Effective entity resolution methodology for improving data quality and reliability of service-oriented applications

Ewa Musial
University at Albany, State University of New York, emusial@albany.edu

The University at Albany community has made this article openly available. Please share how this access benefits you.

Follow this and additional works at: https://scholarsarchive.library.albany.edu/legacy-etd

Part of the Computer Engineering Commons, and the Computer Sciences Commons

Recommended Citation

This Dissertation is brought to you for free and open access by the The Graduate School at Scholars Archive. It has been accepted for inclusion in Legacy Theses & Dissertations (2009 - 2024) by an authorized administrator of Scholars Archive. Please see Terms of Use. For more information, please contact scholarsarchive@albany.edu.
EFFECTIVE ENTITY RESOLUTION METHODOLOGY FOR IMPROVING DATA QUALITY AND RELIABILITY OF SERVICE-ORIENTED APPLICATIONS

by

Ewa Musial

A Dissertation
Submitted to the University at Albany, State University of New York in Partial Fulfillment of the Requirements for the Degree of Doctor of Philosophy

College of Computing and Information
Department of Computer Science
2014
Abstract

This dissertation proposes new paradigms for improving the testing, reliability of service-oriented applications as well as the quality of data. Since it is difficult to track information flowing through the multiple tiers of an application, testing service-oriented systems can be very challenging. We present a methodology for testing service-oriented applications that takes into account all the components, including services, external services, and data components. The results of our experiments demonstrate that this approach greatly improves the effectiveness of testing service-oriented applications.

To examine the effects of invalid data on the reliability of services and service-oriented applications, first we developed an approach to quantify the quality of database relations and compute the quality of data components based on the type of interactions they have with software components. Then, we developed a methodology that incorporates data quality into reliability modeling and can therefore better account for the failures caused by invalid data. Our empirical results show that our model provides more accurate estimations of reliability of service-oriented applications than traditional approaches by detecting 16% more faults.

Recognizing the importance of data quality, we developed an Entity Resolution (ER) algorithm that not only provides a blocking scheme, but also a 2-stage comparison selection process. It efficiently removes oversized blocks and identifies comparisons that are most likely to contain duplicates. Our empirical results demonstrate the usefulness of our algorithm with respect to both the efficiency and effectiveness. For the former, our algorithm reduces the number of comparisons that need to be resolved. For the latter, it increases the number of detected duplicates.
Acknowledgment

I dedicate this dissertation to my beloved parents, Anna and Zbigniew Musial, who have always encouraged me to pursue my dreams. Thank you, Mom and Dad!

I would like to express my deepest gratitude to my advisor, Professor Mei-Hwa Chen, for her guidance, motivation, and for teaching me that sometimes the journey is more important than the final destination. Without her guidance and persistent help, I would have never completed this thesis.

Professor Sekharipuram S. Ravi has also been a great help in the process of writing this dissertation. His kindness, patience, advice and encouragement have helped me make it through the most challenging times. I would also like to offer my special thanks to him for his careful and detailed feedback, and for sparking my interest in probabilistic and randomized algorithms.

This thesis would not have been possible without the support and professional expertise generously offered by Professor Jeong-Hyon Hwang. His meticulous comments were an enormous help to me. I am also grateful for Dr. Stephan M. Gerali, who, through numerous and patient email exchanges, helped me understand the many different types of implementations of web services.

I received an abundance of help from everyone at the New York State Police department. I would like to express the deepest appreciation to Gloria Brewer and Joseph Zanotelli, both of whom supported my professional and academic development along the
way. They inspired me to grow, and provided me with an opportunity to gain experience in distributed technologies, and service-oriented applications.

This dissertation also greatly benefited from discussion with and comments from my good friend Justin Williams. I would like to thank him for countless hours spent on discussing service-oriented applications, web-services, and other programming concepts. We would probably starve in the process if it was not for Justin’s wonderful wife Mim, making her trademark dumplings. I will be always in her debt.

I owe my deepest gratitude to Lenny Casper, who helped me in many ways to go through this quiet, complicated time. His wisdom, optimism and unquestionable faith in my abilities was encouragement that helped me to complete this thesis.

Family has also played an important role in the completion of this dissertation. Special thanks to my husband Alex for helping in many ways often invisible to others and for providing me with motivation. Thanks also to Angie, for offering her support and encouragement, and to my niece Veronika and my nephew Ethan for their limitless curiosity and optimism that have made my journey so much easier.

And finally, I want to thank Joanna, my sister and dearest friend, for never giving up on me and for enthusiastically reminding me the importance of completing my journey. Your unconditional love and endless understanding were a key to this successful ending.
Table of Contents

1 Introduction .......................................................................................................................................................... 1

2 Literature Review ............................................................................................................................................... 10
  2.1 Software Testing ........................................................................................................................................ 10
    2.1.1 Coverage Criteria .............................................................................................................................. 10
    2.1.2 Testing Web Applications .................................................................................................................. 12
    2.1.3 Testing Service-Oriented Applications and Web Services ............................................................. 13
  2.2 Software Reliability Models ....................................................................................................................... 15
    2.2.1 Software Reliability Growth Models ................................................................................................. 15
    2.2.2 Architecture-Based Reliability Models ............................................................................................... 18
      2.2.2.1 State-Based Models ..................................................................................................................... 19
      2.2.2.2 Path-Based Models ..................................................................................................................... 20
      2.2.2.3 Additive Models .......................................................................................................................... 22
    2.2.3 Reliability Models for Service-Oriented Applications and Web Services ........................................ 24
  2.3 Data Quality ............................................................................................................................................... 29
    2.3.1 Data Quality Metrics and Assessment ............................................................................................... 30
    2.3.2 Improving Data Quality ..................................................................................................................... 33
  2.4 Entity – Resolution ..................................................................................................................................... 34
    2.4.1 Entity - Resolution - Homogenous Data ............................................................................................ 35
    2.4.2 Entity - Resolution - Heterogeneous Data ......................................................................................... 39

3 Modeling and Testing of Service-Oriented Applications ............................................................................. 42
  3.1 Modeling of Service-Oriented Applications ............................................................................................... 43
    3.1.1 Modeling intra-tier dependencies ...................................................................................................... 44
      3.1.1.1 Presentation Tier .......................................................................................................................... 44
      3.1.1.2 Application Sub-Tier .................................................................................................................. 45
      3.1.1.3 Service Sub-Tier ......................................................................................................................... 46
        3.1.1.3.1 Atomic/ Composite Web Services ......................................................................................... 47
        3.1.1.3.2 Explicit Compositions ......................................................................................................... 47
        3.1.1.3.3 Implicit Compositions ......................................................................................................... 49
      3.1.1.4 Data Tier ...................................................................................................................................... 52
    3.1.2 Modeling Inter-tier Dependences ........................................................................................................ 52
  3.2 Testing of Service-Oriented Applications ................................................................................................. 55
    3.2.1 The Test Model .................................................................................................................................. 57
List of Figures

Figure 1 A three-tier architecture ................................................................. 43
Figure 2 An example of a Graphical User Interface ........................................ 53
Figure 3 A partial graph of an integrated dependence graph ........................ 53
Figure 4 The architecture view of the system ................................................ 82
Figure 5 The state view of the system ............................................................ 88
Figure 6 The transition matrix ....................................................................... 89
Figure 7 Two Collections of records ............................................................... 93
Figure 8 A pruned list of blocks & comparisons with traditional methods ....... 93
Figure 9 A pruned list of blocks & comparisons with our novel token and comparison selection method ................................................................. 94
Figure 10 Bloom Filter Algorithm ................................................................. 106
List of Tables

Table 1 Results of fault detection ..................................................................................... 62
Table 2 Results of fault distribution .................................................................................. 62
Table 3 Results of after-deployment fault detection ......................................................... 63
Table 4 After-deployment fault distribution ..................................................................... 63
Table 5 Results of fault detection ..................................................................................... 64
Table 6 Results of data quality ........................................................................................ 83
Table 7 Reliability and transition probability ................................................................. 87
Table 8 List of invalid data ............................................................................................... 88
Table 9 Data used in the first part of the case study ....................................................... 110
Table 10 Data used in the second part of the case study ................................................ 111
Table 11 Results of 2-stage Model for $D_{\text{movies}}$ .................................................... 114
Table 12 Brute Force results for $D_{\text{movies}}$ ................................................................. 114
Table 13 Results of 2-stage Model for $D_{\text{infoboxes}}$ .................................................... 115
Table 14 Brute Force results for $D_{\text{infoboxes}}$ ................................................................. 115
Table 15 Results of 2-stage Model for $D_{\text{movies}}$ .................................................... 115
Table 16 Results of 2-stage Model for $D_{\text{infoboxes}}$ .................................................... 115
Chapter 1

1 Introduction

Service-oriented applications have been widely adopted by the industrial, governmental and educational sectors to support inter-organizational communications. They are suitable for distributed systems since they allow a smooth interaction between business partners. The majority of applications that require inter-organizational resource sharing adopt a Service-Oriented Architecture (SOA) to reduce the complexity of interoperability. The benefits of SOA lie in decoupling clients and servers and standardizing the language in the middle. Thus, clients and servers can be created using different programming languages and still communicate by exchanging XML messages. Exchange of these messages among providers and requestors needs to be coordinated thus SOA solutions are often based on web services, defined as “modular, self-contained, self-describing, software components available over the Web” [41]. On the provider side, web services interact with their execution contexts in order to fulfill service requests. An execution context may include web server, application server, and database management systems. In addition, they may need to call required services from other applications. On the consumer side, businesses use web services to access resources from other organizations. Therefore, a service-oriented web application may include both provided and requested services and an execution context consisting of heterogeneous components. It is normally multi-layered, similar to multi-tiered web applications, where a presentation tier renders user interfaces, a logic tier processes business logics and a data tier stores persistent data. Information flows though these tiers to provide correct functionality of
applications. A distinct feature of service-oriented applications is the inclusion of a service sub-tier in the logic tier. The sub-tier is responsible for providing an interface to services that can be consumed by and from both internal and external organizations. Thus, the logic tier contains the business logic of both service and application components.

It is difficult to track the information flowing through these tiers without a model that properly renders the cross tier interactions. When a service fails to deliver the expected results, the responsible faults may reside in any tier of the application, or may simply be propagated from external services. To expose faults in testing, developers must take into account both the internal structure of the applications as well as their communications with external services and resources. Consequently, testing such applications requires creating test cases that can thoroughly exercise service request and response scenarios across tiers and components. This is challenging due to the heterogeneity of the applications. Heterogeneity of the applications is even more encouraged since SOA enables cooperation among different business partners. Since each of the businesses within a partnership will likely have developed their own applications, traditional testing methods will not work, because those assume that all software is developed within and for a single company. Thus, testing strategies must be developed that take into account the performance of external resources.

A number of techniques for testing web services have been proposed, the majority of which mainly focus on testing message exchanges through SOAP or other protocols, or service orchestration and choreography. There are also various strategies for testing web applications. However, the integration testing of services with their supporting components has rarely been explored.
Responding to this need for integration testing, we present in the third chapter a new technique for testing service-oriented applications. To examine the relationships between services and their execution context, we offer a dependence model for service-oriented applications, depicting intra- and inter-tier dependence relationships between services (both internal and external) and other components. Based on this dependence model, we propose a testing strategy combining the strengths of both black box and white box testing approaches. Since the implementation of external web services is hidden from the service consumers, we propose testing them without examining their internal structure. On the other hand, the dependence model captures the internal structure of the applications, enabling careful testing at the source code level to reveal code defects.

We first associated the dependence model with the specification to conduct specification testing, and then used node and dependence coverage criteria to guide additional test case generation and determine test adequacy. The concept of using node and edge coverage criteria to improve fault detection ability has been proven to be effective [108]. In node and edge coverage criteria, a node in a dependence graph may represent a user action (presentation tier), a service (service tier), a function (application tier), or a database table (data tier). An edge represents the control/call and the data dependence between nodes. The edge coverage will guide the selection of viable sequences of service invocations that are potentially fault-prone. This flexibility ensures that every function/action provided by every service will be tested. We adopted this concept to address the new problems that have arisen in service-oriented applications, which consist of heterogeneous components and often share resources with external organizations. The proposed model takes into account the complete structures of applications, including all of
the tiers and sub-tiers, which enables the computation of dependencies between and among all the heterogeneous components. The dependency information is useful to determine viable sequences of operations for testing. Our testing strategy not only improves the reliability of provided services, but also helps uncover faults propagated from external services and their misuse. More importantly, the majority of the proposed modeling and testing tasks can be automated which significantly improves the efficiency and effectiveness of testing.

Aside from testing, reliability is another very important aspect of service-oriented applications, particularly given the popularity of these applications. Traditionally, data definitions have been closely associated with designated business applications, often held within an organization, but not intended for outside use. Exposing these data resources through web services makes the data available to a variety of internal and external clients. This requirement increases the chance of unpredictable and invalid inputs, because not all users will have been trained by the organization, and external users may have varying levels of technical sophistication. Consequently, there may be a higher number of data quality problems in web services as opposed to traditional software applications, which will degrade the reliability of web services.

Invalid data introduces potential threats to the reliability of web services. A number of reports have shown that poor quality data in data sources of crucial applications has a significant impact on business [13,31,79,83,93]. Redman [93] reported the negative consequences of poor data quality, such as customer dissatisfaction, less effective decision making, and higher operational costs. Eckerson [31] provided an overview of the most common sources of data defects and gave multiple examples of companies that have
suffered losses due to invalid data. The Ohio Environmental Protection Agency reported [79] that an average of 5,000 data errors were being reported every month in their online Electronic Discharge Monitoring Report system. Pierce [83] showed that up to 6% of invalid data in US postal services were reported every year.

While a number of models exist for estimating the reliability of traditional software in which code defects are considered the main causes of failure, determining the reliability of service-oriented applications and web services introduces new challenges. Some studies have highlighted the importance of reliability of web services or evaluated the reliability of service-oriented applications.

Therefore there is an urgent need for a new way to measure service reliability that takes into account data quality. To respond to this challenge, in the fourth chapter we first developed an approach to measure data quality. Then, we modeled the interactions between services and data sources. Based on this model, we computed the reliability of web services by using an architecture-based reliability model that integrated data quality with software component reliability.

We examined how data quality affects performance reliability by conducting a case study on a service-oriented application developed and used by the New York State Police department. We show that for a deployed stable resource-driven system, the majority of the implementation defects may have been fixed. However, the application’s reliability may still be fluctuating because of its dynamically changing resources, particularly the data sources. This type of system utilizes the data stored in persistent storages such as Database Management Systems to provide services. The data are continuously updated, but some of them may have been entered incorrectly. For these invalid data, it is very likely that when
used by the system to provide service, the system will produce unacceptable results. Our empirical observation shows that more than 16% of failures were caused by invalid data during our observation period. Majority of the existing approaches for reliability estimation do not consider failures caused by invalid data; thus they cannot provide accurate reliability estimates. The results of our case study show that by considering data quality, the reliability modeling can provide more accurate estimates.

Recognizing the need for higher data quality, we focus on the Entity Resolution (ER) process that is widely recognize as a method of improving data quality. Entity resolution is the process of determining which records in a collection represent the same entity [16,32]. Thus, the process provides means of identifying duplicate records. Problem of duplicate records arises in many data mining projects especially considering the explosive growth of large-scale data, integration of multi-source databases, and increasing heterogeneity of data formats. It has huge practical implications in a wide variety of commercial, scientific, and security domains considering a growing number of both relational database management systems and NoSQL databases.

The major challenges in ER are computational complexity and effectiveness of matching techniques. Exhaustive ER algorithms that compare every pair of records comes with a very high computational cost \(O(n^2)\) [9,21,27,80,81,116] where \(n\) denotes the number of records in the data collection. This is very expensive for large datasets and normally is not scalable.

Blocking is one of the methods proposed to make ER more efficient. The process involves grouping records based on some criterion/criteria called blocking key/keys.
Records are split based on a common value of criterion/criteria such as: an attribute or a set of attributes. Only records within the same block need be compared since those belonging to two different blocks are unlikely to match. The approaches evolved over the years from the standard blocking [44] in which records are clustered into blocks if they share the identical blocking key into more advanced methods such as sorted neighborhood methods [22,45], canopy clustering [21,22], suffix array indexing [1], Q-grams [8].

Despite the decrease in computational complexity, blocking may not sufficient to restrict number of records comparisons to a point where the existing approaches would be feasible for large datasets. To increase the likelihood of finding matches, records are either placed in multiple blocks or the blocking criteria are loose and create large blocks. The number of required comparisons in large blocks may not be much lower than the number proposed by exhaustive ER implying a minimal increase in ER efficiency. Thus, the main objective of our study was to present an entity resolution algorithm that effectively processes the pairwise comparisons within blocks to establish comparisons that are more likely to find matches. Intuitively, the run time savings for blocks becomes significant if we only resolve comparisons that are the most likely to contain matching records.

To tackle the problem of overwhelming number of pairwise comparisons within blocks we derived an entity resolution algorithm that not only involves a blocking scheme but also contains a 2-stage comparison selection phase. First, drawing on the work of [80,81,82] we divided records into blocks using an attribute-agnostic blocking technique. Contrary to the existing approaches, however, we applied a token selection process to avoid creating large blocks of records that do not have much in common. Our token selection process disregarded tokens that occurred in the majority of the records, considering them
too common. Since this process is domain independent, the possibility of creating large blocks remained. Therefore, in the second step of our approach, we evaluated the blocks and excluded those with the largest number of possible comparisons. Once we had a list of filtered blocks, we applied the 2-stage comparison selection technique.

The 2-stage comparison selection process identifies comparisons that are most likely to find duplicates. In the first stage, we used information pertaining to the records themselves, such as the number of tokens they had in common or the number of blocks they belonged to. We applied a cheap similarity measure to group comparisons into three groups: those that were going to be resolved, those that were not going to be resolved since the likelihood of their matching records was too small; and those that that needed further analysis.

In the second stage, we used the characteristics of the block itself to further analyze comparisons from the third category. We focused on the length and frequency of the token that created the block. Once the analysis was completed, we were left with a list of comparisons to be resolved. We then preformed a resolution process while merging matching records.

We found Bloom-filter structures suitable for the efficient processing of comparisons since their adding and querying operations have computational complexity $O(1)$. They also use very limited storage space by avoiding storing the information included in records. Thus, we used Bloom filters to compute a similarity measure for each comparison.
Our experimental evaluation shows that the block filtering process combined with the 2-stage comparison selection process decreases the number of comparisons needed by an average of 40% in respect to other existing approaches [80,81], thus improving the efficiency of the entity resolution process while improving the effectiveness of the algorithm. The algorithm runs significantly faster than the results reported in [81] since we eliminate a greater number of oversized redundant blocks in the first step of our algorithm. Thus, in the comparison selection process we evaluate a smaller number of comparisons. Also, by employing our 2-stage comparison selection process we discover more duplicates. Each additional duplicate decreases number of comparisons since once records are identified as duplicates they are not compared again in subsequent blocks. We also obtain time savings by introducing Bloom filters to do operations on comparisons.
Chapter 2

2 Literature Review

2.1 Software Testing

Testing plays an important role in measuring the quality of software products. It reveals the existence of faults and verifies whether software satisfies a particular set of requirements. Consequently, it is essential in providing high quality software solutions. In order to ensure the high quality software all its artifacts and aspects needs to be verified.

There is a broad range of techniques that allows evaluating different pieces of software. Several of such methodologies and techniques have been proposed over the years.

2.1.1 Coverage Criteria

Testing software often involves collections of test cases called test suites, designed specifically for the testing process. In order to measure the adequacy of a test suite, researchers propose a set of criteria that a given test suite should satisfy. For traditional programs, these criteria include coverage of statements, branches, and definition-use associations of variables, and paths.

The concept of paths may derive from data flow analysis techniques in which “the test data exercise certain paths from a point in program where a variable is defined to points where the variable is subsequently used” [37]. Path selection criteria based on data flow relationships are needed for determining paths with high probability of revealing faults. The execution of these paths is a part of software testing process. Data flow-based testing for traditional test suite programs has been well studied [23,37]. In [37] Frankl et al.
modified an existing concept of data flow (DF) testing criteria proposed in [89,90] by eliminating unexecutable definition-use associations. According to their specification, unexecutable definition-use associations are those that do not have an executable complete path that covers them. The test criteria may require all of the associations to be covered. Then there may be a case where there is no test suite that is able to satisfy the given adequacy criteria. To avoid this undesirable situation, Frankl et al. proposed “requiring the test data to exercise only those definition-use associations that are executable”.

In [17] Chen et al. showed that coverage criteria based on object definition-use associations are effective in detecting faults related to inheritance, polymorphism/dynamic binding, and memory management. They proposed a class testing technique that reduces number of test cases needed for subclass testing and polymorphism testing. Their approach uses function dependency relationships to identify a group of inherited functions that need to be retested in the subclasses. Analysis of both direct and indirect function dependence relationships allows them to specify how many polymorphic substitutions needs to be exercised in order to provide adequate testing while avoiding exhaustive class testing.

Clarke et al. also focused on data flow based methodologies when determining coverage criteria [23]. Since it is not practical to test all paths in a program, they evaluated existing path selection criteria techniques based on data flow analysis [23]. Their evaluation consisted of providing descriptions, a comparison, as well as pros and cons of the path selection methods presented in [53,76,77,78,89,90]. They also highlighted the subsumption relationships and overlaps among the evaluated methodologies, finding that understanding these relationships helps to determine methodologies’ strengths and
weaknesses. Finally, Clarke et al. offered modifications to definitions of criteria to ensure their hierarchical completeness.

Kapfhammer and Soffa [48] emphasized the importance of testing not only an application but also its interaction with a database. In their study, a database is treated as a crucial aspect of a software system environment. Thus, in order to test database applications, they proposed a family of test adequacy criteria including all-database-DUs, all-relation-DUs, all-attribute-DUs, all-record-DUs, and all-attribute-value-DUs. These metrics are intended to identify defects that are commonly found in applications that interact with databases.

In [70] Memon et al. proposed coverage criteria for graphical user interfaces (GUIs) based on events and event sequences. Their criteria provide an objective assessment of test quality. Since GUIs provide front-ends to software, they can be complicated in structure and consist of a large number of elements. To tackle this challenge, Memon et al. proposed decomposing GUIs into smaller, manageable parts called components that can be tested in isolation. Events occurring within a component are identified using event-flow graphs and evaluated using intra-component criteria. Events occurring between components are captured in the integration tree and evaluated using inter-component criteria.

2.1.2 Testing Web Applications

A number of models for analyzing and testing web applications have been proposed. Ricca and Tonella [94] proposed a UML-based navigation model to represent the structures of web applications. Based on the model, a path expression can be generated that facilitates test case generation. Because the number of paths can be explosively large,
they suggested a set of coverage criteria, including coverage of pages, hyperlinks, definition-uses, all-uses, and all-paths to determine test adequacy based on each criterion. The definitions and the uses refer to the data flows between web pages.

A similar model proposed by DiLucca et al. [29] includes unit testing that uses web pages as test units and function level testing that makes use of decision tables to aid test case generation.

Liu et al. [59] proposed a Web Application Test model (WATM) where each component of the web application is considered as an object, and the elements of an HTML or XML document are considered as structured variables within an object. Liu proposed five levels of dataflow testing, including: function, function cluster, object, object cluster, and application level.

Tonella and Ricca [104] proposed a control-flow based model for analyzing control transfers within server pages, and based on their control flow model they proposed structural coverage criteria including paths, branches, and nodes.

2.1.3 Testing Service-Oriented Applications and Web Services

A number of techniques for testing service-oriented applications have been developed. In [67] Mei et al. proposed coverage criteria to evaluate the quality of test sets for service-oriented applications called Web Services Business Process Execution Language (WS-BPEL) applications. Business logic for these types of programs is captured by BPEL. XPath, a query language, is used to extract content from an XML message returned by web services. Mei et al. emphasized the importance of testing XPath and its
interactions with BPEL in order to provide sufficient test coverage of WS-BPEL programs. To achieve the goal, similarity to traditional approaches, they aimed to model BPEL applications as a control graph. They modeled conceptual variables in XPath and ordinary variables in BPEL by reusing the concept of data flow testing presented in [36]. These sets of variables allowed identifying data flow associations between XPath and BPEL. Based on these associations the family of testing criteria for measuring quality of test suites was derived.

In 2009 Mei et al.’s focus moved towards testing criteria for service choreography [68]. To facilitate data flow testing for choreography applications, they proposed a Label Transition Systems (LTS) – Based Choreography (C-LTS) Model.

Within the same year, they introduced new techniques for test case prioritization in the testing of service–oriented business applications [69]. The business workflow for these types of applications often includes interactions with web services defined using Web Services Description Language (WSDL) and XPath. Thus, their multi-tier model takes into account coverage at three levels: level 1: workflow; level 2: workflow and XPath; level 3: workflow, XPath, and WSDL. The methodology is presented from the perspective of regression testing. Rerunning all of the tests during the regression testing stage may not be feasible. Therefore, in order to maximize the effectiveness of the regression testing and reduce the number of test runs they presented prioritization techniques for each of the three levels.

Tsai et al. studied testability evaluation criteria for service-oriented architecture (SOA) software. They discussed the degree to which an SOA-based application is designed to facilitate the creation of testing criteria [105]. Tsai introduced a framework
that eliminates test cases with overlapping coverage by addressing SOA dependability and trustworthiness.

Bartolini et al. [6] introduced a method of obtaining an automated test suite for web services. The generation of the test cases is driven by coverage of operations and provided XML schema. The final result of the study is an early implementation of the WS-TAXI framework. This framework may reduce efforts required to test web services but does not address the dependencies of web service within multi-tier web applications. Automatic test case generation for web services is also discussed by Bai et al. [3].

### 2.2 Software Reliability Models

Software reliability models estimate the existing reliability and predict the future reliability of software systems. In order to perform these tasks, they identify general dependencies of the failure process on fault introduction, fault removal and use of the software systems. Various models cover different scenarios where features may or may not be added and faults may or may not be removed [72].

#### 2.2.1 Software Reliability Growth Models

Software reliability growth models generally presume that the reliability of software increases with the removal of the faults. Faults are removed immediately after being discovered. For these types of models, it is typical to assume that failures are independent from each other and that their occurrence follows a random process [72].
The first software reliability models were proposed in the late sixties and early seventies. Since then, a number of different software reliability models have been introduced [64]. Models are classified according to attributes such as time domain, category, type of distribution, class, and family [63].

In 1968, Pollack proposed one of the first software reliability growth models. It views reliability growth and prediction as a Bayesian framework. The authors recognized two possible states of the system: repaired and unrepaired. Based on an analysis of the states, and parameters such as failure rate when system is in the unrepaired state, failure rate when system is in the repaired state, and repair probability, they presented methods of determining current and future reliability. These strategies are provided for both discrete and continuous models, and take into consideration cases of known and unknown parameters [85].

A few years later, Musa introduced a basic execution time model [71]. In contrast to Pollack’s approach, Musa employed execution time rather than a wall-clock time. Also, in Musa’s model, failures are assumed to occur as a nonhomogenous Poisson process, not a Bayesian process. The model is based on two parameters: initial failure intensity at the start of test execution and the total number of failures that would occur in infinite time. These parameters need to be estimated or predicted in order to use the model. Failures are assumed to occur as a nonhomogenous Poisson process. To overcome the difficult of dealing with fitting data from highly nonuniform operational profiles and improve the forecasting of failures and faults, Musa and Okumoto introduced another execution time model called the Logarithmic Poisson Execution Time Model [73]. It also uses nonhomogenous Poisson processes, but, in contrast to the basic model, the change in the
failure intensity function becomes smaller with failures experienced. Therefore, faults discovered in the beginning of test execution have a greater impact on reducing the failure intensity function than those discovered closer to the end of the process [63]. This model is based on two parameters: the total number of failures and the failure intensity decay parameter. Similarly to the basic execution model, the parameters need to be estimated in order to use the model. Both models allow determining the current number of failure per unit time, the number of failures experienced after, for example, 10 or 100 execution hours of execution, the total number of remaining failures, and the additional execution time required to reach the failure intensity objective.

The necessity of considering the impact of encountered failures on the failure intensity function has also been studied by Littlewood in his scholastic reliability growth model [56]. He pointed out that faults fixed earlier will tend to result in greater improvements to the system than those removed later. In order to take this fact into consideration, he proposed a gamma distribution to represent uncertainty about the occurrence rate of the faults. His model allows determining the total execution time required to reach the failure intensity objective as well as the additional expected number of failures that must be experienced to reach the failure intensity objective.

Littlewood shifted his focus into Bayesian models [55,57]. These models assume continuous time processes. The idea of exploring continuous time processes was initiated by the author’s interest in a real-time system in a power station. The proposed methodology allows the possibility of reduction in reliability due to poor quality of fault fixes. To reflect this possibility, after each repair a sequence of software programs is generated. The new version of the program may be better or worse than the previous one. Therefore, Littlewood
et al. chose the exponential distribution to model time between failures with a certain failure rate, which is treated as a random variable that is assumed to follow a Gamma distribution.

Goel et al’s software reliability model belongs to a group of nonhomogeneous Poisson models [39]. The presented model differs from other models by the form of mean value function, which is an exponential function that depends on two parameters: the expected total number of failures to be eventually detected and the initial number of faults in the system. The latter value is considered a random variable while in the approaches of other researchers it was an estimated constant. Making time between failures dependent on the time of previous failure is another novel feature of this model.

2.2.2 Architecture-Based Reliability Models

Architecture-based models are a group of models often used for large systems. In this type of model, assessing the reliability of systems depends on factors such as components and interactions among them. State-based models, path-based models, and additive models are the three main categories of architecture-based models. In state-based models a control flow graph is used to model software architecture. A Markovian behavior of transfer of control among program modules is assumed. There are three Markovian processes that can be applied: discrete time Markov chain, continuous time Markov chain, or semi Markov. Path-based models consider the execution paths of the program to compute reliability. The reliability of each path is computed as the multiplication of the reliabilities of its components. Additive models state that the reliability of each component can be modeled by a nonhomogeneous Poisson process. They assume a specific
distribution process for each component's failure behavior and propose computing the cumulative failure and failure intensity functions for the system as a sum of the corresponding functions of all the components belonging to the system.

2.2.2.1 State-Based Models

Cheung’ model is categorized by Goseva-Popstojanova et al. as a state-based model [40]. It was introduced in 1980 as one of the earliest models that viewed the reliability of the system with respect to the reliabilities of components and transition probabilities. He developed a model by identifying the components of the software application and creating a control flow graph [18]. He assumed that the transfer of control between components had a Markov property. The reliability of the system was computed based on the reliability of each individual component and the measured inter-component transition probabilities. Based on the control graph Cheung created a transition matrix. Its entries represented the probabilities of transition between states. This matrix allowed determining the overall reliability of the system. The reliability of the system can be also interpreted as the sum of reliabilities of all paths that start at the entry node and end at the exit node [40].

In 2006 Wang et al. extended Cheung’s model [108]. They developed an architecture-based reliability model that takes heterogeneity of software architecture into account. They incorporated a white-box approach on top of a traditional Markov-based reliability model. In addition to three attributes introduced by Cheung: system architecture, component reliability and transition probabilities, they utilized styles to characterize the organization of a software system. The four styles are as follow: batch-sequential, parallel/pipe filter, fault tolerance, call-and-return. In a batch-sequential style, components are executed in a sequential manner i.e. only a single component is executed in any instance
of time. In a parallel/pipe filter style, a set of components is executed simultaneously to improve performance. In a fault tolerance style, a set of fault forbearing components included backup components to compensate for failures of the primary. In a call-and-return style, a calling component (caller) can invoke a called component (callee) a number of times, and each time the callee returns the requested service back to the caller. The reliability of components and the transition probabilities among components are evaluated according to the architecture styles. Then, similarly to Cheung’s model, the transition matrix is constructed. The reliability of the system is computed using entries from the matrix.

2.2.2.2 Path-Based Models

Poore et al. proposed an approach to make developers more open towards reliability planning and certification [87]. One of their strategies employed component reliabilities and transition probabilities. These parameters were used to perform what-if analyses to examine the sensitivity of the system's reliability to its components. In order to represent the system components and their interactions, a control graph is presented. This graph contains two parameters: component reliabilities and transition probabilities. Each component has a default value of 1.0 and a transition probability estimated based on the system design and the intended operational profile. It is not specified how this graph can be constructed, however it can be assumed that knowledge of the system domain could be used. Poore et al. designed the Cleanroom Reliability Manager [86] to calculate system reliability based on the transition probabilities and component reliabilities.
Yacoub et al. extended Poor’s model by providing methods to obtain model parameters such as transition reliabilities and constructing a formal traversal algorithm. Similarly to Poore’s approach they proposed a technique based on scenarios [120]. This technique helps to analyze the sensitivity of the reliability of systems that are hierarchically distributed in nature. System components/subsystems and link reliabilities are included in the Component-Dependency Graphs (CDG). This graph shows the execution dependency between components/subsystems and possible execution paths. The traversal algorithm (SBRA) is applied to the graph. This algorithm uses reliabilities of components, subsystems and links to analyze the sensitivity of the system's reliability. In 2004, they expanded their concepts by distinguishing different categories of scenarios for component-based applications. In addition, they defined the reliability of the application as a function of component reliability, a function of transition reliability and a function of the scenario profile [121].

Yacoub’s model was improved by Zhang et al. through treating reliability of the components as a function of operational profile [123]. The algorithm proposed by Yacoub assumes that component reliability changes from being constant to being dependent on where the component occurs in the system, and how it is transitioning between states. Therefore, the reliability of the components changes when the operational profile is modified. The model combines the idea of subdomains proposed by Hamlet [43] and path-based architecture model [120,121].

Hamlet et al. conducted a study to establish the reliability of components from the perspective of system developers [43]. Following industrial standards, they assumed components are described in a handbook. Each component is supposed to have a data sheet
entry that includes a description of component functionality, and this provides a system constraint that allows future users to assess subjectively the component's usefulness. Hamlet et al. questioned the lack of sufficient information about component quality within the data sheet. To address this issue they proposed adding quality information. A list of component input subdomains, reliability mappings and profile-transformation mappings are key values that need to be included in the data sheet [42]. These values allow systems developers to compute the reliability of prospective components that may be placed in their system. Since the reliability of the component depends on the operational profile of the system, the data sheet takes profile as a parameter. Hamlet et al. emphasized that data sheets are not sufficient to make decisions about system properties. They promoted trying various system designs to satisfy the system requirements.

2.2.2.3 Additive Models

Additive models are the smallest group of architecture-based models. One additive model was introduced by Xie et al. in 1995 [119]. They approached software reliability starting with the assumption that for each subsystem or module within the system, testing and reliability assessment can be performed independently. Moreover, they assumed that there are no issues with the interfaces of any two components. The failure process of subsystems and modules are further assumed to follow nonhomogeneous Poisson process. Evaluating the system failure intensity function and the mean value function are the main results of the study. The system failure intensity function can be computed as a sum of modules failure intensity functions. Similarly, the system mean value function is a function of a sum of subsystem level mean value functions. The model presented is very generic
and does not analyze deeply the architecture of the systems. It recognizes the existence of subsystems but does not distinguish styles of their interactions.

Everett introduced another additive reliability model. It uses the Extended Execution Time reliability growth model to evaluate the reliability of components [34,35]. In his studies, he emphasized the effect of the order of test cases on reliability growth. He proposed a six step strategy for performing software reliability analysis before testing begins. The six steps are as follows: divide a system into components, characterize properties of each component, characterize how usage stresses a component, model reliability of each component, superimpose component reliabilities and confirm the model during testing. Similarly to Xie, he computed the cumulative failure and failure intensity functions for the system as a sum of corresponding functions of all the components belonging to the system.

Since architecture-based models can be used in the beginning phase of the projects, they can help with optimizing and improving code. In addition, they take into consideration an operational profile of the systems. Therefore, they do not have a false assumption that each component contributes equally to the reliability of the systems. The necessity of knowledge of the internal software structure can be viewed as an obstacle. It is quite common that only developers can offer this kind of expertise. Overall, architecture-based models gained their popularity in the last two decades due to the growing complexity and size of software systems [40].
2.2.3 Reliability Models for Service-Oriented Applications and Web Services

Determining the reliability of service-oriented applications introduces new challenges not typically encountered in other types of applications. Historically, a lot of software products have been developed within a single company. This way the developers had full control of the development process. For service-oriented applications this is often not the case. Service-oriented applications promote smooth interactions between loosely coupled services through standard protocols. They may use services offered by other providers and written by different developers. Services are often implemented in a form of web services defined as “modular, self-contained, self-describing, software components available over the Web” [41]. Dietel et al. defined reliability of web services as: “the ability of web service to function correctly and provide consistent service, even in the event of system or network failure” [28]. It is important to note that service-oriented applications include not only web services but also the execution context consisting of heterogeneous components. On the server side, web services interact with their execution context to fulfill service requests. An execution context may include application components rendering graphical user interfaces and processing business logic, and databases storing persistent data. On the client side, external services are requested to meet the organization’s business needs that require resources from other organizations. The requested external resources may also be used by services provided by the organization. Existing software reliability models focus on determining the reliability of atomic and composite web services. These models often ignore the execution context; instead they employ a black-box approach when computing the reliability of the atomic web services. Then they are using the available information about the architecture of the composite web services to compute their
reliabilities. In the end, the reliability of a service-oriented application is viewed as the reliability of entire set of web services.

Wang et al. suggested determining the reliability of service-oriented applications by looking at the reliability of atomic, composite web services as well as the reliability of service pool and data reliability [109]. Data reliability, functional reliability, binding reliability and connecting reliability are factors in evaluating reliability of atomic web services. The reliabilities of both atomic and composite web services are estimating using an exponential distribution formula defined by Musa. The model also introduces a strategy of picking up the best service out of the pool of services as a backup. The service pool contains services with the same functionality. In order to locate the best alternative service, the reliability of the service pool is analyzed with a Discrete Time Markov Chain (DCMC) model. The state of the pool changes whenever a current invoked service fails. The analysis of the reliability of the entire system involves examining the reliability of all atomic and composite web services with and without the pool mechanism. In the case where the pool mechanism is employed it is necessary to establish a service pool for each atomic service in the composite service.

Li et al. viewed web service applications as a hierarchical set expression of a composite web service [54]. To evaluate overall reliability of web service applications they proposed determining the reliability of atomic and composite services. The reliability of atomic web services is calculated based on an extended UDDI model that captures invocation, transition and failure information. In contrast to Wang’s, Li’s group uses the Business Process Execution Language (BPEL) specification to discover the structures of composite web services. Analysis of the provided information allows them to create a
Structure Analysis Chart that shows the internal logic relationships of composite web services. Reliability computation equations are built for all kinds of service composition structures. The Structure Analysis Chart is necessary to obtain a set expression of a composite web service. The algorithm to calculate global reliability based on this set expression is the final result of the study.

Tsai et al. introduced the Service-Oriented Software Reliability Model (SORM) [105]. Similarly to Li, they assumed that values captured in the UDDI registry can be used when determining the reliabilities of atomic web services. Nevertheless, they used a different mechanism for testing web services by using the UDDI servers introduced in 2003 [106]. In this former paper, Tsai et al. proposed a strategy that employs a check-in and check-out verification mechanism that was supposed to be added to the existing UDDI protocol. Service providers would be required to provide not only web services but also test scripts in order to register their services. Before the services could be registered the UDDI server would perform a check-in mechanism consisting of running the test scripts to verify correctness of the web services. UDDI server would perform a check-out mechanism consisting of providing web services after they passed the client’s testing. First, a client would receive the requested candidate web services with the test scripts. She/he would then run the test scripts and, if satisfied, the web services would be made available to her/him. If the registry is public, this process can take place without additional constraints. When a fee is requested for web service usage, the service provider may not be willing to supply the web services just for testing. Tsai et al. proposed a number of ways to address this issue; usage count, time count, access restriction, or a hybrid approach. In addition, they suggested a hierarchical organization of the scripts. Child scripts inherit their parents’ tests.
but not the other way around. Furthermore, Tsai et al. applied group testing and majority voting to evaluate the reliabilities of atomic services. A single input can be forwarded to a group of services with the same functionality. The results from all services are sent to a voting service that detects faults by comparing the output of each service with the weighted majority output. Once the reliabilities of atomic services are determined, they proposed constructing a scenario model to calculate the reliability of composite web services and the entire system. The operational profile and architecture of the system are used to construct the scenario model. It is assumed that system behaviors are specified by a few system scenarios. The scenario is defined as a sequence of activities connected by the four operators: sequence, loop, choice and concurrency. Tsai et al. showed that the overall reliability of the system is based on the reliabilities of the scenarios.

An interesting reliability analysis has been presented by Tian et al. [103]. They describe how workload may influence the reliability of web applications. Number of hits, number of bytes transferred, number of users, and number of sessions are some of the possible workload measures. The workload measures can be extracted from web server access logs. Information about errors can be found in the web server error log. Comparison of web site usage and failures occurred provide valuable information of the relationship between these two factors. Tian et al. discovered that there is almost a linear relation between cumulative errors and cumulative bytes transferred. Thus, they argued that knowledge of workload patterns can be helpful in identifying risks related to various workloads and determining the operational reliability of Web sites.

Sasikaladevi et al. provided a new reliability evaluation framework. Two quality attributes are considered when evaluating reliability: availability and accessibility.
Availability is defined as the measure of web service being present or ready for immediate use. Accessibility represents the capability to service a web service request. Based on these two attributes an availability matrix and accessibility matrix are created. The vectors V and X are derived from the matrices with maximum value on every row in each matrix. By comparing the vectors’ values with the threshold values provided by consumers, it is possible to establish if evaluated services are better or worse than those currently used by consumers. Based on this analysis consumer services are replaced by new ones or, in the case where the tested services are worse, the next set of services is tested. Reliability assessment is done by the services broker on behalf of the consumer [97].

Similarly to [91], Zo et al. focused on the quality of services [126]. Among non-functional requirements such as reliability, security, cost and performance, they examined the importance of reliability in selecting from a pool of alternative web services. It is assumed that reliability measures for individual web services are available. Establishing methods of selecting a set of web services to support a given task is the main goal of the paper. The problem is presented as a case when m tasks need to be completed using n web services. The total number of viable combinations is given by multiplication m * n. As a result, an exhaustive search is not a feasible solution. Instead the effectiveness of a genetic algorithm, which is an adaptive search technique, is evaluated. The results of applying the genetic algorithm to a drop-ship business process indicated that the method was very useful in selecting best services to support business processes.

Liu et al. proposed enhancing the reliability of BPEL processes by improving fault handling [59]. Their strategy employs Event-Condition-Action (ECA) rules that specify fault handling logic. These rules contain three parts; event, condition, and action. The
event is used for fault characterization and provides a unique name for the fault. The condition indicates if a given fault needs to be handled. The action shows how the fault should be handled though a fault-tolerant pattern. Four different patterns are presented; ignore, skip, retry, alternate. Depending on the type of fault, one or a combination of patterns can be applied. All three parts of the ECA rules need to be transformed into BPEL code. Lie et al. provided a description of how the transformations can be done. In parallel to defining the fault handling logic, developers are encouraged to specify the normal business logic in the initial BPEL process. Both, the ECA rules and the normal business logic are used to generate the fault-tolerant BPEL process. In order to assist developers, the authors constructed a tool that allows entering/deleting ECA rules and generating the integrated fault tolerant BPEL process documentation.

### 2.3 Data Quality

A variety of approaches have been proposed for assessing, analyzing and improving data quality [5,60,65]. A number of reports have shown that poor data quality in applications has a significant impact on their effectiveness [13,31,83,93]. Redman [93] reported negative consequences of poor data quality such as customer dissatisfaction, less effective decision making, and higher operational cost. Eckerson [31] provided an overview of the most common sources of data defects and gave multiple examples of companies that have suffered losses due to invalid data.
2.3.1 Data Quality Metrics and Assessment

Numerous studies have been proposed for data quality assessment [84,92,110]. Pipino et al. derived techniques to develop usable data quality metrics. They discussed multiple quality attributes such as free-of-error, completeness, consistency, believability, timeliness, and accessibility. According to their study, measuring data quality involves objective assessment and subjective assessment. Subjective assessment presents users’ perception of the data, whereas objective assessment is obtained using functional forms such as simple ratio, min or max operation, and weighted average. In the end, the results from both assessments are compared. In case conflicting assessment issues arise, the data providers need to determine causes and take necessary actions to improve data quality [84]. Similarly to [84], Wang et al. analyzed multiple data quality dimensions [110]. They used an empirical approach in their research to capture data quality dimensions that are important to data customers [110]. They yielded the final list of twenty dimensions that encapsulated 118 data quality attributes. The importance of determining the list of quality dimensions and attributes from collected customer’s responses instead of defining them theoretically based on researchers’ experience is the main focus of the study. In addition, obtained results presented a hierarchical view of data quality dimensions. Data quality dimensions are defined as sets of data quality attributes that represent a single aspect or construct of data quality. In the other words, dimensions are sets of attributes of data quality e.g. relevancy, completeness etc. In order to obtain a set of possible attributes, they collected data from data users. 179 attributes were identified as a result of the first survey. The importance of the attributes was captured in the second survey when data consumers were asked to rate importance of each attribute on a scale from 1 to 9, where 1 denoted
extremely important and 9 denoted not important. In the second study, dimensions were grouped into categories, and then the correctness of the grouping was certified by data consumers. Finally, four categories of dimensions were defined: intrinsic data quality (DQ), contextual DQ, representational DQ and accessibility DQ. The intrinsic DQ consisted of dimensions such as accuracy, objectivity, believability and reputation; contextual DQ consisted of value-added, relevancy, timeliness, completeness and appropriate amount of data; representational DQ consisted of interpretability, easy of understanding, representational consistency and conciseness of representation; and accessibility DQ consisted of accessibility and access security. From the analysis of the obtained set of dimensions Wang et al. drew the conclusion that quite often Information Technology (IT) professionals have a different perspective of data than that presented by customers. Therefore, they emphasized the importance of customer input when capturing aspects of data quality. Ultimately, customers judge whether or not a product meets their expectations. Essentially, the framework proposed by Wang et al. can make information system (IS) managers better understand and meet their data consumers’ data quality needs.

Strong et al. continued to focus on viewing data quality from the perspective of data consumers [100]. She performed an analysis of quality problems in three organizations: Golden Air, Better Care and HyCare. The data was collected via interviews with the companies’ employees. Three different patterns of quality issues were identified: an intrinsic data quality pattern, an accessibility data quality pattern and a contextual data quality pattern. The first was observed to be due to data mismatches. These issues made data sources inaccurate and not believable. Accessibility concerns were discovered in all three companies though they were different in nature. Golden Air experienced accessibility
problems due to unreliable data communication lines. Better Care had to deal with access security concerns, whereas HyCare personnel encountered difficulties with interpreting medical codes which became a barrier to accessibility. Strong et al. also observed three types of contextual data problems: incomplete data, inconsistent data definitions and representations, and inability to combine data coming from different sources. Consequently, data consumers experienced difficulties with retrieving data, with matching hospital claims and with mismatches between their inventory system and physical warehouse counts. Strong et al. in their conclusions emphasized the importance of data quality dimensions as well as the contextual analysis of data quality.

Similarly, other researchers such as Bailey et al. and Ives et al. viewed data quality attributes and dimensions such as accuracy, timeliness, precision, completeness, relevancy as factors influencing user information satisfaction [4,47].

The importance of data quality has been also emphasized in work of Evan and Shankaranarayanan [33]. They presented a dual assessment methodology to quantify the quality of data. In their study, the authors argued that evaluating data quality outside its context was not sufficient. They distinguished evaluating data quality independent of the context in which the dataset is used, referring to this process as “impartial assessment,” and measuring the impact of data quality issues within a context, referring to this process as “contextual assessment”. They associated impartial assessment with costs, since it provides a proportion of how many records are defective over the total number or records. Contextual assessment is linked to benefits gained from improving data quality since it provides a measure of how improving data quality will affect its usability.
2.3.2 Improving Data Quality

Cong et al. presented an approach for improving data quality [25]. They focused their attention on two dimensions: accuracy and consistency. To improve the consistency of data they proposed using a class of constraints called “conditional function dependencies” which was originally presented in [15]. These constraints extend the notion of traditional functional dependencies (FDs) by attempting to evaluate the consistency of data via incorporating bindings of semantically related values. Two algorithms based on conditional function dependencies were developed to find repairs that are required for data cleanup and to incrementally identify repairs after the state of a database has changed. The repairs may require changing one of multiple attributes in the record. In order to determine which attribute should be adjusted, they placed a weight on each attribute of each tuple in the dataset based on the presumed accuracy of the attribute. Also, the cost on updating the attribute was taken into consideration when deciding the specifics of the repair.

In [97] Scannapieco et al. proposed a Unified Modeling Language (UML) profile for data quality. They noticed that an effective representation for data quality improvement processes is a crucial but yet not fully explored research area. Thus, their methodology adopts the existing Information Product - Information Production Map (IP-MAP) presented in [99] with a novel UML-based modeling formalism to produce the IP-UML methodology. They derived the Data Analysis Model, the Quality Analysis Model and the Quality Design Model to develop the UML profile. These three models allow specifying which data elements are crucial, detecting and representing data quality requirements e.g.
identifying data quality dimensions, verifying the current state of processes producing data, and providing directions for improvements. They are derived by a data improvement process consisting of three phases: Data Analysis, Quality Analysis and Quality Improvement Design.

Prasad et al. provided another data quality improvement methodology [88]. It includes 4 stages: the investigation stage, the standardization stage, the de-duplication stage and the survivorship stage. These stages provide means of identifying types of data quality issues, correcting spelling errors, unifying data formats and optionally eliminating duplicate records. In addition to standard data cleansing, they presented a data quality workflow that incorporates synonym finding, a ripple down rules (RDR) framework and a variant finder. RDR clarifies and organizes rules identified in the standardization stage whereas the variant finder is helpful in discovering data inconsistencies and identifying duplicate records. The synonym finder can be used to populate dictionary entries that are utilized to detect data inconsistencies.

2.4 Entity – Resolution

Entity resolution (ER) is the process of determining which records in a collection represent the same entity [16,32]. The problem arises quit often when integrating data coming from multiple resources. ER techniques are increasingly being used in many application areas to improve data managements, processing and analysis.

An exhaustive ER process has time complexity of $O(n^2)$ [9,21,27,80,81,116] where $n$ denotes the number of records in the data collection, since in the worst case each record needs to be compared with every other record in the collection and often involves
complicated logic for deciding whether records are matching. Various techniques for homogenous and heterogeneous data have been proposed to make ER more efficient by dividing records into blocks and only comparing records within the same block to reduce the number of costly pairwise comparisons.

2.4.1 Entity - Resolution - Homogenous Data

The high importance and difficulty of the ER process triggered a huge amount of research aiming to address the ER problem especially for homogenous data. Homogenous data is often characterized by a schema which formally describes its consistent structure. To improve efficiency of ER, records are split into blocks according to some criterion/criteria. Assuming that the schema is available in advance, the most appropriate attributes are selected as blocking criteria called blocking keys. Usually the most distinctive attributes of the data are considered to be the most appropriate. If records share the same value for the blocking key, they are assigned to the same block. Only records within the same block need be compared since those belonging to two different blocks are unlikely to match.

The sorted-neighborhood method is one of the common blocking techniques proposed to reduce the number of pairwise comparisons. In this approach, first it is necessary to select a key for each record in the list by extracting a relevant field or set of fields. Then records are sorted based on the key. A window of a fixed size $w$ is moved over sorted records. Only records within the window are compared. That is why the method has the total time complexity of $O(wn)$ where $n$ denotes the total number of records in the database [22,44]. One disadvantage of this approach is that the window size is difficult to
configure. If the window size is selected too small, some matches might be missed, since a number of records that have the same value of sorting key will be larger than the window size. On the other hand, too large window results in many unnecessary comparisons. Even with the ideal window size, many unnecessary comparisons are executed, since clusters of records with the same value of sorting key likely have different sizes.

Christen et al. proposed Canopy Clustering as a solution for resolving large data sets [21,22,66]. The key idea involves using a cheap approximate distance measure to divide the data into overlapping subsets called canopies. A randomly chosen record becomes an initial canopy center. Then, based on calculated distances, records are placed in one or more canopies according to selected distance thresholds. Only records placed in the same canopy will be compared. A traditional clustering algorithm such as Greedy Agglomerative Clustering is executed to calculate the distance between records in the same canopy. A disadvantage of this approach is that the distance thresholds are difficult to configure. If they are too low, some matching records will be missed since they will be placed in different canopies. If thresholds are too high, a significant number of unnecessary comparisons will be performed. Also this method is sensitive to the choice of initial clustering center, with different initial centers often corresponding to different clustering results with different accuracies.

Aizawa et al. [1] presented a blocking technique that exploits a suffix array structure. It provides a domain-independent method of grouping possible associated records. The proposed framework is based on key-based blocking but uses a suffix array to store tokenized key attribute values. Two levels of granularity are provided; tokens can be either characters or terms. An inverted index is used to identify records containing
tokens from the suffix array. The records may be placed in multiple blocks. Similarly to other approaches, only records within the same block are compared.

In Q-gram techniques, the values of blocking keys are converted into lists of Q-grams [8], which are sub-strings of q consecutive characters. Each record is inserted into multiple blocks using a Jaccard-based similarity threshold between the lists of Q-grams and its blocking keys. The Jaccard similarity measure between two strings \( b_i \) and \( b_j \) is calculated as the number of common Q-grams among \( b_i \) and \( b_j \), divided by number sum of all q-grams in the two strings. Once records are placed in blocks, pairwise comparisons may be used to determine which are records are duplicates.

Most previous work focuses on finding the best blocking criteria and then processing blocks separately. In [116], Whang et al. took a step forward with the introduction of an interactive blocking technique. In their study, records are placed in blocks based on their common characteristics. The algorithm allows multiple blocking criteria. As a result, a single record may be placed in multiple blocks. Rather than stopping after processing records in each block, the results are relayed to subsequently processed blocks. A similar exploitation of associations between different blocks of records has been discussed in [10,30].

Among these methods, few have focused on scalability. Whang and Molina [112,114] proposed a solution that addresses scalability. Their framework resolves multiple datasets of records simultaneously using several processors. The framework allows “plugging in” existing ER algorithms. In order to coordinate the various ER algorithms, a scheduler is introduced. Since datasets may be too large to be loaded into memory, the framework uses traditional blocking techniques. These techniques cannot guarantee that
all matching records will be compared. The scalability of the framework relies on the performance of the specific ER algorithms “plugged in” the framework and does not take into consideration that the matching process may need to be done in an on-line mode. Consequently, it may not be the best solution for real-time systems.

To tackle this challenge, a recent line of work focuses on near real-time entity resolution. Real-time systems are characterized by a limited amount of time for performing computational tasks. In [115] Whang and Molina presented a “pay-as-you-go” approach, which provides partial results where time constraints are enforced (e.g., in real-time systems). Their concept supports obtaining partial results “gradually” so it is possible to attain some initial results faster than in traditional ER algorithms. In this methodology, candidate pairs are ordered by likelihood of a match. The ER algorithm performs record comparisons taking into consideration that first records that are more likely to match. As a result, a subset of matching pairs is discovered faster than by considering candidate pairs in an arbitrary order. Thus, with a limited amount of time, an incomplete analysis may still provide sufficient results.

In [22] Christen and Gayler employed three indexing methods for the real-time entity resolution. The authors distinguished two different phases of the process: building an index using a cleaned and duplicate-free static database, and querying the existing index against a stream of incoming records. The result is a ranked list of matches retrieved from the index for each incoming record. These approaches improve matching speed at the cost of lower effectiveness compared to standard blocking techniques.

Three effective ER algorithms: G-Swoosh, R-Swoosh, and F-Swoosh were presented by Benjelloun et al. in [9]. These algorithms employ match and merge functions.
The match function determines if two records are duplicates, whereas the merge function combines information from two matching records into a new record. The assumption is made that both merge and match function are black-box methods. If the match and merge functions have following properties; idempotence, commutativity, associativity and representativity they are considered to lead to efficient algorithms. Thus, assuming the existence of the match and merge functions with these desirable properties, the authors present three ER algorithms. The G-Swoosh algorithm is a simple, naive algorithm that does not require the match and merge functions to satisfy any particular properties. In order to eliminate some of the pairwise comparisons, the G-Swoosh algorithm maintains two sets $I$ and $I'$. Set $I$ contains records that have not yet been compared, whereas set $I'$ contains records that have all been compared with each other. The R-Swoosh algorithm takes advantage of the match and merge functions having the desirable properties to avoid many comparisons. F-Swoosh has a similar structure to that of R-Swoosh but at each step it tries to avoid performing redundant or unnecessary feature value comparisons.

### 2.4.2 Entity - Resolution - Heterogeneous Data

Dealing with unstructured or semi-structured data is a difficult problem, since attribute names may not match across records. Also, heterogeneous data sets are often not described by a consistent schema. To tackle some of the challenges Papadakis et al. introduced an attribute-agnostic blocking scheme [80,81,82] based on splitting attributes into primitive tokens. Since an attribute-agnostic blocking scheme allows entities to be placed in multiple blocks, the blocks are inevitably overlapping, which increases the
number of possible pairwise comparisons. This problem is partially addressed by the introduction of a framework for techniques designed to minimize the number of comparisons. They improved this approach by accommodating the Attribute Clustering Blocking and the Comparisons Scheduling in [81]. Both of these methods are used to achieve a balance between efficiency that depends on the number of pairwise comparisons, and effectiveness that expresses the number of detected pairs of matching entities. The main idea is that matching records likely have at least one token in common regardless of the attribute names, and how the record data is distributed across attributes. As in the homogenous case, records that contain the same value are placed in the same block. Only records belonging to the same block will be compared. Preprocessing of the blocks is then required to make the number of pairwise comparisons more manageable.

Similar to their previous work, Papadakis et al. in [82] concentrated on eliminating unnecessary blocks prior to or during the resolution process. The process of pruning is improved by using a graph where each node in the graph denotes an entity and edges connect entities that are compared in at least one block. Since entities can be assigned to multiple blocks the method skips adding an edge if it already exists. Edges between non-matching entities may also be eliminated. The pruning process further removes edges with low weights where the weights depend on the number of common tokens two records share. The assumption is made that the higher number of shared tokens, the higher the likelihood of two records being a match. The algorithm thus aims to retain only the \( k \) edges with the maximum weight. By eliminating redundant edges, the number of pairwise comparisons is decreased thus increasing the efficiency of the algorithm. Papadakis et al. focused on finding duplicates between two collections of records. Initially, a problem of developing
methods for finding duplicates among multiple data sets was discussed by Fellegi and Sunter in [36]. Their statistical model provided an algorithm for resolving pairs of records between two collections under the assumption that there are no equivalent records within each collection. It was the first formal model for entity resolution for multiple data sets.
Chapter 3
3 Modeling and Testing of Service-Oriented Applications

This chapter discusses the need for comprehensive testing methods for distributed systems in general, with application to service-oriented applications in particular. A typical service-oriented application employs a structure composed of logically separated operational layers often referred to as a multi-tier or n-tier architecture. This type of architecture provides a flexible and scalable solution since developers can work independently on each tier and can upgrade or replace individual components in response to changes in requirements.

Tiers communicate with each other to transfer data and provide required functionality. Figure 1 includes a presentation tier that renders user interfaces, a logic tier that processes business logic, and a data tier that maintains persistent data. In service-oriented applications, the logic tier contains 2 sub-tiers. Functionality is exposed through services in the service sub-tier, and implemented in the application sub-tier. To fulfill each service request, each tier must perform internal tasks as well as interacting with the other tiers, making it very difficult to test each application thoroughly, since these intra and inter-dependence relationships may cause failures that will only be encountered when a specific sequence of requests is made. To help counteract this, we present a model that depicts the dependence relationships within each tier as well as those across tiers. Our dependence model plays a crucial rule in testing service-oriented applications because it provides a way of creating test cases to cover scenarios missed by cases developed based on specification requirements documents. It is also used to guide the selection of viable execution paths that
encompass dependencies among sequences of requirements; such relationships are common sources of code-based defects.

Figure 1 A three-tier architecture

3.1 Modeling of Service-Oriented Applications

To examine the relationships between services and their execution context, we present a dependence model for service-oriented applications, depicting intra- and inter-tier dependence relationships between services (both internal and external) and other components.
3.1.1 Modeling intra-tier dependencies

In our previous research in testing and maintaining multi-tier web applications [62,111,124], we developed a multi-tier dependence model for a 3-tier architecture comprised of an intra-tier dependency model for each of the presentation, logic and data tiers, and programmatically-generated integrated and summary models of inter-tier dependencies [111]. The intra-tier models used representations specific to the nature of the tier: Web GUI trees for the presentation tier, System Dependence Graphs for the logic tier, and Data Object trees for the data tier. We now propose extending and specializing this dependence model for multi-tier service-oriented applications. In this model, the logic tier is divided into two tiers: a service tier and an application tier to represent services and other components in the application, respectively.

3.1.1.1 Presentation Tier

The presentation tier can consist of web applications that use HTML files, JavaServer Pages (JSPs), and/or PHP, each represented by a DOM-like GUI tree. A GUI tree consists of two types of nodes, document nodes and statement nodes, connected via three types of edges: control dependence, link dependence, and data dependence edges. A document node represents the structure of an HTML statement, and a statement node represents the embedded logic code in the server page. This tier can also include desktop applications that use technologies such as Java Swing and Visual Basic to implement Graphical User Interfaces.

Although they are not web pages, most GUIs follow a Model-View-Controller (MVC) architecture design and are event-driven, so they can be modeled using a DOM-
like GUI tree in which a node represents a GUI element such as a button, a text field, or a label, and an edge represents a call to the corresponding event handler, or a control or data dependence between two nodes. Figure 2 shows an example user interface used in our case study, and its corresponding GUI tree is partially shown in Figure 3, where the EditRecord function is called when the data of a record is entered and the Edit button is clicked.

The purpose of modeling the presentation tier is to facilitate the construction of test cases. Because user inputs and actions are registered in the presentation tier, with guidance from the dependence relationships it is possible to create sequences of user actions and operations that test the full scope of what a user can accomplish through the interface.

3.1.1.2 Application Sub-Tier

**Function Dependence Graphs.** The application tier can be implemented in many different languages, such as Java, C/C++, Perl, and C#. In our previous work [17] we proposed an object-oriented testing technique based on the function dependence relationship to improve the reliability of object-oriented programs.

We define: for functions $f_1, f_2, f_3$ **depends on** $f_2$ if and only if at least one of the following holds:

1. $f_1$ uses a variable $x$ that is defined in $f_2$,
2. $f_1$ calls $f_2$ and uses the return value of $f_2$,
3. $f_1$ is called by $f_2$ and uses a parameter $p$ that is defined in $f_2$,
4. $f_1$ uses a variable $x$ and $x$ uses a variable $y$ which is defined in $f_2$. 
Based on this relationship, we can construct a function dependence graph for the components in the application tier. A function dependence graph is represented as a directed graph \(<V, E>\) with each node in \(V\) representing a function (method), and each edge in \(E\) representing a dependence relationship between two functions. The nodes are connected via several types of edges including data, call and parameter in/out dependence edges.

3.1.1.3 Service Sub-Tier

The service sub-tier represents distributed functionality that is frequently implemented using industry-standard web services. In order to provide complex logic, applications provide compositions of services. *Orchestration* is used to specify the order in which services are called during execution. The orchestrated composition can be created in an implicit or an explicit way. Modeling explicit and implicit compositions employ the same concepts but the technical procedures vary. In explicitly defined compositions, the invocation order is often provided in a declarative language, whereas in implicitly defined compositions, the invocation order is defined in one of the procedural languages. Before discussing the service intra-tier dependence model, we provide comparisons between atomic versus composite web services, and explicit versus implicit compositions. Understanding, the nature of atomic, composite web services as well as different types of compositions is important since modeling strategies may vary from type to type.
3.1.1.3.1 Atomic/Composite Web Services

In order to provide required functionality, service providers cannot rely only on one single service. Often services need to be composed through service composition to achieve a specific goal. Thus, services can be classified as atomic and composite. Atomic web services do not depend on other services to perform required functions. Composite web services are a composition of several services. From the service consumer perspective, a composite web service may be considered a simple service, since only the service interface is visible to her/him. The interface specifies the operations and input/output parameters needed for invoking the service, but does not provide any details on the actual implementation of the service. The business logic is hidden from service consumers making calls to other services invisible.

3.1.1.3.2 Explicit Compositions

A composition is said to be explicitly defined when its implementation is defined separately from the components it invokes. Consider how composition works in WS-BPEL, for example, one of the most popular languages used to explicitly define service compositions. BPEL processes specify the order in which participating web services will be invoked as part of the composition. A typical BPEL process receives a request and in order to fulfill it, invokes one or more web services using steps called activities. BPEL provides both primitive and structured activities. Primitive activities represent basic constructs and are used for common tasks, such as invoke, receive, reply, assign, wait, and terminate. Structured activities are combinations of atomic activities, such as sequence, flow, switch, while, and pick.
BPEL provides synchronous and asynchronous processes. The client invokes a synchronous BPEL process on a port and waits to receive a response on the same port. As soon as the client receives a response from the BPEL process it continues with its operations. On the BPEL process side, the synchronous BPEL process receives a client request and sends back a reply to the same port on which the client is waiting. This type of process is used for processes that return a result in a relatively short period of time [118].

For an asynchronous process on the client side, the client invokes the asynchronous BPEL process and continues with its operations. The asynchronous BPEL process receives the request on one of the ports and sends back the reply by invoking the client on another port specified in the request. This type of process is used for business operations that occur over a span of hours or days [118].

BPEL also specifies how business processes interact with other web services. Web services are provided by business partners. To connect these partners, it is important to specify relationships between them. Every partner that is involved in the interaction with a BPEL process has to have a WSDL description file. PartnerLinkTypes are part of service specification and are thus placed in WSDL. They describe how the service partners interact and what they have to offer [118]. They also identify PortTypes, which specify collections of operations along with the input/output messages that their partners must support.

While business logic itself is described in BPEL process specification documentation, there are still other components needed to define service composition. In explicit compositions Web Services Description Language (WSDL) provides detailed information about the services, including their input and output parameters. Data types used by variables are defined in SML Schema Definition (XSD).
3.1.1.3.3 Implicit Compositions

Using a declarative language such as BPEL is not the only way to create a composition of web services. Many procedural programming languages like Java or .NET are capable of providing the functionality needed for the orchestration of web services. Java, for instance, can perform message exchanges, invoke web services, and delineate business logic. XML is supported by Java. The Java API for XML-based RPC (JAX-RPC) enables developers to create SOAP-based interoperable and portable web services. Axis is an XML based web service framework that creates a set of special Java classes capable of using (JAX) RPC to exchange WSDL messages. This type of composition is implicit since there is not a clear separation between the implementation of the composition and the components that it invokes. Orchestration is built using the same language as the components that it orders [49].

While business logic itself is described in procedural programming language there are still other components needed to define service composition. Similarly to the explicit compositions Web Services Description Language (WSDL) provides detailed information about the services, including their input and output parameters in the implicit compositions. Data types used by variables are defined in SML Schema Definition (XSD).

There exist two types of dependencies among services: an intra-service dependency, when operations included within a single service are dependent on each other,
and an inter-service dependency when operations belonging to two different services depend on each other. We propose an **Operation Dependence Graph** for modeling the intra-service dependencies and a **Service Dependence Graph** for modeling inter-service dependencies.

**Operation Dependence Graph**: The service tier defines a set of *provided services* normally described in WSDL files and implemented using languages such as Java, C/C++, and C#, regardless of whether service compositions are defined in an explicit or an implicit way. A service publishes a set of operations, and each operation is implemented by a web method. We use the concept of function dependence graph to parse the web methods, and construct an Operation Dependence Graph (ODG) based on the function dependence graph of the implementing web methods. The ODGs depict intra-service dependence relationships for each service. In some cases, especially for the external web services, for which the definitions are only available in a WSDL file, we construct the ODG by using the *def-use* associations of parameters defined in the operations. *Def-use* associations are those in which a parameter is considered *defined* if its state is changed by a function of an application and is considered *used* if its value is referenced by a function. Data dependence relationships can also be computed through the transitive closure of dependence relationships in the integrated dependence graph.

**Service Dependence Graph**: Dependence relationships may also exist between services; these dependencies may be introduced by the *def-use* associations between their parameters or by the function dependence relationships in the web methods that implement operations belonging to two different services. We construct Service Dependence Graphs to depict inter-service dependence relationships, where each node represents a service, and each
edge represents a dependence relationship between two services. Figure 3 illustrates the dependencies between the two operations: editing and searching. The EditRecord function calls the Transaction service, which calls the EditRecordData function in the application tier, and the EditRecordData function updates the record table. The search function calls the Search service, which then calls the GetRecordData function in the application tier, and the GetRecordData function gets records from the record table. Therefore, there is an inter-service dependency between the Transaction and the Search services, because the Transaction service defines records and the Search service uses records. In BPEL based applications, BPEL specification documents are analyzed to discover inter-service dependencies. On the other hand, if the service business logic is implemented in an implicit way analyzing classes with web method calls reveals inter-service dependencies.

In addition, a service may make requests to external web services provided by other organizations. In Figure 3, the external services are denoted by double circles. The signatures of the external services are defined in the WSDL files. We parse the WSDL files to identify call dependencies between the internal and the external services and use the transitive closure to identify data dependencies between the services. For example, a service $S_1$ calls an external service $ExtS_1$ and obtains information $x$ from $ExtS_1$ as a return value. The value of $x$ is then passed as a parameter to another internal service $S_2$; $S_2$ uses $x$ and passes $x$ to an external service $ExtS_2$. In this scenario, $ExtS_2$ is data dependent on $ExtS_1$, and we add a data dependence edge from the node $ExtS_2$ to the node $ExtS_1$. In Figure 3, the Transaction service calls the external Information Center service to get record data, and a Search service calls the Stolen Property service to verify if the record data belongs to the
stolen property. The Stolen Property service is data-dependent on the Information Center service because of the *def-use* association of record data.

3.1.1.4 Data Tier

**Data Object Trees:** The data tier is modeled based on the structural level, such as a database schema for database tables, or an XML schema for XML files. Each piece of data is represented by a data object and each database table or XML file is modeled by using a data object tree (DOT). A DOT consists of three types of nodes: a root node containing the identity of the object; a non-leaf node representing the complex type attributes of an object; and a leaf node representing a simple type attribute of an object. The nodes are connected via three types of edges: control dependence, link dependence and data dependence edges. Modeling these external resources is essential for shared-resource applications and for configuration-oriented applications. Many of the dependencies in the applications are induced by the data in the external resources, which cannot be identified without taking these external resources into account.

3.1.2 Modeling Inter-tier Dependences

In a typical service execution scenario, data are input from the presentation tier or extracted from external services. These data are carried by a service and processed by the service realization class, which may request other services and call an application
function to fulfill the original service.

Figure 2 An example of a Graphical User Interface

Figure 3 A partial graph of an integrated dependence graph

With these scenarios, there are interactions between presentation and service tiers, service and application tiers, and application and data tiers. To analyze the dataflow of
service-oriented applications, it is necessary to model the inter-tier dependencies based on the nature of the interactions among them.

**Step 1:** Integrating the presentation tier with the service tier: A call edge is used to connect a node in the presentation tier to a node in the service tier if the node handles an action and invokes an operation in a service. Parameter-in/out edges connect actual-in/out, and formal-in/out nodes.

**Step 2:** Integrating the service tier with the application tier: The interactions between the service tier and the application tier are made through function calls. To integrate these two tiers, we add call and parameter-in/out edges between the two tiers.

**Step 3:** Integrating the application tier with the data tier: The interactions between the application tier and the data tier are made via data passing, fetching and storing; thus the integration will focus on the dataflow between the two tiers. A piece of external data is considered **defined** if its state is changed by a function of an application and is considered **used** if its value is referenced by a function. A data dependence edge holds from a DOT node to a function node if the function uses the data. A data dependence edge holds from a function node to a DOT node if the function defines the data.

**Step 4:** Computing the transitive closure of dependences: After the inter-tier edges are connected, we use a forward slicing approach [46] to compute the transitive closure of dependences. We add a dependence edge to any two nodes if one is reachable from the other through a traversal of dependence edges.
3.2 Testing of Service-Oriented Applications

Service-oriented applications provide remote functionality to clients from both internal and external organizations and may acquire information from external services in order to do so. Since the successful performance of these applications depends upon so many different factors, any methods for testing of them must ensure both the reliability of the provided services and the dependability of the required services. This is difficult to achieve. First, a service provider should fully test every service so it can fulfill every request successfully. Although the design of the services strives for a high degree of loose coupling, there can still be inter-dependence relationships between operations or services. The inter-dependence relationship may cause failures that are only encountered when a specific sequence of requests is made. Because the uses of the provided services can be varied and are subject to user inputs, faults caused by these inter-dependencies can be difficult to detect without knowing the sequence of service requests.

Secondly, from the service consumer’s perspective, information received from the external services may be incorrect due to faults in the external services or misuse of the services. The faults that propagate to the applications can greatly increase the complexity and difficulties in testing due to consumers’ limited knowledge of the external services. The limited knowledge comes for that fact that information published in WSDL files normally contains the signatures of the operations only without implantation details. Existing approaches [6] use the coverage of the operations to determine the adequacy of the test suite. However, covering all operations may not be practical because not all the operations will be used by the applications; in many cases, only a small portion of them are relevant to the application. Moreover, even if an operation is covered with a test case,
it cannot be guaranteed that data received from the operation in the runtime is dependable. For example, in our case study, a search operation should have returned all the owners of a property, but the operation contained a fault and did not return the complete list. In testing, a property that has only one owner was used as the test input, and the fault was not detected, until in the runtime, a user searched for a property with multiple owners and then the incomplete data was discovered. Thus, it is not sufficient to cover only the operations; stronger criteria should be employed to ensure that information received from the external services can be dependable. The use of external services must be fully exercised before these services can be integrated with the application.

To tackle these new challenges, we developed an integrated testing strategy to ensure the reliability of provided services and the dependability of required services. To provide reliable services, each operation provided by a service and any possible sequences of requests should be tested. Nevertheless, testing all combinations can be too costly. A more affordable and effective way is to identify viable sequences that may potentially reveal faults. We adopt well-developed dataflow based testing strategies [37] to identify these sequences by analyzing dataflow between operations and services. If a piece of data is defined in an operation \( op_1 \) and used in another operation \( op_2 \), then there should be at least one test case that makes a request to \( op_1 \) followed by a request to \( op_2 \). This allows a confirmation that the piece of data is correctly defined in \( op_1 \) and correctly used in \( op_2 \). For example in our case study, in an update operation, the operation successfully updates the name of a user, and both original and updated names are associated with the record of the user. While the updated name is used in a search operation, the record of the user cannot be found because of a fault in the search operation that did not check the field of the updated
name in the record. This fault cannot be found if the two operations are tested separately, and the test of search operation does not use an updated name. Because the updated name is defined in the update operation and used in the search operation, the search operation is data dependent on the update operation. As a result, it is important to test these two operations together and following the correct order.

To test the dependability of the external services our approach is to test all the operations used by the application. For each operation we use equivalence classes to partition the input domain of the inputs and boundary value analysis to create test inputs. An operation is considered covered only if every equivalence class is tested. Furthermore, we test the sequence of operation invocations based on the dependencies between operations and services.

3.2.1 The Test Model

Our testing strategy combines the strengths of both black box and white box testing techniques and aims at covering all the requirements documented in the specification, missing requirements, and critical combinations of requirements. The proposed test model includes three major phases. In phase 1, software requirement specifications are used to create test cases covering all the requirements. The majority of companies perform this phase in integration testing. The second phase aims at testing services and functions that are not listed in the requirement specifications due to incompleteness. The process checks the nodes in the integrated dependence graphs that are not covered in phase 1, and creates additional test cases to cover them. Furthermore, most specification documents only state
individual requirements and do not provide information of the orders of combinations of the requirements. For example, if there are \( n \) input options, each of them invoking a service operation, then there can be up to \( 2^n \) possible input combinations. Exhaustively testing all the combinations is not practical and most likely not feasible. Therefore, in phase 3, we use the integrated dependence graph to guide the selection of viable execution paths that encompass sequences of requirements containing dependence relationships that are common sources for code-based defects. We investigate dependence edges that are not covered in either phase 1 or phase 2, and create test cases to exercise uncovered dependence edges combining a multiple number of requirements. For example, in Figure 3 there will be a test case exercising the Edit Button followed by an execution of the Search Button, because there is a data dependence edge between these two buttons. These test cases are necessary for revealing many types of faults that are difficult to detect by testing the requirements independently.

**Specification coverage testing:** In this phase, test cases are created to cover requirements documented in the specification. Software functional specifications describe the behavior of the system; they can be written in many different ways at various levels of formalism. Semi-formal use case modeling has been broadly adopted in a wide range of application domains. A *use case* is a description of a set of sequences of actions, including variants that a system performs to yield an observable result of value to an actor. To automatically generate test cases, we integrate use cases with the GUI trees in the dependence graph. Each action described in the use case is associated with the document node representing the action. For example, an action requiring the user to enter a login id will be associated with the login id node in the login GUI tree. With this association, test
inputs can be created according to GUI trees to cover all the use cases in the specification. In addition, the constraints of the inputs are denoted in the associated document nodes. Based on the constraints, equivalent classes of the input domains are constructed and boundary values are selected from each class to create test inputs e.g. consider an input option of the number from 1 to 99. To apply boundary value analysis, we would take the minimum and maximum (boundary) values (1 and 99 in this case) as test inputs. Also, we would test a value below lower limit e.g. 0 and a value greater than upper limit e.g. 100. We would have four test inputs. Furthermore, we would select a single value from range 1 to 99 as a valid test case. During each execution of the test cases, the nodes and edges traversed are marked as covered, and a node is considered covered only if all the constraints are tested. Note that for services provided to external users only, there is no information on the external GUIs, and test cases can only be created using the direct calls to the services.

**Node coverage testing:** After specification testing, we check the uncovered nodes in the integrated dependence graph and created test cases to cover these nodes. For external services, only the operations used by the application are tested. Therefore, only the operation nodes connected to the GUI tree are covered.

**Edge coverage testing:** After all the nodes are covered, additional test cases are created to exercise uncovered edges in the dependence graph.

Test cases for covering nodes and edges are generated by using the to-be-covered node/edge as the slicing criterion for the backward slicing of the dependence graph.
3.3 Evaluation

We have conducted two studies to investigate the validity and the effectiveness of our testing methodology on industrial systems. Two empirical studies were conducted on the systems developed and used by the New York State Police department. These systems are responsible for maintaining pistol permit records, collecting data, and retrieving crucial intelligence.

3.3.1 Case study 1

The aim of this study was to evaluate the effectiveness of our testing approach to providing reliable services. Our focus was to observe how many of the faults missed by our traditional specification based testing, could be detected by using the node and edge coverage of the dependence graph. The system used in this study contained 796 Java classes, 289,539 lines of code, 80 database tables, and provided six web services for both internal and external users. In addition, it called two external web services. The six web services were implemented to provide easier communication with software applications in other government agencies and law enforcement. This system uses a multi-tier architecture, in which the presentation tier is implemented in Java Swing (for internal uses) and is responsible for receiving requests and passing the requests to their corresponding web services. The web services in the service tier are defined as a set of WSDL interfaces implemented by Java classes. These web services call functions from the application tier, which are implemented in Java and are in charge of processing business logic and
interacting with databases in the data tier. Oracle 9i is used in the data tier to store persistent data.

The system was developed by using an Agile process, which delivers new releases to users frequently. We chose one of the releases for the first part of this study, where the (internal) users tested the system based on its specification and their experiences of using a previous version of the system. To apply our testing methodology, we first constructed the dependence graph of the system and evaluated the coverage of nodes and edges by the transactions executed by these users. We used the tool WebMTA [111] for Java-based web applications to construct the dependence graphs and instrument the applications to monitor test coverage. The presentation tier of this application is a desktop application, so we constructed the GUI trees and integrated them with the dependence graph manually. We further created additional transactions to cover all the nodes and edges in the graph, in which test sequences were generated by the tool, but the inputs were manually created. In the second part of the study, the system was deployed on Feb. 25, 2010 after its specification based testing. We did not have time to do analysis on code and edge coverage, or create additional test cases because of the pre-set deployment deadline. These tasks were performed after deployment for the purpose of investigating the effectiveness of the proposed strategy. This was done by comparing the faults detected in our after-deployment testing and those detected by the Quality Assurance (QA) team after users reported failures.

Table 1 shows the results of the first part of the study. In the specification-based testing, the users executed 1,760 transactions; 31 faults were detected in this phase. These transactions covered 89% of the nodes in the dependence graph. 83 more transactions were created to cover the remaining 11% of the nodes and detected eight more faults. Additional
102 transactions were created to cover 6% of the edges and detected 10 more faults. Among these faults, 6 of them were detected by using 59 transactions covering inter-service dependence edges. One fault was detected by 9 transactions covering intra-service dependence edges, and 3 faults were detected by 34 transactions covering service-application dependence edges. Using our methodology 18 additional faults were detected (58% more faults than using a traditional approach). The distribution of the faults is shown in Table 2.

Table 1 Results of fault detection

<table>
<thead>
<tr>
<th>Specification</th>
<th>Nodes</th>
<th>Edges</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>#T</td>
<td>#F</td>
</tr>
<tr>
<td></td>
<td>1760</td>
<td>31</td>
</tr>
</tbody>
</table>

Table 2 Results of fault distribution

<table>
<thead>
<tr>
<th></th>
<th>Inter-service</th>
<th>Intra-service</th>
<th>Service-application</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>#T</td>
<td>#F</td>
<td>#T</td>
</tr>
<tr>
<td></td>
<td>59</td>
<td>6</td>
<td>9</td>
</tr>
</tbody>
</table>

Table 3 shows the results of the after-deployment testing. 50 defects were detected by the QA team after the failures reported by the users over the past months. Among these defects, 33 were categorized as software faults and the remaining 17 were the requests for improvement or specification changes. The additional testing conducted by one of the authors used 254 transactions to cover the remaining 7% of nodes and 30 faults were detected. To cover the remaining 3% of edges, an additional 147 transactions were created and 12 faults were detected. The entire 33 user-reported faults were also detected by using
our approach. Among the 12 faults, 6 were detected by 86 transactions covering inter-service dependence edges, 2 were detected by using 24 transactions covering intra-service dependence edges, and 4 were detected by using 37 transactions covering service-transaction edges.

Table 3 Results of after-deployment fault detection

<table>
<thead>
<tr>
<th>Reported faults</th>
<th>Node Coverage</th>
<th>Edge Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>#F</td>
<td>#T</td>
</tr>
<tr>
<td>----------------</td>
<td>---</td>
<td>----</td>
</tr>
<tr>
<td>33</td>
<td>254</td>
<td>30</td>
</tr>
</tbody>
</table>

Table 4 After-deployment fault distribution

<table>
<thead>
<tr>
<th>Inter-service</th>
<th>Intra-service</th>
<th>Service-application</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>#T</td>
<td>#F</td>
</tr>
<tr>
<td>86</td>
<td>6</td>
<td>24</td>
</tr>
</tbody>
</table>

3.3.2 Case study 2

The purpose of this study was to investigate if our testing approach could effectively detect faults propagated from the external web services used by our system. For this system, testing was done with a focus on the external web service since the goal of the program was to build an interface with the external web service. The system used in this study allows users to retrieve data from a number of external data sources by using an external web service to access millions of records. It consists of 45 Java classes that contain approximately 11,000 lines of code. The New York State Police system communicates with the external web service using the SOAP messaging standard. The external web service provides 114 operations described in the provided Schema.xml. Most of them are Request-
Respond type, and 104 operations were used by our system. The results are displayed using JavaServer Faces technology.

During user acceptance testing, 36 test cases were created in Hewlett-Packard (HP) Quality Center, 30 faults were detected. These test cases are different from those in case study 1. Each test case recorded in the testing tool contains a sequence of actions, and each of the actions include a number of transactions. We used our tool to construct the dependence graph and test sequences, but did not use it to construct the GUI trees and the test inputs. We discovered that the 36 test cases covered about 92% of the total number of nodes. We used 21 test cases to cover the remaining nodes and edges and detected 16 more faults. 14 of the faults were discovered by covering the call edges between the application and the operations in the external web service. These faults were due to missing requirements and misuse of the service. 2 faults were detected by exercising the intra-service dependence edges in the service; these two faults were difficult to detect without using our dependence model. Table 5 shows the results of the testing.

<table>
<thead>
<tr>
<th>Specification</th>
<th>Node</th>
<th>Edges</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>#T</td>
<td>#F</td>
</tr>
<tr>
<td>36</td>
<td>31</td>
<td>8</td>
</tr>
</tbody>
</table>

We observed that covering 8% of nodes and 4% of edges increased the number of detected faults by 38% and 6%, respectively.
3.3.3 Discussion

From these two case studies, we learned that the provision of reliable services relies on understanding the interactions in the service-oriented applications and the dependence relationship between services, components, and databases. Although, full understanding of relationships requires the construction of dependence graphs with our tool support, the benefit of using these graphs to guide testing is much greater than its cost. In these studies, even though the GUIs, the test inputs, and other application-specific parts of the graphs were constructed manually, others were automatically generated by our tool. Thus, the proposed technique is scalable to industrial applications and significantly reduces maintenance costs, while also providing the benefit of detecting those difficult-to-reveal faults that were missed by the specification-based testing.
Chapter 4
4 Effect of Data Quality on the Reliability of Service-Oriented Applications

Service-oriented applications and web service technologies are one recent answer to inter-organizational integration challenges. With the emerging popularity of these types of applications, the reliability of web services, defined as the probability of failure-free software operation for a specified period of time in a specified environment, has become an important metric for determining quality of service. There are a number of models for estimating the reliability of traditional software in which code defects are considered the main causes of failures. Web services rely heavily on internal and external data sources to provide their functionality. A service request is considered to have failed if the service fails to deliver valid data to its customers. The causes of this type of failure can be code defects in software components or invalid data in internal or external data sources. Therefore, the quality of data sources is as critical as the quality of software components to the reliability of web services.

A number of reports have shown that poor quality data in data sources for crucial applications have a significant business impact [13,31,79,83,93]. Redman [93] reported the negative consequences of poor data quality, such as customer dissatisfaction, less effective decision making, and higher operational cost. Eckerson in [31] provided an overview of the most common sources of data defects and gave multiple examples of companies that have suffered losses due to invalid data. The Ohio Environmental Protection Agency reported [79] that an average of 5,000 data errors was being reported every month in their online Electronic Discharge Monitoring Report system.
Traditionally, data definitions have been closely associated with designated business applications; an organization’s data resources were not meant for global use by different applications across various organizations. Most web services, on the contrary, are designed to make data available to many internal and external businesses. This requirement increases the chance of unpredictable and invalid inputs, since users represent various levels of technical sophistication and are not limited to the organization's trained personnel. Consequently, the number of data quality problems in web services is likely to be higher than that in traditional software applications. These problems can introduce negative effects on the reliability of web services and thus service-oriented applications.

This chapter discusses how data quality can affect reliability performance. It provides an insight into a deployed, stable, resource-driven system in which the majority of implementation defects have been fixed. However, system reliability is still fluctuating because of dynamically changing resources, particularly data sources. This type of system utilizes data in persistent storage mechanisms such as Database Management Systems (DBMS) to provide services. Data are continuously updated, but some data may have been entered incorrectly. For these invalid data it is very likely that when used by the system to provide service, the system will produce unacceptable results. Our empirical observation shows that more than 16% of failures were caused by invalid data during our observation period. Existing approaches for reliability estimation do not consider failures caused by invalid data; thus they cannot provide accurate reliability estimates. There is an urgent need for a new way to measure service reliability that takes into account data quality.

To tackle this challenge, we first developed an approach to measure data quality, and then we modeled the interactions between services and data sources. Based on this
model, we computed the reliability of web services by using an architecture-based reliability model that integrates data quality with software component reliability.

Data used by service-oriented applications can be stored in a Database Management System, a file, or other means. The sources of data can be migrated from a legacy DBMS, received from external services, or entered by organizational personnel. In this study, we focus on data stored in a DBMS; however, the same approach can be applied to other types of data storage as well.

4.1 Modeling Data Quality

A web service interacts with either internal or external data sources to fulfill requests. To investigate how invalid data can affect the reliability of web services, it is necessary to examine the interactions between web services and both internal and external data sources.

Basic data operations have been described by Kilov in the CRUD (Create-Read-Update-Delete) model [50]. In [95] Saleh, Kulczycki, and Blake derived a generic data model from a set of models that encompass relational and object-oriented databases. In this generic model, a data source is considered a set of entities, and an entity is a set of records each of which has zero or more attributes. The generic model also includes functional descriptions of the basic data operations by adopting the CRUD (Create-Read-Update-Delete) model [50] to describe service-data interactions. A GenericDataModel class was used to formally specify data sources and their interactions with web services. This generic model provides a rigorous foundation for describing service-data interactions.
Data flows from web services (such as “create,” “update” and “delete”) to data sources; more importantly, it flows from the data sources to the web services and thus to the users (“read”) as well. The aim presented in this methodology is to measure the effect of data quality on the reliability of web services. Therefore, the focus is put on the quality of data flow from data sources to the users. In the following, we first provide a description of which data quality attributes are considered. We then show how to measure data quality and incorporate it into the architecture-based reliability model for measuring service reliability.

4.1.1 Data Quality Assessment

4.1.1.1 Data Quality Dimensions

Some studies have proposed models for data quality assessment [84,91,110]. Pipino et al. [84] derived techniques to develop usable data quality metrics. They discussed multiple quality attributes such as free-of-error, completeness, consistency, believability, timeliness, and accessibility. According to their approach measuring data quality involves both objective assessment and subjective assessment. Similar to this study, most existing techniques use multiple data quality dimensions and focus on accuracy, precision, reliability, currency, completeness, relevancy, and timeliness [4,47]. Wang and Strong [110] introduced several dimensions for measuring data quality yielding a final list of twenty dimensions that encapsulates 118 data quality attributes. The importance of determining the list of quality dimensions and attributes from collected customer’s responses instead of defining them theoretically based on researchers’ experience is the
main focus of the study. Data quality dimensions are defined as sets of data quality attributes that represent a single aspect or construct of data quality. In the other words, dimensions are sets of attributes of data quality such as relevancy, or completeness. In order to obtain a set of possible attributes, they collected data from data users. 179 attributes were identified as a result of the first survey. The importance of the attributes was captured in the second survey when data consumers were asked to rate importance of each attribute on a scale from 1 to 9, where 1 denoted extremely important and 9 denoted not important. In the second study, dimensions were grouped into categories, and then the correctness of the grouping was certified by data consumers. Finally, four categories of dimensions were defined: intrinsic data quality (DQ), contextual DQ, representational DQ and accessibility DQ. The intrinsic DQ consisted of dimensions such as accuracy, objectivity, believability and reputation; contextual DQ consisted of value-added, relevancy, timeliness, completeness and appropriate amount of data; representational DQ consisted of interpretability, easy of understanding, representational consistency and conciseness of representation; and accessibility DQ consisted of accessibility and access security. From the analysis of the obtained set of dimensions Wang et al. drew the conclusion that quite often Information Technology (IT) professionals have a different perspective of data than that presented by customers. IT professionals often consider the quality of their data being higher than the customers do. In [5] Batini et al. also discussed a number of dimensions including: accuracy, completeness, consistency, currency, timeliness etc.

In this study we focus on three dimensions; accuracy, completeness, and consistency, which are among the most common ones. We discuss these dimensions with regard to entities and records. The term entity refers to a real-world object such as a person,
an address or a vehicle. Entities are described by their properties, called attributes. **Records**

are the technical representation of the entities. Records may be database records/tuples that
consist of a set of attributes’ names, a set of values from entities and, possibly, a unique
identifier. A **data element** is defined to be a value associated with an attribute name in a
record.

Accuracy is defined as the closeness between a value from an entity and its
representation in the data stored i.e. a record. There are two types of inaccuracy, syntactic
and semantic inaccuracy. Syntactic errors are invalid values, whereas semantic errors are
valid values that are untrue. A misspelled data value is an example of syntactic inaccuracy,
while an incorrect value, for e.g., a wrong address, would be considered semantic
inaccuracy. Misspelled values can be viewed as a violation of a valid value constraint from
the family of domain constraints. This type of constraint may need to comply with specific
constraints defined by industry domain experts. For example, organizations may impose
terminology restrictions in order to meet vocabulary standards, such as constraining
medical terms to those in a standard healthcare dictionary, or limiting name spellings to
those specific to a culture. When verifying syntactic accuracy, for example, the name
‘Stven’ will be marked incorrect since it does not match any value from the list of valid
names. At the same time, when verifying semantic accuracy, the name ‘John’ may be
syntactically valid, but may be marked incorrect due to semantic inaccuracy if ‘John’ is not
the true name of the person whose record is being verified.

Human error is the main reason for syntactic inaccuracy – users tend to make
spelling errors unless they are forced to select a value from a drop down menu. On the
other hand, semantic inaccuracy can be caused by changes that have not been captured.
Entities often change: people move, or get married, cars change owners, and providers of telephone numbers may change. These changes often go unnoticed without all the necessary forms being filled out. As a result, data stores becomes outdated and thus inaccurate.

Completeness is defined as “the extent to which data are of sufficient breadth, depth, and scope for the task at hand” [110]. For the relational database model this definition may be understood in terms of the presence/absence of attribute values in the records. If the attribute is not mandatory, a null value may be assigned. Null is a special marker that has the general meaning of a missing value. If the attribute is mandatory, a nonnull value is expected to be assigned. Nevertheless, if data was migrated or converted from another source, even mandatory attributes may contain null values. Data purging is another common cause of data incompleteness. Data purging is a process of removing some of the old data to make way for more data. During this process some data that fits the purging criteria may be removed when it should be left alone. Lack of attribute constraints in place is another common reason for missing data. In order to save time and effort, users tend to avoid filling fields that are not mandatory. As a result, important information may end up being missing.

Consistency refers to conformance to semantic rules defined over data sources, such as integrity constraints. One example of a semantic integrity constraint is a condition that the budget in a department cannot be less than the sum of the salaries of the employees who work in that department. Format constraints, on the other hand, identify the expected form in which the attribute values are supposed to entered/stored, preventing inconsistent data entry. For example Social Security numbers must contain only numeric values and
have the format 111-11-1111. Modern databases allow enforcing of many integrity constraints. Nevertheless, changes in requirements, data migration, and human error as well as programming errors can introduce violations of constrains. The changes in requirements and data migrations make data especially prone to inconsistency. For example, over the years users may have been entering dates in a format that allowed two digits for a year value. When the year 2000 approached requirements may have changed to require four digits for a year value. In some cases, legacy data may not have been updated causing data format inconsistency. Thus, it is important to periodically verify whether existing data satisfies the above constraints.

Data accuracy is difficult to verify. Some existing techniques apply object identification strategies including probabilistic [36], empirical [12], and knowledge-based techniques [61,102].

In our approach we built a domain-specific data dictionary for organization-specific data, such as product name, product id, etc. For non-domain-specific data we applied existing tools using common knowledge-based ontologies such as YAGO [122] to assess the accuracy of each data element.

In order to measure completeness, mandatory attributes need to be identified for each relation in the data store. Homogenous data is often characterized by database schema, with the schema providing information about each attribute. Once the mandatory attributes are identified, necessary regular expressions can be defined and used to verify how many records are missing values for each attribute.
Similarly, to establish consistency we first analyzed data definition files such as database schema to identify constraints applicable to each attribute. Then, we scanned the database relations using pre-defined regular expressions to identify inconsistent records. There are many software tools available for checking data completeness and consistency that allow using customized regular expressions [98, 101].

4.1.1.2 Duplicate Records

Incompleteness and inconsistency may cause not only errors within data but also introduce duplicate records. In addition, human errors as well as merging processes may create multiple (duplicate) records for the same entity. Existence of duplicate records decreases quality of data since it causes discrepancy between data and entities it represents. This discrepancy is a reason of data inaccuracy.

4.1.2 Quality of Data Components

In order to compute quality of data components we need to establish data quality of records. Records are often database tuple that consist of attributes. A data element is defined to be a value associated with an attribute name in a database record. If any data element for a record is inaccurate, or incomplete, or violates one of its constraints then we set the record’s quality to 0, otherwise, it will be 1. To compute the quality of a database relation, we first compute the total number of records containing at least one invalid data element. Then the quality of each relation \{T_1,\ldots,T_n\} is computed using the following formula:
\[ Q(T_i) = 1 - \frac{f}{k} \]

*Equation 1*

where \( f \) denotes the total number of records containing at least one invalid data element and \( k \) denotes the total number of records in relation \( T_i \).

A data component consists of a dataset containing attributes from one or several database relations \( \{T_1, \ldots, T_n\} \). If a data component consists of a dataset from one database relation, its quality equals the quality of the relation. Otherwise, once the qualities of all the relations are calculated, the quality of the data component is calculated based on the architecture style of its interaction with the software component. Depending on the interactions with the software component, the quality of a data component can be computed as follows:

- **sequence style**: if a software component contains a sequence of calls to several different database relations, the quality of the corresponding data component equals:

\[
Q_{sequence} = \prod_{i=1}^{n} Q(T_i)
\]

*Equation 2*

where \( n \) denotes the total number of relations in the data component and \( Q(T_i) \) denotes the quality of the relation \( T_i \). Sequential calls are common among web services since services often perform a series of data manipulations in order to provide expected results. In this architectural style, each step depends on the results of the previous one. For example, in order to return invoice information a software component may first look up customer id in the Customer relation and then, using the retrieved id, may return results from the Invoice
relation. In this case, the results returned from the Invoice relation depend on the quality of the id retrieved from the Customer relation. Thus, the quality of returned data is dependent on the quality of data from both relations.

- **condition style:** if a software component contains a condition statement that allows accessing from data several relations, the quality of the corresponding data component equals:

\[
Q_{\text{condition}} = \sum_{i=1}^{n} P_i Q(T_i)
\]

*Equation 3*

where \( n \) denotes the total number of relations in the data component, \( Q(T_i) \) denotes the quality of the relation \( T_i \) and \( P_i \) denotes the probability of choosing certain condition value. Selecting a certain condition value \( i \) implies accessing data from the relation \( T_i \). Thus, the total quality of the data component is a sum of all possible combination of probabilities from the condition and the quality of the accessed relations. For example, in order to provide customer with the best hotel rate the condition first checks which hotel name was entered. Depending on the chosen hotel name, the service will provide information about available rooms.

- **parallel style:** If the software component contains a number of parallel (concurrent) calls to different relations, the quality of the corresponding data component equals:

\[
Q_{\text{parallel}} = 1 - \prod_{i=1}^{n} (1 - V(T_i))
\]

*Equation 4*
where \( n \) denotes the total number of relations in the database component and \( Q(T_i) \) denotes the quality of the relation \( T_i \). This architecture style allows services to retrieve/enter data from/to multiple relations simultaneously. For example, the software component based on the input criteria may simultaneously search multiple insurance providers to provide a customer with a list of possible options. The returned results depend on multiple relations but the results coming from different provider are simply combined and not dependent on each other.

- **loop style:** if a software component contains a loop that allows it to repeatedly call the same relation or set of relations, the quality of the corresponding database component equals:

\[
Q_{\text{loop}} = \prod_{l=1}^{I} V(set(T))
\]

*Equation 5*

where \( l \) denotes the expected number of loop iterations. For example, this type of operation may be used by a service that provides a summary of the online activities for all store customers for the last six months. Since the loop itself may contain different architectures of calls, it is necessary to compute the quality of the set of relations called within the loop \( Q(set(T)) \) first, using the equations described above.

In our methodology we first measured the quality of each database relation using *Equation 1*. Then, we defined all data components called by services and established their architecture styles. The architecture styles were obtained from BPEL descriptions, if they were available, or by parsing the source code and constructing the styles based on the
interactions among the services. Finally, we calculated the quality of each data component using Equations 2-5. Once we established the quality of all data components, we incorporate them into a Markov-based reliability model that will be described in the next section.

4.2 Reliability Modeling of Service-Oriented Applications

4.2.1 Reliability Model of Service-Oriented Applications

In our previous work [17] we presented an architecture-based reliability model taking into account different architecture styles, including batch-sequential, pipe-filter, call-and-return, and fault tolerance styles, and applied a Discrete Time Markov Chain to compute the reliability of a system with heterogeneous architecture styles. This model was designed for web based applications. In this approach, based on the reliabilities of the components, their transition occurrences and information about the architecture, we build a state model. The state model is then used to compute the reliability of the entire system.

The reliability of the components and their transition occurrences can be predicted by using the framework proposed by Cheung, Roshandel, Medidovic, and Golubchik [19], which utilizes information sources available early in the development lifecycle and applies simulation, analysis, stochastic modeling and a hidden Markov model to predict component reliability and transition occurrences at an early stage of component development. For components that have already been developed, we can apply techniques proposed by Hamlet, Mason, and Woit [42,43], which exploit information collected during testing to estimate reliability and transition occurrences.
In Section 4.1 we described how to compute the quality of data components based on the quality of data sources. We will need qualities of data components for our reliability model.

Our reliability model for service-oriented applications, which is based on this heterogeneous architecture model [108], supports two levels of reliability estimation. At the atomic service level, components are computation modules implementing the service and data modules that support persistent data. The architecture of the service can be determined at the design stage or obtained from the source code by using a reverse engineering approach. At the composite service level, a component represents an atomic service. The architecture can be obtained from BPEL descriptions, if they are available, or by parsing the source code and constructing the architecture based on the interactions among the services. With this information on the service architecture, component (computation modules) reliability, and transition occurrences, we can apply our architecture-based model to compute the reliability of the atomic service first. Then, we can use the atomic service reliabilities and the connection model obtained from BPEL descriptions, specifications, or source code to compute the reliability of composite services. Once that is completed we can calculate the reliability of the service-oriented application. The details of the modeling approach are as follows:

**Step 1:** Identify the architecture styles in a system, based on the design specification of a system or its source code. From the high-level design specification, interactions among data and software components can be examined and the architectural styles can be characterized from observations of the interactions among components.
Step 2: Transform the identified architectural styles into state models. Once the architectural styles in a system are identified, the components and their interactions in each style become a sub-graph of the overall architecture view. The details of this modeling technique can be found in [108].

Step 3: Integrate the state models into a global state model of the system, based on the overall system architecture. The integration simultaneously addresses different component interactions among associated state models.

Step 4: Construct the transition matrix $M$ based on the global state model of the system and compute the reliability of the system using the following equation:

$$R = (-1)^{k+1} R_k \frac{|E|}{|I - M|}$$

*Equation 6*

where $M$ is a $k \times k$ matrix and $R_k$ is the reliability of the exit state. $|I - M|$ is the determinant of matrix $(I - M)$, and $|E|$ is the determinant of the remaining matrix, excluding the last row and the first column of $(I - M)$. The differences between our model and existing models are (1) in our model the finer-grained approach considers the implementation of a service, and on the service level, client-side components can also be taken into account. (2) Our model can use source code to identify the architecture styles, which does not rely on the availability and correctness of BPEL. Moreover, it can guide the creation of a BPEL file for the service. (3) Our model addresses the commonly used call-and-return style which has not been addressed by the existing models. (4) Our Markov-based computation takes into account all types of iterations and does not require prior knowledge of the number of iterations. (5) Most importantly, we consider data sources used by the services as
4.3 Evaluation

4.3.1 Case Study 1

We conducted an empirical study on a service-oriented application developed and used by the New York State Police department. The aim of this study is to investigate the extent to which data quality affects the reliability performance of the system. This system is responsible for maintaining records on pistol permits and gun transactions within New York State. An older version of the system had been used by the New York State Police Department for many years. This system was re-designed and re-implemented recently using a service-oriented architecture to improve its usability and reliability, and interoperability between agencies. The new system was launched in May 2010. In addition to upgrading the code, the databases were also migrated from a legacy system to a new Database Management System. In this study, we focused on one of the services provided by the system named “transaction.” This service enables users to enter permit applications, permit amendments, renewal, dispositions, and dealers etc., using the operations published in its WSDL file. The operations were implemented by 13 Java components consisting of 300 classes. In addition, there are four components using the Hibernate framework to interact with six database relations stored in the Database Management System and eight components implementing client-side interfaces. To investigate the effect of data quality on the reliability of this service-oriented system, we first estimated the reliability of the system with and without data components, and then compared the estimates with the observed reliability. The usage of the system vary per user. A given transaction can be
repeated multiple numbers of times. There is no enforced order of transactions; therefore, we applied the architecture-based reliability model described in Section 4.2.2 to compute the reliability of the system. In step 1 we constructed the architecture view of the system depicted in Figure 4. (Step 1 from section 4.2.2) and identified three architecture styles. Components C_2, C_3, C_{18}, and C_{19} are in call-and-return style, where C_2 is a caller to C_3, and C_{18} is a caller to C_{19}. Components C_{17}, C_{18}, and C_{20} are executed in parallel and the other components are running sequentially. D_1, D_2, D_3, and D_4 are data components, where D_1 consists of four database relations T_1, T_2, T_3, and T_4, D_2 contains the database relation T_5, D_3 contains the database relation T_6, and D_4 contains database relation T_2. Moreover, four components, C_2, C_7, C_9, and C_{19}, interact with the data components, D_1, D_2, D_3, and D_4, respectively.

Figure 4 The architecture view of the system
The next 2 steps are not identified in the section 4.2 since they are part of the experimental study not the modeling phase, but they are described in section 4.1.

In step 2, we used a tool, QuickAssess [98] to measure the quality of the database relations. Quick Assess is a data quality assessment tool which consists of a set of field validators that can be applied to database attributes. The validators are written in Java in the form of regular expressions. The tool allows, for example, specifying a data format as MM-DD-YYYY or MM/DD/YY. Also, it allows specifying that the date should not be in the future or the year should not be less than 1900. The Quick Assess takes Excel or plain text files as input files. For each of the attributes within the input file the user specifies which validator should be used. Quick Assess requires specifying constraints per attribute, so we identified the applicable constraints for the data elements. The quality of a relation is computed as 1 minus the total number of invalid records over the total number of records in the relation, where a record is considered invalid if at least one of its entries (data elements) contains data with incorrect value, format, type, length, or constraint (Equation 1). The results of data quality are shown in Table 6.

<table>
<thead>
<tr>
<th>Relation</th>
<th>Total Number of Records</th>
<th>Number of Invalid Records</th>
<th>Quality</th>
</tr>
</thead>
<tbody>
<tr>
<td>T_1</td>
<td>676,406</td>
<td>2,502</td>
<td>0.9963</td>
</tr>
<tr>
<td>T_2</td>
<td>4,063,263</td>
<td>4,480</td>
<td>0.9989</td>
</tr>
<tr>
<td>T_3</td>
<td>2,550,537</td>
<td>240</td>
<td>0.9999</td>
</tr>
<tr>
<td>T_4</td>
<td>327,522</td>
<td>2,260</td>
<td>0.9931</td>
</tr>
<tr>
<td>T_5</td>
<td>279,801</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>T_6</td>
<td>289</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
In step 3 we computed the quality of the data components. $D_1$ consists of four database relations, $T_1$, $T_2$, $T_3$, and $T_4$, and each request sent from $C_2$ requires fetching data from each of the four relations. Therefore, the quality is computed as the product of the quality of the four relations. $D_2$, $D_3$, and $D_4$ consist of datasets from relations $T_5$, $T_6$, and $T_2$, respectively. Therefore, their reliabilities are equal to the quality of the corresponding database relations. The results are shown in Table 6.

The next step is not identified in the section 4.2 as a part of the modeling phase, but it is described in section 4.1.

In step 4, we measured the reliability of each software component. The reliability of a component was measured by using the methodology proposed by Hamlet [42]. We partitioned the components into 312 subdomains and created test cases to cover each subdomain. In addition, we applied a pair-wise approach to cover all combinations of two subdomains. Most of the test cases were created using JUnit and executed automatically using a collection of APIs called Fixtures for Easy Testing (FEST) that simulates actual users’ gestures. For each component, the reliability was computed as the total number of successful test executions over the total number of test executions, which are shown in Table 7, where $r_i$ denotes the reliability of component $C_i$.

In step 5, we constructed a state machine based on the architecture of the system. (Steps 2 and 3 from the section 4.2.2), Figure 5 shows the state view of the system. The reliability of each state was computed by using the reliabilities of the components executed in the state and their configurations, which is shown in Table 7, where $R_i$ denotes the reliability of state $s_i$. The transition probabilities between states were computed based on
an operational profile, which was computed from the audit logs of the older version of the system with some adjustments. The transactions were logged into four database relations. The entries in these relations contain transaction type, parameters involved in the transaction, time stamp and user name. Each of the components in our model is responsible for one transaction type. To calculate the transaction type occurrences we computed the total number of transactions from all four relations and calculated the number of transactions for each transaction type. Finally, based on the state model and transaction type occurrences we calculated the probabilities of state transactions [108]. The transition probabilities are listed in Table 7, where $P_{i,j}$ denotes the transition probability from State $s_i$ to State $s_j$. This information was then used to construct a transition matrix in step 6 (Step 4 from the section 4.2.2), shown in Figure 6, and by using Equation 6 we computed the reliability of the system, which is 0.986. To observe the extent to which the data quality affects the reliability estimate of the system, we also computed the reliability of the system excluding the four data components. The result showed that without taking into account data quality, the reliability of the system was computed as 0.997.

The next task was to compare these reliability estimates with the actual reliability of the system. We observed the uses of the system for three months. The result of each transaction executed during these three months was recorded in an audit log. The total number of user transactions recorded within the observation period was 253,595 and the total number of problems reported was 4,988. Among these failures, 4,164 transaction failures were caused by defects in the code, server failures, or incorrect exception handling, etc., and 824 transaction failures were caused by invalid data, shown in Table 8. These results show that 16.52% of failures were caused by invalid data, and suggest that
continuous data cleanup can improve the reliability of the system significantly. The observed reliability was computed by using the Nelson model [75], which computes the reliability as 1 minus the total number of failed transactions divided by the total number of transactions, and the result is 0.9803. This result shows that without considering data quality, the model overestimated the reliability by 0.017, whereas including the reliability of data components the overestimate was reduced to 0.0057, which is more than a 50% of improvement. Although the absolute difference is small, with large numbers of transactions, this small number can have a significant impact on the uses of the system. Thus, the results of our case study show that by considering data quality, the reliability modeling can provide more accurate estimates.
Table 7 Reliability and transition probability

<table>
<thead>
<tr>
<th>Component Reliability</th>
<th>State Reliability</th>
<th>State Transition Probabilities</th>
</tr>
</thead>
<tbody>
<tr>
<td>r1 = 1.000</td>
<td>R1 = 1.000</td>
<td>P1,2 = 1.000</td>
</tr>
<tr>
<td>r2 = 0.990</td>
<td>R2 = 0.990</td>
<td>P2,3 = 0.397</td>
</tr>
<tr>
<td>r3 = 1.000</td>
<td>R3 = 1.000</td>
<td>P2,4 = 0.008</td>
</tr>
<tr>
<td>r4 = 1.000</td>
<td>R4 = 1.000</td>
<td>P2,7 = 0.020</td>
</tr>
<tr>
<td>r5 = 0.993</td>
<td>R5 = 0.993</td>
<td>P2,8 = 0.020</td>
</tr>
<tr>
<td>r6 = 1.000</td>
<td>R6 = 1.000</td>
<td>P2,9 = 0.004</td>
</tr>
<tr>
<td>r7 = 1.000</td>
<td>R7 = 1.000</td>
<td>P2,10 = 0.020</td>
</tr>
<tr>
<td>r8 = 1.000</td>
<td>R8 = 1.000</td>
<td>P2,11 = 0.020</td>
</tr>
<tr>
<td>r9 = 1.000</td>
<td>R9 = 1.000</td>
<td>P2,12 = 0.002</td>
</tr>
<tr>
<td>r10 = 1.000</td>
<td>R10 = 1.000</td>
<td>P2,15 = 0.116</td>
</tr>
<tr>
<td>r11 = 1.000</td>
<td>R11 = 1.000</td>
<td>P2,16 = 0.388</td>
</tr>
<tr>
<td>r12 = 1.000</td>
<td>R12 = 1.000</td>
<td>P2,18 = 0.008</td>
</tr>
<tr>
<td>r13 = 0.667</td>
<td>R13 = 0.667</td>
<td>P3,2 = 1.000</td>
</tr>
<tr>
<td>r14 = 0.667</td>
<td>R14 = 0.667</td>
<td>P4,3 = 0.220</td>
</tr>
<tr>
<td>r15 = 1.000</td>
<td>R15 = 1.000</td>
<td>P4,6 = 0.780</td>
</tr>
<tr>
<td>r16 = 1.000</td>
<td>R16 = 1.000</td>
<td>P5,2 = 1.000</td>
</tr>
<tr>
<td>r17 = 1.000</td>
<td>R17 = 1.000</td>
<td>P6,2 = 1.000</td>
</tr>
<tr>
<td>r18 = 1.000</td>
<td>R18 = 1.000</td>
<td>P7,2 = 1.000</td>
</tr>
<tr>
<td>r19 = 0.952</td>
<td></td>
<td>P8,2 = 1.000</td>
</tr>
<tr>
<td>r20 = 1.000</td>
<td></td>
<td>P9,2 = 1.000</td>
</tr>
<tr>
<td>r21 = 1.000</td>
<td></td>
<td>P10,2 = 1.000</td>
</tr>
<tr>
<td>D1 = 0.9883</td>
<td></td>
<td>P11,2 = 1.000</td>
</tr>
<tr>
<td>D2 = 1.000</td>
<td></td>
<td>P12,13 = 0.15</td>
</tr>
<tr>
<td>D3 = 1.000</td>
<td></td>
<td>P12,14 = 0.850</td>
</tr>
<tr>
<td>D4 = 0.9989</td>
<td></td>
<td>P13,2 = 1.000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>P14,2 = 1.000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>P15,2 = 1.000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>P16,17 = 1.000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>P17,2 = 1.000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>P17,18 = 1.000</td>
</tr>
</tbody>
</table>
Table 8 List of invalid data

<table>
<thead>
<tr>
<th>Types</th>
<th>Description</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inaccuracy</td>
<td>Missing values</td>
<td>39.80%</td>
</tr>
<tr>
<td></td>
<td>Incorrect values</td>
<td>4.20%</td>
</tr>
<tr>
<td></td>
<td>Wrong length</td>
<td>12.50%</td>
</tr>
<tr>
<td></td>
<td>Incorrect types of values</td>
<td>4.20%</td>
</tr>
<tr>
<td></td>
<td>Incorrect address format</td>
<td>22.20%</td>
</tr>
<tr>
<td>Incompleteness</td>
<td>Missing values</td>
<td>14.30%</td>
</tr>
<tr>
<td>Inconsistency</td>
<td>Incorrect foreign key constraint</td>
<td>2.80%</td>
</tr>
<tr>
<td>0</td>
<td>( P_{1,2} )</td>
<td>0</td>
</tr>
<tr>
<td>-----</td>
<td>-------------</td>
<td>-----</td>
</tr>
<tr>
<td>0</td>
<td>( R_{1,2} )</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>( R_{1,3} )</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>( R_{1,4} )</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>( R_{1,5} )</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>( R_{1,6} )</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>( R_{1,7} )</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>( R_{1,8} )</td>
<td>0</td>
</tr>
</tbody>
</table>

**Figure 6 The transition matrix**
Chapter 5
5 Entity Resolution

Entity resolution (ER) is the process of identifying records that represent the same entity [16,32]. The term entity refers to a real-world object such as a person, an address or a vehicle. Entities are described by their properties, called attributes. Records are the technical representation of the entities. Records may be database records/tuples that consist of a set of attribute names, a set of values from entities and possibly a unique identifier.

Solving the ER problem, so that records that refer to the same entity are located and merged, can be viewed in various contexts: detecting all the duplicates within one collection of records, for example, or finding duplicate records between two collections under the assumption that there are no duplicate records within each collection. The latter is referred to as the Clean-Clean ER problem [80,81], where each collection is duplicate-free but the two collections overlap.

Most entity resolution algorithms are designed for homogenous data. Homogenous data is often characterized by a schema which formally describes its consistent structure. Assuming that a schema is available in advance, the consistent attributes are used to identify duplicate records.

Heterogeneous data, on the other hand, is characterized by a lack of standardization of data definitions and data structures. As a result, data sets may contain missing or inconsistently named data. For example, in a record, an attribute describing place of birth may be named “PLACE_OF_BIRTH”, while the analogous attribute name within another record may be called “birth_place”. Since attribute names may not match across records,
and data may be missing, the entity resolution process is more complicated than for homogenous data. In this study we focus on the Clean-Clean ER problem for heterogeneous data.

An exhaustive ER process has $O(n^2)$ time complexity [9,21,27,80,81,116] where $n$ denotes the number of records in the data collection, since in the worst case each record needs to be compared with every other record in the collection and often involves complicated logic for deciding whether records are matching. Various techniques [8,21,24,36,45,116] have been proposed to make ER process more efficient by dividing records into blocks and only comparing records within the same block to reduce the number of costly pairwise comparisons.

Nevertheless, processing all the pairwise comparisons within each block is still not feasible since it reduces time complexity to $O\left(\frac{n^2}{b}\right)$ where $n$ denotes the number of records in the data collection $b$ denotes the number of blocks.

With a growing amount of stored data, even after applying blocking techniques an entity resolution process often requires hundreds of millions of pairwise comparisons. Thus, the main objective of our study is to present an entity resolution algorithm that effectively processes the pairwise comparisons within blocks. Intuitively, the runtime savings for blocks becomes significant if we only resolve comparisons that are the most likely to contain matching records.

With this in mind, we propose an entity resolution algorithm that not only involves a blocking scheme but also contains a 2-stage comparison selection phase. First, inspired by existing approaches, we divide records into blocks using an attribute-agnostic blocking
technique [80,81,82]. In contrast with the existing approaches, however, we do not create blocks out of every identified token. In order to save time and space we provide a token selection process that eliminates tokens that are likely to create large blocks of records that do not have much in common. Since the token selection process is domain independent we cannot disregard the possibility of some large blocks being created. That is why, in the second step of our algorithm, we analyze the created blocks and eliminate those with the largest number of possible comparisons. In the third step, the 2-stage comparison selection technique is applied to the remaining blocks.

The 2-stage comparison selection phase is based on the idea of using two different sources of information to determine which comparisons are most likely to find duplicates. In the first stage, we use information pertaining to the records themselves, such as the number of tokens they have in common or the number of blocks they belong to. In the second stage, we use characteristics of the block itself, for example the frequency of the token that created the block. We use different metrics in each stage. In the first stage, we use a cheap similarity measure to group comparisons into three categories: those that are going to be resolved, those that are not going to be resolved since the likelihood that they compare matching records is too small; and those that need further analysis. In the second stage of selecting comparisons, we use a comparison utility measure to further analyze the comparisons from the third category. Once this stage is completed, we are left with a list of comparisons that will be resolved. We then perform a resolution process while merging matching records.
5.1 Motivational Example/Motivations

Figure 7 Two Collections of records

- **block b₁ (The):**
  - comp(p₂, q₁)
  - comp(p₂, q₂)

- **block b₂ (Ackerman):**
  - comp(p₂, q₂)

- **block b₃ (1984):**
  - comp(p₂, q₃)

- **block b₄ (unknown):**
  - comp(p₂, q₂)

Figure 8 A pruned list of blocks & comparisons with traditional methods
Consider two collections, $C_1$ and $C_2$, shown in Figure 7, each containing a set of records. Each record consists of a set of attributes comprised of a name (shown in bold) and a value. To process these records we build blocks using tokens, following Papadakis [80,81]. Each distinct token $t_i$ corresponds to a separate block $b_i$ containing all records originating from either collection $C_1$ and $C_2$ that contain $t_i$ in some attribute value - regardless of the associated attribute name. A token $t_i$ is contiguous string of characters contained in an attribute value existing in at least one record in collection $C_1$ and at least one record in collection $C_2$. Each block $b_i$ is a union of two inner blocks $b_{1i}$ and $b_{2i}$ with records in $b_{1i}$ originating from collection $C_1$ and records in $b_{2i}$ originating from collection $C_2$. $t_i$ needs to be shared by both input sets of attribute values, so that the resulting inner blocks are non-empty [82]. Existing approaches perform pairwise comparisons among corresponding inner blocks with the possible exclusion of blocks that are too large and those comparisons identified as unnecessary [80,81]. Figure 8 shows the possible list of the blocks and comparisons.

**Figure 9 A pruned list of blocks & comparisons with our novel token and comparisons selection methods**

<table>
<thead>
<tr>
<th>Block</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$b_1$ (Xanthippe):</td>
<td>$\text{comp}(p_1, q_1)$</td>
</tr>
<tr>
<td>$b_2$ (Ackerman):</td>
<td>$\text{comp}(p_2, q_2)$</td>
</tr>
<tr>
<td>$b_3$ (1984):</td>
<td>$\text{comp}(p_2, q_2)$</td>
</tr>
</tbody>
</table>
Our goal is to reduce the number of comparisons and increase number of identified duplicates. Thus, first we eliminate some of the blocks, then we achieve further reductions using comparison selection process. The comparison selection process also increases the number of detected duplicates.

Some blocks may be created using very common words such as “the,” “a,” “that” etc. To avoid creating large blocks with extremely common words that are likely to be of little value in grouping matching records, we exclude them from the set of tokens in the first place. If the token occurs in majority of records we consider it to be a *stopword*. In our experiments we considered occurrence in 80% of records as being a threshold for token elimination. We exclude all the *stopwords* from the list of tokens. We also exclude generic words such as “unknown”, or “unk,” used to indicate a lack of data rather than a definite value. As a result, we avoid creating a number of unnecessary blocks e.g. blocks $b_1$ and $b_4$ from Figure 9.

In addition most of the comparison selection methods are only based on the similarity measure of compared records. In that case comparison among records $p_1$ and $q_1$ are omitted in the traditional methods since the records do not have many tokens in common. On the other hand, our novel approach will consider this comparison since token “Xanthippe” is an uncommon token. We introduce a comparison utility measure that evaluates comparisons based on the information about blocks themselves not just records involved. The fact that “Xanthippe” is such an uncommon token is a good indication that records sharing it are very likely to match.

By the end of the comparison selection process, we are left with a list of comparisons containing possible matching records. By excluding blocks that present a low
probability of finding matches from the pool of blocks, $B$, and selecting comparisons using both information about records and tokens themselves we drastically decrease the number of pairwise comparisons required, improving efficiency and increase the number of detected duplicates, improving effectiveness.

### 5.2 Overview and Contributions

Focusing on the Clean-Clean ER problem, we present the Selective Comparison Algorithm that takes as an input two collections $C_1$ and $C_2$ of records and produces a high-precision list of duplicates with low computational cost. These collections are “clean” which implies each collection is duplicate-free but the two collections overlap. A record within the collections consists of a set of attributes comprised of a name and a value. We employ agnostic-attribute blocking [80,81,82] but we do not create blocks from all the identified tokens. Instead, we first perform lexical analysis and filter out tokens that too common (those that occur in 80% of records). Since even less common tokens may create large blocks, we remove blocks where the total number of comparisons is greater than a given threshold $\gamma$. Since resolving all the comparisons within the remaining blocks is still not feasible, we introduce the 2-stage comparison selection technique. The comparison selection process aims to identify record comparisons that are most likely to find duplicates. In the first step, for each pair of records in a comparison we use a Bloom filter technique to calculate a record similarity measure $S(p_i,q_j)$ to establish the degree of similarity among the two compared records. Based on the similarity measure, we split comparisons into three categories using two thresholds $T_1$ and $T_2$ where $T_1 > T_2$. All comparisons whose similarity measure is greater than threshold $T_1$ will be resolved in the
last step of the algorithm. Comparisons with similarity measure less than $T_2$ are considered to be useless and comparisons with measure greater than $T_2$ and less than $T_1$ will be further analyzed in the stage 2 of the process. In the second stage we compute comparison utility measure(s) for the comparisons that were identified as requiring further analysis. Comparisons with the utility measure $U(comp(p_i,q_j),t_i)$ greater than a given threshold $\mu$ will be resolved in the last step of the algorithm. $t_i$ denotes a token that was used to create block $b_i$ that $comp(p_i,q_j)$ belongs to. Comparisons with the utility measure $U(comp(p_i,q_j),t_i)$ less than $\mu$ will not be resolved. In the last step of our algorithm, we apply a match function to each comparison from the list. The match function returns true if the records are duplicates and false otherwise. Duplicate records are merged to form the final result.

**Our main contributions are:**

1. We present a new entity resolution algorithm that significantly reduces the number of pairwise comparisons needed to detect duplicates and increases the number of detected duplicates.

2. We present 2-stage comparisons selection process that helps to identify comparisons that are most likely to contain matching records.

3. We improve existing blocking techniques by employing Information Retrieval (IR) techniques - such as lexical analysis of tokens and filtering out *stopwords* from the list of tokens. As a result, we avoid creating blocks that can group records that do not have much in common.

4. We introduce a Bloom filter based structure to compute a similarity measure for records, and to track already discovered duplicates. We show that Bloom filters
structures are suitable for efficient processing of comparisons since their adding
and querying operations have computational complexity $O(1)$ and they use very
limited storage space since they avoid storing any actual information included
in records.

5. We conduct experiments on two real data collections to demonstrate the
effectiveness and effectiveness of our approach and compare it with the existing
ER algorithms from [80,81].

We have implemented our algorithms using an attribute-agnostic blocking scheme
[80, 81, 82]. Nevertheless, our comparison selection method is not bound to any specific
record blocking techniques. It can be easily incorporated into more traditional homogenous
blocking techniques since it aims to complement existing entity resolution techniques, not
to replace them. In case of homogeneous data, the different records’ similarity measure and
comparison utility measure can be used.

5.3 Problem Definition

Definition 1.1

A match function $m(p_i, q_j)$ is a Boolean method that takes as an input a pair of
records $p_i, q_j$ and returns true if the records are duplicates and false otherwise.

Definition 1.2

Two records are considered to be matching (are duplicates) if a match function
takes them as an input and returns true.
Definition 1.3

A comparison $comp(p_i, q_j)$ is said to be processed/resolved if a match function is applied to the records $p_i, q_j$ to determine if they are duplicates.

Definition 1.4

A similarity measure $S(p_i, q_j)$ is a method that, when applied to a pair of records $p_i, q_j$, returns the degree of their similarity. It returns values between $[0,1]$.

Definition 1.5

Comparison utility measure is a method $U(comp(p_i, q_j), t_l)$ that, when applied to comparison $comp(p_i, q_j)$ and token $t_l$, returns the degree of comparison’s usefulness.

Definition 1.6

Let $X$ denote the set of true matching pairs while $Y$ denotes the set of pairs identified as matching. Then the precision

$$Pr = \frac{|X \cap Y|}{|Y|}$$

while the recall

$$Re = \frac{|X \cap Y|}{|X|}$$

and

$$F\text{-measure} = \frac{2 \cdot Pr \cdot Re}{Pr + Re}$$

Equation 7
Problem statement

Given two duplicate-free (Clean-Clean) collections of records $C_1$ with $n$ number of records and $C_2$ with $m$ number of records and value $\tau \in [0,1]$, find an efficient algorithm that outputs a set of matching pairs of records and whose F-measure $\geq \tau$ and has time complexity much lower than $O(n \cdot m)$ in practice.

5.4 Selective Comparison Algorithm

In this section we discuss all the steps of our entity resolution algorithm that is used to identify and merge matching records among two data collections $C_1$ and $C_2$. The main goal of the framework is to perform entity resolution with a minimal number of pairwise comparisons while detecting as many duplicates as possible. Resolving pairwise comparisons is an expensive process since it requires a match function to be applied to each pair of records. Match functions are often complicated, and usually require computationally intensive methods such as string similarity measures [20,51,52,74]. By using a blocking technique and introducing the 2-stage comparison selection process we select the comparisons that are most likely to identity matching records. Thus, we decrease the number of pairwise comparisons and increase efficiency. We also want to make sure that our algorithm is effective, meaning that it discovers a high percentage of duplicate records. In order to insure this, we only eliminate a well-established number of blocks and comparisons and resolve comparisons that are the most likely to contain duplicates. We do not focus on improving effectiveness in the sense of improving a match function (oracle) since we assume it correctly determines if records are/are not matching.

In order to provide a complete description of our algorithm we discuss each step in a separate sub-section (sub-sections: 5.4.1 – 5.4.5).
5.4.1 Token Selection

Attribute-agnostic blocking requires a list of tokens $T=\{t_i, \ldots, t_n\}$ that are used to group records. The list of tokens $T$ is created from all the records’ attributes values that exist in at least one record in collection $C_1$ and one record in collection $C_2$. This assumption is weak and may result in the creation of a large number of tokens. Since each token $t_i$ is a base for a block $b_i$, a large number of blocks may therefore be built. We propose adding token selection criteria that decrease the number of blocks being created. We also propose performing text operations on the list of tokens to make them more consistent. The two processes we suggest are: lexical analysis, elimination of stopwords.

Lexical analysis includes: removing hyphens, punctuation marks, extra spaces, and unifying the case of letters.

In the elimination of stopwords process, stopwords are defined as those tokens that occur the majority of records (in our experiment in at least 80% of records). This idea is an extension of the stopwords concept from the Information Retrieval field [2]. Words such as articles, prepositions and conjunctions have little meaning by themselves and may lead to grouping records that are unrelated thus introducing noise into the entity resolution task. One way to reduce the noise is to reduce the set of tokens that are used to create blocks. Rather than maintain a list of common stopwords, we allow the datasets to tell us which words to use by excluding words which are too frequent among the entities from the set of tokens. There are two benefits of eliminating stopwords from the set of tokens. First, stopwords elimination reduces the number of blocks needed to be processed. Second, it reduces data storage requirements considerably. Also, we propose improving the efficiency
of blocking without influencing its effectiveness by removing generic words such as “unknown”, or “unk” from the list of tokens. These words are used as placeholders for blank values and do not introduce any grouping value.

5.4.2 Building Blocks

The goal of blocking is to split a data collection into blocks such that records belonging to the same block are likely to match. Records are placed in the same block $b_i$ if they share the token $t_i$ [80,81]. Since there are two collections of data $C_1$ and $C_2$, each block $b_i$ contains records originating from collection $C_1$ or $C_2$. Thus, each block $b_i$ is a union of two inner blocks $b_{1i}$ and $b_{2i}$ with records in $b_{1i}$ originating from collection $C_1$ and records in $b_{2i}$ originating from collection $C_2$. The blocks can be implemented using a record level inverted index which contains a list of references to records for each token $t_i$ since each token $t_i$ is used to create a block $b_i$. The index will contain references to records coming from both inner blocks $b_{1i}$ and $b_{2i}$ for a token $t_i$. This type of implementation allows establishing the sizes of blocks easily. It is also very helpful when retrieving list of records assigned to a block.

5.4.3 Block Filtering Process

Regardless of token selection scheme, there may well be some large blocks created. Some of the block tokens may not be considered stopwords since they do not occur on majority of records, but they may still may be too common to be useful for comparisons. We propose a selection process that will eliminate some of the blocks from $B$, while losing as few potentially useful blocks as possible. Our selection process is based on the total
number comparisons blocks introduce. The total number of comparisons \( c(b_i) \) of a block \( b_i \) equals to \( |b_{1i}| \cdot |b_{2i}| \) where \( |b_{1i}| \) indicates the cardinality for inner block \( b_{1i} \) and \( |b_{2i}| \) indicates the cardinality of blocks \( b_{2i} \). The total number of comparisons is a good indication of the workload block \( b_i \) introduces. We suggest computing the average number of comparisons in blocks \( B \) denoted as \( \text{avg}(B) \) and keeping the block \( b_i \) if

\[
c(b_i) < \alpha \text{avg}(B) \text{ where } \alpha \in \mathbb{R}^+
\]

*Equation 8*

The value of \( \alpha \) is established experimentally. The cost-benefit tradeoff needs to be considered when choosing \( \alpha \). The lower the value of \( \alpha \), the larger the number of blocks that will be discarded. The large number of eliminated blocks increases efficiency since it decreases number of pairwise comparisons. At the same time the effectiveness of the algorithm may be effected since some matching records may be inadvertently discarded.

### 5.4.4 Comparisons Selection Process

In this study, we used an attribute-agnostic token blocking scheme introduced by Papadakis [80,81,82]. This approach is powerful but it has some drawbacks. First, the large number of tokens will create a large number of blocks. In order to resolve these blocks, an overwhelming number of pairwise comparisons may be needed. Second, each record may be placed in multiple blocks. Thus, blocks may overlap, introducing redundancy and increasing the number of necessary pairwise comparisons. In order to reduce the number of required comparisons we propose the 2-stage comparison selection process.
5.4.4.1 Stage One

This stage of comparison selection process will be performed using a cheap similarity measure. If the similarity measure of two records \( p_i, q_j \) from \( \text{comp}(p_i, q_j) \) is greater than a given threshold \( T_1 \) the comparison \( \text{comp}(p_i, q_j) \) will be resolved. If the similarity measure is less than \( T_1 \) but greater than \( T_2 \) the comparison will be analyzed further during stage two of the comparison selection method. Otherwise if the similarity measure of records \( p_i, q_j \) is less than \( T_2 \) the comparison \( \text{comp}(p_i, q_j) \) will be discarded. Several different cheap similarity measures are discussed in [8,36,52]. It appears that using Jaccard similarity measure based on the number of common tokens between records \( p_i, q_j \) produces some of the best results. The number of common blocks shared between a pair of records \( p_i, q_j \) is proportional to their similarity. This implies a likelihood that they are matching.

To calculate the number of common tokens the index may be used. In that case the forward index stores a list of blocks for each records. The time needed for the calculation is \( O(\max \{ n \log n, m \log m \}) \) where \( n \) denotes the number of tokens assigned to the records \( p_i \) and \( m \) denotes the number of tokens assigned to the records \( q_j \) – this is the time needed to sort both list of blocks. The time to actually calculate the number of common tokens is \( O(n+m) \). In order to make this process more efficient we propose using Bloom filters.

Bloom filters were described by B. Bloom in 1970 [14] and are widely used in distributed network services, spell checkers, and differential file updating. They are used to solve the membership problem of a set since they can determine efficiently if an element is a part of a set \( A \). Bloom filters are implemented using a vector (a filter) with \( m \) bits. Initially, all bits are set to 0. For each element \( a \in A \), \( k \) hash functions are used to set bits at
position $h_1(a), h_2(a), \ldots, h_k(a)$ to 1 in the vector. In order to determine whether an element $b$ is a member of $A$, the $h_1(b), h_2(b), \ldots, h_k(b)$ are checked. If one of them is 0 then $b$ is definitely not part of the set $A$. If all of them are set to 1, $b$ is considered a potential member of the set. It is only a potential member since a Bloom filter may return true for elements that are not actually in the set (false-positives). This happens, since errors can occur when two or more elements are mapped to the same position in the vector.

In our context, we maintain a Bloom filter (a vector) to represent a record $p_i$ from an inner block $b_{1i}$ from the collection $C_1$. Initially all bits in the vector are set to 0 (Steps 4-6 of Algorithm 1). For each token $t_x$ from records $p_i$ we use $k$ hash functions to set bits at position $h_1(t_x), h_2(t_x), \ldots, h_k(t_x)$ to 1 in the vector. (Steps 7-11) Then, we determine if tokens from a record $q_j$ are members of the list of tokens of record $p_i$. (Steps 13-15) Function check ($t_y$) returns true if and only if the $h_1(t_y), h_2(t_y), \ldots, h_k(t_y)$ are all set to 1. The estimated count of tokens shared among two records in increased every time method check returns true. (Step 14).
Figure 10 Bloom Filter Algorithm

5.4.4.2 Stage Two

In the second stage we propose an additional evaluation on comparisons that have
similar similarity measure greater than $T_2$ but less than $T_1$. In this stage, the focus is on the
information about the set of blocks $b_i$ that contain a comparison $\text{comp}(p_i, q_j)$. We focus on
length and frequency of the token $t_i$ that created block $b_i$. This is important since it is
possible that two records $p_i, q_j$ may not have many tokens in common but, for example, they
share the same unusual token “zookeeper,” and thus there is a greater chance that they are
duplicates than two records that share “Smith”. Also, in general, the greater the length of
the token, the greater the chance of two records referring to the same thing. If two records
share the same token “Galifianakis,” for example, they more probably refer to the same actor than two records sharing the same “john” token. Therefore, we calculate two utility measures for each comparison \( \text{comp}(p_i,q_j) \). Let \( b_i \) be a block containing the comparison \( \text{comp}(p_i,q_j) \). The utility measure \( U_1(\text{comp}(p_i,q_j), t_i) \) is based on frequency of the token \( t_i \) that created \( b_i \). This can be easily calculated for each comparison using the inverted index from section 5.4.2. We suggest computing the average frequency of tokens \( T=\{t_i,..,t_n\} \), denoted as \( \text{avgFreq}(T) \). Comparisons that have utility measure:

\[
U_1(\text{comp}(p_i,q_j), t_i) = \lfloor \delta \cdot \text{avgFreq}(T) \rfloor \text{ where } \delta \in \mathbb{R}^+
\]

*Equation 9*

will be resolved. The value of \( \delta \) is established experimentally.

The second comparisons utility measure \( U_2(\text{comp}(p_i,q_j), t_i) \) is based on the length of the token \( t_i \) that created \( b_i \). We suggest computing the average length of the tokens \( T=\{t_i,..,t_n\} \) and denoted as \( \text{avgLen}(T) \). Comparisons that have their utility measure

\[
U_2(\text{comp}(p_i,q_j), t_i) > \zeta \cdot \text{avgLen}(T) \text{ where } \zeta \in \mathbb{R}^+
\]

*Equation 10*

will be resolved. The value of \( \zeta \) is established experimentally.

At the end of this stage, some comparisons that were initially classified as needing more analysis will be added to the list of comparisons that will be resolved.

Since we are dealing with a Clean-Clean ER problem we need to compare each pair of records \( p_i, q_j \) only once. The blocking technique that we are using introduces a possibility of \( \text{comp}(p_i,q_j) \) existing in multiple blocks. To avoid processing the same comparison more
than once we propose using a Bloom filter. Similarly to the Algorithm 1 we will represent tokens of \( p_i \) as a Bloom filter. We will use only one hash function that maps the index of the token to the index in the Bloom filter vector. Then we will hash tokens of \( q_i \). Every time check(\( t_y \)) returns true we check if the index of Bloom filtered \( t_i \) is greater than index of \( t_y \). If at any point index of Bloom filtered \( t_i \) is greater than index of \( t_y \) we discard the comparison from block \( b_i \) knowing there is another block \( b_j \) with a lower index that already contains comparison \( \text{comp}(p_i, q_j) \). Otherwise, we will continue until we check all the tokens from \( q_i \) and then conclude that \( b_i \) is actually the first block that contains comparison \( \text{comp}(p_i, q_j) \). Thus, we will keep the comparison to be resolved in the next step of the algorithm.

### 5.5 Resolution Process and Merging Records

As mentioned in the above section, we assume the existence of a match function that correctly determines whether two records are/are not matching. Once we have a list of comparisons to be resolved, we start applying the match function to one comparison at a time. Inspired by positive results reported by Papadakis we order comparisons before applying the match function using comparison scheduling method described in [81]. If the match function returns true we consider the two records to be duplicates and merge the records. Merged records can be then used in the following phases of the entity resolution process. Merging of records for homogeneous data has been discussed in [9,30,116]. The merging process (also called enriching the references) implies that after merging the references \( p_i \) and \( q_j \), all the attributes of \( p_i \) can also be considered as belonging to \( q_j \). If attribute names are the same but attributes values are different we propose combining
attribute values. For example, if \( p_i \) has a zip code 12304 and \( q_j \) has a zip code 12309, after records are merged to form \( r_{ij} \), both of the zip codes will be considered in future steps. If attribute names are the same and attributes values are the same we propose merging the attribute. Merging records in heterogeneous data is more complicated since not only attributes values but also attributes names may differ among two records. When merging two records \( p_i \) and \( q_j \), we propose keeping all the distinct attributes values and names for all non-shared token values. Records with common token values \( t_1, \ldots, t_n \) share the same attribute values (tokens) but may have different attribute names. In this case, we suggest keeping the distinct attribute names for common tokens. Due to the nature of the CleanER problem, each record has at most one duplicate. Thus, records that are compared and identified as duplicates do not need to be compared again. Duplicates are merged in the block in which they are identified, and any other comparisons containing either of the two duplicate records are discarded. We use a Bloom filter to keep track of all the records discovered so far. As mentioned above, Bloom filters are well designed for establishing the membership of an element in as set. The Bloom filter \( BD \) represents the records that have been identified as duplicates. When the algorithm discovers that two records \( p_i, q_j \) are duplicates, their indices are hashed using one/multiple hash function \( 1,2,\ldots,k \), and the bits at positions \( h_1(p_i), h_2(p_i), \ldots, h_k(p_i), h_1(q_j), h_2(q_j), \ldots, h_k(q_j) \) are set to 1 in the vector. Before we resolve a pair of records \( p_k \) and \( q_l \), we check whether they are members of the Bloom filter vector, \( BD \). If neither \( p_k \) nor \( q_l \) have been identified as duplicates, is a member of \( BD \) we resolve the comparison and merge the records, otherwise we skip the comparison. In the end of this process we will end up with a list of merged records.
5.6 Evaluation

We conducted an empirical study on 2 pairs of datasets: Internet Movie Database (IMDB) and DBPedia movies, and 2 versions of DBPedia [80,81,82]. IMDB and DBPedia are real world, large heterogeneous datasets. IMDB is a popular source for movie, TV and celebrity content. DBPedia extracts structured information from Wikipedia and makes this information available on the World Wide Web. The English version of DBPedia describes 4.0 million things including movies. Each of these sets contains a collection of movies. The aim of this study was to investigate the extent to which our algorithm can decrease the number of comparisons while achieving high effectiveness. We focused on evaluating the performance of the 2-stage comparison selection process.

In the first part of the study we focused on the 27,615 movie records coming from DBpedia and the 23,182 movie records coming from IMBD. As shown in Table 9 these two collections have 22,405 duplicates. Following Papadakis [80,81,82], we used the “imdbid” attribute included in all records as a match function (oracle) that determines which records are duplicates.

Table 9 Data used in the first part of the case study

<table>
<thead>
<tr>
<th>D_{movies}</th>
<th>DBPedia</th>
<th>IMDB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Entities</td>
<td>27,615</td>
<td>23,182</td>
</tr>
<tr>
<td>Duplicates</td>
<td>22,405</td>
<td></td>
</tr>
</tbody>
</table>
In the second part of the study we focused on the 1,190,734 movie records coming from DBpedia\textsubscript{1} and 2,164,058 movie records coming from DBpedia\textsubscript{2}. DBpedia\textsubscript{1} contains data from October 2007 whereas DBpedia\textsubscript{2} contains data from October 2009.

*Table 10 Data used in the second part of the case study*

<table>
<thead>
<tr>
<th>$D\textsubscript{infoboxes}$</th>
<th>$DBPedia\textsubscript{1}$</th>
<th>$DBPedia\textsubscript{2}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Entities</td>
<td>1,190,734</td>
<td>2,164,058</td>
</tr>
<tr>
<td>Duplicates</td>
<td>892,586</td>
<td></td>
</tr>
</tbody>
</table>

As shown in Table 10 these two collections have 892,686 duplicates. As before, we used the “imdbid” attribute in our match function.

We implemented our experiments using Java 1.6, using a Windows-based machine with 8GB of RAM and a 2.3 GHz processor for the first part of our study and a Linux machine with 24 GB of memory for the second part of our study.

Drawing on the work of [80,81,82,112,115] we assumed the existence of a match function (oracle) that correctly identifies duplicates. We then used five measures to evaluate the performance of our approach.

1. Precision, defined as the percentage of identified duplicates that are actual duplicates. [30,115]. Similar to [11,80,81] we did not need to pay particular attention to precision since all the matches found by our algorithm are duplicates due to a use of oracle.

2. Recall, defined as the percentage of correct duplicates that were identified.
3. The F-measure [112,116], defined as the harmonic mean of precision and recall.

4. The Reduction Ratio [8], defined as $RR = 1 - \frac{cd}{N}$ where $cd$ denotes the number of comparisons required by a new algorithm, and $N$ denotes the number of comparisons required by a baseline algorithm.

5. The comparisons-to-duplicates ratio, which measures computational efficiency.

One purpose of the block filtering process is to eliminate oversized blocks that are likely to incur a large number of comparisons. Both parts of the study show a pattern of a small number of blocks being oversized and suitable for elimination with the majority requiring further analysis. For the block filtering process we discarded about 10% of blocks in the first part of the study and about 1% of blocks in the second part of the study.

Our 2-stage comparison selection method further reduced the number of comparisons by eliminating comparisons with a small number of common blocks, and by using information about token length and frequency to ensure that unusual comparisons were preserved for the final resolution process. Also, we discovered more duplicates since we kept comparisons that are the most likely to contain duplicates.

For the first step of the 2-stage comparison selection method we used Jaccard similarity measure based on the number of common tokens between records $p_i, q_j$ as a similarity measure. We used Bloom filters to compute the number of common tokens between two records. Also in this step we perform a check using Bloom filters to avoid comparing the same records twice. We used Entities Similarity described by [81] to compute the thresholds to identify those comparisons that are going to be resolved, those
that are not going to be resolved since the likelihood that they compare duplicates is too small; and those that that need further analysis in the second step on the selection process.

In the second stage of the comparison selection process we filtered the comparisons based on the length and frequency of a token they shared. We preserved comparisons with very low token frequency and significant token length for the final resolution process.

Using this method, in the first part of the study we were able to discover 96% of duplicates, reducing the number of comparisons by 99% relative to the baseline (brute force Table 12), while maintaining a recall of 0.96. We needed only 2.40 comparisons on average to discover each duplicate. We also needed only 0.22 minutes to complete the process. The time improvements were due two factors: first our block filtering process eliminates oversized redundant blocks, second Bloom filters speed up operations needed for each comparison. Table 11 contains our results.

With the 2-stage comparison selection method in the second part of the study we were able to discover 94% of duplicates while reducing the number of comparisons by 99% relative to the baseline (brute force Table 14), with a recall of 0.94. We needed 32.53 comparisons to discover each duplicate. Table 13 contains our results.

Tables 15 and 16 compare our results with those based on the Efficiency Workflow presented by Papadakis et al. [81] and those reported by Papadakis el al. in [80]. Papadakis et al. reported results obtained based on Iterative Blocking introduced by Whang et al. in [116].

In the first part of the case study, our algorithm detected 42 more duplicates relative to the Efficiency Workflow using 41% less comparisons.
In the second part of the case study, our algorithm detected 2,504 duplicates using 39% fewer comparisons.

We also observed an improvement in time. We run our experiments in the similar setup as described by Papadakis et al. in [80,81]. Our algorithm needed only 0.22 of a minute to complete for the smaller datasets using 8 GB machine and 2.4 hours for the larger datasets with an access to only 16GB of the server’s memory. We improved the time by 80% in comparison to results reported by Papadakis et al.

Our algorithm runs significantly faster than the results reported in [81] since we eliminate a greater number of oversized redundant blocks in the first step of our algorithm. Thus, in the comparison selection process we evaluate a smaller number of comparisons. Also, by employing our 2-stage comparison selection process we discover more duplicates. Each additional duplicate decreases the number of comparisons since once records are identified as duplicates they are not compared again in subsequent blocks. We also saved some time by introducing Bloom filters to do operations on comparisons.

<table>
<thead>
<tr>
<th>Table 11 Results of Selective Comparison Algorithm for Dmovies</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Selective Comparison Algorithm</strong></td>
</tr>
<tr>
<td>Comparisons</td>
</tr>
<tr>
<td>Dmovies</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 12 Brute Force results for Dmovies</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Brute Force</strong></td>
</tr>
<tr>
<td>Comparisons</td>
</tr>
<tr>
<td>Dinfoboxes</td>
</tr>
</tbody>
</table>
Table 13 Results of Selective Comparison Algorithm for $D_{\text{infoboxes}}$

<table>
<thead>
<tr>
<th>Selective Comparison Algorithm</th>
<th>Comparisons</th>
<th>Duplicates</th>
<th>$\text{Ratio}\ \text{comps/dups}$</th>
<th>Re</th>
<th>F-measure</th>
<th>RR</th>
<th>Time(hrs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$D_{\text{infoboxes}}$</td>
<td>$2.73 \cdot 10^7$</td>
<td>839,790</td>
<td>32.53</td>
<td>0.94</td>
<td>0.97</td>
<td>0.99</td>
<td>2.40</td>
</tr>
</tbody>
</table>

Table 14 Brute Force results for $D_{\text{infoboxes}}$

<table>
<thead>
<tr>
<th>Brute Force</th>
</tr>
</thead>
<tbody>
<tr>
<td>$D_{\text{infoboxes}}$</td>
</tr>
</tbody>
</table>

Table 15 Performance of Several Algorithms for $D_{\text{movies}}$

<table>
<thead>
<tr>
<th>Method Name</th>
<th>Selective Comparison Algorithm</th>
<th>Papadakis' Efficiency Workflow</th>
<th>Iterative Blocking[116]</th>
<th>Brute Force</th>
</tr>
</thead>
<tbody>
<tr>
<td># of Comparisons</td>
<td>$5.17 \cdot 10^4$</td>
<td>$8.71 \cdot 10^4$</td>
<td>$10.41 \cdot 10^6$</td>
<td>$6.40 \cdot 10^8$</td>
</tr>
<tr>
<td># of Duplicates</td>
<td>21,491</td>
<td>21,449</td>
<td>22,268</td>
<td>22,405</td>
</tr>
<tr>
<td># of Coms/Dups</td>
<td>2</td>
<td>4</td>
<td>467</td>
<td>28565</td>
</tr>
</tbody>
</table>

Table 16 Performance of Several Algorithms for $D_{\text{infoboxes}}$

<table>
<thead>
<tr>
<th>Method Name</th>
<th>Selective Comparison Algorithm</th>
<th>Papadakis' Efficiency Workflow</th>
<th>Iterative Blocking[116]</th>
<th>Brute Force</th>
</tr>
</thead>
<tbody>
<tr>
<td># of Comparisons</td>
<td>$2.73 \cdot 10^7$</td>
<td>$4.46 \cdot 10^7$</td>
<td>$2.55 \cdot 10^9$</td>
<td>$2.58 \cdot 10^{12}$</td>
</tr>
<tr>
<td># of Duplicates</td>
<td>839,790</td>
<td>837,286</td>
<td>891,604</td>
<td>892,586</td>
</tr>
<tr>
<td># of Coms/Dups</td>
<td>33</td>
<td>53</td>
<td>2860</td>
<td>2890478</td>
</tr>
</tbody>
</table>
Chapter 6
6 Conclusion

In this thesis, we developed new methods for improving testing, and reliability of service-oriented applications. We provided metrics to measure the influence of data quality on software reliability and introduced an entity resolution algorithm. Our entity resolution algorithm effectively improves data quality by identifying duplicate records.

In Chapter 3, we discussed the development of an approach for better understanding and more effectively testing new and emerging service-oriented applications. With this type of applications it is necessary to take their presentation, business logic, and data tiers into account. In order to take into account all the tiers in these complex, heterogeneous systems, we developed an integrated dependence model for service-oriented applications. This model is the very first of its type and it examines relationship between all the tiers with applications. By utilizing dependence models, the proposed testing strategy can cover requirement specifications, search for and identify missing requirements, and create critical combinations of requirements. This approach goes beyond normal industrial practices and exceeds the capability of existing testing techniques for service-oriented applications. The results obtained from our empirical studies demonstrate that the model is not only practical but also effective for industrial large-scale systems. It improves the testing of the provided services and ensures the dependability of the required services.

In Chapter 4, we presented a new perspective on estimating the reliability of web services. Traditional reliability modeling approaches focus on failures caused by residual defects in the code. Web-based services work in a distributed and dynamic fashion where
the reliability of services can be affected by their execution context and runtime
environment. Recent studies have attempted to address the issues of server and network
availability and identify the types and amounts of usage loads that may lead to service
failures. However, few studies have explored the effect of data quality on the reliability of
services despite the fact that they rely heavily on internal/external data.

Service-oriented applications are normally designed for inter-organizational
communications and resource sharing. By their nature, services use sources of data that
come from different resources through a variety of means and that are continuously updated.
Thus, it is likely that the quality of data for services is not as good as for traditional intra-
organizational software. If a data source provides invalid data to a service, then the service
may not be able to deliver the expected results, and hence the service may fail.

To examine the effects of invalid data on the reliability of services, we developed
a methodology that incorporates data quality into reliability modeling and can therefore
provide a better estimation that can account for failures caused by invalid data. In our case
study, we observed that more than 16% of service failures were caused by invalid data.

This result suggests that the quality of data components can be as important as that
of software components, and that reliability modeling should include data components as
well. To facilitate this, we have developed an approach to quantify the quality of database
relations and compute the quality of data components based on the type of interactions they
have with software components. These interactions are captured in a flexible
heterogeneous architecture-based model that addresses different architecture styles. Our
reliability model provides more accurate estimations by taking into account the quality of
data sources. Moreover, it addresses different architecture styles and can be further
extended to include external web services, web servers, and other components in the execution context.

Inspired by our results demonstrating the importance of data quality, in the fifth chapter of the thesis, we presented an Entity Resolution (ER) algorithm that improves the quality of data by identifying duplicate records among heterogeneous datasets. Despite the large number of recent research efforts to provide methods for data deduplication, entity resolution for heterogeneous data has not yet been fully explored. In addition, existing approaches focus on blocking methods, which do not sufficiently decrease the computational complexity of the entity resolution processes. Thus, we presented an ER algorithm that not only provides a blocking scheme, but also the 2-stage comparison selection process. Contrary to the existing approaches, we carefully avoid building blocks that may contain records that do not have much in common. In addition, once we established the list of blocks, we are able to process the pairwise-comparisons within blocks to resolve only those that are likely to result in finding duplicates.

Since ER techniques need to be applied to large amounts of data, it is important for them to be effective and computationally efficient. The results of our experimental studies verified the usefulness of blocking scheme, the 2-stage comparison selection process, and use of Bloom filters with respect to both effectiveness and efficiency. For the former, we observed that we could find useful comparisons more effectively if we took into consideration the uniqueness of shared tokens and their lengths in addition to the number of common tokens. Thus, the 2-stage comparison selection process increased the number of duplicates being found. In addition, token, block and comparisons selection processes resulted in the reduction of necessary comparisons by 40 percent on average, increasing
the efficiency of the ER process. Consequently, we noticed that less than 2.40 comparisons (for smaller collections) and 32.53 comparisons (for large collections) were needed to discover a duplicate, making our approach scalable for small, medium and large datasets. We also observed an improvement in running time due to decreased number of pairwise comparisons and use of Bloom filters.

In the future, we hope to focus our efforts on employing probabilistic and randomized algorithms in the entity resolution processes.
References:


118 *WS-BPEL 2.0 for SOA Composite Applications with Oracle SOA Suite 11g*  Matjaz B. Juric, Marcel Krizevnik


