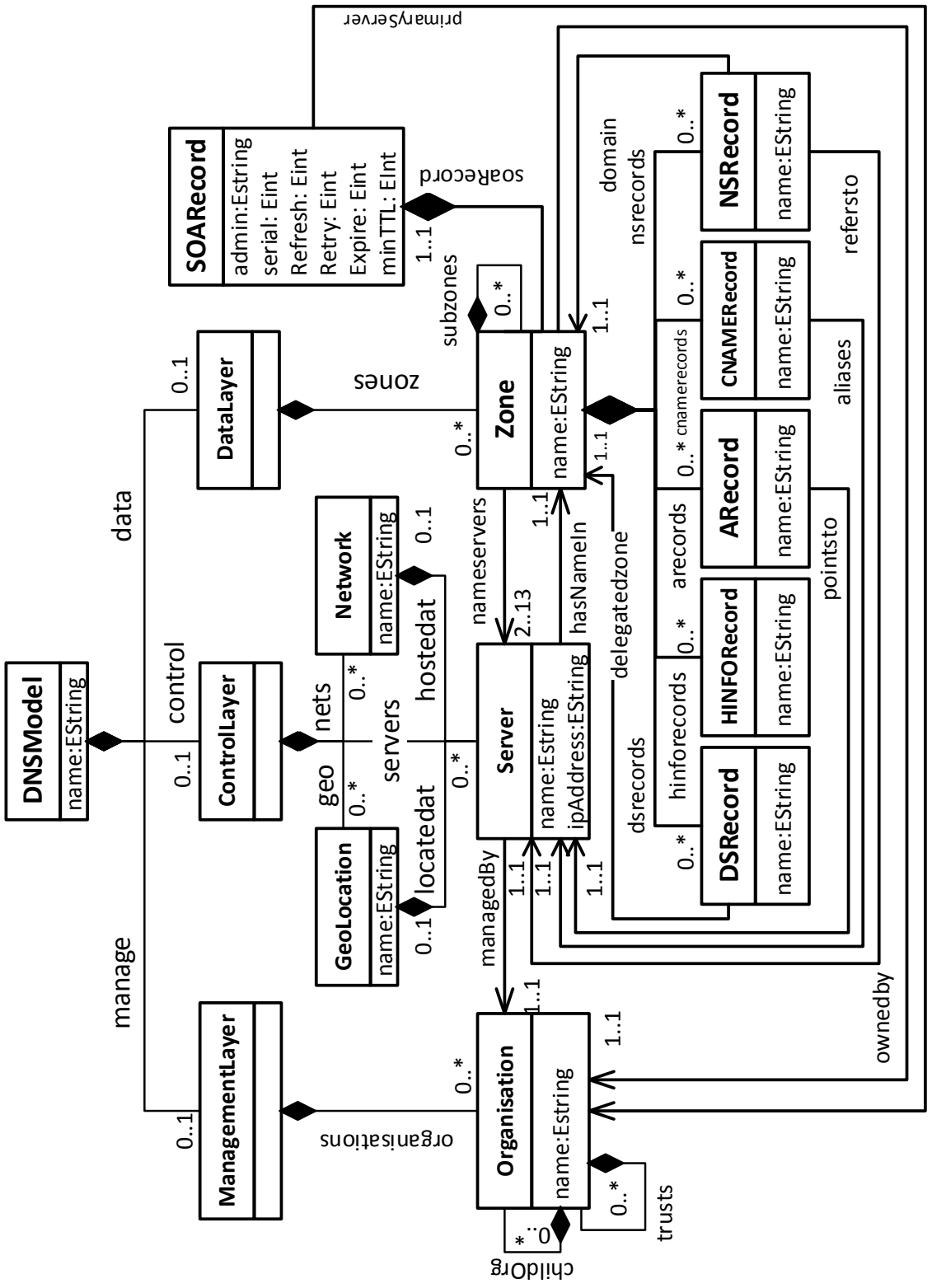


FIGURE 3.1: The DNS Dependency Model Specified in Ecore (The Meta-Meta Model of EMF).

identity and a set of properties. In addition to these, composed entities have a list of contained entities, which are primitive or composed entities. A composed entity type is one that contains other entities. The model supports the following composed entities: Organization, Server and Zone. In order to describe a composed entity we have to specify its properties, containment structure (i.e. the entities that it contains), relations and container entity. Three specific dependencies are present within the DNS operational planes and they are the following:

- **Parent Dependency:** resolving the name of a domain name is always dependent on resolving its parent name since the resolver must learn the authoritative servers for a zone from referrals from the zone's hierarchical parent.
- **Authoritative Name Server (NS) Dependency:** A zone is said to depend on a name server if the name server could be involved in the resolution of names in that zone.
- **CNAME Aliasing Dependency:** the resolution of an alias is always dependent on the resolution of its target CNAME. If a resolver receives a response indicating that the name in question is an alias to another name, it must subsequently resolve the target of the alias, and so on until an address is returned.

Although EMF models show a graph-like structure and can be transformed similarly to graphs [33], there is a an important difference. In contrast to conventional graphs, EMF models have a distinguished tree structure which is defined by the containment relation between their classes. An EMF model should be defined such that all its classes are transitively contained in the root class. Complete details of the DNS Model is included in Appendix A.

3.4 DNS Model Quality

In their article, Mohagheghi et al. [53] present the results of a systematic review of literature discussing model quality in model-based software development. From 40 studies covered in the review, the authors identified six classes of quality goals, called 6C goals, in model-based software development. They state that other quality goals discussed in literature can be satisfied if the 6C goals are in place. Here we shortly introduce the identified 6C goals.

- *Correctness:* A model is correct if it includes the right elements and correct relations between them and if it includes correct statements about the domain. Furthermore, a model must not violate rules and conventions. This definition includes syntactic correctness relative to the modelling language as well as semantic correctness related to the understanding of the domain.

- *Completeness:* A model is complete if it has all necessary information that is relevant, and if it is detailed enough according to the purpose of modelling. For example, requirement models are said to be complete when they specify all the black-box behaviour of the modelled entity, and when they do not include anything that is not in the real world.

- *Consistency:* A model is consistent if there are no contradictions within. This definition covers horizontal consistency concerning models/diagrams on the same level of abstraction, vertical consistency concerning modelled aspects on different levels of abstraction as well as semantic consistency concerning the meaning of the same element in different models or diagrams.

- *Comprehensibility:* A model is comprehensible if it is understandable by the intended users, either human users or tools. In most of the literature, the focus is on comprehensibility by humans including aspects like aesthetics of

a diagram, model simplicity or complexity, and the use of the correct type of diagram for the intended audience. Several authors also call this goals pragmatic quality.

- *Confinement:* A model is confined if it agrees with the modelling purpose and the type of system. This definition also includes relevant diagrams on the right abstraction level. Furthermore, a confined model does not have unnecessary information and is not more complex or detailed than necessary.

- *Changeability:* A model is changeable if it can be evolved rapidly and continuously. This is important since both the domain and its understanding as well as requirements of the system evolve with time. Furthermore, changeability should be supported by modelling languages and modelling tools as well.

The selection of main quality aspects may vary dependent on the intended modelling purpose and demonstrates the complexities and challenges of this basic task. In this context, the following aspects which are most relevant to our model:

- The most important property of a domain analysis model is that it models the problem domain in the right way, i.e. choosing the right elements and claiming the right statements. So, 6C goal *Correctness* is an essential quality aspect that has to be considered when applying a model quality assurance process. In our model, we included the relevant operational entities within the DNS system from the perspective of authoritative domain managers. Those model elements include the different operational layers and their components *(i.e Resource Records, Zones, Servers, networks, GeoLocations and Organisations)*. We presume that each and every component in the model complies completely with the specifications and operational guidelines of the DNS protocol.

- Since analysis models will be used for communicating with problem domain experts who are typically experts in the operation and management of the

DNS but inexperienced in modelling, it is also important that the model is easily understandable. This implies that the model must not allow different interpretation results. Our model does not have unnecessary information that make it more complex. So, 6C goals *Comprehensibility, Consistency,* and *Confinement* can be seen as essential quality aspects.

- Since the purpose of our model is to detect misconfiguration and bad deployment choices in the model instances from the perspective of domain administrators and zone managers, we include just the elements that represent the main operational components within the DNS system from this perspective and the relationships that reflect the various inter-dependencies within and between these components. In this sense our model satisfies the 6C goal *Completeness.*

- Furthermore, since our model is simple and manageable, the model quality goal *Changeability* is present and new models can be easily generated by applying model transformation techniques on the initial model to improve/optimize the quality attributes of the operational domain names system as defined in Chapter 2.

Chapter 4

DNS Structural Metrics

Measurement plays a critical role in effective and efficient system development and operation. Therefore, we need to be able to provide accurate information, recommendations and guidelines to system designers and managers to help them make informed decisions, plan and allocate resources for the different system configurations and deployment layouts. To compute a metric on a given system, we need to extract a model of the system. This model is extracted and stored based on a meta-model that specifies the relevant entities and their relevant properties and relations.

Throughout this chapter, we utilize a set of *Structural Metrics* defined over the *DNS Dependency Model* as indicators of external quality attributes of the domain name system. We apply some machine learning algorithms in order to construct *Prediction Models* of the perceived quality attributes of the operational system out of the structural metrics of the model and evaluate the accuracy of these models. Assessing the quality attributes of the DNS at an early stage of the design/deployment should enable system administrators to avoid the implications of defective and

low-quality designs and deployment choices and identify configuration changes that might improve the availability, security, stability and resiliency postures of the DNS.

4.1 Definitions and Basic Concepts

A model structural metric or generally any measure is a homomorphism from an empirical relational system to a numerical relational system; therefore, it is imperative that measures be theoretically analysed within the framework of measurement theory. We open this section with a set of general definitions on measurement, and measurement related concepts. The definitions are based on [55]:

Definition 4.1. (**Measurement**) Measurement is defined as the process by which numbers or symbols are assigned to attributes of entities in the real world in such way as to describe them according to clearly defined rules.

This definition of measurement requires some explanations and several further definitions. The concepts used in this definition such as entity or attribute will be defined next.

Definition 4.2. (**Entity**) We define an entity as the subject of the measurement process. An entity might be an object within a model instance, or a system specification or a phase of a project.

Definition 4.3. (**Attribute**) An attribute is a feature or property of the entity. For example, an attribute of a server is its name or IP address, and an attribute of a zone's SOA Record may be its primary server name or the value of its expiry parameter.

Informally, the assignment of numbers and symbols must preserve any intuitive and empirical observations about the attributes and entities.

4.2 DNS structural metrics

To the extent of our knowledge, only very few preliminary studies for defining suitable metrics to measure the quality attributes of the DNS system have been conducted [56], [57], [58]. Even within these existing works, not much theoretical or empirical evaluation of the proposed metrics has been done. An important step towards improved quality assurance of the DNS is a precise quantification of its quality attributes.

In this chapter, we pursues this line of argument by assessing the characteristics of of the DNS based on a limited set of structural metrics of the DNS Dependency Model. The significance of these metrics relies on a thorough empirical validation of their connection with quality attributes. The main idea behind the design of these metrics has been comprehensiveness and simplicity.

We tried to cover as many structural characteristics of the DNS model as possible. To achieve this, metrics proposed in the areas of DNS management [7], [13], [59], object-oriented software design [60], [61], [62], [63] software model design [64], [65] and even business process models [66], [67], [68] have been considered.

In order to offer a systematic approach, we focused our research on four internal characteristics that are essential to DNS interdependencies (i.e. size, structural complexity, dependency and delegation/inheritance) and classified the metrics based on these criteria.

4.2.1 Measures of Size

Applying size metrics at the system level we can get a good overview of the dimensions of the system. Applying this category of metrics at the model component level

we expect, on the one hand, to detect the servers, zones and organisations that play an important role in the system design, and on the other hand, to find the name servers and zones that introduce extremely large or complex dependency chains and third party influence.

Table 4.1: DNS Model Structural Size Metrics.

No.	Metric	Symbol	Explanation
1	Attack Surface	AS(z)	The set of all elements in the zone,s dependency graph.
2	Redundancy	R(z)	Minimum number of name servers that if failed altogether will render the domains under the zone(z) unresolvable.
3	Authoritative Name Servers	NS(z)	Total number of authoritative name servers of the zone(z)
4	Number of Zones	Zones(z)	Total Number of Zones influencing the resolution of domain names under zone(z).
5	Number of Orgs	Org(z)	Total number of organisations within the dependency graph of zone(z).
6	In-Bailiwick Servers	Is(z)	Number of authoritative name servers with names within the zone(z).
7	Out-Of-Bailiwick Servers	Os(z)	Number of authoritative name servers with names outside the zone(z).

The DNS Dependency Model has various levels of abstraction (e.g. system level, zone/server level, resource record level) and size measures can be defined over the system model for each level. At the system level a commonly used metric is the Attack Surface (AS) or total number of elements within the system model. At a lowest abstraction level, the number of zones can be indicative of the influence of zones (direct and third party) on the resolution of domain under a particular zone.

4.2.2 Measures of Structural Complexity

Complexity metrics enable us to make a first assessment of the structural complexity of the given system. One important necessity for DNS proper operation is careful

coordination between zone administrators and system managers hosting the authoritative name servers of the zone. Lack of such coordination can result in increased risk of failure. The coordination spans both hierarchically (i.e., between parent and child zones) and laterally, between organizations hosting each other's zone data (i.e, between name servers operators).

Structural complexity of the model is a reflection of the amount of coordination needed to properly manage a certain zone. There are two metrics used to quantify the structural complexity of a DNS zone. The first metric which measures the lateral complexity of the zone is *Administrative Complexity (AC)* [7] which describes the diversity of a zone, with respect to organizations administering its authoritative servers. The second metric that measures the hierarchical complexity of the zone is the *Hierarchical Reduction Potential (HRP)* [7], which quantifies how much the ancestry of a zone might be reasonably consolidated to reduce hierarchical complexity.

TABLE 4.2: DNS Model Structural Complexity Metrics.

No.	Metric	Symbol	Explanation
8	Administrative Complexity	AC(z)	How many organizations can be involved in managing the authoritative name servers of the zone.
9	Hierarchy Reduction Potential	HRP(z)	How much reduction in hierarchy complexity that can be attained by consolidating the zone records within the parent zone of z.
10	Network Diversity	NetD(z)	Number of distinguished networks' Autonomous System (AS) numbers which host the authoritative name servers of the zone.
11	Geographical Diversity	GeoD(z)	Number of distinguished geographical locations (countries) which host the authoritative name servers of the zone.
12	Controllability	Co(z)	$Co(z) = \frac{Is(z)}{I(z)+Os(z)}$

4.2.3 Measures of Dependency/Influence

An analysis of the dependency paths in in the model instance (i.e. the Dependency Graph) is necessary to determine the level of influence of other zones on the resolution of domain names under the zone in concern. Interesting metrics in this category are the number of direct and third party zones and organisations that influence the resolving of domain names. A cyclic zone dependency occurs when two or more zones depend on each other in a circular way and the metric *Dependency Cyles* is a measure of such configuration.

Table 4.3: DNS Model Structural Dependency/Influence Metrics.

No.	Metric	Symbol	Explanation
13	Influencing Zones	I(z)	The set of all zones in the zone(z) dependency graph.
14	Directly Configured Zones	DCZ(z)	The number of directly configured zones.
15	ThirdPartyZones	TPZ(z)	Number of zones influencing the resolution of zone (z) not explicitly configured by zone(z) administrator.
16	Directly Configured Organisations(z)	DCO(z)	Number of organisations managing zones that are explicitly configured by the current zone(z) administrator.
17	Third Party Organisations(z)	TPO(z)	Number of organisations that manage zones that are not explicitly configured by the current zone(z) administrator.
18	Dependency Cycles	Cycles(z)	Number of dependency cycles along the name servers query chains and forming cyclic query paths.

4.2.4 Measures of Delegation and Inheritance

In the software engineering realm, the need to measure delegation and inheritance structures is emphasized by many researchers [69], [70]. They suggest that the

measurement should refer to the depth and the node density within the system model. Within the DNS realm, resolution of domain names under a certain zone z using the list of name servers configured for that zone, will follow any of the query resolution paths defined by the names of those servers equally since they will be selected, each with equal probability. However, its resolution remains entirely dependent on its parent zone, $Parent(z)$, regardless of which server in the name server list is selected for query and how far such name server is located within the system's model instance.

TABLE 4.4: DNS Model Structural Delegation/Inheritance Metrics.

No.	Metric	Symbol	Explanation
19	Depth	D(z)	How far the current zone from the ROOT or how deep the zone in the DNS tree (Data Plane).
20	Minimum Query Path	MinQP(z)	Number of name servers involved in the resolution of domain names of zone(z) through the shortest query path.
21	Maximum Query Path	MaxQP(z)	Number of name servers involved in the resolution of domain names of zone(z) through the longest query path.
22	Average Query Path Length	AQP(z)	Average number of name servers involved in the resolution of domain names of zone(z) through all query paths.

4.3 Interpretation Model

For the proper interpretation of each structural metric defined over the DNS model, we give the metric definition, context, usability, how to measure , metric range and a formula for computing that metric. The definition of a complex metric might rely on one or more basic metrics. Detailed descriptions of complex metrics as well as a

comprehensive list of basic metrics can be found in the corresponding interpretation models in Appendix B.

Table 4.5 shows the interpretation model for the metric *Administrative Complexity* [7]. In this section, we present the structural metrics defined over the *DNS*

TABLE 4.5: Interpretation of the Administrative Complexity Metric.

Metric	Administrative Complexity.
Definition	Describes the diversity of a zone with respect to the organisations administering its authoritative name servers.
Context	Zone
Usability	The advantage of mutual hosting of zones between organizations is an increased availability but at the same time increased potential of failure and instability of the zone resolution process.
How to Measure	Count the number of servers managed by each distinguished organization within the set of authoritative name server of zone(z).
Metric Notation	O_z: set of organizations administering authoritative name servers hosting zone (z); n: total number of authoritative name servers of zone (z); NS_z^o: the subset of name servers administered by organization o in O_z.
Range	$0 \geq$AC≥ 1
Formula	$Ac(z)$ =1-$\sum_{o=1}^{o=n}$ $(\frac{NS_z^o}{NS_z})^n$.

Dependency Model and computed on a model instance (Dependency Graph). Figure 4.1 shows part of simple Dependency Graph for the *zone(NIC.AA)*. Based on the interpretation model presented in Table 4.5, the *Administrative Complexity* metric is calculated by counting the number of directly configured authoritative name servers of the zone (z) that are managed by the same organisation and apply the corresponding formula.

$$Ac(\text{"}NIC.AA\text{"}) = 1\text{-}\sum_{o=1}^{4} (\tfrac{1}{4})^4 = 0.984375.$$

The value of this metric is high since each authoritative name server of the zone is managed by a totally different organisation so the amount of coordination (or lateral complexity) needed for such a configuration is expected to be very high. Table 4.6

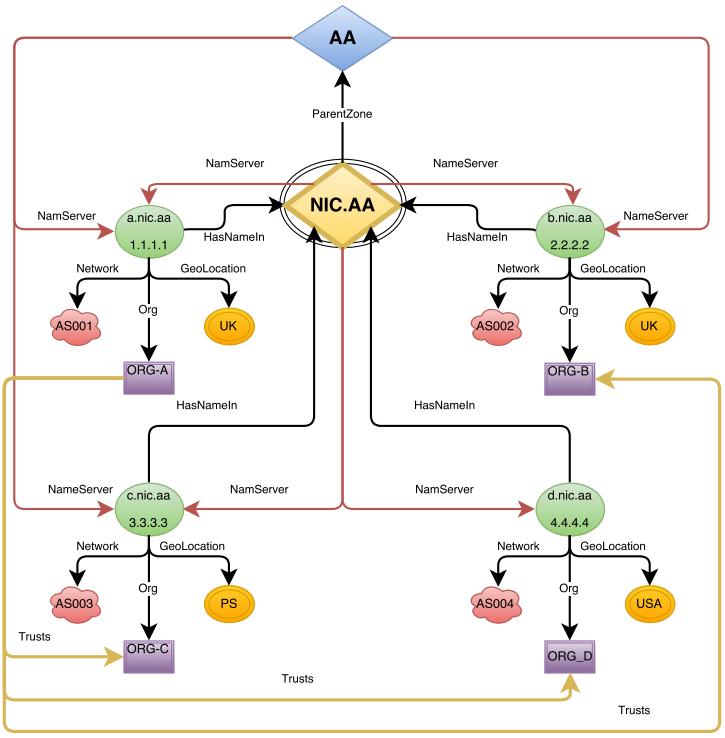


FIGURE 4.1: Example of a DNS Model Instance (i.e. Dependency Graph) for $Zone(NIC.AA)$.

shows the values of structural metrics calculated over the model instance shown in Figure 4.1.

4.4 Theoretical Background

Next we are going to identify the key mechanisms that are involved in the theoretical definition and implementation of structural metrics of the *DNS Dependency Model*.

4.4.1 Key Mechanisms

In general, metrics fall into two big categories: group building and property computing. The former category is mainly used for understanding a system while the

TABLE 4.6: Values of Structural Metrics Calculated over the Model Instance Shown in Figure 4.1.

Metric	Value	Metric	Value
Size Metrics		Dependency Metrics	
AS(z)	9	I(z)	2
R(z)	3	DCZ(z)	2
NS(z)	4	TPZ(z)	0
Zones(z)	2	DCO(z)	4
Org(z)	4	TPO(z)	0
Is(z)	4	Cycles(z)	3
Os(z)	0		
Complexity Metrics		Inheritance Metrics	
Ac(z)	0.98	D(z)	3
HRP(z)	0.5	MaxQP(z)	3
NetD(z)	4	MinQP(z)	3
GeoD(z)	3	AQP(z)	3
Co(z)	1		

latter is used for the assessment of the system. The implementation of each of these requires a particular set of key mechanisms [71].

Group Building Analyses construct collections of model entities that are associated by a particular rule with the analysed entity. Building a group for a model entity requires a set of four elementary mechanisms:

- *Navigation.* All, except trivial metrics, are based on multiple entities so it is necessary to be able to browse through the model, going from the analysed entity to a related entity (e.g., from a zone to its parent zone) or to a group of related entities from the model (e.g., from a zone to the group of its name servers).

- *Selection.* Every model entity is described by various attributes but only some of these are of interest in the context of a particular analysis. Therefore, selection mechanism should enable the definition of a "view of interest" by

choosing only a subset of an entity's attributes (e.g., the IP address of an (anycast) name server from the perspective of the system administrator).

- *Set Arithmetic.* Groups of entities are after all built by means of set arithmetic. The most used set operations in analyses are: the addition of an entity to a group and the union of two or more groups. The *Administrative Complexity* formula shown in Table 4.5 is an example of a structural metric calculated using some set arithmetic operations.

- *Filtering.* An essential mechanism for building a group with a particular property is applying some filtering conditions to an initial larger group. For example, getting the group of name servers that has their names in a certain zone *(In-bailiwick servers, Is(z))* requires first a navigation to the group of name servers associated with a certain zone and then it requires also a filtering operation that builds a new group that keeps only the name server who has its name in that particular zone.

Property Computing Analyses associate a new, non-elementary, property to an entity. Usually, computing a property is preceded by the construction of an appropriate product between two sets is also needed. For example getting all the zones that influence the resolution of domains under another zone is needed in the context of resolution dependency (influence) metrics group. Thus, we may say that in most of the cases property computing analyses imply a group building analysis. Therefore, all key mechanisms identified before are, in principle, needed for a property computing analysis. Additionally, in order to compute a property, usually a numeric or boolean value computed from a group associated with the entity, we need a fifth mechanism:

- Property Aggregation. This mechanism allows us to compute and associate a single value for a group, a value which is aggregated from the values of each

part of the group. Probably the most simple property aggregation is to get the cardinality of a group (used in the computation of most metrics). Depending on the type of properties of the entities belonging to the group more such aggregations can be imagined (e.g., sum or average for numerical properties, logical AND for booleans). An example of such property aggregation is computing the *Average Query Path, AQP(z)* for a zone.

4.4.2 Measurements Frameworks

The DISTANCE measurement framework [72] proposes a set of mandatory properties, i.e., non-negativity, identity, symmetry, and triangular inequality that need to be satisfied by any metric in order for it to be considered an acceptable measurement-theoretic metric. On the other hand, the Property-Based Measurement framework [73] provides a set of desirable properties for different metric types: size (non-negativity, null value, additivity), length (non-negativity, null value, identity, monotonicity), and complexity (non-negativity, identity, symmetry, additivity, monotonicity) and recommends that these properties are satisfied as much as possible.

We investigate to what extent our proposed metrics are able to respect these properties in light of the DISTANCE framework. All of the introduced metrics respect the four mandatory properties required by the DISTANCE framework to form a valid metric space. Therefore, the important consequence of satisfying these four properties is that all of our proposed metrics are theoretically valid DNS model structural metrics. The set of metrics also satisfy most of the recommended and desirable features based on the property-based framework.

4.5 Predictive Models

Predictive models [74] are created to best predict the probability of an outcome based on some prior observations. In the following sections, we apply some machine learning algorithms in order to construct *Prediction Models* of the perceived quality attributes of the operational system out of the structural metrics of the DNS model. Assessing these quality attributes at an early stage of the design/deployment enables us to avoid the implications of defective and low-quality designs and deployment choices and identify configuration changes that might improve the availability, security, stability and resiliency postures of the DNS.

The predictive models are built based on the methodology outlined in Figure 4.2. These models take the structural metrics of a DNS model instance as input and try to find the most relevant value of the quality attribute for the given model.

The DNS quality prediction models are developed based on the following machine learning techniques [75]:

- *Random Forest (RF)*: Random forests or random decision forests are an ensemble learning method for classification, regression and other tasks, that operate by constructing a multitude of decision trees at training time and outputting the class that is the mode of the classes (classification) or mean prediction (regression) of the individual trees.
- *Simple Logistic Regression (SLR)*: SLR is a binary logistic model which is used to estimate the probability of a binary response based on one or more predictor (or independent) variables (features).
- *Locally Weighted Learning (LWL)*: Locally Weighted Learning is a class of function approximation techniques, where a prediction is done by using an approximated local model around the current point of interest.

- *Pruning rule based classification tree (PART)* A Rule-based classifier that makes use of a set of IF-THEN rules for classification. The pruning mechanism's efficiency determines the size and accuracy of the final model.

These machine learning techniques are commonly used in the domain of software engineering for tasks such as software quality analysis and effort prediction. The models take the structural metrics of a DNS model instance as input and try to find the most relevant value of the quality attribute for the given model.

We employ the Waikato Environment for Knowledge Analysis (WEKA) [76], which is a widely used suite of machine learning techniques, to train and test our predictive models. We use two totally independent datasets. One set is used to train the models and build the prediction models and the other is used to test the developed model. For each of the four quality attributes, one instance of each of the mentioned predictive models is developed (4 model types and 4 characteristics = 16 predictive models).

In order to evaluate the accuracy of the developed predictive models, we employ two strategies, namely the percentage of correctly classified instances within the test dataset and the area under the receiver operating characteristic (ROC), or ROC curve. ROC Curve is a graphical plot that illustrates the performance of a binary classifier system as its discrimination threshold is varied. The curve is created by plotting the true positive rate (TPR) against the false positive rate (FPR) at various threshold settings.

4.6 Experimental Assessment

The purpose of this study is to identify any significant relationship between a set of structural metrics defined over a DNS model and the subjective perception of domain

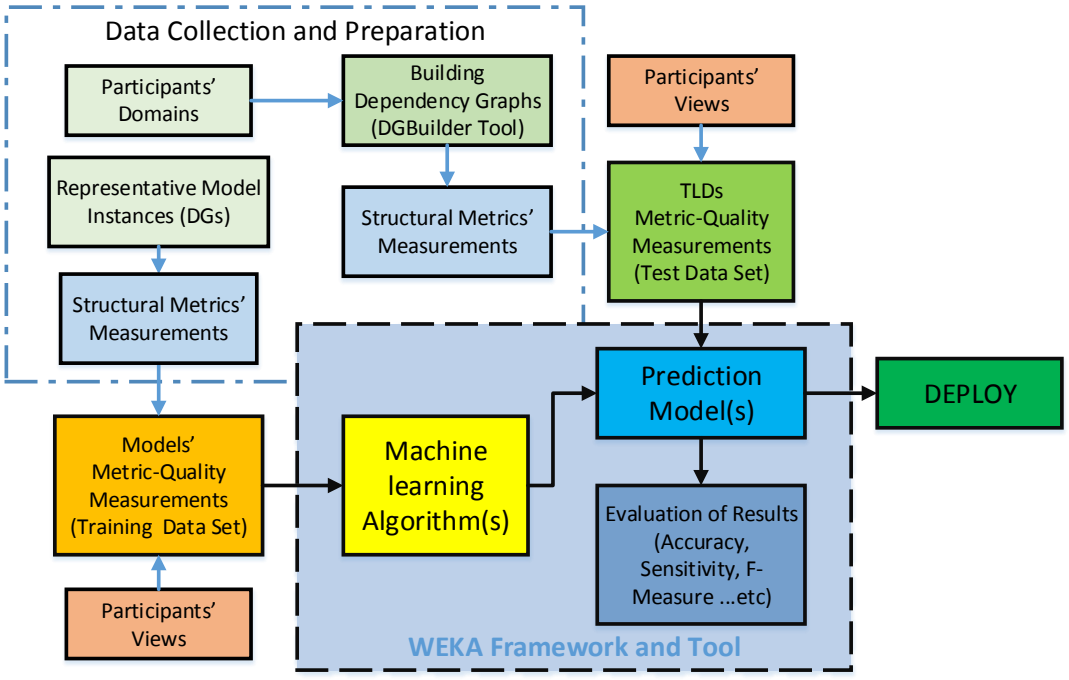


FIGURE 4.2: Methodology of Building DNS Quality Prediction Models

experts of the DNS quality attributes. Another purpose of the study is to evaluate how well different prediction models based on the proposed structural metrics can perform in assessing the perceived quality attributes of the system. These objectives will be achieved by conducting a controlled experimentation and employing a set of statistical analysis techniques.

4.6.1 Hypotheses

- **H1:** Correlations exists between a set of DNS dependency model structural measures and a set of perceived quality attributes of the DNS.
- **H2:** Prediction models built based on the proposed structural metrics of the DNS Model are accurate and effective in predicting the quality attributes of the DNS system.

4.6.2 Variables

In order to proceed with the experiment, the defined hypotheses need to be mapped onto a set of measurable independent and dependent variables. An independent variable is the variable that is changed or controlled in a scientific experiment to test the effects on the dependent variable. These variables are evaluated in the experiment and will be used in the analysis phase.

Independent Variables: Representative set of eleven structural metrics defined over the DNS model as shown in Table 4.7. We selected the eleven metrics out of the *DNS Metrics Suite* with representative metrics from each category.

TABLE 4.7: List of Structural Metrics Used in the Empirical Assessment.

Number	**Measure**	**Symbol**
1	Attack Surface	AS
2	Number of Name servers	ANS
3	Network Diversity	NETD
4	Geographical Diversity	GEOD
5	Redundancy	RED
6	Administrative Complexity	AC
7	Average Query Path	AQP
8	Direct Zones	DCZ
9	Third Party Zones	TPZ
10	Directly Configured Organizations	DCO
11	Third Party Organizations	TPO

To get metrics measurements, we used 10 different model instance of the DNS model and measured those metrics on each of them. We don't have a pre-defined store of such models and have to build them using our DG-Builder tool. We also built the dependency graphs for 15 Top-Level-Domains (TLDs) that are managed by the participants of our experiment. This group of TLDs has a diverse range of dependency graphs from small and compacted ones to large and widely spread ones.

Dependent Variables: Four external quality attributes of the DNS system (i.e. availability, security, stability and resiliency) are considered to be the dependent variables.

4.6.3 Collection of Data

The subjective opinions of the participants about the quality attributes of the DNS system were collected using an online questionnaire. During the period of the survey, the participants had the opportunity to ask questions to the experimenter. The questionnaire consisted of 45 questions divided in 3 sections as follows:

1. Each participant was asked to answer about 10 general questions related to their experience with the DNS system as well as the TLD they are responsible for.

2. Then, the participant was asked to evaluate the perceived quality attributes of a set of 9 dependency graphs presented as instances of the DNS model.

3. Finally, the participants were asked to assess the quality attributes of the TLDs under their own management.

The questions assess the quality attributes of the DNS model by asking the participants to select one of the five linguistic values shown in Table 4.8. The survey included instructions, background information, tips and hints for each question.

TABLE 4.8: Linguistic Values used for the subjective evaluation of DNS qualities.

Very Low	Low	Medium	High	Very High
(1)	(2)	(3)	(4)	(5)

4.6.4 Participants

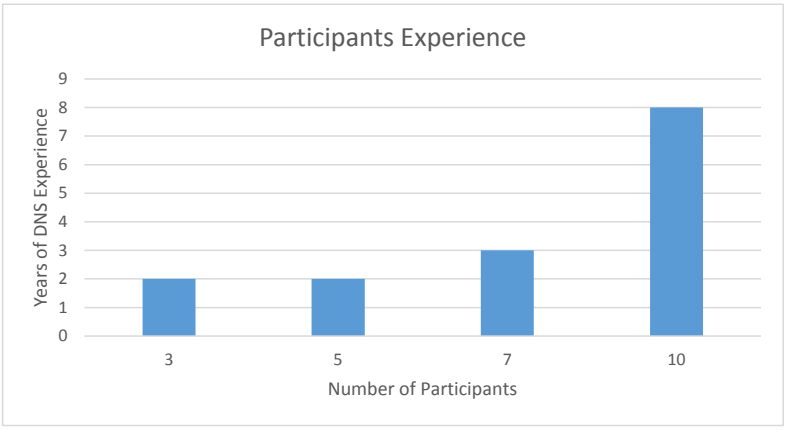
The participants were all TLD administrators responsible for managing one or more top-level domains. They were from different geographical locations with 5 from the Middle East, 3 from Europe, 1 from the Americas, 4 from the Asia Pacific region and 2 from Africa. Those administrators have a good range of DNS experience ranging from 3 to 10 years of experience.

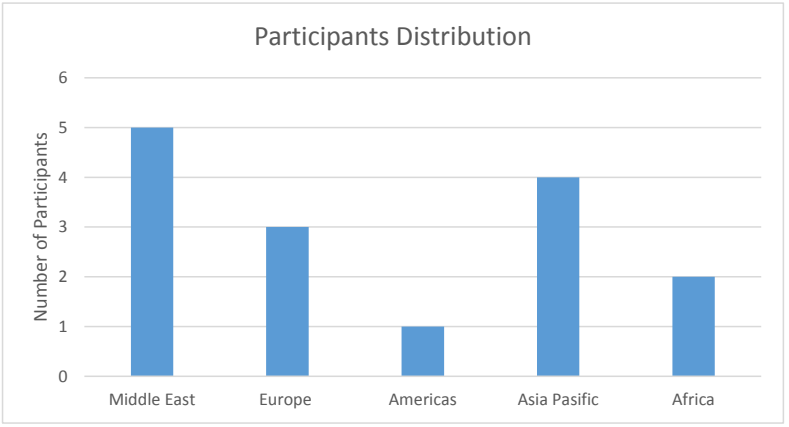
The TLDs managed by those administrators have various number of registered domain names ranges from a couple of thousand up to millions of domain names. Figures in Table 4.9 show the geographical distribution of the participants, their experience with the DNS system and the number of domain names registered under their TLDs.

It is clear that the set of participants are representative of a good spectrum of DNS operators around the world and their views can be effectively used in our experiment.

In order to establish the extent of consensus among the subjective opinions provided by the participants, we perform an inter-rater reliability analysis. We employ an intra-class correlation (ICC) [77] which is used to assess the consistency, or conformity, of measurements made by multiple observers measuring the same quantity. Table 4.10 reports the results of this statistical test based on a two way random effects model with a confidence interval of 95%.

As seen in this table, the single measure reliability of the four quality attributes is higher than 0.67, which shows that a reasonable agreement between the participants exists in terms of the perceived values for these attributes for each of the objects of the study.





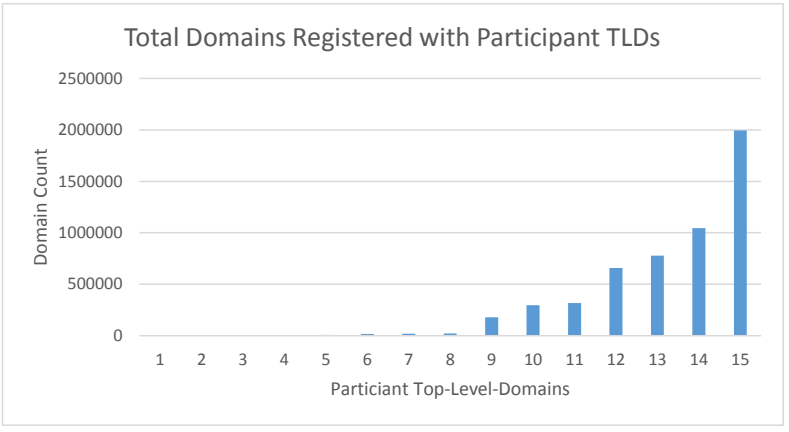


TABLE 4.9: Participants of the Empirical Assessment.

Table 4.10: Intra-Class Correlation (ICC).

Quality Attribute	ICC Single Measure
Availability	0.705
Security	0.712
Stability	0.709
Resiliency	0.68

4.6.5 Metric-Quality Correlation Analysis

In this section we will evaluate the first hypothesis which states that a meaningful correlation can be found between a set of DNS model structural measures and a set of quality attributes of the DNS system (H1).

In order to test this hypothesis we asked the participants to key in their views regarding the perceived quality attributes of a set of 9 DNS Dependency Model instances (i.e. Dependency Graphs). The models varied in terms of their metric values as shown in Table 4.11. The empirical data that were collected are also quantitatively reasonable from the perspective of the amount of data. We obtained 540 data points from the subjective opinions of the participants regarding the models (9 dependency models, 15 participants, 4 quality attributes).

Table 4.11: Measurements of Metrics on the 9 DNS Model Instances.

Model	AS	ANS	NETD	GEOD	Red	AC	AQP	DCZ	TPZ	DCO	TPO
M-1	34	3	3	1	3	0.89	4	3	5	3	6
M-2	21	4	4	1	4	0.98	3	4	4	4	3
M-3	13	4	4	1	4	0.84	2	2	0	4	1
M-4	10	4	1	4	4	0.43	2	2	0	1	1
M-5	19	3	2	1	3	0.44	4	4	1	2	3
M-6	10	4	4	4	4	0.5	2	1	1	4	0
M-7	15	6	2	2	2	0.89	2	2	0	2	1
M-8	16	2	1	1	2	0.5	2	4	1	1	2
M-9	21	8	8	8	8	0.84	2	2	0	8	1

The metric-quality correlation analysis shows that some of the metrics are in fact correlated to certain quality attributes with various coefficients. The technique that

we explore is the use of Spearman's Rho correlation, namely to identify relationship between the measured metrics of the models and the four quality attributes. Spearman's Rho correlation coefficient is a statistical measure of the strength of a monotonic relationship between paired data and its value ranges from -1 to 1.

Table 4.12: Metric-Quality correlations (Spearman's Rho).

Metrics	Availability	Security	Stability	Resiliency
AS	-.819*	-.685*	-.757*	0.33
ANS	0.258	-0.079	-0.01	0.02
NETD	0.037	-0.273	-0.027	.666*
GEOD	-0.056	-0.302	-0.086	.777*
TPZ	-.828*	-.703*	-.743*	0.248
AQP	0.109	-0.011	-0.004	-0.129
RED	0.02	-0.252	0.127	0.185
AC	0.05	-0.177	0.094	-.536*
DCZ	-0.276	-0.479	-0.355	.685*
DCO	0.105	-0.225	0.045	.698*
TPO	-.768*	-.609*	-.739*	0.156
*. Correlation is significant at the 0.05 level (2-tailed).				

According to Spearman's correlation, a correlation with a significance value greater than 0.50 can be considered to be significant, and therefore, in our work, such correlations are considered to be meaningful and are marked as shown in Table 4.12. As it can be seen, significant correlations can be found between some of the metrics and the four DNS quality attributes. This shows that the structural metrics defined for a DNS model can be used as early indicators for external quality attributes of the DNS. In addition, the correlations can be explained by the following two points:

- Metrics that reflect third party influence (as a result of peering with external organizations for secondary server hosting and placing servers under third party zones) such as AS, TPO and TPZ has clear negative impact on the availability, security and stability of the DNS. Choosing servers with names under other zones (increasing third party zones) provides zone redundancy but may

incur security and stability threats to the zone due to increasing the Attack Surface (AS) metric of the model.

DNS administrators should try to avoid such practice by reducing the size of their dependency graph (AS metric) by placing authoritative name servers for a certain zone under the same zone.

- Physically distributing the servers (geographical and network wise diversity metrics) ensures a certain degree of resistance against different types of failures and subsequently have positive impact on the resiliency of the whole system. Resiliency of the DNS is positively correlated with those metrics that are directly configured by the system administrator such as (GeoD, NetD, AC, DCZ and DCO).

  DNS administrators have to pay more attention regarding the deployment of their servers geographically and from a network distribution prospective. Also coordination with peer hosting organisations is vital in case of failures and the necessity to reduce this metric and consequently reduce zone complexity is clear to guarantee a higher level of resiliency of the system.

4.6.6 Prediction Models

In this section, we will apply some machine learning algorithms in order to construct prediction models of the quality attributes of the DNS system out of the structural metrics of the dependency model and evaluate the accuracy of these models (H2).

We used the measured structural metrics of the 9 models with the perceived quality attributes as keyed in by the participants as the training dataset for the prediction models. As far as the test dataset is concerned, we constructed the Dependency Graphs of the 15 participants' TLDs using our DGBuilder tool and then measured the various structural metrics on these models. We combined this data with the

perceived quality attributes from the participants concerning their own TLDs to construct the test dataset. The two data sets used in this experiment are totally independent and they can be effectively used to train the models and test their performance.

TABLE 4.13: Performance of the Predictive Models in terms of the correctly classified instances out of the test dataset.

Classifier Name	Availability	Security	Stability	Resiliency
RF	73%	47%	53%	40%
LWL	53%	53%	67%	33%
SL	7%	53%	20%	27%
PART	20%	73%	20%	73%

Figures from Figure 4.3 to Figure 4.6 show the different parameters used to evaluate the performance of the different prediction models on the test dataset. Model accuracy is measured by the area under the ROC curve. An area of 1 represents a perfect test; an area of 0.5 represents a worthless test. Table 6 shows the percentage of correctly classified instances using each of the predictive models and Figure 5 shows the performance of the predictive models in terms of the area under the ROC curve and other useful model performance indicators.

The results of applying the evaluation strategies on the produced models indicate that the RF classifier outperformed other classifiers in producing the best prediction model for the DNS availability, while LWL is the best for stability. PART outperformed other classifiers in predicting the quality attributes of security and resiliency.

4.6.7 Threats to Validity

Empirical evaluation is always subject to different threats that can influence the validity of the results. We will specifically refer to the aspects of our experiment

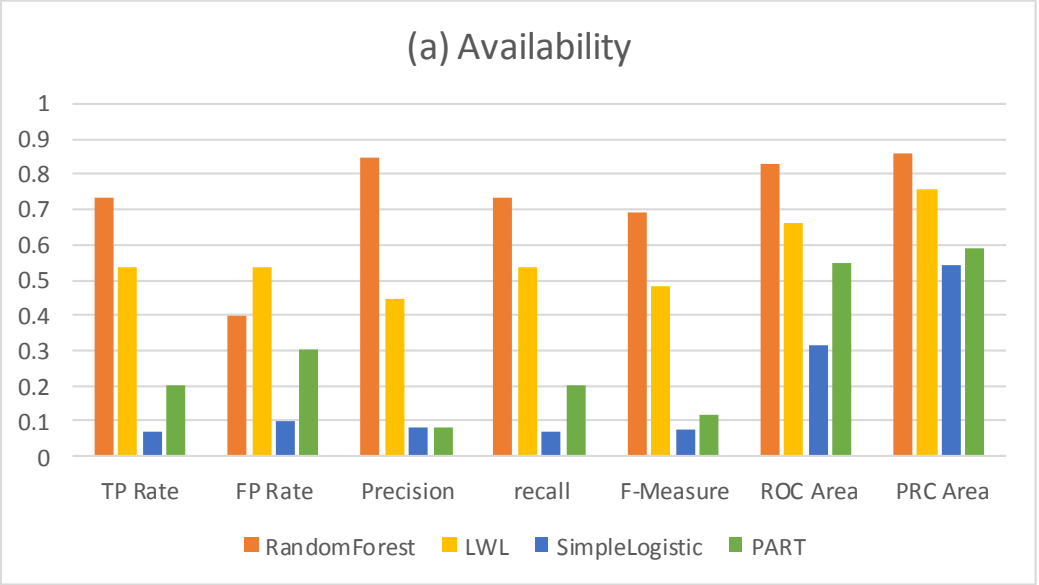


FIGURE 4.3: Availability Prediction Models and their Performance Indicators.

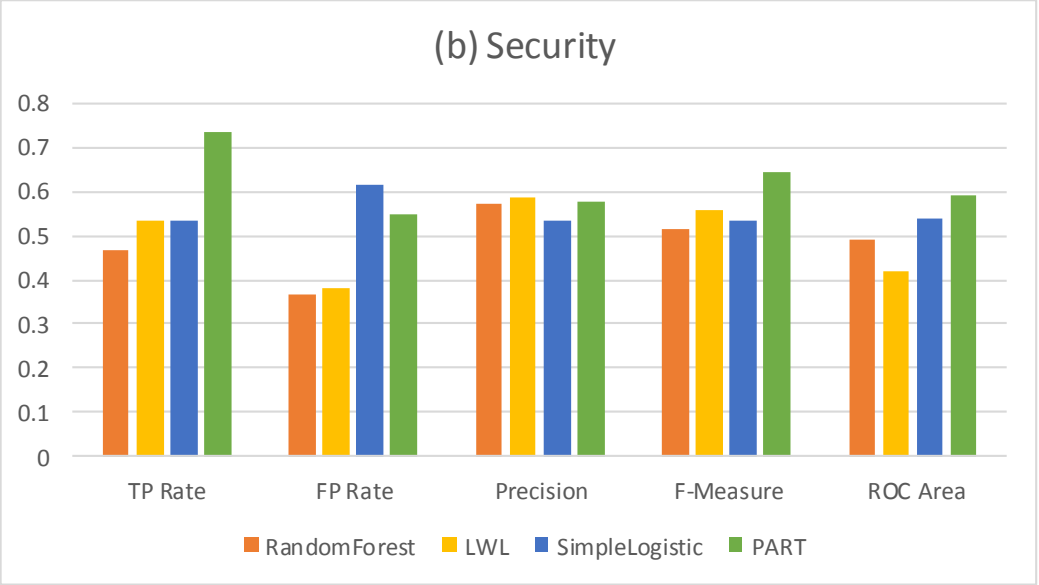


FIGURE 4.4: Security Prediction Models and their Performance Indicators.

that may have been affected by these threats.

Conclusion Validity: In our experiment, a limited number of data points were

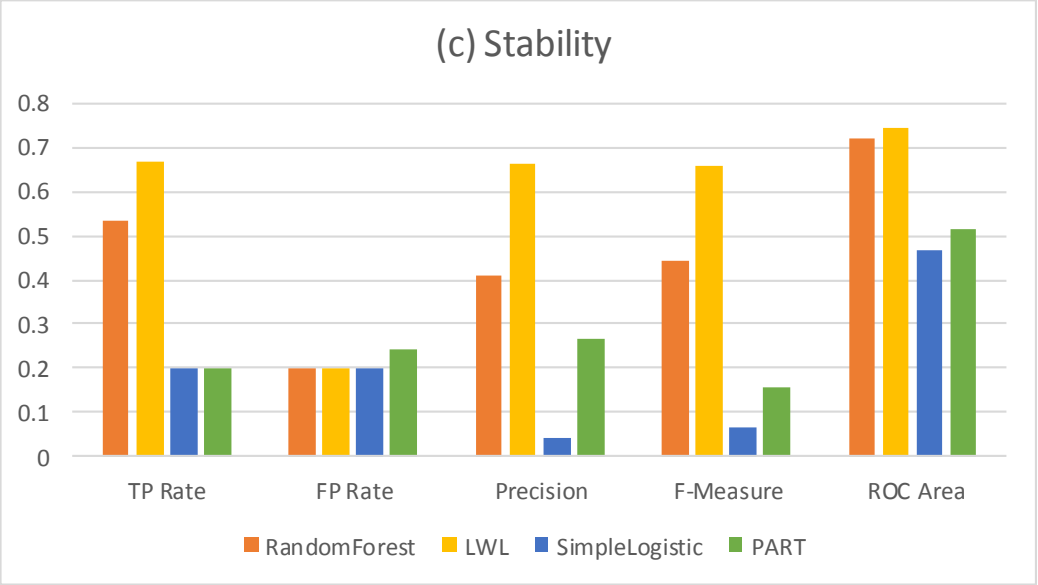


FIGURE 4.5: Stability Prediction Models and their Performance Indicators.

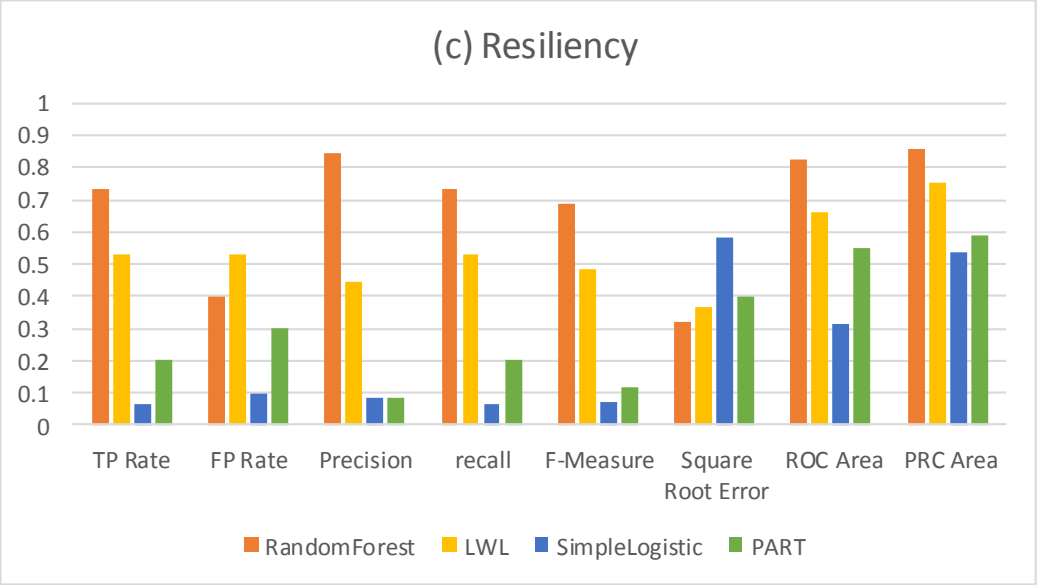


FIGURE 4.6: Resiliency Prediction Models and their Performance Indicators.

collected due to the limited number of participants amongst the DNS operators. In addition, there were almost no models at our disposal and we have to build

customized models using our DGBuilder tool. These limitations may pose threats to the drawn conclusions.

Construct Validity: The dependent variables which are the four quality attributes of the DNS model were measured using the subjective opinion of the participants. The threat posed by using subjective measurement mechanisms is that different participants may have different attitudes toward the evaluation of these attributes. In general, the participants of this experiment have a considerable number of years of experience within the DNS administration and their subjective views does capture what we claim to measure. It should also be noted that the used set of metrics may not be comprehensive and other consecutive research could further complete this proposed set by defining new metrics from other perspectives.

Internal Validity: Each of the dependency models represented different DNS system configuration and deployment structure. However, the models were simple enough to be understandable by the participants and they were given enough time (2 weeks) to become familiar with the concepts, structure and components of each model. The use of the 5-point Likert scale could have impacted the internal validity of the experiment due to the discrete nature of this ordinal scale in capturing the participants' views.

External Validity: The following two issues were considered for external validity:

- the models used in the experiment are representative of wide range of real-world operational configurations and deployment choices.
- We needed participants with high level of industrial experience to be able to complete the experiment and the target group of TLD operators did the job perfectly.

Another threat to validity may be related to the tools that were used; however, since the tools were used to build the models and extract the metrics; we believe that it possibly affected all of the model measurements in the same way.

4.6.8 Discussion

In our assessment experiment, we used a Likert scale to key in the perceived quality attributes of the models. We can consider them to be measurements on a quantitative scale, since they represent different levels of perceived qualities. Likert scales contain multiple items and can be taken to be ordinal scales so descriptive statistics can be applied, as well as correlation analyses, factor analyses, analysis of variance procedures, etc. (if all other design conditions and assumptions are met). Any means and standard deviations obtained from rating data (such as the Likert Scale) are perfectly valid as descriptions of participants' behaviour, i.e. how participants responded when faced with a question and asked to pick a response [78].

In supervised classification problems with ordered classes, it is common to assess the performance of the classifier using measures more appropriate for nominal classes, regression problems or preference learning [79, 80]. Baccianella [79] addresses the adaptation of existing measures (Mean Absolute Error) to unbalanced data, while Gaudette [80] compares existing measures concluding that Mean Absolute Error and Mean Square Error are the best performance metrics. Other strategies encompass the use of rank order measures [81, 82] or the adaptation of the ROC curve [83].

As it can be seen from Figure 4.3 to Figure 4.6, the Mean Absolute Error (MAE) is in the worst case less than 0.4 out of 5. We can find an upper and lower bound on the accuracy of the predictive models. Since the values of the quality attributes to be predicted are natural numbers from 1 to 5, the error of around 0.4 can either be rounded up to 1 for the worst case, or considered as is for the best case. If we

consider the worst case, the accuracy of the predictive models will be $\frac{5-1}{5} = 80\%$; however, for the best case, this is equivalent to $\frac{(5-0.4)}{5} = 92.5\%$ accuracy for the predictive model.

Even for the worst case, the accuracy rate of the predictive models is quite high and supports our hypothesis that acceptable predictive models can be built from structural metrics of the DNS model in order to predict the DNS quality attributes of availability, security, stability and resiliency.

However, the application of these measures (MAE) faces some difficulties in the context of ordinal classification [84]. This will be investigated more in future work in order to apply a better evaluation parameter for the prediction models.

The Ordinal Classification Index (OCI) proposed in [84] will be used since it captures how much the result diverges from the ideal prediction and how "inconsistent" the classifier is in regard to the relative order of the classes. This metric is defined directly on the Confusion Matrix (CM) specifically to evaluate the performance in ordinal data classification.

4.6.9 Conclusions

Our findings demonstrate the potential of the proposed DNS Model structural metrics to serve as validated predictors of the operational system quality. This work has implications both for research and practice. The strength of the correlation of structural metrics with different quality aspects clearly shows the potential of these metrics to accurately capture aspects that are closely connected with actual usage. From a practical perspective, these structural metrics can provide valuable guidance for the DNS system managers and zone administrators, in adjusting their configurations and deployment layout to improve the quality attributes of their systems.

Chapter 5

The ISDR Method

During the past thirty years the Domain Name System (DNS) has sustained phenomenal growth while maintaining satisfactory user-level performance. However, the original design focused mainly on system robustness against physical failures, and neglected the impact of operational errors such as misconfigurations and bad deployment choices. Although DNS troubleshooting techniques and problem identification methods have been proposed and several tools have been built, most of these methods and tools apply their detection techniques directly on the zone files through a predefined zone schema and integrity constraints. They don't take into account the inter-dependencies stemming from the hierarchical nature of the DNS or the zone administrators practices.

Instead, we propose a model-based approach that subsumes all the steps necessary to identify, specify and detect the DNS operational bad smells. We utilize dependency graphs (as instance of the DNS Dependency Model) to identify, detect and catalogue operational bad smells. Our method deals with smells on a high-level of abstraction using a consistent taxonomy and reusable vocabulary, defined by the DNS Model. The method is used to build a diagnostic DNS quality advisory tool

that detects configuration changes that might decrease the robustness or security posture of domain names before they become into production.

5.1 Bad Smells

In software engineering, bad smells in code [85] identify risks to quality attributes of a software system. Code "bad" smells, a term originally coined by Kent Beck in [86] to refer to "a code smell is a surface indication that usually corresponds to a deeper problem in the system". The concept of code smells has been proposed to characterize different types of design shortcomings in code. Additionally, metric-based detection algorithms claim to identify the "smelly" components automatically. They are widely used for detecting refactoring opportunities in software [46]. Some studies [87], [88] have also used the historical data to identify the spots, where programmers have made changes or refactorings to the software.

Joshua Garcia et. al [89], introduced the concept of architectural "bad smells", which are frequently recurring software designs that can have non-obvious and significant detrimental effects on system life cycle properties, such as understandability, testability, extensibility, and re-usability. They define architectural smells and differentiate them from related concepts, such as architectural anti-patterns and code smells.

We transfer these ideas to the realm of the DNS, where operational *bad smells* are defined as configuration and deployment choices by zone administrators that are not errant or technically incorrect, and do not currently prevent the system from doing its designated functionality. Instead, they indicate weaknesses that may impose additional overhead on DNS queries, or increase the system vulnerability to threats,

or increase the risk of failures in the future. Despite its tremendous success, DNS is not without weakness.

The critical importance of the DNS places a high standard on its resilience and its design warrants further examination, as evidenced by the following example. During January 2001 all the authoritative servers for the Microsoft DNS domain became inaccessible [2]. This failure was due to a simple configuration mistake where Microsoft placed all its DNS servers behind the same network router, despite the well documented guidelines on geographically dispersing DNS servers [9], and the switch failed. During this event, the number of DNS queries for the Microsoft domain seen at the F root server surged from the normal 0.003% of all the queries to over 25%. Other root servers had similar increase of queries for the Microsoft (microsoft.com) domain.

Typical DNS smells have to do with redundancies, ambiguities, inconsistencies, incompleteness, non-adherence to DNS design conventions, best practices or standards, and so on. The challenge is to come up with a comprehensive and commonly accepted list of DNS operational smells, as well as tool support to detect such smells. What is also needed is a good understanding of the relation between those smells and correction mechanisms (in the form of graph-based refactorings), in order to be able to suggest, for any given smell, appropriate refactorings that can remove this smell.

5.2 The ISDR Method

The ISDR (**I**dentification, **S**pecification, **D**etection and **R**efactoring) method proposed throughout this chapter is composed of four stages as shown in Figure 5.1 and

produces the DNS operational bad smells and refactoring catalogues. The following items summarise these four stages:

- *Identification*, through domain analysis using DNS standards in the form of Request for Comments (RFCs), best practices and policy documents, literature review and DNS expert views.
- *Specification* of a set of operational bad smells using a reusable vocabulary and classification of the bad smells in a taxonomy that shows the scope of the inspection element or plane and system's external qualities affected by the smell.
- *Detection* of bad smells in the form of general detection queries, procedures and formulas.
- *Refactoring* as a correction mechanism to the operational bad smells in the form of graph-based model transformations. Other correction mechanisms may be formulated in the form of reports or reconfiguration recommendations.

In the following subsections, we elaborate on each step of the ISDR method more with clarifying examples and validate the method using case studies.

5.2.1 Bad Smells Identification

The first stage in our method consists of performing deep analysis of the DNS standards, Request for Comments (RFCs), best practices and policy documents to identify weaknesses in configuration and deployment choices made by administrators that may impose additional overhead on DNS queries, or increase the system vulnerability to threats, or increase the risk of cascaded failures. In the following, we show, through several examples, how this step has been used to extract smells along

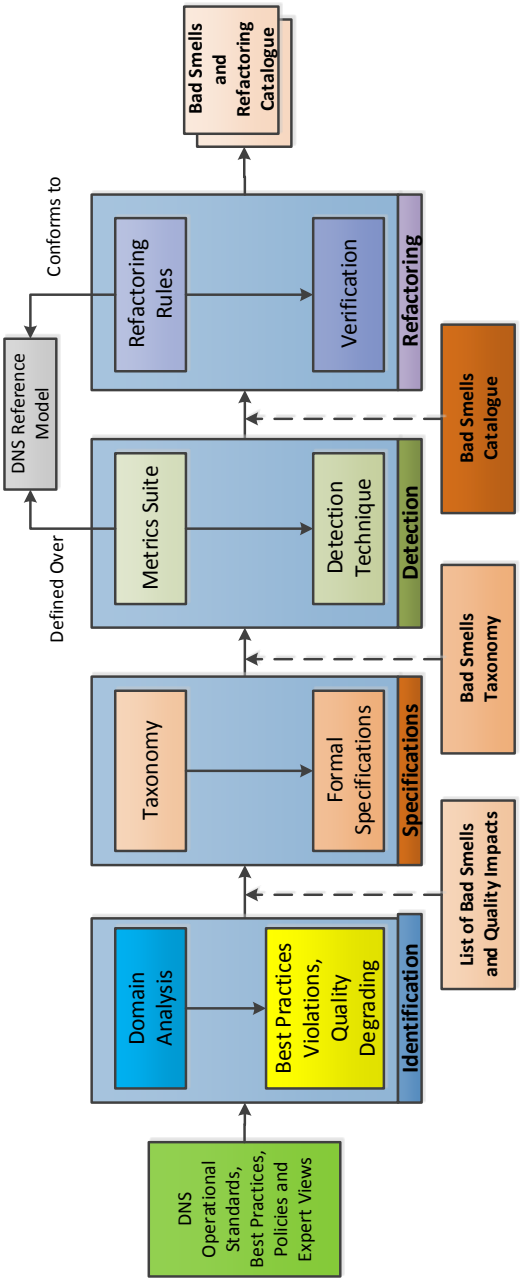


FIGURE 5.1: The ISDR Method.

with quality attributes that may be affected by the presence of such misconfiguration or bad deployment and how to detect the occurrence of such smell in the zone's dependency graph.

1. When a parent zone P delegates part of its name space to a child zone C, P stores a list of NS resource records for the authoritative servers of zone C. This list of NS resource records are kept both at the parent and the child zone. Whenever the operator of zone C makes changes to one or more of $C's$ authoritative servers, he must coordinate with the operator for zone P to update P accordingly. In reality, there are cases where changes made at the child zone are not reflected at the parent zone, usually due to *"bad"* coordination between them. As a consequence, the NS RR set of the child zone can be completely different from the NS RR set of the parent zone. Even though the parent and the child zone are not required to list the same NS RR set (due to DNS specifications' ambiguity in this regard), *delegation inconsistency* can affect the availability and stability of the zone.

2. When name servers are selected as secondary name servers for a zone, they should be placed in a topologically resilient manner. If more than one name server is on the same physical subnet, then any outages on that subnet would affect all of the name servers on it; as such, the recommendation is that name servers be allocated on distinct physical subnets and distinct geographical locations. It's important when picking geographical locations for secondary name servers to minimize latency as well as increase reliability.

3. The DNS system relies heavily on replication (based on zone file transfers) to achieve its reliability goals, but this form of replication typically requires cooperation and coordination with other DNS administrators and hosting organisations including parent zone managers. Such coordination is essential to the stable and resilient operation of the DNS and the ability for a quick recovery from system failures.

4. Distributed management is crucial in achieving DNS's scalability, however it also leads to inconsistencies due to mistakes in coordinating zone configurations

and changes. While human induced configuration errors are a well-known fact, DNS delegation configurations require consistency across administrative boundaries, a condition that is even more prone to errors. Unfortunately the current system does not provide any automated means to communicate for coordination. Today configurations are communicated manually, and this process is highly subject to errors.

Table 5.1 shows examples of design rules, best practices and operational recommendations within the different DNS operational planes that are being used in the identification stage of the ISDR method.

Table 5.1: Identification of Bad Smells in the DNS Planes

No.	Design Rule/Best Practice	Reference	Effects On Quality	Bad Smell(s)
Data Plane				
1	SOA records with various timers have been set (far) too low or (far) too high. Especially for top level domain name servers. This causes unnecessary traffic over international and intercontinental links.	RFC1537	Availability, security and stability	Zone Thrush and Zone Drift
Control Plane				
2	It is required to have at least two nameservers for every domain, though more is preferred with secondary servers topologically and geographically dispersed.	RFC1912, 2182	Availability, security, stability and resilience	LowANS,False or Diminished redundancy
Management Plane				
3	The trust relationships involved in zone transfer are still very much a hop-by-hop matter of name server operators trusting other name server operators rather than an end-to-end.	RFC3833	Security and availability	Betrayal By Trusted Server

5.2.2 Formal Specifications

The weaknesses identified in the previous step, termed as *operational bad smells*, are then defined using certain key terms, unified vocabulary and reusable concepts in this domain. Providing the bad smells in a flat structure without defining any categories or relationships among the smells hinders their formal specification, comparison, and, consequently, detection. Key concepts are identified in the text-based descriptions of smells in the literature. They form a unified vocabulary of reusable concepts to describe smells. The concepts, which constitute a vocabulary, are combined to specify smells systematically and consistently. In addition to a unified vocabulary of reusable concepts, a taxonomy and classification of smells are defined using the key concepts.

According to Cambridge dictionary, a taxonomy is "A system for naming and organizing things [. . .] into groups which share similar qualities". Some taxonomies, such as the taxonomic organisation of species in a biological context, are hierarchical, but this is not a prerequisite. The taxonomy highlights and charts the similarities and differences among smells and their key concepts. We developed a taxonomy that describes the structural relationships between the various bad smells. The taxonomy has an important role in defining the scope of inspection and highlighting the metrics or structural properties related to the bad smell. It classifies the bad smells based on the following categories:

1. Operational plane: Data, control and management planes.
2. Affected entity types: Single type, inter-type, intra-type, or inter-zone.
3. Property of the smell: Lexical, structural or measurable.

We distinguish bad smells occurring in and among resource records within zones (single type, inter-type, intra-type and intra-zone). We further divide those subcategories into structural, lexical, and measurable smells. This division helps in identifying appropriate detection techniques. For example, the detection of a structural smell may essentially be based on static analyses of the model instance; the detection of a lexical smell may rely on best practices or guidelines recommendations analysis; the detection of a measurable smell may use metrics. Our classification is generic and may classify smells in more than one category (e.g., Corrupted Parent).

Figure 5.2 shows a graphical representation of the DNS operational bad smells taxonomy. The taxonomy is generic and defines a bad smell in more than one category. It can easily be extended by defining new categories of bad smells based on subsequent iterations of the DNS operational domain analysis. So far we have already identified 19 bad smells that can be used as a representative set that spans the different operational planes with various detection properties. Since several smells are closely related and the number of the smells is quite high, we feel that this taxonomy, which categorizes similar bad smells, is beneficial. We believe that the taxonomy makes the smells more understandable and recognizes the relationships between the smells.

5.2.3 Detection

One can develop a set of simple mechanisms to detect some of the lurking errors identified in the zone configuration and deployment choices of the DNS. Delegation inconsistency and lame delegation errors can be detected by a simple process between parent and child zones to periodically check the consistency of the NS records stored at each place. Cyclic zone dependency can be detected via automatic checking by trying to resolve a name through each of the authoritative servers in the zone.

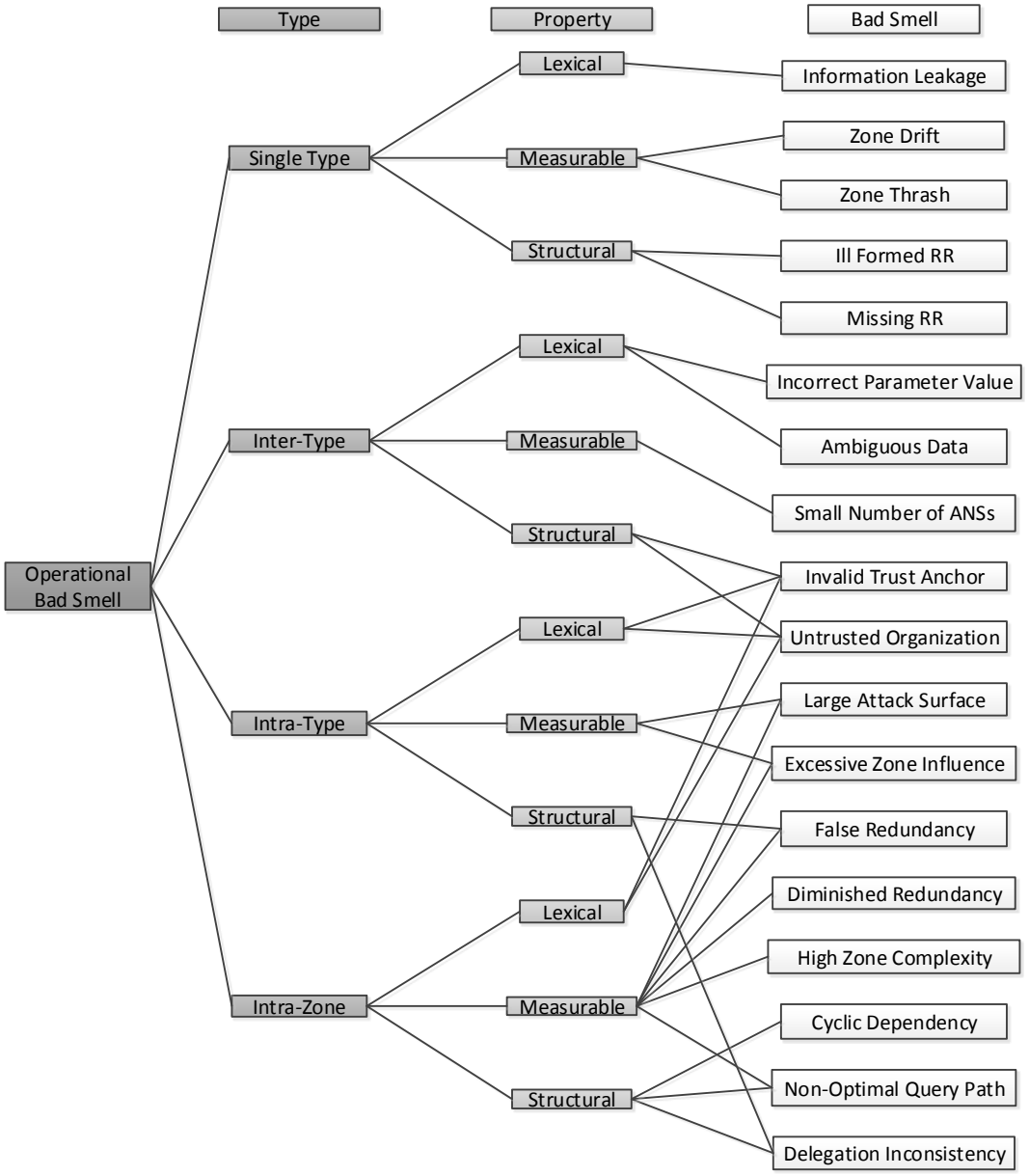


FIGURE 5.2: Bad Smells Taxonomy.

Although there may not be a single check to detect the diminished server redundancy problem, automatic periodic measurement between servers of the same zone on their IP address distance, geographical distance and hop count distance can effectively reflect the diversity degree in their placement. These simple checks are absent from the original DNS design, not because they are difficult to do but a lack of appreciation

of the severity of human-introduced errors.

In order to be able to detect bad smells occurring in model instances of our *DNS Model*, we need to capture deviations of the particular instance model from the good and recommended operational best practices. Lexical and structural properties are used to detect some of the bad smells using direct queries on the instance model such as *(Are there any cycles in the dependency graph?)*.

The metric-based approach combines a set of metrics and set operators to compare them against absolute or relative threshold values. Setting the absolute or relative operational metrics threshold values can be done using local policy constraints or best practices from the wider DNS domain literature and expert views. In the context of metrics-based analysis techniques, the *DNS Structural Metrics Suite* defined over the DNS Model and presented in Chapter 4 is a valuable tool that is being utilised in detecting the presence of measurable bad smells in the DNS model instance.

5.2.4 Refactoring

In the area of object-oriented programming, refactoring [90] is the technique of choice for improving the structure of existing code without changing its external behaviour. Graph-based, general refactoring rules [91] are being suggested to remove the bad smells identified and detected in the previous stages. The general approach of refactoring [46] is to include the following steps: (1) identify the location for refactoring, (2) determine which refactoring rules should be applied and on what sequence, (3) guarantee that refactoring rules are preserving the external behaviour of the system, (4) application of selected refactoring rules, (5) assess the effect of refactoring on the system's external qualities and (6) maintain the consistency between the refactored elements and other system artefacts.

5.2.5 Bad Smells' Quality Impacts

In the DNS realm, operational bad smells are configuration and deployment choices, made by zone administrators that are not totally errant or technically incorrect, and do not currently prevent the system from doing its designated functionality. Instead, they indicate weaknesses that may impose additional overhead on DNS queries, or increase the system vulnerability to threats, or increase the risk of failures in the future. For example, best practices for ensuring availability and security of the DNS

Table 5.2: DNS Operational Bad Smells

No	Bad Smell	Quality Impacts			
		Availability	Security	Stability	Resiliency
1	Unnecessary RRs (Information Leakage)		X		
2	Large Parameter Value (Zone Thrush)	X	X	X	
3	Small Parameter Value (Zone Drift)	X	X	X	
4	Ill-Formed RRs	X			
5	Missing RRs	X		X	
6	Incorrect Parameter Data		X	X	
7	Ambiguous Data	X	X		
8	Small Number of ANSs	X		X	X
9	Invalid Trust Anchor		X		
10	Untrusted-Peer Organisation	X	X	X	
11	Large Attack Surface	X	X	X	X
12	Excessive Zone Influence		X	X	
13	False Redundancy	X			X
14	Diminished Redundancy	X			X
15	High Zone Complexity			X	X
16	Cyclic Dependency	X			X
17	Non-Optimal Query Path	X			X
18	Delegation Inconcsistency	X		X	

infrastructure recommend (1) defining a number of name servers for each domain, (2) configuring these name servers under at least two different parent domains and

(3) placing the physical name servers, hosting the zone files for the domain, in separate networks. The redundancy provides for stability of the domain and prevents single point of failure. In particular, if one of the parent domains is not accessible, the domain will remain functional via the other parent domain; in case one of the networks, hosting the name servers, is under attack, the other name server, located in available networks, can be reached.

On the flip side, while ensuring availability, this redundancy introduces new dependencies which can be utilised to attack the domain. Specifically, if vulnerability exists in a network or a name server hosting the domain, it can be exploited to attack the domain, e.g., inject spoofed DNS record for domain hijacking. Through an extensive literature review, we found that the presence of operational bad smells have direct impact on the external qualities of the domain name system and Table 5.2 shows the quality impacts resulted from the presence of the various already identified bad smells.

5.2.6 Bad Smells Catalogue

The set of identified bad smells is being formally specified in concise and reusable terms based on a template that includes the bad smell name, type, inspection plane(s), description, occurrences, quality impacts and detection strategies. The *bad smells catalogue* is being expanded further by including refactoring rules for each smell and how these rules have to be applied on the model instance to eliminate the concerned bad smell. Example of catalogue entry is shown in Table 5.3 while the complete catalogue is listed in Appendix C.

TABLE 5.3: Catalogue Entry for the Cyclic Dependency Bad Smell.

Name	Cyclic Dependency.
Type	Intra-Zone, Structural.
Inspection Planes	Data and Control Planes.
Occurrences	Cyclic zone dependency occurs when two or more zones depend on each other in a circular way.
Quality Impacts	Reduced availability and reduced resiliency.
Detection Strategy	Is there any cycle in the Dependency Graph? (Query on the DNS Operational Model Instance).
Correction Mechanism (Refactoring)	Add a glue record for the (out-of-bailiwick) authoritative name servers involved in the cycle in the zone file.

5.3 Method Validation

We validate our method by applying it and its associated execution technique to a bad smell that has been already identified as one of the most important misconfigurations in the literature. [7, 9, 15, 17, 19].

Case Study: Cyclic Dependency

To achieve acceptable geographical and network diversity, zone administrators often establish mutual arrangement with peer organizations to host each other's zone files. Authoritative name servers located in other zones are normally identified by their names instead of their addresses and called out-of-bailiwick name servers. A cyclic zone dependency [15] occurs when two or more zones depend on each other in a circular way.

Table 5.4 shows that the zone (example.com) has 4 authoritative name servers responsible for resolving domain names under this zone as defined in its parent zone (.com). Two servers (ns1 and ns2.example.com) are *in-bailiwick* servers and it is mandatory to include their IP addresses in the parent zone in order to properly resolve domain names under that zone. The other two servers (dns1 and dns2.example.net) are located in another zone and there is no need to include their

IP addresses in the (.com), example.com parent zone file. On the other hand, the (.net) zone which is the parent of the (example.net) zone, is served by two *out-of-bailiwick* name servers located in the (example.com) zone.

Table 5.4: Content of Zone File for Case Study.

$ORIGIN .com.		
example.com.	NS	ns1.example.com.
example.com.	NS	ns2.example.com.
example.com.	NS	dns1.example.net.
example.com.	NS	dns2.example.net.
ns1.example.com.	A	1.1.1.1
ns2.example.com.	A	1.1.1.2

$ORIGIN .net.		
example.net.	NS	ns1.example.com.
example.net.	NS	ns2.example.com.

In this example, the two zones work nicely under normal circumstances but if (for any reason), both in-bailiwick name servers become unavailable, both example.com and example.net zones will not be reachable because the IP addresses of the other two authoritative name servers can't be resolved. This example illustrates the failure dependency between zones, where the failure of some servers in one zone will render the other zone unreachable. The quality impacts of such a bad smell are significant reduction on availability and resiliency of the zone against multiple points of failure.

Checking each zone individually for configuration errors will not lead to the detection of this *Cyclic Dependency* bad smell since they are both configured correctly. On the other hand, constructing the dependency graph will easily show the occurrence of *two* distinct circular paths that identify the presence of this particular smell. Figure 5.4 shows the concerned part of the dependency graph of our example.

Cyclic Dependencies can be eliminated by the *creation* of specific resource records (RRType: A) for both out-of-bailiwick servers (dns1 and dns2.example.net) in the (.com) zone. This enables resolving the domain names under the (example.com) and (example.net) zones even when the two in-bailiwick servers are unreachable. We execute this correction mechanism in the form of a graph transformation based refactoring rule *(CreateARecord)* as shown in Figure 5.4. Since we have two distinct

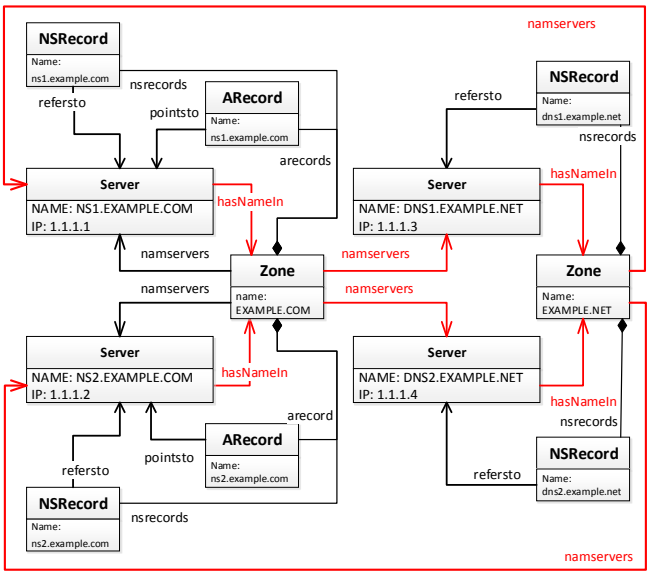


FIGURE 5.3: Part of the Dependency Graph of the Case Study.

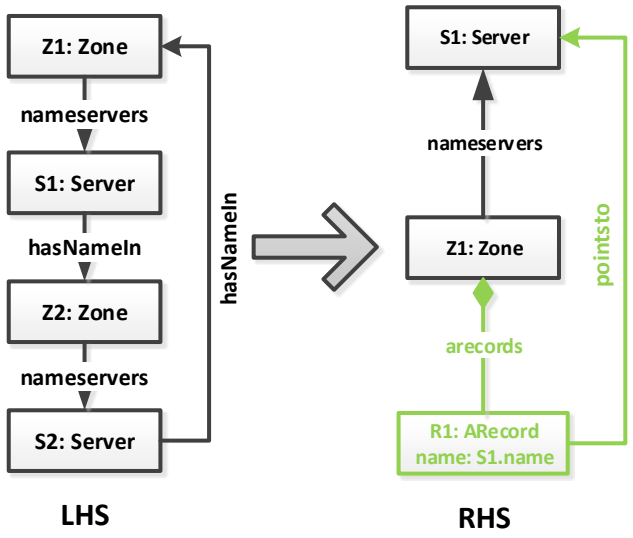


FIGURE 5.4: Refactoring Rule: CreateARecord.

matches for the LHS of the rule on the actual instantiation of the model (the dependency graph in Figure 5.3), then the rule needs to be applied twice in order to remedy all occurrences of the bad smell. A new zone file can then be automatically generated from the newly transformed model instance (i.e. dependency graph) as

shown in Table 5.5 and/or a set of recommendations can be presented to the system administrator to eliminate the bad smell.

Table 5.5: New Zone File Generated After Executing the Refactoring Rule(s).

$ORIGIN .com.		
example.com.	NS	ns1.example.com.
example.com.	NS	ns2.example.com.
example.com.	NS	dns1.example.net.
example.com.	NS	dns2.example.net.
ns1.example.com.	A	1.1.1.1
ns2.example.com.	A	1.1.1.2
dns1.example.net.	A	1.1.1.3
dns2.example.net.	A	1.1.1.4

$ORIGIN .net.		
example.net.	NS	ns1.example.com.
example.net.	NS	ns2.example.com.

Chapter 6

ISDR Method Implementation

The *ISDR method* lays the basis for developing a visual advisory tool *(the DNS Advisor)* for system administrators to analyse, discover, and remedy operational bad smells. This chapter presents the application of the various techniques within the ISDR method utilizing the tools within the EMF Refactor Framework [38] and our dependency graph builder *the DGBuilder* tool. The ISDR method is executed on a particular instance of the DNS model (i.e. Dependency Graph) using the following steps:

- **Step 1:** Extract the dependency graph from the zone configuration file and the authoritative name servers' deployment using the DGBuilder tool.
- **Step 2:** Query the dependency graph for any bad smell using the methods and metrics defined in the Bad Smells Catalogue using the specifications of the techniques through the EMF Refactor Framework.
- **Step 3:** Apply relevant refactoring rule(s) on all matching occurrences of the LHS of the rule on the instance model. A new dependency graph is generated as an output of this step.

- **Step 4:** New zone file(s) and authoritative name servers' deployment layout can be automatically generated from the new Dependency Graph or a set of recommendations can be presented to the system administrator with relevant quality impacts.

In this chapter, we discusses the specifications, tools and implementation of the ISDR method techniques and presents several case studies for its validation. Section 6.1 explains the various tools used in our implementation including our in-house developed *DGBuilder* component.

The DGBuilder is used to build the DNS model instance (Dependency Graph) for a "live" zone out from the zone configuration and authoritative name servers' deployment layouts. Section 6.2 describes how the various ISDR techniques (including metrics, smells and refactorings) are specified using the EMF Refactor framework and then applied on the generated model instances as case studies to validate the method and verify its usefulness in detecting and refactoring DNS operational bad smells. Section **??** gives a short background on related work in the software engineering fields of bad smells detection and refactoring.

6.1 Tool Support

The application of the ISDR method, in a systematic process, can automatically direct the zone administrator to places in the zone file that contain potential design and deployment problems that may compromise availability, security, stability or resiliency of the domain name system before the changes become into production. Zone administrator are able to run several scenarios and apply several refactoring rules through the tool to determine the solution that best meets their local policies. The following frameworks and modelling languages have been identified and used in order to implement the ISDR method as part of the DNS advisory tool.

6.1.1 Eclipse and EMF Modelling

The last decade witnessed a dramatic growth of software intricacy and different techniques and methodologies have been proposed to ease complex system development. Model Driven Engineering (MDE) [50] shifts the focus of software development from coding to modelling and lets software architects harness the opportunity of dealing with higher-level abstractions.

The EMF project [38] is a modeling framework and code generation facility for building tools and other applications based on a structured data model. From a model specification described in XMI, EMF provides tools and runtime support to produce a set of Java classes for the model, along with a set of adapter classes that enable viewing and command-based editing of the model, and a basic editor. EMF (core) is a common standard for data models, many technologies and frameworks are based on.

6.1.2 Henshin

The Henshin [41] project provides a state-of-the-art model transformation language for the Eclipse Modelling Framework. Henshin supports both direct transformations of EMF single model instances (endogenous transformations), and translation of source model instances into a target language (exogenous transformations). The Henshin language and toolset supports, the following features:

- Expressive transformation language with a graphical syntax, pattern matching and control-flow constructs with parameter passing,
- Support for endogenous and exogenous transformations, with natural treatment and efficient in-place execution of endogenous transformations,

- Formal graph transformation semantics, with arbitrary mixing of different graph transformation styles (DPO/SPO),
- Efficient interpreter engine based on constraint solving, and verification using state space tools and many other features.

6.1.3 EMF Refactor

EMF Refactor [1] is an existing Eclipse project which can calculate metrics and perform refactorings on Ecore and UML models. In particular, EMF Refactor supports metrics reporting, smell detection, and refactoring for models being based on the Eclipse Modeling Framework. The following techniques can be used in a concrete specification of a new EMF model metric, smell, or refactoring:

- Model metrics can be concretely specified in Java, as OCL expressions, by Henshin pattern rules, or as a combination of existing metrics using a binary operator.
- Model smells can be concretely specified in Java, by Henshin pattern rules, or as a combination of an existing metric and a comparator like greater than (>).
- The three parts of a model refactoring can be concretely specified in Java, in Henshin (pattern rules for precondition checks; transformations for the proper model change), or as a combination of existing refactoring using the CoMReL language.

Figure 6.1 shows the architecture of the EMF Refactor specification module using a UML component model. The specification module provides the generation of custom EMF Quality Assurance (QA) plugins containing the metric-, smell-, or

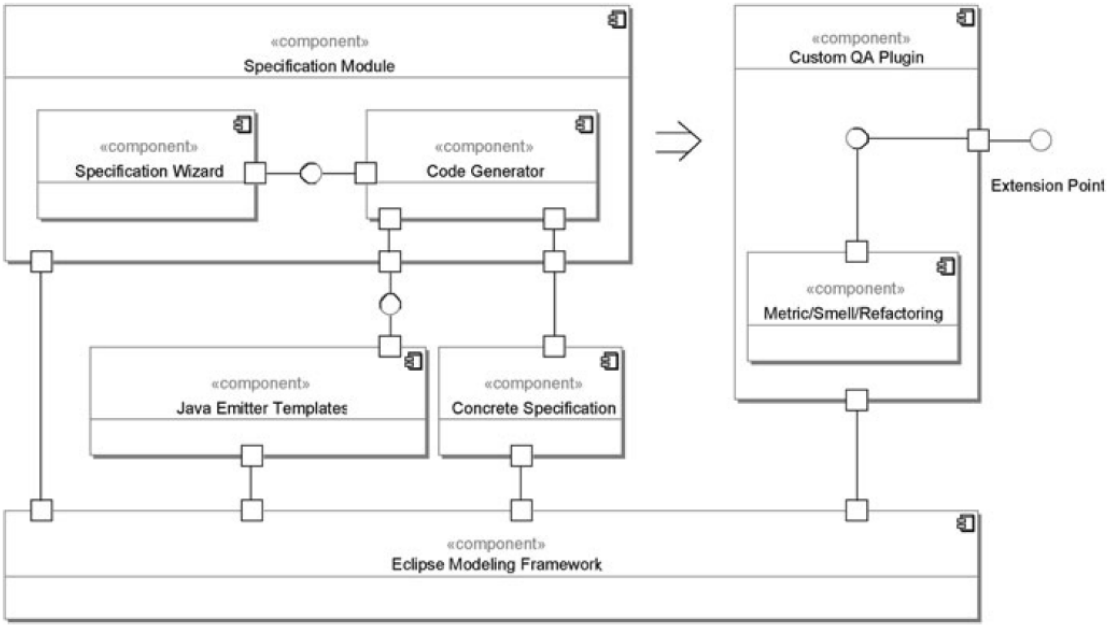


FIGURE 6.1: The EMF Refactor Specification Module. Adapted from [1].

refactoring-specific Java code. The specification module provides wizard-based specification processes (Specification Wizard component within the Specification Module in Figure 6.1). After inserting specific information (like the name of the metric, smell, or refactoring, defined over the corresponding model/meta-model), the code generator uses the Java Emitter Templates Framework [92] to generate the specific Java code required by the corresponding extension point.

6.1.4 Dependency Graph Builder (DGBuilder)

Example instances of the model can also be created using the EMF Examples Creation Wizards. Using this method, the different model elements can be created as children of the main (root) model element (DNSModel). On the other hand and in order to generate the model instance for various zones' configurations and sever deployments, the DGBuilder tool through the algorithm listed in Listing 1 is used. The dependency graph building tool (DGBuilder) is integrated directly into the

eclipse development environment by creating a dynamic instance of the DNS model in the *Runtime IDE*. The DG building algorithm is composed of the following three main steps:

1. **Step One [Lines 1-6]:** Necessary (Infrastructure) resource records are extracted from the zone file (Z) using the DNS *dig* utility. We limited our focus in this research on the infrastructure DNS resource records, which affect the delegation consistency, security and the stable operation of the DNS system. In order to ensure the correctness of the information retrieved, all authoritative name servers (queried from the parent zone) are queried and just the agreed upon results are returned. Analysing the zone file and extracting the dependencies between the different resource records and their corresponding data layer elements in the model, is done in this step.

2. **Step2 [Lines 7-17]**: Physical elements (servers, geographical locations, networks, and organisations) with their attributes are constructed based on the information extracted from the chain of authoritative name servers and organizations involved in the resolving process of domains under that particular zone (Z). All types of dependencies and recursive queries are followed to get the full dependency graph of the zone in the three operational planes. Certain utilities are being used such as MaxMind [45] GeoIP database system for extracting geographical locations associated with server IP address, Team Cymru's [46] WHOIS querying system and BGP Toolkit [47] for IP addresses, BGP Prefixes and ASN Numbers.

3. **Step3 [Lines 18-23]:** DG instance model is built based on a predefined template that ensures that it is conforming to the DNS model. When the model instance file is imported to the Eclipse runtime IDE, model validation is done to ensure the correctness of the model instance generated.

Algorithm 1 Dependency Graph (Model Instance) Generation Algorithm.

Require: Z: Zone name
1: XMI ← ∅ /* Initialize the array representing the output file
2: RNS = Pick One of the ROOT DNS Servers (a,b,c ... m.root-servers.net)
3: Get SOA Record for Zone Z /Dig Command [dig +nocmd @RNS $domain SOA +noall +answer]
4: PNS ← Primary Name Server /* Extract the Primary Name Server from the SOA Record Data Elements
5: Query PNS for Authoritative Name Servers List of Zone (Z) [Dig Command [dig +nocmd @PNZ Z NS +noall +authority +answer]
6: S = List of Host Names of all Authoritative Name Servers (ANS) for the Zone(Z)
7: **for** NS ← each member of S **do**
8: Query PNS for ANS Records of Zone(Z)
9: **if** S_i **then** /* Only Live and Authoritative Servers are Queried
10: S ← S∪S_i
11: S ← Unique Ordered List(S)
12: **end if**
13: Initialize Arrays(Zones, Servers, Geos, Nets and Orgs)
14: **for** NS ← each member of S **do**
15: Get All(Z) /Get lists of (Zones, Servers, Orgs, Geos and Nets)
16: **end for**
17: **end for**
18: Build Arrays of (Zones, Servers, Orgs, Geos and Nets) with Unique Keys Reference
19: Build Tree(Z) /* Delegation Tree of Zone(Z)
20: Build CL=ControlLayer(Z)
21: Build DL=DataLayer(Z)
22: Build ML=ManagementLayer(Z)
23: XMI=concatenate (Header,CL,DL,ML,Footer)
24: **return** Xmi file containing the complete DG for Zone(Z)

6.2 ISDR Techniques

The ISDR method is implemented in two steps:

1. **Techniques' Specifications:** which include the specification of metrics, smells and refactorings defined over the DNS Model which is being modelled as an

EMF E-core Model. The techniques are specified using the *Specification Wizard* available through the *Specification Module* within EMF Refactor Framework as shown in Figure 6.1.

2. **Techniques' Application:** which includes all the steps needed for the application of the specified techniques on instances of the DNS model (i.e. Dependency Graphs) generated by the DGBuilder tool. This is done through the various extension points available through the *Custom QA Plugin* component as shown in Figure 6.1.

Throughout our implementation, we have two running instances of Eclipse as shown in Figure 6.2. In the first instance, called the "Modelling IDE", we defined the model and generated code from it. The second instance, called the "Runtime IDE", is started from the Modelling IDE and contains instances of the generated model and the plugin projects where metrics, smells and refactorings are specified and then applied on a particular model instance.

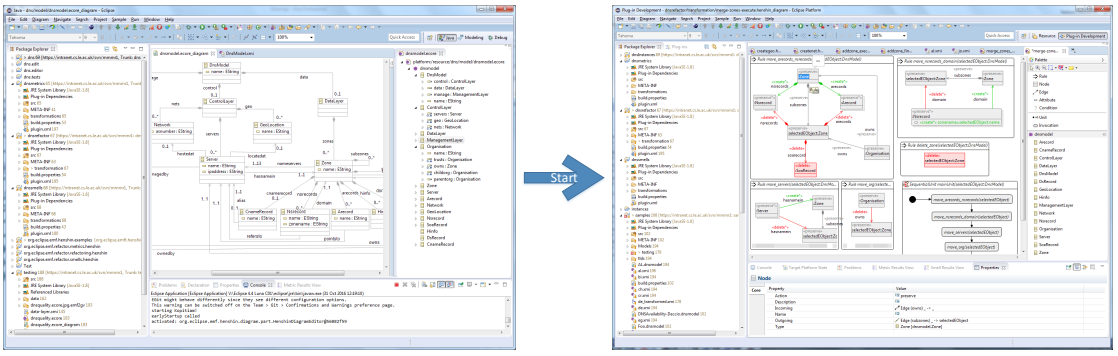


FIGURE 6.2: ISDR Method Specification and Application Environments.

In order to specify the various techniques (metrics, smells and refactorings) to implement our ISDR method, we define several plug-in projects within the Eclipse *Runtime IDE* instance and then import them back within the *Modelling IDE* environment. Deploying the model as a plugin registers the generated model and enable

us to use it in defining the new techniques. Three different plug-in projects were created to hold the definitions and source code of the various techniques as follows:

- *DNSMetrics* to include all metrics definition including Java, OCL and Henshin-based metrics.
- *DNSSmells* to hold all smells definitions including measurable and structural ones.
- *DNSRefactor* to hold all refactoring specifications including any initial, final and execution rules.

In order to provide more details about the specification process of each technique, we will use the DNSSEC zone complexity and associated techniques as an example of our implementation of the ISDR method.

6.2.1 Techniques' Specification

6.2.1.1 Metrics

The disregard for DNS as well as DNSSEC [25, 26] maintenance can result in increased failure potential. One important necessity is careful coordination between zone administrators and system managers both hierarchically (i.e., between parent and child zones) and laterally, between organizations hosting each other's zone data (i.e, between name servers operators). The hierarchical relationship is the most crucial part of the DNSSEC data plane, since the chain of trust extends vertically, and a break in the chain results in general failure to the whole name space below. However, this coordination is less demanding because it generally involves only two entities. Problems caused by lateral coordination may be less severe since the zone will usually have multiple name servers.

TABLE 6.1: Metric Hierarchical Reduction Potential (HRP) Interpretation Model.

Metric	Hierarchical Reduction Potential (HRP)
Definition	Quantifies how much the ancestry of a zone might be reasonably consolidated to reduce hierarchical complexity.
Usability	A greater HRP value indicates that minimizing hierarchical complexity might reduce failure potential.
How to Measure	We express the HRP of zone z, having m + 1 ancestral zones, as the fraction of layers that could be reduced if the number of zones is consolidated to m' + 1
Example	While delegation is necessary in many cases, there are some cases in which collapsing a delegated zone is both reasonable and possible. For example, if example.com and sub.example.com are two zones administered by the same organization, the zone data for sub.example.com might trivially be migrated to the example.com zone and the delegation to sub.example.com removed. This consolidation reduces the number of zones ancestral to sub.example.com by 0.25 from 4 to 3.
Range	$1 < HRP \leq 0$
Formula	$HRP(z) = \frac{m-m'}{m+1}$

There are two metrics used to quantify the complexity of a DNS zone. The metrics themselves are calculated independent of DNSSEC deployment, but higher metric values may increase the failure potential for signed zones because they indicate more areas where problems may occur.

The first metric is the *Hierarchical Reduction Potential (HRP)* [93], which quantifies how much the ancestry of a zone might be consolidated to reduce hierarchical complexity. The second metric is *Administrative Complexity* [7] which describes the diversity of a zone, with respect to organizations administering its authoritative name servers. The interpretation model of the *HRP* metric is shown in Table 6.1 while the interpretation model of the *Administrative Complexity* metric is shown in Table 4.2. In order to measure the HRP, several other metrics have to be calculated first and they are:

1. *Zone Depth:* The depth of a zone is measured by its distance from the root

zone. For example, zone z has ancestry $z(0), z(1), ..., z(m)$ comprised of $m+1$ zones and has a depth of m. Each ancestral zone $z(i)$ contributes to the failure potential for zone z, as it is an additional requirement of DNSSEC correctness that must be consistent. Listing 6.1 shows how the zone depth metric is being specified in the DNSMetrics plugin project using Java.

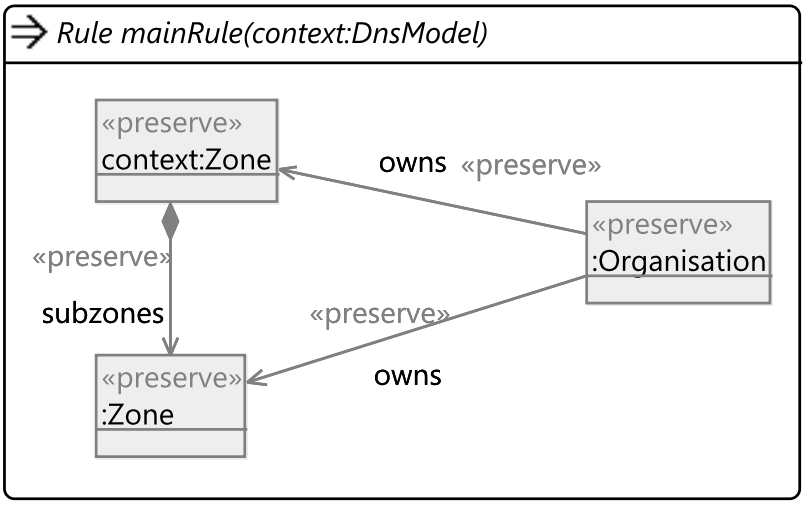


FIGURE 6.3: Henshin Rule for Calculating the HRPD Metric of a Zone.

2. *Hierarchical Reduction Potential Domains (HRPD):* In order to measure the number of consolidated zones that can be achieved by merging two zones with parent-child relationship if they both are being managed (owned) by the same organisation, we use the Hierarchical Reduction Potential Domains (HRPD) metric, which shows the number of subzones that can be consolidated in the parent zone. The metric is defined in the DnsMetrics plugin project using Henshin rule as shown in Figure 6.3.

```
/*
 * Java Specification of the Zone Depth Metric.
 */
package org.eclipse.emf.refactor.metrics;
import java.util.List;
import org.eclipse.emf.ecore.EObject;
import org.eclipse.emf.refactor.metrics.interfaces.IMetricCalculator;
public final class DEPTH implements IMetricCalculator {
```

```
        private List<EObject> context;
        @Override
        public void setContext(List<EObject> context) {
                this.context=context;
        }
        @Override
        public double calculate() {
                dnsmodel.Zone in = (dnsmodel.Zone) context.get(0);
                double ret = 1.0;
                 while (in.getParentzone() !=null) {
                        ret++;
                    in=in.getParentzone();
                   }
                return ret;
        }
}
```

LISTING 6.1: Zone Depth Specification in Java

Finally, the *Hierarchical Reduction Potential (HRP)* of the zone can be calculated using the corresponding formula in its interpretation model, shown in Table 6.1, as a compositional metric. The generated Java source code for the metric is shown below.

```
/*
 * Calculating the HPR Metric as a Compositional Metric Using Java.
 */
package org.eclipse.emf.refactor.metrics;
import java.util.List;
import org.eclipse.emf.ecore.EObject;
import org.eclipse.emf.refactor.metrics.interfaces.IMetricCalculator;
import org.eclipse.emf.refactor.metrics.interfaces.IOperation;
import org.eclipse.emf.refactor.metrics.core.Metric;
import org.eclipse.emf.refactor.metrics.operations.Operations;
public final class HRP implements IMetricCalculator {
        private List<EObject> context;
        private String metricID1 = "dnsmetrics.hrpmd";
        private String metricID2 = "dnsmetrics.depth";
        IOperation operation = Operations.getOperation("Division");
        @Override
        public void setContext(List<EObject> context) {
                this.context = context;
        }
        @Override
        public double calculate() {
                Metric metric1 = Metric.getMetricInstanceFromId(metricID1);
                Metric metric2 = Metric.getMetricInstanceFromId(metricID2);
                IMetricCalculator calc1 = metric1.getCalculateClass();
                IMetricCalculator calc2 = metric2.getCalculateClass();
                calc1.setContext(this.context);
                calc2.setContext(this.context);
                return operation.calculate(calc1.calculate(),calc2.calculate());
```

```
        }
}
```

LISTING 6.2: Generated Code for Calculating HPR (Metric Composition)

6.2.1.2 Bad Smells Specification

Based on the taxonomy developed in Chapter 5, bad smells can be classified according to their type (Lexical, Measurable or Structural). In this section, we present examples of the three categories of bad smells and how they are being specified within the EMF Refactor Framework.

- **Measurable Bad Smells:** Measurable bad smells such as the *Excessive Zone Complexity* smell can be specified through the definition of a smell in the DNSSmells plugin project. The smell is specified based on the measured value of the HRP metric of a certain model instance of a particular zone. A threshold value for the metric (i.e $HRP > 1$) can be set by the system administrator as the basis for the detection of such a bad smell. A value of HRP greater than the threshold value indicates that there is a high probability of failure potential (presence of the bad smell). Setting a certain threshold value for the metric HRP is done through a project specific configuration page based on the zone administrator's needs and specific constraints.

- **Lexical Bad Smells:** A lexical property relates to the vocabulary used to name a zone, server, or a resource record. Giving the same server different names, ill-formed resource records, the presence of unnecessary resource records within a zone and duplicate resource records for certain hosts are examples of such smells. Lexical smells identification and detection is straightforward and rely mainly on text-based best practices or guidelines recommendations analysis.

- **Structural Bad Smells:** A *Cyclic Dependency Smell* [32] occurs when two or more zones depend on each other in a circular way. As shown in ISDR method validation case study in Section 5.3, checking each zone individually for configuration errors may not lead to the detection of this bad smell since they may both be configured correctly. On the other hand, constructing the dependency graph will easily show the occurrence of circular paths that identify the smell. Structural bad smells can be specified directly using a Henshin rule.

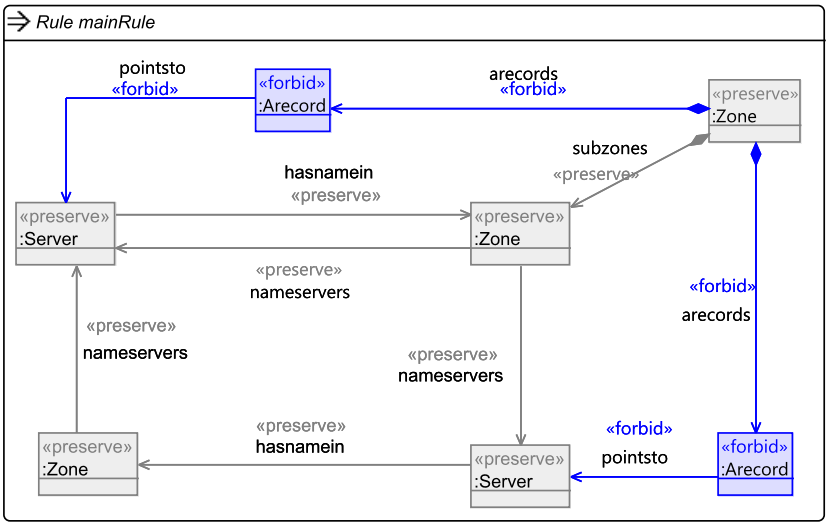


FIGURE 6.4: Specification of the Bad Smell Cycling Dependency Using Henshin.

Figure 6.4 shows the Henshin rule to check for any cycle patterns within the model instance that will reflect the presence of the *Cycling Dependency* bad smell. The pattern specifies two zones with parent/child relationship that must be found in the model (tagged by «preserve») and a cyclic dependency of ($Zone_1 \xrightarrow{nameservers} Server_1 \xrightarrow{hasnamein} Zone_2 \xrightarrow{nameservers} Server_2 \xrightarrow{hasnaemin} Zone_1$) sequence path (tagged by «preserve» tags). The *Negative Application Condition (NAC)* checks for the absence of any *ARecord* for the *out-bailiwick* name servers of a *Zone* in the parent *Zone* (tagged by «forbid» tags). It also excludes any *ARecords* for *in-bailiwick* servers from the particular zone to prevent counting of these occurrences as occurrences of cyclic dependency

smells.

The smell detection tool in EMF Refactor uses Henshin's pattern matching algorithm to detect rule matches. The pattern rule must be named mainRule in order to be executed by the Henshin adapter.

6.2.1.3 Refactorings

EMF Refactor supports three concrete mechanisms for EMF model refactoring specification. As for metrics and smells, refactorings can be specified using Java and the language API generated by EMF. A direct way to specify a model refactoring straight forwardly is to use Henshin. A concrete refactoring specification in Henshin requires up to three parts (i.e., specifications for initial checks, final checks, and the proper refactoring execution rules).

1. **Initial Check:** The initial checks ensure that all preconditions are met before executing the actual refactoring. The checks are applied on the rule's contextual model element *selectedEObject* and related components present in the refactoring rule(s). Here, each conflicting situation is defined by a rule pattern using the abstract syntax of the underlying DNS model. Furthermore, parameters in the main checking unit must be equally named to the corresponding ones in the main execution unit. All these checks are implemented separately and executed through a so called *Independent Unit*. Independent Units have an arbitrary number of sub-units that are checked in non-deterministic order for execution.

   For example, in order for the bad smell *Cyclic Dependency* to be rectified, resource records of RRType-A *(ARecord)* model element should be created in the parent zone for out-of-bailiwick name servers to make sure that their names are resolvable in all cases. A refactoring rule named *CreatARecord* is used to

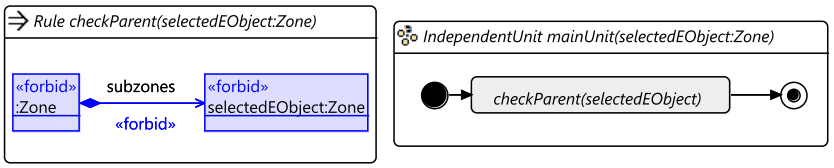


FIGURE 6.5: Specification of Initial Checks for CreateARecord Refactoring.

remedy for this bad smell and Figure 6.5 shows Henshin rule specifying the initial checks of this refactoring. Rule *checkParent* checks whether the selected zone has a parent zone included in the model. The absence of a parent zone is modelled using tags «forbid». Such precondition check rules are contained in a Henshin Independent Unit (called *mainUnit*) to be executed. If any of the rules can't be applied, Henshin uses the rule's description value to provide a detailed error message to the system administrator.

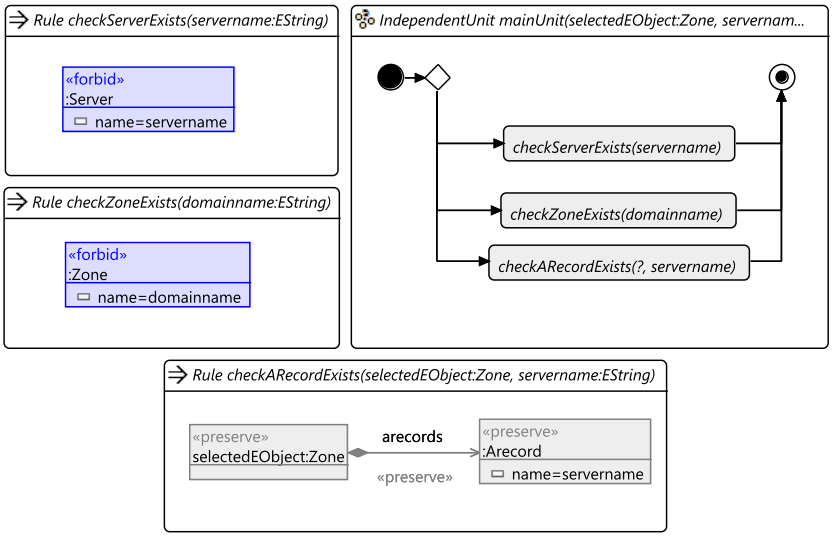


FIGURE 6.6: Specification of Final Checks for CreateARecord Refactoring.

2. **Final Check:** Final checks are applied on the selected model contextual element and check the applicability of the rule parameters and references. In our *CreateARecord* example, there are three final conditions that have to be checked:

- First, there must be a server with the user specified name (*servername*) that is already exists as a physical server element in the *ControlLayer* of the DNS instance model. The rule pattern for the absence of such a class is shown in rule *CheckServerExists* in Figure 6.6 using the «forbid» tag.
- The second precondition that has to be checked is specified by rule *checkZoneExists*. Besides the already known parameters *selectedEObject* and *servername*, this rule has another parameter, *domainname*. The rule checks whether there exists a zone with a name equals to the *domainname* parameter for the *ARecord* to be created.
- The third and final precondition to be checked is whether there already exists an *ARecord* model element with the same specified name in the zone and this is accomplished through the rule *CheckARecordExists*.

If all of these checks are passed successfully, Henshin executes the main refactoring execution unit and another model is automatically created.

3. **Refactoring Rule Execution:** The specified execution rule performs the actual model change, i.e., it creates one or more *ARecords* in the parent zone of the specified contextual element of the model. As it is the case with all Henshin refactorings, the actual execution is again packaged into a so called *Sequential Unit*. A sequential unit has an arbitrary number of sub-units that are executed in the given order. In this case, the unit is configured to fail if not all specified rules can be executed and in case of a failure all changes will be automatically undone.

6.2.2 Techniques' Application

To calculate relevant metrics, detect bad smells and apply refactorings, the tool environment supports a project-specific configuration for all these techniques.

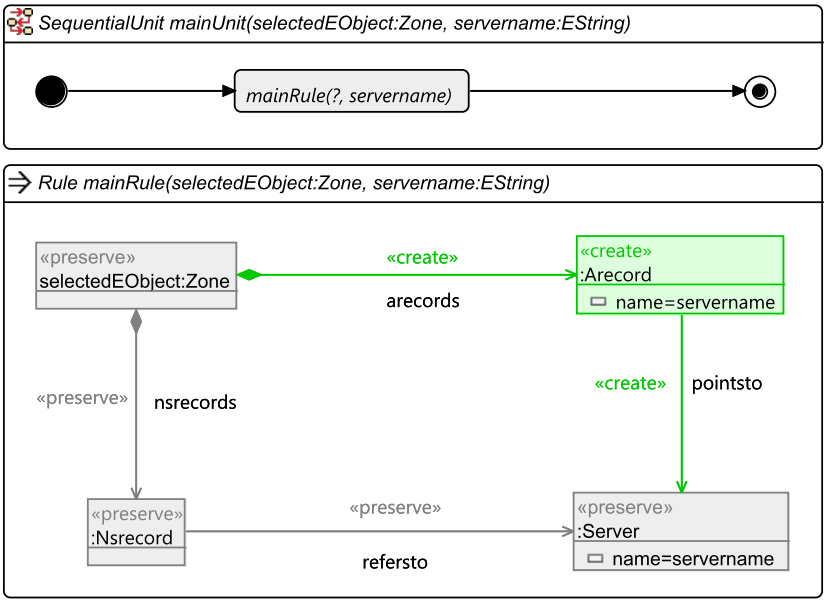


FIGURE 6.7: Execution Unit for CreateARecord Refactoring.

6.2.2.1 Metrics Calculation

The metrics configuration is managed by means of a dedicated project properties page. On this page, all existing model metrics for the DNS Model are listed. They are structured with respect to the corresponding element type the metrics are calculated on (the context). Through this configuration page, we can activate all model metrics or a partial list of them. Some of the metrics activated in this example are: for the DNSModel: (Zones, Servers, Orgs and AS), for the Zone contextual element: *(ANS, Depth, HRP and GeoD)*, and for the SOA record timers: *(Expiry, Retry, minTTL and Refresh)*.

The calculation of metrics on a specific model element is started from its context menu. The metrics can be calculated on a specific model element or transitively for all elements of the model. Relevant metrics are calculated based on the context of metric defined earlier in the metric specification module.

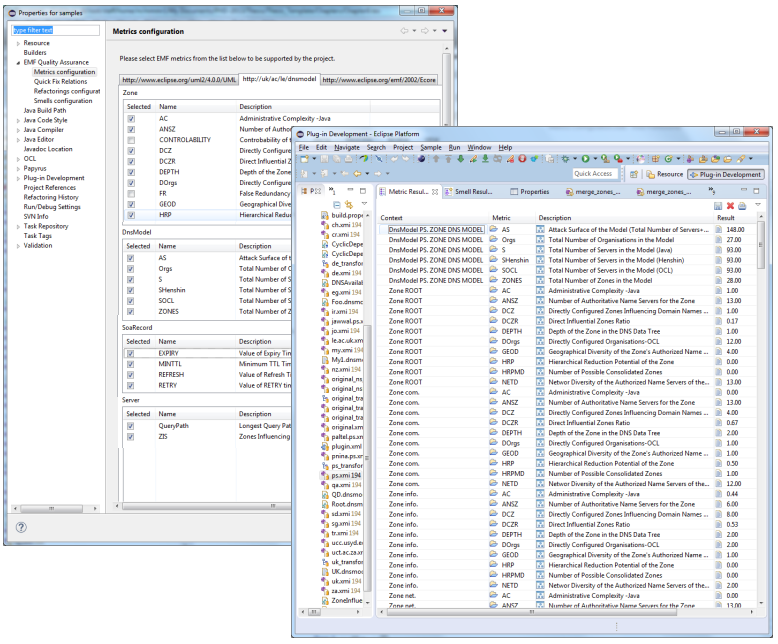


FIGURE 6.8: Metric Configuration Page and the Calculation of these Metrics for the DnsModel of Zone (.PS)

Figure 6.8 shows the project-specific configuration page as well as the calculated results on all elements of the DnsModel instance (the Dependency Graph) of the .PS (Palestine) Country-Code Top-Level-Domain (ccTLD). The results view shows that the model has an Attack Surface metric (AS) of 148 which shows all elements (Servers, Zones and Organizations) involved in the domain resolution process of the specified zone. For each zone, further metrics are shown such as the number of authoritative name servers, direct and third-party zones influencing the resolution of domain names under the (.PS) zone and so on.

6.2.2.2 Bad Smells Detection

The discussion of metrics results shows that a manual interpretation of metric values seems to be unsatisfactory and error-prone. So, another static model analysis technique is required, more precisely an automatic detection of model smells. As for model metrics, the tool provides a configuration page of specific model smells that

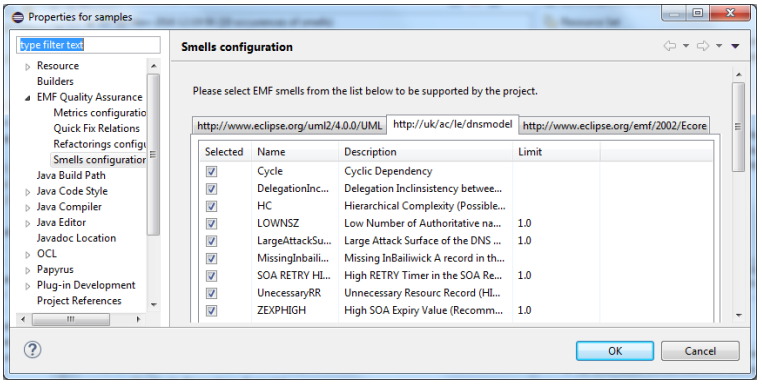


FIGURE 6.9: Smells Configuration Page for the DnsModel.

are relevant for the current project. Figure 6.9 shows the configuration dialogue listing the system-known model smells with respect to some DNS Model instances. For a metric-based model smell, a corresponding threshold (Limit) can be configured.

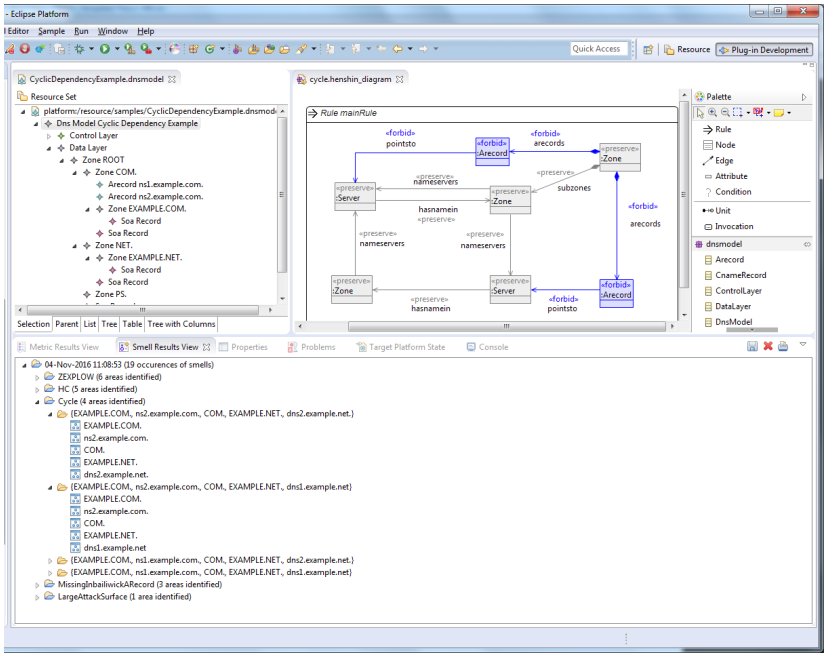


FIGURE 6.10: Detection of Cycling Dependency Using Henshin Rule.

The matches found represent the existence of model smells in the instance model. The smell detection tool provides a highlighting mechanism for all involved model elements in a bad smell occurrence within the standard tree-based instance editor.

Those model elements are highlighted in order for the system administrator to easily spot them. A screen shot of the detection of the *Cyclic Dependency* bad smell is shown in Figure 6.10.

6.2.2.3 Refactorings

Besides manual changes, model refactoring is the technique of choice to eliminate occurring smells. The next step during the ISDR application is to interpret the results of the smell detection analysis and decide on potential correction mechanisms in the form of graph-based refactoring rules that can be used to remedy or eliminate those smells.

The tool provides mechanisms to provide DNS system administrators with a quick and easy way (1) to erase DNS model smells by automatically suggesting appropriate model refactorings, and (2) to get warnings in cases where new model smells occur due to applying a model refactoring. After invoking a refactoring, refactoring-specific basic conditions are checked (initial precondition check). Then, the user has to set all needed parameters to execute the refactoring where final checks are applied (final precondition check) before executing the final step of the refactoring application process. Figure 6.11 shows the work flow of the various steps used to apply the refactoring on a certain model.

In order to propose suitable refactorings respectively to inform about potential new smells, the tool must be provided with information on the relations between model smells and model refactorings. Given a concrete model smell occurrence, several refactorings can be suitable to erase it. The tool environment provides the ability to configure this relationship between model smells and model refactorings.

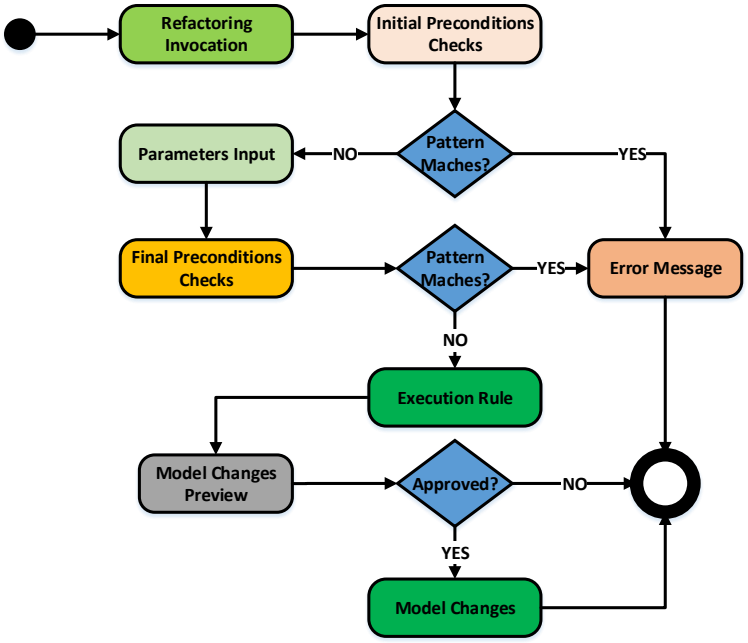


FIGURE 6.11: Refactoring Execution Workflow.

Figure 6.12 shows the property page called the Quick Fix Relations page for (de-)activating appropriate relations. A pragmatic way is to manually define these relations. Here, the advantage is that DNS Zone managers and system operators can adjust the implementation of model smells and model refactorings to the fact that they are going to be related. A manually defined relation is done by the system administrator with the definitive goal to erase a model smell using a given model refactoring.

The application of a certain model refactoring can be triggered by using two alternative ways:

- First, it can be invoked from within the context menu of at least one model element in the standard tree-based EMF instance editor. Dependent on the selected element(s), only those refactorings are provided in the menu being defined for the corresponding model element type(s).

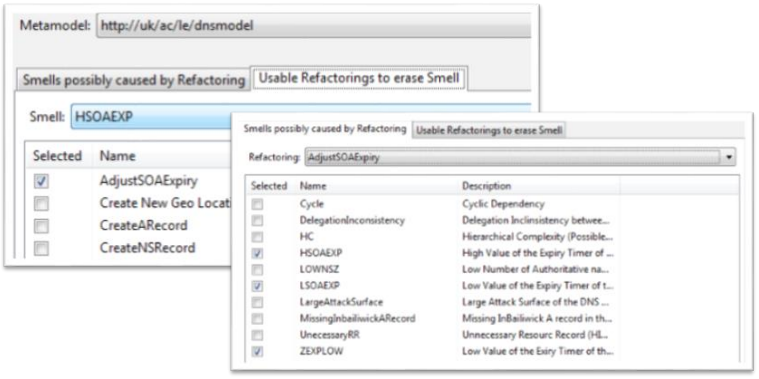


FIGURE 6.12: Quick Fix Matrix Configuration Page.

- The second way to trigger a model refactoring is to use the quick fix mechanism of the smell results view. Starting from this view, our tool environment provides a suggestion for potential refactorings according to pre-defined smell-refactoring relations and a dynamic analysis of applicable model refactorings.

Chapter 7

DNS Model Transformation

The chapter starts with an overview of the refactoring implementation tool used to implement the DNS model transformation followed by investigating the behaviour preservation properties of the proposed refactorings. Refactoring rules' analysis, priority analysis and checks for execution dependability and conflict analysis through the critical-pairs analysis technique are also conducted.

The chapter details the process of integrating the *DNS Metrics Suite, Quality Prediction Models* and the *ISDR Method* into a DNS model transformation advisory tool and then proceeds further by giving a concrete example of DNS model transformation using the developed tool.

The tool presents the DNS system administrators with refactoring opportunities to remedy for bad smells detected in their zone configurations and name servers' deployments. The tool also provide the administrators with recommendations where the administrator have full control on the decision to implement any of these refactorings or recommendations based on his/her special DNS configurations and deployment considerations.

7.1 Model Refactoring

Model-driven engineering (MDE) is a discipline in software engineering that relies on models as first class entities and that aims to develop, maintain and evolve software by performing model transformations. Based on the language in which the source and target models of a transformation are expressed, a distinction can be made between endogenous and exogenous transformations. Endogenous transformations are transformations between models expressed in the same language. Exogenous transformations [44] are transformations between models expressed using different languages. Typical examples of endogenous transformation are:

- Model Refactoring, a change to the internal structure of a model to improve quality characteristics of the system without changing its observable behaviour. The graph-based rules defined over the DNS model and proposed as part of the ISDR method are examples of such refactorings.
- Model Optimization, a transformation aimed to improve certain operational qualities (e.g., performance) of the system.

Refactoring implementation tools can be classified based on their degree of automation: Manual, Semi-Automated and Fully-Automated. A fully-automated tool provides automatic detection and correction of design defects without user intervention. Semi-automated tools require interaction with the user throughout the refactoring process. Semi-automated tools assist the user by proposing refactoring opportunities and their suggested solutions; however, the decision to perform the actual transformation is left to the user. Manual refactoring tools leave the process of model smell detection and application decision to the user completely.

In our approach, we follow the semi-automated approach. Our tool propose to the DNS system administrators several refactoring opportunities to remedy for bad

smells detected in their zone configurations and provide justified recommendations where the administrators have full control on the decision to implement any of them based on their local policies, special DNS configurations and deployment considerations.

7.1.1 Behaviour Preservation

A transformation is behaviour-preserving if the explicit or implicit constraints of the behaviour in the source model remain fulfilled in the target model after the transformation has been executed. Defining the notion of behaviour preservation can be done in many ways. Most researchers [94] agree that a full guarantee on preservation of behaviour is impossible. Therefore they use a relaxed notion of behaviour preservation, demanding that the program/system will perform the same actions before and after executing the refactoring. For each refactoring, one may list behaviour-related properties that need to be preserved, and that can be verified statically [95].

At its core DNS is a simple protocol with requests and responses each generally contained in a single UDP packet. Further, resolving a hostname requires only a small number of transactions. The simple protocol and process, however, belies much complexity in the modern DNS ecosystem. A DNS request triggered by a user clicking a link in a web browser may now travel through multiple layers of DNS resolvers or public DNS service providers. Therefore, while the DNS protocol is itself simple, much of the resolution process and the actors involved are largely hidden from view [96].

Since our DNS model is built based on the authoritative zone managers' and system administrators' point of view, we are concerned about refactoring rules that preserves DNS behaviour as observed from those vantage points. We only look at

notions of behaviour preservation that can be detected statically and do not rely on sophisticated data- and control-flow analysis or type inferencing techniques. This restriction to the static structure of the model is important because DNS configuration and deployment structure are all that our refactoring tool may operate on.

In order to achieve this, we need a precise definition of DNS "behaviour" in general, and for DNS dependency model in particular. We consider three types of DNS behaviour preservation, based on the fact that they are important and non-trivial for the selected refactorings. From a graph-based point of view, all refactorings are defined over the DNS Dependency Model so each and every component in the model instance conforms with its type as defined within the reference model and the transformed dependency graph is still a valid instance of the DNS model. A refactoring is behaviour preserving if the following implications are true:

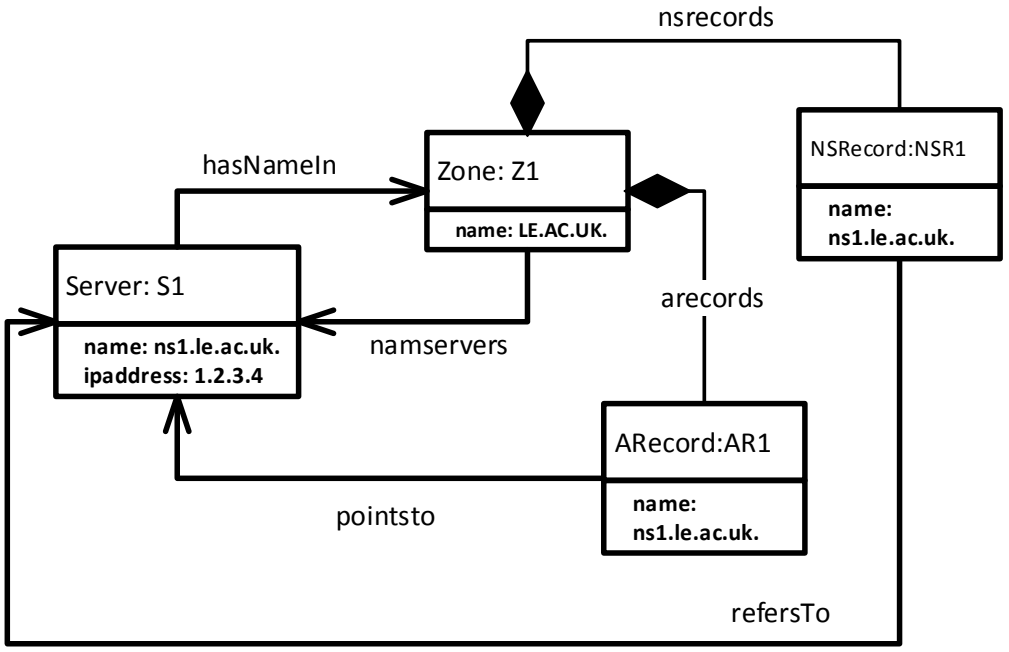


FIGURE 7.1: Instance Graph of Binding-Preserving Property.

- Query Resolution Preserving: If for all sets of names resolution is correct in the source, it should be so in the target. This means that for any set of input domain names submitted to a name server through a graph-based query on the model, the produced set of Authoritative Resource Records (SOA record, NSRecord and related ARecords) returned from the concerned Zone are correct and valid records (based on the DNS protocol specifications) before and after applying the refactoring. These resource records do not need to be the same since some of the refactorings may modify their attributes and/or associations.

- Binding Preserving: The DNS specifications calls for any zone to hold the authoritative binding between IP addresses and hostnames for its own name servers (in-bailiwick name servers). A refactoring is binding-preserving if for each physical or logical name *server:S1* in the refactoring, there is a corresponding *(ARecord:AR1)and NSRecord* within (*arecords* containment relation to) the *Zone:Z1* that holds the name of (with *hasNameIn* relation to) and *pointsto* and *refersto* that particular *server:S1* respectively.

7.1.2 Analysis of Model Refactoring Rules

Deciding what to refactor and which refactoring to apply still remains a difficult manual process, due to the many dependencies and interrelationships between relevant refactorings. In order to solve this problem, two analysis techniques are applied to help the DNS system administrator make an informed decision of which refactoring is most suitable in a given context and why.

7.1.2.1 Conflicts and Dependencies

Graph transformation theory allows us to compute conflicts and dependencies of transformations by relying on the idea of critical pair analysis [97]. Critical pair

analysis is known from term rewriting and can be used to check if a rewriting system contains conflicting computations.

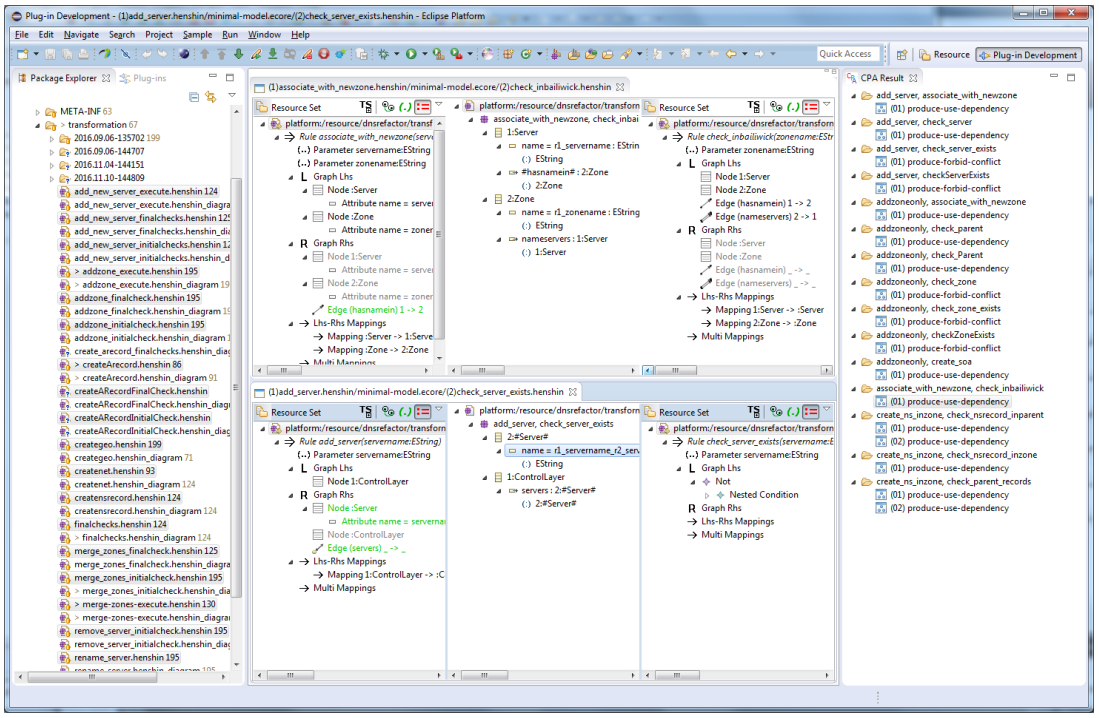


FIGURE 7.2: Critical Pairs Analysis.

Critical pairs formalize the idea of showing a conflicting situation in a minimal context. From the set of all critical pairs we can extract the objects and links which cause conflicts or dependencies. The reasons why graph rules can be in conflict are threefold:

1. One rule application deletes a graph object (i.e., a node or edge) which is in the match of another rule application (delete-use conflict).

2. One rule application produces graph objects that give rise to a graph structure that is forbidden by a NAC of another rule application (produce-forbid conflict).

3. One rule application changes attributes being in the match of another rule.

The graph transformation tool, Henshin, recently provided an algorithm implementing this analysis. Henshin also provides several refactoring execution control-flow mechanisms (called *units*) to resolve any *produce-use-dependancy* issue between the rules. Figure 7.2 shows the results of running the critical pairs analysis on the set of refactoring rules defined over the DNS model.

To illustrate the usefulness of these analysis in the context of the DNS refactorings, let's take the bad smell *Excessive Zone Influence* as an example. Figure 7.3 shows that several refactorings can be applied to eliminate this bad smell. Within the priority of execution (2), we have the two refactorings *MergeZone* and *Delete Zone*.

Refactoring *MergeZone* can not be executed if the relevant zones to be merged happen to include a zone that already has been deleted by the DeletZone refactoring so these two refactorings are in conflict. To resolve such situation we include a *sequential unit* within the MergeZone refactoring specifications to check for the existence of the zones to be merged before executing the refactorings on the model instance. Another issue is the dependency of the *DeleteZone* refactoring on the *DeleteA/NS/DS/Records* refactorings.

A zone can't be deleted unless all its resource records are deleted or moved to the new parent zone. Sequential units controls the execution of the different refactorings and contributes to resolving these issues too.

7.1.2.2 Execution Scope and Priorities

It should be noted that a refactoring rule is just one of the options to eliminate a bad smell. It can also take more than one rule application to resolve the situation, so a single rule specifies an incremental improvement, which may have to be repeated or combined with others. For example, to eliminate the *cyclic dependency* bad smell there could be another rule for creating a new server under another external zone

rather than adding a "glue" (ARecord) for out-of-bailiwick servers. When deciding on the scope and priorities of executing the applicable and non-conflicting refactoring rules the following general guidelines are applied :

	Refactorings Number ->	1	2	3	4	5,6	7,8	9	10,11	12	13	14	15	16	17	18	19	20	21	22
Bad Smell Number	Bad Smell / Refactoring Rule	AddNewNet	DeleteNet	AddNewGeoLocation	DeleteGeoLocation	Create/Delete A Record	Create/Delete NS Record	Delete Unnecessary Records	Create/Delete DS Record	CreateSOARecord	DeleteSOARecord	Adjust SOA Parameter	Create new zone	Merge Zones	Delete Zone	Rename Server	Add New server	Delete Server	Move Server Location	Modify Server IP
1	Information Leakage							X												
2,3	Zone Drift/Thrush											X								
4	Ill-Formed RRs					X	X			X	X	X								
5	Missing RR					X	X			X										
6	Incorrect Parameter Data						X					X								
7	Ambiguous Data		X		X	X	X			X	X			X	X	X		X		
8	Absence of Multiple RRS		X		X	X	X											X		
9	Invalid Trust Anchor								X											
10	Untrusted-Peer Organisation															X	X	X		
11	Large Attack Surface		X		X	X	X						X	X	X	X	X	X		
12	Excessive Zone Influence		X		X	X	X			X	X			X	X	X		X		
13	False Redundancy	X		X		X	X										X	X		
14	Diminished Redundancy	X	X	X	X	X													X	X
15	Administrative Complexity										X									
16	Cyclic Dependency					X														
17	Non-Optimal Query Path		X		X	X	X			X	X				X	X		X		
18	Delegation Inconsistency					X	X													
	Scope	Primitive (Supporting) Rules				Zone Admin's Own Decisions							Coordination with Other Zones' Admins			Cost and Access Permissions Considerations				
	Priority	0				1							2			3				

FIGURE 7.3: Refactoring Rules Execution Scope and Priorities.

1. Rules related to modifications within the zone administrator's own administrative domain (i.e resource records modifications within the same zone) are first to execute since they are the easier and the most cost effective way to remedy or eliminate a bad smell.

2. Rules that need coordination with other zones' administrators without any additional cost or resource utilisation are executed next. Examples of such rules

are those who modify attributes of external servers hosted at organisations where their exists previous mutual free or service-level agreements.

3. Finally, rules that have overhead regarding resource utilisation, cost, new service agreements and access permissions considerations are executed next.

There are four refactoring rules that have been classified as primitive or supporting rules since they are just needed to support the execution of other rules such as creating or deleting a network AS number or server geographical location. Figure 7.3 shows how the proposed refactorings are prioritised based on the above mentioned guidelines.

7.1.3 Quality Impacts of Model Refactorings

An established way of evaluating the impact of refactorings on the quality attributes of a software artefact is to compute metrics on its initial version and on the refactored version [98]. Nevertheless, metrics alone do not provide a clear answer to the question of whether the refactorings improve the quality attributes of the system. For that, it is necessary to find an alignment of metrics to quality attributes, i.e., whether a lower/higher value of a metric improves/worsens a given quality attribute of the operational system. We conducted an empirical study reported in Chapter 4, that helped us to justify the quality improvements of our refactorings (apart from relying on our own experience in the field).

Model Structural Metrics are the key factor here since refactorings change the metrics of the model and hence have a direct effect on the model quality and consequently the perceived system quality attribute. We used the assessment experiment to prove that the DNS model structural metrics can effectively be used as early indicators of the quality of the model and the perceived quality of the system.

7.2 DNS Model Transformation

A model transformation mechanism takes as input a model to transform, the source model, and produces as output another model, the target model. Model transformation can be used as a correction mechanism based on the detection of bad smells that affect certain aspects of the DNS quality attributes. It can also be carried out to improve certain aspects of quality attributes of the original system in addition to eliminating bad smells related to a particular qulaity attribute.

The approach illustrated in Figure 7.4 is a bad-smells-driven DNS model transformation method. The main goal of this method is to remedy or eliminate the existence of bad smells within the model instance (i.e dependency graph). The method takes a certain zone configuration and servers' deployment structure and apply the ISDR method bad smells detection techniques and then applies the correction mechanisms to detected bad smells in the form of graph-based refactorings and finally generate a new zone configuration and name servers layout with associated recommendations. The process is executed in eight steps which are detailed next:

1. **Generation of Initial Dependency Graph.** In this step, the dependency graph is generated using the *DGBuilder* or using the *Example EMF Model Creation Wizard*. The DNS Model instance is generated using the interdependencies extracted from the zone configuration and the authoritative name servers' deployment structure.

2. **Metric Calculation and Prediction Models.** In this step, metrics are calculated on the model instance (using the metrics specification techniques defined as part of the ISDR Implementation) and the prediction models developed as part of the empirical assessment (detailed in Chapter 4 are utilized to get an initial values for the quality attributes ($Q_{Availability}$, $Q_{Security}$, $Q_{Stability}$, $Q_{Resiliency}$).

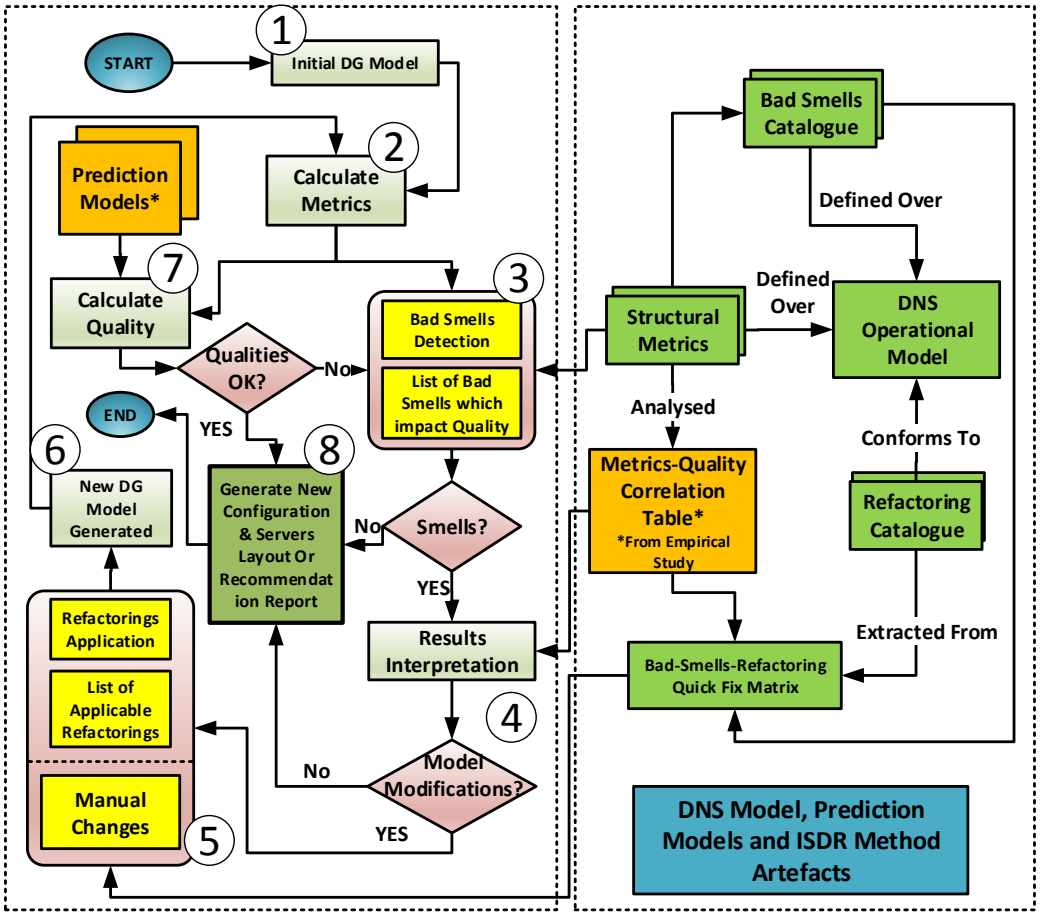


FIGURE 7.4: Bad-Smells-Driven DNS Model Transformation Methodology.

3. **Bad Smells Detection.** In this step, the model instance is checked using the techniques developed as part of the ISDR method implementation to detect any occurrence of a bad smell.

4. **Results Interpretation.** In this step, system administrators manually check the results and interpret them based on their local policies and constrains and decides if a model transformation refactoring is needed.

5. **Model Transformations Refactorings.** In this step, applicable refactorings (defined as part of the ISDR Method techniques) are executed on the initial model instance and the bad smell is rectified.

6. **Generation of The New Dependency Graph.** A new model instance is generated and the two model instances are visually and textually compared. Proposed changes on the initial model instance are displayed to the system administrator.

7. **Predicting Quality Attributes of Newly Generated Model Instance.** In this step, new values for the quality attributes is computed (based on the metrics calculated on the newly generated model instance and fed to the prediction models) and their values are compared with the previously calculated values to see the effect of the refactorings on the overall quality attributes of the system.

   Steps 3-7 are repeated to eliminate other bad smells within the newly generated model instance.

8. **Generation of New Zone Configuration and Servers' Layout.** As a final step in the process and after eliminating all bad smells of the model instance, a new zone configuration and server layout report is generated with recommendations listing what changes need to be done.

7.3 Implementation of the DNS Advisor Prototype

This section describes the implementation of a prototype, that can be both seen as a reference implementation of the methodology, but also a means to evaluate the approach described in this chapter. All the tools used for the prototype were implemented in Eclipse and are used as plugins to this IDE. The tools involved, as well as the artefacts of each step of the ISDR method artefacts are shown in Figure 7.4.

7.3.1 Prototype Architecture

This prototype architecture supports the several steps of the methodology described above and utilises the ISDR method artefacts and techniques as outline in Chapter 6. The final implementation of the DNS advisor dashboard and recommendation plugins is straightforward and was not included in the prototype, and can be part of future work.

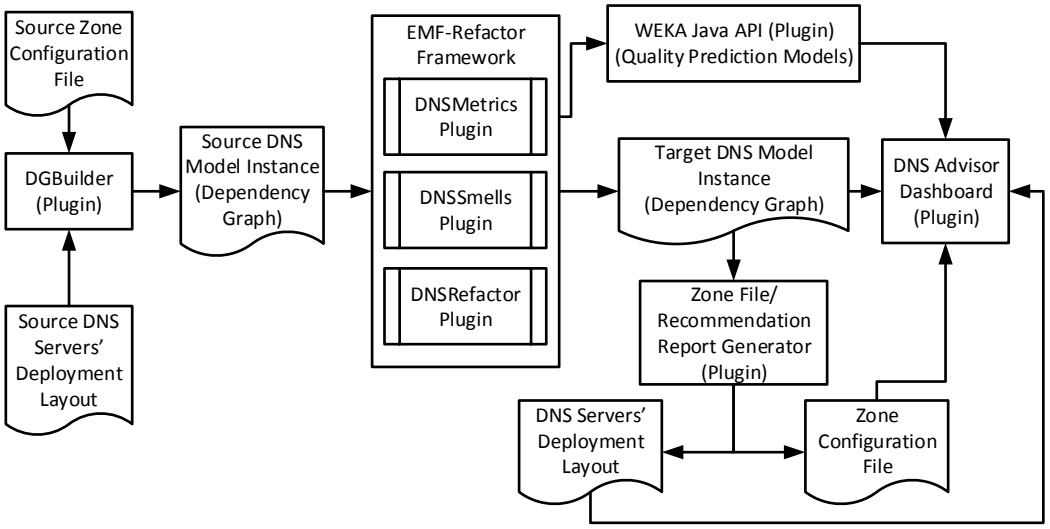


FIGURE 7.5: Prototype Architecture.

Summarising the defined process, the starting point is the source zone file configuration of the concerned zone and its authoritative name servers deployment structure. The DGBuilder takes this information and generate an XMI file representing the DNS model instance (i.e. the source dependency graph). A new target model instance generated after detecting and then eliminating the bad smells of the source model instance and applying the corresponding refactoring rules to generate the target model instance (i.e. the target dependency graph).

The final step is reversing the first one by generating a new zone file configuration out of the target model instance and new name servers layout with a recommendation

report of the required changes. The impacts of the changes on the perceived quality attributes of the model are displayed on the dashboard as a visual guidance to the zone administrators to support their decisions in an informative and meaningful way. Figure 7.5 shows the prototype architecture which is built on top of the Eclipse Modelling Framework.

7.3.2 Prototype Case Study

To validate the implementation of the prototype, the example zone *(EXAMPLE.COM.)* is used.

The implementation process based on the methodology shown in Fiure 7.4 followed the following steps:

1. The initial dependency graph was built using the DGBuilder tool utilising the zone configuration of the zone (EXAMPLE.COM) and its name servers' deployment layout and fed directly as an XMI file to the Eclipse Modelling Platform. The *dependency graph* of the zone is shown in Figure 7.6 and the model tree and textual views of the generated XMI file within the Eclipse IDE are shown in Figure 7.7.

2. The structural metrics of the initial model instance are calculated, and using the prediction models developed in Chapter 4, the initial value of ($Q_{Availability=2}$, $Q_{Security=3}$, $Q_{Stability=2}$, $Q_{Resiliency=2}$).

3. Running the bad smells detection will reveal if there is any improvement to the model in the form of erasing the smell or new smells have been caused by the application of the concerned refactoring. Table 7.1 shows the detected bad smells in the model instance as well as the suggested refactoring to be

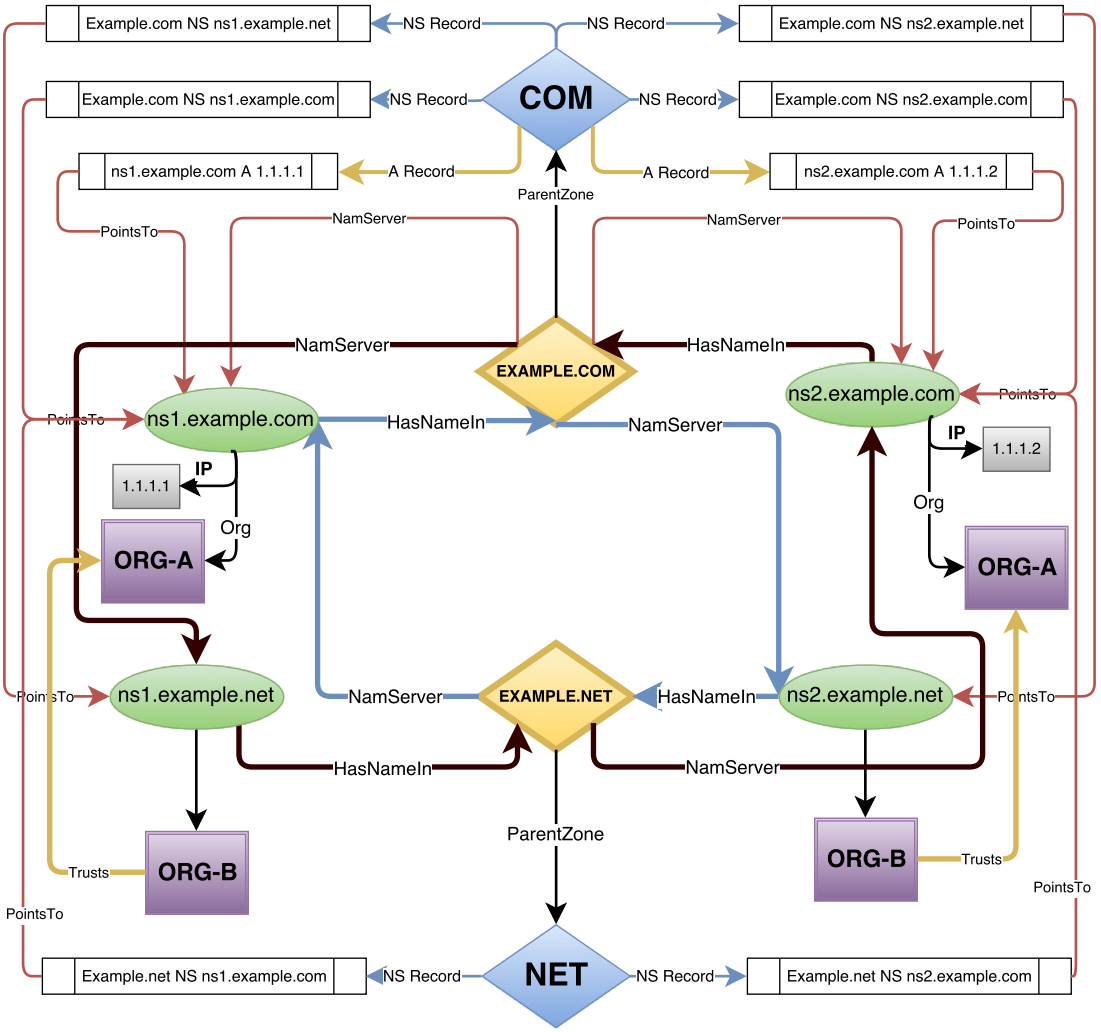


FIGURE 7.6: Example DNS Model Instance (Dependency Graph).

applied to eliminate those bad smells in each iteration of the DNS model transformation process.

4. The system administrator decides to eliminate all four bad smells by applying the corresponding refactorings.

5. The refactorings are applied on the model instance or manual changes are applied to eliminate the currently identified bad smell.

6. After applying the refactoring on the original model instance, the tool generates a new model instance with *_transformed* suffix. The two models can

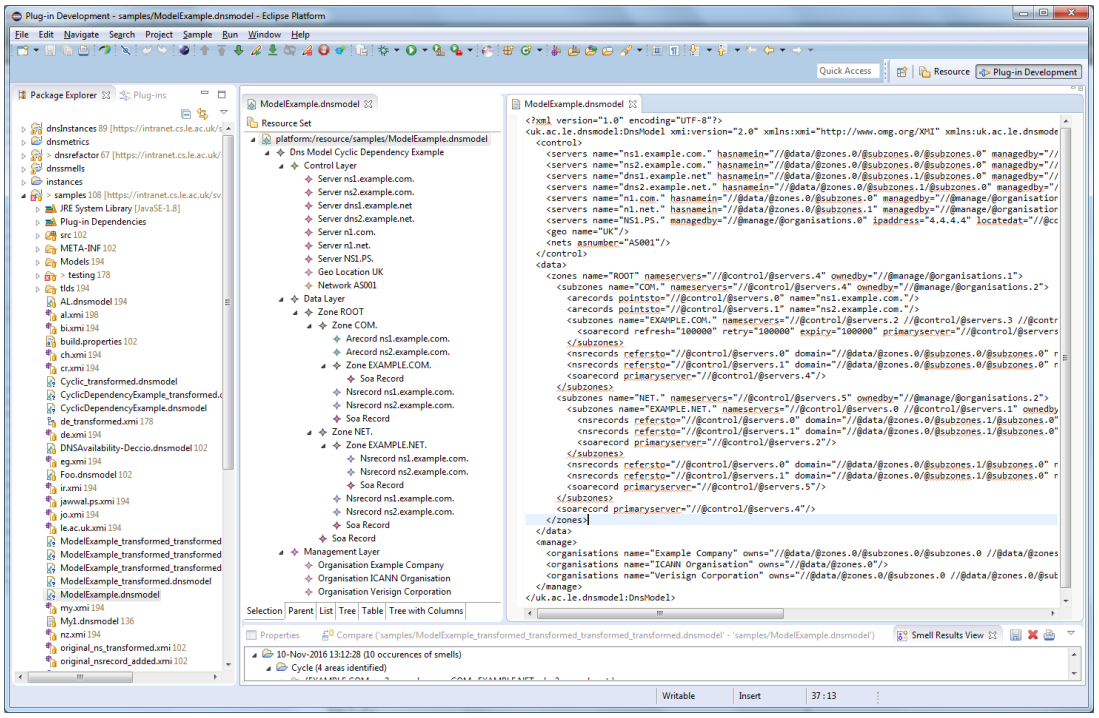


FIGURE 7.7: Example DNS Model Instance (Model and Textual Views).

be visually and textually compared using *EMF Compare* and changes are displayed and highlighted. DNS system administrators will able to evaluate how much improvement that has been achieved and if the same refactoring or other ones are need to get rid of all occurrences of bad smells in the model and improve the quality of the system.

7. Each time a refactoring is applied, structural metrics are calculated on the newly generated model and the values are fed to the corresponding prediction model to get a value for the quality attribute Q_{new}. This value is compared with the Q_{old} value for each of the four quality attributes to see if there has been any improvement in those quality attributes. Table 7.2 shows the values of those metrics for each iteration. Figure 7.8 shows a screen shot of the model instances after applying the set of refactorings to erase the corresponding bad

smells. It shows how a particular occurrence of the smell has been erased with no additional smells generated.

8. A new zone configuration and server layout report is generated with recommendations listing what changes that need to be carried out.

TABLE 7.1: Bad Smells Detected on the Model Instance and the Proposed Refactorings.

Iteration#	Bad Smell	Refactorings
1	Cyclic Dependency	CreateARecord
2	Diminished Network Redundancy	CreateNewNet + MoveServerNet
3	Diminished Geographical Redundancy	CreateNewGeoLocation + MoveServerLocation
4	Small Number of ANSs.	AddNewServer + CreateNSRecord + CreateArecord

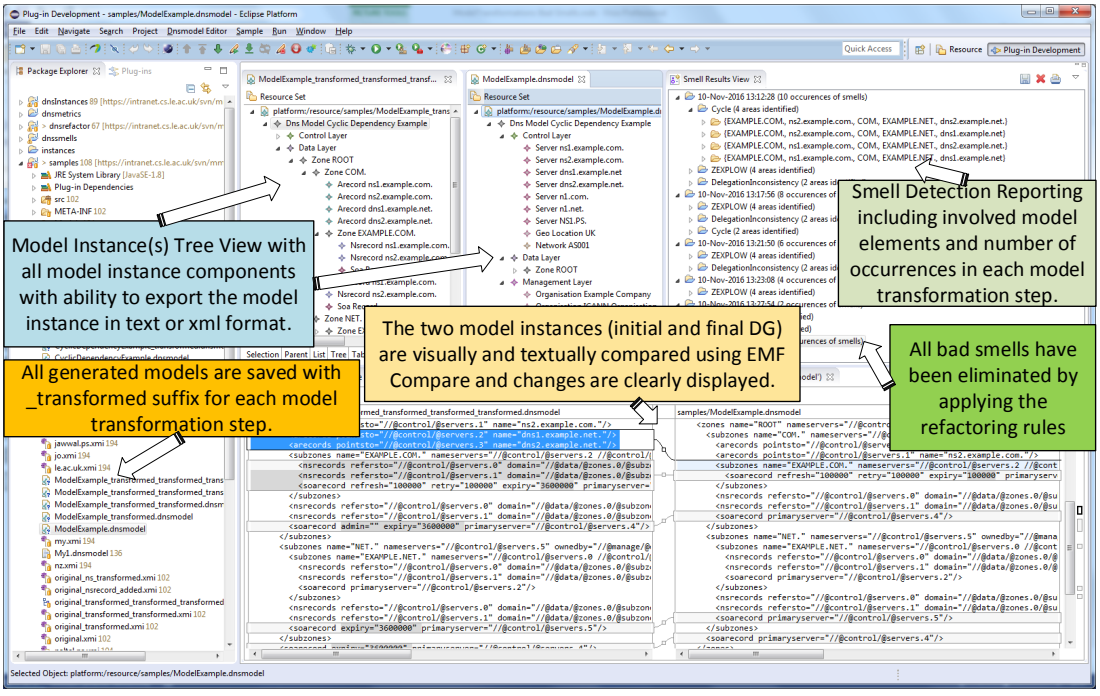


FIGURE 7.8: DNS Model Instances Transformation Using Model Compare.

Figure 7.9 shows a screen shot of the WEKA environment which shows the predicted *Stability* quality attribute for each model transformation iteration. The predicted

TABLE 7.2: Measurements of Metrics on the Generated DNS Model Instances.

Inst.#	AS	ANS	NETD	GEOD	Red	AC	AQP	DCZ	TPZ	DCO	TPO
Initial	10	4	1	1	4	0.5	3	2	2	2	1
1	12	4	1	1	4	0.5	3	2	2	2	1
2	18	4	4	4	4	0.5	3	2	2	2	1
3	21	5	5	5	4	0.5	3	3	2	3	1
4	24	6	6	6	6	0.5	3	4	2	4	1

values were calculated based on the *LWL* prediction model developed as part of the empirical assessment reported in Chapter 4. For each iteration of applying the refactorings, predicted quality attributes are calculated as shown in Table 7.3. The system administrator will be able to terminate the process at any stage based on the reported qualities of the system and the local policies, intents and constrains. After eliminating the bad smells, the final value of (Q_{Target}) of the target model instance should be improved over the source quality attribute (Q_{Source}) so the changes are then committed and the process is terminated.

TABLE 7.3: Effects of Applying Refactoring on the Perceived Quality Attributes of the DNS Model Instances.

Iteration#	Refactoring	Availability	Security	Stability	Resiliency
0	Source Model Instance	2	2	2	2
1	CreateARecord	2	4	2	2
2	MoveServerNet	2	4	2	2
3	MoveServerLoc	4	4	3	3
4	AddNewServer	3	4	4	3

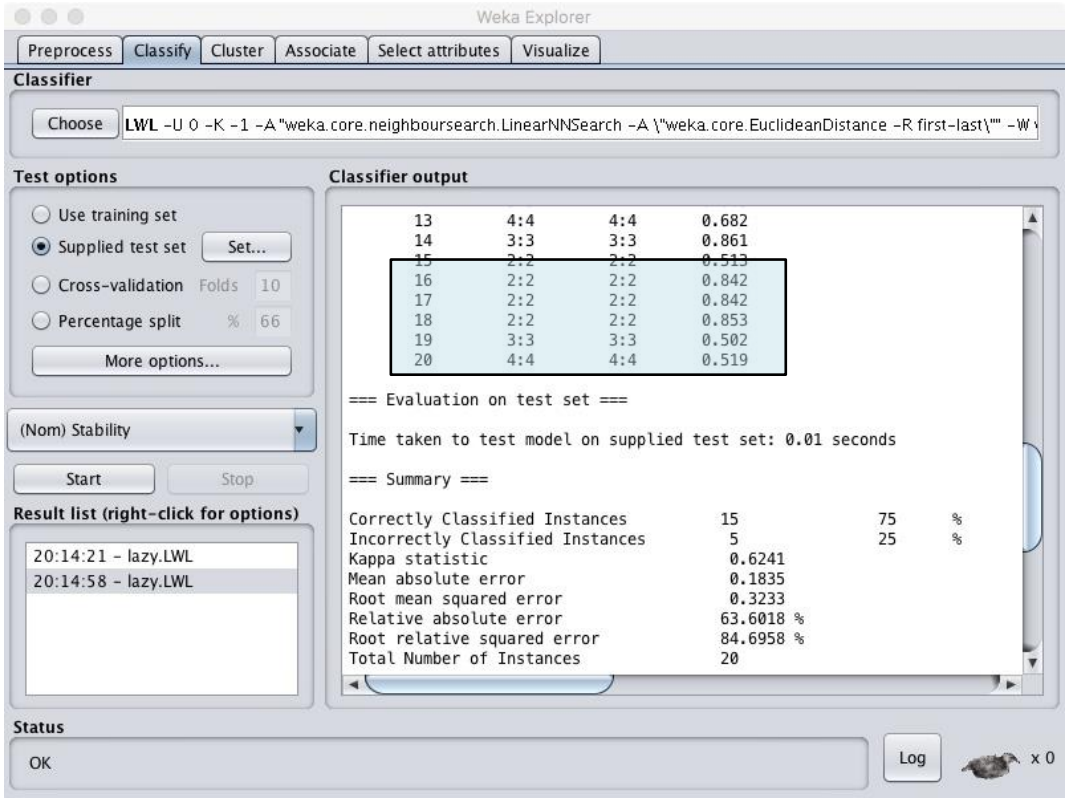


FIGURE 7.9: Predicted Values of Quality Attribute (Stability) for the Various Iterations of the Transformed DNS Model Instances.

Chapter 8

Related Work

In this chapter, we provide a survey of related work in the subjects discussed within the various chapters of this research and relate it tom our research contributions. The research deals with many subjects from different disciplines and project them to the DNS realm.

8.1 DNS in Operation

Since the inception of the domain name system (DNS) in 1983, there has been a large body of work [6, 10, 14, 21–24] on understanding its operation, availability, security and stability. Issues as understanding the DNS ecosystem [99–102], DNS components' behavior [96, 103, 104], security issues of resolution [9], applications for malicious actors detection and profiling [105–107], DNS stability and resiliency [108, 109], etc. have been widely researched.

8.1.1 DNS Interdependencies

The DNS is a complex distributed system, a system of systems composed of a highly interconnected infrastructure, protocols and operations procedures. DNS name dependencies are analysed in [8, 16, 17] and [7], in which the potential for a large number and variety of servers affecting name resolution is demonstrated.

Ramasubramanian, et al. [16] demonstrate the far-reaching effects of DNS dependencies. Their results show that a domain name relies on 44 name servers on average. Several surveys of production DNS deployments have been conducted [13, 15, 17] with various misconfigurations are analysed.

So far the main efforts in addressing the problem have focused on informing the operators about the existence of DNS configuration errors, either by Internet RFCs [5, 6] or with directives set by specific organizations [4].

8.1.2 DNS Measurements

Individual operators and independent researchers have measured various aspects of the DNS and from various prospective such as from the user, resolver or network points of view.

Deccio et al. [7] perform further examination of name resolution behaviours to create a formal model of name dependencies in the DNS and quantify the significance of such dependencies. Shulman et al. [11] studied the operational characteristics of the DNS infrastructure and how some factors impact resilience, stability and security of the DNS services.

DNS availability and robustness have been analysed in other studies [57, 110]. In these empirical studies. DNS availability was measured from a perspective of responsiveness of resolvers and authoritative servers, and diversity of DNS performance from different client perspectives. Deccio et. al [7] derived a theoretical availability model and methodology to systematically identify such misconfigurations and quantify their impact on availability. In their analysis, they applied their theoretical model to a deployment of a domain name and its dependencies to measure its availability.

Casalicchio et al. [18] proposed a framework for the evaluation of the health and security levels of operational DNS. To the extent of our knowledge, only very few preliminary studies for defining suitable metrics to measure the quality attributes of the DNS system have been conducted [7]. Even within these existing works, not much theoretical or empirical evaluation of the proposed metrics has been done.

8.1.3 DNS Troubleshooting

The state of the DNS is presented in several surveys of production DNS deployments [15, 17]. Various misconfigurations are analysed, including lame delegation, diminished server redundancy, cyclic dependencies, and inconsistency of NS RRsets between parent and child zones.

Despite all the existing efforts, DNS configuration errors are still widespread today [19], and one of the main reasons is the lack of adequate tools to help DNS operator identify and correct configuration errors in their own domains. Previous studies are largely based on empirical analysis, whereas in this paper we derive a formal operational model and methodology to systematically identify misconfigurations and bad deployment choices in the form of operational bad smells [15].

Although several DNS troubleshooting techniques and problem identification methods [111, 112] have been proposed and several tools [113–115] have been built, most of these methods and tools apply their detection techniques directly on the zone files through a predefined zone schema and a specified set of integrity constraints.

Chandramouli and Rose [116] considered integrity constraints for Resource Records (RRs) from single and multiple zones. They found that many integrity constraints have to be satisfied across zones.

Danzig et al. [117] provided an extensive analysis of the DNS traffic observed on a root name server. They identified a number of DNS implementation bugs and found that such bugs incurred unnecessary wide-area DNS traffic by a factor of twenty.

Jung et al. [20] measured the DNS performance in term of query response time perceived by DNS resolvers, and studied the effectiveness of caching. They found that the query response time is highly related to the number of referrals, and that the majority of queries complete in less than 100 milliseconds. They further concluded that DNS caching works effectively even when the TTL value of host names is as low as a few hundred seconds, as long as the domain servers' A RRs are cached.

Liston et al. [57] studied the diversity in DNS performance perceived by a number of geographically distributed clients. They showed that the mean response time for name lookups at different sites varies greatly, and the performance of root servers and TLD servers have the least impact for non-cached entries. In this paper we examine the diversity in server placement and its impact on zones availability.

Pang et al. [110] measured the availability of the individual DNS authoritative and caching servers and studied the different server deployment strategies. Both these works measure the reliability of individual components of the system, whereas in our study we measure the reliability of the DNS infrastructure, and more specifically we show how it is affected by human errors.

Casalicchio et al. [18] proposed a set of metrics to be used to evaluate the health of the DNS by measuring the DNS along three dimensions, namely vulnerability, security and resilience. Most of these studies can detect only a subset of the DNS infrastructure-related configuration errors. On the other hand, they implement diagnostic tests that can identify errors related with application-specific resource records. They do not take into account the inter-dependencies stemming from the hierarchical nature of the DNS, zone administrators' practices and deployment choices.

There are a number of companies and individuals that look into the problem of lame delegation. Men & Mice [118] periodically measures the lame delegation as it appears under the com domain; Team Cymru [119] collects the BIND log files from a number of DNS servers and extracts from them a list of lame servers; and Credentia [120] provides lame delegation statistics on the TLD zones.

So far, the main efforts in addressing the problem have focused on informing the operators about the existence of DNS configuration errors, either by Internet RFCs or with directives set by specific organizations. Despite all the existing efforts, DNS configuration errors are still widespread today and one of the main reasons is the lack of adequate tools to help DNS operator identify and correct configuration errors in their own domains.

8.2 Bad Smells

In this section, we present related work in the field of code smells identification, specification and detection in object oriented programming and how it expanded into the fields of model bad smells detection and model refactoring.

8.2.1 Bad Smells Idetification

There is a large body of work on the identification of problems in software testing [121], databases [122], and networks. Several books have been written on smells [86, 123, 124] in the context of object-oriented programming.

In their paper, Min Zhang et. al. [85] conducted a systematic literature review to investigate the current status of knowledge about Code Bad Smells. The link between the structure of the software and some of its quality attributes (i.e. maintainability) is established in [125]. In [126], the study shows that software structure, which was measured using source code metrics, could predict maintainability of the software. Another study also shows that source code metrics and perceived maintainability have a correlation [127].

Marinescu [128] presented a metric-based approach to detect code smells. Alikacem and Sahrouri [129] proposed a language to detect violations of quality principles and smells in object-oriented systems. Mens and Tourwe [46] have conducted a comprehensive survey of software refactoring. While software refactoring has started focusing on restructuring of programs, the research has extended to model refactoring [38].

8.2.2 Bad Smells Detection

All the techniques for detecting code smells in source code have their roots in the definition of code design defects and heuristics for identifying those that are outlined in well-known books: [86], [130–132]. The first by Webster [131] describes pitfalls in Object-Oriented (OO) development going from the management of a project through the implementation choices, up to the quality assurance policies. The second by Riel [132] defines more than 60 guidelines to rate the integrity of a software design.

Fowler [86] defines 22 code smells together with refactoring operations to remove them from the system. Brown et al. [130] describe 40 anti-patterns together with heuristics for detecting them in code [133].

Detecting model smells using object-oriented metrics is known as Metric-based Refactoring, a term coined by (Simon et al. 2001) who used metrics to identify smells in object-oriented code. An important issue with using model metrics as a smell detection strategy is the specification of a threshold value for the metrics as it has decisive influence on detection accuracy.

Marinescu [128] identified three ways of parameterizing threshold values for metrics used for smell detection (1) Empirical results from metrics' authors and similar past experiences, (2) using a Tuning Machine to automatically find proper threshold values for regulating the detection strategy [134] and (3) analyzing multiple versions for change stability information or persistency of a design flaw over time [135].

8.3 Refactoring

The term refactoring was originally used in the software industry for source code restructuring by William Opdyke [90]. The main aim of refactoring is to reduce software complexity by "changing a software system in such a way that it does not alter the external behavior of the code, yet improves its internal structure" [86].

With the popularity of MDA and UML, recent approaches for refactoring have elevated it to a more abstract level of design models. While a refactoring is a solution to a single problem, the refactoring pattern is generic and well documented.

8.3.1 Refactoring Techniques

Mens et al. provide a survey of existing research in the field of software refactoring [46], where they identify six distinct activities the refactoring process consists of. Wake, in the Refactoring Workbook [136], aims to provide practice in the identification of the most important smells and practice with the most important refactoring techniques, with particular emphasis on discovering new refactorings.

On the level of formalisation: a refactoring rule is such a pattern that is implemented in a tool or formalised with mathematical means [91]. Graph transformation systems (GTS) are well-suited to model refactoring and, more generally, model transformation [44]. Model refactorings based on GTS can be found in [46], [137–140].

Our objectives are similar to those of previous DNS operation studies but our approach differs. We use terminologies and concepts well established in the software engineering subject and project them in the Domain Name System realm. Our method utilizes a set of measurable, structural and lexical properties defined over a DNS model to detect the smells in early stages of the DNS deployment. It also suggests graph-based refactoring rules as correction mechanisms for those bad smells.

8.3.2 Refactorings Analysis

With respect to the contributions of this thesis, we discuss two main threads of related work. First, behaviour preservation properties, priorities and orchestration, dependability and conflicts analysis techniques. Second, work done in the field of graph- based model transformation.

Behaviour Preservation. The most common approaches to behaviour preservation rely basically on checking given models and their refactored versions. Mens [95] and Bottoni et al. [141] use graph transformations to describe refactorings for

models. The application of this formalism comes with the additional benefit of formal analysis possibilities of dependencies between different refactorings. Rangel et al. [140] introduced a more general technique for checking behaviour preservation of refactorings defined by graph transformation rules. They used double pushout (DPO) rewriting with borrowed contexts, and, exploiting the fact that observational equivalence is a congruence, They show how to check refactoring rules for behaviour preservation. They concluded that when rules are behavior-preserving, their application will never change behaviour, i.e., every model and its refactored version will have the same behaviour.

Critical Pairs. Critical pair analysis was first introduced for term rewriting, and later generalised to graph rewriting [142]. The idea of critical pair analysis is quite simple. In [97], the formalism of critical pairs was explained and related to the formal property of confluence of typed attributed graph transformations. In [143], critical pair analysis is used to detect conflicting requirements in independently developed use case models. In [144], critical pair analysis has been used to increase the efficiency of parsing visual languages by delaying conflicting rules as far as possible. In [145], graph transformation dependency analysis has been used for the purpose of detecting and resolving inconsistencies in design models.

Refactorings Orchestration. Because of the huge search space when searching for possible model transformations for a given set of input/output model pairs, search-based techniques have been applied to automate this complex task [146–149].

8.4 Graph-Based Model Transformation

The motivation behind model-driven software development is to move the focus of work from programming to solution modelling. The model-driven approach has a

potential to increase development productivity and quality by describing important aspects of a solution with more human-friendly abstractions and by generating common application fragments with templates.

The applicability of graph transformations for model transformations rests upon the fact that most models exhibit a graph-based structure. The initial graph representing a model evolves through the application of graph transformation rules until the execution stops and we obtain the output graph, i.e., the output model. Beirmann et. al. [42] presented an approach to define EMF model refactoring methods as transformation rules being applied in place on EMF models. Performing an EMF model refactoring, EMF transformation rules are applied and can be translated to corresponding graph transformation rules.

In [150], object-oriented programs are represented as graphs before applying graph transformations for refactoring this abstract representation. Furthermore, Bottoni et al. [151] use graph transformations to describe refactorings for models. Introduction to the concept of model refactoring using UML models as candidates for refactoring was first proposed by Sunyé et al [152].

Mens et al. [153] and Mohamed et al. [154] provided introduction, overview and taxonomy of model refactoring literature. These two reviews describe the state-of-the-art and taxonomical classification of various model refactoring approaches respectively. Mens et al., in the book titled "Model-Driven Software Refactoring" [155], were the first ones to publish a review on model-driven refactoring. Mohamed et al.'s (Mohamed et al. 2009) review emphasized classifying the existing model refactoring approaches based on a feature-model driven taxonomy. They extended the model transformation feature diagram presented by Czarnecki and Helson [156] by adding concepts specific to model refactoring domain.

Chapter 9

Conclusions and Future Work

In this thesis we have analyzed the various interdependencies stemming from the delegation-based structure of the DNS. We developed a conceptual model that abstracts all these dependencies with the three operational planes.

We introduced structural metrics as indicators of the quality of the operational system and used them to build predictive models for the availability, security, stability and resilience quality attributes of the system.

We also introduced the concept of bad smells to the DNS arena and build a bad smells catalogue with graph-based refactoring suggested as correction mechanism for the bad smells and built an advisory tool as an implementation of the methods, techniques and quality assurance framework proposed in this research.

The tool, in a systematic process, can automatically direct the zone administrator to places in the zone file that contain potential design and deployment problems that may compromise availability, resiliency, stability or security of the domain name system. We summarize our conclusions and elaborate on future research in this final chapter.

9.1 Conclusions

DNS relies on a delegation-based architecture, where resolution of a name to its IP address requires resolving the names of the servers responsible for that name. The graphs of the inter-dependencies that exist between name servers associated with each zone are called Dependency Graphs.

In Chapter 3 we analysed the various interdependencies within the three operational planes of the DNS and constructed a DNS Dependency Model as a unified representation of these Dependency Graphs.

In Chapter 4, we utilized a set of Structural Metrics defined over this model as indicators of external quality attributes of the domain name system. We explored the inter-metric and inter-quality relations further in order to quantify the indicative power of each metric. We applied machine learning algorithms in order to construct Prediction Models of the perceived quality attributes of the operational system out of the structural metrics of the model. Assessing these quality attributes at an early stage of the design/deployment enables us to avoid the implications of defective and low-quality designs and deployment choices and identify configuration changes that might improve the availability, security, stability and resiliency postures of the DNS.

In Chapter 5, the model-based ISDR method was presented. The method subsumes all the steps necessary to identify, specify, formalize and detect operational bad smells. The method deals with smells on a high-level of abstraction using a consistent taxonomy and reusable vocabulary defined by the DNS model. The set of identified bad smells has been formally specified in a bad smells catalogue.

The method laid the basis for developing a visual advisory tool for system administrators to identify, analyse, discover, and remedy operational bad smells. Case

studies were used to validate the method and its usefulness in identifying and detecting bad smells has been verified.

In Chapter 6, the various techniques of the ISDR method (including the metrics, bad smells and refactorings) were specified using the EMF Refactor Quality Assurance Frame work and other modelling tools. The various steps included in the process of implementing the ISDR method techniques were presented and case studies were used as an implementation examples of the method.

In Chapter 7, behaviour preservation properties, conflicts, dependabilities and priorities of the proposed refactoring rules were analysed. Graph-based model transformation tools along with the ISDR method techniques and DNS quality prediction models were utilised to build a prototype of the DNS advisory tool. Case studies and concrete examples were developed to validate the correctness and evaluate the applicability of the tool.

In Chapter 8, a summary of related work in the fields of DNS management and troubleshooting, bad smells identification and detection, software modelling, refactoring and model transformations was presented.

The DNS will continue to play an integral role in the Internet usability and more applications rely on the effective operation of its infrastructure. The models, metrics and other techniques presented in this dissertation can assist DNS administrators in better understanding their DNS deployments and avoiding name resolution failure by properly engineering and maintaining their DNS infrastructures.

The diagnostic and advisory tool consider several properties and metrics from the DNS dependency model and use them to detect bad smells and suggest graph-based refactorings to eliminate such smells. It also enables the system administrator to use the structural metrics of the DNS model instance to predict the perceived quality attributes of the system. Zone administrator will be able to run several scenarios

and apply several refactoring rules through the tool to determine the solution that best meets their local policies.

9.2 Research Limitations

A limitation of a study design or instrument is the systematic bias that the researcher did not or could not control and which could inappropriately affect the results. In our research we were faced with the followin main limitations:

- **Models**. Lack of DNS models to use in our assessment forced us to build such models from operational DNS configurations and deployment layouts using in-house developed tools.
- **Metrics**. It should be noted that the used set of metrics may not be comprehensive and other consecutive research could further complete this proposed set by defining new metrics from other perspectives.
- **Tools**. In implementing the different ISDR techniques, we used the EMF Refactor framework and its associated tools, specifications and techniques. The tool has not gone through extensive testing or industry-scale implementation since its use is limited to the research and academic utilisation only.
- **Assessment Experiment**. Another limitation which we faced in our assessment experiment, is the limited number of data points that were collected due to the limited number of participants amongst the DNS operators. Another limitation is posed by using subjective measurement mechanisms is that different participants may have different attitudes toward the evaluation of these attributes. The detailed limitations faced in the assessment experiment were clearly explained in the Threats to Validity section in Chapter 4.

9.3 Future Work

Due to the growing interconnectivity of critical infrastructure assets, a DNS fault under certain conditions could have serious national and international implications. There are number of ways in which this work can be extended and we plan to grow our research in the following key directions:

9.3.1 Extending the DNS Operational Model

The DNS Operational model developed as part of this research was limited to the static structure (design and deployment) of the Domain Name System. Extending the model to include additional components of the system such DNS resolvers and end-users as well as modelling the system from different vantage points will widen the understanding of the DNS and enable the system administrators to try different configuration and deployment scenarios and simulate their effects on the overall system performance. We also plan to introduce some additional elements to the DNS Model to represent the dynamic behaviour of the system and its various components. We expect that modelling these factors would reduce the error and lead to a better performance of a DNS reference model on the prediction of DNS behaviour.

9.3.2 DNS Structural Metrics and Prediction Models

We will conduct a broad empirical validation of assessment techniques, by implementing the comprehensive metrics suite. We aim to perform this validation on most of the current general top-level-domains (gTLDs) and country code top-level-domains (ccTLDs). The analysis results will be used to calibrate the metric calculation, bad smells detection as well as the quality assessment techniques.

One problem that we will be tackling during this task is the reluctance of many TLD operators to share their data and internal measurements due to confidentiality and privacy concerns. Sharing is always a big challenge, technical issues are marginal with respect to business, privacy, confidentiality and legal issues. Another issue was the need for objective mechanisms for measuring the value of the real quality attributes of the DNS.

Our experience shows that although subjective opinions of the participants are good measures of the perceived quality attributes of the models, still the use of objective measures to second the obtained results is valuable. We believe that both theoretical and empirical development of appropriate objective measures of the various DNS actual quality attributes is required. However, further experimentation with industrial scale DNS models and more participants are required to fully verify the conclusions of our work.

We plan to present the method to major TLD operators and DNS industry partners to investigate the scope for commercial exploitation of this method and associated techniques. We will be looking into joint research and knowledge transfer with TLD operators, ICANN technical work groups (specially the DNS Security, Stability and Resiliency workgroup), DNS Operations, Analysis, and Research Center (DNS-OARC) and other industry partners to utilise the results of this research to improve the current status of DNS metrics and DNS quality prediction models.

9.3.3 DNS Quality Indicators

Currently, there is little consensus on the right measures and acceptable performance levels for the DNS as a whole related to availability, security, stability and resiliency. Individual operators and independent researchers have measured various aspects of the DNS, but to date little progress has been made in defining and implementing

standard, system-wide metrics or acceptable service levels. As mentioned before, quality assessment is not a goal in itself. The real goal is to improve the quality of the system. This is achieved by eliminating all bad smells that negatively affect the quality attributes of the system.

There are many examples of failed attempts to over-summarize a complex system's status into a single indicator. The problem is that, in a complex system, a single numeric value can't express the system's condition in a usable way. One implementation would be to use a dashboard to convey the status of operational qualities of the system.

We plan to improve the ability of the DNS system administrators to comprehend about the system operational qualities by presenting them with a dashboard of indicators that measure roughly the four quality attributes of availability, security, stability, and resiliency of the system based on the developed prediction models. This dashboard will be tightely connected with a list of bad smells that may be present in the model instance along with applicable and recommended refactorings.

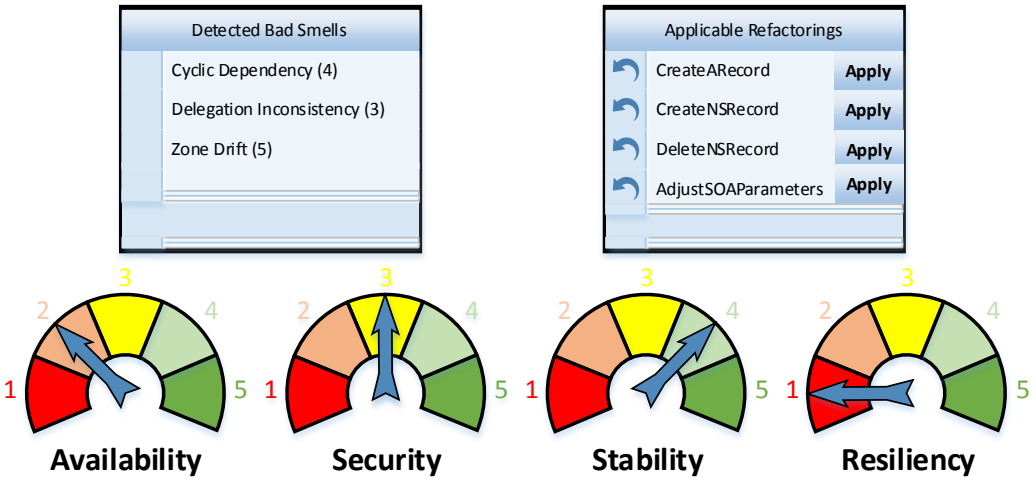


FIGURE 9.1: Implementation-Level DNS Quality Dashboard.

The important point is not that the factors are displayed on the board in a concise numerical way, but that each of the measured elements has designated green, light green, yellow, orange and red zones, (that resembles the 5-points Likert-Scale used in our empirical assessment and shown in Table 4.8) indicating whether a particular measurement is within tolerance for the system. Figure 9.1 shows how the DNS quality dashboard indicators can be implemented. There is no single indicator of quality on the display, yet the system administrator could tell immediately from the presented quality indicators and bad smells whether the system is performing well or not.

Appendix A

The DNS Dependency Model

DNSModel

A root node is necessary in the EMF model to contain all model elements. The *DNSModel* is the root element of our DNS Model.

Attributes:

- *Name: EString:* One instance of the model represents one type graph. The *name* attribute is the name of this instance of the model.

Associations:

- *DataLayer [0..1]:* A collection of elements (such as *Zones* and *Resource Records*) that represent the data operational plane of the DNS.
- *ControlLayer [0..1]:* A collection of elements (*Servers*, *Networks* and *Geolocations*) that represents the control plane elements of the operational DNS.
- *ManagementLayer [0..1]:* A collection of (*Organisations*) that represent the entities responsible for managing and hosting the DNS *Servers* and *Zones*.

Constraints:

1. There can be only one *DNSModel* component in a model instance.
2. There can be just only one component of *DataLayer* in the model instance.
3. There can be just only one component of *ControlLayer* in the model instance.
4. There can be just only one component of *ManagementLayer* in the model instance.

Layers in the DNS Dependency Model represent the different operational planes of the DNS system. The DataLayer model component contains the *Zone* element and its *subzones* and all logical elements within the DNS zone file called the *Resource Records* while the control layer and management layer contain representation of the physical elements in the model such as servers, networks, geographical locations and management organisations.

A.1 Modelling the Data Layer

The overall trust in DNS depends upon the integrity of the zone file data [Ref:DNS and BIND, Fourth Edition]. The zone file hosted on an authoritative name server consists of various types of records called Resource Records (RRs). Associated with each DNS resource record is a type (RRtype). An RR of a given RRtype in a zone file provides a specific type of information. A zone file generally consists of multiple RRs of a given RRtype with some integrity constraints (e.g., there can be only One SOA RR in a zone file). It can also have multiple RRs for the same domain name and same (or different) RRtype (e.g., multiple authoritative name servers or mail servers for a domain say services.example.com).

Zone

The *Zone* model element is an abstraction of a single administrative unit within the domain name system data space.

Attributes:

- *name: EString:* An attribute identifies the name of this element.

Associations:

- *nameservers [2..13]:* Associates the *Zone* with a set of authoritative name *Servers* that are responsible for giving answers in response to questions asked about names in this particular *zone*.
- *ownedby [1..1]:* Associates the *zone* to the *Organisation* that is designated as the management entity for this *Zone*.
- *subzones [1..1]:* Associates the *zone* to other *zones* that are sub-zones (child/-parent relationship) of this *Zone*.

Constraints:

1. There can be a minimum of two and maximum of 13 name *Server*s components associated with this particular *zone*.
2. The *zone* can be owned by one *organisation* only.
3. The *Zone* can contain unlimited number of *sub-zones* (also of type *zone*) with one parent *zone* only.

We limit our focus in the DNS model to the set of resource records within the zone file called infrastructure resource records. These are the set of resource records

that are essential to the DNS dependencies and ensure the consistency and stable operation of the system. The set of RRs includes the following:

SOARecord

The *SOARecord* (Start of Authority Record) which identifies the authoritative main configuration parameters of a *zone*.

Attributes:

- *admin: EString.* An attribute specifies the mailbox of the person responsible for this zone.
- *serial: EInt.* The version number of the original copy of the zone file.
- *refresh: EInt.* Represents the time interval before the zone should be refreshed.
- *retry: EInt.* Represents the time interval that should elapse before a failed refresh should be retried.
- *expire: EInt.* Represents the The time interval (in seconds) that specifies the upper limit on the time that can elapse before the zone is no longer authoritative.
- *minTTL: EInt.* Represents the time that should be exported with any RR from this *zone*.

Associations:

- *PrimaryServer [0..1]:* Associates the *SOARecord* to the *Server* that is the original or primary source of data for this zone.

Constraints:

1. There can be only one *SOARecord* component in a model instance associated with a particular Zone.

NSRecord

The *NSRecord* (Name Server Record) which identifies the authoritative name *server*(s) for the *zone*.

Attributes:

- *name: EString:* An attribute identifies the name of this element.

Associations:

- *nsrecords [0..*]:* Associates the *NSRecord* to the *Zone* that contains this record to identify the set of authoritative name servers to this particular *Zone*.
- *refersto [1..1]:* Associates the *NSRecord* to the *Server* that is designated as an authoritative name server for this *Zone*.

Constraints:

1. There can be only one *Server* component associated with this particular *NSRecord*.

ARecord

The *ARecord* (Address Record) which identifies the IP address of a particular *Server*. An A record maps a domain name to the IP address (IPv4 such as 192.168.23.12) of the computer hosting the domain. Simply put, an A record is used to find the IP address of a computer connected to the internet from a name. For IPv6 (the length of an IPv6 address is 128 bits, compared with 32 bits in IPv4), the record name is

the AAAA Record which is an identical form of the ARecord but with the IP given in IPv6 format.

Attributes:

- *name: EString:* An attribute identifies the name of this element.

Associations:

- *arecords [0..*]:* Associates the *ARecord* to the *Zone* that contains this record to identify the IP address of one of the authoritative name servers to this particular *Zone*.
- *pointsto [1..1]:* Associates the *ARecord* to the *Server* that is designated as an authoritative name server for this *Zone*.

Constraints:

1. There can be only one *Server* component associated with this particular *ARecord*.

CNAMERecord

The *CNAMERecord* (Canonical Name Record) which specifies that the name is an alias for another *Server* name. When a DNS resolver encounters a CNAME record while looking for a regular resource record, it will restart the query using the canonical name instead of the original name. The canonical name that a *CNAMERecord* points to can be anywhere in the DNS, whether local or on a remote *Server* in a different DNS *Zone*.

Attributes:

- *name: EString:* An attribute identifies the name of this element.

Associations:

- *cnamerecords [0..*]:* Associates the *CNAMERecord* to the *Zone* that contains this record to identify the alias of an authoritative name servers to this particular *Zone*.
- *aliasto [1..1]:* Associates the *CNAMERecord* to the authoritative name *Server* that is aliased by this record.

Constraints:

1. There can be only one *Server* component associated with this particular *CNAMERecord*.

HINFORecord

The *HINFORecord* (Host Information Record) A HINFO-record specifies the host server's hardware type, CPU and operating system.

Attributes:

- *name: EString:* An attribute identifies the name of this element.

Associations:

- *hinforecords [0..*]:* Associates the *HINFORecord* to the *Zone* that contains this record.

Constraints:

1. There can be only one *HINFORecord* component contained in this particular *Zone*.

A.2 Modelling the Control Layer

The DNS control layer subsumes the abstraction of the interconnected graph of authoritative name servers of a certain zone and their network and geographical distribution along with inter-dependencies and other relations that represents the control structure of the domain name system. It contains the following model elements:

Netwok

The *Network* which identifies the network Autonomous System (AS) Number where a name *Server* is hosted at.

Attributes:

- *name: EString:* An attribute identifies the AS number of the network.

Associations:

- *nets [0..*]:* Associates the *Network* to the *ControlLayer* that this network is part of.

Constraints:

1. There can be only one *Network* component with a particular *name/AS Number* in the associated *ControlLayer*.

GeoLocation

The *GeoLocation* which identifies the geographical location where a name *Server* is physically located at.

Attributes:

- *name: EString:* An attribute identifies the name of the geographical location (country).

Associations:

- *geo [0..*]:* Associates the *GeoLocation* to the *ControlLayer* where this location belongs to.

Constraints:

1. There can be only one *GeoLocation* component with a particular *name* in the associated *ControlLayer*.

Server

The *Server* (Name Server) which identifies the physical or logical name*Server*.

Attributes:

- *name: EString:* An attribute identifies the name of this element.
- *ipaddress: EString:* An attribute identifies the internet protocol (IP) address for this *Server*.

Associations:

- *hasNameIn [0..*]:* Associates the *Server* to the *zone* that the host name of the *server* is registered under that particular *zone*.
- *hostedAt [1..1]:* Associates the *Server* to the *network* that the *server* is connected to.

- *locatedAt [1..1]:* Associates the *Server* to the *GeoLocation* that the *server* is located at.
- *managedBy [1..1]:* Associates the *Server* to the *organisation* that is responsible for managing it.
- *servers [0..*]:* Associates the *Server* to the *ControlLayer* that contains this server in the textitDNSModel.

Constraints:

1. The *Server* can be *managedBy* just one *Organisation* component.
2. The *Server* can be *hostedAt* only one *Network* component.
3. The *Server* can be *locatedAt* only one *GeoLocation* component.
4. The *Server* can *hasNameIn* just one *Zone* component in the model.

A.3 Modelling the Management Layer

Organisation

The *Organisation* which identifies the entity that owns a certain *Zone* or manages the *Server* hosting a copy of a particular *Zone* within the Dependency Graph of a particular Zone.

Attributes:

- *name: EString:* An attribute identifies the name of this element.

Associations:

- *owns [0..*]:* Associates the *Organisation* to the *Zone* that is managed by this particular *Organisation*.
- *childorg [0..*]:* Associates the *Organisation* to other organisations which are part (Parent/Child relation) of this particular *Organisation*.

Constraints:

1. There can be only one *Organisation* component that owns a particular *Zone*.

Appendix B

DNS Metrics Suite

B.1 Size Metrics

Metric:	**Attack Surface**
Symbol:	$AS(z)$
Definition:	We define the attack surface of a system in terms of the system's attackability along three abstract dimensions: zone, server, and organisation.
Usability:	Intuitively, the larger the attack surface, the more likely the system will be attacked, and hence the more insecure it is.
How to Measure:	Count the total number of zones, servers and organisation within the DNS model instance
Example:	The Attack Surface of the model instance presented in Figure 7.6 is 10 where we have 4 zones (COM, EXAMPLE.COM, NET and EXAMPLE.NET), 4 name servers (NS1,NS2.EXAMPLE.COM and DNS1,DNS2.EXAMPLE.NET) and 2 organisations (ORG-A and ORG-B).

Metric Notations: Let $|Zones|$ be the total number of zones, $|Servers|$ be the total number of name servers and $|Orgs|$ be the total number of organisations within the model instance.

Formula: $AS(z) = |Zones| + |Servers| + |Orgs|$

Metric: **Redundancy**

Symbol: $R(z)$

Definition: The redundancy is the size of the smallest set of redundant name servers at any point in the zone's required resolution path and if failed, will render the zone's domain names unresolvable.

Usability: Redundancy is considered the "availability bottleneck" of a domain name. If all servers comprising the redundancy of a domain name were to fail, then the name would be rendered unavailable.

How to Measure: The methodology for determining the redundancy of a zone name is to compute the logical expression representing the resolution path(s) of the zone and then this set is reduced to conjunctive normal form (CNF), returning a set of disjunctions.

Example: The sets of servers comprising the redundancy of example.com in Figure 7.6 are: 1.1.1.1,1.1.1.2 and 1.1.1.3, 1.1.1.4. That is say that if all NS1.EXAMPLE.COM, NS2.EXAMPLE.COM, DNS1.EXAMPLE.NET and DNS2.EXAMPLE.NET are unavailable, then EXAMPLE.COM zone is rendered unavailable. The Redundancy of EXAMPLE.COM is 4

Formula:	The actual redundancy algorithm was implemented as part of the ISDR techniques in Java and is not listed in this work.

Metric:	**Number of Authoritative Name Servers**
Symbol:	$ANS(z)$
Definition:	The set of Authoritative name servers for zone (z) as configured in z and Parent(z).
Usability:	Authoritative name servers are the ones holding a copy of the zone file and responsible for answering authoritatively for any request regarding any domain name under the zone z.
How to Measure	Count the number of servers associated with the zone through the association *nameservers*.
Example:	The number of authoritative name servers for zone EXAMPLE.COM in the model instance presented in Figure 7.6 is 4 while the ANS(EXAMPLE.NET) is 2.
Formula:	Counting the number of authoritative name servers of a zone. It depends on the implementation of the model and the query language. for OCL it can be calculated on a zone based on this query: ***self.nameservers->size().***

Metric:	**Number of Zones**
Symbol:	$Zones(z)$
Definition:	Total number of zones within the model instance.
Usability:	The metric can be used as an indication of how many zones influencing the resolution of domain names under the zone z.
How to Measure	Count all elements in the model of instance type zone.

Example:	The number of zone influencing the resolution of domain names under the zone (EXAMPLE.COM) in the model instance presented in Figure 7.6 is 4 (COM,EXAMPLE.COM,NET,EXAMPLE.NET). The root zone was excluded since it is required for the resolution of every zone in the domain name space.
Formula:	Counting the number of zones in the model. It depends on the implementation of the model and the query language. Using OCL it can be calculated on a zone based on this iterated query over all zone elements in the model: ***self.data.zones->collect(z : Zone | z.subzones->asOrderedSet())->size().***

Metric:	**Number of Organisations**
Symbol:	$Org(z)$
Definition:	Total number of Organisations within the model instance.
Usability:	The metric can be used as an indication of how much coordination is needed between various institutions hosting name servers included in the resolution path(s) of domain under zone z.
How to Measure	Count all elements in the model of instance type Organisation.
Example:	The number of organisations involved in the resolution of domain names under the zone (EXAMPLE.COM) in the model instance presented in Figure 7.6 is 2 (ORG-A and ORG-B). The root zone organisation (ICANN/IANA) was excluded since it is required for the resolution of every zone in the domain name space.

Formula: Counting the number of organisations in the model. It depends on the implementation of the model and the query language. for OCL it can be calculated on a zone based on this iterated query over all zone elements in the model: ***self.manage.organisations->collect(o : Organisation | o.childorg->asOrderedSet())->size().***

Metric: **Number of In-Bailiwick Servers**

Symbol: $Is(z)$

Definition: The number of authoritative name servers of zone z and who has their names in the same zone z.

Usability: This measure is an indication of how the name servers are within the administrative authority of the zone administrator out of the total number of authoritative name servers of the zone.

How to Measure Count the number of authoritative name servers of a zone where the name of the server is under the same zone.

Example: The number of In-Bailiwick name server for the zone (EXAMPLE.COM) in the model instance presented in Figure 7.6 is 2 (NS1.EXAMPLE.COM and NS2.EXAMPLE.COM) while it equals to 0 for the zone (EXAMPLE.NET) since both name servers are within the zone (EXAMPLE.COM).

Formula: Counting the number of authoritative name servers for a zone in the control plane of the model where they have (hasNameIn) the same zone z. It can be implemented in OCL as follows: ***self.nameservers.hasnamein.name.equalsIgnoreCase(self.name)->count(true).***

Metric:	**Number of Out-of-Bailiwick Servers**
Symbol:	$Os(z)$
Definition:	The number of authoritative name servers of zone z and who has their names under a zone name other than the zone z.
Usability:	This measure is an indication of how the name servers are outside the administrative authority of the zone administrator.
How to Measure	Count the number of authoritative name servers of a zone where the name of the server is under the same zone.
Example:	The number of Out-Of-Bailiwick name server for the zone (EXAMPLE.COM) in the model instance presented in Figure 7.6 is 2 (DNS1.EXAMPLE.NET and DNS2.EXAMPLE.NET) while it equals to 2 for the zone (EXAMPLE.NET) since both name servers are within the zone (EXAMPLE.COM).
Formula:	$Os(z) = ANS(z) - Is(z)$. It can also be implemented in OCL as follows: ***self.nameservers.hasnamein.name.equalsIgnoreCase(self.name)->count(false)***

B.2 Measures of Structural Complexity

Metric:	**Administrative Complexity**
Symbol:	$AC(z)$
Definition:	Describes the diversity of a zone with respect to the organisations administering its authoritative name servers.

Usability: The advantage mutual hosting of zones between organizations is an increased availability but at the same time increased potential of failure and instability of the zone resolution process.

How to Measure: Count the number of authoritative name servers managed by each organization involved in the dependency graph of zone (z).

Metric Notations: O_z: denotes the set of organizations administering authoritative name servers hosting zone (z); n: total number of authoritative name servers of zone (z); $|ANS|_z^o \subseteq |ANS|_z$: the subset of name servers administered by organization o in O_z.

Example: For example, assuming the servers ns.example.com and ns.example.net are the authoritative name servers for the zone(EXAMPLE.COM) and are operated by two separate organizations. the administrative complexity of EXAMPLE.COM with n = 2 is:

$AC(EXAMPLE.COM) = 1 - ((\frac{1}{2})^2 + (\frac{1}{2})^2) = 0.5$

Formula: $AC(z) = 1 - \sum_{o \in O_z} (\frac{ANS_z^o}{ANS_z})^n$.

Metric: **Hierarchical Reduction Potential (HRP)**

Symbol: $HRP(z)$

Definition: Quantifies how much the ancestry of a zone might be reasonably consolidated to reduce hierarchical complexity.

Usability: A greater HRP value indicates that minimizing hierarchical complexity might reduce failure potential.

How to Measure: We express the HRP of zone z, having m + 1 ancestral zones, as the fraction of layers that could be reduced if the number of zones is consolidated to $m' + 1$.

Example: While delegation is necessary in many cases, there are some cases in which collapsing a delegated zone is both reasonable and possible. For example, if example.com and sub.example.com are two zones administered by the same organization, the zone data for sub.example.com might trivially be migrated to the example.com zone and the delegation to sub.example.com removed. This consolidation reduces the number of zones ancestral to sub.example.com by 0.25 from 4 to 3.

Formula: $HRP(z) = \frac{m-m'}{m+1}$

Metric: **Network Diversity**

Symbol: $NETD(z)$

Definition: This is a metric of the diversity of placement of servers in terms of the number of distinctive networks identified by their Autonomous Numbers (AS) hosting the various authoritative name servers of a zone z.

Usability: Authoritative name servers have to be placed in diverse networks to avoid any single point of failure and to improve the resiliency of the overall system.

How to Measure Count the ordered set of networks in terms of their AS numbers hosting the authoritative name servers of zone z.

Example: The model instance provided in Question 7 in the DNS survey in Appendix E shows that all name servers for the zone (NIC.AA) are placed within the same network AS number since all of them are within the same subnet.

Formula: Count the ordered set of network AS numbers identifying where the authoritative name servers are hosted. In OCL this can be implements by executing the following query: $NETD(z)$ = ***self.nameservers.hostedat.asnumber->asOrderedSet()->size()***

Metric: **Geographical Diversity**

Symbol: $GEOD(z)$

Definition: This is a metric of the diversity of placement of servers in terms of the number of distinctive geographical locations (identified by the alpha-2 country code) where the various authoritative name servers of a zone z are located at.

Usability: Authoritative name servers have to be placed in diverse geographical locations to avoid any single point of failure such as power outages and natural and man-made disasters.

How to Measure Count the ordered set of Geographical Locations in terms of their country codes hosting the authoritative name servers of zone z.

Example: The model instance provided in Question 9 in the DNS survey in Appendix E shows that all name servers for the zone (NIC.AA) are placed in the UK so $GEOD(NIC.AA) = 1$.

Formula: Count the ordered set of geographical locations identifying where the authoritative name servers are located. In OCL this can be implements by executing the following query: $GEOD(z)$ = ***self.nameservers.locatedat.name->asOrderedSet()->size()***

Metric: **Controlability**

Symbol: $Co(z)$

Definition: This is a measure of how much control the system administrator can exert on his zone as a result of hosting his zone within name servers outside of his administrative jurisdiction or authority.

Usability: The metric can be used to assess the amount of coordination needed for the maintenance of a consistent and stable copy of the zone between the various hosting name servers.

How to Measure Calculate the number of in-bailiwick and out-of-bailiwick name servers of the zone.

Example: For the example model instance in Figure 7.6 Is(Example.COM)=2 and Os(EXAMPLE.COM)=2 then Co(EXAMPLE.COM)=$\frac{2}{2+2}$ = 0.5. While Co(EXAMPLE.NET)=$\frac{0}{2} = 0$.

Formula: $Co(z) = \frac{Is(z)}{Is(z)+Os(z)}$

B.3 Measures of Dependency/Influence

Metric:	**Influencing Zones**
Symbol:	$I(z)$
Definition:	The set of zones included in the model instance and influencing the resolution of domain names under a certain zone z.
Usability:	This metric it is generally representative of the diversity of zones that influence resolution of domain names under zone z and as indication of the trusted computing base of the zone.
How to Measure	Count all zones present within the model instance of the zone
Example:	For the example model instance in Figure 7.6, the total Iz(EXAMPLE.COM)=5.
Formula:	Using OCL it can be calculated on a zone based on this iterated query over all zone elements in the model: ***self.data.zones->collect(z : Zone | z.subzones->asOrderedSet())->size().***

Metric:	**Directly Configured Zones**
Symbol:	$DCZ(z)$
Definition:	The set of zones included in the model instance and influencing the resolution of domain names under a certain zone z and explicitly configured by the zone's administrator.
Usability:	This metric it is generally representative of the explicitly configured zones that influence resolution of domain names under zone z.
How to Measure	Count all the zones directly associated with the authoritative name servers of the zone. Included in this set are the parent zones of any ANS or alias targets in the model.

Example:	For the example model instance in Figure 7.6, DCS(EXAMPLE.COM)= (example.com and example.net)=2.
Formula:	Using OCL it can be calculated on a zone based on this query: ***self.nameservers.hasnamein->asOrderedSet()->size().***

Metric:	**Third Party Zones**
Symbol:	$TPZ(z)$
Definition:	The set of zones included in the model instance and influencing the resolution of domain names under a certain zone and stemming from the inter-zone dependencies.
Usability:	This metric it is generally representative of the influence of zones that are not explicitly configured by the zone administrator. They are the result of the inter-zone name dependencies within the model instance.
How to Measure	Count all zones present within the model instance of the zone and subtract from it the number of the directly configured zones.
Example:	For the example model instance in Figure 7.6, the total Iz(EXAMPLE.COM)=5.
Formula:	$TPZ(z) = Zones(Z) - DCZ(z)$.

Metric:	**Directly Configured Organisations**
Symbol:	$DCO(z)$

Definition:	The set of organisations managing the zones included in the model instance and influencing the resolution of domain names under a certain zone z and explicitly configured by the zone's administrator.
Usability:	This metric it is generally representative of the explicitly configured organisations that influence resolution of domain names under zone z. These organisations may have close mutual cooperation agreements to ensure certain service levels of the system.
How to Measure	Count all the organisations managing the directly configured authoritative name servers of the zone.
Example:	For the example model instance in Figure 7.6, $DCO(EXAMPLE.COM) = (ORG - AandORG - B) = 2$ while $DCO(EXAMPLE.NET) = (ORG - B) = 1$.
Formula:	Using OCL it can be calculated on a zone based on this query: ***self.nameservers.managedby->asOrderedSet()->size().***

Metric:	**Third Party Organisations**
Symbol:	$TPZ(z)$
Definition:	The set of organisations managing the zones included in the model instance and influencing the resolution of domain names under a certain zone z and explicitly configured by the zone's administrator.

Usability: This metric it is generally representative of the explicitly configured organisations that influence resolution of domain names under zone z. These organisations may have close mutual cooperation agreements to ensure certain service levels of the system.

How to Measure Count all the organisations managing name servers in the model and subtract the number of organisations managing the directly configured authoritative name servers of the zone.

Example: For the example model instance in Figure 7.6, $TPO(EXAMPLE.COM) = (ORG - B) = 1$.

Formula: $TPO(z) = ORG(Z) - DCO(z)$.

Metric: **Dependency Cycles**

Symbol: $Cycles(z)$

Definition: A cyclic zone dependency occurs when two or more zones depend on each other in a circular way.

Usability: A name which is a dependency of itself is effectively "unavailable". Cyclic dependencies potentially decrease the redundancy of a domain name for an ignorant resolver. It also affects other quality attributes of the operational system.

How to Measure Query the model about the presence of a pattern specifies two zones with cyclic dependency of ($Zone_1 \xrightarrow{nameservers} Server_1 \xrightarrow{hasnamein} Zone_2 \xrightarrow{nameservers} Server_2 \xrightarrow{hasnaemin} Zone_1$) sequence path.

Example: The model instance in Figure 7.6 has 4 cycles included in the resolution paths of the domains under the zone EXAMPLE.COM.

Formula:	Henshin-Based cyclic dependency pattern matching rule as shown in Figure 6.4.

B.4 Measures of Delegation and Inheritance

Metric:	**Depth**
Symbol:	$D(z)$
Definition:	The depth of a zone is defined as its distance from the root zone.
Usability:	Each ancestral zone z(i) contributes to the failure potential for zone z, as it is an additional requirement of DNS and DNSSEC correctness that must be consistent.
How to Measure	Calculate the number of ancestry zones of zone z which are zones with Parent(z) relationship in the model instance.
Metric Notations	Zone z has ancestry $z(0), z(1), ..., z(m)$ comprised of $m+1$ zones and has a depth of m.
Example:	For example, $zone(SUB.EXAMPLE.COM)$ has $m = 3$. The $Depth(z) = 4$ since it spans the zones $(ROOT, COM, EXAMPLE.COM and SUB.EXAMPLE.COM)$.
Formula:	$Depth(z) = m + 1$

Metric:	**Minimum Query Path**
Symbol:	$MinQP(z)$

Definition: The $MinQP(z)$ for a domain name refers to the minimum number of authoritative name servers which a DNS resolver must query to resolve the name under a particular zone z.

Usability: Domain names with larger MinQPs may result in additional resolution overhead for an ignorant DNS resolver. However, caching minimizes overhead of subsequent lookups.

How to Measure An Algorithm that recursively performs a conversion of the Boolean expression for resolving domains under zone z hrough all possible paths through every authoritative name server, into disjunctive normal form (DNF). Each resulting conjunction corresponds to a complete set of servers that may be queried to resolve domains under z. The set of conjunctions having minimum size is returned.

Example: The MinQP(NIC.AA) shown in the model instance presented in Question 9 in Appendix E is 3. (Resolution path through ns1.nic.aa, Zone(NIC.AA) name server then one name server of zone(AA) and finally one of zone(ROOT) name servers.

Formula: The DNF algorithm was implemented in Java as part of the ISDR method techniques.

Metric: **Maximum Query Path**

Symbol: $MaxQP(z)$

Definition: The $MaxQP(z)$ for a domain name refers to the maximum number of authoritative name servers which a DNS resolver must query to resolve the name under a particular zone z.

Usability: Domain names with large MaxQPs result in additional resolution overhead.

How to Measure An Algorithm that recursively performs a conversion of the Boolean expression for resolving domains under zone z through all possible paths through every authoritative name server, into disjunctive normal form (DNF). Each resulting conjunction corresponds to a complete set of servers that may be queried to resolve domains under z. The set of conjunctions having maximum size is returned.

Example: The MaxQP(NIC.AA) shown in the model instance presented in Question 9 in Appendix E through the ANS ns1.info represents the longest query path and equals to 6. (Resolution path through one name server of the following zones (NIC.AA, AA, ROOT, INFO, ORG, EDU).

Formula: The DNF algorithm was implemented in Java as part of the ISDR method techniques.

Metric: **Average Query Path**

Symbol: $AQP(z)$

Definition: The $AQP(z)$ for a domain name refers to the average number of all query paths through all authoritative name servers which a DNS resolver must query to resolve the name under a particular zone z.

Usability: Domain names with larger AQPs results in additional resolution overhead for an ignorant DNS resolver. It may also implies impacts on other quality attributes of the domain name system.

How to Measure An Algorithm that recursively performs a conversion of the Boolean expression for resolving domains under zone z, into disjunctive normal form (DNF). Each resulting conjunction corresponds to a complete set of servers that may be queried to resolve domains under z. The average of all sets of calculated conjunctions is returned.

Example: The DNF(NIC.AA) shown in the model instance presented in Question 9 in Appendix E is Average(DNF(ns1.nic.aa), DNF(ns2.nic.aa)+DNF(ns1.info) which equals to Average(3+3+6) =4. .

Formula: The DNF algorithm was implemented in Java as part of the ISDR method techniques.

Appendix C

Bad Smells Catalogue

Name:	**Unnecessary RR (Information Leakage)..**
Type:	Inter-Zone and Measurable.
Insp. Planes:	Data Layer and the Zone's Resource Records.
Occurrences:	The presence of certain RRs (such as HINFO Records) reveals sensitive information needed for launching targeted attacks. The HINFO RR is generally used to carry information about a host such as the O/S name, version, its latest installed patch and other sensitive information. This information could be potentially used to launch targeted attacks on such hosts. Depending upon whether the attacked host is a DNS name server, mail server or web server, the adverse consequences of such attacks could be different.
Quality Impacts:	Serious impacts on system security.
Detection:	Quering the model for the occurrence of such records within the DataLayer. A graph-based Henshin rule can be used to detect such structural bad smell.
Refactorings:	RemoveHinfoRecord.

Name:	**Large Parameter Value (Zone Drift)**.
Type:	Inter-Zone and Measurable
Insp. Planes:	Data Layer and Zone's SOA Record.
Occurrences:	Large parameter values in the RDATA portion of the zone's SOA Record could result in either no answers or obsolete (unusable) answers resulting in denial of service. For example the "refresh" data item in the RDATA field of a SOA RR specifies the frequency with which secondary authoritative name servers should initiate zone transfers in order to keep their zone file contents in synch with the primary authoritative name servers. Similarly the "retry" data item in the same field of the same RR tells the frequency with which the secondary name server should make retry attempts in case a refresh attempt is unsuccessful. The "Expiry" data item in the same RDATA field denotes the time duration after which the secondary name server should make no more attempts at refresh but instead lets its zone file contents expire.
Quality Impacts:	Large value for the data items (i.e., "refresh", "retry" and "expiry") could result in mismatch of data between secondary name servers (that provide fault tolerance) and primary name server resulting in serving either a empty response or obsolete response. Frequent occurrences of zone drift could potentially result in denial of service to DNS resolvers using those secondary name servers. This type of bad smells has direct impact on degrading the availability, security and stability of the system.

Detection:	Check the value of the various parameters within the SOA resource record model component and compare them with the corresponding recommended values.
Refactorings:	AdjustSOAParameters.

Name:	**Small Parameter Value (Zone Thrash)**.
Type:	Inter-Zone and Measurable.
Insp. Planes:	Data Layer and Zone's SOA Record.
Occurrences:	For example if the "refresh" value in SOA RR is very small, the secondary authoritative name server will be performing frequent zone transfers from the primary authoritative name server. As another example, if the "MinTTL" data item in a SOA RR is small, those RRs that have used this default value will expire much more quickly in the cache of the caching name server. Hence the DNS resolver will have to make more frequent queries to the authoritative name servers instead of relying on its cache. This will result in more frequent queries to primary and/or secondary authoritative name servers and has the potential to degrade performance (by increasing query response time). This situation is called "Zone Thrash".
Quality Impacts:	A different set of security impacts occur if the parameter values in RDATA field of the SOA Record are too small. This type of bad smells has also direct impact on degrading the availability, and stability of the system.
Detection:	Check the value of the various parameters within the SOA resource record model component and compare them with the corresponding recommended values.
Refactorings:	AdjustSOAParameters.

Name: **Ill Formed Associative RR**.

Type: Inter-Zone and Structural.

Insp. Planes: Data Layer and Zone's Resource Records.

Occurrences: Access to certain domains and/or services require two RRs (or RRsets) in the zone file to be retrieved. The first RR (RRset) will only provide the fully qualified domain name (FQDN) of the domain/service (e.g., NS and MX RRs that provide the FQDN (e.g., ns1.example.com) of the name server and mail server respectively for a domain). The second RR (RRset) then provides the IP address for the retrieved FQDN through an A/AAAA RR (host to IP Address mapping RR). The second RR (i.e., A/AAAA RR) is called the associative RR since it provides the actual network address (IP address) to reach the host providing a specific service that is referenced in the first RR (NS or MX RR). If the associative RR either contains an invalid IPv4 /IPv6 address or the RR itself is missing, then the host providing the internet-based service becomes inaccessible and hence is susceptible to denial of service attacks.

Quality Impacts: Missing or ill-formed associative RRs results in inaccessibility and severely degrading the availability of Internet domain names and associated services.

Detection: Check the referential integrity (associations and references) between the related resource records such as NSRecords and ARecords. A graph-based Henshin rule can be used to detect the presence of the associative resource records and check if the reference to the same logical or physical servers.

Refactorings: Create/DeleteARecord, Create/DeleteNSRecord and Create/DeleteDSRecord.

Name:	**Missing RR**.
Type:	Single-Type and Structural.
Insp. Planes:	Data Layes and Zone's Resource Records.
Occurrences:	Certain critical services such as name resolution and email transfer/access need to be hosted on multiple servers to provide fault tolerance. Hence there should be multiple RRs for RRtypes representing those services. Specifically multiple RRs should be present for all authoritative name servers associated with a domain.
Quality Impacts:	Missing certain resource records will have a serious implications on the availability and stability of the system.
Detection:	Check the presence of redundant instances of resource records. A graph-based Henshin rule can be directly used to detect the presence/absence of such resource records in the model instance.
Refactorings:	Create/DeleteARecord, Create/DeleteNSRecord and Create/DeleteDSRecord.

Name:	**Incorrect Parameter Value.**.
Type:	Inter-Type and Lexical.
Insp. Planes:	Data Layer and Zone's DNSSEC related Resource Records.
Occurrences:	Incorrect parameter values in the zone file's digital signature records (RRSIG RRs) will render the DNSSEC security service non-usable. For example, if the signature is not currently valid (current date is not between signature inception and expiry dates), then a DNSSEC-aware resolver will not use it to validate the integrity of the RRset covered by the signature.

Quality Impacts: This type of bad smells has sever impacts on the security and stability of the domain name system.

Detection: Check the DNSSEC related RRSIG records for the signature expiration parameter and make sure the TTL value matches the TTL value of the RRset it covers. DNSSEC resource records are not shown within the DNS Model for space considerations but they are dealt with the same as pther critical infrastructure DNS resource records in the model.

Refactorings: AdjustRRSIGParameters.

Name: **Ambiguous Data.**.

Type: Inter-Type and Lexical.

Insp. Planes: Data Layer and all Zones and associated Resource Records.

Occurrences: Certain data content scenarios are high risk from the point of security but nonetheless needs policies for proper usage of their underlying RRs. Examples of such scenarios are multiple IP addresses for a given host (i.e., multiple A RRs for a given host identified by a fully qualified domain name (FQDN)).

Quality Impacts: The availability and security of the system is highly affected by this type of bad smells.

Detection: Check for the presence of duplicate records or components within the model instance.

Refactorings: Create/DeleteARecord and Create/DeleteNSRecord.

Name: **Small Number of ANS/Absence of Multiple RRs**.

Type: Inter-Type and Measurable.

Insp. Planes: Control Layer and Server Components.

Occurrences: A major reason for having multiple servers for each zone is to allow information from the zone to be available widely and reliably to clients throughout the Internet, that is, throughout the world, even when one server is unavailable or unreachable. Multiple servers also spread the name resolution load, and improve the overall efficiency of the system by placing servers nearer to the resolvers.

Quality Impacts: This smell has direct impacts on the availability, stability and resiliency of the overall system.

Detection: Count the number of name servers associated with the zone and check if they are above a pre-defined threshold value. The threshold value is set based on the local administrator needs and governing policies.

Refactorings: Create/DeleteARecords, Create/DeleteNSecords, AddNewNet, AddNewGeoLocation and AddNewServer.

Name: **Invalid Trust Anchor**.

Type: Inter-Type and Structural.

Insp. Planes: Data Layer and the Zone's DNSSEC-related Resource Records.

Occurrences: DNSSEC adds complexity to the requirements for name resolution, and increases the potential for failure. Any server or zone misconfiguration in the line of trust between anchor and query name widens the target of error. To validate the DNSSEC verification keys, DNS resolvers obtain a corresponding a DS RR from the parent zone, which contains a hash of the public key of the child; the resolver accepts the DNSKEY of the child as authentic if the hashed value in DNSKEY is the same as the value in the DS record at the parent, and that DS record is properly signed (in a corresponding RRSIG record). Since the DS record at the parent is signed with the DNSKEY of the parent, authenticity is guaranteed and the trust chain is secure.

Quality Impacts: This bad smell affects mainly the security quality attribute of the system..

Detection: Detection is done by checking the continuity of the trust chain through the verification of keys between the current zone and its ancestors as well as the ancestors of all authoritative name servers of that particular zone all the way to the *root* zone. If DS RRs are present in a parent zone, but none of them correspond to any self-signing DNSKEYs in the child zone, then the chain of trust is broken.

Refactorings: Create/DeleteDSRecord.

Name: **Untrusted Peer Organisation/Corrupted Parent**.

Type: Intra-Type and Lexical.

Insp. Planes: DNSModel Instance.

Occurrences: A zone trusts its parent to perform the delegation on its behalf. This trust model is appropriate for zones within the same organisation, however, there is risk involved in trusting the parent not to abuse inter-organisational trust relationship. Since the organisation controlling the parent zone is holding the secret signing keys, it can forge the delegation records of the child and then provide this forged (yet correctly signed) DS RR, thus misleading the clients of the child zone into accepting a forged DNSKEY and trusting resource records signed with it.

Quality Impacts: Availability, security and stability of the system is affected by the presence of this bad smell.

Detection: Check the consistency and correctness of all types of key distribution between zones with inter-organisational relationship by following the the child/parent delegation chain up to the *root* zone. For peer organisations, this should be done for all authoritative name servers and their ancestors up to the *root* zone.

Refactorings: Add/DeleteZone and Create/DeleteDSRecord.

Name: **Large Attack Surface**.

Type: Intra-Zone and Measurable.

Insp. Planes: DNSModel Instance.

Occurrences: The *attack surface* metric, $AS(z)$, is an indicator of the system's security. The larger the attack surface, the more insecure the system. In our model, AS(z) is the total number of (Zones, Servers and organisations) in the model.

Quality Impacts: Large Attack Surface affects all the quality attributes of the system.

Detection: Calculating the *Attack Surface*, $AS(z)$, metric and comparing them to certain thresholds set based on local policies.

Refactorings: Add/DeleteZone, Add/DeleteOrganisation, and Add/Rename/DeleteServer.

Name: **Excessive Zone Influence**.

Type: Intra-Zone and Measurable.

Insp. Planes: Data Layer of the DNS Model.

Occurrences: The set of influential zones is a measure of the Trusted Computing Base (TCB) of the zone. It is generally representative of the diversity of directly and third party zones that influence resolution of domain names under z.

Quality Impacts: Security and stability of the system are affected by the presence of this bad smell.

Detection: Calculating the *Directly Configured Zones*, $DCZ(z)$, metric and *Third Party Zones*, $TPZ(z)$, metric and comparing them to certain thresholds set based on local policies.

Refactorings: DeleteZone, DeleteSOARecord, RenameServer and Add/DeleteServer..

Name: **False-Redundancy**.

Type: Intra-Zone and Measurable.

Insp. Planes: DNS Model Instance.

Occurrences: DNS uses redundancy as one of the two mechanisms for high availability - the other one is caching.

Quality Impacts: If all servers comprising the redundancy of a domain name were to fail, then the name would be rendered unavailable so this smell has direct impact on the availability and resiliency of the system.

Detection: If the value of the *redundancy*, $R(z)$, metric for a zone is less than the number of authoritative name servers of that particular zone, $ANS(z)$, then the true redundancy is less than the redundancy configured by the zone administrator. False redundancy could also result from a narrower bottleneck in downstream resolution query paths of domains under z.

Refactorings: Create/DeleteARecords, Create/DeleteNSecords, AddNewNet, AddNewGeoLocation and AddNewServer.

Name: **Excessive Zone Complexity**.

Type: Intra-Zone and Measurable.

Insp. Planes: DNSModel.

Occurrences: One important necessity is careful coordination both hierarchically (i.e., between parent and child zones) or laterally, between organizations hosting each others' data.

Quality Impacts: High zone complexity value increases the failure potential for signed and unsigned zones because they indicate more areas where problems may occur. Direct impacts for such smell are evident on the stability and resiliency of the system.

Detection: The hierarchical relationship complexity is measured using the *hierarchical reduction potential* metric, HRP, and the lateral coordination complexity is measured by the *administrative complexity* metric, $AC(Z)$. Comparing these metrics with a predefined threshold values will reveal the existence of such a bad smell in the model instance.

Refactorings: DeleteZone, DeleteSOARecord, RenameServer, Add/DeleteServer.

Name: **Diminished Redundancy**.

Type: Measurable and Inter-zone.

Insp. Planes: Control Layer.

Occurrences: A number of problems in DNS operations today are attributable to poor choices of secondary servers for DNS zones. The geographic placement as well as the diversity of network connectivity exhibited by the set of DNS servers for a zone can increase the reliability of that zone as well as improve overall network performance and access characteristics. When all redundant servers are located within the same physical location, connected to the same network, placed within the same address prefix.

Quality Impacts: Reduced availability, decreased resilience, and the system become susceptible to single point of failure at certain granularity.

Detection: Queries on the dependency graph regarding the following metrics: a) number of authoritative name servers, b) geographical locations servers are placed in, c) networks connected to, and d) BGP prefixes.

Refactorings: Applying the MoveServerLocation refactoring rule will ensure the availability of the zone and its resilience to a single point of failure.

Name: **Cyclic Dependency**.

Type: Intra-Zone and Structural.

Insp. Planes: Data and Control Layers.

Occurrences: Cyclic zone dependency occurs when two or more zones depend on each other in a circular way. This type of interdependency creates a "chicken and egg" problem; one cannot resolve a name in zone Z_1 without first resolving a name in Z_2 and vice versa.

Quality Impacts: Reduced availability and reduced resiliency.

Detection: Is there any cycle in the Dependency Graph?. A query on the DNS Model Instance that can be implemented using Henshin as shown in Figure 6.4.

Refactorings: Add a glue record (ARecord) for the (out-of-bailiwick) authoritative name servers involved in the cycle in the zone file.

Name: **Non-Optimal Query Path**.

Type: Intra-Zone and Measurable.

Insp. Planes: The DNSModel Instance.

Occurrences: When the number of servers which a resolver must query to resolve the name is larger that the MinQP.

Quality Impacts: Name servers with larger query paths result in additional resolution overhead for an ignorant resolver so the quality attribute directly affected by this smell is the availability attribute.

Detection: Calculating the MinQP metric for a zone and compare each query resolution path through each name server with the minimum value. The name server with query path > MinQP contributes to the occurrence of this bad smell.

Refactorings: Create/DeleteNSRecord and Create/DeleteARecord, Create/DeleteZone, and Add/DeleteServer.

Name: **Delegation Inconsistency**.

Type: Intra-Zone and Structural.

Insp. Planes: Data Layer and Zone/Parent Zone NS Records .

Occurrences: When a parent zone P delegates part of its name space to a child zone C, P stores a list of NS resource records for the authoritative servers of zone C. This list of NS resource records are kept both at the parent and the child zone. In order to maintain a consistent answer sets of authoritative name servers for a certain zone and its contents, it is essential to maintain the same NS resource record set in both the parent and child zones.

Quality Impacts: Lacking this consistency reflects the failure of coordination among NS servers' operators. Such inconsistency will affect the availability and stability of the DNS query process for this particular zone.

Detection: Pattern-based smells (i.e., smells that are detectable by the existence of specific anti-pattern subgraphs) can be specified by Henshin rules. This smell can be detected by checking for any mismatches between the set of NS records within a zone with its parents' set of NS records defined as name servers for that particular zone.

Refactorings: Create/DeleteNSRecord and Create/DeleteARecord.

Appendix D

Refactoring Catalogue

Name:	**CreateARecord.**
Context:	Zone Model Component.
Priority:	(1), Zone's administrator own decisions.
Pre-conditions:	The current zone should be a child zone since the *ARecord* has to be created in both the current *zone* and its Parent *zone* in order to resolve the names of in-bailiwick and out-of-bailiwick name servers properly.
Parameters:	*servername: EString.* The name of the *Server* that the *ARecord pointsto.*
Final Checks:	If the input name *Server* already exists in the model instance's *ControlLayer.*
Quality Impacts:	Creating "glue" *ARecord* improves the availability of the domain name since it affects the ability of external DNS(s) to correctly resolve the name of the assigned name server and hence make the domains under the zone resolvable.

Refactoring Steps:	1. From the model instance's *ControlLayer*, locate the name *Server* which the *ARecord* will *pointto* to get the parameter *servername*. 2. Execute the refactoring on the context *zone* (*SelectedEObject*), then input the parameter *servername* of the *Server* to be associated with the new *ARecord*.
Implementation:	Implementation example using Henshin graph-based rules and units is shown in Figure 6.5, Figure 6.6 and Figure 6.7.

Name:	**DeleteARecord.**
Context:	Zone Model Component.
Priority:	(1) Zone's Administrator Own Decisions.
Pre-conditions:	The current zone should be a child zone since the *ARecord* has to be deleted from both the current zone and its Parent *zone* in order to prevent any occurrence of the *Delegation Inconsistency* bad smell as a result of deleting the record.
Final Checks:	The referenced *point(ed)to server* has to be present in the model instance's *ControlLayer*.
Parameters:	*servername: EString.* The name of the *Server* that the *ARecord pointsto*.
Final Checks:	None.
Quality Impacts:	Deleting "glue" *ARecord* contribute to the elimination of the *Delegation Inconsistency* bad smell so the availability of the zone is improved.

Refactoring Steps:	1. From the model instance's *ControlLayer*, locate the name *Server* which the *ARecord point(s)to* to get the parameter *servername*. 2. Execute the refactoring on the context *zone* (*SelectedEObject*), then input the parameter *servername* of the *Server* that is associated with the *ARecord* to be deleted.

Name:	**CreateNSRecord.**
Context:	Zone Model Component.
Priority:	(1), Zone's administrator own decisions.
Pre-conditions:	The current zone should be a child zone since the *NSRecord* has to be created in both the current *zone* and its Parent *zone* in order to resolve the names of in-bailiwick and out-of-bailiwick name servers properly.
Parameters:	*servername: EString.* The name of the *Server* that the *NSRecord refersto.*
Final Checks:	If the input name *Server* already exists in the model instance's *ControlLayer.*
Quality Impacts:	Creating *NSRecord* improves the availability of the domain name since it affects the ability of external DNS(s) to correctly resolve the name of the assigned name server and hence make the domains under the zone resolvable.
Refactoring Steps:	1. Locate the name *Server* which the *NSRecord* will *refersto* to get the parameter *servername.* 2. Execute the refactoring on the context *zone* (*SelectedEObject*), then input the parameter *servername* of the *Server* to be associated with the new *NSRecord.*

Implementation:	Implementation example using Henshin graph-based rules and units is similar to the refactorings shown in Figure 6.5, Figure 6.6 and Figure 6.7.

Name:	**DeleteNSRecord.**
Context:	Zone Model Component.
Priority:	(1) Zone's Administrator Own Decisions.
Pre-conditions:	The current zone should be a child zone since the *NSRecord* has to be deleted from both the current zone and its Parent *zone* in order to prevent any occurrence of the *Delegation Inconsistency* bad smell as a result of deleting the record.
Final Checks:	The referenced *refersto server* has to be present in the model instance's *ControlLayer*.
Parameters:	*servername: EString.* The name of the *Server* that the *NSRecord refersto*.
Final Checks:	None.
Quality Impacts:	Deleting a *NSRecord* contribute to the elimination of the *Delegation Inconsistency* bad smell so the availability of the zone is improved.
Refactoring Steps:	1. From the model instance's *ControlLayer*, locate the name *Server* which the *NSRecord refersto*, to get the parameter *servername*. 2. Execute the refactoring on the context *zone* (*SelectedEObject*), then input the parameter *servername* of the *Server* that is associated with the *NSRecord* to be deleted.

Name:	**DeleteHinfoRecord.**
Context:	Zone Model Component.
Priority:	(1) Zone's Administrator Own Decisions.
Pre-conditions:	Presence of unnecessary *HinfoRecord* within the concerned *zone*.
Quality Impacts:	Deleting the *HinfoRecord* will improve the security of the system since it prevents any information leakage that can be exploited by attackers.
Parameters:	None.
Refactoring Steps:	1. Locate the *HinfoRecord* within the concerned *zone*. 2. Execute the refactoring on the context *zone* (*SelectedEObject*), where the *HinfoRecord* is to be deleted.

Name:	**CreateSOARecord.**
Context:	Zone Model Component.
Priority:	(1) Zone's Administrator Own Decisions.
Pre-conditions:	There should not be any *SOARecord* associated with the concerned *zone*.
Quality Impacts:	Improved availability of the zone since the zone will not be resolvable without a correctly configured start of authority record (*SOARecord*).

Parameters:

- *admin: EString.* An attribute specifies the mailbox of the person responsible for this zone, *primary: EString.* The FQDN of the name server that was the original or primary source of data for this zone,
- *serial: EInt.* The version number of the original copy of the zone file, *refresh: EInt.* Represents the time interval before the zone should be refreshed.
- *retry: EInt.* Represents the time interval that should elapse before a failed refresh should be retried, *expire: EInt.* Represents the value that specifies the upper limit on the time interval that can elapse before the zone is no longer authoritative, *minTTL: EInt.* Represents the time that should be exported with any RR from this *zone*.

Refactoring Steps:

.

1. Locate the zone element where the *SOARecord* will be created.
2. From the model instance's *ControlLayer*, locate the name *Server* which will be the primary source of data for the *zone* and will be used as the *primary* parameter.
3. Decide on best recommended values for the timers *serial, refresh, retry, expire and minTTL* used as parameters for the refactoring. These parameters have to be soundly used to avoid any bad smells regarding the timers of the *SOARecord*.
4. Execute the refactoring using the specified parameters.

Name: **DeleteSOARecord.**

Context:	Zone Model Component.
Priority:	(1) Zone's Administrator Own Decisions.
Pre-conditions:	A *SOARecord* has to be present in the model instance and associated with the selected *zone*.
Quality Impacts:	Deleting a zone *SOARecord* by itself will render the zone and its domains unresolvable. This refactoring is executed as part of a multi-refactorings units to reduce the HRP complexity of a zone as part of merging two zones or to remove a zone from the *DataLayer* of the model instance. This rfactoring affects both the availability and the stability of the system.
Parameters:	None.
Refactoring Steps:	1. Locate the *SOArecord* within the concerned *zone*. 2. Execute the refactoring on the context *zone* (*SelectedEObject*), where the *SOARecord* is to be deleted.

Name:	**AdjustSOAParameters.**
Context:	Zone, SOARecord Model Component.
Priority:	(1) Zone's Administrator Own Decisions.
Pre-conditions:	(1) A *SOARecord* has to be present in the model instance and associated with the selected *zone*.
Quality Impacts:	Adjusting the SOA timers to best recommended values will affect the availability, security, stability and resiliency of the DNS. These timers are vital to the basic functionality and performance of the system.

Parameters: SOARecord parameters as listed in the *CreateSOARecord* refactoring above. These parameters have to be soundly used to avoid any bad smells regarding the timers of the *SOARecord*.

Final Checks: (1) A name server that will be used as the primary name server (with *nameserver=primary* parameters) has to be present in the *ControlLayer* of the model instance.

Refactoring Steps:

1. Locate the zone element where the *SOARecord* parameters will be modified.
2. From the model instance's *ControlLayer*, locate the name *Server* which will be the primary source of data for the *zone* and will be used as the *primary* parameter.
3. Decide on best recommended values for the timers *serial, refresh, retry, expire and minTTL* used as parameters for the refactoring.
4. Execute the refactoring using the specified parameters.

Name: **CreateNewZone.**

Context: DataLayer Model Component.

Priority: (2) Coordination with other Zones' Administrators.

Pre-conditions: (1) Coordination with the new zone administrator to host a secondary zone in a name server with a name registered under their zone name. (2) Existence of a Parent Zone for the new zone in the *DataLayer* of the current model instance. This condition does not apply to the *ROOT* zone.

Quality Impacts: Creating a new zone contributes to improving stability of the system but increases the query overhead and consequently degrading the availability of the system. It affects the security of the system by increasing the attack surface and vulnerable points in the operational system. It increases the points of failure that affects the resiliency of the system.

Parameters: *newzonename: EString.* The name of the new zone to be created.

Final Checks: (1) No zone with the name *zonename* exists under the current *Parent Zone*.

Refactoring Steps:

1. Locate the zone element (Parent Zone) where the new *zone* will be created.
2. Execute the refactoring using the specified *newzonename* parameters. In order to ensure the model correctness and consistency, this refactoring should be followed by a *CreateSOArecord* to define the Start Of Authority for the new zone and its associated parameters.

Name: **DeleteZone.**

Context: Zone Model Element.

Priority: (2) Coordination with other Zones' Administrators.

Pre-conditions: (1) Coordination with the zone administrator for the zone to be deleted to remove any copy of the zone data from the secondary name server with a name registered under that zone name. (2) The current *zone* should not be a parent of any sub-zone or associated with any resource records in the model instance.

Quality Impacts:	Deleting a zone contributes to improving the security, stability and resiliency of the system by reducing the third party zones, servers and organisations involved in the resolution process as well as reducing the attack surface of the system.
Parameters:	None.
Final Checks:	None.
Refactoring Steps:	1. Locate the zone element that will be deleted. 2. Execute the refactoring on the specified zone. In order to ensure the model correctness and consistency, this refactoring should be followed by checking if there is any dangling components (servers, zones ,organisations and resource records) of the model which has to be removed from the model instance.

Name:	**MergeZones.**
Context:	Zone Model Component.
Priority:	(2) Coordination with other Zones' Administrators.
Pre-conditions:	(1) The zone to be merged should have a parent *zone* within the *DataLayer* of the model instance. (2) The zone to be merged should not have any sub-zones defined within it and no resource record is associated with it (empty zone).
Quality Impacts:	Merging two zones has direct impact on the stability and resiliency of the system by reducing the hierarchical complexity of the zone. It also improved the security of the system by reducing the attack surface. Its impact on the availability is positive since it reduces the query path for resolving domain names under the zone.
Parameters:	None.

Refactoring Steps:

1. Locate the *zone* to be merged with its parent within the *Data-Layer* of the model instance.
2. Execute the refactoring on the specified *zone*. In order to ensure the model correctness and consistency, this refactoring should be followed by proceeded by modifying the resource records (except the zone's *SOARecord* which has to be deleted) to reflect the merging of the zones and moving these records to the *Parent Zone*.

Name:	**AddNewServer.**
Context:	ControlLayer Model Component.
Priority:	(3) Coordination with other Administrators, Cost and Access Permissions.
Pre-conditions:	Since this refactoring is tightly associated with costs, access control, permissions and coordination with other entities, it has to be done in full compliance with local policies and management constraints.
Quality Impacts:	Adding new servers will affect the availability quality attribute negatively by adding additional query paths. It will increase the attack surface of the system so it affects the security of the system. On the other hand it improves the stability of the system by increasing the authoritative name servers for a zone and improves the resiliency of the system.

Parameters:

- *servername: EString.* Defines the name of the server to be created.
- *zonename: EString.* Defines the name of the zone that this server will act as a name server.
- *hasnamein: EString.* Defines the name of the zone that hosts the name of the server to be added.
- *network: EString* Defines the AS number of the *network* that the *server* is connected to.
- *geoloc: EString* Defines the *GeoLocation* that the *server* is located at.
- *ipaddress: EString* Defines the IP Address assigned to the new *Server*.
- *orgname: EString* Defines the name of the *organisation* that manages this particular server.

Final Checks: (1) There should not be a *server* with a name identical to the parameter *servername* in the *ControlLayer* of the model instance. (2) There should be zones with the defined *zonename* and *hasnamein*, network with the defined *network* parameter, GeoLocation with the defined *geoloc* parameter, and organisation with the defined *orgname* parameter in the *ControlLayer* of the model instance.

Refactoring Steps:

1. Locate the *ControlLayer*) where the new *server* will be created.
2. Execute the refactoring using the specified parameters for the new *server*. In order to ensure the model correctness and consistency and prevent the introduction of any *Delegation-Inconsistency* bad smell, this refactoring should be followed by a *CreateNSrecord and CreateArecord* in the associated zone and its parent zone.

Name:	**RenameServer.**
Context:	Server Model Component.
Priority:	(3) Coordination with other Administrators, Cost and Access Permissions.
Pre-conditions:	Since this refactoring is tightly associated with costs, access control, permissions and coordination with other entities, it has to be done in full compliance with local policies and management constraints.
Quality Impacts:	Renaming servers will modify the whole model instance by removing the components related to the oldname and introducing new model components associated with the new server name. Usually this refactoring is used to reduce the overall model size and reduce the attack surface so all quality attributes are positively affected.
Parameters:	(1) *oldservername: EString.* The current server name, *newservername: EString.* The new name to be assigned to the server. (2) All parameters listed within the *AddNewServer* refactoring above.

Final Checks: (1) There should not be a *server* with a name identical to the *newservername* parameter. (2) All components associated with the newservername (such as the zonename, hasnamein, geolocation, network ..etc) should be present within the model instance.

Refactoring Steps:

1. Locate the *server*) that need to be renamed.
2. Execute the refactoring using the specified parameters for the new *server*. In order to ensure the model correctness and consistency and prevent the introduction of any *DelegationInconsistency* bad smell, this refactoring should be followed by modifying any reference to the old server by using *Delete/CreateNSrecord and Delete/CreateArecord* in the associated zone(s) and their parent zone(s).

Name: **DeleteServer.**

Context: Server Model Element.

Priority: (2) Coordination with other servers' managers and associated zone administrators.

Pre-conditions: (1) Coordination with the organisations managing/hosting the concerned *server* (and any associated *zone* administrator) to be deleted to remove any copy of the zone data from that particular name server. (2) The concerned *server* should not be assigned as name server or associated with any zone within the current model instance. (3) Since this refactoring is tightly associated with costs, access control, permissions and coordination with other entities, it has to be done in full compliance with local policies and management constraints.

Quality Impacts: Deleting a server contributes to improving the security, stability and resiliency of the system by reducing the third party servers, zones, and organisations involved in the resolution process as well as reducing the attack surface of the system.

Parameters: None.

Final Checks: None.

Refactoring Steps:

1. Locate the server element that will be deleted within the *ControlLayer* of the model instance.
2. Make sure that the server is not associated with any other component within the model instance.
3. Execute the refactoring on the specified server. In order to ensure the model correctness and consistency, this refactoring should be followed by checking if there is any dangling components (servers, zones ,organisations and resource records) which has to be removed from the model instance.

Name: **MoveServerLocation.**

Context: Server Model Component.

Priority: (3) Coordination with other Administrators, Cost and Access Permissions.

Pre-conditions: Since this refactoring is tightly associated with costs, access control, permissions and coordination with other entities, it has to be done in full compliance with local policies and management constraints.

Quality Impacts: Moving a server to a new geographical location will improve its resiliency due to improving its failure likelihood. This also improves the geographical diversity of the name servers. It may affect the security and availability negatively due to the increased query overhead and extra need for coordination with external and far away entities.

Parameters: *newlocation:EString.* The new geographical location that the server will be moved to.

Final Checks: The new *geoLocation* identified by the *newlocation* parameter should be present within the *ControlLayer* of the model instance.

Refactoring Steps:

1. Locate the new geographical location identified by the *newlocation* parameter within the *ControlLayer*.;
2. Locate the *server* that need to be moved within the *ControlLayer*
3. Execute the refactoring on the specified server. In order to ensure the model correctness and consistency, this refactoring should be followed by checking if the old *geoLocation* is not associated with any other *server* and then removed from the model instance accordingly.

Name:	**ModifyServerIP/MoveServerNet.**
Context:	Server Model Component.
Priority:	(3) Coordination with other network Administrators, Cost and Access Permissions.
Pre-conditions:	Since this refactoring is tightly associated with costs, access control, permissions and coordination with other entities, it has to be done in full compliance with local policies and management constraints.
Quality Impacts:	Moving a server to a new network improves its resiliency due to improving its failure likelihood. This also improves the network diversity of the name servers and avoids single points of failure. It may affect the security and availability negatively due to the extra need for coordination with external and far away entities.
Parameters:	*newnetwork:EString.* The new network AS number that the server will be connected to.
Final Checks:	The new *network* identified by the *newnetwork* parameter should be present within the *ControlLayer* of the model instance.

1. Locate the new network identified by the *newnetwork* AS parameter within the *ControlLayer.*;
2. Locate the *server* that need to be moved within the *ControlLayer*

Refactoring Steps:

3. Execute the refactoring on the specified server. In order to ensure the model correctness and consistency, this refactoring should be followed by checking if the old *network* AS number is not associated with any other *server* and then removed from the model instance accordingly.

Appendix E

DNS Operational Model Survey

E.1 Background

The goal of this study is to investigate the influence of the Domain Names System (DNS) configurations and deployment choices made by system administrators and zone operators (modelled as Dependency Graphs) and their impact on a subset of DNS operational quality attributes. The DNS Operational Model is an attempt to describe the Domain Name System operational world for a particular operational goal (detecting violations of the design and deployment principles) at the authoritative level. For detecting problems in the configuration and deployment of the DNS, we have to search for certain patterns representing those problems in the instances of the operational model of the system.

We appreciate if you can provide us with your valuable input by answering all the 30 questions listed below. Filling out this survey will take approximately 30-45 minutes of your valuable time.

E.2 General Questions

1. How many years have you been involved in DNS Management?

Less than	1	2	3	4	5	6	7	8	9	10	or more
	○	○	○	○	○	○	○	○	○	○	

2. What Top-Level-Domain (TLD) are you responsible for?

_____

3. How many domains are registered under your TLD?

_____

4. How familiar are you with current deployment structure of your TLD?

Never Heard About It	○
Heard about it but never used it	○
Looked at It	○
Looked at it in details	○
Designed it	○

5. How many times do you update your TLD zone every day?

Real-Time (click 0)	0	1	2	3	4	5	6	7	8	9	10	Times
	○	○	○	○	○	○	○	○	○	○	○	

6. How frequently have you changed the DNS structure for your TLD?

During the last 5 years	1	2	3	4	5	Times
	○	○	○	○	○	

7. How many security incidents have you faced?

During the last 5 years	1	2	3	4	5	6	7	8	9	10	Incidents
	○	○	○	○	○	○	○	○	○	○	

8. How many DNS configuration/deployment faults have you faced?

During the last 5 years	1	2	3	4	5	6	7	8	9	10	Incidents
	○	○	○	○	○	○	○	○	○	○	

E.3 Models and Metrics

Here is an example of the DNS Dependency Graph and how it reflects the physical and logical operational world of the DNS system from the Authoritative Zone Operator point of view. In the following sections, we present several dependency graphs of DNS configurations and deployment choices and would like to know their perceived operational quality attributes from your point of view.

DNS Qualities Definitions:

Availability: "The ability of a domain name to be reliably resolved using the DNS"

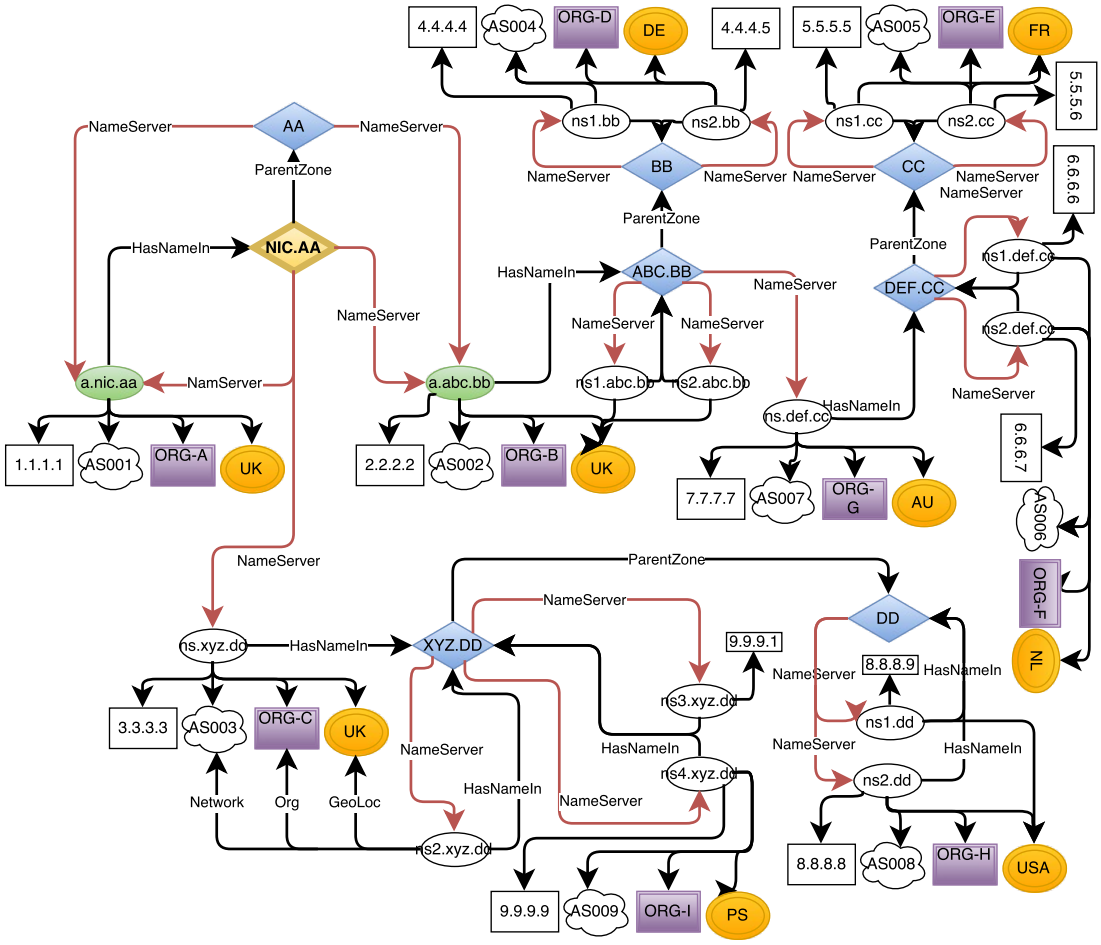
Security: "The ability of the components of the DNS to protect the integrity of DNS information and critical DNS system resources."

Stability: "The ability of the entire name resolution system and its component parts to be able to respond to DNS queries."

Resilience: "The ability of the DNS to provide and maintain an acceptable level of name resolution service in the face of faults and challenges to normal operations."

Please note that due to space limitations, and clarity of the images, we present parts of the DNS operational Models in the following questions. Please, based on your own experience, rate the perceived qualities of these model instances:

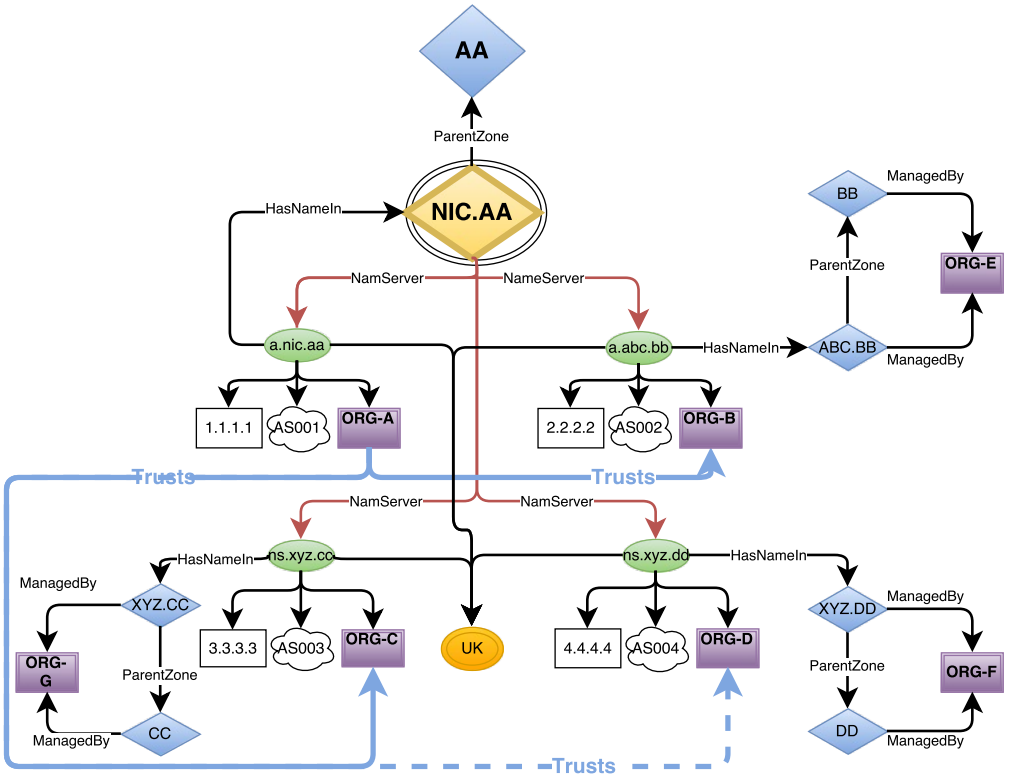
1. How do you rate the Quality Attributes of This DNS Model?



Quality Attribute	Very Low	Low	Medium	High	Very High
	1	2	3	4	5
Availability	◯	◯	◯	◯	◯
Security	◯	◯	◯	◯	◯
Stability	◯	◯	◯	◯	◯
Resiliency	◯	◯	◯	◯	◯

2. Can you justify or comment on your choices? _____

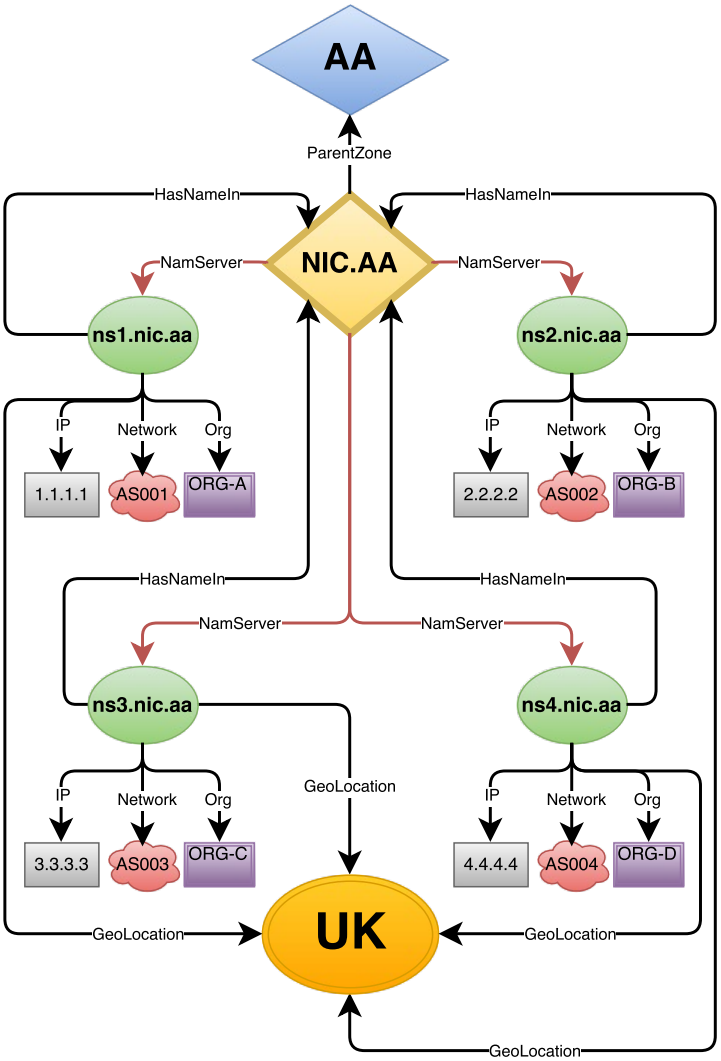
3. How do you rate the Quality Attributes of This DNS Model?



Quality Attribute	Very Low	Low	Medium	High	Very High
	1	2	3	4	5
Availability	◯	◯	◯	◯	◯
Security	◯	◯	◯	◯	◯
Stability	◯	◯	◯	◯	◯
Resiliency	◯	◯	◯	◯	◯

4. Can you justify or comment on your choices? ————

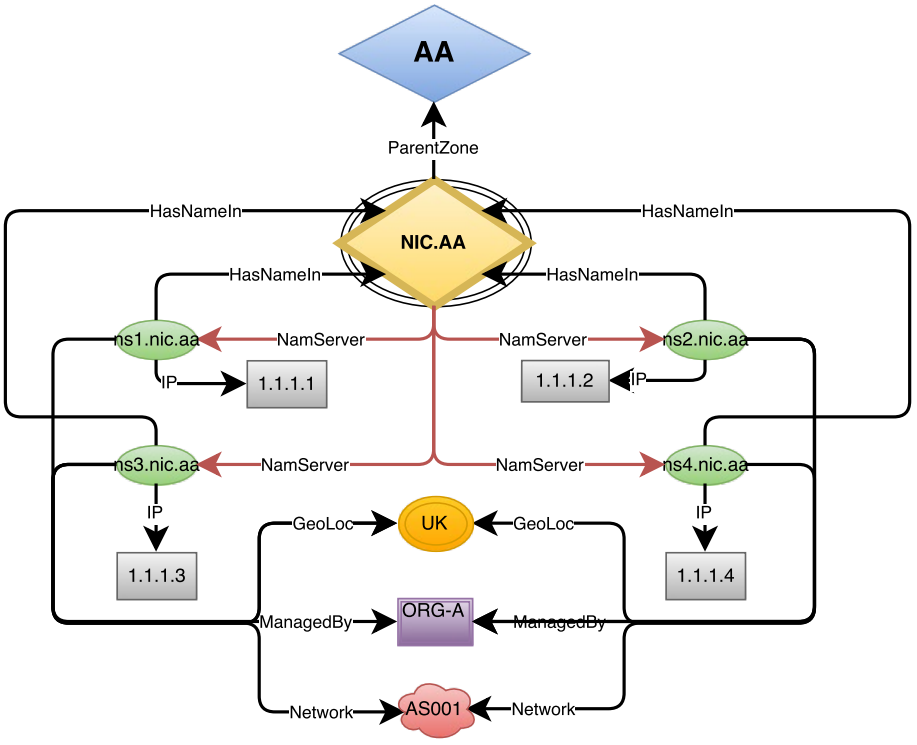
5. How do you rate the Quality Attributes of This DNS Model?



Quality Attribute	Very Low	Low	Medium	High	Very High
	1	2	3	4	5
Availability	◯	◯	◯	◯	◯
Security	◯	◯	◯	◯	◯
Stability	◯	◯	◯	◯	◯
Resiliency	◯	◯	◯	◯	◯

6. Can you justify or comment on your choices? ————

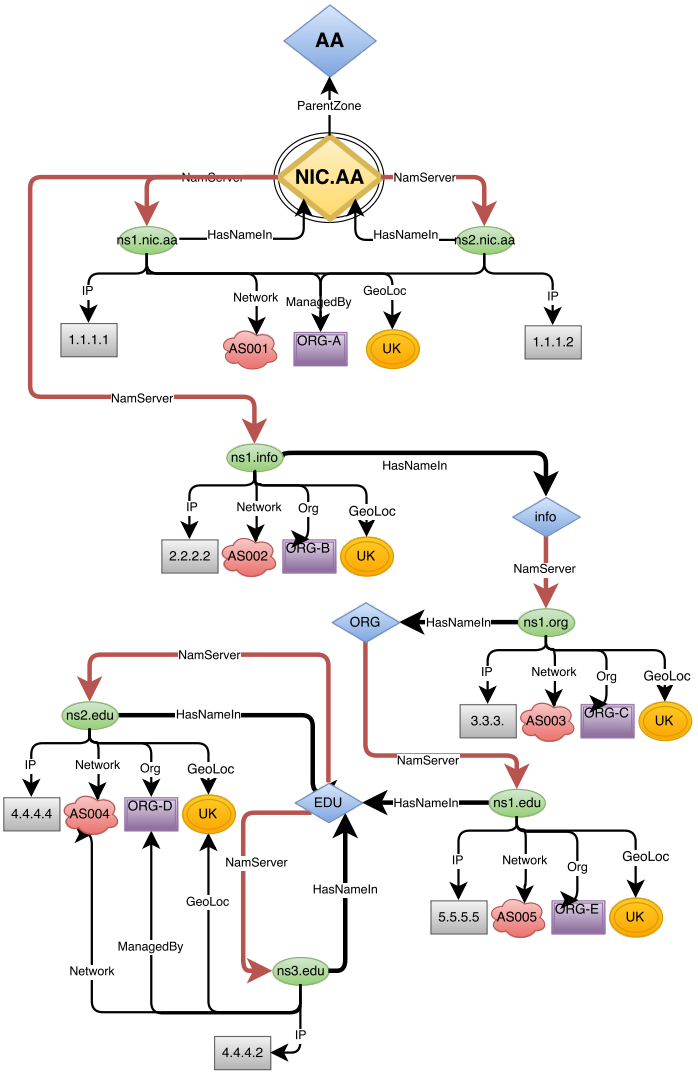
7. How do you rate the Quality Attributes of This DNS Model?



Quality Attribute	Very Low	Low	Medium	High	Very High
	1	2	3	4	5
Availability	◯	◯	◯	◯	◯
Security	◯	◯	◯	◯	◯
Stability	◯	◯	◯	◯	◯
Resiliency	◯	◯	◯	◯	◯

8. Can you justify or comment on your choices? ————

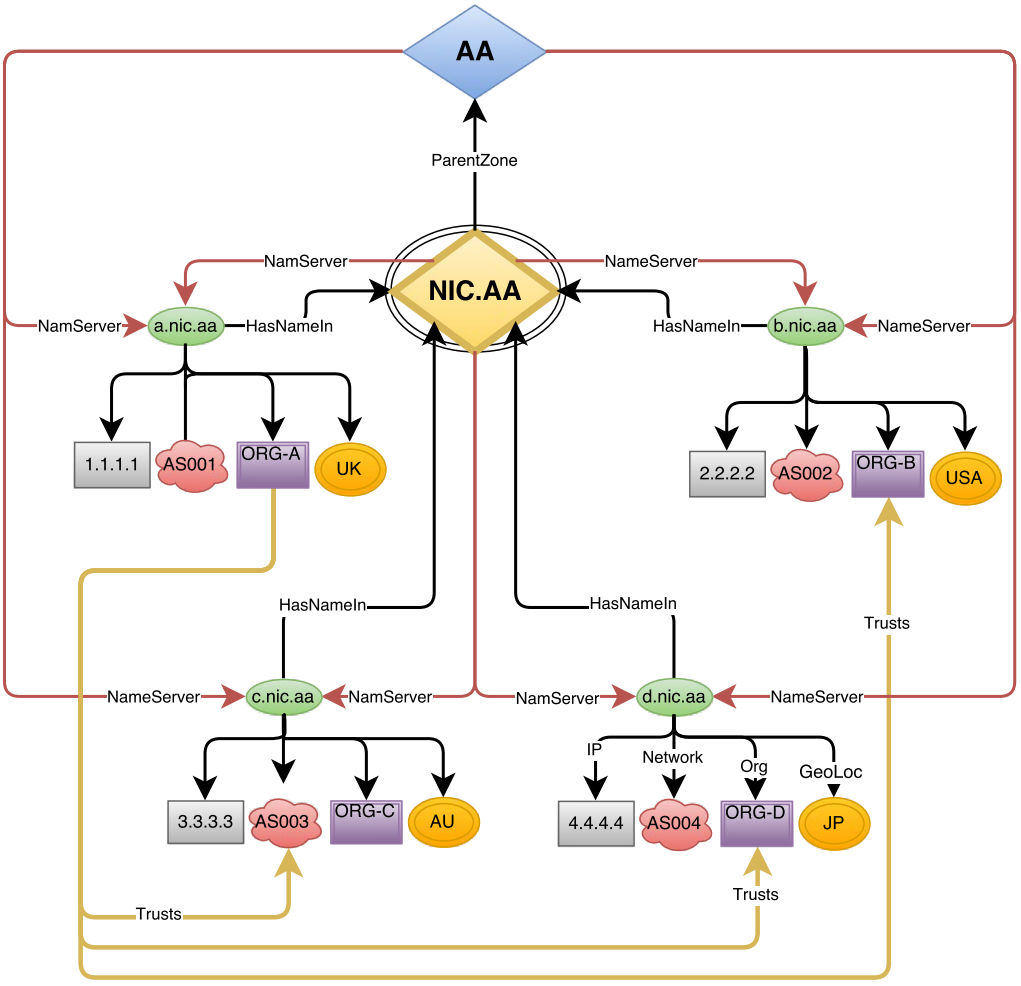
9. How do you rate the Quality Attributes of This DNS Model?



Quality Attribute	Very Low	Low	Medium	High	Very High
	1	2	3	4	5
Availability	◯	◯	◯	◯	◯
Security	◯	◯	◯	◯	◯
Stability	◯	◯	◯	◯	◯
Resiliency	◯	◯	◯	◯	◯

10. Can you justify or comment on your choices? _____

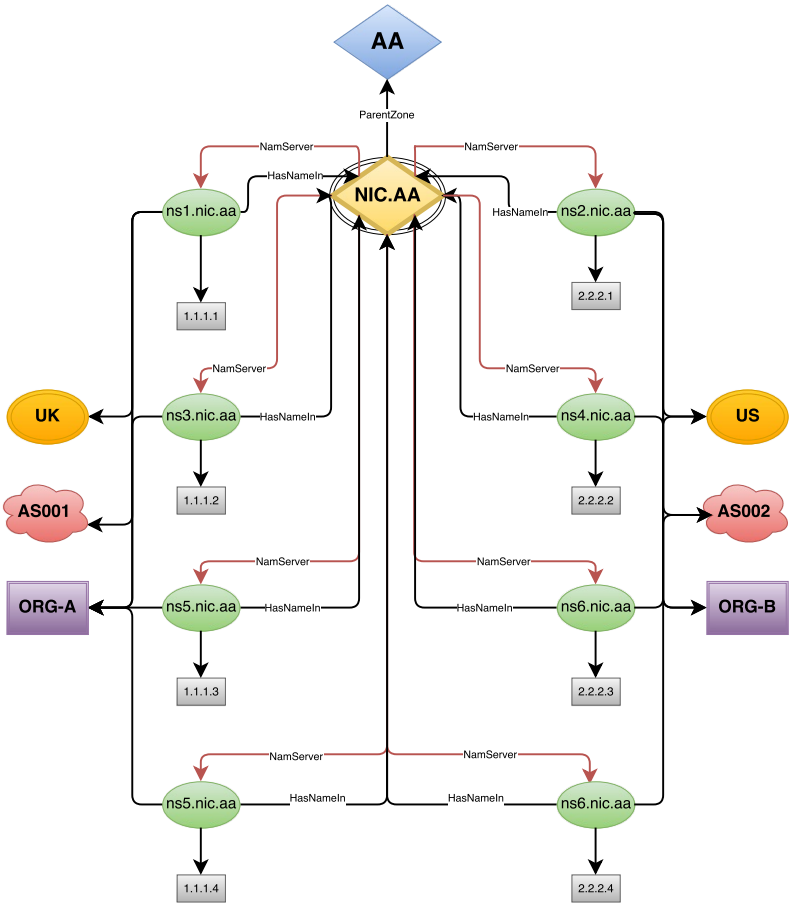
11. How do you rate the Quality Attributes of This DNS Model?



Quality Attribute	Very Low	Low	Medium	High	Very High
	1	2	3	4	5
Availability	◯	◯	◯	◯	◯
Security	◯	◯	◯	◯	◯
Stability	◯	◯	◯	◯	◯
Resiliency	◯	◯	◯	◯	◯

12. Can you justify or comment on your choices? ————

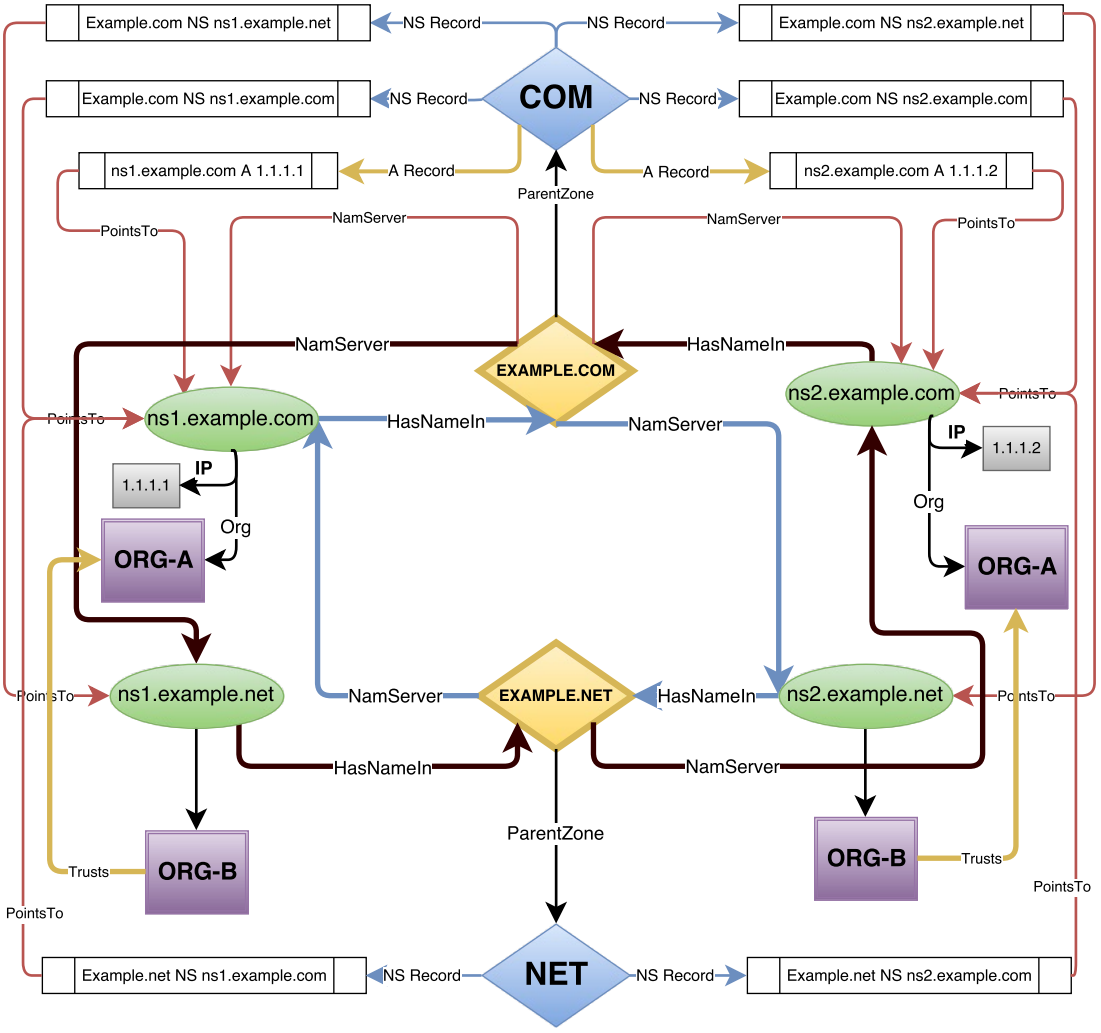
13. How do you rate the Quality Attributes of This DNS Model?



Quality Attribute	Very Low	Low	Medium	High	Very High
	1	2	3	4	5
Availability	◯	◯	◯	◯	◯
Security	◯	◯	◯	◯	◯
Stability	◯	◯	◯	◯	◯
Resiliency	◯	◯	◯	◯	◯

14. Can you justify or comment on your choices? _____

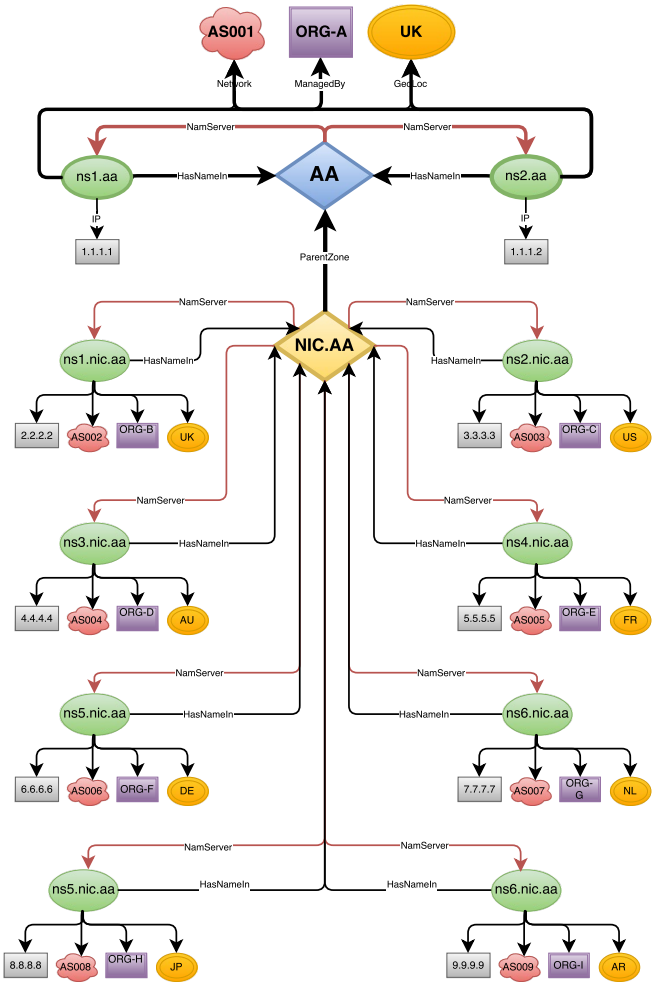
15. How do you rate the Quality Attributes of This DNS Model?



Quality Attribute	Very Low	Low	Medium	High	Very High
	1	2	3	4	5
Availability	◯	◯	◯	◯	◯
Security	◯	◯	◯	◯	◯
Stability	◯	◯	◯	◯	◯
Resiliency	◯	◯	◯	◯	◯

16. Can you justify or comment on your choices? ————

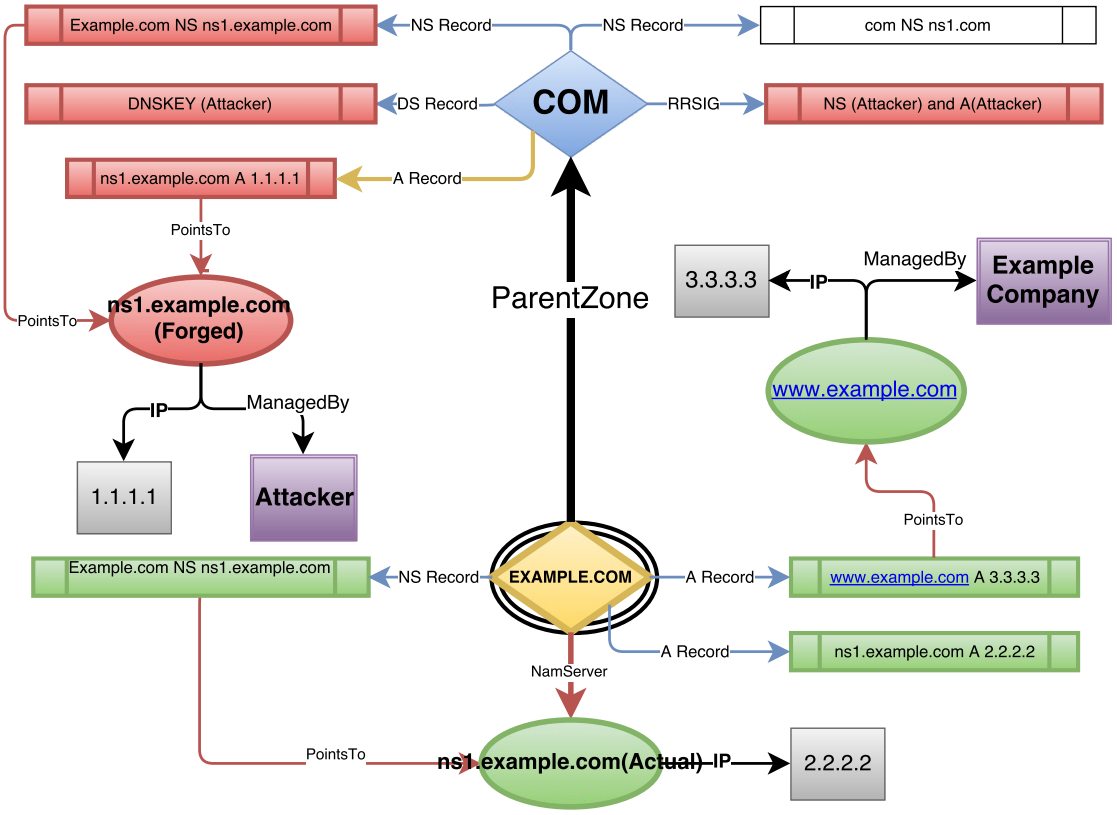
17. How do you rate the Quality Attributes of This DNS Model?



Quality Attribute	Very Low	Low	Medium	High	Very High
	1	2	3	4	5
Availability	◯	◯	◯	◯	◯
Security	◯	◯	◯	◯	◯
Stability	◯	◯	◯	◯	◯
Resiliency	◯	◯	◯	◯	◯

18. Can you justify or comment on your choices? _____

19. How do you rate the Quality Attributes of This DNS Model?



Quality Attribute	Very Low	Low	Medium	High	Very High
	1	2	3	4	5
Availability	◯	◯	◯	◯	◯
Security	◯	◯	◯	◯	◯
Stability	◯	◯	◯	◯	◯
Resiliency	◯	◯	◯	◯	◯

20. Can you justify or comment on your choices? ————

E.4 Assessing TLD Quality Attributes

On a scale of 1 to 5, How do you rate the quality attributes of your "own TLD" (as listed in the first section of this questionnaire) current configuration and DNS servers' deployment structure?

Quality Attribute	Very Low	Low	Medium	High	Very High
	1	2	3	4	5
Availability	◯	◯	◯	◯	◯
Security	◯	◯	◯	◯	◯
Stability	◯	◯	◯	◯	◯
Resiliency	◯	◯	◯	◯	◯

Submission Message

DNS Operational Model Survey

Thank you for your valuable input.

Wish you a pleasant day.

Bibliography

[1] T Arendt, F Mantz, and G Taentzer. Emf refactor: specification and application of model refactorings within the eclipse modeling framework. In *of the BENEVOL workshop*, 2010.

[2] Microsoft Corporation. Microsoft responds to dns issues. Technical report, Microsoft Corporation, 2001. URL http://www.microsoft.com/en-us/news/press/2001/jan01/01-24dnspr.aspx.

[3] Inc. Dyn. Ddos attack against dyn managed dns, incident report, 2016. URL https://www.dynstatus.com/incidents/nlr4yrr162t8.

[4] Working Group 4. Final report: Dns best practices. Technical report, The Communications Security, Reliability and Interoperability Council III, 2012. URL http://transition.fcc.gov/bureaus/pshs/advisory/csric3/CSRICIII_9-12-12_WG4-FINAL-Report-DNS-Best-Practices.pdf.

[5] D. Barr. Rfc 1912: Common dns operational and configuration errors. *International Engineering Task Force, Status: Standard*, 1996. URL http://www.ietf.org/rfc/rfc1912.txt.

[6] Elz Robert, Bush Randy, Bradner Scott, and Patton Michael. Rfc 2182: Selection and operation of secondary dns servers. *International Engineering Task Force, Status: Standard*, 1997. URL http://www.ietf.org/rfc/rfc2182.txt.

[7] C. Deccio, J. Sedayao, K. Kant, and P. Mohapatra. Measuring availability in the domain name system. In *INFOCOM, 2010 Proceedings IEEE*, pages 1–5, March 2010. doi: 10.1109/INFCOM.2010.5462270.

[8] Eric Osterweil, Danny McPherson, and Lixia Zhang. Operational implications of the dns control plane. *IEEE Reliability Society Newsletter*, 2011. URL `http://rs.ieee.org/images/files/newsletters/2011/2_2011/OperationalImplicationsoftheDNSControlPlane.pdf`.

[9] A. Herzberg and H. Shulman. Dnssec: Security and availability challenges. In *Communications and Network Security (CNS), 2013 IEEE Conference on*, pages 365–366, Oct 2013. doi: 10.1109/CNS.2013.6682730.

[10] D Conrad. Towards improving dns security, stability, and resiliency, 2012.

[11] Haya Shulman and Shiran Ezra. Poster: On the resilience of dns infrastructure. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 1499–1501. ACM, 2014.

[12] Keyu Lu, Kaikun Dong, Cuihua Wang, and Haiyan Xu. Dns configuration detection model. In *Systems and Informatics (ICSAI), 2014 2nd International Conference on*, pages 613–618. IEEE, 2014.

[13] Duane Wessels, Marina Fomenkov, Nevil Brownlee, and kc claffy. Measurements and laboratory simulations of the upper dns hierarchy. In Chadi Barakat and Ian Pratt, editors, *Passive and Active Network Measurement*, volume 3015 of *Lecture Notes in Computer Science*, pages 147–157. Springer Berlin Heidelberg, 2004. ISBN 978-3-540-21492-2. doi: 10.1007/978-3-540-24668-8_15.

[14] Cricket Liu and Paul Albitz. *DNS and Bind.* " O'Reilly Media, Inc.", 2006.

[15] Vasileios Pappas, D. Wessels, D. Massey, Songwu Lu, A. Terzis, and Lixia Zhang. Impact of configuration errors on dns robustness. *Selected Areas in*

Communications, IEEE Journal on, 27(3):275–290, April 2009. ISSN 0733-8716. doi: 10.1109/JSAC.2009.090404.

[16] Venugopalan Ramasubramanian and Emin Gün Sirer. Perils of transitive trust in the domain name system. In *Proceedings of the 5th Conference on Internet Measurement 2005, Berkeley, California, USA, October 19-21, 2005*, pages 379–384. USENIX Association, 2005. URL `http://www.usenix.org/events/imc05/tech/ramasubramanian.html`.

[17] Andrew J. Kalafut, Craig A. Shue, and Minaxi Gupta. Understanding implications of dns zone provisioning. In *Proceedings of the 8th ACM SIGCOMM Conference on Internet Measurement*, IMC '08, pages 211–216, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-334-1. doi: 10.1145/1452520.1452546.

[18] Emiliano Casalicchio, Marco Caselli, Alessio Coletta, Salvatore Di Blasi, and IgorNai Fovino. Measuring name system health. In Jonathan Butts and Sujeet Shenoi, editors, *Critical Infrastructure Protection VI*, volume 390 of *IFIP Advances in Information and Communication Technology*, pages 155–169. Springer Berlin Heidelberg, 2012. ISBN 978-3-642-35763-3. doi: 10.1007/978-3-642-35764-0_12.

[19] Keyu Lu, Kaikun Dong, Cuihua Wang, and Haiyan Xu. Dns configuration detection model. In *Systems and Informatics (ICSAI), 2014 2nd International Conference on*, pages 613–618. IEEE, 2014. doi: 10.1109/ICSAI.2014.7009359.

[20] Jaeyeon Jung, Emil Sit, Hari Balakrishnan, and Robert Morris. Dns performance and the effectiveness of caching. *IEEE/ACM Transactions on networking*, 10(5):589–603, 2002.

[21] M. Lotter. Rfc 1033: Domain administrators operations guide. *Work in Progress*, 1987. URL `http://www.ietf.org/rfc/rfc1033.txt`.

[22] Paul Mockapetris. Rfc 1034: Domain names: concepts and facilities. *Work in Progress*, 1987. URL `http://www.ietf.org/rfc/rfc1034.txt`.

[23] Paul Mockapetris. Rfc 1035: Domain namesâĂŤimplementation and specification. *Work in Progress*, 1987. URL `http://www.ietf.org/rfc/rfc1035.txt`.

[24] Kevin R Fall and W Richard Stevens. *TCP/IP illustrated, volume 1: The protocols*. addison-Wesley, 2011.

[25] Network Working Group et al. Request for comments (rfc) 4033,âĂIJ. *Protocol Modifications for the DNS Security Extensions,âĂİ Mar*, 2005.

[26] R Arends, R Austein, M Larson, D Massey, and S Rose. Protocol modifications for the dns security extensions (2005). *RFC4035*, 2005.

[27] James F Kurose and Keith W Ross. *Computer networking: a top-down approach*, volume 4. Addison Wesley Boston, USA, 2009.

[28] ICANN Security and Stability Advisory Committee. Response to recent security threats,. Technical report, ICANN, 2008. URL `https://www.icann.org/news/announcement-2008-07-03-en`. Los Angeles, California.

[29] ICANN. Measuring the health of the domain name system, report of the second annual symposium on dns security, stability and resiliency (1-3 february 2010, kyoto university kyoto, japan). Technical report, ICANN, 2010. URL `https://www.icann.org/en/system/files/files/dns-ssr-symposium-report-1-03feb10-en.pdf`. Los Angeles, California.

[30] ICANN. Measuring the health of the domain name system, report of the 3rd global symposium (rome, italy, october 19-20, 2011). Technical report, ICANN, 2011. URL `https://www.gcsec.org/keyportal/uploads/dns_ssr3_report_20120210_001.pdf`. Los Angeles, California.

[31] Hartmut Ehrig, Reiko Heckel, Grzegorz Rozenberg, and Gabriele Taentzer. *Graph Transformations*. Springer, 2008.

[32] Reiko Heckel. Graph transformation in a nutshell. *Electronic notes in theoretical computer science*, 148(1):187–198, 2006.

[33] H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of Algebraic Graph Transformation (Monographs in Theoretical Computer Science. An EATCS Series)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006. ISBN 3540311874.

[34] Andrea Corradini, Ugo Montanari, Francesca Rossi, Hartmut Ehrig, Reiko Heckel, and Michael Löwe. Algebraic approaches to graph transformation-part i: Basic concepts and double pushout approach. In *Handbook of Graph Grammars*, pages 163–246, 1997.

[35] Alfio Martini, H Ehrig, and D Nunes. *Elements of basic category theory*. Techn. Univ., Fachbereich 13, Informatik, 1996.

[36] Reiko Heckel, Jochen Malte Küster, and Gabriele Taentzer. Confluence of typed attributed graph transformation systems. In *International Conference on Graph Transformation*, pages 161–176. Springer, 2002.

[37] Michael R. Berthold, Ingrid Fischer, and Manuel Koch. Attributed graph transformation with partial attribution, 2000.

[38] Enrico Biermann, Karsten Ehrig, Christian Köhler, Günter Kuhns, Gabriele Taentzer, and Eduard Weiss. EMF model refactoring based on graph transformation concepts. *ECEASST*, 3, 2006. URL `http://eceasst.cs.tu-berlin.de/index.php/eceasst/article/view/34`.

[39] Brent Hailpern and Peri Tarr. Model-driven development: The good, the bad, and the ugly. *IBM systems journal*, 45(3):451, 2006.

[40] Claudia Ermel, Enrico Biermann, Johann Schmidt, and Angeline Warning. Visual modeling of controlled emf model transformation using henshin. *Electronic Communications of the EASST*, 32, 2011.

[41] Thorsten Arendt, Enrico Biermann, Stefan Jurack, Christian Krause, and Gabriele Taentzer. Henshin: advanced concepts and tools for in-place emf model transformations. In *Model Driven Engineering Languages and Systems*, pages 121–135. Springer, 2010. doi: 10.1007/978-3-642-16145-2_9.

[42] Enrico Biermann, Claudia Ermel, and Gabriele Taentzer. Formal foundation of consistent emf model transformations by algebraic graph transformation. *Software & Systems Modeling*, 11(2):227–250, 2012.

[43] Krzysztof Czarnecki and Simon Helsen. Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3):621–645, 2006.

[44] Tom Mens and Pieter Van Gorp. A taxonomy of model transformation. *Electronic Notes in Theoretical Computer Science*, 152:125–142, 2006.

[45] Shane Sendall and Wojtek Kozaczynski. Model transformation the heart and soul of model-driven software development. Technical report, Swiss Federal Institute of Technology in Lausanne (EPFL), 2003.

[46] T. Mens and T. Tourwe. A survey of software refactoring. *Software Engineering, IEEE Transactions on*, 30(2):126–139, Feb 2004. ISSN 0098-5589. doi: 10.1109/TSE.2004.1265817.

[47] Arie Van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: An annotated bibliography. *Sigplan Notices*, 35(6):26–36, 2000.

[48] OMG OMG. Meta object facility (mof) core specification, 2014.

[49] OMG OMG. Mof 2.0 / xmi mapping specification, 2008.

[50] Douglas C Schmidt. Model-driven engineering. *COMPUTER-IEEE COMPUTER SOCIETY-*, 39(2):25, 2006.

[51] J Mukerji and J Miller. Mda guide version 1.0. 1, 2003, 2015.

[52] Hans-Erik Eriksson, Magnus Penker, Brian Lyons, and David Fado. *UML 2 Toolkit*. Wiley Publishing, 2003. ISBN 0471463612, 9780471463610.

[53] Parastoo Mohagheghi, Vegard Dehlen, and Tor Neple. Definitions and approaches to model quality in model-based software development–a review of literature. *Information and Software Technology*, 51(12):1646–1669, 2009.

[54] Jos Warmer, Anneke Kleppe, and Wim Bast. Mda explained: The model driven architecture: practice and promise. *Boston et al*, 2003.

[55] Ludwik Finkelstein. Widely, strongly and weakly defined measurement. *Measurement*, 34(1):39–48, 2003.

[56] Sebastian Castro, Duane Wessels, Marina Fomenkov, and Kimberly Claffy. A day at the root of the internet. *ACM SIGCOMM Computer Communication Review*, 38(5):41–46, 2008.

[57] Richard Liston, Sridhar Srinivasan, and Ellen Zegura. Diversity in dns performance measures. In *Proceedings of the 2nd ACM SIGCOMM Workshop on Internet measurment*, pages 19–31. ACM, 2002.

[58] Yuji Sekiya, Kenjiro Cho, Akira Kato, and Jun Murai. Research of method for dns performance measurement and evaluation based on benchmark dns servers. *Electronics and Communications in Japan (Part I: Communications)*, 89(10):66–75, 2006.

[59] E Casalicchio, M Caselli, D Conrad, J Damas, and I Nai Fovino. Reference architecture, models and metrics. *GCSEC technical document, Version*, 1, 2011.

[60] Wei Li and Sallie Henry. Object-oriented metrics that predict maintainability. *Journal of systems and software*, 23(2):111–122, 1993.

[61] Shyam R Chidamber and Chris F Kemerer. A metrics suite for object oriented design. *IEEE Transactions on software engineering*, 20(6):476–493, 1994.

[62] Mark Lorenz and Jeff Kidd. *Object-oriented software metrics: a practical guide*. Prentice-Hall, Inc., 1994.

[63] Hongmin Lu, Yuming Zhou, Baowen Xu, Hareton Leung, and Lin Chen. The ability of object-oriented metrics to predict change-proneness: a meta-analysis. *Empirical software engineering*, 17(3):200–242, 2012.

[64] Jeffrey Gennari and David Garlan. Measuring attack surface in software architecture. Technical report, Technical Report CMU-ISR-11-121, Carnegie Mellon University, 2012.

[65] Danijel Radjenović, Marjan Heričko, Richard Torkar, and Aleš Živkovič. Software fault prediction metrics: A systematic literature review. *Information and Software Technology*, 55(8):1397–1418, 2013.

[66] Boudewijn Van Dongen, Remco Dijkman, and Jan Mendling. Measuring similarity between business process models. In *Seminal Contributions to Information Systems Engineering*, pages 405–419. Springer, 2013.

[67] Henrik Leopold, Sergey Smirnov, and Jan Mendling. On the refactoring of activity labels in business process models. *Information Systems*, 37(5):443–459, 2012.

[68] Gregory S Hornby. Measuring complexity by measuring structure and organization. In *2007 IEEE Congress on Evolutionary Computation*, pages 2017–2024. IEEE, 2007.

[69] Samar Mouchawrab, Lionel C Briand, and Yvan Labiche. A measurement framework for object-oriented software testability. *Information and software technology*, 47(15):979–997, 2005.

[70] Erik Arisholm and Dag IK Sjoberg. Evaluating the effect of a delegated versus centralized control style on the maintainability of object-oriented software. *IEEE Transactions on software engineering*, 30(8):521–534, 2004.

[71] Radu Marinescu and Michelle Lanza. Object-oriented metrics in practice, 2006.

[72] Geert Poels and Guido Dedene. Distance-based software measurement: necessary and sufficient properties for software measures. *Information and Software Technology*, 42(1):35–46, 2000.

[73] Lionel C Briand, Sandro Morasca, and Victor R Basili. Property-based software engineering measurement. *IEEE Transactions on Software Engineering*, 22(1):68–86, 1996.

[74] Daniel T Larose. *Discovering knowledge in data: an introduction to data mining*. John Wiley & Sons, 2014.

[75] Jiawei Han, Jian Pei, and Micheline Kamber. *Data mining: concepts and techniques*. Elsevier, 2011.

[76] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H Witten. The weka data mining software: an update. *ACM SIGKDD explorations newsletter*, 11(1):10–18, 2009.

[77] John J Bartko. On various intraclass correlation reliability coefficients. *Psychological bulletin*, 83(5):762, 1976.

[78] James Dean Brown. Likert items and scales of measurement. *Shiken: JALT Testing & Evaluation SIG Newsletter*, 15(1):10–14, 2011.

[79] Stefano Baccianella, Andrea Esuli, and Fabrizio Sebastiani. Evaluation measures for ordinal regression. In *Intelligent Systems Design and Applications, 2009. ISDA'09. Ninth International Conference on*, pages 283–287. IEEE, 2009.

[80] Lisa Gaudette and Nathalie Japkowicz. Evaluation methods for ordinal classification. In *Canadian Conference on Artificial Intelligence*, pages 207–210. Springer, 2009.

[81] JWT Lee and Da-Zhong Liu. Induction of ordinal decision trees. In *Machine Learning and Cybernetics, 2002. Proceedings. 2002 International Conference on*, volume 4, pages 2220–2224. IEEE, 2002.

[82] Sophie Vanbelle and Adelin Albert. A note on the linearly weighted kappa coefficient for ordinal scales. *Statistical Methodology*, 6(2):157–163, 2009.

[83] Willem Waegeman, Bernard De Baets, and Luc Boullart. A comparison of different roc measures for ordinal regression. In *Proceedings of the 3rd International Workshop on ROC Analysis in Machine Learning., N. Lachiche, C. Ferri, and S. Macskassy, Eds*, pages 63–69. Citeseer, 2006.

[84] Jaime S Cardoso and Ricardo Sousa. Measuring the performance of ordinal classification. *International Journal of Pattern Recognition and Artificial Intelligence*, 25(08):1173–1195, 2011.

[85] Min Zhang, Tracy Hall, and Nathan Baddoo. Code bad smells: a review of current knowledge. *Journal of Software Maintenance and Evolution: research and practice*, 23(3):179–202, 2011.

[86] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Boston, MA, USA, 1999. ISBN 0-201-48567-2.

[87] Serge Demeyer, Stéphane Ducasse, and Oscar Nierstrasz. Finding refactorings via change metrics. In *ACM SIGPLAN Notices*, volume 35, pages 166–177. ACM, 2000.

[88] Katsuhisa Maruyama and Ken-ichi Shima. Automatic method refactoring using weighted dependence graphs. In *Proceedings of the 21st international conference on Software engineering*, pages 236–245. ACM, 1999.

[89] Joshua Garcia, Daniel Popescu, George Edwards, and Nenad Medvidovic. Identifying architectural bad smells. In *Software Maintenance and Reengineering, 2009. CSMR'09. 13th European Conference on*, pages 255–258. IEEE, 2009.

[90] William F. Opdyke. *Refactoring Object-oriented Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, Champaign, IL, USA, 1992. UMI Order No. GAX93-05645.

[91] Dénes Bisztray, Reiko Heckel, and Hartmut Ehrig. Verification of architectural refactorings by rule extraction. In *Fundamental Approaches to Software Engineering*, Lecture Notes in Computer Science, pages 347–361. Springer Berlin Heidelberg, 2008. ISBN 978-3-540-78743-3. doi: 10.1007/978-3-540-78743-3_26.

[92] Java Emitter Templates. Part of the eclipse modeling framework, see jet tutorial by remko pompa at http://eclipse. org/articles. *Article-JET2/jet_ tutorial2. html*, 69, 2016.

[93] Casey Deccio. Maintenance, mishaps and mending in deployments of the domain name system security extensions (dnssec). *International Journal of Critical Infrastructure Protection*, 5(2):98–103, 2012.

[94] Matthias Biehl. Literature study on model transformations. *Royal Institute of Technology, Tech. Rep. ISRN/KTH/MMK*, 2010.

[95] Tom Mens, Serge Demeyer, and Dirk Janssens. Formalising behaviour preserving program transformations. In *International Conference on Graph Transformation*, pages 286–301. Springer, 2002.

[96] Kyle Schomp, Michael Rabinovich, and Mark Allman. Towards a model of dns client behavior. In *International Conference on Passive and Active Network Measurement*, pages 263–275. Springer, 2016.

[97] Tom Mens, Gabriele Taentzer, and Olga Runge. Detecting structural refactoring conflicts using critical pair analysis. *Electronic Notes in Theoretical Computer Science*, 127(3):113–128, 2005.

[98] Manuel Wimmer, Salvador Martínez Perez, Frédéric Jouault, and Jordi Cabot. A catalogue of refactorings for model-to-model transformations. *Journal of Object Technology*, 11(2):2–1, 2012.

[99] C Grothoff, M Wachs, H Wolf, and J Appelbaum. Special-use domain names of peer-to-peer name systems. Technical report, IETF Internet Draft, 2013.

[100] ICANN Security and Stability Advisory Committee. Invalid top level domain queries at the root level of the domain name system, 2010. URL `http://bit.ly/1mDxRJO`.

[101] P Faltstrom, R Austein, P Koch, et al. Design choices when expanding the dns. Technical report, International Engineering Task Force, 2009.

[102] Paul Mockapetris and Kevin J Dunlap. *Development of the domain name system*, volume 18. ACM, 1988.

[103] Hongyu Gao, Vinod Yegneswaran, Yan Chen, Phillip Porras, Shalini Ghosh, Jian Jiang, and Haixin Duan. An empirical reexamination of global dns behavior. *ACM SIGCOMM Computer Communication Review*, 43(4):267–278, 2013.

[104] Thomas Callahan, Mark Allman, and Michael Rabinovich. On modern dns behavior and properties. *ACM SIGCOMM Computer Communication Review*, 43(3):7–15, 2013.

[105] Vern Paxson, Mihai Christodorescu, Mobin Javed, Josyula Rao, Reiner Sailer, Douglas Lee Schales, Mark Stoecklin, Kurt Thomas, Wietse Venema, and Nicholas Weaver. Practical comprehensive bounds on surreptitious communication over dns. In *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*, pages 17–32, 2013.

[106] Manos Antonakakis, Roberto Perdisci, Wenke Lee, Nikolaos Vasiloglou II, and David Dagon. Detecting malware domains at the upper dns hierarchy. In *USENIX security symposium*, volume 11, pages 1–16, 2011.

[107] Manos Antonakakis, Roberto Perdisci, David Dagon, Wenke Lee, and Nick Feamster. Building a dynamic reputation system for dns. In *USENIX security symposium*, pages 273–290, 2010.

[108] Liang Zhu, Zi Hu, John Heidemann, Duane Wessels, Allison Mankin, and Nikita Somaiya. Connection-oriented dns to improve privacy and security. In *2015 IEEE Symposium on Security and Privacy*, pages 171–186. IEEE, 2015.

[109] Kyle Schomp, Tom Callahan, Michael Rabinovich, and Mark Allman. On measuring the client-side dns infrastructure. In *Proceedings of the 2013 conference on Internet measurement conference*, pages 77–90. ACM, 2013.

[110] Jeffrey Pang, James Hendricks, Aditya Akella, Roberto De Prisco, Bruce Maggs, and Srinivasan Seshan. Availability, usage, and deployment characteristics of the domain name system. In *Proceedings of the 4th ACM SIGCOMM conference on Internet measurement*, pages 1–14. ACM, 2004.

[111] Vasileios Pappas, Patrik Fältström, Daniel Massey, and Lixia Zhang. Distributed dns troubleshooting. In *Proceedings of the ACM SIGCOMM workshop on Network troubleshooting: research, theory and operations practice meet malfunctioning reality*, pages 265–270. ACM, 2004. doi: 10.1145/1016687.1016694.

[112] Casey Deccio. Visual dnssec troubleshooting with dnsviz. Technical report, Sandia National Laboratories, 2010. URL `http://dnsviz.net/`.

[113] Softrix Technologies. Dns checker tool, 2012. URL `http://www.dnschecker.org/`.

[114] Elvsoft. Intodns, dns troubleshooting tool, 2012. URL `http://www.intodns.com/`.

[115] SE and AFNIC. Zonemaster tool, 2012. URL `http://www.zonemaster.net/`.

[116] R. Chandramouli and S. Rose. An integrity verification scheme for dns zone file based on security impact analysis. In *Computer Security Applications Conference, 21st Annual*, pages 10 pp.–321, Dec 2005. doi: 10.1109/CSAC.2005.9.

[117] Peter B Danzig, Katia Obraczka, and Anant Kumar. An analysis of wide-area name server traffic: a study of the internet domain name system. *ACM SIGCOMM Computer Communication Review*, 22(4):281–292, 1992.

[118] Steven Cheung and Karl N Levitt. A formal-specification based approach for protecting the domain name system. In *Dependable Systems and Networks,*

2000. DSN 2000. Proceedings International Conference on, pages 641–651. IEEE, 2000.

[119] Team Cymru. Ip to asn mapping. *http://www. team-cymru. org/Services/ip-to-asn.html*, 2008.

[120] Credentia. lame delegation statistics on the tld zones, 2004. URL `http://www.credentia.cc/research/cctlds/`.

[121] Dag IK Sjøberg, Jo Erskine Hannay, Ove Hansen, Vigdis By Kampenes, Amela Karahasanovic, N-K Liborg, and Anette C Rekdal. A survey of controlled experiments in software engineering. *IEEE transactions on software engineering*, 31(9):733–753, 2005.

[122] James W Moore and Alain Abran. *Guide to the software engineering body of knowledge*. IEEE Computer Society, 2004.

[123] Robert C Martin. *Clean code: a handbook of agile software craftsmanship*. Pearson Education, 2009.

[124] Girish Suryanarayana, Ganesh Samarthyam, and Tushar Sharma. *Refactoring for software design smells: Managing technical debt*. Morgan Kaufmann, 2014.

[125] Mika Mantyla, Jari Vanhanen, and Casper Lassenius. A taxonomy and an initial empirical study of bad smells in code. In *Software Maintenance, 2003. ICSM 2003. Proceedings. International Conference on*, pages 381–384. IEEE, 2003.

[126] H. Dieter Rombach. A controlled expeniment on the impact of software structure on maintainability. *IEEE Transactions on Software Engineering*, SE-13 (3):344–354, 1987.

[127] Dennis Kafura and Geereddy R. Reddy. The use of software complexity metrics in software maintenance. *IEEE Transactions on Software Engineering*, 13(3): 335, 1987.

[128] R. Marinescu. Detection strategies: metrics-based rules for detecting design flaws. In *Software Maintenance, 2004. Proceedings. 20th IEEE International Conference on*, pages 350–359, Sept 2004. doi: 10.1109/ICSM.2004.1357820.

[129] E.-H. Alikacem and H.A. Sahraoui. A metric extraction framework based on a high-level description language. In *Source Code Analysis and Manipulation, 2009. SCAM '09. Ninth IEEE International Working Conference on*, pages 159–167, Sept 2009. doi: 10.1109/SCAM.2009.27.

[130] William H Brown, Raphael C Malveau, Hays W McCormick, and Thomas J Mowbray. *AntiPatterns: refactoring software, architectures, and projects in crisis*. John Wiley & Sons, Inc., 1998.

[131] Vivek Shah, Marcos Sivitanides, and Roy Martin. Pitfalls of object-oriented development, 2004.

[132] Arthur J Riel. *Object-oriented design heuristics*. Addison-Wesley Longman Publishing Co., Inc., 1996.

[133] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrea De Lucia, and Denys Poshyvanyk. Detecting bad smells in source code using change history information. In *Automated software engineering (ASE), 2013 IEEE/ACM 28th international conference on*, pages 268–278. IEEE, 2013.

[134] Cristina Marinescu, Radu Marinescu, Petru Florin Mihancea, and R Wettel. iplasma: An integrated platform for quality assessment of object-oriented design. In *In ICSM (Industrial and Tool Volume*. Citeseer, 2005.

[135] Nikolaos Tsantalis and Alexander Chatzigeorgiou. Identification of extract method refactoring opportunities for the decomposition of methods. *Journal of Systems and Software*, 84(10):1757–1782, 2011.

[136] William C Wake. *Refactoring workbook*. Addison-Wesley Professional, 2004.

[137] Guilherme Rangel, Barbara König, and Hartmut Ehrig. Bisimulation verification for the dpo approach with borrowed contexts. *Electronic Communications of the EASST*, 6, 2007.

[138] Enrico Biermann, Karsten Ehrig, Christian Köhler, Günter Kuhns, Gabriele Taentzer, and Eduard Weiss. Emf model refactoring based on graph transformation concepts. *Electronic Communications of the EASST*, 3, 2007.

[139] Berthold Hoffmann, Dirk Janssens, and Niels Van Eetvelde. Cloning and expanding graph transformation rules for refactoring. *Electronic Notes in Theoretical Computer Science*, 152:53–67, 2006.

[140] Guilherme Rangel, Leen Lambers, Barbara König, Hartmut Ehrig, and Paolo Baldan. Behavior preservation in model refactoring using dpo transformations with borrowed contexts. In *International Conference on Graph Transformation*, pages 242–256. Springer, 2008.

[141] Paolo Bottoni, Manuel Koch, Francesco Parisi-Presicce, and Gabriele Taentzer. Termination of high-level replacement units with application to model transformation. *Electronic Notes in Theoretical Computer Science*, 127(4):71–86, 2005.

[142] Paolo Bottoni and Gabriele Taentzery. Efficient parsing of visual languages based on critical pair analysis and contextual layered graph transformation. *a+ a*, 2:2, 2000.

[143] Praveen Jayaraman, Jon Whittle, Ahmed M Elkhodary, and Hassan Gomaa. Model composition in product lines and feature interaction detection using critical pair analysis. In *International Conference on Model Driven Engineering Languages and Systems*, pages 151–165. Springer, 2007.

[144] Frank Hermann, Hartmut Ehrig, Ulrike Golas, and Fernando Orejas. Efficient analysis and execution of correct and complete model transformations based on triple graph grammars. In *Proceedings of the First International Workshop on Model-Driven Interoperability*, pages 22–31. ACM, 2010.

[145] Tom Mens, Ragnhild Van Der Straeten, and Maja DâĂŹHondt. Detecting and resolving model inconsistencies using transformation dependency analysis. In *International Conference on Model Driven Engineering Languages and Systems*, pages 200–214. Springer, 2006.

[146] Marouane Kessentini, Houari Sahraoui, and Mounir Boukadoum. Model transformation as an optimization problem. In *International Conference on Model Driven Engineering Languages and Systems*, pages 159–173. Springer, 2008.

[147] Marouane Kessentini, Houari Sahraoui, Mounir Boukadoum, and Omar Ben Omar. Search-based model transformation by example. *Software & Systems Modeling*, 11(2):209–226, 2012.

[148] Dionysios Efstathiou, James R Williams, and Steffen Zschaler. Crepe complete: Multi-objective optimization for your models. In *CMSEBA@ MoDELS*, pages 25–34, 2014.

[149] Martin Fleck, Javier Troya, and Manuel Wimmer. Marrying search-based optimization and model transformation technology. *Proc. of NasBASE*, 2015.

[150] Tom Mens, Niels Van Eetvelde, Serge Demeyer, and Dirk Janssens. Formalizing refactorings with graph transformations. *Journal of Software Maintenance and Evolution: Research and Practice*, 17(4):247–276, 2005.

[151] Paolo Bottoni, Francesco Parisi Presicce, and Gabriele Taentzer. Specifying integrated refactoring with distributed graph transformations. In *International Workshop on Applications of Graph Transformations with Industrial Relevance*, pages 220–235. Springer, 2003.

[152] Gerson Sunyé, Damien Pollet, Yves Le Traon, and Jean-Marc Jézéquel. Refactoring uml models. In *International Conference on the Unified Modeling Language*, pages 134–148. Springer, 2001.

[153] Tom Mens. Introduction and roadmap: History and challenges of software evolution. In *Software evolution*, pages 1–11. Springer, 2008.

[154] Maddeh Mohamed, Mohamed Romdhani, and Khaled Ghedira. Classification of model refactoring approaches. *J. Object Technol.(JOT)*, 8(6):143–158, 2009.

[155] Tom Mens, Gabriele Taentzer, and Dirk Müller. Model-driven software refactoring. *Model-Driven Software Development: Integrating Quality Assurance*, pages 170–203, 2008.

[156] Krzysztof Czarnecki and Simon Helsen. Classification of model transformation approaches. In *Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture*, volume 45, pages 1–17. USA, 2003.