FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Reliable Provisioning of Data-Centric and Event-Based Applications in the Cloud

DISSERTATION

zur Erlangung des akademischen Grades

Doktor der technischen Wissenschaften

eingereicht von

Dipl.-Ing. Waldemar Hummer
Matrikelnummer 0416710

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Univ.-Prof. Dr. Schahram Dustdar

Diese Dissertation haben begutachtet:

(Univ.-Prof. Dr. Schahram Dustdar)

(Prof. Dr. Mauro Pezzè)

Wien, 29.01.2014

(Dipl.-Ing. Waldemar Hummer)

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Reliable Provisioning of Data-Centric and Event-Based Applications in the Cloud

DISSERTATION

submitted in partial fulfillment of the requirements for the degree of

Doktor der technischen Wissenschaften

by

Dipl.-Ing. Waldemar Hummer
Registration Number 0416710

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Univ.-Prof. Dr. Schahram Dustdar

The dissertation has been reviewed by:

(Univ.-Prof. Dr. Schahram Dustdar)

(Prof. Dr. Mauro Pezzè)

Wien, 29.01.2014

(Dipl.-Ing. Waldemar Hummer)

Erklärung zur Verfassung der Arbeit

Dipl.-Ing. Waldemar Hummer
Josefstädter Straße 14/2/58, 1080 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum) (Unterschrift Waldemar Hummer)

Abstract

The past decade of distributed systems research has been shaped, among others, by three major trends: Service-Oriented Architecture (SOA) is a popular paradigm for implementing loosely coupled distributed applications; Event-Based Systems (EBS) are gaining momentum as a means for encoding complex business logic based on correlated event messages; moreover, Cloud Computing (CC) has fostered advanced virtualization and resource allocation techniques, further shaping the implementation possibilities of SOA and EBS. Distributed computing systems in general, and applications in the Cloud in particular, are often burdened with stringent requirements concerning reliability and security, dictated by business objectives (e.g., cost-benefit tradeoffs), contractual agreements (e.g., service level agreements, SLAs), or laws. One approach to reliability is software testing, which aims at identifying and avoiding software-induced faults in the first place. A second important aspect of reliability is fault management, which involves different challenges such as fault detection and runtime adaptation. Additionally, security and access control play a crucial role, particularly for multi-tenant Cloud environments. Systematic consideration of these aspects in the software development and validation process is a key concern and requires precise knowledge about the type and nature of potential threats to reliability.

This doctoral thesis tackles the aforementioned challenges and contributes a set of novel methods and tools for reliable provisioning of data-centric and event-based applications in the Cloud. The primary types of considered applications are business processes and workflows which integrate services and particularly data from a plurality of sources, leveraging established concepts of SOA and EBS. The framework introduced in this thesis provides a robust, scalable, and secure execution environment for such applications. The contribution is split up into three core parts. First, *WS-Aggregation* is an event-based data processing platform that features elasticity, optimized load distribution, runtime adaptation, and fault management. Second, the *TeCoS* platform is used to perform systematic testing for application bugs and integration issues such as data incompatibilities. Third, the *SeCoS* framework enforces access control policies to assure responsibilities and avoid unauthorized actions. The approach is thoroughly evaluated and put into perspective with existing work. A multitude of representative experiments have been conducted with the implemented prototypes, deployed in different commercial and open source Cloud environments. The large-scale performance tests demonstrate the elasticity under changing workload patterns. The second class of experiments evaluates the testing approach by injecting various faults into running application instances. This evaluation shows that the system correctly identifies faults and reports the achieved test coverage in different configurations. In a third step, the access control enforcement procedure is evaluated for consistency and performance.

Kurzfassung

Das vergangene Jahrzehnt der Forschung von verteilten Systemen war unter anderem geprägt von drei Trends: Service-Oriented Architecture (SOA) hat sich als verbreitetes Paradigma für lose gekoppelte verteilte Applikationen etabliert; Event-basierte Systeme (EBS) ermöglichen komplexe Applikationslogiken basierend auf korrelierten Event-Daten; darüber hinaus hat Cloud Computing die Technologien für Virtualisierung und Ressourcen-Allokation vorangetrieben, wodurch fortgeschrittene Implementierungsmöglichkeiten für SOA und EBS entstehen. Verteilte Systeme im Allgemeinen, sowie Applikationen in der Cloud im Speziellen, sind häufig strengen Anforderungen an Verlässlichkeit und Sicherheit unterworfen. Die Anforderungen ergeben sich aus mehreren Gründen wie Geschäftszielen (z.B. Kosten/Nutzen-Abwägungen), vertraglichen Verpflichtungen (z.B. Dienstgütevereinbarungen), oder Gesetzen. Software-Testen gilt als zentraler Ansatz mit dessen Hilfe versucht wird, Fehler zu identifizieren bzw. zu vermeiden. Ein weiterer wichtiger Aspekt ist Fehler-Management, mit unterschiedlichen Herausforderungen wie Fehler-Identifikation sowie Anpassung des Systems an Laufzeit-Fehler. Zudem haben Sicherheit und Zugriffskontrolle einen hohen Stellenwert, insbesondere in Cloud-Umgebungen mit mehreren Mandanten. Die systematische Berücksichtigung dieser Aspekte ist eine zentrale Problematik, welche präzises Wissen über potenzielle Fehler und Gefährdungen erfordert.

Die vorliegende Dissertation behandelt die ebengenannten Probleme und entwickelt neue Methoden und Werkzeuge für den verlässlichen Betrieb von datenzentrierten und event-basierten Applikationen in der Cloud. Die primäre Art der hier untersuchten Anwendungen sind Geschäftsprozesse und technische Abläufe, bei denen eine Vielzahl von technischen Services, insbesondere auch Daten, aus verschiedenen Quellen verarbeitet und integriert werden. Das vorgestellte System bietet eine robuste, skalierbare und sichere Plattform für derartige Applikationen. Die Forschungsbeiträge werden anhand dreier Teile diskutiert. Das Kernstück ist *WS-Aggregation*, eine verteilte Plattform zur Verarbeitung von event-basierten Daten mit Funktionalitäten für Skalierbarkeit, optimierte Lastverteilung, Laufzeit-Adaptierung, sowie Fehler-Management. Die *TeCoS* Plattform bietet Methoden für systematisches Testen und identifiziert Anwendungsfehler sowie Integrationsprobleme, wie z.B. Daten-Inkompatibilitäten. Mittels Regeln zur Zugriffskontrolle stellt die *SeCoS* Plattform Verantwortlichkeiten sicher und blockiert unerlaubte Zugriffe. Der gesamte Ansatz wird umfangreich evaluiert und mit existierenden Arbeiten verglichen. Eine Vielzahl von repräsentativen Experimenten in unterschiedlichen Cloud-Umgebungen demonstrieren die Performance, Skalierbarkeit und Elastizität des Systems. Die zweite Art von Experimenten evaluiert den Test-Ansatz und zeigt durch gezielten Einbau von Fehlern in laufende Applikationen, dass das System korrekt Fehler erkennt. Der dritte Teil der Evaluierung untersucht den Mechanismus zur Zugriffskontrolle unter den Gesichtspunkten Konsistenz und Laufzeitverhalten.

Acknowledgements

This PhD thesis marks the finish line for roughly a decade of my academic education in Computer Science. Successfully finishing this PhD demanded from me, above all, two main ingredients: a genuine impetus to constantly dig deep into scientific problems and solutions, paired with an immense level of determination and endurance. Looking back, I firmly believe that this work would have never been possible without the profound support of many different persons and the enjoyable circumstances I have been able to work in. It can therefore not be stressed enough that this thesis is not the achievement of a single person, but also the result of the caring and considerate environment of people who have all contributed in their individual ways.

First and foremost, I would like to express my gratitude to Prof. Schahram Dustdar who has greatly guided me throughout the dissertation and provided a highly productive and pleasant working environment at the Distributed Systems Group. I am also most grateful to Prof. Mauro Pezzè for serving as the examiner of the thesis. Furthermore, I want to thank all my colleagues from TUVIE, UNIVIE, and WU for the fruitful discussions and collaborations, most notably Philipp Leitner, Christian Inzinger, Benjamin Satzger, Alessio Gambi, Patrick Gaubatz, Mark Strembeck, and Uwe Zdun, and many more colleagues from Vienna and different parts of the globe who are not all listed here but have also significantly supported and influenced me. I am also immensely grateful for the outstanding opportunity of collaborating with IBM in the course of two research internships, which provided me with invaluable personal and technical experiences. My special thanks go to Orna Raz, Onn Shehory and Eitan Farchi from IBM Haifa, as well as Florian Rosenberg, Fábio Oliveira and Tamar Eilam from IBM T.J. Watson in New York.

My deepest and sincerest thanks are dedicated to my family, particularly my loving parents who supported me emotionally, intellectually, and financially throughout all these years. My success will always be equally your success! My beloved sisters Karo and Nora have been most supportive by always lending an ear for my problems and accompanying me through the highs and lows. Last but not least, it makes me happy and proud to have shared so many joyful and unforgettable moments with dear friends and companions, who have always provided the highly desirable distraction and have greatly shaped me into the person I am today.

Waldemar Hummer
Vienna, December 2013

The research for this thesis has received funding from the European Community's Seventh Framework Programme under grant agreements 215483 (S-Cube), 257483 (Indenica), 257574 (FITTEST). The work is partially supported by the Austrian Science Fund (FWF), grant number P23313-N23 (Audit 4 SOAs).

Table of Contents

List of Figures

List of Tables

List of Listings

Publications

This thesis is based on work previously published in scientific conferences, workshops, journals and books. These core papers are listed here once, and for brevity are not explicitly referenced throughout the thesis. Parts of these papers are contained in verbatim. Please refer to Chapter B in the appendix for a full publication list of the author of this thesis.

- Waldemar Hummer, Florian Rosenberg, Fábio Oliveira, and Tamar Eilam. Testing idempotence for infrastructure as code. In *14th ACM/IFIP/USENIX Middleware Conference*, pages 368–388, 2013. **Best Student Paper Award**

- Waldemar Hummer, Florian Rosenberg, Fábio Oliveira, and Tamar Eilam. Automated testing of chef automation scripts. In *ACM/IFIP/USENIX Middleware Conference (tool demo track)*, 2013

- Waldemar Hummer, Orna Raz, Onn Shehory, Philipp Leitner, and Schahram Dustdar. Testing of Data-Centric and Event-Based Dynamic Service Compositions. *Software Testing, Verification and Reliability (STVR)*, 23(6):465–497, 2013

- Waldemar Hummer, Patrick Gaubatz, Mark Strembeck, Uwe Zdun, and Schahram Dustdar. Enforcement of Entailment Constraints in Distributed Service-Based Business Processes. *Information and Software Technology (IST)*, 55(11):1884–1903, 2013

- Waldemar Hummer, Benjamin Satzger, and Schahram Dustdar. Elastic Stream Processing in the Cloud. *Wiley Interdisciplinary Rewiews: Data Mining and Knowledge Discovery*, 3(5):333–345, 2013

- Waldemar Hummer, Christian Inzinger, Philipp Leitner, Benjamin Satzger, and Schahram Dustdar. Deriving a unified fault taxonomy for distributed event-based systems. In *6th ACM International Conference on Distributed Event-Based Systems (DEBS)*, pages 167–178, 2012

- Waldemar Hummer, Philipp Leitner, and Schahram Dustdar. WS-Aggregation: Distributed Aggregation of Web Services Data. In *ACM Symposium On Applied Computing (SAC)*, pages 1590–1597, 2011

- Waldemar Hummer, Orna Raz, Onn Shehory, Philipp Leitner, and Schahram Dustdar. Test coverage of data-centric dynamic compositions in service-based systems. In *4th International Conference on Software Testing, Verification and Validation (ICST'11)*, pages 40–49, 2011

- Waldemar Hummer, Patrick Gaubatz, Mark Strembeck, Uwe Zdun, and Schahram Dustdar. An integrated approach for identity and access management in a SOA context. In *16th ACM Symposium on Access Control Models and Technologies (SACMAT)*, pages 21–30, 2011

- Waldemar Hummer, Philipp Leitner, Benjamin Satzger, and Schahram Dustdar. Dynamic migration of processing elements for optimized query execution in event-based systems. In *1st International Symposium on Secure Virtual Infrastructures (DOA-SVI'11), OnTheMove Federated Conferences*, pages 451–468, 2011

- Waldemar Hummer, Benjamin Satzger, Philipp Leitner, Christian Inzinger, and Schahram Dustdar. Distributed Continuous Queries Over Web Service Event Streams. In *7th IEEE International Conference on Next Generation Web Services Practices (NWeSP)*, pages 176–181, 2011

- Waldemar Hummer, Orna Raz, and Schahram Dustdar. Towards Efficient Measuring of Web Services API Coverage. In *3rd International Workshop on Principles of Engineering Service-Oriented Systems (PESOS), co-located with ICSE'11*, pages 22–28, 2011

- Waldemar Hummer, Philipp Leitner, and Schahram Dustdar. A Step-by-Step Debugging Technique To Facilitate Mashup Development and Maintenance. In *4th International Workshop on Web APIs and Services Mashups, co-located with ECOWS'10*, 2010

- Christian Inzinger, Waldemar Hummer, Benjamin Satzger, Philipp Leitner, and Schahram Dustdar. Identifying incompatible service implementations using pooled decision trees. In *28th ACM Symposium on Applied Computing (SAC), DADS Track*, pages 485–492, 2013

- Christian Inzinger, Waldemar Hummer, Benjamin Satzger, Philipp Leitner, and Schahram Dustdar. Towards identifying root causes of faults in service-based applications. In *31st International Symposium on Reliable Distributed Systems (poster paper)*, pages 404–405, 2012

CHAPTER 1

Introduction

Throughout the last years, the world has witnessed a tremendous growth in electronic data, generated globally by a vast spectrum of different providers. The continuing trend towards collection and automated processing of data has hit virtually every field of our society: financial data, health care data, data from the retail sector, data in social networks, open government data, or sensor data, to name just a few examples. Today, the World Wide Web (WWW) is the primary channel for data dissemination – via human-readable Web documents as well as machine-processable Web Services (WS) [364]. The volume and velocity of global data are growing at exponential rates – Cisco's *Visual Networking Index* [152] forecasts a global annual Internet Protocol (IP) traffic of 1.4 zettabytes in 2017, up from 523 exabytes in 2012[1].

The sheer amount of data available on the Web has opened up exciting application areas to provide added value services and to generate higher-level information from raw data. Data Mining [122] is a prominent research field that deals with selecting, integrating, transforming, and evaluating data to extract knowledge and patterns of interest. In addition, methods from Business Process Management (BPM) [338] can be applied to create well-defined workflows for (semi-)automated data processing by machines and humans. The advances in Event-Based Systems (EBSs) [97, 207, 237] and Stream Processing [15, 17, 312] provide the technological foundation for continuous queries over streams of data. The Web is a vivid environment with new data and service providers joining at rapid pace. Applications for data processing are hence rarely hard-wired, but rather configured to dynamically select among the best available providers at runtime. The Service-Oriented Architecture (SOA) [92, 96] is a popular paradigm to support dynamic service binding, effectively decoupling service consumers from providers. To cope with large amounts of data, particularly under fluctuating work loads, flexible and reliable system architectures for distributed processing are essential. Cloud Computing [10, 48] has gained significant importance as a means to dynamically allocate and release computing resources to implement scalable applications in a cost-efficient way.

[1] 1 zettabyte = 10^{21} bytes; 1 exabyte = 10^{18} bytes

Distributed computing systems in general, and data-centric applications in the Cloud in particular, are often burdened with stringent requirements concerning reliability and security [13], dictated by business objectives (e.g., cost-benefit tradeoffs), contractual agreements (e.g., Service Level Agreements), or laws. Software reliability is defined by the ANSI/IEEE as the "*probability of failure-free software operation for a specified period of time in a specified environment*" [8]. One key approach to reliability is **software testing** [30], which attempts to identify and avoid software-induced faults in the first place. In software testing, the application or system under test is typically probed with a multitude of different inputs generated using a certain strategy (e.g., those inputs that are likely to cause a failure). The notion of test coverage [211, 379] attempts to quantify the degree of thoroughness of testing and measures the percentage of tested parts of a program, with respect to code statements, branches, paths, predicates, and more. By achieving a high test coverage, the likelihood of failures caused by the software decreases, and hence reliability increases (by tendency). A second important aspect of reliability is **adaptability and fault-tolerance** [165]. A fault tolerant system is able to adapt and re-organize itself in the presence of (or anticipation of) faults, effectively allowing it to maintain correct functionality and consistent Quality of Service (QoS) over long run times. Fault tolerance or fault management in a wider sense entails different runtime challenges such as fault detection, isolation, or recovery. Implementing fault handling mechanisms involves anticipation of unforseen situations and hence requires precise knowledge about the type and nature of faults that may occur. To avoid runtime failures such as overloaded nodes during load bursts, one particularly important type of adaptability in the context of event-based data processing is elasticity, i.e., the ability of a system to scale up and down dynamically. While scalability is a static property which tells whether systems can grow in size at all (e.g., can be deployed onto multiple computing nodes), elasticity is the dynamic property which indicates that the system is able to adapt its scale dynamically during runtime in response to changes in the environment [2]. Elastic behavior often leads to a multitude of complex internal changes in a system or application, which are error-prone and hence further increase the necessity of thorough testing and reliability mechanisms. A third challenging aspect for reliable application provisioning in the Cloud is **security and access control**. Due to the multi-tenancy inherently encountered in Cloud environments, blocking unauthorized access to protected resources is crucial. The concept of Role-Based Access Control (RBAC) [284] has become the de-facto standard for modeling and enforcement of authorization policies. Beyond this basic mapping of roles, subjects and permissions there are more advanced access control concepts which allow complex relationships between different rights and duties, for instance binding or mutual exclusion of duties [357]. Such concepts are important in the context of data-centric applications in the Cloud, to avoid fraud and data abuse (for instance, considering a hospital software with sensitive patient data). However, the current development support for advanced security features in most Cloud platforms is poor, hence developers tend to hard-code tailor-made security enforcement procedures, which are complex and error-prone.

Overall, reliable systems must be shown to function correctly and be able to efficiently retain continuous functionality in the presence of changes in the environment, such as load fluctuations or changes in QoS requirements. Cloud environments pose difficult challenges to application development and operations. The challenges are influenced by business goals (e.g.,

SLAs, cost-benefit tradeoffs), technical requirements (e.g., scalability/elasticity, multi-tenancy) and legal obligations (e.g., security and access control). Provisioning of data-centric and event-based applications in the Cloud hence requires robust designs, thorough testing, and flexible adaptation mechanisms.

1.1 Problem Statement

The aforementioned challenges open a broad field of research questions to be solved. To narrow down the scope of this thesis, a detailed problem statement is given in the following. First, in order to clarify the problem domain, the concrete context and type of applications considered in this work are outlined in Section 1.1.1. Second, the core research questions are formulated in Section 1.1.2.

1.1.1 Problem Domain and Context

The core goal of this thesis is to reliably provision data-centric and event-based applications in Cloud environments. In a narrow sense, *provisioning* refers to the set of activities performed to prepare an application and bring it into a state where it is usable for end users, involving development (programming), testing, configuration, and deployment. However, in this thesis we consider end-to-end provisioning in a wider sense [348], also including runtime activities like monitoring, optimization, adaptation, elastic scaling, and fault management. The term *data-centric* means that the primary focus is on applications in which data flow (and data processing) is an integral part, in contrast to applications that focus mostly on control flow for process orchestration, e.g., steering of an industrial production process or similar. The latter class of applications is considered as well (particularly regarding security and access control), but the majority of contributions in this thesis are tailored to data-centric applications. The term *event-based* indicates that asynchronous processing over continuous streams of event data receives special attention.

The abstracted overview in Figure 1.1 outlines the problem domain and depicts the core stakeholders and system resources. The applications under consideration integrate and process data from a multitude of services, which are accessible via standardized (Web) interfaces and operate either under external control or within the same Cloud environment as the application. Various heterogeneous services can be integrated: invokable Remote Procedure Call (RPC) style Web services (marked with an incoming arrow in the figure), or queryable databases (cylindric symbol), or Publish/Subscribe (Pub/Sub) based services which continuously publish event data to the subscribed applications (marked with an outgoing arrow). The services are typically configured with security and access control measures (indicated by a lock symbol), although entirely open and unsecured services may also exist. The applications are deployed on an elastic middleware platform, which is itself deployed on a variable set of Virtual Machines (VMs). The figure illustrates three active VMs and two grayed out VMs which are currently idle but can be acquired in the future to handle peak loads.

Two main classes of applications are considered, yet similar concepts apply to both: 1) purely technical automated workflows where the main focus is to query and integrate data (e.g.,

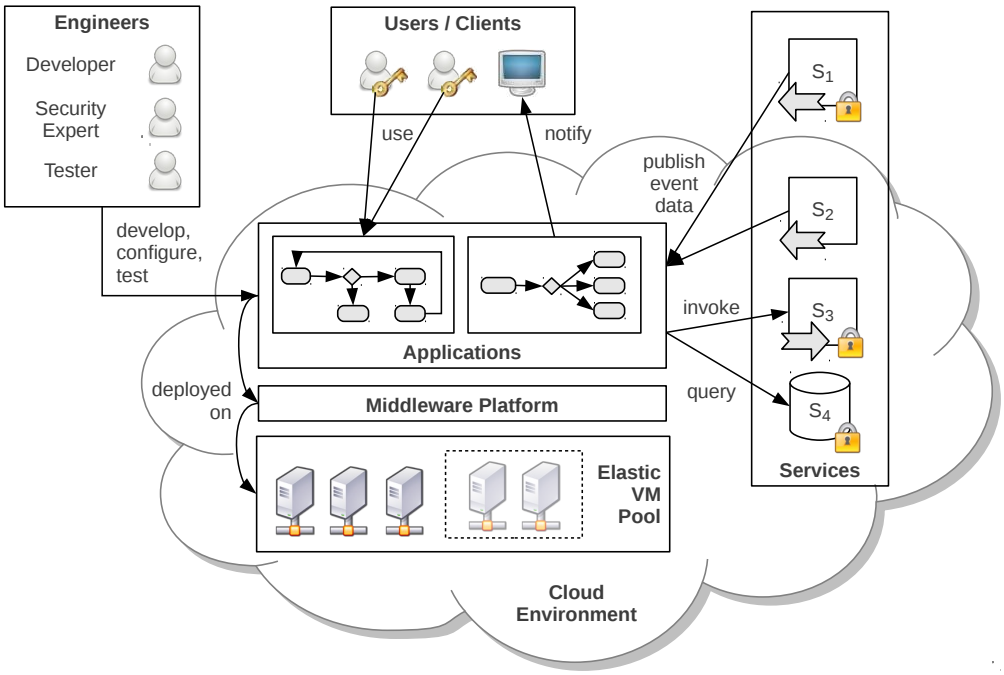


Figure 1.1: Problem Domain and Application Context

pattern detection over event streams with stock market data), and 2) semi-automated workflows with human participation, where the users perform part of the activities (e.g., a patient examination process in a hospital). The human users (subjects) are equipped with responsibilities and access rights (indicated by a key symbol). The applications are created by different engineers with specialized domain knowledge: developers implement the application business logic, security experts define access control policies, and testers systematically assess the correct functioning of the applications. The middleware platform is the crucial part which should provide support for all aspects of the application provisioning: communication with the services, enforcement of access control policies, performance monitoring, runtime adaptation and elasticity, as well as support for test execution.

1.1.2 Research Questions

The contributions of this thesis are aligned along three main research questions (Q1-Q3), which are introduced and briefly discussed in the following.

Q1: What are suitable methods and a supporting system to reliably execute event-based data processing applications in the Cloud, leveraging elasticity and dynamic resource allocation?

The primary concern for event-based data processing applications is to continuously adapt to fluctuations in the volume and rate at which requests and data items arrive, in order to ensure

consistent quality of service (QoS). The recend trend towards Cloud computing has fostered the advance of elastic computing systems, a new breed of software systems leveraging dynamic resource allocation to balance (cost) efficiency with QoS guarantees [91]. Under elasticity, the allocated resources are aligned with the system's workload: load bursts may require rapid scale-out, whereas the amount of computing resources during regular operation is reduced to a minimum. The decision when and how to scale involves multiple factors of influence. Deployed in a distributed environment, the workload should be evenly balanced among an appropriate number of participating compute nodes, while minimizing the network traffic between nodes. Redundant processing improves on reliability, e.g., in the face of node outages, but adds overhead to the system, resulting in increased costs. Achieving an optimal tradeoff between these dimensions is a non-trivial problem which is still actively researched and has not been entirely solved. What is more, elastic behavior typically induces a series of complex and error-prone reconfigurations within the computing platform. For data processing systems, such as event processing networks, the reconfiguration might involve migration of state data, buffers, event subscriptions, and more. For such systems to operate reliably, potential runtime faults need to be anticipated and systematically dealt with. While some aspects of the aforementioned challenges have been studied before (see discussion in Section 2), no solution currently integrates reliable provisioning across all Cloud layers, covering the infrastructure layer, the platform, as well as the application layer.

Q2: Which testing and fault localization techniques can be applied to ensure reliable provisioning of data-centric and event-based applications in the Cloud?

Data-centric and event-based applications in the Cloud are confronted with various threats to reliability. In this thesis, our focus is on two distinct aspects. First, from an application level perspective, combining heterogeneous data sources from multiple providers introduces potential integration issues. As outlined in Section 1.1.1, the processing workflow is typically broken down into individual steps (activities) which are steered and interconnected by data dependencies. These dependencies are essential for correct functioning of the applications, hence sensitive runtime errors, such as syntactic and semantic data incompatibilities, have to be anticipated and dealt with. Second, on the platform and Cloud infrastructure layer, the dynamics of elasticity aggravate the challenge of reliable application provisioning. The key prerequisite for implementing elasticity is that the underlying infrastructure resources must be deployed and configured reliably. This requirement is often assumed blindly, but if that assumption breaks then the elasticity of the system is likely to break. For both aspects, systematic testing by instantiating the application under different configurations is an essential activity to identify bugs and issues upfront. However, due to the complexity of the problem space, in general not all potential configurations can be tested. Suitable testing techniques, combined with tailored test coverage metrics, are required to achieve a certain level of confidence that the application operates properly under different runtime conditions. Since upfront testing cannot anticipate all potential changes in the environment during operation, additional runtime monitoring must be employed to detect and localize faults, which can then be fixed by the developers. Despite the large body of existing work on testing and fault localization, the two particular aspects discussed here are not yet covered in the scientific literature.

Q3: How can security and access control policies be enforced in the context of data processing applications and workflows?

Ensuring security and data privacy is an ever-increasing concern in today's computing landscape. The prevalence of sensitive personal electronic data in various areas such as e-health applications, social networks, or government platforms, has spurred private and public data protection initiatives, such as the 2012 European Commission's proposal for a *General Data Protection Regulation* [72]. Particularly Cloud environments, which are inherently faced with multi-tenancy, require systematic techniques to protect data from unauthorized access. Applications for data processing, such as business process workflows with direct human involvement, are hence often subject to security constraints and access control policies. Security and access control has been an actively researched field with many seminal results, including cryptographic schemes [310], generic access control models (e.g., RBAC), or task-based entailment constraints [357]. Despite the majurity of the field, there is still a lack for systematic development support and platform-level enforcement integrated with state-of-the-art technologies such as the Web Services Business Process Execution Language (WS-BPEL) [246], allowing users to define custom access constraints to be enforced at runtime.

1.2 Scientific Contributions

Guided by the research questions formulated in Section 1.1.2, the core contributions are highlighted here to provide an outline of the work carried out in the scope of this thesis. Figure 1.2 depicts an overview of the main system artifacts and connects them to the scientific contributions, which are discussed in more detail in the following.

Contribution 1: Comprehensive fault taxonomy for event-based systems. The first contribution studies faults and reliability in EBSs. This work is driven by the fact that there is currently no common understanding of faults in EBSs, which starkly contrasts with other fields where faults are well-understood, e.g., faults in operating systems [12], faults in object-oriented software [36], or faults in service-based applications [43, 59]. An extensive literature review has been conducted, in which five core sub-areas of EBSs are identified. These areas share many commonalities, and in fact use slightly different terminology and approaches for very similar concepts. The commonalities among the five areas are captured in a generic model for EBSs, which is partly based on previous work (e.g., [236, 304, 351]). The model is used to derive a taxonomy with 30 distinct faults in EBSs, based on well-established dimensions defined in the seminal work by Avizienis et al. [13]. The developed taxonomy provides a foundation for implementing fault diagnosis [161] and fault tolerance [163, 165] techniques, which are relevant for the remaining contributions of this thesis. Contribution 1 has originally been presented in [136].

Contribution 2: Query model and elastic platform for event processing. The second contribution introduces WS-Aggregation, a distributed and elastic platform for event-based data processing (*WS* stands for Web services [364], which is the primary technology used for the underlying data sources). Using a specialized query model and the query language termed Web services Aggregation Query Language (WAQL), the platform allows to integrate and connect event-based data sources on the basis of declarative data dependencies. Since applications with

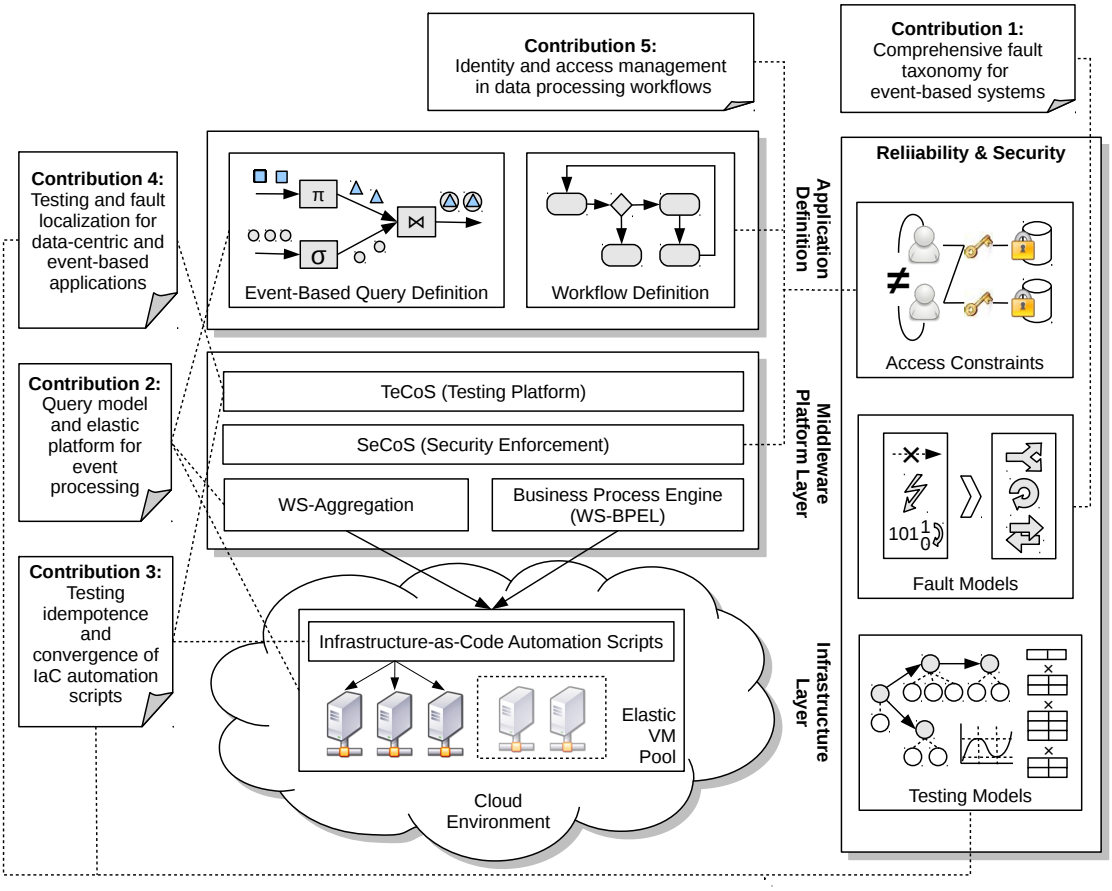


Figure 1.2: Overview of System Artifacts and Scientific Contributions

large dependency graphs are hard to develop and maintain, an integrated debugging framework supports the developer inspecting and asserting the application state at different times of the execution. WS-Aggregation works mainly with Extensible Markup Language (XML) data, hence the query language is based on XML Query Language (XQuery). WS-Aggregation is tailored to the Cloud and the built-in elasticity features allow to scale the system dynamically to a multitude of aggregator nodes running on distributed machines. Different optimization approaches are employed in order to minimize resource utilization and maintain reliable operation of the platform. For instance, 3-way querying is a possibility to optimize requests by reducing the amount of data transferred between aggregator nodes. To cope with varying workloads WS-Aggregation performs dynamic query distribution and optimized placement of processing elements. The optimization algorithm achieves a tradeoff between three dimensions: equal load distribution, low data transfer, and no duplicate buffering of events. One of the key technological challenges for supporting runtime adaptations is consistent migration of state data, which is systematically solved and evaluated within this thesis. Contribution 2 has originally been presented in [137, 139, 141, 147, 148].

Contribution 3: Testing idempotence and convergence of IaC automation scripts. In the third contribution, the focus is shifted to reliable provisioning on the middleware platform and infrastructure layer. The runtime adaptations of elastic systems, as exemplified in WS-Aggregation, require instantiation of new VMs which are hooked into the running platform on demand. These VMs need to be equipped with a particular Operating System (OS) and software stack, and it is often required to perform custom configurations on each of the newly launched VMs. These configurations are typically encoded in automation scripts which involve a series of tasks to bring the machine into a desired state, e.g., installing/updating software packages, starting services, writing to configuration files, etc. In recent years, the concept of Infrastructure as Code (IaC) [150, 240] has emerged as a means for systematic development of automation logic, following key principles in software engineering. IaC automations should be robust and repeatable: if the execution fails at some point, e.g., caused by a temporary network failure while downloading software packages, it should be possible to re-run the entire automation until the system eventually converges to the desired state. One of the cornerstones for robust and repeatable automation logic is idempotence [74]. An idempotent task is designed to have an effect only upon the first execution, whereas all subsequent executions should not produce any state changes or undesired side-effects. This thesis provides a testing framework for IaC automations which is able to detect idempotence issues. Starting from an abstracted model of the automation definition, state transition graphs (STGs) are constructed which represent the possible states of the system along different possible executions. The STG is used to derive test cases with particular focus on testing idempotence. The prototype implementation targets the popular IaC framework Chef [240, 253], but the approach is generally applicable. The evaluation has been conducted based on a large set of publicly available Chef scripts, which has successfully revealed a multitude of non-trivial bugs and idempotence issues. Contribution 3 has been published in [145,146].

Contribution 4: Testing and fault localization for data-centric and event-based applications. The fourth contribution revolves around identification of faults in data processing applications. The assumed application model is a Service-Based Application (SBA) that defines abstract services which are bound to concrete candidate services at runtime. This procedure, denoted dynamic binding [52], is a common practice in SBAs for runtime switching between services with favorable properties such as low costs or high QoS. However, integrating data and services from different providers creates high potential for faults, e.g., syntactic or semantic data incompatibilities. Two approaches are proposed to deal with such incompatibilities: *ex ante* testing, and *ex post* fault localization. First, the testing method helps to detect data integration issues up front. Based on the application definition and data dependencies between single services, data flow graphs are constructed and used to systematically derive test cases as different instantiations of the application. The proposed k-node data flow coverage metric allows testers to steer the testing process and limit the size of test suites. Based on the results gathered during testing, detailed coverage reports are generated for both the Application Programming Interface (API) level and the composition level of the entire application. Second, the fault localization technique detects faults occurring at runtime and attempts to analyze the root cause, i.e., the sequence of processing steps in the application that is likely responsible for causing a fault. The execution of applications is monitored and each processed request is stored as a trace file, containing the user input, application output, as well as variations in the internal processing (e.g.,

binding of concrete services). Efficient machine learning techniques are employed to process very large sets of traces. The concepts are implemented in the framework denoted TeCoS (Test Coverage for SBAs), and have been successfully applied and evaluated based on two distinct application definition models, WS-Aggregation and WS-BPEL. Contribution 4 has originally been published in [142–144, 154, 155].

Contribution 5: Modeling and automatic enforcement of access constraints in business processes. The fifth contribution introduces SeCoS (Secure Collaboration in SBAs), a framework for Identity and Access Management (IAM) [107] in service-based applications and business processes. The approach integrates three main concepts: Single Sign-On (SSO), RBAC, and task-based entailment constraints [315, 357]. SSO fosters security in cross-organizational applications by allowing users who are registered with one organization to also use protected services of trusted partners. Using RBAC, security experts are able to define access permissions, group them into roles, and assign those roles to subjects. The approach in this thesis uses a Policy Decision Point (PDP) to intercept any service invocations and check whether the invoking subject has the required access permissions. Moreover, task-based entailment constraints allow definition of fine-grained security policies, typically in the form of pairwise relations between tasks such as mutual exclusion or binding of duty. Within this thesis, the focus is primarily in devising re-usable concepts to provide convenient development support for applications with access restricted services. A Domain-Specific Language (DSL) for defining RBAC rules and access policies is proposed. The DSL artifacts are utilized to enrich business process definitions with security annotations. During deployment of the application, model transformations are applied to the annotated process definition, in order to have the process follow the required security enforcement procedure. The advantage of this approach based on model transformation is that the enforcement procedure takes place on the process level and does not pose additional requirements on the middleware, i.e., the process execution engine. Comprehensive experiments have been performed which empirically verify the consistency and analyze the performance overhead of security enforcement. Contribution 5 has originally been presented in [134, 135].

1.3 Thesis Organization

The remainder of this thesis is structured as follows. Chapter 2 provides background information and discusses the state of the art in areas related to this thesis. The five contributions outlined in Section 1.2 are then grouped into three main chapters. Chapter 3, which covers contributions 1, 2, and 3, introduces the data integration and event processing platform WS-Aggregation, derives the fault taxonomy for EBSs, discusses optimized query distribution and migration of processing elements, and details the approach for testing idempotence of IaC scripts. The discussion of the TeCoS platform in Chapter 4 covers contribution 4, including the method for testing of data-centric and event-based applications, as well as the machine learning based fault localization technique. Finally, Chapter 5 covers contribution 5 and focuses on automated enforcement of access control policies in data processing workflows and business processes. Each of the three core chapters (Chapters 3, 4, 5) follow a similar structure, including use cases, approach description, implementation details, evaluation, related work, and conclusions. Finally, Chapter 6 concludes the thesis with a reflection on the contributions and outlook for future work.

CHAPTER 2

Background

This chapter provides background information about well-established concepts and technologies which form the basis for the work carried out in this thesis.

2.1 Event-Based Systems and Data Stream Processing

Event-Based Systems (EBS) [97, 207, 237] in various fashions have gained considerable momentum as a means for encoding complex business logic on the basis of correlated, temporally decoupled event messages. Based on existing models for event and stream computing (most notably [206, 236, 304]), the most important terms related to EBS are defined. The core artifacts and terminology are illustrated in Figure 2.1. Various research sub-areas use slightly different terminology, hence Figure 2.1 contains alternative terms (in brackets) for the core concepts. These terms are used interchangeably throughout the thesis.

An *event* is an object encoding something that is happening for the purpose of computer processing (e.g., the change of a stock price). Typically, events are of a certain *type*, have a *timestamp*, and contain further specific data. A *complex event* results from applying processing steps, like aggregation and filtering, to one or more other events. Events are emitted by (event) *sources* and consumed by (event) *sinks*. An (event or data) *stream* is a linear sequence of events, typically ordered by time. Streams are usually considered (potentially) infinite, and a *window* [15, 39, 261] is some finite portion of a stream.

To distinguish different types of windows (or window queries), we formalize an event stream E as a sequence of events $E = \langle e_1, e_2, e_3, \ldots \rangle$. A window query, at each point in time, evaluates a set of active (or open) windows, denoted W. A window $w \in W$ is a subsequence of the event stream, denoted $w \subseteq E$ (based on the notation for subsets). Each window w has a start condition ($s(w)$) and an end condition ($e(w)$). The window types are illustrated in Figure 2.2 based on a simple exemplary event stream with "start" and "end" event types. One basic type of query window is the *growing window* which binds data values to be available over the entire stream. Another simple type is the *single item window* where each single event is considered separately.

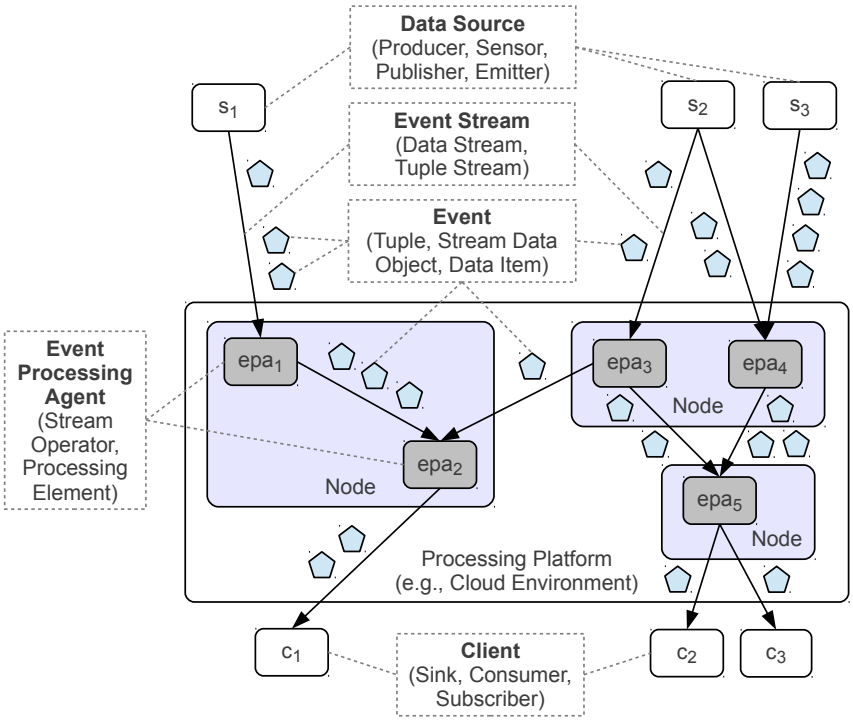


Figure 2.1: Core Artifacts and Terminology of Event-Based Systems

The single item window is particularly suitable for *stateless* operators, which only consider one event at a time and do not need to store the state of previous items. In *tumbling window* queries, new windows are only created if there is currently no other open window (i.e., $|W| = 0$). With a *sliding window* query, a new window is always opened if $s(w)$ holds. Finally, in a *landmark window* query, once a window has been opened, it remains open indefinitely (until the end of the stream is reached or the query is explicitly closed). In contrast to the growing window which binds some values over the entire stream, landmark windows consider different portions of the stream over time. More specialized characterizations of window types (e.g., time-based or count-based) are discussed in [261].

Event processing agents (EPAs) are software modules that consume events, process them, and output new events. Their behavior is defined by an *event processing language*, i.e., a high-level language, for instance based on an SQL dialect. *Stream processing*, *stream computing*, and *complex event processing* (CEP) are synonyms for performing computations with event streams as input. The behavior of the processing is define by an *event processing network* (EPN), a directed graph consisting of EPAs (as vertices) and event channels (as edges); the latter define the flow of events. While an EPN solely defines the logical connection between EPAs, the physical deployment is achieved by mapping EPAs to concrete computing *nodes*. A *data stream management system* (DSMS) is a software system whose main responsibility is the execution of one or multiple EPNs. It has responsibilities comparable to a traditional database management

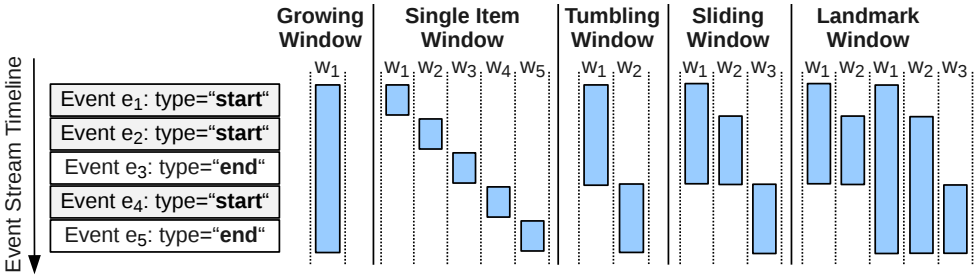


Figure 2.2: Different Types of Event Stream Query Windows (based on [39])

system (DBMS), with some notable differences, as shown in Table 2.1.

DBMS	**DSMS**
Persistent relations	Transient streams
One-time queries	Continuous queries
Random access	Sequential access
Optimized access plans can be derived	Unpredictable data characteristics and arrival patterns

Table 2.1: Comparison of DBMS and DSMS

The main challenges for stream processing are founded in the variability, unpredictability, burstiness, and volume of stream rates and data characteristics. Further aspects like changing processing objectives, e.g., higher quality of data requirements, and changing environmental circumstances, e.g., increased cost of computing resources, reinforce the challenge. Cloud computing, in particular the Infrastructure as a Service (IaaS) layer [224, 372] (see Section 2.3), provide a new level of flexibility in resource management, which provides a solid basis for implementing highly elastic stream computing.

2.2 Service-Based Applications

Service-Oriented Computing (SOC) has become a prevalent paradigm for creating loosely coupled distributed applications and workflows. Services, the main building blocks for SOC, constitute individual computing units, made available in a computer network using standardized interface description and message exchange [96, 258]. Service-Based Application (SBA) denotes an application that is built primarily by means of services, typically by composing a multitude of services into a business process or workflow. SOA forms the architectural underpinning for SOC. Conceptually, SOA involves three main actors: 1) *service providers* implement services and make them available at a certain location (endpoint) in the network; 2) *service registries* store information about services, and providers can publish their services in such registries; 3) *service consumers* discover (find) services by querying a service registry, bind to the obtained service references and execute the services' operations. This model of three collaborating actors is often referred to as the *SOA triangle* [231].

Today, *Web Services* [364] represent the most common way of implementing SOAs. The WS technology stack is defined by a plurality of specifications, most notably Web Services Description Language (WSDL) for description of the service interface, XML Schema Definition (XSD) for defining input and output data types, and Simple Object Access Protocol (SOAP) as message exchange protocol. SOAP defines an XML-based message format, split up into a body which contains the actual invocation data, and an optional header for additional data and cross-cutting concerns like addressing information, security tokens, or coordination/transaction contexts. The feature richness of SOAP-based Web services tends to make their execution engines complex and heavy-weight. The Representational State Transfer (REST) paradigm has been proposed as a more light-weight alternative [262]. With so-called RESTful services, no predefined message format is required, each service is treated as a resource addressable via a unique Uniform Resource Locator (URL), and the service operations are aligned with the standard operations in the Hypertext Transfer Protocol (HTTP) [104] (GET, POST, PUT, DELETE, etc).

2.2.1 Web Services Business Process Execution Language

The Web Services Business Process Execution Language (WS-BPEL) [246] is a standardized language for defining Web service based workflows. The XML-based language provides constructs for invocations to external Web services, manipulation of input and output data, as well as typical workflow constructs for controlling data flow and control flow (e.g., loops, branches, or parallel flows).

Figure 2.3 illustrates an exemplary travel itinerary process. The process receives as input the target city and desired date, and returns available flights, hotel rooms, as well as visa information for the destination country. For brevity, the process follows a simple sequential procedure and represents only a subset of the activities and control flow structures available in WS-BPEL, namely *receive*, *assign*, *flow*, *invoke*, and *reply*. Variable names start with a dollar sign ($) and sub-elements of a variable are accessible via XML Path Language (XPath) [362] expressions. The service invocation *getCountry* determines the country of the given city, *getLocations* looks up the user's current geographical coordinates (e.g., using Global Positioning System (GPS) on a mobile device), which are used to find available flights (*getFlights*). The process additionally invokes *getVisaInfo* and *getHotels*, which can be done in parallel (indicated by the enclosing *flow* activity). Input and output values of service invocations are manipulated using *assign* activities. Finally, the process returns the collected results to the caller via a *reply* activity.

WS-BPEL's combination of high level of abstraction on the one hand (external service calls make up the largest part of most WS-BPEL processes) and computational universality on the other hand (BPEL is *Turing complete* [337]) has led many researchers to study various aspects of this language and the associated execution model. The research efforts range from dynamic monitoring [20] and adaptation [233], to analysis of termination and reachability [256], to (unit) testing [87, 196, 218] of WS-BPEL processes. Testing of (single) Web services is essentially limited to black-box techniques on the interface level [26, 371]. In contrast, WS-BPEL exposes (part of) the processing logic and hence allows to apply advanced techniques such as data flow based testing [222], or group testing [332], amongst others.

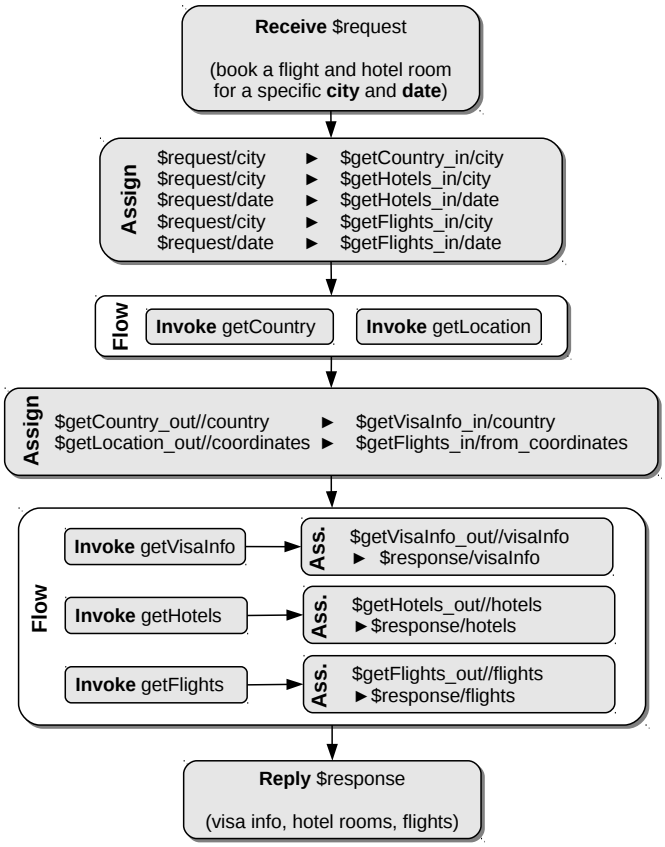


Figure 2.3: Simple WS-BPEL Example Process (Travel Itinerary Planning)

2.2.2 Dynamic Service Selection and Binding

A central concept for SBAs is the clear separation between interface contracts (e.g., defined using WSDL) and concrete service implementations. Hence, SBAs typically distinguish between *abstract services* which define the design-time capabilities, and *concrete services* which provide the actual runtime implementation.

In many industries, the technical interfaces of these abstract services are nowadays governed by industry standards, specified by bodies such as the *TeleManagement Forum*[1] (TMF), the *Association for Retail Technology Standards*[2] (ARTS) or the *International Air Transport Association*[3] (IATA). Standardized interfaces facilitate the integration of services provided by different business partners into a single SBA. Additionally, as oftentimes a multitude of potential providers are offering implementations of the same standardized interfaces, SBAs are enabled to dynamically switch providers at runtime, i.e., dynamically select the most suitable implementation of a given standardized interface based on fluid business requirements.

[1] `http://www.tmforum.org/`
[2] `http://www.nrf-arts.org/`
[3] `http://www.iata.org/`

Figure 2.4 illustrates the generic application model with dynamic binding, which we utilize throughout this thesis. For simplification, the terms *service* (operation-centric, possibly with human involvement) and *data source* (data-centric, typically with a purely technical query interface) are used interchangeably. Abstract services and abstract data sources define the interfaces and schemata, respectively, whereas their concrete counterparts represent the technical endpoints available to the application. The *candidacy assignment* determines which concrete services map to which abstract services. Moreover, the model captures *data flow* or *control flow* defined over pairs of abstract services. Finally, a concrete runtime *instantiation* in this generic application model defines *bindings* between abstract and concrete services. This model can be directly mapped to various technologies, importantly WS-BPEL and WS-Aggregation, as will be discussed in more detail in Section 4.3.3.

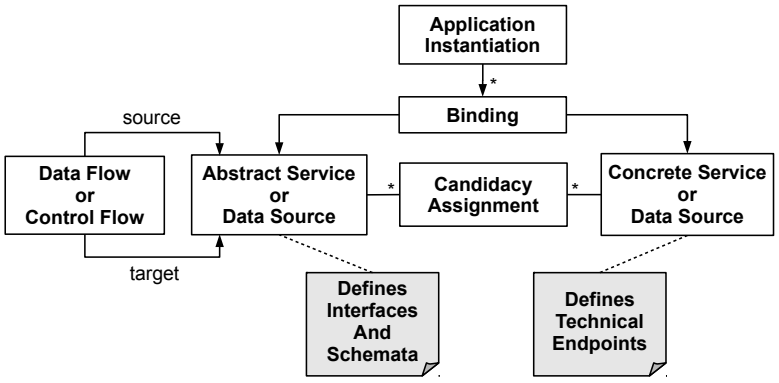


Figure 2.4: Generic Application Model With Dynamic Service Binding

Dynamic service selection and binding has received considerable attention from the services research community in the past. In a slightly different context, the concept of dynamic binding has been applied in object-oriented programming [99], where methods are dynamically looked up to allow sub-classing for specialized implementations. The early work by Casati et al. [58] applies dynamic binding to composite Web services and introduces the *eFlow* framework. In the Vienna Runtime Environment for Service-oriented Computing (VRESCo) [140, 230], service bindings are parameterized with queries to evaluate the current QoS of services, and applications automatically bind to the best service instance available. Several other works have also addressed dynamic binding or build their contributions on top of this concept [52, 84, 174].

Unfortunately, practice has shown that standardized interfaces alone do not guarantee compatibility of services originating from different partners. Many industry standards are prone to underspecification, while others simply allow multiple alternative (and incompatible) implementations to co-exist. Consequently, there are practical cases, where SBAs, which should work correctly in the abstract, fail to function because of unexpected incompatibilities of service implementations chosen at runtime. Tsai et al. [331, 332] have proposed the concept of *group testing* to validate the correctness of different combinations of concrete services. This thesis extends the concept of group testing and proposes an advanced data-flow based testing approach to identify service integration issues and data incompatibilities (see Chapter 4).

2.3 Cloud Computing

Cloud Computing (CC) [10, 48, 224, 341] is an emerging trend in the computing industry, which focuses on multi-tenant virtualized service and resource provisioning. The National Institute of Standards and Technology (NIST) defines CC as follows:

> » *Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction.* « *[224]*

The core motivation behind CC is to leverage economies of scale [10] by consolidating computing resources into large data centers, thereby achieving better utilization and reducing overall energy consumption and maintenance costs. In fact, Cloud Computing covers multiple conceptual levels and involves a variety of different technological underpinnings, hence the term refers to a business concept rather than a concrete technology. In recent years, a general, abstracted picture of the CC technology stack has emerged, involving three core layers: Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS) [224, 372]. IaaS in a narrow sense mostly refers to computational resources, hence the infrastructure layer is often further sub-divided into additional services such as storage or communications (cf. Figure 2.5).

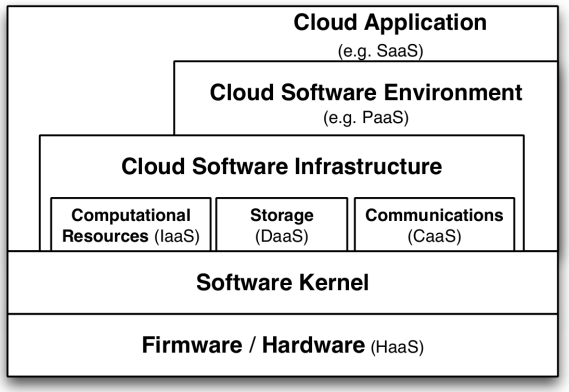


Figure 2.5: Layers of the Cloud Computing Stack (taken from [372])

The IaaS layer allows dynamic provisioning of low-level computing resources, such as computing machines, storage disks, communication links, network devices, etc. Amazon's Elastic Compute Cloud (EC2)[4] was among the first public IaaS offerings and remains a dominant player in this market sector today. Virtualization technologies like *Xen* [22], *VMWare ESX* [346] or *KVM* [173] allow to run multiple VMs on a single host, effectively isolating each VM's kernel, filesystem, network stack, and other parts of the OS. Since the aforementioned virtualization techniques are considered rather heavy-weight, a more light-weight alternative with increased performance is container-based OS virtualization [307], where VMs typically execute within

[4] `http://aws.amazon.com/ec2/`

the same kernel and filesystem as the host's OS. Popular implementations of container-based virtualization include *Virtuozzo*, *Linux VServer*, and *Linux Containers* (LXC).

The PaaS layer builds on the underlying infrastructure and provides a programmable software environment, which is used to host and manage multiple applications within a common platform [184]. Examples of popular PaaS providers to date are *Salesforce*, *Google App Engine*, or *Heroku*. Proponents of PaaS argue that development of applications in online cloud platforms increases productivity and reduces software costs [184]. Compared to traditional software engineering approaches and in-house hosting, developers building on PaaS are relieved from the costly configuration and maintenance of the middleware software which hosts their applications. Additionally, PaaS providers are able to apply suitable application monitoring and platform customizations, in order to optimize the deployment topology and utilization of the underlying infrastructure.

At the topmost layer, SaaS provides software services and applications to the end users. The users directly interact with the SaaS and are typically not aware of the underlying layers, in particular the infrastructure and platform on which the applications are running. A popular example for SaaS is Netflix[5], an online video broadcasting service which, at the time of writing, is among the biggest users of Amazon's cloud infrastructure[6][7], making up for more than 30% of the total Internet Service Provider (ISP) traffic in the United States [241].

2.3.1 Elastic Computing

The advanced resource allocation and virtualization capabilities provided by the Cloud have fostered a new trend towards elastic computing [91]. The term elasticity is well-understood in physics (e.g., in material sciences [11]) and economics (e.g., price elasticity [325]), among other fields. In general terms, elasticity denotes the (potential) change of some variables in response to the change of other variables. In the computing domain, an Elastic System (ES) is a system which dynamically adjusts key properties in response to external stimuli, i.e., user requests. Examples of such elasticity properties are number of used resources (resource elasticity), cost of operation (cost elasticity), or QoS (quality elasticity) [91]. In the scope of this thesis, resource elasticity is the primary focus of interest. Section 3.4 discusses how the WS-Aggregation platform optimizes runtime configurations by elastically acquiring and releasing computing nodes.

Elasticity, if not properly designed, may result in harmful system behaviors or unexpected costs [175]. Exposing an ES to operations which change its elasticity state can cause subtle modifications, which may become, in the worst case, uncontrollable or irreversible. For example, an ES may start to acquire resources in an uncontrolled way, it may oscillate between alternative allocations of resources, it may be unable to scale back to its initial configuration, it may fail to allocate resources on time to provide consistent quality of service (QoS), and more [162]. Reliable provisioning of elastic applications hence requires a rigorous engineering and validation approach. Our initial work on systematic testing of elastic computing systems can be found in [108–110], but these results are out of scope and not discussed in detail within this thesis.

[5] `http://www.netflix.com`

[6] `http://www.zdnet.com/the-biggest-cloud-app-of-all-netflix-7000014298/` (accessed 2013-12-15)

[7] `http://aws.amazon.com/solutions/case-studies/netflix/` (accessed 2013-12-15)

2.3.2 Automated Resource Provisioning – DevOps and Infrastructure as Code

The ever-increasing importance of elasticity for Cloud applications creates a strong demand for flexible deployment of infrastructure and middleware components, allowing to adaptively scale applications up and down based on QoS characteristics. A common impediment to this demand for flexible deployments is the well-known tension between software developers and operators: the former are constantly pressured to deliver elastic and highly dynamic applications to satisfy QoS- and cost-related goals, whereas the latter must keep production systems stable at all times. Not surprisingly, operators are reluctant to accept changes and tend to consume new code slower than developers would like.

In order to repeatedly deploy middleware and applications to the production environment, operations teams typically rely on automation logic. As new application releases become available, this logic may need to be revisited to accommodate new requirements imposed on the production infrastructure. Since automation logic is traditionally not developed following the same rigor of software engineering used by application developers (e.g., modularity, re-usability), automations tend to never achieve the same level of maturity and quality, incurring an increased risk of compromising the stability of the deployments.

This state-of-affairs has been fueling the adoption of *DevOps* [150, 204, 288] practices to bridge the gap between developers and operators. One of the pillars of DevOps is the notion of *Infrastructure as Code* (IaC) [150, 240], which facilitates the development of automation logic for deploying, configuring, and upgrading inter-related middleware components following key principles in software engineering. IaC automations are expected to be repeatable by design, bringing the system to a *desired state* starting from any arbitrary state. The notion of *idempotence* has been identified as the foundation for repeatable, robust automations [46,74]. To realize this model, state-of-the-art IaC tools, such as Chef [253] and Puppet [269], provide developers with abstractions to express automation steps as idempotent units of work. To illustrate the execution model of IaC automations, a brief introduction to Chef is provided in the following.

```
1  directory "/ws-aggregation" do
2    owner "root"
3    group "root"
4    mode 0755
5    action :create
6  end
7  package "tomcat6" do
8    action :install
9  end
10 service "tomcat6" do
11   action [:start, :enable]
12 end
```

Listing 2.1: Declarative Chef Recipe

```
1  bash "build php" do
2    cwd Config[:file_cache_path]
3    code <<-EOF
4
5  tar -zxvf php-#{version}.tar.gz
6  cd php-#{version}
7  ./configure #{options}
8  make && make install
9
10 EOF
11   not_if "which php"
12 end
```

Listing 2.2: Imperative Chef Recipe

In Chef terminology, automation logic is written as *recipes*, and a *cookbook* packages related recipes. Following a declarative paradigm, recipes describe a series of *resources* that should be in a particular state. Listing 2.1 shows a sample recipe for the following desired state: directory "/ws-aggregation" exists with the specified permissions; package "tomcat6" is installed; OS service "tomcat6" runs and is configured to start at boot time.

Each resource type (e.g., `package`) is implemented by platform-dependent providers that properly configure the associated resource instances. Chef ensures the implementation of resource providers is idempotent. Thus, even if our sample recipe is executed multiple times, it will not fail trying to create an existing directory. These declarative, idempotent abstractions provide a uniform mechanism for repeatable execution. Repeatability is essential, as recipes are run periodically to override out-of-band changes, i.e., prevent drifts from the desired state. In other words, a recipe is expected to continuously make the system converge to the desired state.

Supporting the most commonly occurring configuration tasks, Chef currently provides more than 20 declarative resource types whose underlying implementation guarantees idempotent and repeatable execution. However, given the complexity of certain tasks that operators need to automate, the available declarative resource types may not provide enough expressiveness. Hence, Chef also supports traditional imperative scripting embedded in script resource types such as *bash* (shell scripts) or *ruby_block* (Ruby code). Listing 2.2 illustrates an excerpt from a recipe that installs and configures PHP (taken from Opscode [255]).

This recipe excerpt shows the common scenario of installing software from source code—unpack, compile, install. The imperative shell statements are in the *code* block (lines 5–8). As an attempt to encourage idempotence even for arbitrary scripts, Chef gives users statements such as `not_if` (line 11) and `only_if` to indicate conditional execution. In our sample, PHP will not be compiled and installed if it is already present in the system. Blindly re-executing those steps could cause the script to fail; thus, checking if the steps are needed is paramount to avoid errors upon multiple recipe runs triggered by Chef.

2.4 Basic Terminology of Dependability – Faults, Errors, Failures

The seminal work by Avizienis et al. [13] defines the core terminology related to dependability and security, which is central to the work in this thesis and hence briefly revisited here. A fundamental understanding of these basic concepts is a key prerequisite for further discussion of the contributions achieved within this thesis.

The functionality provided by a certain system or application is denoted service[8]. A service is delivered by a service provider and consumed by a service user. Services are delivered at different levels of granularity; the service provider and user, respectively, can for instance be a software component that offers a certain functionality for another component, or an entire enterprise information system that performs computations and queries on behalf of a human operator. The system boundary where service delivery happens is denoted service interface. The state of the provider is defined by the internal state (not perceivable by the user) and the external state (perceivable at the service interface). Under perfect conditions, the system delivers *correct service*, i.e., it does what it is intended to do, as described by its functional specification. A service **failure** is *"an event that occurs when the delivered service deviates from correct service"* [13]. Given the definition of correct service, a failure means that the service does not comply with the functional specification. A service can also be seen as a sequence of transitions

[8]Note that the term *service* is also used as a shortcut for an invokable Web service [364] throughout this thesis. Hence, depending on the context, *service* refers either to the provided functionality itself, or to the entity providing the functionality.

in the external state of the system, hence a service failure means that one or more external states of the system deviate from the correct service state. This deviation in state is denoted an **error**. Finally, the cause of an error is denoted a **fault**.

Faults can have various manifestations and characteristics. First, the origin and occurrence of a fault can be either internal or external. For instance, an incorrectly encoded algorithm in a software system is considered an internal fault, whereas an inappropriate interaction of a user with the system is considered an external fault. Second, faults can be either *active* or *dormant*: a fault is active if it causes an error (i.e., external state that deviates from the correct state), otherwise the fault is dormant. For a more comprehensive discussion of fault manifestations and characteristics the interested reader is referred to Avizienis et al. [13].

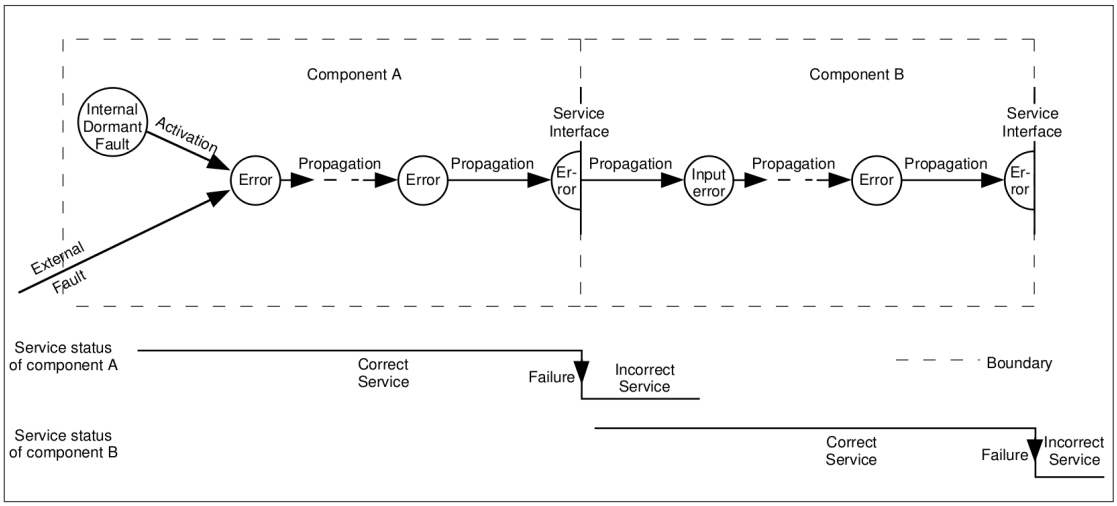


Figure 2.6: Fault Activation, Error Propagation, and Service Failure (taken from [13])

The process of transforming a dormant fault into an active fault, usually caused by applying a specific input to a component which contains the dormant fault, is denoted *fault activation*. Errors, on the other hand, do not possess this dimension of capability: an error either exists (i.e., is currently reflected in the system's state) or does not exist. However, *errors* can be *propagated* and successively transformed into other errors. Figure 2.6 (taken from [13]) illustrates the relationships between faults, errors, and failures, and exemplifies fault activation and error propagation based on two components A and B which are connected via a service interface. The dormant fault in Component A, in combination with an external fault, gets activated and leads to an error in this component. This error gets propagated, possibly multiple times, until it reaches the service interface connected to Component B, where the propagated error becomes an input error. The propagation may potentially continue to the next boundary (interface) of Component B. Figure 2.6 also illustrates that, from an external point of view, the components deliver correct service up to the point where the error propagates through the service interface, because only at that point is the external state influenced by the error.

The concepts and terminology discussed in this section are directly applicable to the class of applications studied in this thesis (cf. Section 1.1.1). Internal and dormant faults might be

encoded in the implementation of a data processing platform or in the definition of applications running on the platform. Since the business logic of data processing workflows typically depends on external data, external faults (e.g., data incompatibilities) need to be anticipated and dealt with. Moreover, given that the platform middleware is deployed in a Cloud environment with complex reconfigurations to achieve elasticity, dormant faults may only be revealed if a particular deployment topology or resource allocation gets activated.

2.4.1 Fault Management and Fault Tolerance

As outlined in Section 2.4, software faults exist in various manifestations and pose a threat to applications' reliability. Since the existence of faults can almost never be ruled out entirely in complex software stacks, systematic dealing with faults becomes a necessity. A distributed system which is capable of anticipating and gracefully handling the occurrence of faults that result in failures (e.g., by hiding the failures from other processes) is denoted *fault tolerant* [321].

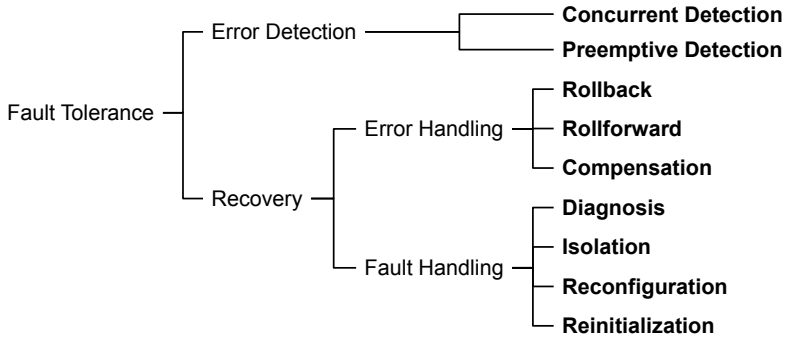


Figure 2.7: Fault Tolerance Techniques (based on [13])

Fault tolerance techniques in distributed systems are commonly divided into different techniques [13, 321], which are illustrated in Figure 2.7 and briefly discussed in the following. The two main categories of fault tolerance are *error detection* (i.e., identifying the presence of errors), and *recovery*, which is defined as transforming *"a system state that contains one or more errors and (possibly) faults into a state without detected errors and without faults that can be activated again"* [13]. Error detection can be either *concurrent* (happening during service delivery) or *preemptive* (suspends service delivery and checks the system for dormant faults). Recovery can be further split up into *error handling* and *fault handling*. The former eliminates errors that are currently present in the system state, and the latter prevents faults from getting (re-)activated. For error handling, we distinguish *rollback* (resetting the system back to a state without errors), *rollforward* (fixing the system by directly removing error states), and *compensation* (redundant processing which allows to mask the error). For the category of fault handling, four inter-related techniques are distinguished: *diagnosis* identifies the causes of errors; *isolation* turns a fault into the dormant state by excluding it (physically or logically) from service delivery; *reconfiguration* modifies task assignments to move tasks away from faulty components; and *reinitialization* records the new configuration (e.g., after isolation or reconfiguration) to permanently block eliminated faults.

Randell postulates that fault tolerance *"must be based on the provision of useful redundancy"* [274]. The concept of redundancy has various manifestations, tackling different kinds of potential faults. The straight-forward method for redundancy is *replication of programs*. Assuming a program fails due to an error in the underlying operating system or hardware platform, the computation performed by the failed program can be re-executed by a new instantiation of the same program, possibly on a different platform. However, simple replication is not sufficient if the program itself is faulty, i.e., exhibits an implementation that leads to a error or produces incorrect results. To overcome this issue, *N-version programming* [64] achieves redundancy by running multiple implementation variants of a program. The basic conjecture is that independent programming efforts can *"reduce the probability of identical software faults occurring in two or more versions of the program"* [64]. A related approach is *agreement* [263], where multiple entities have to reach a consensus for a given problem. Agreement is often used in situations where either of the participating entities is assumed to be faulty or not trustworthy.

Various aspects of faults in event-based data processing systems have been studied. Paradis and Han [260], as well as Ruiz et al. [281], survey fault management in Wireless Sensor Networks (WSNs). Data delivery in such networks is inherently prone to faults, because communication links are fragile (e.g., unreachability or congestion) and sensor nodes may fail (due to, e.g., depletion of batteries or physical destruction). In this thesis, hardware faults are not the primary concern, yet many other fault issues in WSNs also apply to the domain of data-centric and event-based applications in the Cloud. As part of the contributions in this thesis, we provide a systematic study and taxonomy of faults in EBSs (see Section 3.2).

2.5 Software Testing

Modern computing systems, and in particular applications in the Cloud, are often burdened with stringent reliability requirements. Software testing [30, 238] is therefore an integral part of the software development and quality assurance process. Because testing is a very broad field that involves a multitude of different aspects, there is no *one-size-fits-all* definition of the field. Myers et al. define software testing as *"[. . .] the process of executing a program with the intent of finding errors"*. This definition emphasizes the execution of the System Under Test (SUT) and the core goal of finding programming errors (also denoted bugs). In contrast, Beizer [30] argues that *"Bug prevention is testing's first goal"*, with the rationale that a prevented bug is better than a detected one.

Different types of testing methodologies and techniques have emerged in the past. In this section we briefly discuss the core concepts which are relevant within the scope of this thesis. Bertolino [34] provides a comprehensive overview and future research directions in software testing research. In her roadmap, Bertolino speaks of (past) *achievements*, (current and medium-term) *challenges*, and (long-term or unrealistic) *dreams*. The achievements include well-studied fields such as protocol testing, reliability testing, or object-oriented testing. Ongoing challenges, among others, are domain-specific test approaches, on-line testing, and costs of testing. One of the dreams that the testing community has in mind is 100% automatic testing – entirely eliminating the need for human involvement in terms of test case selection, manual test refinement etc. Another dream suggested by Bertolino is test-based modeling; it refers to

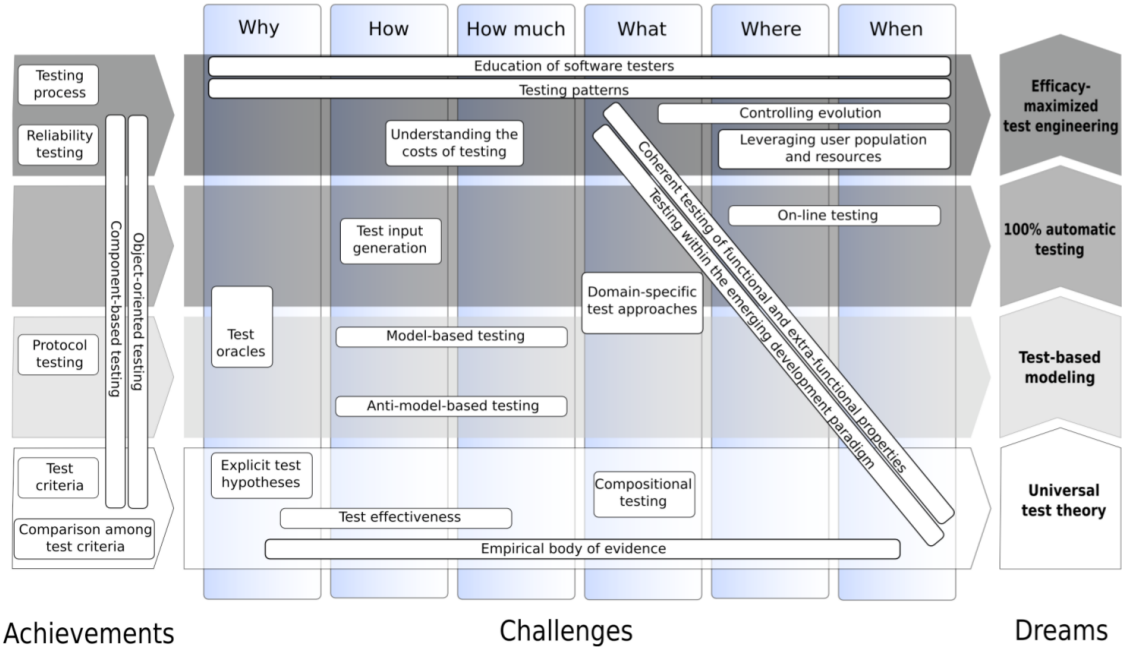


Figure 2.8: Software Testing Research Roadmap (taken from [34])

a software development approach in which we would abandon the current practice of finding well-suited testing approaches by exploiting existing modeling frameworks (e.g., Unified Modeling Language (UML) [249]), and instead reverse the approach by focusing on how to define the model such that the software can be effectively tested.

The following subsections discuss three core concepts: model-based testing (Section 2.5.1), combinatorial testing (Section 2.5.2), as well as test coverage and adequacy (Section 2.5.3).

2.5.1 Model-Based Testing

In practice, the complexity of software systems and the environment they operate in are often infeasible to capture entirely. Hence, efficient test methods require suitable abstractions which are able to accurately capture the crucial aspects of the SUT. Model-based testing (MBT) [334, 335] is the general framework which provides the corresponding methodological underpinnings. In principle, MBT defines a multi-step process for generating and executing test suites from abstracted system models [334]. The first step of the process is to establish the test requirements and derive the system model. The second step is to define the test selection criteria, which aim at finding test cases that are likely to detect failures (plus their underlying faults). Step three is to formalize the test selection criteria and materialize them into a test case specification. In step four, the test case specification and the system model are combined to generate a suite of executable test cases. The test cases, encoded as test scripts, are executed in step five, and finally a test report is created by analyzing the test results and comparing expected with actual outputs provided by the test cases.

While the general process is clear-cut, existing MBT approaches largely vary when it comes to details. The proposed test coverage criteria are highly domain-specific, and the system models range from, for instance, logic-, graph-, or time-based models, to stochastic and deterministic models, to discrete and continuous models, and more. Utting et al. [335] establish a comprehensive taxonomy for MBT which lists general characteristics of MBT approaches along the dimensions test model, test generation, and test execution.

2.5.2 Combinatorial Testing

Detecting an implementation error (*bug*) in a software system is often compared to finding the proverbial needle in the haystack. Typically, bugs are only revealed if the SUT follows a certain execution path combined with a specific sequence of internal state transitions. Execution paths and state transitions are often influenced by parameter input values, which are externally provided by human users or machines interacting with the system. Combinatorial Testing (CT) [71, 242] tackles the problem of systematically probing the SUT with combinations of parameter values, with the aim of triggering (activating) faults. It should be noted that the term *input parameter* can be understood in various ways. In a narrow sense, input parameters are the values that are passed along with a function call, for instance the API parameters of a software module. In a wider sense, input parameters also constitute features and configuration options of the SUT. For instance, in the context of SBAs with dynamic binding, the runtime mapping of abstract to concrete services can be considered as a parameter in the combinatorial space.

The problem of *combinatorial explosion* leads to the fact that practically no technique can feasibly provide complete (exhaustive) testing [178, 242]. Hence, CT attempts to select a subset of the parameter combinations to keep test suites easy to manage and execute. Domain knowledge about the SUT (e.g., identifying parameters that certainly never trigger faults [242]) can help to keep the test sets relatively small, while still covering a large portion of the relevant paths and state transitions. One of the most common forms of CT is n-*way* testing [71], where n is the number of parameters for which groupwise combinations should be covered. For instance, given a set of parameters P, 2-way (or pairwise) testing generates test inputs such that each pair $(p_1, p_2) \in P \times P$ of parameters (with $p_1 \neq p_2$) is covered with all possible value combinations.

The comprehensive survey by Nie and Leung [242] categorizes CT into eight core sub-areas: modeling (interrelations of parameter values), test case generation, constraints (avoiding invalid test cases in generation), failure characterization and diagnosis, application of CT, test case prioritization, metric (measuring the effectiveness of fault detection), and evaluation (quantifying the degree of improvement of software quality). In their study, by far the largest part of surveyed papers is on test case generation, i.e., efficiently computing a (near-)minimal test suite (also denoted *covering array*) for given requirements. Finding optimal covering arrays is an NP-hard computational problem [242] and various researchers have proposed different solutions, ranging from random approaches and greedy algorithms to search heuristics and mathematical methods.

2.5.3 Test Coverage and Adequacy

One of the key issues in software testing is to objectively measure the quality of tests and their capability to show that a program is in fact error-free if the tests execute successfully [379].

Assessing the adequacy of tests has been a long-studied research problem over the previous decades, largely influenced by the seminal early work of Goodenough and Gerhart [118].

A central principle for measuring test adequacy is test coverage, i.e., the extent to which a certain aspect of the SUT is covered by the test set. Coverage can be defined over various criteria; for example, statement coverage denotes the percentage of the total code instructions that were executed by the tests, and branch coverage is the ratio of executed code branches to the total number of branches. In general, test data adequacy criteria are defined as a function $C : P \times S \times T \rightarrow [0, 1]$, where P is a set of programs, S is a set of specifications of the software, and $T = 2^D$ is the powerset of test cases over the program inputs D [379]. A value of $C(p, s, t) = r$ means that the adequacy of testing p against the specification s is of degree r according to the criterion C. For example, if C_{CT} refers to combinatorial input coverage, then $C_{CT}(p, s, t) = 0.9$ iff the tests in t cover 90% of the input parameter combinations in D.

Intuitively, one could assume that increasing test coverage leads to increased reliability, but the exact relationship is difficult to establish, since it depends on the context of the concrete test setting. Malaiya et al. [211] study the relations between testing time, coverage, and reliability. Their evaluation compares the growth rate of different coverage metrics with rising number of test cases or identified defects. One of the key findings is that, event under 100% test coverage for a given metric, the SUT need not necessarily be free of defects.

Offutt et al. [252] have devised general coverage criteria for graph-based specifications, for instance STGs. Four testing goals (with increasing level of coverage) are distinguished to derive test cases from state-based specifications. *Transition coverage* means that each transition in the graph should be covered by (at least) one test case. *Full predicate coverage* requires that one test case should exist for each clause on each transition predicate. The *transition-pair coverage* goal extends on transition coverage and ensures that for each state node all combinations of incoming and outgoing transitions are tested. Finally, *full sequence coverage* requires that each possible (and relevant) execution path is tested, usually constrained by applying domain knowledge to ensure a finite set of tests [252].

While the generic coverage criteria by Offutt et al. [252] are simple to apply and verify, they lack the expressive power to model complex cases of test coverage. Hong et al. [130] developed a more comprehensive calculus of test coverage and generation based on temporal logic. Their approach utilizes the branching-time temporal logic named Computational Tree Logic (CTL). The SUT is modeled using an Extended Finite State Machine (EFSM) which captures the internal system states and state transitions. Formulas in CTL are constructed from propositions, modal operators, and (temporal) quantifiers, which allow to impose conditions (or constraints) over the states of the EFSM. For instance, for a given state s and the terminal state $exit$, the formula $\mathbf{EF}(s \wedge \mathbf{EF}\ exit)$ expresses a program execution which covers the state s (possibly covering other states before and after) and eventually reaches the state $exit$. A finite sequence of states that satisfies a certain condition formula is denoted a *witness* of this condition, and this state sequence forms the materialization of a *test case*. A test suite is defined as a set of test cases, that is, a set of *"finite executions of the EFSM such that for every formula, the test suite includes a finite execution which is a witness for the formula"* [130]. The problem of minimal test generation aims at finding the smallest test suite covering all formulas, according to either 1) the number of test sequences, or 2) the total length of all test sequences.

CHAPTER 3

WS-Aggregation: Reliable Event-Based Data Processing with Elastic Runtime Adaptation

3.1 Introduction

In recent years, academia and industry have increasingly focused on EBS and Complex Event Processing (CEP) [97] for Internet-scale data processing and publish-subscribe content delivery. Today's massive and continuous information flow requires techniques to efficiently handle large amounts of data, e.g., in areas such as financial computing, online analytical processing (OLAP), wireless and pervasive computing, or sensor networks [237]. In most of these application areas, filtering and combining related information from different event sources is crucial for deriving knowledge and generating added value on top of the underlying (raw) data. Platforms that are specialized in continuously querying data from event streams face difficult challenges, particularly with respect to performance and reliability. Evidently, continuous queries that consider a window of past events (e.g., moving average of historical stock prices in a financial computing application) require some sort of buffering to keep the relevant events in memory. State-of-the-art query engines are able to optimize this buffer size and to drop events from the buffer which are no more needed (e.g., [195]). However, a topic that is less covered in literature is how to optimize resource usage for a system with multiple continuous queries executing concurrently. Moreover, to allow for reliable event processing, fault management and fault tolerance are key concerns for EBS. In order to implement a fault-tolerant eventing platform, precise knowledge about the type and nature of faults is vital.

This chapter presents WS-Aggregation as the first main contribution of this thesis. WS-Aggregation is an elastic distributed platform for event-based processing of Web services and data. The platform allows multiple parallel users to perform continuous queries over heterogeneous data sources (cf. Section 1.1.1). The core contributions discussed in this chapter are fourfold.

- One of the key challenges addressed in the context of WS-Aggregation is fault management and fault tolerance. We perform an extensive literature review which reveals that there is a plethora of different types of event processing platforms which exhibit similar characteristics and potential faults. We categorize EBSs into five core sub-areas, and derive a common system model that covers the key aspects of all sub-areas. Based on the common system model, different fault classes and fault sources are surveyed and discussed. Details follow in Section 3.2.

- Second, the query model and basic processing of WS-Aggregation are introduced in Section 3.3. WS-Aggregation employs a collaborative computing approach where data processing tasks (i.e., queries) are split into multiple sub-tasks, which are then assigned to one or more aggregator nodes. If a query involves input data from two or more data sources, each of the inputs may be handled by a different aggregator. Data dependencies between these inputs are specified declaratively using a specilized query language, termed WAQL. At runtime, the platform collects all data, orchestrates the data flow between the aggregator nodes, and asynchronously returns result updates to the clients.

- Throughout various experiments we observed that query distribution and placement of processing elements has a considerable impact on the performance of the framework. To study these effects, three main aspects are taken into account. First, computing nodes have resource limits, and in times of peak loads the system needs to be able to adapt and reorganize. Second, complex queries over event streams require buffering of a certain amount of past events, and the required memory should be kept at a minimum. Finally, if the tasks assigned to collaborating nodes contain inter-dependencies, possibly a lot of network communication overhead takes place between the aggregator nodes. We propose an approach which considers all of the above mentioned points and seeks to optimize the system configuration. Details are discussed in Section 3.4.

- To support the elasticity features of WS-Aggregation, frequent deployment and configuration of new computing nodes is required on the infrastructure layer. If a new VM is started and integrated into the platform, the software stack needs to be configured: downloading of software updates, installation of the middleware platform code, starting of services, writing values into a service registry, etc. The automation scripts implementing this type of infrastructure management should be resilient to faults, reliably making the system converge to the desired state. If the automation fails at some point (e.g., due to temporary network downtime while downloading software updates) it needs to be able to repeat the failed steps and finish the procedure successfully. In Section 3.5 we devise a generic approach for functional testing of automation scripts, applicable to WS-Aggregation as well as other systems where reliable VM deployment and configuration is crucial.

In the remainder of this chapter, we first discuss the concepts of the aforementioned core contributions in Sections 3.2, 3.3, 3.4, and 3.5, respectively. Selected implementation details are then highlighted in Section 3.6. Different aspects of the approach and the implemented prototype are evaluated in Section 3.7. Finally, the approach is put into perspective with the related work in Section 3.8.

3.2 Common Model and Fault Taxonomy for Event-Based Systems

In this section, we establish a model for EBSs, which serves as the basis for discussion. The goal of the model is to capture specifics of different variants of EBSs. In particular, the model is derived from various previous publications in five sub-areas, which we briefly discuss in Section 3.2.1. The challenge in defining such a reference model is the tradeoff of including as many aspects and different viewpoints as possible, while at the same time keeping the complexity at a minimum, providing the necessary level of generality.

3.2.1 Specialized Types of Event-Based Systems

Within this work we have evaluated various publications related to EBSs (published as books, journal or conference contributions) and have extracted five main sub-areas of this field.

The first principal field involves event-based information dissemination [210,267] and event-driven programming models [79], including content filtering [167], message oriented middleware [19], notification services [56], message-passing systems [95], or tuple spaces [116]. Examples where this field plays a key role are emergency control systems, real-time collaboration, or active databases [220]. Here we collectively refer to this class of systems as **Event-Driven Interaction Paradigms (EDIP)**. The event-based interaction mode has also gained importance in software engineering, e.g., for graphical user interface software [323], in the context of specification of system architecture [209, 212], or under the term *implicit invocation* [112].

The second field is **Event Stream Processing (ESP)** [15, 17, 312], which deals with continuous queries over data streams, often with a focus on high-frequency events and scalability. Example applications are financial services [21], stock trading platforms [1], or network traffic management [17].

The third field is **Complex Event Processing (CEP)** [97, 207, 208], which covers the core concepts of causal event histories, event patterns, event filtering and event aggregation [208]. Application areas include geospatial event processing [97], RFID-based product monitoring [347], or online fraud detection [97, 297].

The fourth main area of interest is event-driven **Wireless Sensor Networks (WSN)** [5, 47, 281], including applications of ubiquitous computing [326], intrusion detection [177], or monitoring of environment (temperature) data [281]. Among the key problems in this field are energy-efficiency, event routing or data aggregation [176].

Concept [236]	**EDIP**	**ESP**	**CEP**	**WSN**	**EDBPM**
event	notification	tuple	event	datum	invocation
producer	publisher	source	producer	sensor	service/ activity
consumer	subscriber	sink	consumer	sink	
event processing	service	operator	agent	node	
channel	channel	stream	event bus	link	service bus
derived event	merged message	event pattern	complex event	fused information	composite service

Table 3.1: Different Terminology for Similar Concepts

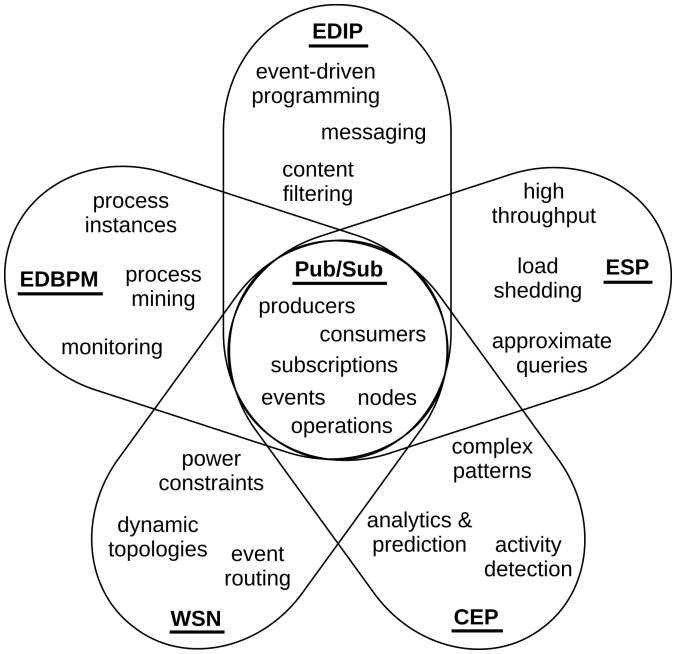


Figure 3.1: Sub-Areas of Event-Based Systems

The fifth area is **Event-Driven Business Process Management (EDBPM)** [339, 355], with popular examples including road tolling systems [97], order and shipment management [280], or workflows of telecommunication providers [229].

Figure 3.1 contains five elliptic shapes that represent the five mentioned event processing areas. The intersection of the shapes is a circle containing concepts of the **Publish/Subscribe (Pub/Sub)** interaction scheme [98, 103]. Pub/Sub elements are commonly found in all areas (e.g., events, producers, consumers), although sometimes slightly different terminology is used in each field. Moreover, there are characteristics and challenges specific to each area, which are printed in the non-intersecting parts of the figure. To further illustrate some of the commonalities among the sub-areas, Table 3.1 lists typical terminology referring to similar concepts.

The concept of events plays a role in many other research areas that are rather remotely related to our choice, for instance interconnect solutions for large scale multiprocessor systems [68], discrete event systems [272], or events/signals in operating systems [128]. Although partly applicable to our approach, these areas are not explicitly included in this discussion.

3.2.2 Description of the Common Model for Event-Based Systems

In the following we discuss the artifacts of the common model for EBSs. The model we propose is based on previous work that has tried to capture common features of event-based systems and applications, most notably in [236, 304, 351]. The fundamental concepts, artifacts and entities of the model (printed in bold) are introduced below. The core elements and relationships of the model are depicted in the form of a UML class diagram in Figure 3.2.

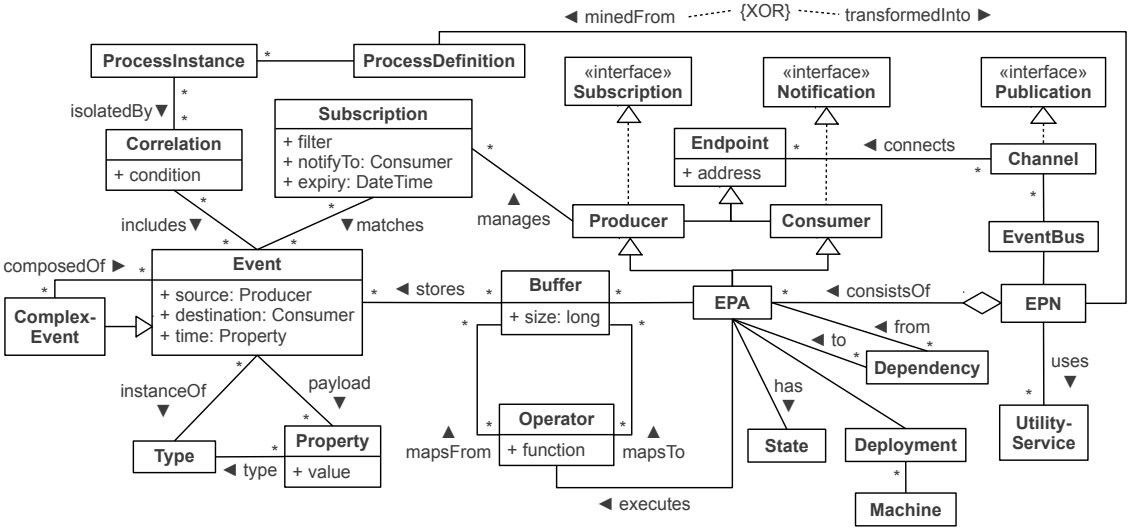


Figure 3.2: Excerpt of the Common Model for Event-Based Systems

- An **event** is "*anything significant that happens or is contemplated as happening*" [236]. Research distinguishes **simple events** (events which do not represent a set of other events) and **complex events** (events that span multiple other events). An event has a certain **type** and an arbitrary number of **properties**. Standard properties are the **source**, **destination** and the **time** at which the event occurred. Additionally, the event may be associated with application-specific properties, denoted as event **payload**.
- Events are typically sent from a **producer** (often termed **source** in sensor networks) to one or more **consumers** (**sink** in sensor networks) through a communication **channel**. Between the event producer(s) and end-consumer(s) there is an **event processing network** (EPN) consisting of **event processing agents** (EPAs) connected by event channels. The channel may be a direct connection (i.e., the producer is able to contact the consumer(s) directly), or it may be part of an **event bus** whose responsibility is to deliver the events accordingly. Typically, additional functionalities and responsibilities are attributed to the event bus, such as storage, registry or access control services [236]. For simplicity, we summarize these artifacts under the term **utility services**, which the EPN interacts with. Another critical utility service is **time synchronization**, required for correct and consistent timestamping of events. In stream processing, EPNs and data flow dependencies between EPAs are often modeled as a directed **acyclic** graph (DAG). However, in general the connections between EPAs may also be **cyclic**.
- Event consumers register a **subscription** with a producer to receive **notifications** about certain events. The subscription **filter** specifies which events are of interest, based on the type and/or payload. Upon receipt of an event, consumers **react** by performing an (application-dependent) **task**, e.g., operating a physical switch, invoking an electronic service, initiating a business process etc. Subscriptions must be unambiguously **identifiable**, and consumers should be **addressable**. If notifications cannot be directly **pushed** to consumers (via a **publishing interface**), they are requested in **pull** mode.

- **Correlation** [177, 280] means identifying and grouping events that are logically linked together (from the application point of view). A correlation **condition** is a function that determines for a set of events whether or not they are correlated. In the simplest case, **correlation properties** are defined on two or more event types and all instances of these types for which the properties **match** are considered correlated [280]. Event **isolation** aims at dividing the total set of events into (usually disjoint) sets of correlated events (a typical example is isolation of business process instances).
- The logic of an event-based business process is specified in a **process definition** (e.g., using a graphical notation or some formalization like *Petri nets*) of which multiple **instances** can exist. The process definition is either known a priori and **transformed** into an EPN, or the definition is learned by monitoring the EPN using **process mining** techniques [280, 339].
- Communication between EPAs often (although not necessarily) happens **asynchronously** and **non-blocking**. Events that cannot be processed immediately are put to a **buffer** (or queue). Buffers are subject to physical resource restrictions and therefore usually have a **limited size** (length).
- Event **routing** may happen either **statically** (according to pre-defined routing tables) or **dynamically** (decision based on the characteristics or capabilities of available EPAs). Dynamic (and resource efficient) routing is a key problem in WSNs [47, 176].
- EPAs process a set of input events and output zero or more (possibly new) events. Thereby, three stages are distinguished [236]: **pattern matching**, **processing** and **emission**. An EPA can be at the same time event consumer and producer, and hence it implements the corresponding interfaces. Incoming events are put to one or possibly multiple (depending on the implementation) **input buffers**. The **operator** of an EPA is responsible for generating events on the **output buffers** and is specified via a **function** that maps from inputs to desired outputs (see details in Section 3.2.3). Additionally, each EPA is associated with a **state** that reflects its current allocation of variables, memory, registers, and other state information. We assume that the state also includes the model instances that are relevant for the EPA, so that the EPA can **reflect** on itself at runtime (e.g., which subscriptions it manages). Apart from the general model entities stored in the state, the notion of state is highly application specific. Hence, we only assume this generic container and make no further restrictions about its representation, in order to keep things simple.
- EPAs are **deployed** on physical **machines** (or computing nodes), and one machine can **host** multiple EPAs. If the deployment changes and the responsibility of hosting an EPA (including its state) is transferred from one machine to another, we speak of **migration** of this EPA.

3.2.3 Modeling the Operation of Event Processing Agents

Because a large part of an EPN's functionality is encoded in the EPAs, they are a primary source for potential faults. Hence, we discuss the internal structure and *modus operandi* of EPAs, as envisioned in the common model, in more detail.

Figure 3.3 depicts a UML component diagram with an EPA connected to two channels. The state associated with the EPA is maintained by a **state manager**, which may be implemented as an actual state machine (automaton) or some other mathematical model. The **input router** is responsible for directing the events received via the **notification interface** to one or more

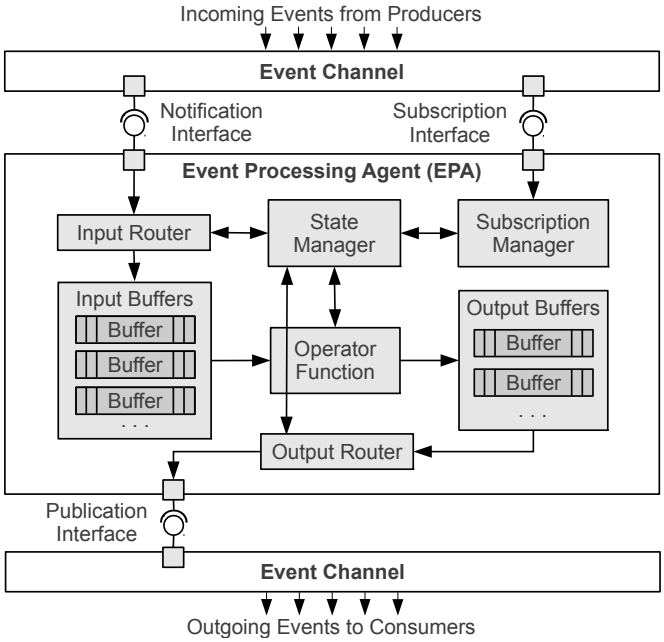


Figure 3.3: Internal Structure and Functionality of Event Processing Agents

input buffers. The EPA also receives requests from the channel via the **subscription interface**. A **subscription manager** is responsible for maintaining subscriptions. More specifically, when a new request comes in, a new *Subscription* model element is instantiated and stored in the state manager. The **output router** is responsible for forwarding events from the **output buffer** to subscribed consumers via the channel's **publication interface**. The figure contains five exemplary buffers for illustration, but the dots ("...") indicate that more buffers are conceivable.

3.2.3.1 Input-Output Operator Function

The **operator function** mediates between the input/output buffers and the state manager. Formally, this function is defined as follows. Let *B* denote the set of buffers (for both input and output), *E* the set of all events, *T* the temporal domain (all possible event timestamps), and *S* the set of possible states. The current content of a buffer is expressed as $\mathbb{N} \rightarrow E$, mapping from the numeric slot position (index) within the buffer to an event ($\varnothing$, the "empty event", is also part of E and hence $\varnothing$ signifies an empty buffer slot).

$$\phi : [T \rightarrow E]^n \rightarrow [T \rightarrow E]^m; n, m \in \mathbb{N}^+ \tag{3.1}$$

A generic operator function ϕ, denoted *stream transformer*, has been previously proposed in [312]. This function (printed in Equation 3.1) takes a set of timestamp/event pairs as input and outputs a new set of timestamp/event pairs.

$$op : (BE \times S) \rightarrow (BE \times S) \tag{3.2}$$

We propose to use a more specific input-output operator function that considers the input/output buffers as well as the state of the EPA. Let $BE := \mathcal{P}(B \times (\mathbb{N} \to E))$ denote the buffered events, i.e., the buffer allocation at any point in time ($\mathcal{P}(x)$ denotes the *powerset* of x). We then define the operator function as printed in Equation 3.2. The input of the op function is a subset of the buffers (BE) together with their content, plus the current state (S) of the EPA. The output of op is a new content assignment for a subset of the buffers (BE), plus a new assignment for the EPA state (S).

Upon arrival, new events are added to the corresponding input buffer(s) (existing events are shifted by one position), and op is executed. The approach provides the expressive power of *event-condition-action* (ECA) rules [220], a popular method for CEP specification. Our formalization is also in line with the stream transformer definition in [312], since the timestamp is accessible as an event property in our model.

We illustrate the operator function with a small example in Equation 3.3. The example considers an EPA which receives numeric event values on two input buffers (ib_1, ib_2) and determines whether the sum of the values is positive or negative. Let $val(e)$ denote the numeric payload of an event $e \in E$. The EPA has two output buffers (ob_1, ob_2). If the sum is positive an event e_{pos} is put to ob_1, otherwise the new event e_{neg} on ob_2 indicates that the sum is negative. Additionally, the EPA can be in a state $INACTIVE$, in which case no output is generated at all.

$$op(\{(ib_1, \{1 \mapsto e_1\}), (ib_2, \{1 \mapsto e_2\})\}, s) := \begin{cases} (\emptyset, s) & \text{if } s = INACTIVE \\ (\{(ob_1, \{1 \mapsto e_{pos}\}), s) & \text{else if } val(e_1) + val(e_2) \geq 0 \\ (\{(ob_2, \{1 \mapsto e_{neg}\}), s) & \text{else if } val(e_1) + val(e_2) < 0 \end{cases} \tag{3.3}$$

3.2.3.2 Event Routing Functions

Particularly in WSNs, dynamic event routing is a key challenge [47, 176]. Our model, therefore, contains two router components, which reflect that routing is decoupled from the input/output operator. Let P denote the set of event producers and C the set of event consumers. The input router is defined via a function $in : (E \times P \times S) \to \mathcal{P}(B)$, which determines for an incoming event $e \in E$, producer $p \in P$, and current state $s \in S$ the subset of input buffers $ib \subseteq B$ to which e is added. Conversely, the output router is defined via a function $out : (E \times \mathcal{P}(B) \times S) \to \mathcal{P}(C)$, which defines for an outgoing event $e \in E$, coming from a subset of the output buffers $ob \subseteq B$, and a current state $s \in S$ the consumers $c \subseteq C$ to which e will be forwarded.

3.2.4 Fault Taxonomy

Based on the model defined in Section 3.2.2, we now establish the fault taxonomy for distributed event based systems (EBSs).

3.2.4.1 Taxonomy Dimensions and Terminology

The influential work by Avizienis et al. [13] studies concepts for dependable and secure computing, and provides a detailed taxonomy framework with multiple fault dimensions (see Section 2.4). Despite the high level of detail, their work still provides general applicability and

builds a solid basis for extended approaches. For instance, Chan et al. [59] have presented a fault taxonomy for Web Service Compositiqons that closely builds on the classification in [13].

Firstly, we recite the most relevant terminology from [13] (see also Section 2.4), put into the context of EBSs. A system delivers **correct service** if it provides the desired functionality, which includes the functionality of end producers and consumers as well as the EPN that mediates between them. A **failure** occurs when the system *"does not comply with the functional specification, or because this specification did not adequately describe the system function"* [13]. As an example, consider a system that analyzes a stream of stock market events and is supposed to indicate if the price of a stock "rises significantly", but no event is generated, even after the price has risen ten consecutive times. Depending on the system function, this behavior may either be a failure caused by incorrect processing, or the failure may be rooted in the fact that the system detects stock rises with a high statistical confidence of 99.9%, whereas the specification (implicitly) assumed a 95% confidence interval. When asking for the manifestation of a failure in the system, we say that a failure is caused by one or more states deviating from the correct service state. This deviation is denoted as **error**. The assumed cause of an error, either internal or external, is called a **fault**. In the stock price example, a possible error is that an EPA was unable to store new incoming events, and the probable fault that lead to this error is a buffer overflow. Note that not all faults cause an error and therefore lead to a system failure: *"A fault is **active** when it causes an error, otherwise it is **dormant**"* [13].

3.2.4.2 Fault Classes

In [13], 16 elementary fault classes are derived from eight basic viewpoints. We have identified 12 of these fault classes as highly relevant for our purpose. The *phase of creation or occurrence* distinguishes between faults that are introduced at development time or during operation (execution) of the system. *System boundaries* refers to the distinction whether a fault is caused internally within the system or caused by external input received at the service interface or from the environment. *Persistence* determines whether the fault is continuous our bounded in time (i.e., persistent or transient). The *dimension* indicates faults that affect (or originate in) either software or hardware. The *phenomenological cause* of a fault can be either rooted in natural phenomena (on which humans have limited or no influence) or in active human participation. The *capability* dimension acknowledges that some faults are introduced inadvertently (or accidentally), while other faults result from lack of professional competence, strategy or planning.

The other four fault classes in [13] make a distinction between *malicious* and *non-malicious faults* as well as *deliberate* and *non-deliberate faults*). These four latter types of faults are particularly important in the area of security, which is not the core focus here. Hence, our approach captures the technical manifestations of deliberate and malicious acts (e.g., an overloaded channel or buffer overflow caused by a denial-of-service attack), but our fault taxonomy does not explicitly distinguish purely security-related dimensions. In return, we add two important fault classes concerning the *level in solution stack*, namely *platform faults* versus *business logic faults*. The former class of faults has its roots in the implementation of the underlying event processing platform and may potentially affect all of the applications on top of it, whereas the latter fault class is tightly connected to the specific business applications deployed on the platform. Figure 3.4 contains a schematic view of the seven dimensions and 14 classes used in our fault taxonomy.

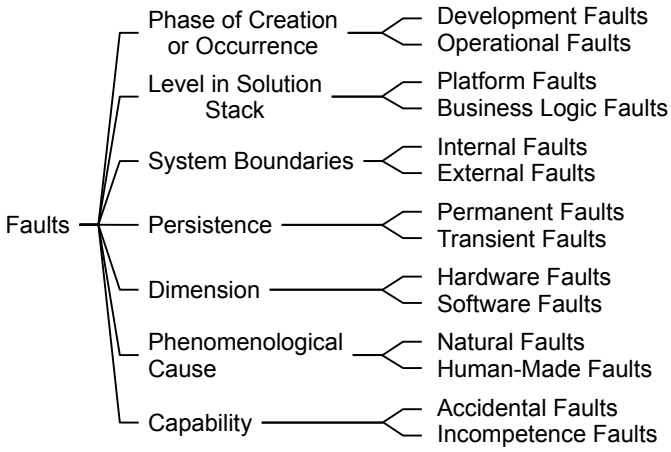


Figure 3.4: Elementary Fault Classes for Event-Based Systems (based on [13])

3.2.4.3 Fault Sources

Besides classes (types) of faults, our taxonomy also considers the sources of faults, i.e., the artifacts of the system which are potentially or positively responsible for causing the fault. Figure 3.5 depicts the six categories of fault sources, which have been extracted and compiled from earlier work on fault localization [12, 311] and root cause analysis [193]. Where applicable, the fault source description refers back to elements of the model discussed in Section 3.2.2.

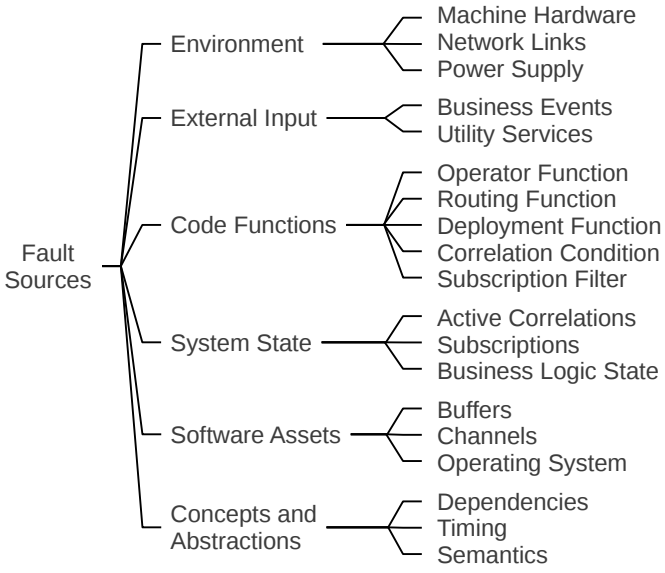


Figure 3.5: Fault Sources in Event-Based Systems

The *environment* refers to the physical platform on which the event-based system operates, i.e., machines, network links, and power supply. Power supply plays a key role, particularly if

the event data are only stored in volatile (non-persistent) memory, which is flushed in case of a power outage. *External input* is received from business events and utility services; both types of inputs can potentially influence the reliable operation of the system. The fault source category named *code functions* refers to processing logic encoded as operators, state transitions, queries, algorithms, etc. Evidently, code functions are at the core of event processing and can introduce faults into the system. The *system state* includes both the model-related configuration of the system (e.g., active correlations, subscriptions) and application-specific business logic state (e.g., state $INACTIVE$ in the example in Section 3.2.3.1). *Software assets* are self-contained components that the system builds on, which are known to operate well under controlled conditions but fail under certain circumstances (e.g., buffer overflow of a channel, or kernel error of an operating system). The *concepts and abstractions* category captures additional aspects that play a role in the processing, such as timing aspects or dependencies. The third part of this category is denoted semantics; for instance, if an event-processing platform performs dynamic re-configuration and re-deployment, it must be ensured that the newly configured system is semantically equivalent to the previous state and still fulfills all requirements.

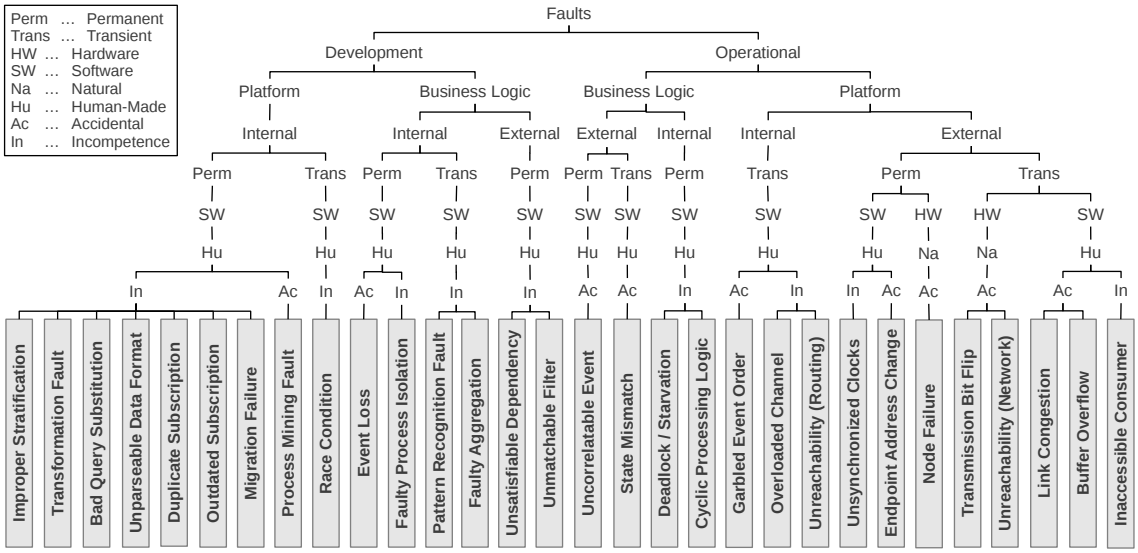


Figure 3.6: Taxonomy Tree for Faults in Event-Based Systems

3.2.5 Discussion of Identified Faults

In Figure 3.6, we identify and classify 30 fundamental faults (without claim of completeness), which are the result of literature review guided by combining different fault classes and model artifacts. The figure depicts a classification tree in which the leaf nodes are fault descriptions, and each level represents a pair of fault classes from Figure 3.4.

Moreover, Figure 3.7 lists for each of the 30 faults the sources which are likely involved in the creation of the fault. The matrix contains fault sources on the left-hand side and fault examples on the top. Each intersection of fault type and matching fault example is marked with

a red dot. Note that a fault may potentially be rooted in more than one fault source. We have attributed each fault to one of the core sub-areas of event-based systems that we identified in Section 3.2.1 (see "Main Areas Affected" at the bottom of Figure 3.7). This is of course a strong simplification – in fact, the distinction is not always clear and the areas partly overlap, i.e., faults may play a role in more than one area.

3.2.5.1 Faults in Publish/Subscribe Systems

Publish/Subscribe is the basis for many types of event-based systems. The main challenges are related to management of subscriptions, filtering, and dissemination of information [98]. On the network level, multicast transmission between one producer and multiple subscribers is often achieved using point-to-point communication primitives. This may easily lead to **Overloaded Channels**, which is the first (leftmost) fault example in Figure 3.7. For illustration purposes we briefly explain the fault classes for this fault (see Figure 3.6). An overloaded channel is an *operational fault*, because its phase of creation/occurrence is attributed rather to execution time than to development time. It is not particularly business logic-specific, but a general *platform fault*. The fault usually comes into existence by *external* influences or inputs. Although it remains active for a while, its presence is bounded in time (until the traffic drops to an acceptable level) and hence we note that it is *transient*. Moreover, the fault is a *software fault* and it is *human-made*. Concerning the fault capability, a channel overload is considered *accidental*, especially if the underlying network is commonly used by multiple external systems.

As subscriptions have an expiry time in our model, it may occur that an event producer garbage collects an expired subscription, while one of the consumers still retains a reference and attempts to modify it. This type of fault is denoted **Outdated Subscription** and can be compared to a *dangling pointer* in programming languages. This fault is generally attributed to the platform and the development phase, because outdated subscription references should be properly garbage collected. Analogously, a subscription should be kept alive as long as there exist any references to it. Similarly, a **Duplicate Subscription** is a development fault because one would expect that the platform takes care of eliminating such duplicates. Concerning delivery of event messages in push mode, an **Inaccessible Consumer** occurs if the endpoint address of the consumer is not available. This can have several reasons, e.g., the consumer process has no privileges to open a listening network socket, or network packets are dropped due to firewall rules, etc.

3.2.5.2 Faults in Event-Driven Interactions

Related to the Inaccessible Consumer fault in Pub/Sub is the problem of **Endpoint Address Change**, which we attributed to the EDIP category. This fault happens when the logical consumer of an event subscription is moved to a different physical machine or connection without updating its references. A main difference is that an address change is usually permanent, whereas an inaccessible consumer may be a permanent or temporary (transient) fault.

Another weak point in EDIP (and also Pub/Sub systems) is the evaluation of filters. Popular implementation variants include topic-based filtering and content-based filtering [98]. While topic-based filtering is usually straight-forward, content-based filtering is more dynamic and flexible. For instance, in [167] a content-based Pub/Sub system with routing based on *Bloom*

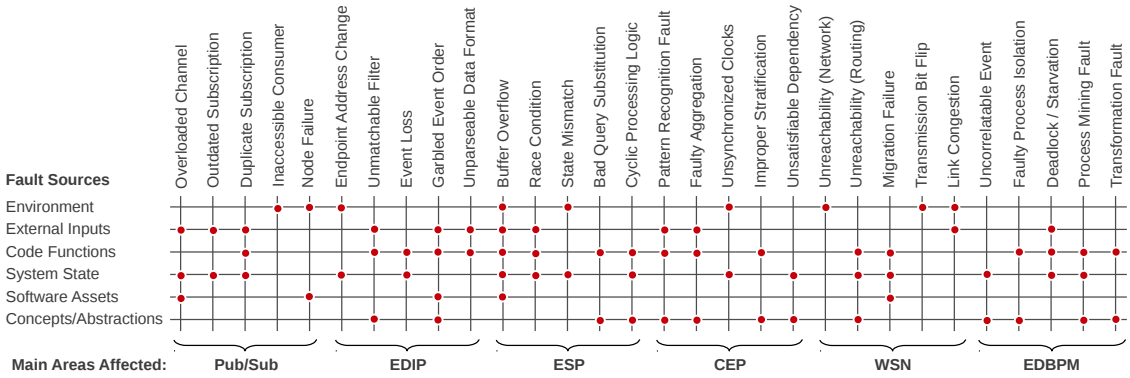


Figure 3.7: Factors of Influence Responsible for Different Faults

filters has been proposed. This leaves room for optimizations in the dissemination procedure, but also opens possibilities for the introduction of new faults. We collectively refer to this class of problems related to evaluation of subscription filters as **Unmatchable Filter**.

Event Loss is a potential problem in most event-based systems. In contrast to load shedding [1, 81], where strategies for deliberate dropping of events are put into action, event loss in CEP can also happen unintentionally. It is particularly relevant when an event is expected to be handed from a source to a destination EPA and the global behavior depends on the operator of each involved EPA along the path. A problem closely related to event loss is **Garbled Event Order**. For instance, the *Aurora* platform implements delayed processing in periods of high load, which may result in the situation that already emitted results need to be revised [1]. This feature is prone to errors and it is crucial to restore the correct message order. Applied to our model, garbled event order may also happen if the input router or output router functions are not correctly synchronized.

In some cases, event-driven interaction faults are less sophisticated but simply caused by an **Unparseable Data Format** of the underlying events. There exists a variety of different character encodings and document markup languages used to encode the payload (event properties), and the communication will most likely fail if the event producer and consumer do not utilize the same standards.

3.2.5.3 Faults in Event Stream Processing

We now focus on faults that are related to the main challenges in event stream processing (ESP) systems. An obvious and often studied source of problems lies with high frequency and volume of arriving data items. Resource limitations of the machines on which EPAs are deployed allow only a certain amount of events (or tuples, as often denoted in stream processing) to be stored and processed per time unit, depending on the complexity of the performed operation. A **Buffer Overflow** occurs if an EPA cannot allocate new memory to buffer an incoming event. This problem has been intensively studied and different solutions were proposed. One solution is load shedding [1, 81], where exactness/accuracy in the operator function is sacrificed for continuous availability by heuristically dropping (or delaying) events that are (deemed to be) of less

importance. Partial fault tolerance under high load and the effects of bursty tuple loss have been intensively studied based on a fault injection framework in [163].

A **Node Failure** occurs when the event processing system experiences hardware failures or faults due to “heisenbugs" in the underlying operating system or platform [303]. This type of fault also plays a key role in WSNs, for instance caused by depletion of batteries. We assume that the effects of a node failure (e.g., loss of state information) are permanent and cause the EPA operator to stop functioning immediately. From the viewpoint of the event processing system, a node failure happens accidentally and cannot be directly attributed to human incompetence. Evidently, node failure poses a key problem to any type of system that is supposed to operate reliably. EPA replication is a commonly used technique for reliable operation in the presence of node failures, which is intensively studied in [303].

Another key issue with parallel processing of events (in particular for ESP, but also in other areas) is that improperly synchronized code can cause **Race Conditions** when common resources (e.g., buffer memory) are accessed. Synchronization errors may lead to unexpected side effects and inconsistent states. These problems are often hard to debug and only revealed nondeterministically under high load.

Reliable reflection about the system state plays a key role for the operator and routing function of EPAs. If the actual state of the system is not accurately reflected in the saved state of an EPA, we speak of a **State Mismatch**. Two main reasons may be responsible for a state mismatch: either the output of an EPA’s operator function is faulty and generates a (locally) inaccurate state, or the state transfer of a global state between multiple EPAs is not transactionally safe.

Queries over streams that involve multiple nested operations are often decomposed and rewritten/optimized based on query plans to achieve high performance [1, 370]. The goal is to map the original query to a new query which has more desirable characteristics (e.g., better suited for distribution or avoids unnecessary execution steps), while at the same time preserving the exact semantics of the original query. If the latter condition does not hold, a **Bad Query Substitution** fault occurs that may have an effect on the system functionality.

Finally, if the processing graph defined by the dependencies within an EPN contains cycles, the system may get into a state in which events are “trapped” by getting forwarded indefinitely. We refer to faults of this type as **Cyclic Processing Logic**. A possible technique to avoid this fault is to attach metadata to event messages that circulate in the system, for instance a time-to-live counter that specifies how many times the event may be passed on to the next EPA.

3.2.5.4 Faults in Complex Event Processing

Complex Event Processing (CEP) is concerned with complex interactions and event patterns often spanning multiple sources. A complex event is one that is derived or aggregated from multiple other events. A frequently encountered problem are **Pattern Recognition Faults**. One manifestation of this fault is when a relevant pattern exists but is not detected. The other possibility is that a complex event (e.g., credit card fraud [297]) may be mistakenly assumed to exist although in fact it does not. Pattern recognition is often achieved using event automata [297] which define states and state transitions. In our model, event automata correspond to the EPA state combined with its operator function. Pattern recognition faults are hence rooted in the in-

correct handling of the EPA's state by this function. The general case of events being incorrectly combined into complex (or aggregate) events is denoted as **Faulty Aggregation**. Aggregation functions, including simple aggregates (count, sum, average, minimum, maximum, etc.) or mathematical functions over event sequences (e.g., cross-correlation coefficients), are a vital foundation for defining and executing CEP business logic [297, 347, 370]. As indicated in Figure 3.6, faulty aggregation usually affects the business logic (e.g., when combining multiple correlated "item" events into a single "package" event in a warehouse [347]).

Time-sensitive CEP queries and event composition often rely on synchronized time measurements among the producers, to retain either absolute time difference or relative order of events [197]. In our model, time is expressed as a special case of the EPA state, which is periodically changed (irrespective of event inputs) and shared among the EPAs. **Unsynchronized Clocks** are therefore a source of failure which may cause misbehavior in a CEP system. Synchronization is also a key issue in wireless sensor networks [5].

Stratification [181] is the process of splitting up the EPN dependency graph into independent sub-graphs, denoted as stratums, to achieve parallelism and early filtering of events. The semantics and dependencies of the processing graph must be retained and hence the non-trivial stratification algorithm in [181] is itself a potential source of faults. The effect of **Improper Stratification** could be for instance that events are filtered out too early, or that the wanted effect of load distribution is not achieved and one node or channel gets overloaded.

CEP systems may run into the problem of an **Unsatisfiable Dependency**, where an EPA is in a state in which it expects a certain event to arrive but this event cannot be delivered, e.g., caused by event loss or a cyclic dependency. Note that we have to make a subtle distinction here: the processing graph of an EPN *can* in fact be cyclic (e.g., an EPA may consume and process events that were emitted by itself); however, the causal dependency of correlated or complex events must not be cyclic. This fault should be considered with close attention, because a single unsatisfiable dependency can bring the entire system to a halt/deadlock (see also discussion of business process deadlocks in Section 3.2.5.6). Wherever possible, the platform should provide means for statically checking circular dependencies in the event processing business logic.

3.2.5.5 Faults in Sensor Networks

WSNs consist of a collection of densely deployed sensor nodes whose purpose is to sense and process information from the environment [5]. The key challenges intrinsic to WSNs arise from the circumstance that sensor nodes are limited in power, often prone to failures, and connected by unreliable communication channels. Moreover, the position of sensor nodes is not static and the topology of a WSN changes frequently. An adaptive and fault-tolerant routing mechanism is therefore required.

Figure 3.6 contains two fault cases that are related to **Unreachability**. Firstly, unreachability on the **Network** level means that a sink is unable to receive any messages from a source node, either because its communication link is down, the signal is noisy, or the node is out of range of any other nodes and hence there is no physical path from source to sink. If there is a possible path between two nodes, but a packet (or message) does not find its way to the receiver, we speak of a **Routing** related unreachability, also denoted path fault [327]. For instance, a possible reason may be that the routing algorithm partitions a set of network nodes into multiple routing

domains, which are connected by a single coordinator. If this coordinator fails to work properly, the sub-networks become disjoint and messages from one domain cannot reach another domain.

Since positions of nodes and topologies in WSNs can change frequently, tasks and node responsibilities are often assigned dynamically. If a task is migrated from one node to another, the operator logic as well as the EPA state need to be marshalled and transmitted over the network. The duration of transmission is a critical time window because during that time it must be ensured that no events are either lost or double-processed [141]. Moreover, all dependencies to other nodes need to be updated as soon as the task has been transferred. If this complex procedure does not complete transactionally safe, we speak of **Migration Failure**.

Hardware constraints play an important role in WSNs. Due to unreliable transmitters or noisy signals, it is possible that **Transmission Bit Flips** occur. A bit flip simply means that part of the data has changed its representation (e.g., from "0" to "1") during transmission and that neither side of the communication realized this error. Various error-detection codes (like checksums or parity bits) have been proposed to avoid transmission errors in WSNs, and there is an obvious tradeoff between degree of error-robustness and computational complexity [5].

Link Congestion occurs if network links operate beyond capacity, induced by high amounts of data or too many senders writing to the same medium. Media access control (MAC) techniques like *Carrier Sense Multiple Access* (CSMA) regulate the access to a commonly used transmission medium, but there is a practical limitation to the number of devices served by the same network [5]. A link congestion is different from an overloaded channel, because network links may be shared with external systems, whereas channels are considered an internal platform component.

3.2.5.6 Faults in Event-Driven BPM

Event-driven business process management (EDBPM) is concerned with utilizing events to steer the orchestration of workflows and services. EDBPM can be approached from different sides: the process definition is either known explicitly and can be transformed into an EPN (e.g., [194]), or the process has an implicit model that is discovered from event logs (also denoted process mining) [339], or the challenge is to measure the fit between event logs and the process model [279]. Algorithms in these areas are complex and often make probabilistic assumptions, and hence pose a potential source of faults. A **Process Mining Fault** implicates that the (probabilistic) assumptions in the algorithm to derive a process model are inaccurate, whereas a **Transformation Fault** denotes an incorrect mapping from the original process definition to tasks on the eventing platform.

When a process or sub-process is triggered by an event, the process engine needs to be able to correlate the event to previous or currently active process instances. An **Uncorrelatable Event** occurs if the process to which the event seems to belong either does not yet exist, or does not exist anymore (because it has been finished or forcibly terminated), or has never existed (caused by faulty correlation). Note that an event which triggers an entirely new instance (and is hence not in a correlation with a previous process) does not fall into this fault category, because that reflects a regular situation. Another correlation-related defect is **Faulty Process Isolation**, which means that a set of events representing a model process instance does not correspond to reality or the actual business case. Faulty isolation is particularly problematic in process mining,

because most algorithms that learn the structure of processes rely on the fact that the instances from which they learn are correct.

Process Deadlock and **Starvation** are defects related to the business logic which prevent the process from continuing its execution. In a deadlock the process arrives at a state in which the process definition allows neither termination nor moving into any successor state [336], for instance because it is waiting for a particular event. Starvation principally means that processes are competing for a resource and one process with less priority is discriminated against competitors. As an example, imagine two consumers c_1 and c_2 which are supposed to receive events of type t_1 with a fair distribution (e.g., strictly alternating order). If the event router happens to favor c_1, then c_2 may starve and wait forever or time out after not receiving an event for a while.

3.2.6 Relation of the Fault Taxonomy to Other Contributions

The fault taxonomy discussed in this section builds the basis for other contributions in this thesis. The query model of WS-Aggregation, discussed in Section 3.3.3, is capable of detecting and resolving *Unsatisfiable Dependencies* and *Cyclic Processing Logic*. In Section 3.4, we tackle in particular the issues of *Overloaded Channels*, *Buffer Overflow* and *Node Failure* by continuously monitoring and optimizing the placement of processing elements (i.e., EPAs) in the WS-Aggregation platform. *Migration Failure* is explicitly addressed in Section 3.6.3, where we introduce a transactional scheme for migration of event buffers and subscriptions. Additionally, if the migration involves deployment of a new VM host in the Cloud, Section 3.5 tackles reliable provisioning and configuration on the infrastructure level. In Chapter 4 we introduce our approach for functional testing of data-centric and event-based applications, which addresses integration issues and incompatibilities on the business logic level, such as *Unmatchable Filter*, *Uncorrelatable Event* and *Pattern Recognition Fault*.

3.3 Event-Based Continuous Queries in WS-Aggregation

This section introduces the query model and architecture of the WS-Aggregation platform. We first introduce an application scenario in Section 3.3.1 which illustrates the processing of interdependent event data streams and serves as the basis for further discussion. Next, we outline the system architecture of WS-Aggregation in Section 3.3.2. The query model is detailed in Section 3.3.3, and the approach for distributed query execution is discussed in Section 3.3.4. Section 3.3.5 briefly discusses the Cloud-based elasticity mechanism employed in the platform.

3.3.1 Application Scenario

To illustrate the query model of WS-Aggregation we consider a simple scenario from the financial computing domain, in which Web services provide live data about companies and stock prices. The aim is to combine the information in an XML document that is actively updated when the underlying data change. Figure 3.8 illustrates, on a high level of abstraction, how data and events are received and processed.

We distinguish three basic types of receiving data from the sources: (1) the *StockPrice* and *StockTrade* services allow to subscribe for certain events transmitted using WS-Eventing, (2) the

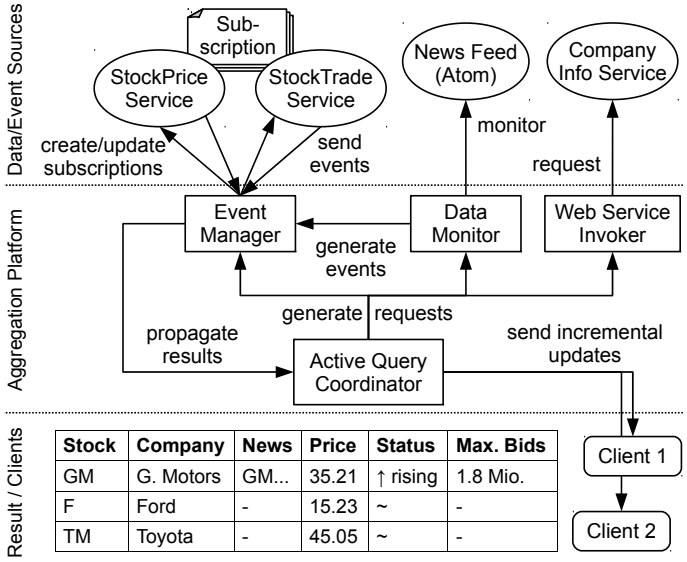


Figure 3.8: Event-Based Continuous Data Processing Scenario

News feed is regularly monitored for changes, (3) the *Company Info* service contains rarely changing, static data. The clients specify a query to receive aggregated data that are incrementally updated as the query executes. The aggregation platform mediates between the data providers and consumers, and coordinates the query execution. The required core components from a high-level perspective are as follows. An Event Manager (EM) maintains subscriptions with the target services and receives events. The Web Service Invoker (SI) is responsible for performing synchronous service requests, and a Data Monitor (DM) repeatedly retrieves data from the monitored resource and generates an event to report any changes. The collected results from EM, DM and SI are fed into the Active Query Coordinator (AQC), which updates all dependencies and generates new requests as needed. To the clients the platform appears as a single entity, but in fact the system is distributed over several computing nodes, be it for performance reasons or due to higher-level constraints (e.g., the StockPrice and StockTrade services should report to physically separated endpoints).

The resulting document contains a table with the current stock prices and general company information. Furthermore, the result indicates when a stock has three or more consecutive price rises, in which case the largest bid volume is displayed. If the platform detects that a stock has risen *and* traders are placing high volume bids (e.g., $\geq$ 1 million), the table should display live news about these companies.

3.3.1.1 Interdependent Event Streams

Figure 3.9 illustrates two sample event streams of the StockPrice and StockTrade services. Initially, a subscription for StockPrice exists, and the service continuously sends stock price events. When three consecutive rises are detected for *GM* (bold text), the Event Manager requests a new subscription with the StockTrade service to receive all *bids* and *asks* for *GM* placed on the mar-

ket. Finally, this subscription is destroyed when five consecutive ticks (also in bold text) are below the last price of the rising sequence (35.27).

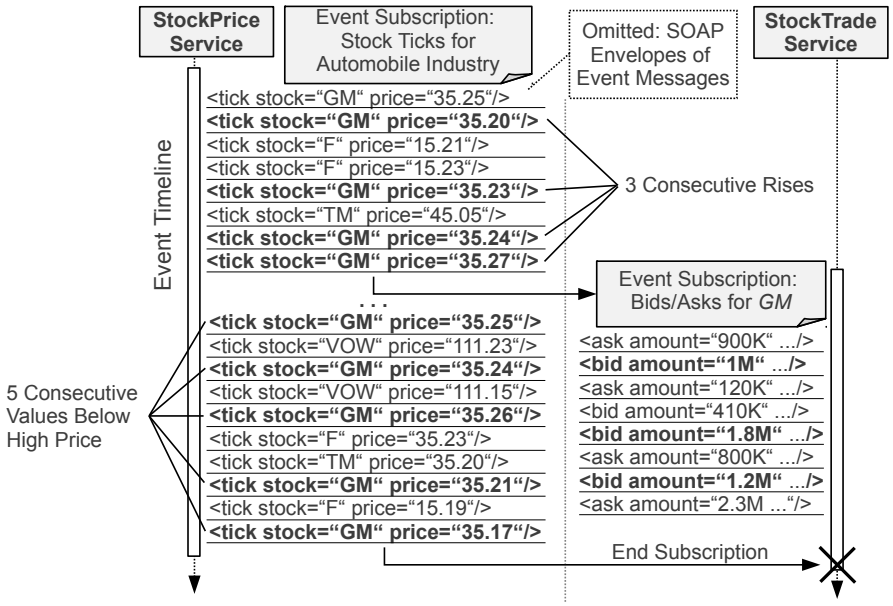


Figure 3.9: Sample Event Streams and Lifecycle of Event Subscriptions

3.3.2 System Architecture

The coarse-grained system architecture of WS-Aggregation is illustrated in Figure 3.10. The *gateway* service (G) acts as the single endpoint the clients communicate with. A variable number of *aggregator* nodes (A) serve incoming query requests. Each aggregator retrieves data from one or more target services ($s_1, \ldots, s_6$). The aggregators are depicted in a cloud because they are invisible to the clients and may be added to and removed from the system transparently. In basic processing mode, a query that spans multiple data services is handled by a single aggregator node. The advantage of this approach lies in its simplicity and the low internal coordination overhead. However, to optimize the internal performance characteristics, WS-Aggregation focuses on distributed processing for obtaining the final results. As illustrated in Figure 3.10, distributed aggregation makes use of a tree topology among aggregator nodes where the root node in the topology is denoted *master* aggregator, A_M. Each aggregator retrieves data from one or more target services and passes the collected, intermediate results on to its parent node.

WS-Aggregation builds on the SOA paradigm and focuses on the use of a service registry, which fosters a decoupled and flexible architecture. We utilize the VRESCo service registry [140,230] to publish the endpoint information of the gateway, aggregator and data services. Clients use the registry to discover the gateway instance, and the gateway finds all active aggregator nodes contained therein. New aggregator nodes can be seamlessly integrated and existing nodes can be taken off the system. The aggregators themselves determine the endpoints of the data services, for which purpose the registry allows to query services by feature. Faulty or non-

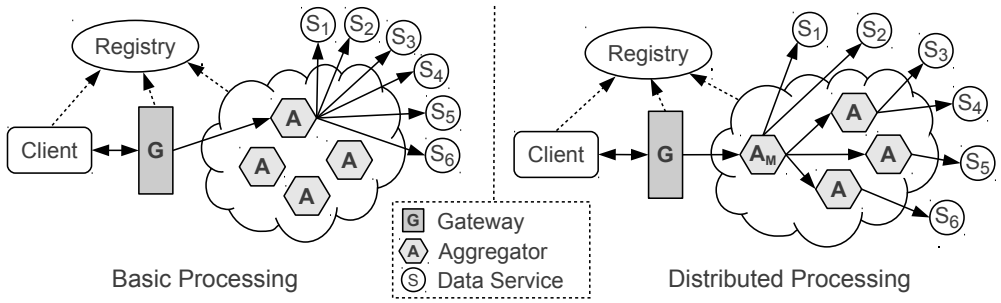


Figure 3.10: WS-Aggregation System Architecture

responsive aggregators are automatically unregistered when detected by the gateway or a partner aggregator. Furthermore, the VRESCo registry stores metadata such as QoS or physical location of services, which serves as the decision basis for optimized query distribution (see Section 3.4).

3.3.3 Query Model of WS-Aggregation

In the following we establish the model for distributed processing of event-based continuous queries that is applied in WS-Aggregation. The model builds the foundation for the concepts discussed in the remainder of this chapter.

Symbol	**Description**
$A = \{a_1, a_2, \ldots, a_n\}$	Set of deployed aggregator nodes.
$Q = \{q_1, q_2, \ldots, q_m\}$	Queries that are handled by the platform at some point in time.
$I = \{i_1, i_2, \ldots, i_k\}$	Set of all inputs over all queries.
$inputs : Q \rightarrow \mathcal{P}(I)$	Function that returns all inputs of a query.
$deps : Q \rightarrow \mathcal{P}(I \times I)$	Function that returns all data dependencies of a query.
$S = \{s_1, s_2, \ldots, s_l\}$	Data sources that emit events over which queries are executed.
$source : I \rightarrow S$	Function to determine the data source targeted by an input.
$query : I \rightarrow Q$	Function to determine the query an input belongs to.
$buf : A \rightarrow \mathcal{P}(S)$	Function to determine which data sources an aggregator buffers.

Table 3.2: Description of Symbols and Variables in Event-Based Query Model

Table 3.2 summarizes the symbols and variables that are used in the formalization. In our model, a number of aggregator nodes (A) are collectively responsible to execute multiple continuous user queries (Q). Each query processes one or more inputs (I) from external data sources (S). The function $inputs$ maps queries to inputs ($\mathcal{P}(I)$ denotes the power set of I), and the function $source$ returns the data source targeted by an input. The actual processing logic of the query is application specific and not directly relevant for our purpose. However, we consider that a query q may contain data dependencies among any two of its inputs

$i_x, i_y \in inputs(q), i_x \neq i_y$. A dependency $(i_x, i_y) \in deps(q)$ means that i_y can only be processed after certain data from i_x have been received, because the data are required either 1) by the request to initiate the event stream from the data source underlying i_y, or 2) by a *preprocessing* query that prepares (e.g., groups, filters, aggregates) the incoming events for i_y. Such dependencies are often seen in continuous queries over multiple data streams [17], where subscriptions are dynamically created (or destroyed) when a specific pattern or result is produced by the currently active streams. An example could be a sensor emitting temperature data in a smart home environment, which only gets activated as soon as another sensor emits an event that a person has entered the room.

Although we use the terms service and data source interchangeably, strictly speaking the notion of data source is narrower, because every entry in S is identified by a pair $(epr, filter)$, where epr is the *Endpoint Reference* [359] (location) of the service and the $filter$ expression determines which types of events should be returned. That is, different data sources may be accessed under the same service endpoint. The $filter$ may be empty, in which case events of all types are returned.

The reason for abstracting inputs from data sources is that different queries may require different data from one and the same source. As an example, assume a data source which every second emits an event with the market price of two stocks, and two queries which compute the *Pearson* correlation as well as the *Spearman* correlation of the historical prices. This means that each of the inputs needs to be processed (computed) separately, but the same underlying event buffer can be used for both inputs. We use the function buf to determine the data sources from which an aggregator "currently" (at some point in time) receives and buffers events.

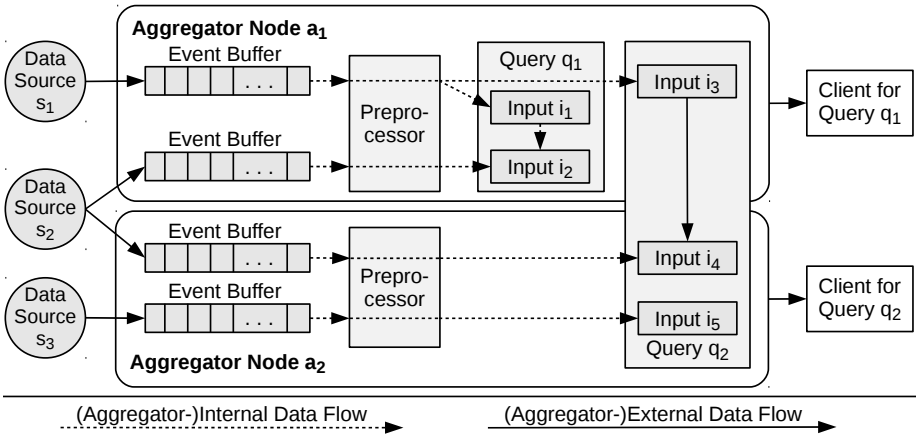


Figure 3.11: Illustrative Instantiation of the Model for Distributed Event-Based Queries

The key aspects of the processing model are illustrated in Figure 3.11, which depicts two aggregator nodes (a_1,a_2) executing two queries (q_1,q_2) consisting of five inputs (i_1,...,i_5) in total. The query execution happens in two steps: firstly, the incoming events are buffered and preprocessed to become the actual inputs (e.g., average value of previous stock prices), and secondly the inputs are joined and combined as defined in the query specification. Dashed lines in the figure indicate aggregator-internal data flow, whereas solid lines stand for data exchanged

with external machines. Aggregator a_1 orchestrates the execution of query q_1 and notifies the client of new results. We hence denote a_1 as the *master* aggregator for q_1 (analogously, a_2 is the master of q_2). The data source s_3 provides data for one single input (i_5), whereas inputs i_1/i_3 and i_2/i_4 are based on the events from s_1 and s_2, respectively. Observe that the events from s_2 are buffered both on a_1 and on a_2, which we denote buffer *duplication*. In Figure 3.11, an arrow pointing from an input i_x to i_y indicates a data dependency, i.e., that i_x provides some data which are required by i_y. In the case of i_1 and i_2, this passing of data happens locally, whereas i_3 and i_4 are handled by different aggregators and hence data are transmitted over the network. We see that assigning i_3 to node a_1 has the advantage that the events from s_1 are buffered only once (for both i_1 and i_3), but is disadvantageous with respect to network traffic between the two aggregators a_1 and a_2. Conversely, s_2 is buffered on both aggregators, reducing the network traffic but requiring more memory. Section 3.4 deals with this tradeoff in more detail and further refines the optimization problem that we strive to solve.

3.3.4 Distributed Query Execution

To serve a large number of simultaneous active queries, the platform employs a scalable distributed processing model with several loosely coupled aggregator nodes working collaboratively. In addition to performance reasons, query distribution may also be required or desired from a higher-level (business) perspective. For instance, the data may have to be physically separated according to business policies. Moreover, if multiple data sources are spread over a large geographical distance, the aggregation can be organized in a location-based hierarchical structure, e.g., with regional and national nodes (for details see [139]).

Hence, the fundamental design principle of WS-Aggregation is that multiple aggregator machines collaboratively process the queries and events requested by the clients. The set of available aggregators is stored in a central service registry, which allows to dynamically select a subset of aggregators responsible for executing each individual query. The overall request is then split up into smaller ("atomic") parts that can be processed by a single node (see *generateRequests* function in Algorithm 1). However, the single parts are not completely isolated units, because of data dependencies which exist between them (see query model in Section 3.3.3). Each time a new event is received and added to the result store (*onEvent* function in Algorithm 1), the dependencies are updated and possibly new request inputs are generated. Note that several inputs, possibly from different aggregation queries, can be affected by an event in the *onEvent* function.

Line 8 in function *generateRequests* indicates that a responsible aggregator is determined for each input. WS-Aggregation supports different configurable distribution strategies, and allows to either specify fixed input-to-aggregator mappings or to assign inputs automatically. In the latter case, the platform performs load balancing. In general, new inputs are assigned to aggregators with the lowest load, based on a combination of Central Processing Unit (CPU) utilization, memory usage, and number of active queries. The second important distribution goal is to bundle (co-locate) inputs with the same underlying event stream. Consider two inputs i_1 and i_2 which receive the same ticks from StockPrice, but use a different preparation query to filter certain information. If these inputs are handled by some aggregator a, a shared event buffer can be used and redundancies are avoided to save memory. Of course, this approach does not scale

Algorithm 1 Processing of Active Query with Dependencies

```
1:  results ← new result store (variable for aggregation results)
2:  function generateRequests(AggregationQuery r)
3:      while r contains independent inputs do
4:          I ← determine independent inputs in r
5:          for all i ∈ I do
6:              G ← generate actual inputs from i
7:              for all input ∈ G do
8:                  aggr ← determine aggregator to handle input
9:                  if aggr is self then
10:                     result ← invoke input on input.target
11:                     result ← apply preparation query to result
12:                     add result to results, update dependencies
13:                 else
14:                     delegate request with input to aggr
15:                 end if
16:             end for
17:         end for
18:     end while
19: end function

1:  function onEvent(Event e) /* called for each incoming event data item */
2:      add e to event buffer of e
3:      for all EventingInput i affected by e do
4:          result ← apply preparation query of i to event buffer of e
5:          add result to results, notify clients, update dependencies
6:          generateRequests(i.aggregationQuery) /* issue new requests */
7:      end for
8:  end function
```

infinitely, and inputs are assigned to new aggregators if the load of a reaches a certain threshold. Optimized load balancing is discussed in detail in Section 3.4 and evaluated in Section 3.7.

3.3.5 Elastic Scaling Using Cloud Computing

To cope with fluctuations in the work load (e.g., handling high volumes of data during load bursts), WS-Aggregation takes advantage of dynamic Cloud resource provisioning to elastically scale the platform up and down. To that end, each aggregator exposes metadata about the current stress level of the machine it is running on, and new machines are requested if all nodes are fully loaded. Conversely, if the nodes operate below a certain stress threshold, the queries can be rearranged to release machines back to the Cloud.

The notion of stress level covers various parameters – it may include CPU and memory usage, list of open files and sockets, length of request queues, number of threads and other metrics. For simplification, we assume that the individual parts of the stress level function are added up and normalized, resulting in a function $stress : A \rightarrow [0, 1]$. Every aggregator provides a metadata interface which can be used to retrieve monitoring information and performance characteristics of the underlying machine. The upper bound of the stress level (value 1) is used to express that an aggregator is currently working at its limit and cannot be assigned new tasks.

In order to achieve optimized query distribution (details see Section 3.4), the nodes' stress levels are continuously monitored. To determine whether a reconfiguration can be applied, it must be ensured that no aggregator node operates beyond a configurable upper-bound stress level λ (e.g., $\lambda = 0.9$). This criterion allows inputs to be removed from machines with high stress level, and may prohibit the assignment of new query inputs. If the algorithm fails to find a valid solution under the given constraints, new machines are dynamically acquired from the Cloud environment and the optimization is restarted.

3.4 Optimized Query Distribution and Placement of Processing Elements

This section provides a detailed solution for the problem of optimized query distribution and placement of processing elements employed in WS-Aggregation. The basis for optimization is the current assignment of inputs to aggregators at some point in time, $cur : I \rightarrow \mathcal{P}(A)$, where $\mathcal{P}(A)$ denotes the powerset of A. We define that $cur(i) = \emptyset$ *iff* input i has not (yet) been assigned to any aggregator node. For now, we assume that each input is only handled by one aggregator, hence $|cur(i)| \leq 1, \forall i \in I$, although WS-Aggregation also allows to assign inputs redundantly to multiple aggregators for fail-safety. The desired result is a new assignment $new : I \rightarrow \mathcal{P}(A)$ in which all inputs are assigned to some aggregator, $|new(i)| = 1, \forall i \in I$. The difference between cur and new constitutes all inputs that need to be migrated from one aggregator to another, denoted as $M := \{i \in I \mid cur(i) \neq \emptyset \wedge cur(i) \neq new(i)\}$.

Migrating a query input may require to migrate/duplicate the event buffer of the underlying data source, if such a buffer does not yet exist on the target aggregator. The technical procedure of migrating event buffers and subscriptions is detailed in Section 3.6.3. The (computational) cost associated with this operation is proportional to the size of the buffer in bytes, expressed as $size : S \times (A \cup \{\emptyset\}) \rightarrow \mathbb{N}$. For instance, the buffer size for a source s on an aggregator a is referenced as $size(s, a)$. If the aggregator is undefined ($\emptyset$), then the buffer size function returns zero: $size(s, \emptyset) = 0, \forall s \in S$. The costs for migration of an input i from its current aggregator to a new node (function $migr$) only apply when the data source of i is not yet buffered on the new node, as expressed in Equation 3.4.

$$migr(i) := \begin{cases} size(source(i), cur(i)), & \text{if } source(i) \notin buf(new(i)) \\ 0, & \text{otherwise} \end{cases} \tag{3.4}$$

In order to decide on actions for load balancing, we need to introduce some notion to express the current load of an aggregator node. In earlier work [148] we observed that the main influencing factor for the aggregators' workload in WS-Aggregation is the number of inputs and the data transfer rate of the underlying event streams. The transfer rate of data streams is therefore continuously measured and averaged over a given time interval (e.g., 1 minute). Every aggregator provides a metadata interface which can be used to retrieve this monitoring information as a function $rate : (S \cup I) \rightarrow \mathbb{R}$, measured in kilobytes per second (kB/s). The $rate(s)$ of a data stream $s \in S$ is the transfer rate of external events arriving at the platform,

and $rate(i)$ for an input $i \in I$ is the internal rate of events after the stream has passed the preprocessor. Based on the data transfer rate, we define the $load$ function for an aggregator $a \in A$ as $load(a) := \sum_{s \in buf(a)} \sum_{i \in I_s} rate(s) \cdot c(i)$, where I_s denotes the set of all inputs targeting s, i.e., $I_s := \{i \in I \mid source(i) = s\}$, and $c : I \to \mathbb{R}$ is an indicator for the computational overhead of the preprocessing operation that transforms the data source s into the input i. The computational overhead depends on the processing logic and can be determined by monitoring. If no information on the running time of a processing step is available, then $c(i)$ defaults to 1. For simplification, the assumption here is that n data streams with a $rate$ of m kB/s generate the same load as a single data stream with a $rate$ of $n*m$ kB/s. We denote the minimum load among all aggregators as $minload := min(\bigcup_{a \in A} load(a))$, and the difference between $minload$ and the load of an aggregator a as $ldiff(a) := load(a) - minload$.

To obtain a notion of the data flow, in particular the network traffic caused by external data flows between aggregator nodes (see Figure 3.11), Equation 3.5 defines the $flow$ between two inputs $i_1, i_2 \in I$. If the inputs are not dependent from each other or if both inputs are handled by the same aggregator, the $flow$ is 0. Otherwise, $flow$ amounts to the data transfer rate ($rate(i_1)$), measured in kB/s.

$$flow(i_1, i_2) := \begin{cases} rate(i_1), & \text{if } (i_1, i_2) \in deps(query(i_1)) \wedge new(i_1) \neq new(i_2) \\ 0, & \text{otherwise} \end{cases} \tag{3.5}$$

Finally, Equation 3.6 introduces $dupl$ to express buffer duplication. The idea is that each data source $s \in S$ needs to be buffered by at least one aggregator, but additional aggregators may also buffer events from the same source (see Figure 3.11). The function $dupl(s)$ hence subtracts 1 from the total number of aggregators buffering events from s.

$$dupl(s) := |\{a \in A \mid s \in buf(a)\}| - 1 \tag{3.6}$$

3.4.1 Optimization Target

We now combine the information given so far in a single target function to obtain a measure for the costs of the current system configuration and the potential benefits of moving to a new configuration. Overall, we strive to achieve a tradeoff between three dimensions: balancing the load among aggregator nodes (*L*), avoiding duplicate buffering of events (*D*), while at the same time minimizing the data transfer between nodes (*T*). The goal of *L* is to keep each node responsive and to account for fluctuations in the frequency and size of incoming event data. The *D* dimension attempts to minimize the globally consumed memory, and *T* aims at a reduction of the time and resources used for marshalling/transmitting/unmarshalling of data.

Figure 3.12 illustrates the tradeoff relationship as a "Magic Triangle": each pair of goals can be fulfilled separately, but the three goals cannot fully be satisfied in combination. For instance, a way to achieve a balanced load for all aggregators (*L*) in combination with no duplicate data source buffers (*D*) is to assign each source to a single aggregator. However, if a single query contains several interdependent data sources on different aggregators (which is likely to occur in this case), the aggregators possibly need to frequently transfer data. Conversely, to achieve

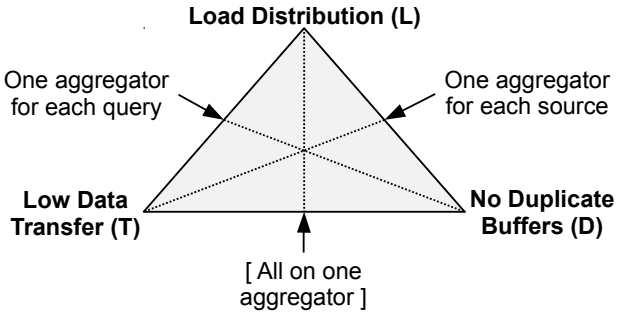


Figure 3.12: Relationship between Optimization Targets

load distribution (*L*) together with low data transfer (*T*), each query with all its inputs could be assigned to a single aggregator, but we observe that duplicate buffers come into existence if any two queries on different aggregators use the same underlying data source. Finally, to achieve both *T* and *D* at the same time, all the processing could be assigned to one single aggregator. As indicated by the brackets in Figure 3.12, this possibility is generally excluded since we are striving for a distributed and scalable solution.

The optimization target function F' in Equation 3.7 contains the three components that are to be minimized. Note that the three parts have different value ranges. Therefore, the target function includes user-defined weights (w_L,w_T,w_D) to offset the differences of the value ranges, and to specify which of the parts should have more impact on the optimization target.

$$F' := w_L * \sum_{a \in A} ldiff(a) + w_T * \sum_{i_1, i_2 \in I} flow(i_1, i_2) + w_D * \sum_{s \in S} dupl(s) \rightarrow min! \tag{3.7}$$

The optimization target in Equation 3.7 only considers how favorable a new system configuration (i.e., assignment of inputs to aggregators) is, but not how (computationally) expensive it is to reach the new setup. To account for the costs of migrating query inputs, we make use of the $migr$ function defined earlier in Section 3.4 and use a weight parameter w_M to determine its influence. The final optimization target function F is printed in Equation 3.8. Note that the additional one-time costs for migration in F are conceptually different from the cost components in F' which apply continuously during the lifetime of the queries.

$$F := F' + w_M * \sum_{i \in M} migr(i) \rightarrow min! \tag{3.8}$$

Depending on the result of the optimization (i.e., the new assignment of inputs to aggregators, function new), for each input $i \in I$ there are three possibilities with regards to migration:

1. If $new(i) = cur(i)$ then there is nothing to do.
2. If $new(i) \neq cur(i) \wedge source(i) \in buf(new(i))$ then the new aggregator $a = new(i)$ needs to be instructed to handle input i, but no migration is required because the target buffer already exists on a.

3. If $new(i) \neq cur(i) \wedge source(i) \notin buf(new(i))$ then we need to perform full migration (or duplication) of the event buffer and corresponding subscription.

3.4.2 Optimization Algorithm

The problem of finding an optimal input-to-aggregator assignment introduced in Section 3.4 is a hard computational problem, and the search space under the given constraints is prohibitively large, i.e., computing exact solutions may be infeasible for non-trivial optimizations with a high number of inputs and aggregators. A formal proof of the problem's intractability is out of the scope of this thesis, but we observe the combinatorial explosion as the algorithm needs to evaluate $\mathcal{O}(|A|^{|I|*red_{max}})$ potentially optimal solutions, where $red_{max} := max(\bigcup_{q \in Q} red(q))$ denotes the maximum level of redundancy. In particular, pruning the solution space is hard to apply because during the search no solution can be easily identified as being suboptimal no matter what other solutions are derived from it. We therefore apply a metaheuristic and use Variable Neighborhood Search (VNS) [123] to approximate a near-optimal solution. The basic principle of VNS is to keep track of the best recorded solution x and to iterate over a predefined set of neighborhood structures which generate new solutions that are similar to x (for more details, see [123]). VNS has been successfully applied in a vast field of problem domains; one example is the multiplayer scheduling problem with communication delays [123], which has similarities to our problem. Figure 3.13 illustrates the encoding of a solution with 3 queries, 10 inputs and a maximum redundancy level of $red_{max} = 2$.

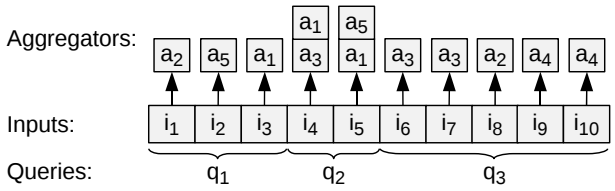


Figure 3.13: Example of Solution Encoding in Optimization Algorithm with $red_{max} = 2$

3.4.2.1 Search Neighborhoods

The definition of neighborhoods in the VNS algorithm allows to guide the search through the space of possible solutions. In the following list of Neighborhoods (NHs), $temp : I \rightarrow A$ denotes the input-to-aggregator assignment of a temporary solution evaluated by the algorithm, and $temp' : I \rightarrow A$ denotes a solution which has been derived from $temp$ as specified by the NH.

- **avoid duplicate buffers NH**: This NH takes a random data source $s \in S$, determines all its inputs $I_s := \{i \in I \mid source(i) = s\}$ and the aggregators responsible for them, $A_s := \bigcup_{i \in I_s} temp(i)$. The NH then generates $|A_s|$ new solutions, in which all inputs in I_s are assigned to one of the responsible aggregators: $|\bigcup_{i \in I_s} temp'(i)| = 1 \wedge \forall i \in I_s : temp'(i) \in A_s$. When this neighborhood gets applied, the newly generated solutions will by tendency have less duplicate buffers.

- **bundle dependent inputs NH**: This NH selects a random query $q \in Q$ and generates new solutions in which all interdependent inputs of q are placed on the same aggregator node. More specifically, for each newly generated solution $temp'$ the following holds: $\forall (i_1, i_2) \in deps(q) : temp'(i_1) = temp'(i_2)$. Note that also transitive dependencies are affected by this criterion. The effect of this neighborhood is a reduced data traffic between aggregators.

- **equal data load per aggregator NH**: This NH selects the two aggregators $a_{max}, a_{min} \in A$ with $load(a_{max}) = max(\bigcup_{a \in A} load(a))$ and $load(a_{min}) = min(\bigcup_{a \in A} load(a))$, and generates a new solution by moving the input with the smallest data rate from a_{max} to a_{min}. More formally, let $I_{max} := \{i \in I \mid temp(i) = a_{max}\}$ denote the set of inputs that are assigned to aggregator a_{max} in the $temp$ solution, then the following holds in every solution derived from it: $temp'(\arg\min_{i \in I_{max}} rate(i)) = a_{min}$.

- **random aggregator swap NH**: This NH simply selects a random subset of inputs $I_x \subseteq I$ and assigns a new aggregator to each of these inputs, $\forall i \in I_x : temp'(i) \neq temp(i)$. The purpose of this NH is to perform jumps in the search space to escape from local optima.

VNS continuously considers neighborhood solutions to improve the current best solution until a termination criterion is reached. The criterion is either based on running time or solution quality. Furthermore, the algorithm only considers valid solutions with respect to the hard constraints. If a better solution than the current setting is found, the required reconfiguration steps are executed. The system thereby stays responsive and continues to execute the affected event-based queries. The effect of different configurations in the optimization procedure is evaluated in detail in Section 3.7.1.4.

3.5 Testing Approach for Reliable Infrastructure Provisioning

The implementation of elastically scaling applications in the Cloud requires mechanisms to frequently integrate new and remove existing computing resources in the system, as exemplified by aggregator nodes that dynamically join and leave the WS-Aggregation platform. Reliable deployment and configuration of computing resources on the infrastructure level, implemented via IaC automation scripts (see Section 2.3.2), is the fundamental prerequisite that enables advanced strategies for elasticity in the first place.

The notion of *idempotence* has been identified as the foundation for repeatable, robust automations [46, 74]. Idempotent tasks can be executed multiple times always yielding the same result. Idempotence is a requirement for *convergence* [74], the ability to reach a desired state under different circumstances in potentially multiple iterations. The algebraic foundations of these concepts are well-studied; however, despite (1) their importance as key elements of DevOps automations and (2) the critical role of automations to enable frequent deployment of complex infrastructures, testing of idempotence in real systems has received little attention. To the best of our knowledge, no work to date has studied the practical implications of idempotence or sought to support developers ascertain that their automations idempotently make the system converge.

We tackle this problem and propose a framework for systematic testing of IaC automation code. Based on a formal model of the problem domain and well-defined test coverage goals, our framework constructs a State Transition Graph (STG) of the automation code under test. The resulting STG is used to derive test cases with the specific goal of detecting idempotence issues. Our prototype implementation is based on Chef, although the approach is designed for general applicability. We rely on Aspect-Oriented Programming (AOP) to seamlessly hook the test execution harness into Chef, with practically no configuration effort required. Since efficient execution of test cases is a key issue, our prototype runs test cases in parallel and utilizes Linux containers (LXC) as light-weight VM environments that can be instantiated within seconds.

Section 3.5.1 provides background on IaC automations and the Chef framework, and highlights typical threats to idempotence. Section 3.5.2 outlines the end-to-end overview of our testing approach. Section 3.5.3 formally models the considered SUT, and Section 3.5.4 discusses STG-based test generation and execution. The approach is extensively evaluated in Section 3.7.2 based on roughly 300 publicly available [255], real-world Chef cookbooks.

3.5.1 Background and Motivation

In this section we briefly revisit the principles behind modern IaC tools and the importance of testing IaC automations for idempotence. Although we couch our discussion in the context of Chef [253], the same principles apply to all such tools.

Idempotence is critical to the correctness of recipes in light of Chef's model of continuous execution and desired-state convergence. Nonetheless, we identify several challenges faced by automation authors when it comes to ensuring that a recipe as a whole is idempotent and can make the system converge to a desired state irrespective of the system's state at the time the recipe execution was initiated. Because of the following challenges, IaC automation authors need support for idempotence and convergence testing.

First, for imperative script resources (such as the example illustrated in Section 2.3.2), the user has the burden of implementing the script in an idempotent way. The user has to decide the appropriate granularity at which idempotence must be enforced so that desired-state convergence can be achieved with no failures or undesirable side effects. This is not trivial for recipes with long code blocks or multiple script resources.

Second, the need to use script resources, not surprisingly, occurs often. For instance, out of 665 cookbooks available in the Opscode community [255], we found that 364 (more than 50%) had to use at least one script resource. What is more, out of 7077 resources that we extracted from all cookbooks, almost 15% were script resources.

Third, although Chef guarantees that the declarative resource types (e.g., `directory`) are idempotent, there is no guarantee that a sequence of multiple instances as a whole is idempotent, as outlined in [74]. Recall that a recipe typically contains a series of several resource instances of different types, and the entire recipe is re-executed periodically.

Finally, if recipes depend on an external component (e.g., a download server), writing the recipe to achieve overall idempotence may become harder due to unforeseen interactions with the external component (e.g., server may be down).

3.5.2 Approach Synopsis

Our work proposes an approach and framework for testing IaC automations (e.g., Chef recipes). To that end, we follow a model-based testing approach [335], according to the process outlined in Figure 3.14. The process contains five main steps with different input and output artifacts. Our test model consists of two main parts: first, a system model of the automation under test and the environment it operates in, including the involved tasks, parameters, system states, desired state changes, etc; second, an STG model which can be directly derived from the system model.

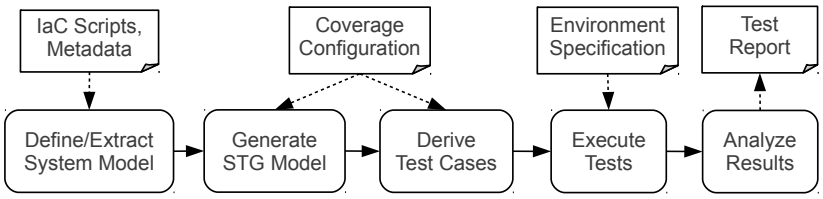


Figure 3.14: Model-Based Testing Process

The input to the first step in Figure 3.14 consists of the IaC scripts, and additional metadata. The scripts are parsed to obtain the basic system model. IaC frameworks like Chef allow to automatically extract most required data, and additional metadata can be provided to complete the model (e.g., precise value domains for automation parameters). Given the sequence of tasks and their expected state transitions, an STG is constructed which models the possible state transitions that result from executing the automation in different configurations and starting from arbitrary initial states. The third task in the process derives test case specifications, taking into account user-defined coverage criteria. The test cases are then materialized and executed in the real system in step four. During test execution, the system is monitored for state changes by intercepting the automation tasks. Test analysis is applied to the collected data in step five, which identifies idempotence issues based on well-defined criteria, and generates a detailed test report.

3.5.3 System Model for Infrastructure Automations

This section introduces a model for the IaC problem domain, as well as a formal definition of idempotence, as considered in this thesis. Our model and definitions in this section provide the foundations for both our test generation method and the semantics of our test execution engine.

Table 3.3 describes each model element and the used symbols. $\mathcal{P}$ denotes the *powerset* of a given set. We use the notation $x[i]$ to refer to the ith item of a tuple x, and $idx(j, x)$ gives the (one-based) index of the first occurrence of item j in tuple x or $\varnothing$ if j does not exist in x. Moreover, $X^{\mathbb{N}} := \bigcup_{n \in \mathbb{N}} X^n$ denotes the set of all tuples (with any length) over the set X.

3.5.3.1 Automation and Automation Tasks

An automation consists of multiple tasks (A), with dependencies (D) between them. We assume a total ordering of tasks, i.e., $\forall a_1, a_2 \in A : (a_1 \neq a_2) \iff ((a_1, a_2) \in D) \oplus ((a_2, a_1) \in D)$. An automation is executed in one or multiple automation runs (R), which in turn consist of a multitude of task executions (E).

Symbol	Description
K, V	Set of possible **state property keys** (K) and **values** (V).
$d : K \to \mathcal{P}(V)$	**Domain** of possible values for a given state property key.
$P := K \times V$	Possible **property assignments**. $\forall\ (k, v) \in P$: $v \in d(k)$
$S \subseteq [K \to V]$	Set of possible **system states**. The state is defined by (a subset of) the state properties and their values.
$A = \{a_1, a_2, .., a_n\}$	Set of **tasks** (or activities) the automation consists of.
$D \subseteq \mathcal{P}(A \times A)$	**Task dependency** relationship: task a_1 must be executed before task a_2 iff $(a_1, a_2) \in D$.
$R = \{r_1, r_2, .., r_m\}$	Set of all historical **automation runs**.
$E = \{e_1, e_2, .., e_l\}$	Set of all historical **task executions**.
$r : E \to R$	Maps task executions to automation runs.
$e : (A \cup R) \to E^{\mathbb{N}}$	**List of task executions** for a task or automation run.
$o : E \to \{success, error\}$	Whether a task execution yielded a **success output**.
$succ, pred : A \to A \cup \varnothing$	Task's **successor** or **predecessor** within an automation.
$st, ft : (E \cup R) \to \mathbb{N}$	Returns the **start time** (st) and **finish time** (ft), e.g., as Unix timestamp.
$t : (S \times A) \to S$	Expected **state transition** of each task. **Pre-state** maps to **post-state**.
$c : E^{\mathbb{N}} \to [S \to S]$	History of actual **state changes** effected by a list of task executions.
$pre, post : A \to \mathcal{P}(S)$ $pre, post : E \to S$	Return all potential (for a task) or concrete (for a task execution) pre-states (pre) and post-states ($post$).

Table 3.3: System Model for Infrastructure Automations

For clarity, we relate the above concepts to a concrete Chef scenario. Consider a Chef recipe that starts and configures a new aggregator node in WS-Aggregation, including installation of the required software dependencies MySQL[1], Sun Java JDK[2], and Maven[3]. For simplicity, we assume that our sample recipe defines four resource instances corresponding to the tasks described in Table 3.4.

#	Task	Parameters
a_1	Install MySQL	-
a_2	Set MySQL password	$p2$ = root password
a_3	Install/Update Java & Maven	$p3$ = operating system distribution (e.g., 'debian')
a_4	Deploy WS-Aggregation middleware	$p4$ = installation path (e.g., '/wsaggr')

Table 3.4: Key Automation Tasks of the Scenario

[1] `http://www.mysql.com/`
[2] `http://www.oracle.com/technetwork/java/javase/downloads/`
[3] `http://maven.apache.org/`

A Chef recipe corresponds to an *automation*, and each resource in the recipe is a *task*. Given our model and the recipe summarized in Table 3.4, we have $A = \{a_1, a_2, a_3, a_4\}$. Note that a_1 could be a package resource to install MySQL, similar to the package resource shown in the recipe of Listing 2.1, whereas a_3 could be implemented by a script resource similar to the one shown in Listing 2.2 (see Section 2.3.2). Table 3.4 also shows the input parameters consumed by each task.

As discussed previously in Section 2.3.2, an automation (Chef recipe) should make the system converge to a *desired state*. Each task leads to a state transition, converting the system from a pre-state to a post-state. A system state $s \in S$ consists of a number of system properties, defined as (key,value) pairs. For our scenario, let us assume we track the state of open ports and OS services installed, such that $K = \{`open\_ports', `services'\}$. Also, suppose that, prior to the automation run, the initial system state is given by $s_0 = \{(`open\_ports', \{22\}), (`services', \{`ssh', `acpid'\})\}$, i.e., port 22 is open and two OS services (`ssh` and `acpid`) are running. After task a_1's execution, the system will transition to a new state $s_1 = \{(`open\_ports', \{22, 3306\}), (`services', \{`ssh', `acpid', `mysql'\})\}$, i.e., task $a_1$ installs the `mysql` service which will be started and open port 3306. Our prototype testing framework tracks the following pieces of state: network routes, OS services, open ports, mounted file systems, file contents and permissions, OS users and groups, cron jobs, installed packages, and consumed resources.

We distinguish the *expected state transition* (expressed via function t) and the *actual state change* (function c) that took place after executing a task. The expected state transitions are used to build an STG (Section 3.5.3.2), whereas the actual state changes are monitored and used for test result analysis.

3.5.3.2 State Transition Graph

The system model established so far in this section can be directly translated into an STG which we then use for test generation. The $STG = (V_G, T_G)$ is a directed graph, where V_G represents the possible system states, and T_G is the set of edges representing the expected state transitions.

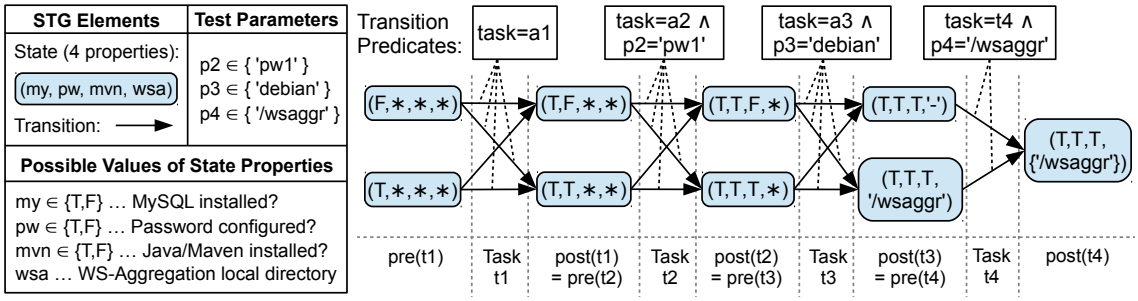


Figure 3.15: Simple State Transition Graph (corresponding to Table 3.4)

Figure 3.15 depicts an STG which contains the pre-states and post-states of the four tasks used in our scenario. For illustration, four properties are encoded in each state: my (MySQL installed?), pw (password configured?), mvn (Java and Maven installed?), and wsa (local in-

stallation path of WS-Aggregation, "-" if not installed). For space limitations, branches (e.g., based on which operating system is used) are not included in the graph. We use a wildcard symbol ($*$) as a placeholder for arbitrary values. The wildcard accommodates the fact that we are only interested in a subset of the system state at different points of the graph. In particular, the pre-states of each task should contain all possible value combinations of the properties the task (potentially) changes. For instance, the automation should succeed regardless of whether MySQL is already installed or not. Hence, the pre-states of task $t1$ contain both values $my = F$ and $my = T$. Note that instead of the wildcard symbol we could also expand the graph and add one state for each possible value, which is not possible here for space limitations.

3.5.3.3 Idempotence of Automation Tasks

Following [74], a task $a \in A$ is *idempotent* with respect to an equivalence relation $\approx$ and a sequence operator $\circ$ if repeating a has the same effect as executing it once, $a \circ a \approx a$. Applied to our model, we define the conditions under which a task is considered idempotent based on the evidence provided by historical task executions (see Definition 3). As the basis for our definition, we introduce the notion of *non-conflicting system states* in Definition 1.

Definition 1 *A state property assignment $(k, v_2) \in P$ is* non-conflicting *with another assignment $(k, v_1) \in P$, denoted $nonConf((k, v_1), (k, v_2))$, if either 1) $v_1 = v_2$ or 2) v_1 indicates a state which eventually leads to state v_2.*

That is, non-conflicting state is used to express state properties which are currently in transition. As an example, consider that k denotes the status of the Apache server. Clearly, for two state values $v_1 = v_2 =$ '$running$', (k, v_2) is non-conflicting with (k, v_1). If v_1 indicates that the server is currently starting up ($v_1 =$ '$booting$'), then (k, v_2) is also non-conflicting with (k, v_1). The notion of non-conflicting state properties accounts for long-running automations which are repeatedly executed until the target state is eventually reached. In general, domain-specific knowledge is required to define concrete non-conflicting properties. By default, we consider state properties as non-conflicting if they are equal. Moreover, if we use a wildcard symbol ($*$) to denote that the value of k is unknown, then (k, v_x) is considered non-conflicting with $(k, *)$ for any $v_x \in V$.

Definition 2 *A state $s_2 \in S$ is* non-conflicting *with some other state $s_1 \in S$ if $\forall (k_1, v_1) \in s_1$, $(k_2, v_2) \in s_2 : (k_1 = k_2) \implies nonConf((k_1, v_1), (k_2, v_2))$.*

Put simply, non-conflicting states require that all state properties in one state be non-conflicting with corresponding state properties in the other state. Based on the notion of non-conflicting states, Definition 3 introduces idempotent tasks.

Definition 3 *An automation task $a \in A$ is considered* idempotent *with respect to its historical executions $e(a) = \langle e_1, e_2, \ldots, e_n \rangle$ iff for each two executions $e_x, e_y \in e(a)$ the following holds:*
$(ft(e_x) \leq st(e_y) \wedge o(e_x) = success) \Rightarrow$
$(o(e_y) = success \wedge (c(\langle e_y \rangle) = \emptyset \vee nonConf(post(e_y), pre(e_y))))$

In verbal terms, if a task execution $e_x \in e(a)$ succeeds at some point, then all following executions (e_y) must yield a successful result, and either (1) effect no state change, or (2) effect a state change where the post-state is non-conflicting with the pre-state. Equivalently, we define idempotence for task sequences (Definition 4).

Definition 4 *A task sequence* $a_{seq} = \langle a_1, a_2, ..., a_n \rangle \in A^n$ *is considered* idempotent *iff for each two sequences of subsequent task executions* $e'_{seq}, e''_{seq} \in (e(a_1) \times e(a_2) \times ... \times e(a_n))$ *the following holds:*
$ft(e'_{seq}[n]) \leq st(e''_{seq}[1]) \Rightarrow$
$((\forall i \in \{1, \ldots, n\} : o(e'_{seq}[i]) = success \Rightarrow o(e''_{seq}[i]) = success) \wedge$
$(c(e''_{seq}) = \emptyset \vee nonConf(post(e''_{seq}[i]), pre(e''_{seq}[i]))))$

Note that our notion of idempotence basically corresponds to the definition in [74], with two subtle differences: first, we not only consider the post-state of tasks, but also distinguish between successful/unsuccessful task executions; second, we do not require post-states to be strictly equal, but allow for non-conflicting states.

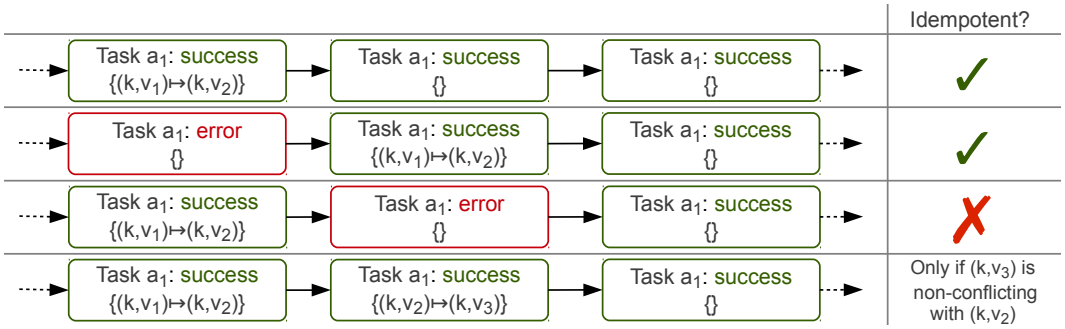


Figure 3.16: Idempotence for Different Task Execution Patterns

Figure 3.16 illustrates idempotence of four distinct task execution sequences. Each execution is represented by a rounded rectangle which contains the result and set of state changes. For simplicity, the figure is based on a single task a_1, but the same principle applies also to task sequences. Sequence 1 is clearly idempotent, since all executions are successful and the state change from pre-state (k, v_1) to post-state (k, v_2) only happens on first execution. Sequence 2 is idempotent, even though it contains an unsuccessful execution in the beginning. This is an important case that accounts for repeatedly executed automations which initially fail until a certain requirement is fulfilled (e.g., WS-Aggregation aggregator node waits until MySQL is configured on another host). Sequence 3 is non-idempotent (even though no state changes take place after the first execution), because an execution with error follows a successful one. As a typical example, consider a task which moves a file using command "`mv X Y`". On second execution, the task returns an error code, because file `X` does not exist anymore. In sequence 4, idempotence depends on whether (k, v_3) represents a state property value that is non-conflicting with (k, v_2). For instance, assume $k =$ '*service.mysql*' denotes whether MySQL is started. If $v_2 =$ '*booting*' and $v_3 =$ '*started*', then a_1 is considered idempotent. Otherwise, if $v_2 =$ '*booting*' and $v_3 =$ '*stopped*', then v_3 is conflicting with v_2, and hence a_1 is not idempotent.

3.5.4 Test Design

This section details the approach for testing idempotence of IaC automations. Section 3.5.4.1 discusses how test cases are derived from a graph representation of the possible system states and transitions, thereby considering customizable test coverage goals. The procedure for applying the coverage configuration to concrete graph instances is discussed in Section 3.5.4.2. Section 3.5.4.3 covers details about the test execution in isolated virtualized environments, as well as test parallelization and distribution.

3.5.4.1 Coverage Goals for STG-Based Test Generation

We observe that the illustrative STG in Figure 3.15 represents a baseline vanilla case. Our aim is to transform and "perturb" this baseline execution sequence is in various ways, simulating different starting states and repeated executions of task sequences, which a robust and idempotent automation should be able to handle. Hence, based on the system model (Section 3.5.3) and a user-defined coverage configuration, we systematically perform graph transformations to construct an STG for the automation which is then used for test case generation. The tester controls different coverage goals which have an influence on the size of the graph and the set of generated test cases. Graph models for testing IaC may contain complex branches (e.g., for different test input parameters) and are in general cyclic (to account for repeated execution). However, in order to efficiently apply test generation to the STG, we prefer to work with an acyclic graph (see below).

In the following we briefly introduce the five test coverage parameters (denoted $idemN$, $repeatN$, $restart$, $forcePre$, and $graph$) which are applied in our approach.

$\underline{idemN}$: This coverage parameter specifies a set of task sequence lengths for which idempotence should be tested. The possible values range from $idemN = 1$ (idempotence of only single tasks) to $idemN = |A|$ (maximum sequence length covering all automation tasks). Evidently, higher values produce more test cases, but lower values entail the risk that problems related to dependencies between "distant" tasks (tasks that are far apart in the task sequence) are potentially not detected (see discussion in Section 3.7.2.2).

$\underline{repeatN}$: This parameter controls the number of times each task is (at least) repeated. If the automation is supposed to converge after a single run (most Chef recipes are designed that way, see our evaluation in Section 3.7.2), it is usually sufficient to have $repeatN = 1$, because many idempotence related problems are already detected after executing a task (or task sequence) twice. However, certain scenarios might require higher values for $repeatN$, in particular automations which are designed to be continuously repeated in order to eventually converge. The tester then has to use domain knowledge to set a reasonable boundary for the number of repetitions to test.

$\underline{restart}$: The boolean parameter $restart$ determines whether tasks are arbitrarily repeated in the middle of the automation ($restart = false$), or the whole automation always gets restarted from scratch ($restart = true$). Consider our scenario automation with task sequence $\langle a_1, a_2, a_3, a_4 \rangle$. If we require $idemN = 3$ with $restart = true$, then the test cases could for instance include the task sequences $\langle a_1, a_1, ... \rangle$, $\langle a_1, a_2, a_1, ... \rangle$, $\langle a_1, a_2, a_3, a_1, ... \rangle$. If $restart = false$, we have additional test cases, including $\langle a_1, a_2, a_3, a_2, a_3, ... \rangle$, $\langle a_1, a_2, a_3, a_4, a_2, a_3, ... \rangle$, etc.

$forcePre$: This parameter specifies whether the constructed graph should enforce that all pre-states for each task are considered. If $forcePre = true$, then for each task $a \in A$ and each potential pre-state $s \in pre(a)$ there needs to exist a node in the STG (as illustrated in Figure 3.15). Note that the potential pre-states should also include all post-states, because of repeated task execution. Contrary, $forcePre = false$ indicates that a wildcard can be used for each pre-state, which would reduce the states in Figure 3.15 from 9 to 5. The latter ($forcePre = false$) is a good baseline case if pre-states are unknown or hard to produce. In fact, enforcing a certain pre-state either involves executing the task (if the desired pre-state matches a corresponding post-state) or accessing the system state directly, which is far from trivial.
$graph$: This parameter refers to the STG-based coverage goal that should be achieved. In [252], four testing goals (with increasing level of coverage) are distinguished to derive test cases from state-based specifications. *Transition coverage*, *full predicate coverage* (one test case for each clause on each transition predicate, cf. Figure 3.15), *transition-pair coverage* (for each state node, all combinations of incoming and outgoing transitions are tested), and *full sequence coverage* (each possible and relevant execution path is tested, usually constrained by applying domain knowledge to ensure a finite set of tests [252]). By default, we utilize transition coverage on a cycle-free graph. Details are discussed in Section 3.5.4.1.

3.5.4.2 Coverage-Specific STG Construction

In Figure 3.17, graph construction is illustrated by means of an STG which is gradually enriched and modified as new coverage parameters are defined. The STG is again based on our scenario (labels with state properties and transition predicates are left out). First, $forcePre = false$ reduces the number of states as compared to Figure 3.15. Then, we require that task sequences of any length should be tested for idempotence ($idemN = \{1, 2, 3, 4\}$), which introduces new transitions and cycles into the graph. The configuration $restart = true$ removes part of the transitions, cycles still remain. After the fourth configuration step, $repeatN = 1$, we have determined the maximum number of iterations and construct an acyclic graph.

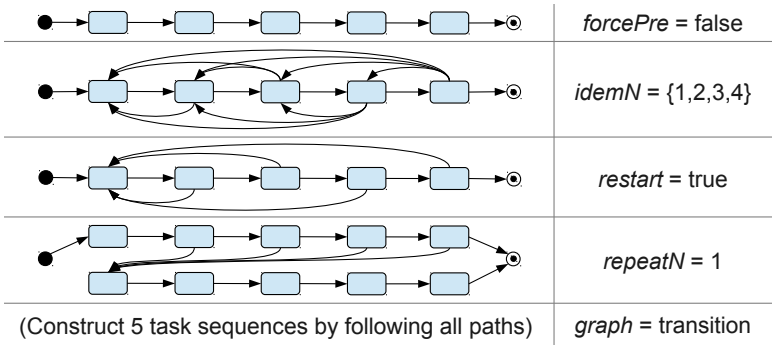


Figure 3.17: Coverage-Specific STG Construction

To satisfy the $graph = transition$ criterion in the last step, we perform a deep graph search to find any paths from the start node to the terminal node. The procedure is trivial, since the graph is already acyclic at this point. Each generated execution path corresponds to one test

case, and the transition predicates along the path correspond to the inputs for each task (e.g., MySQL password parameter $p2$, cf. Figure 3.15). For brevity, our scenario does not illustrate the use of alternative task parameter inputs, but it is easy to see how input parameters can be mapped to transition predicates. As part of our future work, we consider combining our approach with combinatorial testing techniques [242] to cover different input parameters. It should be noted, though, that (user-defined) input parameters in the context of testing IaC are way less important than in traditional software testing, since the core "input" to automation scripts is typically defined by the characteristics of the environment they operate in.

3.5.4.3 Test Execution

The coverage-specific graph-based test model constructed in Section 3.5.4.1 is used to generate executable test automations. The key information for each test case is 1) the input parameters consumed by the tasks, and 2) the configuration of tasks to be repeated. For 1), default parameters can be provided along with the metadata in the system model (cf. Figure 3.14). Moreover, most scripts in IaC frameworks like Chef define reasonable default values suitable for most purposes. For 2), given the path specification for a test case, we traverse along the path and generate a list of task sequences that are to be repeated by the test case.

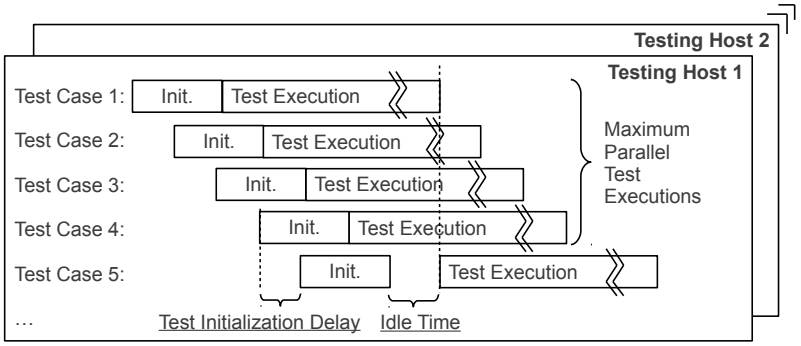


Figure 3.18: Test Execution Pipeline

Since our tests rely on extraction of state information, it is vital that each test be executed in a clean and isolated environment. At the same time, tests should be parallelized for efficient usage of computing resources. VM containers provide the right level of virtualization abstraction for this purpose. A VM operates within a host operating system and encapsulates the filesystem, networking stack, process space, and other state relevant parts of the system. Details about VM containers in our implementation are given in Section 3.6.4.

Having an isolated VM for each test case, the execution can be managed in a testing pipeline, which is illustrated in Figure 3.18. Before the actual execution, each test container is provided with a short initialization time with exclusive resource access for booting the system, initializing the automation environment and configuring all parameters. Test execution is then parallelized in two dimensions: a (bounded) number of test containers can run in parallel on a single host, and the test cases are distributed to multiple testing hosts.

3.6 Implementation

This section covers selected implementation details of the WS-Aggregation prototype that has been developed in the course of this thesis. First, Section 3.6.1 discusses how the query model is implemented with a specialized query language based on XQuery. Section 3.6.2 outlines the internal architecture of WS-Aggregation aggregator nodes and the distributed processing logic for continuous event-based queries. In Section 3.6.3, we introduce the approach for migration of event buffers and subscriptions between aggregator nodes. Section 3.6.4 details the implementation of our framework for testing reliable infrastructure provisioning, and discusses the integration with the Chef platform.

3.6.1 Query Model and WAQL Query Language

A simplified version of the WS-Aggregation query model is illustrated as a UML class diagram in Figure 3.19. The central entity *AggregationQuery* specifies the Endpoint Reference (Endpoint Reference (EPR)) used to receive result updates (*notifyTo*). An aggregation query contains multiple *Input*s (identified by *ID*) that determine how data from external sources are retrieved and inserted into the active query. The *EventingInput* entity creates event subscriptions with an optional *filter* that is evaluated by the target Web service as defined in WS-Addressing. On the other hand, *RequestInput* is used for documents retrieved in a request-response manner. In both cases, the *target* EPR specifies the location of the service. The *interval* attribute allows to continuously monitor a Web service or document for changes.

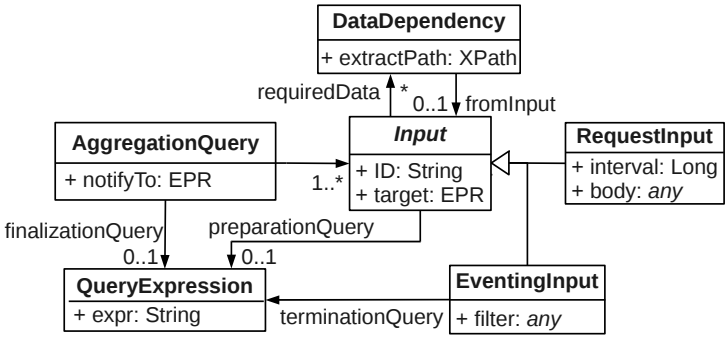


Figure 3.19: Query Model for Continuous Event-Based Data Aggregation

WS-Aggregation employs a specialized query language named *WAQL* (Web services Aggregation Query Language), which is built on *XQuery* 3.0 [361] and adds some convenience extensions, e.g., to express data dependencies between query inputs. The model in Figure 3.19 contains three types of *QueryExpression*s. A *preparationQuery* expression may be used to prepare and transform the result of an Input immediately as it arrives at the platform. In the case of a RequestInput, the preparation query performs a one-time transformation (e.g., extracting certain information of interest), whereas to "prepare" an EventingInput a window query is continuously executed on the event stream to yield new results. To specify the condition for ending an event subscription, an EventingInput is associated with a *terminationQuery*. When this query

yields a *true* result, the target service is automatically invoked with a WS-Eventing *Unsubscribe* message to destroy the subscription. Finally, the *finalizationQuery* combines all the prepared results and constructs the final output document.

3.6.1.1 XQuery Extension for Input Data Dependencies

A core feature in the query model is the concept of data dependencies between two inputs i_1 and i_2, which signifies that i_2 can only be "activated" if certain data from i_1 are available to be inserted into i_2. Activation in this context means that the input becomes usable only when all data dependencies are resolved. The query model in Figure 3.19 associates an Input (*receiving* input), via the association class *DataDependency*, with an arbitrary number of required data from other Inputs (*providing* inputs) of the same query. The attribute *extractPath* is an XPath which points to the data in the providing input. If the optional association *fromInput* is set, the data will be extracted from a specific Input; otherwise, if *fromInput* is unknown, the platform continuously matches *extractPath* against the available inputs and extracts data when this XPath evaluates to *true*.

```
[new] DataDependency ::= "$" Name? "{" PathExpr "}"
[new] EscapeDollar ::= "$$"
[125] PrimaryExpr ::= DataDependency | ...
[145] CommonContent ::= DataDependency | EscapeDollar | ...
[204] ElementContentChar ::= Char - [{}<&$]
[205] QuotAttrContentChar ::= Char - ["{}<&$]
[206] AposAttrContentChar ::= Char - ['{}<&$]
```

Listing 3.1: XQuery Language Extension for Data Dependencies

We propose an XQuery language extension to account for simple modeling of data dependencies as proposed in this thesis. The modifications are printed in Extended Backus-Naur Form (EBNF) syntax in Listing 3.1. The new construct is named *DataDependency* and consists of a dollar sign ("$"), an optional *Name* token referencing the *ID* of the providing input, and an XPath expression (*PathExpr*) specifying the *extractPath* in curly brackets ("`{`", "`}`"). To express that a string "`${foo}`" should be interpreted as a verbatim string and not as a data dependency, a double dollar sign (*EscapeDollar*) is used for escaping ("`$${foo}`"). The *DataDependency* token is added to the definition of *PrimaryExpr* (rule 125 in the current version of XQuery 3.0) and *CommonContent* (rule 145). Furthermore, to satisfy parser consistency of the syntax rules, the single dollar sign needs to be appended to the list of "exceptional" (non-content) characters (rules 204 to 206).

For the actual XQuery processing, we use the light-weight and high-performance *MXQuery*[4] engine. The implementation of the Preprocessor and XQuery language extension for data dependencies is based on JavaCC[5], a parser generator for Java. The EBNF syntax rules of XQuery were extended with the modifications in Listing 3.1 and transformed into the format of JavaCC. The parser generated by JavaCC reads in the extended XQuery expressions and creates an in-memory representation (abstract syntax tree), which is used to extract the data dependencies.

[4] `http://mxquery.org/`
[5] `https://javacc.java.net/`

3.6.2 Aggregator Nodes and Query Processing

WS-Aggregation is implemented in Java using Web services [364] technology, and largely builds on WS-Eventing [360] as a standardized means to manage subscriptions for event notification messages. The platform is designed for loose coupling and elasticity – aggregators may dynamically join and leave the system, and collaborative query execution across multiple aggregators is initiated in an ad-hoc manner. The endpoint references of currently available aggregator nodes are deployed in a service registry.

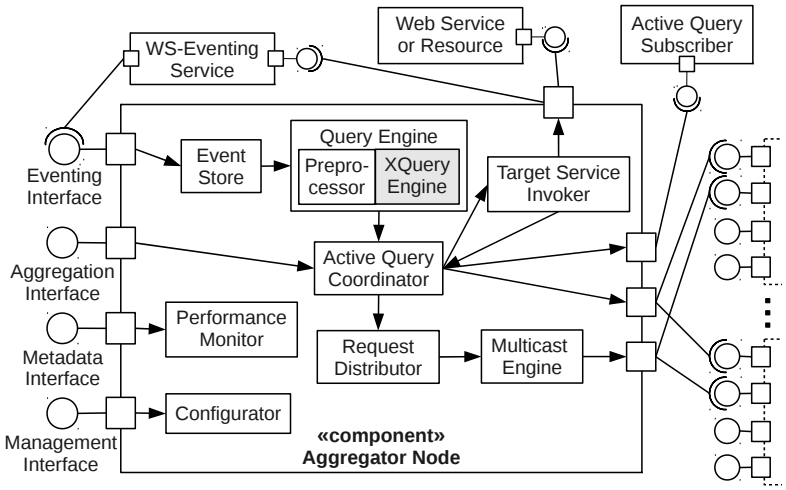


Figure 3.20: Core Components and Connectors of Aggregator Nodes

The internal architecture of aggregator nodes is depicted in Figure 3.20. From an external viewpoint, an aggregator is solely defined by its Web service interfaces. Specialized implementations can be plugged into the platform, and aggregators register themselves in the service registry. The WS-Eventing compliant *Eventing Interface* is used to receive events from data services. the *Event Store* buffers and forwards the events to the *Query Engine* which consists of the *Preprocessor* (responsible for processing the XQuery data dependency extensions) and a third-party (hence depicted in gray) *XQuery Engine*. The Active Query Coordinator (AQC), accessible from the *Aggregation Interface*, maintains aggregation queries, determines which data dependencies are fulfilled and activates new inputs.

The AQC forwards activated inputs to the *Request Distributor*, which implements configurable query distribution strategies. Moreover, the AQC uses the Eventing Interface of partner aggregators to propagate *WindowQueryEvent*s. To communicate with other nodes, the Request Distributor makes use of the *Multicast Engine*, which contacts the Aggregation Interface (to delegate the execution of inputs) or the *Metadata Interface* (to receive metadata such as performance monitoring data) of the partners. Results are pushed to the clients (Active Query Subscribers) by the AQC. Alternatively, clients can poll for new results, which is useful for clients that cannot directly receive push-style notifications (e.g., Web browsers).

3.6.3 Migration of Event Buffers and Subscriptions

One of the technical challenges in our prototype is the implementation of event buffer migration, which becomes necessary when the result of the optimization (see Section 3.4) mandates that certain query inputs be moved between aggregators. The challenge is that transmitting the contents of a buffer over the network is a time-consuming operation, and new events for this buffer may arrive while the transmission is still active. At this point, it must be ensured that the arriving events are temporarily buffered and later forwarded to the target aggregator node. Therefore, transactionally migrating an event buffer while keeping the buffer state consistent at both the sending and the receiving node is a non-trivial task.

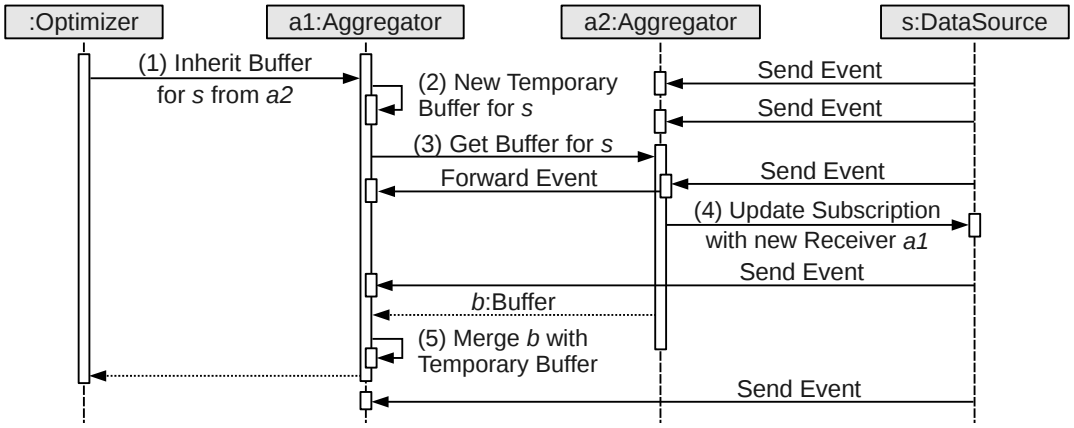


Figure 3.21: Procedure for Migrating Buffer and Event Subscription between Aggregators

Figure 3.21 contains a UML sequence diagram which highlights the key aspects of our solution. It involves the optimizer component which instructs an aggregator $a1$ to *inherit* (become the new owner of) the event subscription for data source s together with the previously buffered events from aggregator $a2$ (point (1) in the figure). Data source d continuously sends events to the currently active subscriber. Before requesting transmission of the buffer contents from $a2$ (3), $a1$ creates a temporary buffer (2). Depending on the buffer size, the transmission may consume a considerable amount of time, and the events arriving at $a2$ are now forwarded to $a1$ and stored in the temporary buffer. The next step is to update the event subscription with the new receiver $a1$ (4). Depending on the capabilities of the data source (e.g., a WS-Eventing service), this can either be achieved by a single *renew* operation, or by a combination of an *unsubscribe* and a *subscribe* invocation. However, the prerequisite for keeping the event data consistent is that this operation executes atomically, i.e., at no point in time both $a1$ and $a2$ may receive the events. Finally, after the transmission has finished, the received buffer b is merged with the temporary buffer (5). If the execution fails at some point, e.g., due to connectivity problems, a rollback procedure is initiated and the entire process is repeated.

3.6.4 Framework for Testing Infrastructure Automation Scripts

In the following we briefly discuss the prototypical implementation of our framework for testing IaC automation scripts. Figure 3.22 illustrates the architecture from the perspective of a single testing host. A Web user interface guides the test execution. Each host contains a test manager to receive requests for new test case executions. The test manager materializes the test cases and creates new containers for each test to execute.

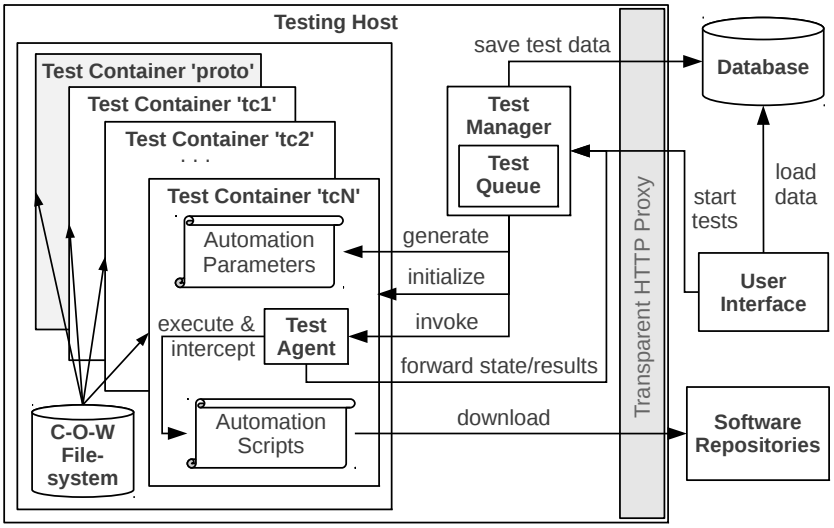


Figure 3.22: Architecture of the Framework for Testing IaC Automations

The framework parallelizes the execution in two dimensions: firstly, multiple testing hosts are started from a pre-configured virtual machine image; secondly, each testing host contains several containers, which each execute one test case in parallel. We utilize the highly efficient Linux containers[6] (LXC). Each container has a dedicated root directory within the testing host's file system. We use the notion of *prototype* container templates (denoted 'proto' in Figure 3.22) to provide a clean operating environment for each test case. Each prototype contains a base operating system (Ubuntu 12.04 and Fedora 16 in our case) and basic services such as a Secure Shell (SSH) daemon. Instead of duplicating the entire filesystem for each test container, we use a *btrfs*[7] copy-on-write (C-O-W) shared filesystem, which allows to spawn new instances efficiently within few seconds. To avoid unnecessary re-downloading of external resources (e.g., software packages), each testing host is equipped with a *Squid*[8] proxy server.

The test agent within each container is responsible for launching the automation scripts and reporting the results back to the test manager which stores them in a database. During test execution, our framework uses *aquarium*[9], an Aspect Oriented Programming (AOP) library for Ruby, to intercept the execution of Chef automation scripts and extract the relevant system state.

[6] `http://lxc.sourceforge.net/`
[7] `https://btrfs.wiki.kernel.org/`
[8] `http://www.squid-cache.org/`
[9] `http://aquarium.rubyforge.org/`

Chef's execution model makes that task fairly easy: an aspect that we defined uses a method join point `run_action` in the class `Chef::Runner`. The aspects then records the state snapshots before and after each task. If an exception is raised during the test execution, the details are stored in the testing DB. Finally, after each task execution we check whether any task needs to be repeated at this time (based on the test case specification).

We have defined an extensible mechanism to define which Chef resources can lead to which state changes. For example, the `user` Chef resource may add a user. Whenever this resource is executed we record whether a user was actually added in the OS. As part of the interception, we leverage this mapping to determine the corresponding system state in the container via Chef's internal discovery tool *Ohai*. We extended *Ohai* with our own plugins to capture the level of detail we required. To support more fine-grained capturing of state changes in the system, we utilize a patched version of *strace*[10] which is able to intercept Unix operating system calls and the state changes associated with these calls. This allows to capture detailed state changes of arbitrary script tasks, even of complex multi-step configuration tasks such as configuring, compiling and installing a software package (cf. Listing 2.1).

3.7 Evaluation

This section evaluates different aspects of the contributions discussed in the context of WS-Aggregation. The evaluation is divided into two main parts. First, Section 3.7.1 evaluates the runtime performance of WS-Aggregation, focusing on elastic Cloud deployment, migration of event buffers, and evolution of query topologies. Second, Section 3.7.2 evaluates the testing framework for infrastructure automations, based on publicly available Chef automation scripts.

3.7.1 WS-Aggregation Runtime Performance

To evaluate the performance of WS-Aggregation, we have set up several experiments in private and public Cloud environments, notably *Eucalyptus*[11], *OpenStack*[12] and *Amazon EC2*[13].

Our performance experiments focus on three aspects: first, the time required to migrate event buffers and subscriptions between aggregators (Section 3.7.1.2); second, evolution of the network topology for different optimization parameters (Section 3.7.1.3); third, performance characteristics of optimization weights (Section 3.7.1.4).

3.7.1.1 End-To-End Framework Performance

To evaluate the end-to-end performance, we have set up a comprehensive test environment in Amazon EC2. An initial number of 15 aggregator nodes was launched at the start of the experiment. During execution, the framework was configured to deploy up to five additional instances, which is achieved using the Web services based API of EC2. Furthermore, we deployed the four scenario Web services from Section 3.3.1, which provide randomized test data.

[10] `http://sourceforge.net/projects/strace/`
[11] `http://www.eucalyptus.com/`
[12] `http://www.openstack.org/`
[13] `http://aws.amazon.com/ec2`

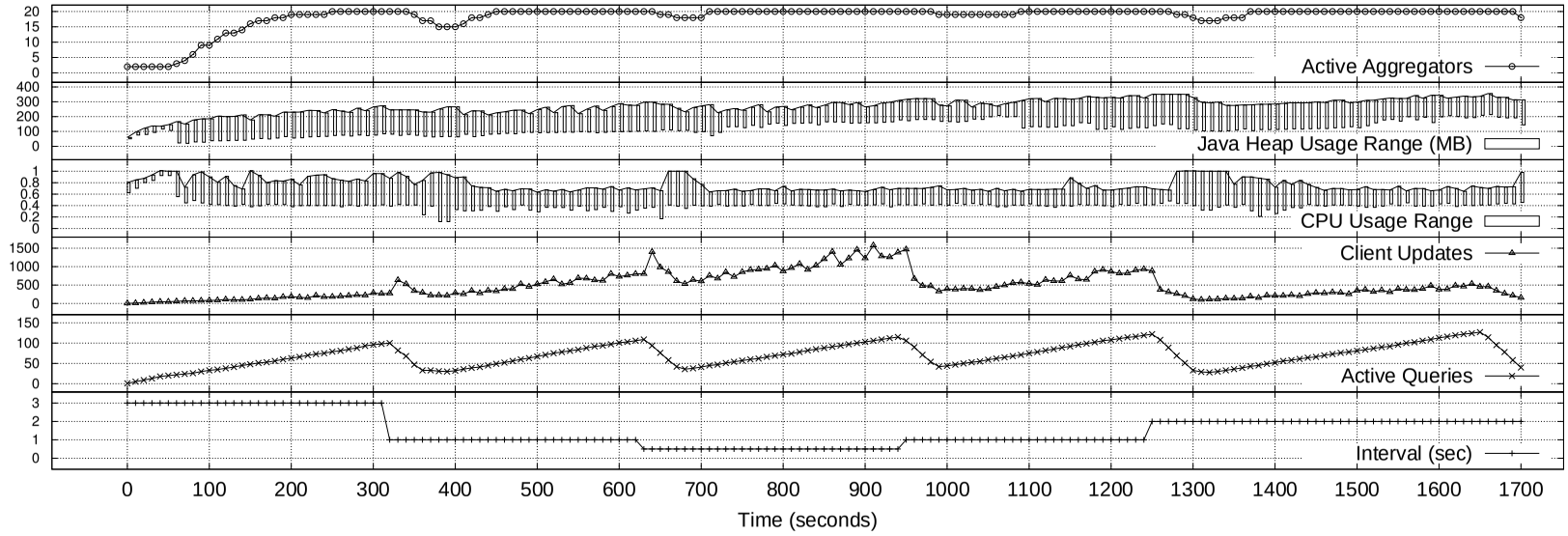


Figure 3.23: Performance Results for Multiple Queries with Varying Event Rates

Figure 3.23 illustrates the results of the scenario execution. The x-axis shows the time in seconds. The lowermost part of the figure plots the interval in which the StockPrice and StockTrade services publish events to the platform. Over time, various test clients deployed outside of EC2 (average latency of 60ms) have requested and terminated multiple (up to 125 simultaneous) executions of the scenario aggregation in different variants (sub-plot *Active Queries*). The number of *Client Updates* per ten seconds (up to 1500 around time point 900) is largely influenced by a combination of event *Interval* (between 0.5 and 3 sec.) and Active Queries, and also depends on the random test data and the state of each active query.

The framework monitors the resource consumption using Java Management Extensions (JMX). Heap memory and CPU consumption are shown in Figure 3.23 with the range (minimum to maximum) and the trendline of the maximum over all active aggregators. The platform heuristically distributes the total load, based on CPU/memory usage, active aggregators and queries. Up to second 50, the queries are handled by only two aggregators, because the aim is to bundle queries for one event stream on the same aggregator. However, after 60 seconds, additional aggregators are involved to avoid performance deterioration due to the increasing load. When ten aggregators are active, the platform requests new machines in addition to the 15 initial instances. The startup (roughly 40 seconds) includes EC2 overhead and the time to initialize and add the new aggregator to the registry. As the active queries decrease, some aggregators become idle (e.g., time 400). The timeout for releasing unused resources is configurable – it should be at least several minutes because aggregators may become used again (e.g., time 410) and most Cloud instances are billed by a fixed time unit (e.g., full hours).

We observe that the load is stable and equally distributed (small CPU and memory ranges) when the number of active queries only slightly changes (e.g., between time 400-600 or 700-900); however, rapid changes in the active queries cause load peaks, as event stores are initialized or terminated and many objects need to be allocated or freed, respectively. Note that the memory consumption grows particularly at the beginning, because the nodes perform internal caching. A factor that evidently raises memory management issues is the need to store past events for evaluation of query windows. Fortunately, the *MXQuery* query engine employs sophisticated algorithms to free unused input buffer items, and we have run several complex queries with literally millions of events without running into memory leaks.

3.7.1.2 Migration of Event Buffers and Subscriptions

Next, we briefly evaluate the time required to migrate event buffers and subscriptions, which is required to switch from the current configuration of a query network topology (i.e., assignment of inputs to aggregators) to a new (optimized) configuration. To measure the actual time required, we have executed various buffer migrations with different buffer sizes. Each data point in the scatter plot in Figure 3.24 represents a combination of buffer size and migration duration. The duration measures the gross time needed for the old aggregator a_1 to contact the new aggregator a_2, transmitting the buffer, feeding the buffer contents into the query engine on a_2, and freeing the resources on a_1.

A linear regression curve is also plotted, which shows an approximate trendline (variance of residuals was 0.2735). Note that the numbers in the figure represent the net buffer size, that is, the actual accumulated size of the events as they were transported over the network (serialized

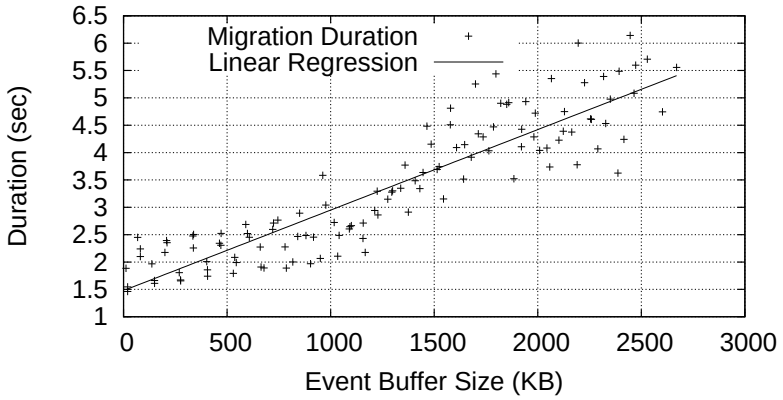


Figure 3.24: Duration for Migrating Event Subscriptions for Different Buffer Sizes

as XML). The gross buffer size, which we evaluate in Section 3.7.1.4, is the amount of Java heap space that is consumed by the objects representing the buffer, plus any auxiliary objects (e.g., indexes for fast access).

3.7.1.3 Evolution of Query Network Topology

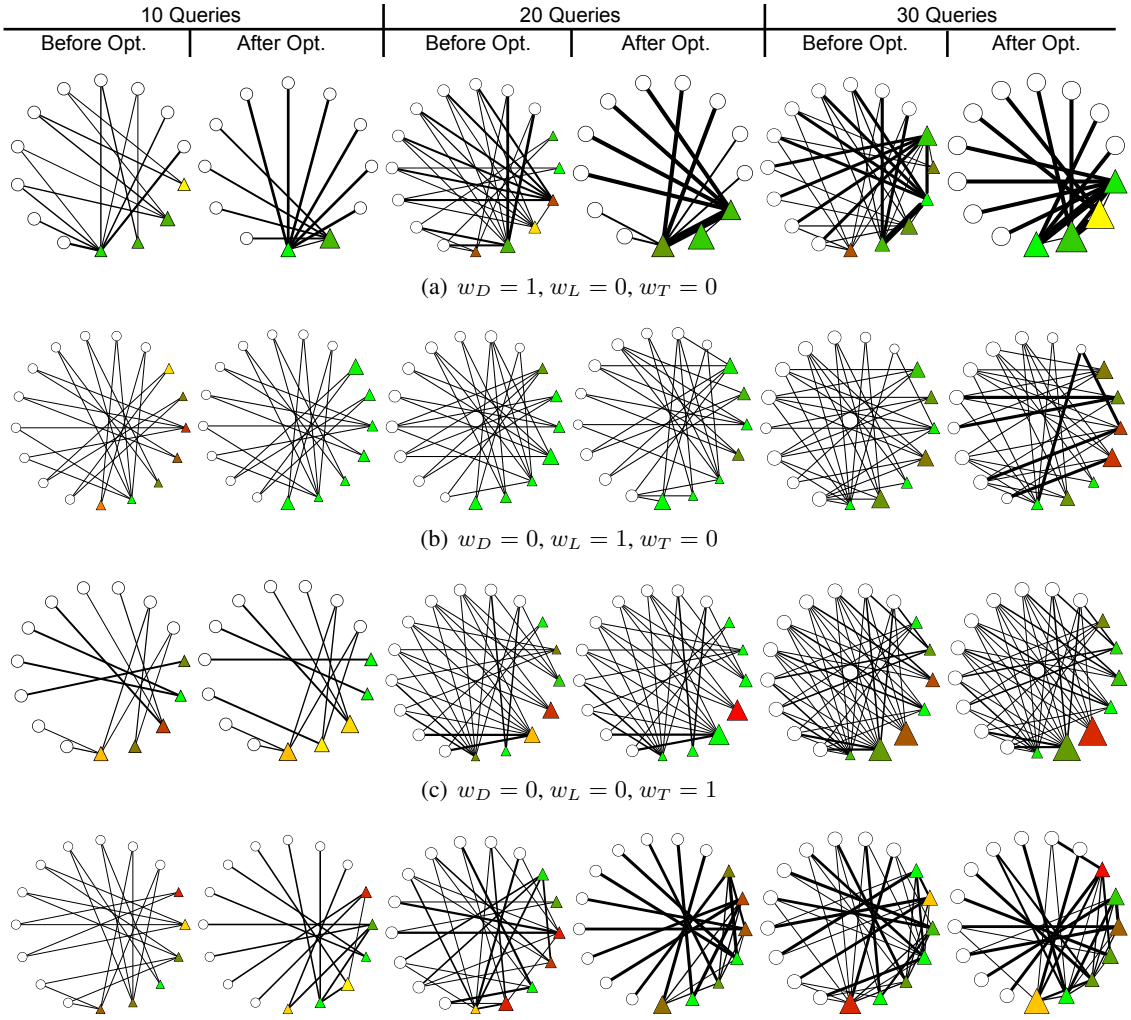
The following experiments study the effects of applying the optimization discussed in Section 3.4. In particular, we investigate the evolution of the network topology (i.e., connections between aggregators and data sources) for different parameter weights. We deployed 10 data sources (each emitting 1 event per second with an XML payload size of 250 bytes) and 7 aggregator nodes, and started 30 eventing queries in 3 consecutive steps (in each step, 10 queries are added). Each query instantiation has the following processing logic:

- Each query q consists of 3 inputs (i_q^1,i_q^2,i_q^3). The inputs' underlying data sources are selected in *round robin* order. That is, starting with the fourth query, some inputs target the same data source (because in total 10 data sources are available) and the buffer can therefore be shared.

- The events arriving from the data sources are preprocessed in a way that each group of 10 events is aggregated. The contents of these 10 events collectively form the input that becomes available to the query.

- Since we are interested in inter-aggregator traffic, each instance of the test query contains a data dependency between the inputs i_q^1 and i_q^2. This means that, if these two inputs are handled by different nodes, the results from i_q^1 are forwarded over the network to the node responsible for i_q^2.

- Finally, the query simply combines the preprocessed inputs into a single document, and the client continuously receives the new data.



(a) $w_D = 1, w_L = 0, w_T = 0$

(b) $w_D = 0, w_L = 1, w_T = 0$

(c) $w_D = 0, w_L = 0, w_T = 1$

(d) $w_D = 1, w_L = 1, w_T = 1; w_M = 1$

Figure 3.25: Query Network Topology With Different Optimization Weights

Figure 3.25 graphically illustrates how the network topology evolves over time for different parameter settings. Each of the subfigures ((a)-(d)) contains six snapshots of the system configuration: for each of the 3 steps in the execution (10/20/30 queries), a snapshot of the system configuration is recorded *before* and *after* the optimization is applied. In each step, the optimization algorithm runs for 30 seconds, and the best found solution is applied. Data sources are depicted as circles, aggregators are represented by triangles, and the nodes with data flow are connected by a line. The size of the nodes and lines determines the load: the bigger a circle, the more event subscriptions are executed on this data source; the bigger a triangle, the more data this aggregator is buffering; the thicker a line, the more data is transferred over the link. Furthermore, the aggregators' colors determine the stress level (green-yellow-red for low-medium-high).

Clearly, the different optimization weights result in very distinct topological patterns. A characteristic outcome of emphasizing the w_D parameter (Figure 3.25(a)) is that few aggregators handle many event subscriptions and are hence loaded with a high data transfer rate. If the goal of preventing duplicate buffering is fully achieved, then there are at most $|S|$ active aggregators (and possibly less, as in Figure 3.25(a)), however, there is usually some inter-aggregator traffic required. In Figure 3.25(b) only the weight w_L is activated, which results in a more dense network graph. The inputs are spread over the aggregators, and in many cases multiple aggregators are subscribed with the same event source. Also in the case where w_T is set to 1, the resulting network graph becomes very dense. We observe that in Figure 3.25(c) there are no inter-aggregator connections, i.e., this setting tends to turn the network topology into a bipartite graph with the data sources in one set and the aggregator nodes in the second set. Finally, in Figure 3.25(d) all weights, including the penalty weight for migration (w_M) are set to 1. Note that the weights are subject to further customization, because setting equal weights favors parameters that have higher absolute values. Future work will evaluate the effect of automatically setting the weights and normalizing the units of the optimization dimensions (D,L,T,M).

3.7.1.4 Performance Characteristics of Optimization Parameters

We now use the same experiment setup as in Section 3.7.1.3 and evaluate in more detail how the performance characteristics evolve over time when optimization is applied. Again, 10 data sources and 7 aggregator nodes were deployed, and multiple queries were successively added.

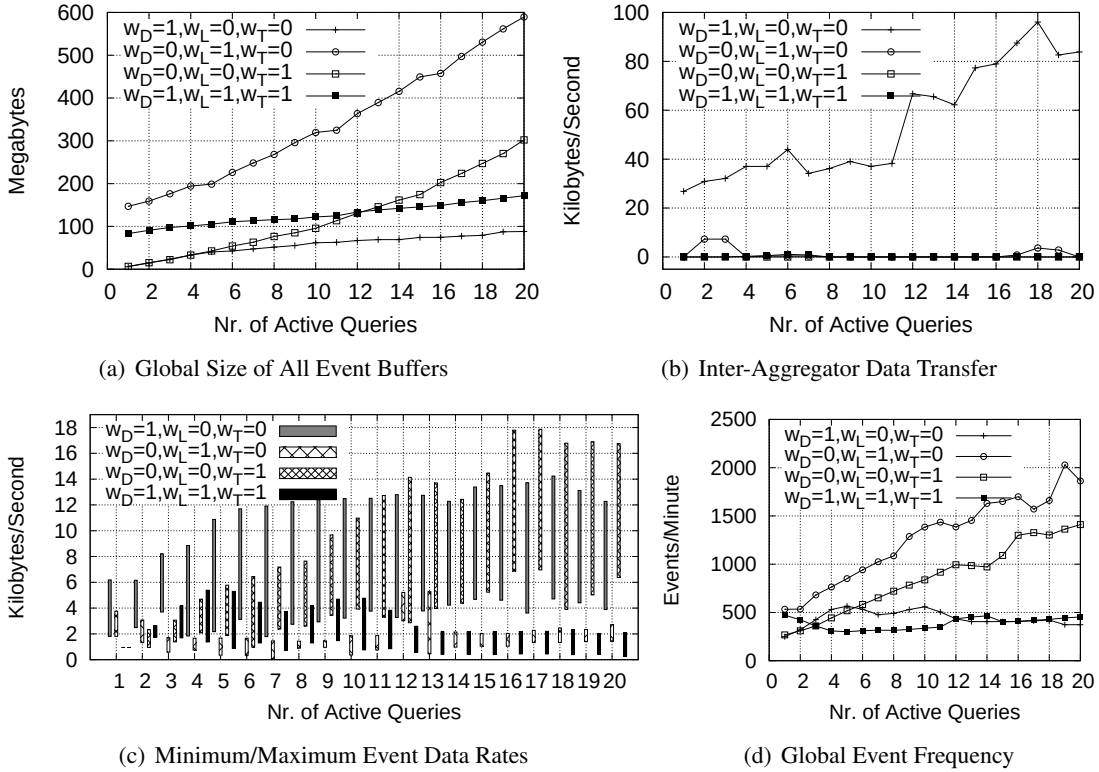
This time we took a snapshot 30 seconds after each query has been added for execution. The 4 subplots in Figure 3.26 illustrate the test results as a trendline over the number of active queries (x axis). To slightly flatten the curves, each experiment has been executed in 5 iterations and the numbers in the figures are mean values. The gross heap size of event buffer objects (Figure 3.26(a)) is determined using the Java instrumentation toolkit (*java.lang.instrument*) by recursively following all object references.

Figure 3.26(a) shows that the global memory usage is particularly high (up to 600MB for 20 queries) for $w_L = 1$ and also for $w_T = 1$. Figure 3.26(b) depicts the inter-aggregator transfer, which in our experiments was quite high for $w_D = 1$, and near zero for the other configurations. The box plots in Figure 3.26(c) show the minimum and maximum event rates over all aggregators. We see that the absolute values and the range difference are high for $w_D = 1$ and $w_T = 1$, but, as expected, considerably lower for the load difference minimizing setting $w_L = 1$. Finally, the combined event frequency of all aggregators is plotted in Figure 3.26(d).

3.7.2 Identified Issues in Real-World Chef Automation Scripts

This section evaluates the testing framework for reliable infrastructure provisioning (Section 3.5), which is employed as part of the elasticity features in WS-Aggregation. To assess the effectiveness of our approach, we have performed a comprehensive evaluation, based on publicly available Chef cookbooks maintained by the *Opscode* community. This large code base of real-world automation scripts allows us to obtain highly representative and reproducible results.

Out of the 665 executable Opscode Chef cookbooks (as of February 2013), we selected a representative sample of 161 cookbooks, some tested in different versions (see Section 3.7.2.4),



(a) Global Size of All Event Buffers

(b) Inter-Aggregator Data Transfer

(c) Minimum/Maximum Event Data Rates

(d) Global Event Frequency

Figure 3.26: Performance Characteristics in Different Settings

resulting in a total of 298 tested cookbooks. Our selection criteria were based on 1) popularity in terms of number of downloads, 2) achieving a mix of recipes using imperative scripting (e.g., `bash`, `execute`) and declarative resources (e.g., `service`, `file`).

In Section 3.7.2.1 we present aggregated test results over the set of automation scripts used for evaluation, Section 3.7.2.2 discusses some interesting cases in more detail, in Section 3.7.2.3 we contrast the test results for different task types, and Section 3.7.2.4 analyzes the evolution of different versions of popular Chef cookbooks.

3.7.2.1 Aggregated Test Results

Here we briefly summarize the test results achieved from applying our testing approach to the selected Opscode Chef cookbooks. For space limitations, we can only highlight the core findings, but we provide a Web page[14] with accompanying material and detailed test results. Table 3.5 gives an overview of the overall evaluation results. The "min/max/total" values indicate the minimum/maximum value over all individual cookbooks, and the total number for all cookbooks.

We have tested a total of 337 cookbooks, selected by high popularity (download count) and number of imperative tasks (script resources). Cookbooks were tested in their most recent

[14] `http://dsg.tuwien.ac.at/testIaC/`

version, and for the 20 most popular cookbooks we tested (up to) 10 versions into the past, in order to assess their evolution with respect to idempotence (see Section 3.7.2.4). As part of the selection process, we manually filtered cookbooks that are not of interest or not suitable for testing: for instance, cookbook `application` defines only attributes and no tasks, or cookbook `pxe_install_server` downloads an entire 700MB Ubuntu image file.

Tested Cookbooks	298
Number of Test Cases	3671
Number of Tasks (min/max/total)	1 / 103 / 4112
Total Task Executions	187986
Captured State Changes	164117
Total Non-Idempotent Tasks	263
Cookbooks With Non-Idempotent Tasks	92
Overall Net Execution Time	25.7 CPU-days
Overall Gross Execution Time	44.07 CPU-days

Table 3.5: Aggregated Evaluation Test Results

The 298 tested cookbooks contain 4112 tasks in total. In our test coverage goal, task sequences of arbitrary length are tested ($\{1, .., |A|\}$), tasks are repeated at most once ($repeatN = 1$), and the entire automation is always restarted from the first task ($restart = true$). Based on this coverage, a total of 3671 test cases (i.e., individual instantiations with different test configurations) were executed. 187986 task executions have been registered in the test database, and 164117 state property changes were registered as a direct result of the task executions. The test execution occupied our hardware for an overall gross time of 44.07 CPU-days. If we extract the overhead of our system, which includes mostly capturing of system state and computation of state changes, the overall net time is 25.7 CPU-days. Due to parallelization (4 testing hosts, max. 5 containers per host) the tests actually finished in much shorter time (roughly 5 days).

The test execution and analysis has led to the identification of 263 non-idempotent tasks. Recall from Section 3.5.3 that a task is considered non-idempotent if any repeated executions lead to a state change or yield a different success result than the first execution of this task.

3.7.2.2 Selected Result Details

To provide a more detailed picture, we discuss interesting cases of non-idempotent recipes. We explain for each case how our approach detected the idempotence issue. We also discuss how we tracked down the actual problem, to verify the results and understand the underlying implementation bug. It should be noted, however, that our focus is on problem detection, not debugging or root cause analysis. However, using the comprehensive data gathered during testing, our framework has also helped us tremendously to find the root of these problems.

Chef Cookbook `timezone`: A short illustrative cookbook is `timezone` v0.0.1 which configures the system's time zone. Table 3.6 lists the three tasks: a_1 installs the package `tzdata`, a_2 writes "UTC" to the configuration file, and a_3 reconfigures the package `tzdata`. All three

tasks write to `/etc/timezone`. Clearly, tasks a_2, a_3 are not idempotent, e.g., considering the sequence $\langle a_1, a_2, a_3, a_1, a_2, a_3 \rangle$. On second execution, a_1 has no effect (package is already installed), but a_2, a_3 are re-executed, effectively overwriting each other's state changes. Note that $\langle a_1, a_2 \rangle$ and $\langle a_1, a_2, a_3 \rangle$ are idempotent as a sequence; however, a perfectly idempotent automation would ensure that tasks do not alternatingly overwrite changes. Moreover, the overhead of re-executing tasks a_2, a_3 can be avoided, which is crucial for frequently repeated automations.

Task	Resource Type	Description
a_1	`package`	Installs `tzdata`, writes "Etc/UTC" to `/etc/timezone`
a_2	`template`	Writes timezone value "UTC" to `/etc/timezone`
a_3	`bash`	Runs `dpkg-reconfigure tzdata`, again writes "Etc/UTC" to `/etc/timezone`

Table 3.6: Tasks in Chef Cookbook `timezone`

Chef Cookbook `tomcat6`: In the popular cookbook `tomcat6` v0.5.4 ($>$ 2000 downloads), we identified a non-trivial idempotence bug related to incorrect file permissions. The version number indicates that the cookbook has undergone a number of revisions and fixes, but this issue was apparently not detected.

Task	Resource Type	Description
...	...	...
a_9	`directory`	Creates directory `/etc/tomcat6/`
...	...	...
a_{16}	`bash`	Copies files to `/etc/tomcat6/` as user `tomcat`; **only** executed if `/etc/tomcat6/tomcat6.conf` does not exist.
...	...	...
a_{22}	`file`	Writes to `/etc/tomcat6/logging.properties` as user `root`.
a_{23}	`service`	Enables the service `tomcat` (i.e., automatic start at boot)
a_{23}	`file`	Creates file `/etc/tomcat6/tomcat6.conf`
...	...	...

Table 3.7: Tasks in Chef Cookbook `tomcat6`

The crucial tasks are outlined in Table 3.7 (the entire automation consists of 25 tasks). Applying the test coverage settings from Section 3.7.2.1, the test suite for this cookbook consists of 23 test cases, out of which two test cases (denoted $t1, t2$) failed. Test t_1 is configured to run task sequence $\langle a_1, ..., a_{21}, a_1, ..., a_{25} \rangle$ (simulating that the automation is terminated and repeated after task a_{21}), and test t_2 is configured with task sequence $\langle a_1, ..., a_{22}, a_1, ..., a_{25} \rangle$ (restarting after task a_{22}). Both test cases failed at the second execution of task a_{16}, denoted $e(a_{16})[2]$ in our model, which copies configuration files to a directory previously created by task a_9. In the following we clarify why and how this fault happens.

The reason why t_1 and t_2 failed when executing $e(a_{16})[2]$ is that at the time of execution the file `/etc/tomcat6/logging.properties` is owned by user `root`, and a_{16} attempts to write to the same file as user `tomcat` (resulting in "permission denied" from the operating system). We observe that task a_{22} also writes to the same file, but in contrast to task a_{16} not as user `tomcat`, but as user `root`. At execution $e(a_{22})[1]$, the content of the file gets updated and the file ownership is set to `root`. Hence, the cookbook developer has introduced an implicit dependency between tasks a_{16} and a_{22}, which leads to idempotence problems. Note that the other 21 test cases did not fail. Clearly, all test cases in which the automation is restarted *before* the execution of task a_{22} are not affected by the bug, since the ownership of the file does not get overwritten. The remaining test cases in which the automation was restarted *after* a_{22} (i.e., after a_{23}, a_{24}, and a_{25}) did not fail due a conditional statement `not_if` which ensures that a_{16} is only executed if the file `/etc/tomcat6/tomcat6.conf` does not exist.

Chef Cookbook `mongodb-10gen`: The third interesting case we discuss is the cookbook `mongodb-10gen` (installs *MongoDB*), for which our framework allowed us to detect an idempotence bug in the Chef implementation itself. The relevant tasks are illustrated in Table 3.8: a_{11} installs package `mongodb-10gen`, a_{12} creates a directory, and a_{13} creates another sub-directory and places configuration files in it. If installed properly, the package `mongodb-10gen` creates user and group `mongodb` on the system. However, since the cookbook does not configure the repository properly, this package cannot be installed, i.e., task a_{11} failed in our tests. Now, as task a_{12} is executed, it attempts to create a directory with user/group `mongodb`, which both do not exist at that time. Let us assume the test case with task sequence $\langle a_1, \ldots, a_{13}, a_1, \ldots, a_{13} \rangle$. As it turns out, the first execution of a_{13} incorrectly creates `/data/mongodb` with user/group set to `root` (instead of `mongodb`). On the second execution of a_{12}, Chef tries to set the directory's ownership and reports an error that user `mongodb` does not exist. This behavior is clearly against Chef's notion of idempotence, because the error should have been reported on the first task execution already. In fact, if the cookbook is run only once, this configuration error would not be detected, but may lead to permission problems at runtime. We submitted a bug report (Opscode ticket `CHEF-4236`) which has been confirmed and fixed by the Chef developers[15].

Task	Resource Type	Description
...	...	...
a_{11}	`package`	Installs package `mongodb-10gen`
a_{12}	`directory`	Creates directory `/data`
a_{13}	`remote_directory`	Creates directory `/data/mongodb` as user/group `mongodb`/`mongodb` and copies configuration files into it.

Table 3.8: Tasks in Chef Cookbook `mongodb-10gen`

Lessons Learned The key take-away message of these illustrative real-world examples is that automations may contain complex implicit dependencies, which IaC developers are often not

[15] `https://tickets.opscode.com/browse/CHEF-4236`

aware of, but which can be efficiently tested by our approach. For instance, the conditional `not_if` in a_{16} of recipe `tomcat6` was introduced to avoid that the config file gets overwritten, but the developer was apparently not aware that this change breaks the idempotence and convergence of the automation. This example demonstrates nicely that some idempotence and convergence problems (particularly those involving dependencies among multiple tasks) cannot be avoided solely by providing declarative and idempotent resource implementations (e.g., as provided in Chef) and hence require systematic testing.

3.7.2.3 Idempotence for Different Task Types

Table 3.9 shows the number of identified non-idempotent tasks (denoted #NI) for different task types. The task types correspond to the Chef resources used in the evaluated cookbooks. The set of scripting tasks (`execute`, `bash`, `script`, `ruby_block`) makes up for 90 of the total 263 non-idempotent tasks, which confirms our suspicion that these tasks are error-prone. Interestingly, the `service` task type also shows many non-idempotent occurrences. Looking further into this issue, we observed that `service` tasks often contain custom code commands to start/restart/enable services, which are prone to idempotence problems.

Task Type	#NI	Task Type	#NI	Task Type	#NI
service	66	directory	10	link	3
execute	44	remote_file	10	bluepill_service	2
package	30	gem_package	7	cookbook_file	2
bash	27	file	5	git	2
template	19	python_pip	5	user	2
script	15	ruby_block	4	apt_package	1

Table 3.9: Non-Idempotent Tasks By Task Type

3.7.2.4 Idempotence for Different Cookbook Versions

We analyzed the evolution of the 20 most popular Chef cookbooks. Table 3.10 lists the results, leaving out cookbooks with empty default recipes (`application`, `openssl`, `users`) and those for which we were unable to find any idempotence issues: `java`, `postgresql`, `mysql`, `build-essential`, `runit`, `nodejs`, `git`, `ntp`, `graylog2`, `python`, `revealcloud`.

Cookbook	i-9	i-8	i-7	i-6	i-5	i-4	i-3	i-2	i-1	i
apache2 (i=1.4.2)	1	1	1	0	0	0	0	0	0	0
nagios (i=3.1.0)	1	1	0	0	0	0	0	0	0	0
zabbix (i=0.0.40)	2	2	2	2	2	2	2	2	2	2
php (i=1.1.4)	1	1	0	0	0	0	0	0	0	0
tomcat6 (i=0.5.4)			3	3	3	3	3	3	2	1
riak (i=1.2.1)	1	1	1	1	1	1	0	0	0	0

Table 3.10: Evolution of Non-Idempotent Tasks By Increasing Version

We observe that, for the cookbooks under test, new releases fixed idempotence issues, or at least did not introduce new issues. Our tool automatically determines these data, hence we argue that it can be used to thoroughly test automations for regressions and new bugs.

3.8 Related Work

In the following we put the contributions presented in this chapter into perspective with previous research. Our discussion focuses on related work in the areas of optimized event processing, placement of processing elements, fault management for event-based systems, as well as reliable infrastructure provisioning.

3.8.1 Optimized Event Processing and Placement of Processing Elements

Due to the large number of its application areas, event processing has attracted the interest of both industry and research [207, 343]. Important topics in CEP include pattern matching over event streams [3], aggregation of events [217] or event specification [77]. In this thesis, the focus is on optimizing the distributed execution of continuous queries over event streams. Hence, we concentrate on some related work in this area in the remainder of this section.

Optimized placement of query processing elements and operators has previously been studied in the area of distributed stream processing systems. Pietzuch et al. [266] present an approach for network-aware operator placement on geographically dispersed machines. Bonfils and Bonnet [38] discuss exploration and adaptation techniques for optimized placement of operator nodes in sensor networks. Our work is also related to query plan creation and multi query optimization, which are core fields in database research. In traditional centralized databases, permutations of join-orders in the query tree are considered in order to compute an optimal execution plan for a single query [160]. Roy et al. [278] present an extension to the *AND-OR* Directed Acyclic Graph (DAG) representation, which models alternative execution plans for multi-query scenarios. Based on the AND-OR DAG, a thorough analysis of different algorithms for multi-query optimizing has been carried out. Zhu et al. [380] study exchangeable query plans and investigate ways to migrate between (sub-)plans.

Seshadri et al. [301, 302] have identified the problem that evaluating continuous queries at a single central node is often infeasible. Our approach builds on their solution which involves a cost-benefit utility model that expresses the total costs as a combination of communication and processing costs. Although the approaches target a similar goal, we see some key differences between their and our work. Firstly, their approach builds on hierarchical network partitions/clusters, whereas WS-Aggregation is loosely coupled and collaborations are initiated in an ad-hoc fashion. Secondly, their work does not tackle runtime migration of query plans and deployments, which is a core focus in this thesis. In fact, WS-Aggregation is a self-adaptive system [42] which implements the Monitor-Analyze-Plan-Execute (MAPE) loop known from Autonomic Computing [171]. In that sense, the purpose of our optimization algorithm is not to determine an optimal query deployment up front, but to apply reconfigurations as the system involves. Chen et al. [65] describe a way to offer continuous stream analytics as a cloud service using multiple engines for providing scalability. Each engine is responsible for parts of the

input stream. The partitioning is based on the contents of the data, e.g., each engine could be responsible for data generated in a certain geographical region.

Several previous publications have discussed issues and solutions related to active queries for internet-scale content delivery. For instance, Li et al. [202] presented the OpenCQ framework for continuous querying of data sources. In OpenCQ a continuous query is a query enriched with a trigger condition and a stop condition. Similarly, the NiagaraCQ system [62] implements internet-scale continuous event processing. Wu et al. [370] present another approach to dealing with high loads in event streams, tailored to the domain of real-time processing of RFID data. Numerous contributions in the field of query processing over data streams have been produced as part of the Stanford Stream Data Manager (STREAM) project [234]. The most important ones range from a specialized query language, to resource allocation in limited environments, to scheduling algorithms for reducing inter-operator queuing. Their work largely focuses on how to approximate query answers when high system load prohibits exact query execution. Query approximation and load shedding under insufficient available resources is also discussed in [14]. Our approach does not support approximation, but exploits the advantages of Cloud Computing to allocate new resources for dynamic migration of query processing elements.

Furthermore, database research has uncovered that special types of queries deserve special treatment and can be further optimized, such as k-nearest neighbor queries [37] or queries over streams that adhere to certain patterns or constraints [16]. WS-Aggregation also considers a special form of 3-way distributed query optimization, which has been presented in [139].

3.8.2 Fault Models for Event-Based Systems

Fowler and Qasemizadeh [106] present a common event model for integrated sensor networks. The model distinguishes four ontologies (event, object, property and time ontology) to represent different aspects of event data. In contrast to our approach their model only focuses on the information associated with an event and does not consider processing logic or topology of EPNs. Other seminal work in the area of models and taxonomies for event-based systems, particular targeting event-based programming systems, has been published by Meier and Cahill [223].

Hadzilacos and Toueg [121] present a comprehensive study of interaction-related faults and fault-tolerance in distributed systems. Various concepts of reliable message delivery are covered, with core focus on message broadcasts. Based on a formal framework, the authors discuss issues such as timeliness or correct ordering of messages. The work is highly influential for event-based systems, particularly for fault types concerning event channels in our proposed model.

Westermann and Jain [351] study commonalities in event-based multimedia applications and discuss features that a common event model should contain. Although the features are largely tailored to multimedia, some aspects apply to event-based systems in general, such as *common base representation*, *application integration*, *common event management infrastructure*, and *common event exploration and visualization tools*. We extend their ideas and argue that integration of different views on event-based systems is vital to improve system dependability. Our model does not yet capture some aspects proposed in [351], most notably uncertainty support, and experiential aspects, which are described as *"ways of exploring and experiencing a course of events to let them [the users] gain insights into how the events evolved"*.

Fault taxonomies for SOA have been discussed in [43] and [59]. The taxonomies are tailored to issues related to service-based computing and Web services (e.g., service discovery, binding, or composition), whereas we focus on specifics of event-based systems.

The authors of [163] have recently proposed a fault injection framework for assessing Partial Fault Tolerance (PFT) of stream processing applications. Their work is tailored to high-frequency data streams and bursty tuple loss. In PFT, there is not only a notion of absolute faults but also of output quality degradation. The level of quality loss is measured using an application-specific output score function. In future work, we also strive to extend our model and approach to support PFT and output quality metrics.

Other researchers' studies focus on the development of software and investigate faults primarily on the source code level. In [90] three main types of faults are identified: *missing*, *wrong* and *extraneous* code constructs. The classification was applied to *diff* and *patch* files of open source projects. The metrics indicate that simple programmer mistakes account for a large portion of faults. For general mistakes like faulty synchronization their approach is certainly applicable in event processing platforms, but with complex interactions in place the relation between failure and responsible artifact becomes hard to assess. A common fault model as presented here can greatly simplify this search.

The work of Steinder and Sethi [311] surveys approaches and techniques for fault localization in computer networks, largely focusing on graph-theoretic fault propagation models like dependency networks and causality graphs. Their contribution provides much more general granularity than our work and parts of the faults discussed here are covered by their approach (e.g., detecting circular dependencies in EPNs). While general fault models have their justification, we argue that it is the fine granularity and domain-specific knowledge that adds to the strength of our approach.

3.8.3 Reliable Infrastructure Provisioning

Existing work has identified the importance of idempotence for building reliable distributed systems [125] and database systems [126]. Over the last years, the importance of building testable system administration [46] based on convergent models [74, 329] became more prevalent. *cfengine* [374] was among the first tools in this space. More recently, other IaC frameworks such aszamboni:12 Chef [253] or Puppet [269] heavily rely on these concepts. However, automated and systematic testing of IaC for verifying idempotence and convergence has received little attention, despite the increasing trend of automating multi-node system deployments (i.e., continuous delivery [133]) and placement of virtual infrastructures in the Cloud [117].

Existing IaC test frameworks allow developers to manually write test code using common Behavior-Driven Development (BDD) techniques. ChefSpec [342] or Cucumber-puppet [180] allow to encode the desired behavior for verifying individual automation tasks (unit testing). Test Kitchen [254] goes one step further by enabling testing of multi-node system deployments. It provisions isolated test environments using VMs which execute the automation under test and verify the results using the provided test framework primitives. This kind of testing is a manual and labor intensive process. Our framework takes a different approach by systematically generating test cases for IaC and executing them in a scalable virtualized environment (LXC) to detect faults and idempotence issues.

Extensive research is conducted on automated software debugging and testing techniques, including model-based testing [268] or symbolic execution [49, 70], as well as their application to specialized problem areas, for instance control flow based [239] or data flow based [144] testing approaches. Most existing work and tools, however, are not directly applicable to the domain of IaC, for two main reasons: (i) IaC exposes fairly different characteristics than traditional software systems, i.e., idempotence and convergence; (ii) IaC needs to be tested in real environments to ensure that system state changes triggered by automation scripts can be asserted accordingly. Such tests are hard to simulate, hence symbolic execution would have little practical value. Even though dry-run capabilities exist (e.g, Chef's *why-run* capability), they cannot replace systematic testing. The applicability of automated testing is a key requirement identified by other approaches [45, 50, 340], whether the test target is system software or IaC.

Existing approaches for middleware testing have largely focused on performance and efficiency. Casale et al. [57] use automatic stress testing for multi-tier systems. Their work places bursty requests on system resources to identify performance bottlenecks as well as latency and throughput degradations. Other work focuses on testing middleware for elasticity [109, 110], which is becoming a key property for Cloud applications. Bucur et al. [45] propose an automated software testing approach that parallelizes symbolic executions for efficiency. The system under test can interact with the environment via a "symbolic system call" layer that implements a set of common POSIX[16] primitives. Their approach could potentially enhance our work and may speed up the performance, but requires a complete implementation of the system call layer.

Other approaches deal with finding and fixing configuration errors [319, 354]. Faults caused by configuration errors are often introduced during deployment and remain dormant until activated by a particular action. Detecting such errors is challenging, but tools like AutoBash [319] or Chronus [354] can effectively help. A natural extension would be to also take into account the IaC scripts to find the configuration parameter that potentially caused the problem. Burg et al. [340] propose automated system tests using declarative VMs. Declarative specifications describe external dependencies (e.g., access to external services) together with imperative test scripts. Their tool then builds and instantiates the VM necessary to run the script. Our approach leverages pre-built LXC containers; dynamic creation of declarative specifications would be possible but building a VM is more costly than bringing up an LXC container.

3.9 Conclusions

This chapter presented WS-Aggregation, a platform for continuous processing of event-based Web services and data. The platform provides a specialized query model focusing on declarative definition of data dependencies between event streams or other query inputs. The system is designed for scalability and distributed query execution, and allows elastic deployment in the Cloud. Reliability and fault management is a central focus in WS-Aggregation; in the following we briefly revisit the different challenges that have been addressed in this chapter.

First, the emerging research field of distributed event-based systems has not yet come to a common and unified understanding of faults. We have taken a step ahead in this direction

[16] Portable Operating System Interface (POSIX)

and present a unified fault taxonomy based on a common model for event-based systems. The taxonomy provides dimensions to obtain a comprehensive allround picture of the system artifacts as well as potential manifestations and sources of faults. We have discussed 30 concrete fault instances that cover all fault types and elements of our common model.

Second, the elasticity features in WS-Aggregation require reliable resource provisioning on the infrastructure level. Given the IaC model of periodic re-executions, idempotence is a critical property which ensures repeatability and allows automations to start executing from arbitrary initial or intermediate states. We propose an approach for model-based testing of IaC automations, aiming to verify whether they can repeatedly make the target system converge to a desired state in an idempotent manner. Our extensive evaluation with real-world IaC scripts from the OpsCode community revealed that the approach effectively detects non-idempotence. Out of roughly 300 tested Chef scripts, almost a third were identified as non-idempotent. In addition, we were able to detect and report a bug in the Chef implementation itself.

Third, the placement of event processing elements plays a key role for the performance and reliability (e.g., coping with load bursts) of distributed data processing. Our proposed approach performs dynamic migration of event buffers and subscriptions to optimize the global resource usage within the platform. The core idea is that event buffers can be reused if multiple query inputs operate on the same data stream. We identified a non-trivial tradeoff that can be expressed as a "magic triangle" with three optimization dimensions: balanced load distribution among the processing nodes, minimal network traffic, and avoidance of event buffer duplication. Variable Neighborhood Search (VNS) has proven effective for solving this hard optimization problem.

We have exemplified our solution on the basis of several experiments carried out with the WS-Aggregation framework. The platform integrates well with the Cloud computing paradigm and allows for elastic scaling based on the current system load and the volume of event data. Our evaluation has illustrated how different optimization parameters can be applied to influence the evolution of the network topology over time. Furthermore, we have evaluated how different performance characteristics evolve in different settings. The experience we have gained in the various experiments conducted has shown that the (short-term) costs of migration or duplication are often outweighed by the (long-term) benefits gained in performance and robustness.

CHAPTER 4

TeCoS: Testing and Fault Localization for Data-Centric Dynamic Service Compositions

4.1 Introduction and Motivation

During the last years, the Service-Oriented Architecture (SOA) [258] paradigm has gained considerable importance as a means for creating loosely coupled distributed applications. Services, the main building blocks of SOA, constitute atomic, autonomous computing units with well-defined interfaces, which encapsulate business logic to provide a certain functionality. Today, Web services [364] are the most commonly used implementation technology for service-based applications (SBAs). One of the defining characteristics of services is their composability, i.e., the possibility to combine individual services into *service compositions*[1] [92]. The de-facto standard for creating composite SBAs with Web services is the Business Process Execution Language for Web Services [246] (WS-BPEL). WS-BPEL uses an XML-based syntax to define the individual activities of the SBA and the control flow between them. The mechanism of dynamic binding in SBAs allows to define a required service interface (abstract service) at design time and to select a concrete service endpoint from a set of concrete candidate services at runtime.

Business- or safety-critical SBAs require thorough testing, not only of the single participants but particularly of the services in their interplay [328]. Initial testing of a dynamic service composition plays a key role for its reliability and performance at runtime. Firstly, the tests may reveal that a concrete service s cannot be integrated, because the results it yields are incompatible with any other service that processes some data produced by s. This incompatibility may be hard to determine with certainty, but testing can indicate potential points of failure. Assume

[1] Note: Throughout this chapter, the term "service composition" is used as a shortcut for "composite service-based application", i.e., a service-based application composed of multiple interconnected services.

a composition is tested n times, each time with a different combination of concrete services. If the test fails x times ($x < n$), and in all these cases the concrete service s was involved, there is possibly an integration problem with s, which can then be further investigated. Secondly, the test outcome can assist in the service selection process, as it enables operators to favor well-performing service combinations and avoid configurations for which tests failed. Furthermore, upfront testing makes it safer and possibly faster to move to a new binding when a new concrete service becomes available. The practical relevance of integration testing of dynamic service compositions is evidenced by ongoing efforts towards definition of standard service interfaces for various industries. For instance, the TeleManagement Forum (TMF[2]) has released an extensive set of best practices and interfaces for the telecommunications domain, allowing providers to implement standardized business processes and to expose their services to the outside world. The high level of adoption of these standards is illustrated in the TMF's 2009 Case Study Handbook [324] which presents 30 real-world solutions from renowned providers. Similar types of standards and interfaces are used in the airline industry (IATA[3]), the retail industry (ARTS [4]), and many others.

In practice, testing dynamic composite SBAs is a challenging task for two main reasons. First, traditional white-box testing procedures are only available if the tester has access to the source code or a model of the service processing logic. However, as a general characteristic of service-oriented software engineering, the implementation of services beyond the interface is usually hidden from service users. In the case of Web services, the interface is provided in the form of a WSDL [365] service description document, combined with XML Schema definitions of the operation inputs and outputs. Second, dynamic service binding is combinatorial in the number of concrete services. That is, for any nontrivial composition, the number of possible combinations may be prohibitively large.

For the first problem, several testing techniques and coverage criteria have been proposed on the service interface level, e.g., [26, 251]. Moreover, recent studies attempt to enable white-box testing approaches for SBAs by creating *testable services* that reflect and report on the degree of test coverage without revealing the actual service internals [24, 94]. The second problem is considered one of the main challenges in service testing [51], and has previously been addressed in group testing of services [331, 332]. These works present very basic high-level service composition models and propose test case generation for progressive unit and integration testing. Our work builds on their approach, and provides techniques for restricting the combinations of services, as well as applying the generated tests to concrete Web service composition technology.

This thesis chapter focuses on integration testing of dynamic service compositions with an emphasis on data-flow centric coverage goals. We propose coverage criteria which target the inter-invocation data dependencies in dynamic service compositions. We show that such dependencies are important to test because of their potential effect on the correct behavior of the composition. We provide a detailed problem formulation, and present a practical, ready-to-use solution that is both based on existing tooling for software testing and applied to actual Web service technology. The major contributions of our work are fourfold:

[2]TeleManagement Forum; `http://www.tmforum.org`

[3]International Air Transport Association; `http://www.iata.org`

[4]Association for Retail Technology Standards; `http://www.nrf-arts.org`

- We illustrate and formalize a data-centric view of service compositions in a high-level, abstract way. We introduce *k-node data flow* coverage as a novel metric expressing to what extent the data dependencies of a dynamic composition are tested. Based on this test coverage metric, we formulate the problem of finding a minimal number of test cases for a service composition as a Combinatorial Optimization Problem (COP). The outcome of the COP is a set of concrete test instantiations of the compositions, which ensures that all relevant data dependencies are covered (hard constraints). Limiting the level of desired coverage via the k-node coverage metric helps to significantly reduce the search space of service combinations. Additionally, testers can assign weights to give precedence to services that are known to be less reliable (soft constraints). Details follow in Section 4.3.
- In order to solve the COP, we make use of FoCuS [151], a tool for coverage analysis and combinatorial test design. We provide an automated transformation from the service composition model to the FoCuS data model. The input to FoCuS is constructed from the composition definition (e.g., WS-BPEL) and meta information about the available services. At this point, we further narrow down the search space by specifying past instantiations of the composition, which constitute existing solutions and can be ignored by the solver. The near-optimal solution computed by FoCuS is then transformed back into the service composition model to construct and execute actual test cases. Details are discussed throughout Section 4.3 and Section 4.5.
- We discuss our approach for analyzing the test results in order to identify faulty services and incompatible service combinations. To assess the outcome of a set of test executions (denoted traces), we introduce two statistical metrics, *fault participation* and *fault contribution*, which are similar to *recall* and *precision* in machine learning or information retrieval (see Section 4.3.5). However, to allow fault localization throughout the evolutionary lifetime of an SBA, we need to consider more complex situations with noisy data, induced by temporary faults or changing fault conditions. To achieve this, we utilize machine learning and propose a noise resilient fault localization technique based on pooled decision trees. The detailed fault localization approach is presented in Section 4.4.
- Section 4.5 discusses the *TeCoS* framework, which is a prototype implementation of the presented approach. TeCoS stores meta information about SBAs, logs invocation traces, and measures different coverage metrics. This combined information is used to generate and execute service composition test cases. We illustrate the support for different types of SBAs by means of a customizable adapter mechanism, which executes the test cases on target platforms. Our evaluation in Section 4.6 analyzes various performance characteristics, and demonstrates the end-to-end practicability of the solution.

4.1.1 Approach Synopsis

In this section, we discuss our solution from a high level perspective. Details of the comprising concepts and elements are presented in the proceeding sections. Figure 4.1 depicts the end-to-end view, including all activities, required inputs, and generated outputs. The input to the first activity is a service composition definition document. This activity converts the input document into a corresponding data flow model, providing an abstraction of the data dependencies between the individual service invocations of the composition (see Section 4.3 for details).

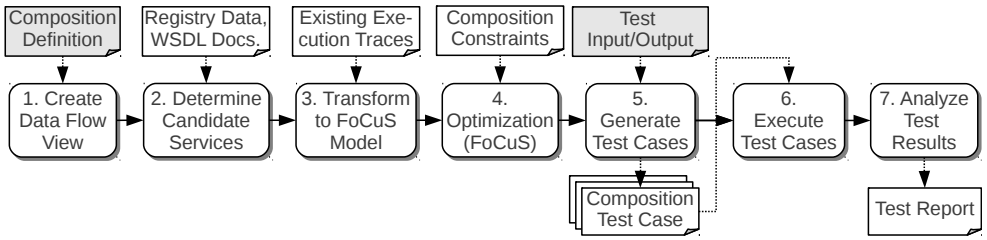


Figure 4.1: End-to-End Testing Approach

The second activity determines the candidate services for each abstract service identified in the data flow view. Then, in the third activity, the combined information is transformed into a model that complies with the FoCuS API. This activity optionally allows to specify existing execution traces of the composition, which are considered as an existing solution by the solver and hence narrow down the search space. The fourth activity is the optimization process (executed by FoCuS), which computes a near-optimal solution for the model produced by activity three. In addition, the optimizer (FoCuS) receives as input a list of additional composition constraints that specify which services should or should not be used in combination in the generated test cases. In activity five, this solution, together with test input/output combinations, is used to generate the concrete test cases for the composition. These tests are then deployed and executed by activity six. Finally, activity seven analyzes the results, determines incompatibilities and potential points of failure, and summarizes the findings in a test report. Within this process, only the composition definition and the test input/output are defined manually by the composition developer/tester (depicted in gray in the figure). Other documents and *all* of the seven activities are automated and require no human involvement.

4.1.2 Roadmap

The remainder of this chapter is organized as follows. In Section 4.2 we introduce an illustrative scenario which serves as the basis for discussion. Section 4.3 introduces a formal model for dynamic data-centric service compositions, defines the coverage goal we aim for, and discusses the combinatorial test design. In Section 4.4 we detail the advanced fault localization approach based on pooled decision trees. Section 4.5 discusses implementation details and outlines the features of the TeCoS framework. Section 4.6 contains a thorough evaluation covering various performance and quality aspects, and Section 4.7 points to related work in the areas of testing and fault localization for SBAs. Section 4.8 concludes with an outlook for future work.

4.2 Scenario

We base the description of dynamic data-centric compositions on an illustrative scenario of a tourist who plans a city trip and requests vacant hotel rooms, available flights and visa regulations for the target destination. This functionality is implemented as a composite Web service using WS-BPEL and WS-Aggregation (see Chapter 3). WS-BPEL is used to model and execute the

business process logic of the composition, whereas the WS-Aggregation platform is tailored to event-based processing of Web service data with continuous queries and high data throughput.

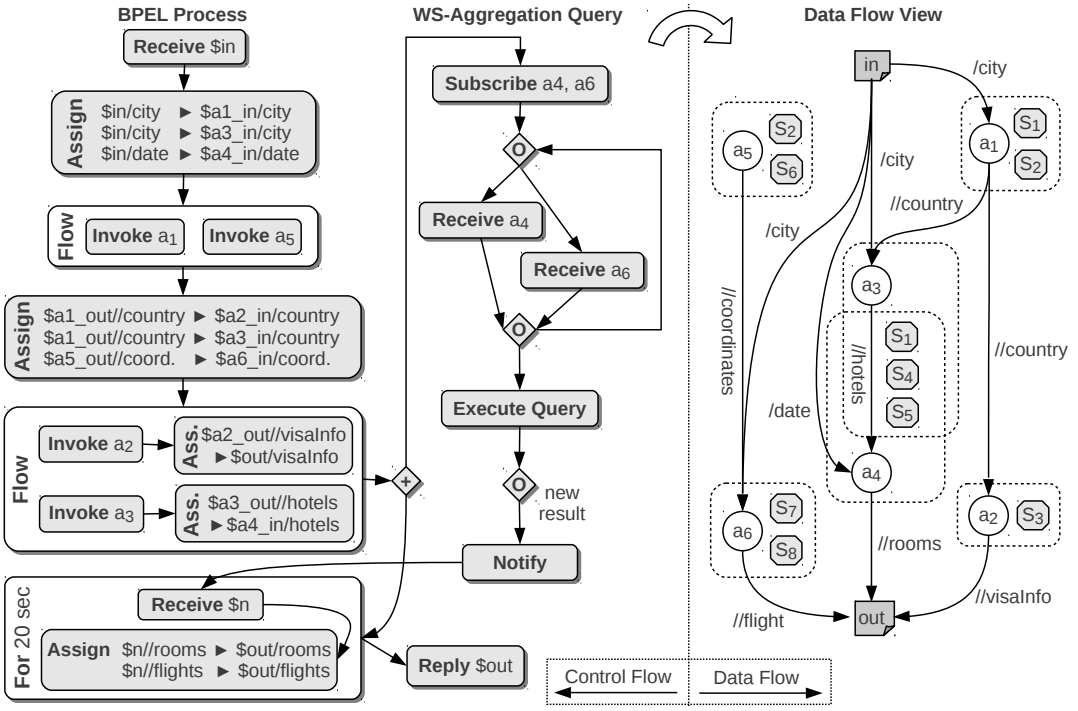


Figure 4.2: Scenario – Composition Business Logic View (left) and Data Flow View (right)

Figure 4.2 depicts the corresponding service composition: the left part of the figure contains a graphical representation of the WS-BPEL process and the WS-Aggregation query processing steps (in BPMN notation [250]), and the right part illustrates the data flow between the services of the process. While the WS-BPEL process and WS-Aggregation query are defined by the composition developer, the data flow view can be generated automatically.

The scenario composition receives an input (`in`) and uses six abstract services $\{a_1, ..., a_6\}$ to produce the output (`out`). The input contains two elements, `city` and `date`. In WS-BPEL, an abstract service is defined as an `invoke` activity, which is associated with input and output variables, e.g., `$a1_in` and `$a1_out` for service a_1. The `assign` activities copy the required output of one abstract service to the input variable of another service. The respective source and destination are defined via XPath [362] expressions.

Service a_1 determines the country of the given city, a_2 returns the visa information of this country. Services a_3 and a_4 retrieve existing hotels in the specified city/country and available hotel rooms, respectively. Finally, a_5 looks up the user's current geographical coordinates (e.g., using GPS on a mobile device), which are used to find available flights with service a_6. While the WS-BPEL process largely follows a sequential procedure, WS-Aggregation maintains event subscriptions to receive asynchronous notification events from the services a_4 (if new rooms become available) and a_6 (if new flight information is available) and performs continuous queries over these events.

In the left part of Figure 4.2, arrows indicate the control flow between the activities. The structured `flow` activities in WS-BPEL signify that the contained activities execute in parallel. The `for` activity defines a loop that executes for 20 seconds and collects notifications received from WS-Aggregation. The diamond shapes represent split and merge nodes, denoted as gateways in BPMN. A diamond with a plus sign ("+") represents a parallel gateway (all outgoing paths are executed), and a diamond with a circle is an inclusive gateway (any combination of paths may be taken, zero to all).

The right part of Figure 4.2 shows the composition business logic view transformed into the composition data flow view. The data flow view used here is a simplified version of the WS-BPEL data dependencies model presented in [377]. Whereas their approach models the actual structure of the WS-BPEL process, our view abstracts from the composition process and is agnostic of control flow instruments, such as the WS-BPEL `flow` or `for` activities. Therefore, our approach lends itself to model data flows in heterogeneous service composition environments (WS-BPEL and WS-Aggregation in our case). The arrows signify the data flow between services. An arrow pointing from a service a_x to a_y, labeled with an XPath expression, means that a part of the output of a_x becomes (part of) the input of a_y. As an additional information, we included the concrete service endpoints in the figure. A dashed rounded box around an abstract service defines the concrete services that can deliver the desired functionality, e.g., the abstract service a_1 is implemented by the concrete Web services s_1 and s_2. At runtime, the composition is instantiated by selecting for each abstract service one of a set of concrete candidate services, often denoted as *dynamic binding* [85]. The service selection process usually happens based on non-functional and QoS [226] (Quality of Service) parameters.

In our scenario we define that the endpoint for a_3 and a_4 must always be the same concrete service (one out of s_1,s_4,s_5). As an example, using s_1 for a_3 and s_4 for a_4 would result in an incompatibility and is not allowed. Note that such constraints are defined as additional information, and are not directly (graphically) reflected in the data flow view.

4.2.1 Sources of Faults in Dynamic Service Compositions

Even in the case that all concrete candidate services have been tested individually (unit testing), the challenge is to ensure that they also function correctly in their interplay (integration testing). The services in our scenario depend on one another, both directly and transitively, via data flow dependencies. If the services are not properly tested in combination, these dependencies can lead to undesired behavior and defects in the composition. The following list contains an excerpt of potential composition faults which raise the necessity of end-to-end integration testing:

* *Syntactic Data Incompatibility*: The coordinates returned by the geographic service (abstract service a_5 in the composition) are encoded as strings. Assume that the concrete service s_2 uses the notation *51° 28' 38" N*, whereas s_6 uses a dash as delimiter, i.e., *51:28:38:N*. If the concrete flight services s_7 and s_8 (a_6 in the composition) are not able to parse both formats correctly, the combination will likely lead to an exception or a faulty composition result.

* *Semantic Data Incompatibility*: Assume that the data formats of s_2 and s_6 are syntactically identical, e.g., both use the format *51:28:38:N* to encode the geographic latitude.

Although the reference point most frequently used today is the Greenwich Meridian, in principle the meridian is a matter of convention and historically different meridians have been used (e.g., Paris Meridian). Evidently, if two services do not follow the same semantic data conventions, the composition becomes faulty. Similar problems can occur if the date and time is handled differently by concrete service. For instance, time zones are a key issue in international trade.

* *Security Enforcement Problem*: Security and data privacy are key requirement for service compositions [55, 134]. Assume that the service level agreements (SLAs) between user and composition provider mandate that sensitive data (e.g., location) are not transmitted in plain text. Therefore, service a_5 in the composition uses cryptography to encrypt the coordinates before they are passed to service a_6. If the concrete service receiving the data (s_7 or s_8) has no matching key for decryption, the composition fails and no flight information is received.

* *Network Partitions*: Also the physical deployment of the services must be taken into account. Assume that the network is partitioned, i.e., the services are deployed in different domains with local IP addresses that are not accessible across the network. This is particularly relevant in decentralized compositions where the services interact directly with each other. Moreover, in the centralized execution model of WS-BPEL, it must be ensured that no concrete services are selected which are inaccessible by the composition engine.

4.2.2 Runtime Composition Instances

Dynamic binding is used to select concrete candidate services at execution time. Table 4.1 lists the possible combinations of concrete services for the scenario composition. The column titles contain the composition's abstract services, and each combination has an identifier (#). Each table value represents a concrete service that is used within a certain combination (row) for a certain abstract service (column). In total, 24 combinations exist for the scenario composition.

#	a_1	a_2	a_3	a_4	a_5	a_6	#	a_1	a_2	a_3	a_4	a_5	a_6
c_1	s_1	s_3	s_1	s_1	s_2	s_7	$\mathbf{c_{13}}$	$\mathbf{s_1}$	$\mathbf{s_3}$	$\mathbf{s_1}$	$\mathbf{s_1}$	$\mathbf{s_2}$	$\mathbf{s_8}$
c_2	s_2	s_3	s_1	s_1	s_2	s_7	c_{14}	s_2	s_3	s_1	s_1	s_2	s_8
c_3	s_1	s_3	s_4	s_4	s_2	s_7	c_{15}	s_1	s_3	s_4	s_4	s_2	s_8
$\mathbf{c_4}$	$\mathbf{s_2}$	$\mathbf{s_3}$	$\mathbf{s_4}$	$\mathbf{s_4}$	$\mathbf{s_2}$	$\mathbf{s_7}$	c_{16}	s_2	s_3	s_4	s_4	s_2	s_8
c_5	s_1	s_3	s_5	s_5	s_2	s_7	$\mathbf{c_{17}}$	$\mathbf{s_1}$	$\mathbf{s_3}$	$\mathbf{s_5}$	$\mathbf{s_5}$	$\mathbf{s_2}$	$\mathbf{s_8}$
c_6	s_2	s_3	s_5	s_5	s_2	s_7	c_{18}	s_2	s_3	s_5	s_5	s_2	s_8
c_7	s_1	s_3	s_1	s_1	s_6	s_7	c_{19}	s_1	s_3	s_1	s_1	s_6	s_8
c_8	s_2	s_3	s_1	s_1	s_6	s_7	$\mathbf{c_{20}}$	$\mathbf{s_2}$	$\mathbf{s_3}$	$\mathbf{s_1}$	$\mathbf{s_1}$	$\mathbf{s_6}$	$\mathbf{s_8}$
c_9	s_1	s_3	s_4	s_4	s_6	s_7	$\mathbf{c_{21}}$	$\mathbf{s_1}$	$\mathbf{s_3}$	$\mathbf{s_4}$	$\mathbf{s_4}$	$\mathbf{s_6}$	$\mathbf{s_8}$
c_{10}	s_2	s_3	s_4	s_4	s_6	s_7	c_{22}	s_2	s_3	s_4	s_4	s_6	s_8
c_{11}	s_1	s_3	s_5	s_5	s_6	s_7	c_{23}	s_1	s_3	s_5	s_5	s_6	s_8
$\mathbf{c_{12}}$	$\mathbf{s_2}$	$\mathbf{s_3}$	$\mathbf{s_5}$	$\mathbf{s_5}$	$\mathbf{s_6}$	$\mathbf{s_7}$	c_{24}	s_2	s_3	s_5	s_5	s_6	s_8

Table 4.1: Possible Combinations of Services
(Compositions printed in bold cover all combinations of service pairs connected by data flows.)

Services with data dependencies have a direct influence on one another and create a potential point of failure (as outlined in Section 4.2.1). Hence, for a composition to be tested thoroughly, all concrete service combinations need to be taken into account for service pairs connected by a data flow. Finding the smallest set of test compositions to satisfy this criterion is a hard computational problem (more details are given in Section 4.3.4). For our scenario, the size of this set is 6 (the largest combination set results from pairing all services of a_1 with all of a_3), and one possible solution is the set $\{c_4, c_{12}, c_{13}, c_{17}, c_{20}, c_{21}\}$ (bold print in Table 4.1). Although this example can be easily solved optimally, for problem instances with additional real-world constraints (e.g., that two specific services must never be used in combination), it becomes harder to determine the minimum size of the solution set, and finding an optimal solution becomes infeasible for larger instances.

4.3 Testing of Dynamic Data-Centric Compositions

In this section we establish the key concepts and terminology which build the foundations for testing of dynamic data-centric service compositions. We formalize a unified and platform-independent service composition model in Section 4.3.1. Based on the composition model we define the test coverage goal that should be achieved in Section 4.3.2. Section 4.3.3 illustrates how the platform-independent composition model is mapped to the concrete platforms WS-BPEL and WS-Aggregation.

4.3.1 Service Composition Model and Composition Test Model

We formally define the service composition model and the composition test model, which serve as the basis for the remaining parts of this chapter. Table 4.2 describes the elements of the service composition model. The left column contains symbols and variable names, the middle column defines the respective symbol, and the right column provides examples in reference to the scenario.

Table 4.3 lists the core elements of the composition test model. The set of test composition instances T is a subset of all possible composition instances. The goal we aim for is to keep the size of T small, in order to reduce the effort for test execution. The occurrence count function o indicates the frequency of binding between abstract and concrete services. Finally, we use the function r to map a composition instance to a test result. For simplicity we assume that there are two result states, *success* and *failed*. The mechanism to determine whether a test case has failed or passed is often termed *test oracle* [277]. Details about test oracles in our approach are given in Section 4.5.5.

Symbol	Description	Example
$T \subseteq C$	Set of actual test composition instances to be executed. This is the encoding of a solution and the goal is to minimize its size.	$T_{min} = \{c_4, c_{12}, c_{13}, c_{17}, c_{20}, c_{21}\}$

Symbol	Description	Example
$A = \{a_1, ..., a_n\}$	Set of abstract services that are used in the composition definition.	$A = \{a_1, ..., a_6\}$
$S = \{s_1, ..., s_m\}$	Set of concrete services available in the service-based system.	$S = \{s_1, ..., s_8\}$
$s : A \rightarrow \mathcal{P}(S)$	Function that returns for an abstract service all concrete services providing the required functionality. $\mathcal{P}(S)$ denotes the power set of S.	$s(a_3) = \{s_1, s_4, s_5\}$
$F \subseteq A \times A$	Set of direct data flows (dependencies) between two services. Possible data flows spanning more than two services can be derived from F (see Section 4.3.2).	$F = \{(a_1, a_2), (a_1, a_3), (a_3, a_4), (a_5, a_6)\}$
$C \subseteq [A \rightarrow S]$	Domain of all possible runtime composition instances. The composition function $A \rightarrow S$ maps abstract services to concrete services.	$C = \{c_1, c_2, .., c_{24}\}$ (cf. Table 4.1)
$R^+, R^- \subseteq \mathcal{P}((A \rightarrow S) \times (A \rightarrow S))$	Restrictions in the use and combination of concrete services. Service assignments in R^+ *must* always be used in combination: $\forall c \in C, (r_1, r_2) \in R^+ : r_1 \in c \Leftrightarrow r_2 \in c$. Services in R^- *must not* be combined: $\forall c \in C, (r_1, r_2) \in R^- : r_1 \notin c \vee r_2 \notin c$.	$R^+ = \{ (a_3 \mapsto s1, a_4 \mapsto s_1), (a_3 \mapsto s4, a_4 \mapsto s_4), (a_3 \mapsto s5, a_4 \mapsto s_5) \}$
$c \in C$	Concrete runtime composition instance.	c_4: $a_1 \mapsto s_2$, $a_2 \mapsto s_3$, $a_3 \mapsto s_4$, $a_4 \mapsto s_4$, $a_5 \mapsto s_2$, $a_6 \mapsto s_7$

Table 4.2: Service Composition Model

$o(s_x, a_y) \stackrel{\text{def}}{=} \lvert\{c \in T : c(a_y) = s_x\}\rvert$	Occurrences of concrete service s_x as implementation of an abstract service a_y in any of the compositions in set T.	$o(s_2, a_5) = 3$ (cf. Table 4.1)
$r : C \rightarrow R$, $R = \{success, failed\}$	Function that determines the test result (success, failed) for a given composition instance. This function performs the task of a *test oracle* [277].	$r(c_{20}) = failed$ (assuming that services s_2, s_6 have data incompatibility)

Table 4.3: Composition Test Model

Additionally, we define the minimum (Equation 4.1) and maximum (Equation 4.2) number of uses of any concrete service for a specific abstract service $a_y \in A$ across all compositions in the test set T. The difference between $min(a_y)$ and $max(a_y)$ indicates how uniformly the

test cases in T are generated. For instance, if we consider the test set $\{c_4, c_{12}, c_{13}, c_{17}, c_{20}, c_{21}\}$ from Table 4.1, then $min(a_1) = max(a_1) = 3$ because both concrete services s_1 and s_3 are used three times for a_1. However, the concrete services for a_6 are less uniformly distributed: $min(a_6) = 2, max(a_6) = 4$.

$$min(a_y) \stackrel{\text{def}}{=} min(\{o(s_x, a_y) : s_x \in s(a_y)\}), a_y \in A \tag{4.1}$$

$$max(a_y) \stackrel{\text{def}}{=} max(\{o(s_x, a_y) : s_x \in s(a_y)\}), a_y \in A \tag{4.2}$$

4.3.2 k-Node Data Flow Coverage Metric

Before proceeding with details of the combinatorial test design, we take a closer look at data dependencies. The data flow view of compositions allows for an analysis of the dependencies between services and possible points of failures. As already mentioned in Section 4.2.1, services with direct data dependencies are prone to errors. However, indirect dependencies in a sequence, e.g., $a_1 \rightarrow a_3 \rightarrow a_4$, should be considered as well. In this example, there are two possible dependencies: firstly, if a_3 passes on to a_4 some part of the (unchanged) input it received from a_1, then a hidden, but direct, dependency between a_1 and a_4 is established; secondly, a_3 may operate differently depending on which service is chosen for a_1, and the output of a_3 affects a_4. Hence, we generalize the coverage metrics for dynamic service compositions and introduce the *k-node data flow*.

Definition 5 *A **k-node data flow** d_k is a sequence $\langle a^1_{dk}, a^2_{dk}, \ldots, a^k_{dk}\rangle$ of abstract services, where $a^1_{dk}, a^2_{dk}, \ldots, a^k_{dk} \in A$, such that $\forall j \in \{1, .., k-1\} : (a^j_{dk}, a^{j+1}_{dk}) \in F$. In the special case where $k = 1$, the list contains only one element: $\langle a^1_d\rangle$. $F_k = \{d^1_k, d^2_k, \ldots, d^l_k\}$ denotes the set of all k-node data flows in a service composition definition.*

Definition 6 *A **data flow coverage** $cvg(d_k) \in \mathcal{P}(S^k)$ of a k-node data flow d_k denotes the set of possible concrete service assignments along the path of d_k. More precisely, $cvg(d_k)$ is a set of concrete service sequences of length k, such that all possible service assignments with respect to d_k are covered: $\forall s^1_{dk} \in s(a^1_{dk}), ..., s^k_{dk} \in s(a^k_{dk}) : \langle s^1_{dk}, ..., s^k_{dk}\rangle \in cvg(d_k)$. For each of the service combinations $\langle s^1_{dk}, ..., s^k_{dk}\rangle \in cvg(d_k)$ the output of service s^i_{dk} becomes input of service s^{i+1}_{dk}, $i \in \{1, \ldots, k-1\}$.*

Definition 7 *An SBA is **k-coverage tested** if, for all j-node data flows d_j, $j \in \{1, .., k\}$, there exist test cases such that all service combinations of $cvg(d_j)$ are covered.*

Definition 7, stated in other words, asserts that a composition is *k-coverage tested* if all data flow paths of length of at most k have been tested with all possible service combinations. Figure 4.3 illustrates this for a value of $k = 3$ in our scenario: inter-service dependencies of the data flow view are depicted as trees, and all paths of length 2 and 3 (outlined with light gray background) need to be covered. Note that the k-coverage criterion is only reasonably applicable if the underlying services have been unit tested with appropriate verification and validation techniques. Also, k-node data flow coverage does not take into account the different

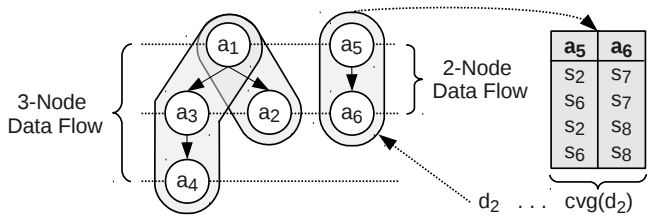


Figure 4.3: Data Flows in Scenario

content of the data passed between concrete services. Hence, the effectiveness of the metric relies on the input data that is used to execute the test cases. The question of which test input is suitable depends on both the service interfaces (e.g., extreme values) and the problem domain. For systematic methods to generation of appropriate test inputs we refer to previous work, e.g., [7, 251, 273].

4.3.3 Mapping the Composition Model to Concrete Platforms

To generate executable tests for our service composition, the platform-specific composition model (query model) must be mapped to the platform-independent composition model (as defined in Section 4.3.1). Reversely, after the test cases have been generated, the concrete composition service bindings must be mapped back to the platform for execution. In the following we discuss the mapping between our composition model and concrete platform models, for WS-BPEL (Section 4.3.3.1) and WS-Aggregation (Section 4.3.3.2). Section 4.5 provides more details about the implementation of the mapping and how the approach is extensible to support further composition paradigms.

4.3.3.1 Mapping for WS-BPEL

The mapping between the WS-BPEL model (more precisely, the relevant parts thereof) and our composition model is depicted in Figure 4.4.

A WS-BPEL *business process* consists of multiple *activities* which define the processing logic. For our purpose, the activities *invoke* and *copy* are most relevant. An *invoke* activity performs a Web service invocation, taking the value of an *input variable* and storing the result to an *output variable*. The *copy* activity uses XPath expressions to process the results received from an invocation and to assign values between output and input variables. In this respect, *invoke* corresponds to abstract service in our composition model, and *copy* represents a data flow. Concrete services are implemented as *Web services* which are addressable using a unique EPR [359]. The *port type* of a Web service defines the set of abstract operations and the messages involved [363]. Hence, the *port type* provides the basis for service candidacy assignment (function $s : A \rightarrow S$) in the composition model. Finally, the *partner link* concept is used in WS-BPEL to model the required relationships between services and partner processes. At runtime, a *partner link* is bound to the location (EPR) of a Web service, which corresponds to the service binding in the composition model.

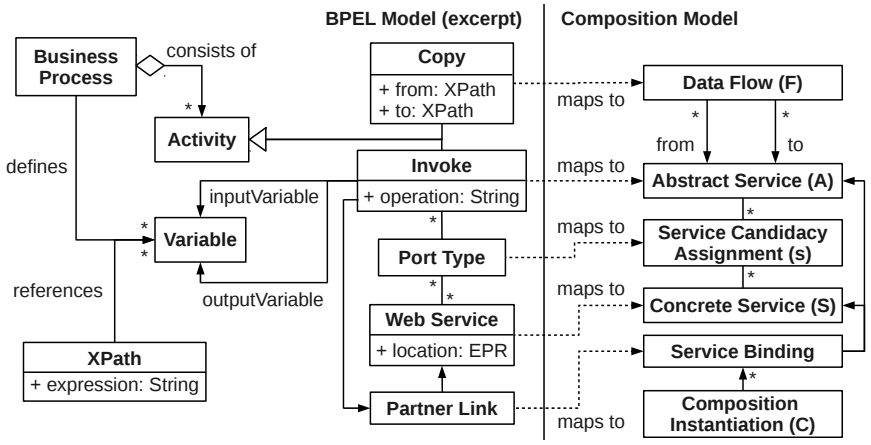


Figure 4.4: Mapping between WS-BPEL Model and Composition Model

4.3.3.2 Mapping for WS-Aggregation

The relationship between the WS-Aggregation query model [139, 148] and the composition model is illustrated in Figure 4.5.

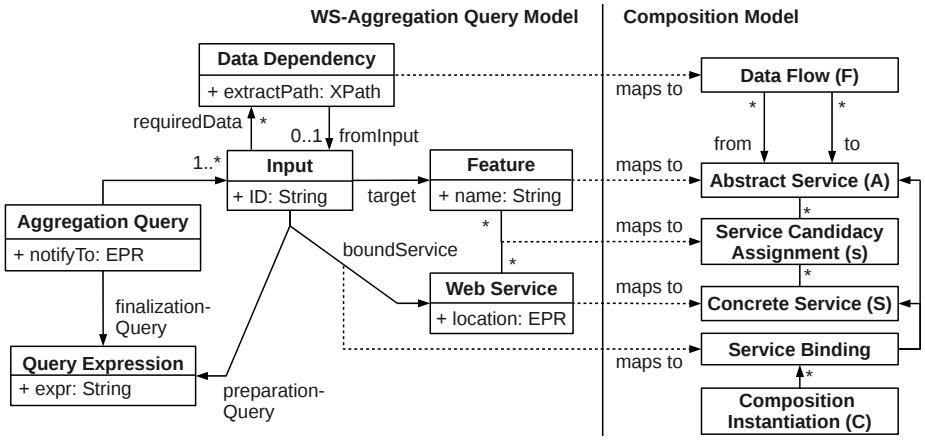


Figure 4.5: Mapping between WS-Aggregation Query Model and Composition Model

The core artifact in the query model is the *aggregation query*, which consists of multiple *input* elements that represent the data received from event streams of external Web services. The attribute *notifyTo* contains the EPR of the client that receives the results from the platform in push mode. For each input, a *preparation query* filters, preprocesses and aggregates the received events. A *finalization query* is used to combine the data into a single result document. The distinction of abstract and concrete services is achieved by an association between *features* and *Web services*: each input targets a certain feature (which maps to an abstract service), and each feature is provided by several Web services (which map to concrete services). This corresponds to the service candidacy assignment (function $s : A \rightarrow S$) in the composition model. Moreover,

inputs are associated with one or more instances of *data dependency*. A dependency between two inputs i_1 and i_2 indicates that certain data (specified via *extractPath*) need to be extracted from the results of i_2 and inserted into i_1. The internal query processor of WS-Aggregation automatically distributes the execution of inputs among the deployed aggregator nodes, resolves dependencies and takes care of proper correlation of the involved event streams. Clearly, input data dependencies can be mapped directly to data flows in the composition model. Finally, a service binding in our model translates to the runtime binding of a Web service for a specific input in WS-Aggregation.

4.3.4 Combinatorial Test Design

Finding the minimal set of composition test cases, taking into account k-node data flow coverage and all of the model's constraints, is a computational problem in the area of combinatorial test design [71, 242], which is known to be NP-hard [66]. The number of possible compositions is exponential in the number of services. It is simple to construct any valid solution, as well as to determine the validity of existing solutions, however, no polynomial-time algorithm exists that can guarantee to deliver the optimal solution. The size difference between 1) a (near-)optimal solution and 2) a test set obtained by full enumeration (cf. Table 4.1) or a simple construction heuristic can be significant. The implication is that in case of 1) the test generation takes longer and the test set executes faster, whereas for 2) the test set can be generated fast and executing the test takes longer due to the increased size of the solution. Note that we speak of *near*-optimality, because in any case we seek for a practicable approach that executes in acceptable time. The optimization details for finding minimal sets of test cases for service compositions are discussed in the following.

Our goal is to keep the required testing effort minimal. Hence, the universal objective function attempts to minimize the number of composition test cases to execute, which is expressed in Equation 4.3.

$$|T| \rightarrow min \tag{4.3}$$

In addition to this universally valid criterion, it is advantageous to specify preferences concerning service reuse. A composition tester who has knowledge about the quality of the individual services may give precedence to certain concrete services. Applied to the scenario, consider that s_1 is known to be well-tested, whereas s_2 is published by a less reliable provider. We introduce the additional symbol $p(s_x, a) \in (0, 1)$ to specify the desired probability that service s_x should be bound to abstract service a in the test cases (e.g., $p(s_1, a_1) = 0.2, p(s_2, a_1) = 0.8$). Based on that, our alternative objective function minimizes the total deviation between the actual and the expected occurrences of all concrete services in the generated test cases (Equation 4.4). Note that the special case in which all probabilities are equal, $\forall a \in A : \forall s_x, s_y \in s(a) : p(s_x, a) = p(s_y, a)$, expresses that all services should be tested with the same intensity. Further note that the extreme values 0.0 and 1.0 are not in the value domain of $p(s_x, a)$; the effect of 0% or 100% probability can be achieved by completely removing the service s_x from the candidate list $s(a)$, or by removing all other services from the candidate list, respectively.

$$|T| + \sum_{a \in A} \sum_{s_y \in s(a)} \left| \frac{o(s_y,a)}{|s(a)|} - p(s_y, a) \right| \rightarrow min \tag{4.4}$$

Equation 4.3 is the default optimization target, if there is no known reason for preferring the testing of some concrete services over others. If there is a reason for a preference, it can be easily expressed by using probabilities with Equation 4.4. Both of the alternative objective functions defined above are subject to the following hard constraints:

$$c(a) \neq \emptyset, \forall c \in T, a \in A \tag{4.5}$$

$$\bigcup_{c \in C, a \in A} c(a) = \bigcup_{a \in A} s(a) \tag{4.6}$$

$$\begin{aligned} \forall \quad & j \in \{1, .., k\} : \\ \forall \quad & d_j \in F_j : \\ \forall \quad & s^1_{dj} \in s(a^1_{dj}), ..., s^j_{dj} \in s(a^j_{dj}) : \\ \exists \quad & c \in T : c(a^1_{dj}) = s^1_{dj} \wedge ... \wedge c(a^j_{dj}) = s^j_{dj} \end{aligned} \tag{4.7}$$

Equation 4.5 signifies that a composition must bind one concrete service to each abstract service. Equation 4.6 expresses that each concrete service that implements one of the abstract services needs to be used at least once in the final set of compositions. Finally, Equation 4.7 ensures that the service composition is k-coverage tested, i.e., that all services with direct and indirect data dependencies (up to flows of length k), are tested with all combinations of concrete services. Note that there is a logical connection between the criteria in Equations 4.6 and 4.7, as follows. According to Equation 4.7, the composition is k-coverage tested (for $k \geq 1$). This trivially implies that the composition is 1-coverage tested. Now, 1-coverage tested means that every single abstract service is tested (at least once) with each of its concrete candidates, which corresponds to the semantics of Equation 4.6. So, in fact Equation 4.6 is an indirect implication of Equation 4.7; however, both equations have been included here for clarity.

4.3.5 Determining Faulty Services and Incompatible Configurations

So far, we have discussed that the composition model is used to compute a (minimal) set T of composition test cases. The detailed procedure for generating and executing the tests will be addressed later in Section 4.5. For now, we assume that the tests have been executed and for each instantiation $c \in T$, a test outcome has been recorded. The test results contain data for non-functional properties, such as the processing time, as well as the functional test data, which informs about whether the test case has succeeded or failed. Recall from Section 4.3.1 that the functional result of a single test case is expressed as a boolean function $r : C \rightarrow \{success, failed\}$.

4.3.5.1 The Fault Participation and Fault Contribution Metrics

Having determined the functional test result of each composition instance, we are able to analyze which service or which combination of services have caused tests to fail. In fact, we are interested not only in detecting the existence of a fault, but in determining faulty concrete services or incompatible combinations thereof. Let $x : A \rightarrow S$ denote an assignment of concrete services to abstract services, which represents any subset of the service bindings applied in a composition $c \in C$, that is, $x \subseteq c$. For instance, take the function values of the scenario composition c_4 in set-of-pairs notation, $c_4 = \{(a_1, s_2),(a_2, s_3),(a_3, s_4),(a_4, s_4),(a_5, s_2),(a_6, s_7)\}$. One possible subset of this composition would be a binding $x = \{(a_4, s_4),(a_5, s_2)\}$. Note that x is also a subset of c_3, c_{15} and c_{16}. The function $succ : \mathcal{P}(C) \rightarrow \mathcal{P}(C)$ returns for a set of service compositions $Y \in \mathcal{P}(C)$ those for which a successful test result has been recorded: $succ(Y) := \{c \in Y \mid r(c) = success\}$. Analogously, the set of failed test compositions in Y is $fail(Y) := \{c \in Y \mid r(c) = failed\}$. Finally, we use the function $match : ((A \rightarrow S) \times \mathcal{P}(C)) \rightarrow \mathcal{P}(C)$, $(x, Y) \mapsto \{c \in Y \mid \forall (a, s) \in x : c(a) = s\}$ to determine all compositions in a set Y, which contain all assignments defined by a binding x.

We then utilize the following two indicators:

* The fault *participation* rate $part(x, T)$ of a binding x denotes the percentage of failed compositions in T that contain all bindings defined by x (in relation to all failed compositions in T):
  $part(x, T) := \frac{|fail(match(x,T))|}{|fail(T)|}$
* The fault *contribution* rate $cont(x, T)$ of a binding x denotes the percentage of compositions in T matching x that failed in the test (in relation to all compositions in T matching x):
  $cont(x, T) := \frac{|fail(match(x,T))|}{|match(x,T)|}$

Information Retrieval	Test Result Analysis
Precision = $\frac{\#(\text{Relevant Docs in Answer Set})}{\#(\text{Answer Set})}$	**Contribution** = $\frac{\#(\text{Failed Tests with Assignment } x)}{\#(\text{Tests with Assignment } x)}$
Recall = $\frac{\#(\text{Relevant Docs in Answer Set})}{\#(\text{Relevant Docs})}$	**Participation** = $\frac{\#(\text{Failed Tests with Assignment } x)}{\#(\text{Failed Tests})}$

Figure 4.6: Analogy Between Fault Contribution/Participation and Precision/Recall

The fault participation expresses the degree to which binding x has been "involved" in the faulty compositions. A higher value of $part(x, T)$ indicates that x is likely (partly) responsible for the fault. On the other hand, fault contribution expresses the degree to which the test cases containing x have led to a faulty result. That is, $cont(x, T)$ indicates whether there have been any successful results for tests including x. Note that these indicators are related to *precision* and *recall* [18], two measures often used in machine learning and in Information Retrieval (IR).

	#	$\mathbf{a_1}$	$\mathbf{a_2}$	$\mathbf{a_3}$	$\mathbf{a_4}$	$\mathbf{a_5}$	$\mathbf{a_6}$	**Result**
Minimal Test Set	$\mathbf{c_4}$	s_1	s_3	s_1	s_1	s_2	s_8	Failed
	$\mathbf{c_{13}}$	s_2	s_3	s_4	s_4	s_2	s_7	Succeeded
	$\mathbf{c_{17}}$	s_1	s_3	s_5	s_5	s_2	s_8	Failed
	$\mathbf{c_{20}}$	s_2	s_3	s_1	s_1	s_6	s_8	Succeeded
	$\mathbf{c_{21}}$	s_1	s_3	s_4	s_4	s_6	s_8	Succeeded
	$\mathbf{c_{12}}$	s_2	s_3	s_5	s_5	s_6	s_7	Succeeded
Additional Tests	$\mathbf{c_2}$	s_2	s_3	s_1	s_1	s_2	s_7	Succeeded
	$\mathbf{c_{10}}$	s_2	s_3	s_4	s_4	s_6	s_7	Succeeded
	$\mathbf{c_{16}}$	s_2	s_3	s_4	s_4	s_2	s_8	Failed

Table 4.4: Illustrative Test Results of Scenario Composition Test Cases T (cf. Table 4.1)

We can draw an analogy between document search in IR and finding incompatible bindings in services testing, which is illustrated in Figure 4.6. The precision of a performed search in IR expresses the fraction of retrieved documents that are relevant with respect to the search criteria. Similarly, fault contribution considers the fraction of test cases that are "relevant", i.e., failed. On the other hand, recall answers the question to which degree the relevant documents have been found by a search. Likewise, fault participation relates the failed test results of a certain service assignment to the set of all relevant (failed) test cases. To obtain a one-dimensional measure, precision and recall are often combined into a harmonic mean, denoted *F measure* [18] (or *F score*), which is also conceivable as an alternative for $part$ and $cont$.

Table 4.4 lists example test results for different test cases of the scenario composition, based on which we discuss the relevance of the indicators $part$ and $cont$ in more detail. The first six entries in the table represent the minimal set of test cases that is required to satisfy the k-node data flow coverage criterion (k=3). For illustration purposes, we have injected an incompatibility between the concrete services s_2 and s_8 with respect to the data flow between the abstract services a_5 and a_6. The goal is to detect this incompatibility based on the test results (*Failed* or *Succeeded*) of the individual test cases, which is illustrated in Table 4.5. This table contains the values for $part$ and $cont$ for six randomly chosen sample bindings. The number of bindings for which we have to compute the indicators depends on the number of abstract and concrete services, as well as the number and length of the composition's data flow paths. In the case of our example, 34 distinct combinations of bindings exist (13 bindings for single services, 15 bindings along the four different 2-node data flows, and 6 bindings along the 3-node data flow). Six out of the 34 binding combinations are printed in Table 4.5. The most interesting cases are those for which $cont$ equals 1, which in this example occurs, as expected, only for the faulty service combination $x : a_5 \mapsto s_2, a_6 \mapsto s_8$. Additionally, for this combination also the value of $part$ is 1, because the example contains only a single faulty service combination, and hence no other combination participates in all faulty test cases. Technically, the binding $x : a_2 \mapsto s_3, a_5 \mapsto s_2, a_6 \mapsto s_8$ also has a value of 1 for both indicators, which results from the fact that s_3 is the only concrete service available for a_2. Our evaluation in Section 4.6.4 discusses how the test indicators evolve if multiple service combinations are causing faults in a composition.

Combination x :	$a_1 \mapsto s_1$	$a_2 \mapsto s_3$	$a_5 \mapsto s_2$	$a_1 \mapsto s_2$, $a_2 \mapsto s_3$	$a_5 \mapsto s_2$, $a_6 \mapsto s_8$	$a_1 \mapsto s_2$, $a_3 \mapsto s_1$, $a_4 \mapsto s_1$	...
Values for Minimal Test Set:							
$\mathbf{part}(\mathbf{x}, \mathbf{T})$	1.0	1.0	1.0	0.0	1.0	0.0	...
$\mathbf{cont}(\mathbf{x}, \mathbf{T})$	0.6667	0.3333	0.6667	0.0	1.0	0.0	...
Values for Minimal Test Set and Additional Tests:							
$\mathbf{part}(\mathbf{x}, \mathbf{T})$	0.6667	1.0	1.0	0.3333	1.0	0.0	...
$\mathbf{cont}(\mathbf{x}, \mathbf{T})$	0.6667	0.3333	0.6	0.1667	1.0	0.0	...

Table 4.5: Exemplary Service Combinations for the Scenarios (6 out of 34 Total Combinations) with Fault Participation and Fault Contribution

4.3.5.2 Additional Tests Beyond the k-Coverage Minimal Test Set

The described technique is suitable to automatically analyze the test results and point the tester towards faults in the service composition. Note that the discussed indicators depend on the number of test results at hand. In general, a higher number of test cases provides stronger evidence. For instance, after executing the first test composition (c_4), which failed, trivially all service assignments of this composition have participated in and contributed to 100% of the faults. Table 4.5 also illustrates that the values change when additional tests (c_2,c_{10},c_{16}) are executed on top of the minimal test set with respect to k-node test coverage. For instance, $part(x, T)$ is 1.0 for $x : a_1 \mapsto s_1$, and the minimal test set of six compositions, but we can exclude x from the potentially problematic assignments as the value drops to 0.6667 after executing the additional tests. Conversely, the assignment $x : a_1 \mapsto s_2, x : a_2 \mapsto s_3$ is not associated with any failed results in the minimal test set, but shows a failed result in the additional tests. This example shows that the minimal test set is generally sufficient for a meaningful test result analysis, but evidently additional test data leads to more precise localization possibilities. In any case, the coverage criterion ensures that all crucial service combinations are tested with respect to direct and transitive data dependencies.

4.3.5.3 Limitations and Advanced Fault Localization Techniques

It should be noted that the fault localization based on the contribution/participation metrics has some limitations. First, it is easier to detect a single data flow incompatibility than to analyze faults in compositions that contain several combinations of incompatible services. The reason is that if multiple incompatibilities exist, the $part$ or $cont$ values will likely never reach a value of 1, and these cases require the tester to define a threshold value or to investigate those service combinations with high $part$ or $cont$ values manually. This aspect is evaluated in more detail in Section 4.6. Moreover, this method of test result analysis is reliable only if we can assume that faults happen deterministically, in the sense that any faulty service assignment x always leads to a failed result in all test cases it participates in: $|fail(match(x, T))| > 0 \Rightarrow |match(x, T)| = |fail(match(x, T))|$. Formulated differently, this means that all tests have to be repeatable, and indeterministic transient faults may lead to incorrect analysis results.

To overcome these limitations, we have developed a more sophisticated fault localization technique which is able to deal with transient faults and changing fault conditions. The details are discussed in Section 4.4.

4.4 Advanced Fault Localization for Transient and Changing Faults

This section extends on the testing approach discussed in Section 4.3 and studies advanced patterns of real-world fault conditions in SBAs. We introduce a novel fault localization technique which is capable of handling transient faults and changing fault conditions. The fault localization discussed here fulfills two purposes: first, it integrates with the upfront testing approach and allows for detailed result analysis of generated test executions; second, it can be applied in the production environment of an SBA to identify and localize faults during run time.

Section 4.4.1 refines the model of SBAs under test, extending the previous model from Section 4.3.1 with services' input/output parameters as well as the notion of execution traces. Sections 4.4.2 and 4.4.3 discuss preprocessing and machine learning techniques used to learn rules which describe the reasons for faults based on the collected trace data.

4.4.1 Extended Service Composition Model

We extend the service composition model from Section 4.3.1 with the notion of service parameters and execution traces (see Table 4.6). The domain of possible input parameters P, each defined by name (N) and domain of possible data values (D) is represented by $P = N \times D$. Function p returns all inputs required by an abstract service, and function d returns the value domain for a given parameter. Moreover, we define $T = \langle t_1, \ldots, t_k \rangle$ as the sequence of logged execution traces $t_x : K \rightarrow V$ in chronological order, mapping the set of keys (K) to values (V). For each key-value mapping, abstract services (A) map to concrete services (S), whereas parameter names (N) map to values in the parameter domain (D).

Symbol	**Description**	**Example**
N	Set of service parameter names.	$N = \{custID, premium, \ldots\}$
D	Domain of service parameter values.	$D = \{'joe123', 'aliceXY', true, false, \ldots\}$
$P = N \times D$	Domain of possible service parameters.	$P = \{(custID,' joe123'), (custID,' aliceXY'), \ldots\}$
$d : N \rightarrow \mathcal{P}(D)$	Returns the value domain for a given parameter.	$d(custID) = \{'joe123',' aliceXY'\}$
$p : A \rightarrow \mathcal{P}(N)$	Returns all input parameters required by an abstract service.	$p(a_1) = \{custID\}$ $p(a_2) = \{premium\}$
$K = A \cup N$	Set of keys encoding an execution trace, consisting of abstract services and parameter names.	$K = \{a_1, a_2, \ldots, custID, premium, \ldots\}$

$V = S \cup D$	Set of values encoding an execution trace, consisting of concrete services and parameter values.	$V = \{s_1, s_2, \ldots, \, 'joe123', 'aliceXY', true, false, \ldots\}$
$T = \langle t_1, \ldots, t_k \rangle$ $t_x : K \to V$	Sequence of logged execution traces in chronological order. Each trace is encoded as a mapping from keys to values.	$T = \{\{a_1 \mapsto s_1, \, custID \mapsto 'joe123', \ldots\}, \ldots\}$

Table 4.6: Extensions to the Service Composition Model

Table 4.6 contains the respective symbols and a brief example for each model artifact. For instance, the table lists an abstract service a_1 which is implemented by concrete services s_1 and s_2. Service a_1 requires as parameter a customer identifier ($custID$) that can take values $'bob123'$ or $'aliceXY'$.

Summarizing the model, the core idea of our approach is to analyze log traces of SBA executions for fault localization. We consider two classes of properties as part of the traces: 1) runtime binding of abstract to concrete services, 2) service input parameters, i.e., data provided by the user to the application as well as data flowing between services.

4.4.2 Trace Data Preparation

Table 4.7 lists an excerpt of six exemplary traces for an imaginative customer-oriented SBA. The table contains multiple rows which represent the traces ($t_1, \ldots, t_6$); the columns contain the bindings for the abstract services ($a_1, a_2, a_3, \ldots$), parameter values ($custID, premium, \ldots$), and the success result of the trace ($r(x)$). Two exemplary parameters for a customer service are in the table: $t_x(custID)$ denotes the identifier of a customer, and the boolean parameter $t_x(premium)$ specifies whether we are dealing with a regular customer or a high-paying premium customer.

We follow the typical machine learning terminology and denote the column titles as *attributes* and the rows starting from the second row as *instances*. The first attribute (t_x) is the instance identifier, and $r(x)$ is denoted *class attribute*.

$\mathbf{t_x}$	$\mathbf{t_x(a_1)}$	$\mathbf{t_x(a_2)}$	$\mathbf{t_x(a_3)}$	..	$\mathbf{t_x(custID)}$	$\mathbf{t_x(premium)}$	..	$\mathbf{r(x)}$
$\mathbf{t_1}$	s_1	s_3	s_7	..	$'joe123'$	$false$	..	$success$
$\mathbf{t_2}$	s_2	s_4	s_6	..	$'aliceXY'$	$true$	..	$success$
$\mathbf{t_3}$	s_1	s_5	s_8	..	$'joe123'$	$false$	..	$failed$
$\mathbf{t_4}$	s_2	s_5	s_8	..	$'bob456'$	$true$	..	$success$
$\mathbf{t_5}$	s_2	s_4	s_7	..	$'aliceXY'$	$true$	..	$success$
$\mathbf{t_6}$	s_1	s_4	s_8	..	$'lindaABC'$	$false$	..	$failed$
..								

Table 4.7: Example Traces with Service Binding and Parameter Values

The number of trace attributes and combinations of attribute values can grow very large. To estimate the number of possible traces for a medium sized application, consider an SBA using 10 abstract services ($|A| = 10$), 3 candidates per abstract service ($|s(a_x)| = 3\ \forall a_x \in A$), 3 input parameters per service ($|p(a_x)| = 3\ \forall a_x \in A$), and 100 possible data values per parameter ($|d| = 100\ \forall a_x \in A, (n,d) \in p(a_x)$). The total number of possible execution traces in this SBA is $3^{10} * 100^{3^{10}} = 5.9049 * 10^{64}$. Efficient localization of faults in such large problem spaces evidently poses a huge algorithmic challenge. Even more problematically, the problem space becomes infinite if the service parameters use non-finite data domains (e.g., $String$). The first step towards feasible fault analysis is to reduce the problem space to the most relevant information. We propose a two-step approach:

1. Identifying (ir)relevant attributes: The first manual preprocessing step is to decide, based on domain knowledge about the SBA, which attributes are relevant for fault localization. For instance, in an e-commerce scenario we can assume that a unique customer identifier ($custID$) does not have a direct influence on whether the execution succeeds or fails. Per default, all attributes are deemed relevant, but removing part of the attributes from the execution traces helps to reduce the search space.
2. Partitioning of data domains: Research on software testing and dependability has shown that faults in programs are often not solely incurred by a single input value, but usually depend on a range of values with common characteristics [353]. Partition testing strategies therefore divide the domain of values into multiple sub-domains and treat all values within a sub-domain as equal. As a simple example, consider a service parameter with type $Integer$ (i.e., $\{-2^{31}, .., +2^{31} - 1\}$), a valid partitioning would be to treat negative/positive values and zero as separate sub-domains: $\{\{-2^{31}, .., -1\}, \{0\}, \{1, .., +2^{31} - 1\}\}$. If explicit knowledge about suitable partitioning is available, input value domains can be partitioned manually as part of the preprocessing. However, efficient methods have been proposed to automatize this procedure (e.g., [67]).

4.4.3 Learning Rules from Decision Trees

Using the preprocessed trace data, we strive to identify the attribute values or combinations of attribute values that are likely responsible for faults in the application. For this purpose, we utilize decision trees [270], a popular technique in machine learning. This has the advantage that the decision making of the resulting trees can be easily comprehended; their knowledge can be distilled for the purpose of fault localization. Also, decision tree training with state of the art algorithms like C4.5 [270] results in comparably fast learning speeds, compared to other machine learning approaches.

Figure 4.7 illustrates decision trees based on the example traces in Table 4.7. The figure shows two variants of the same tree which classifies non-premium customers with a specific service binding ($t_x(a_3) = s_8$). The inner nodes are decision nodes which divide the traces search space, and the leaf nodes indicate the trace results. The left-hand side of the figure shows a regular decision tree where each decision node splits according to the possible values of an attribute. The right-hand side shows the same tree with binary split (i.e., each decision node has two outgoing edges).

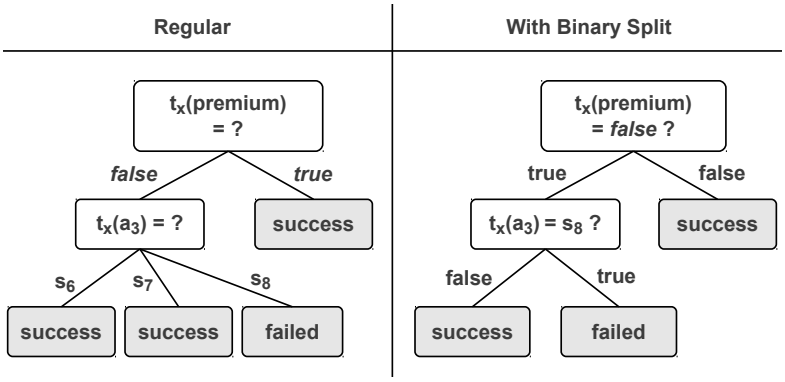


Figure 4.7: Exemplary Decision Tree in Two Variants

The decision tree with binary split is used to automatically derive incompatible attribute values. The basic procedure is to loop over all $failed$ leaf nodes and create a combination of attribute assignments along the path from the leaf to the root node. The detailed algorithm is presented in Algorithm 2. For each $failed$ leaf node, a set E_{temp} is constructed which contains the conditions that are true along the path. The total set of all such condition combinations is denoted E_I. Our approach exploits the simple structure of decision trees for extracting incompatibility rules; other popular classification models (e.g., neural networks) have much more complex internal structures which make it harder to extract the principal attributes responsible for the output [298].

Algorithm 2 Obtain Incompatibility Rules from Decision Tree

1: $E_I \leftarrow \emptyset$
2: **for all** $failed$ leaf nodes **as** n **do**
3: $path \leftarrow$ path of nodes from n to root node
4: $E_{temp} \leftarrow \emptyset$
5: **for all** decision node along $path$ **as** d **do**
6: $c \leftarrow$ condition of d
7: **if** c is $true$ along $path$ **then**
8: $E_{temp} \leftarrow E_{temp} \cup c$
9: **end if**
10: **end for**
11: $E_I \leftarrow E_I \cup E_{temp}$
12: **end for**
13: **for all** $E_x, E_y \in E_I$ **do**
14: **if** E_x is covered by E_y **then**
15: $E_I \leftarrow E_I \setminus E_x$
16: **end if**
17: **end for**

4.4.4 Coping with Transient Faults

So far, we have shown how trace data can be collected, transformed into a decision tree, and used for obtaining rules which describe which configurations have led to a fault. The assumption so far was that faults are deterministic and static. However, in real-life systems which are influenced by various external factors, we have to be able to cope with temporary and changing faults. Our approach is hence tailored to react to such irregularities in dynamically changing environments.

A temporary fault manifests itself in the log data as a trace $t \in T$ whose result $r(t)$ is supposed to be $success$, but the actual result is $r(t) = failed$. Such temporary faults can lead to a situation of contradicting instances in the data set. Two trace instances $t_1, t_2 \in T$ contradict each other if all attributes are equal except for the class attribute:
$\{(k, v) \mid (k, v) \in t_1\} = \{(k, v) \mid (k, v) \in t_2\}, r(t_1) \neq r(t_2)$.

Fortunately, state-of-the-art decision tree induction algorithms are able to cope with such temporary faults which are considered as noise in the training data (e.g., [4]). If the reasons for faults within an SBA change permanently, we need a mechanism to let the machine learning algorithms forget old traces and train new decision trees based on fresh data. Before discussing strategies for maintaining multiple decision trees in Section 4.4.4.2, we first briefly discuss in Section 4.4.4.1 how the accuracy of an existing classification model is tested over time.

4.4.4.1 Assessing the Accuracy of Decision Trees

Let D be the set of decision trees used for obtaining fault combination rules. We use the function $rc : (D \times \{1, \ldots, k\}) \rightarrow \{success, failed\}$, where k is the highest trace index, to express how a desicion tree classifies a certain trace. Over a subset $T_d \subseteq T$ of the traces classified by a decision tree d, we assess its accuracy using established measures true positives (TP), true negatives (TN), false positives (FP), and false negatives (FN) [18]:

- True Positives: $TP(T_d) = \{t_x \in T_d \mid rc(d, x) = failed \wedge r(x) = failed\}$
- True Negatives: $TN(T_d) = \{t_x \in T_d \mid rc(d, x) = success \wedge r(x) = success\}$
- False Positives: $FP(T_d) = \{t_x \in T_d \mid rc(d, x) = failed \wedge r(x) = success\}$
- False Negatives: $FN(T_d) = \{t_x \in T_d \mid rc(d, x) = success \wedge r(x) = failed\}$

From the four basic measures we obtain further metrics to assess the quality of a decision tree. The *precision* expresses how many of the traces identified as faults were actually faults ($TP/(TP + FP)$). *Recall* expresses how many of the faults were actually identified as such ($TP/(TP+FN)$). Finally, the *F1 score* [131] integrates precision and recall into a single value (harmonic mean):
$F1(d) = 2 * \frac{precision \cdot recall}{precision + recall}$

4.4.4.2 Maintaining a Pool of Decision Trees

In the following we discuss our approach to cope with changing fault conditions over time, based on a sample execution of the system model introduced in Section 4.4.1.

Figure 4.8 illustrates a representative sequence of execution traces ($\{t_1, t_2, t_3, \ldots\}$); time progresses from the left-hand side to the right-hand side of the figure. In the top of the figure the

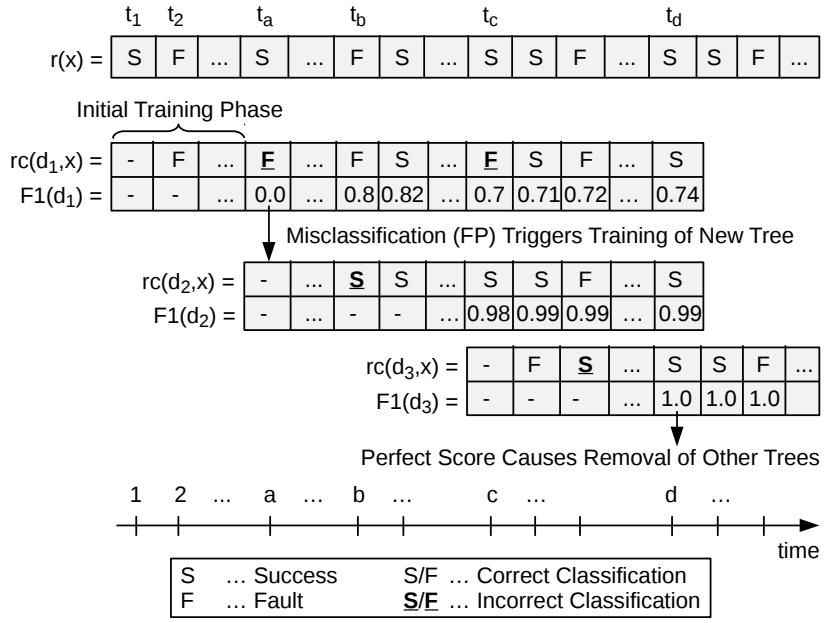


Figure 4.8: Maintaining Multiple Trees to Cope with Changing Faults

trace results ($r(t_x)$) are printed, where "S" represents $success$ and "F" represents $failed$. As the traces arrive with progressing time we utilize deduction algorithms to learn decision trees from the data. At time point 1, the decision tree d_1 is initialized and starts the training phase. The learning algorithm has an initial training phase which is required to collect a sufficient amount of data to generate rules that pass the required statistical confidence level. After the initial training phase the quality of the decision tree rules is assessed by classifying new incoming traces. In Figure 4.8 correct classifications are printed in normal text, while incorrect classifications are printed in bold underlined font.

We have marked four particularly interesting time points (a, b, c, d) in Figure 4.8, which we discuss in the following.

1. At time a the tree d_1 misclassifies the trace t_a as a false positive. This triggers the parallel training of a new decision tree d_2 based on the traces starting with t_a.
2. A false negative by d_2 occurs at time b. However, since this happens during the initial training phase of d_2, we simply regard the trace t_b as useful information for the learner and add it to the training set. No further action is required.
3. Time point c contains another false positive misclassification of d_1. In the meantime, $F1(d_1)$ had risen due to some correct classifications, but now the score is pushed down to 0.7. Again, as in time point a, the generation of a new tree d_3 is triggered.
4. At time d the environment seems to have stabilized and decision tree d_3 reached a state with perfect classification ($F1(d_3) = 1$). At this point, the remaining decision trees are rejected. The old trees are still stored for reference, but are not trained with further data to save computing power.

4.5 Implementation: The TeCoS Framework

The work discussed in this chapter is integrated into the TeCoS (*Te*st *Co*verage for *S*ervice-based systems) framework, which we briefly present in the following. TeCoS aims at providing a holistic view of SBAs by providing various coverage metrics, thereby operating both on the level of the service API and on the service composition level. In previous work (not covered in detail within this thesis), we have presented how TeCoS collects and stores invocation messages at runtime in order to compute API coverage metrics of single services [142], whereas in this thesis the focus is on achieving test coverage for composed services. The task of TeCoS is to provide the data necessary to construct the composition model, as well as to generate and execute the composition test cases. Figure 4.9 sketches the TeCoS architecture in the context of an exemplary service composition. The Web services are provided and hosted by three service providers. A service integrator defines and publishes the composition on top of the services, e.g., as a WS-BPEL process or a WS-Aggregation query. End users invoke the composition, and may also make use of the atomic services ($s_1, ..., s_8$).

The TeCoS Service Broker is a centralized entity that offers a Service Registry to store service metadata, and a Tracing Service which is responsible for logging invocations that occur in the SBA, both for the atomic services and for the composite service (e.g., WS-BPEL process). Our implementation provides an invocation interceptor that is easily plugged into the Java Web services framework. The traces and coverage data can be inspected via the Web User Interface. Here, we do not consider data protection and privacy issues in much detail, but service providers and integrators may choose to host their own instances of the Tracing Service and receive events about newly added data from the broker. For privacy and access control related issues in service compositions, we direct the reader to Chapter 5 of this thesis. The core component, which orchestrates the testing process is the Test Manager (TM). The TM repetitively invokes the Composite Web Service (WS), and each repetition covers one test case. For instance, if the Composite WS is implemented in WS-BPEL, the process definition is instrumented in such a way that it dynamically retrieves from the TM the EPR of the services to invoke (see Section 4.5.4). A service EPR uniquely identifies a service instance and contains the technical information required to invoke the service, such as the service name and its location URL.

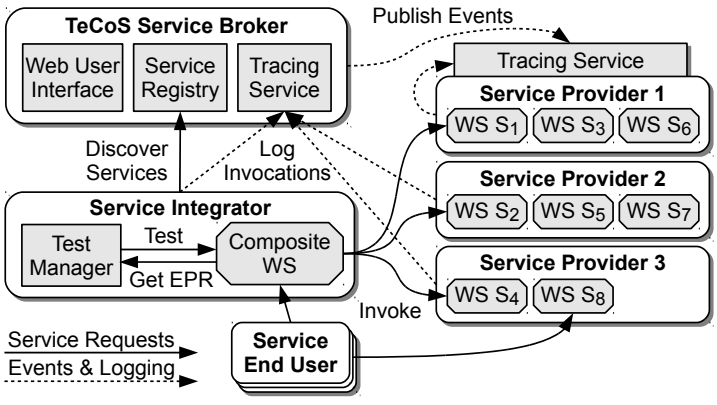


Figure 4.9: Architecture of the TeCoS Framework

TeCoS logs every execution of the service composition in order to collect traces that can be used in the FoCuS solver as described in Section 4.5.3. A crucial point about logging a composite service is to correlate the single invocations performed by one of its runtime instances. Consider two users invoking the WS-BPEL process simultaneously. The order of the subsequent service invocations performed by the two process instances is indeterministic, and hence the exchanged messages need to carry additional information to allow for identification of process instances. Similar to the solution proposed in [205], we use WS-BPEL code instrumentation to inject unique identifiers in the exchanged messages. SOAP is the messaging protocol used by Web services. Whereas the SOAP *body* carries the payload of the message, the SOAP *header* is used to transmit additional information. We use the SOAP header to include the identifier of the WS-BPEL process instance in each invocation it carries out.

In the following we describe the implemented end-to-end solution in more detail, particularly focusing on how test cases are generated and executed in WS-BPEL and WS-Aggregation. Section 4.5.1 shows how target composition platforms can be plugged into TeCoS by means of an adapter mechanism. To further discuss the details of the single steps that have been outlined earlier in Figure 4.1, we divide the procedure of the end-to-end approach into three parts: the test preparation activities before the optimization takes place (Section 4.5.2), the activity where FoCuS obtains the optimized solution (Section 4.5.3), and the part after optimization where the tests are generated, executed and analyzed (Section 4.5.4). Finally, Section 4.5.6 covers implementation details of the fault localization platform discussed in Section 4.4.

4.5.1 Integration of Target Platforms via Extensible Adapter Mechanism

The testing approach and coverage criterion proposed in this thesis are applicable to various service composition techniques. The basic requirement is that the composition model needs to provide a mapping between abstract and concrete services and that compositions can be instantiated with a particular service assignment. Two sample techniques, WS-BPEL and WS-Aggregation, have been exemplified, and as part of our ongoing work we are integrating additional platforms. For instance, in the emerging field of service mashups [32] the Enterprise Mashup Markup Language [6] (EMML) is one recommended way of building light-weight service compositions. Another example is Windows Workflow Foundation [73], a programming model for service-based business processes.

TeCoS provides a flexible adapter mechanism to map the combinatorial test design to concrete composition platforms. The model for these platform-specific test adapters is depicted as a UML class diagram in Figure 4.10. The core interface is `TestAdapter`, which defines methods to initialize an adapter for a certain composition definition (`initialize`), determine the list of candidate services for an abstract service (`getServices`), construct a request for a certain composition instance (`constructRequest`), and execute a single test request (`executeTest`). The final report is composed of one test result for each composition instance, containing the output data, a boolean flag indicating whether the test has failed, and a collection of QoS metrics that are particularly interesting for testing the quality of event-based compositions. Specialized adapters implement the interface `TestAdapter` and provide an extension of the `CompositionDefinition` class. The remaining classes in Figure 4.10 are agnostic of concrete platforms and define the generic composition model with dynamic service assignment.

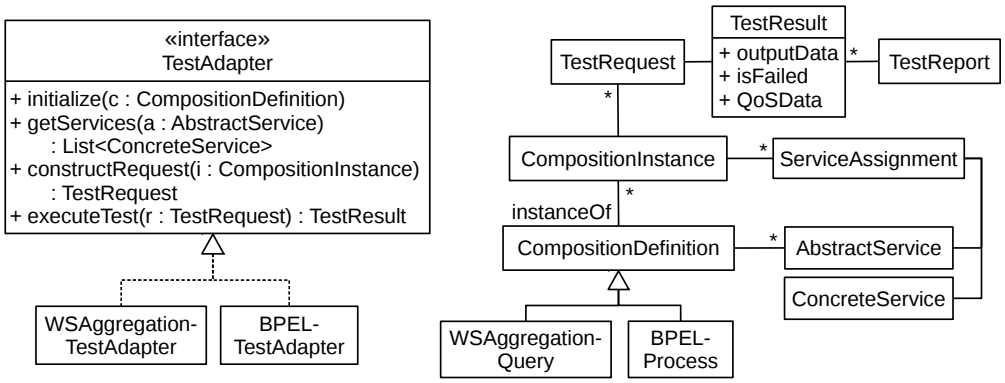


Figure 4.10: Model for Platform-Specific Composition Test Adapters

4.5.2 Test Preparation Steps

The first step in the test preparation is to create the data flow view from the composition definition, which in our scenarios are a WS-BPEL document and a WS-Aggregation query, respectively. In WS-Aggregation, the data flow view can be directly derived from the user request. Each composition query is composed of multiple inputs (which correspond to the abstract services) that receive data from concrete services [139]. The query model allows to define data dependencies between inputs which directly map to the data flows between services as considered in this thesis.

```
1 <bpel:assign xmlns:bpel="...">
2  <bpel:copy>
3   <bpel:from>$a3_out.part1//hotel[$i]</bpel:from>
4   <bpel:to>$a4_in.part1//hotel</bpel:to>
5  </bpel:copy>
6 </bpel:assign>
```

Listing 4.1: Data Dependency in WS-BPEL Variable Assignment

For WS-BPEL, all `assign` activities are analyzed to filter those assignments that copy from an invocation input to an invocation output variable. To detect indirect assignments via auxiliary variables, the `assign` activities are analyzed recursively. A sample `assign` activity is printed in Listing 4.1, `$a3_out` and `$a4_in` denote the variable names pointing to the input and output messages, followed by a dot (**".**") and the target WSDL *message part*. We can ignore all assignments which do not create an inter-invocation data dependency (e.g., those that assign constant values, increase counter variables etc). As we know the pattern for variable names, `$a3_out` and `$a4_in` can be extracted using regular expressions and we have determined a data dependency. Note that this method only works reliably if every invocation in WS-BPEL has its own pair of input/output variables. This is usually the case and otherwise a warning message is issued.

The second test preparation step is determining the concrete candidate services. The service registry contains references to available Web services and their WSDL interface descriptions. The WSDLs are retrieved and parsed and we loop over all `invoke` activities in the WS-BPEL

process: a service becomes a candidate if it implements the `portType` required by an invocation. In WS-Aggregation, the definition of abstract and concrete services is directly incorporated into the query model. Each query input (i.e., abstract service) targets a certain *feature* which is basically a string describing the capability of the service that has to provide this input. The service registry contains the mapping between features and concrete candidate services.

The third and last preparatory step is to combine all the gathered information and transform it into the FoCuS data model. As mentioned earlier in this section, TeCoS provides the logging data of previous executions of the composition under test. From the pool of logged invocations we filter those that carry the same process instance identifier (ID) in the SOAP header, and provide this data as FoCuS traces.

4.5.3 Transformation to FoCuS Data Model

FoCuS [151] implements algorithms for combinatorial test design and is used for optimization of our target function in Section 3.4.1. Note that FoCuS is only one possibility to achieve the optimization step in our end-to-end approach. We have also experimented with other techniques such as Constraint Programming [9] using the Choco[5] solver, but FoCuS has turned out to perform best, even for very large scenarios. In the following, we briefly describe the mapping from the service composition model to the data model used by FoCuS (see Table 4.8).

Service Composition Model	**FoCuS Model**
abstract service	attribute
concrete service	value
interrelated services	restriction
incompatible services	restriction
occurrence precedence	attribute weights
uniform service reuse	attribute weights (=1)
existing execution	trace

Table 4.8: Mapping from Service Composition Model to FoCuS Model

FoCuS uses the notion of *attributes* and *values*. Each abstract service in our model maps to an attribute in FoCuS. The values of the attributes are the concrete services that implement the required interface. To that end, each service is identified by a unique integer number.

FoCuS allows to impose custom *restrictions* on the attributes values. We use restrictions to express that two or more services should 1) never or 2) always be used together. For instance, in the presented scenario, the services a_3 and a_4 are interrelated in the sense that they should be executed by the same concrete service instance, so we add a FoCuS restriction for the attributes' equality: `a3 == a4`. For service combinations that are known to be incompatible (and should hence not occur together in a test case), we add inequality restrictions.

Constraints concerning the uniform reuse of services or precedence of certain service instances (Equation 4.4) are expressed in FoCuS using *attribute weights*. Weights express the ideal distribution of attribute values, and are a possibility to influence the value computation. Expressed in terms of the service composition model, if the candidate services $s(a) = \{s_a^1, s_a^2, .., s_a^j\}$

[5] `http://www.emn.fr/z-info/choco-solver/`

of an abstract service a have weights $w(s_a^1), w(s_a^2), .., w(s_a^j) \in \mathbb{R}$, then the (ideal) probability that service s_a^x is chosen for testing abstract service a is $\frac{w(s_a^x)}{\sum_{s_a^y \in s(a)} w(s_a^y)}$. Note that FoCuS takes weights into consideration during optimization, but gives no guarantee about value distribution.

Finally, FoCuS offers the possibility to provide data about existing previous *traces*. Traces constitute a subset of solutions that should not (or need not) be considered. The algorithm then attempts to cover all remaining value combinations. Adding traces can help reduce the complexity of the problem and the runtime of the algorithm. Section 4.5 gives more information about how the traces are collected.

4.5.4 Generating and Executing Tests

After FoCuS has finished generating a feasible solution for the given model, the automated test execution starts. The goal is to prepare the composition under test to bind to the services determined in the test cases. The solution in WS-Aggregation is straight-forward, as the query model allows to specify the concrete services to be used along with the request. That is, for each of the generated test cases a request reflecting the respective service assignment is sent to the platform. In WS-BPEL, hardcoding the concrete services in the process definition would require the re-deployment of a large number of process instances, which is time-consuming and often infeasible for large test sets. Therefore, we aim at deploying only a single process instance that is able to dynamically look up and bind to the correct EPRs at runtime. This is achieved by instrumenting additional commands into the process definition.

```
<process>
 <import location=".." ../>
 ..
 <partnerLinks>
  <partnerLink name="TMS" ../>
  ..
 </partnerLinks>
 ..
 <variables>
  <variable
     name="instanceID" ../>
  <variable name=".." ../>
  ..
 </variables>
 ..
 <sequence>
  <assign name="a1">..</assign>
  <invoke operation="getEPR"
     partnerLink="TMS" ../>
  <assign name="a2">..</assign>
  <invoke ../>
 </sequence>
 ..
</process>
```

Listing 4.2: Instrumented WS-BPEL

1: add `import` elements for WSDL and XSD imports
2: `tms` ← new `partnerLink` for Test Manager Service
3: `instanceID` ← new `variable`, initialized as GUID
4: **for all** `invoke` activities `i` **do**
5: `pl` ← `partnerLink` of invocation `i`
/ First, add statements to request EPR from Test Manager Service and to set dynamic partner link. */*
6: define `variables` $\texttt{eprIN}_\texttt{i}$ and $\texttt{eprOUT}_\texttt{i}$
7: add `assign` $\texttt{a}_1$: $\texttt{eprIN}_\texttt{i}$ ← name of `pl`
8: add `invoke` $\texttt{i}_1$: $\texttt{eprOUT}_\texttt{i}$ ← `tms.getEPR`($\texttt{eprIN}_\texttt{i}$)
9: add `assign` $\texttt{a}_2$: `pl` ← $\texttt{eprOUT}_\texttt{i}$
/ Second, add statements to set WS-BPEL instance ID in SOAP headers for invocation `i`. */*
10: `hdr` ← new header element for invocation `i`
11: add `assign` $\texttt{a}_3$: `hdr` ← `instanceID`
12: `s` ← new `sequence`: $\texttt{a}_1 \,||\, \texttt{i}_1 \,||\, \texttt{a}_2 \,||\, \texttt{a}_3 \,||\, \texttt{i}$
13: replace invocation `i` with sequence `s`
14: **end for**

Algorithm 3: WS-BPEL Instrumentation Algorithm

The corresponding algorithm is sketched in Algorithm 14. To illustrate the structure of an instrumented WS-BPEL process, the rough skeleton of an example process is printed in Listing 4.2 (dots ".." in the listing indicate that parts have been left out for clarity). After adding the required global definitions and generating a Globally Unique Identifier (GUID) in lines 1-3, the algorithm loops over all `invoke` activities and ensures 1) that the correct EPR for this invocation is retrieved from the Test Manager Service (lines 6-9), and 2) that the process ID is sent along with the invocation (lines 10-11).

The instrumented WS-BPEL process is then deployed to the target WS-BPEL engine. Via the `getEPR(String partnerLinkName)` service method the Test Manager (TM) provides the EPR information according to the current test case. The mapping between abstract and concrete services is stored in a service registry, implemented using the *VRESCo* SOA runtime environment [230]. The TM uses the specified test inputs and repetitively executes the WS-BPEL process. The result of each composition execution is matched against the expected output, which is specified by the composition developer (service integrator) in the form of XPath expressions.

4.5.5 Test Oracle

Test oracle [277] denotes the mechanism to determine whether a result is acceptable and whether a test case has failed or passed. In the case of synchronous (request-response) compositions, the test oracle is defined as a set of assertions over the result data returned by the composition. In the current implementation of TeCoS these assertions are defined as XPath expressions that are required to evaluate to a positive Boolean result (*true*). The oracle evaluates the XPath expressions and simply compares the result to the expected output defined by the tester.

The test oracle becomes more sophisticated for event-based continuous queries. The correct functioning of compositions in WS-Aggregation depends not only on a single result document, but on the timing and sequence of multiple arriving results. Hence, the assertions are defined and evaluated over a multitude of data records. In fact, the oracle definition can itself be seen as a query over the sequence of received result documents, and this sequence has to fulfill certain criteria. For instance, in our scenario we expect that, over time, hotel room availability and flight data will be available for each hotel and location provided to WS-Aggregation. Moreover, we can assume that the final results arrive with roughly the same frequency as the highest frequency of any of the event inputs. If no result is received for a long time period, the tester would define this as an indicator that the composition test case is failed.

4.5.6 Fault Localization Platform

Our prototype implementation of the presented fault localization approach is implemented in Java. We utilize the open-source machine learning framework *Weka*[6]. Weka contains an implementation of the popular *C4.5* decision tree deduction algorithm [271], denoted *J48 classifier* in Weka. C4.5 has been applied successfully in many application areas and is known for its excellent performance characteristics.

[6] `http://www.cs.waikato.ac.nz/ml/weka/`

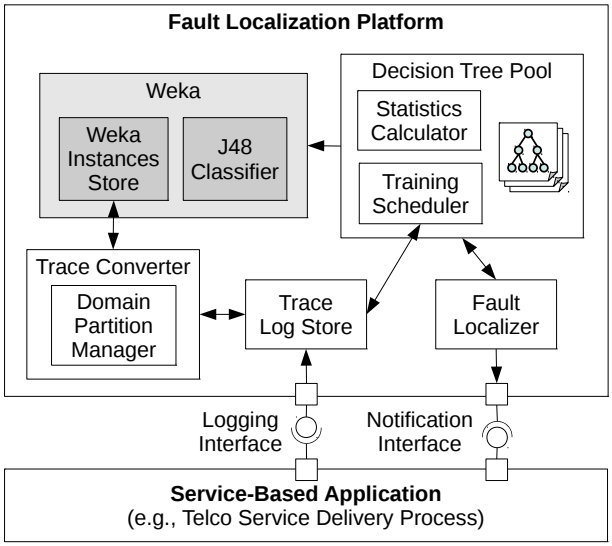


Figure 4.11: Architecture of Fault Localization Platform

Figure 4.11 outlines the architecture of the Fault Localization Platform with the core components. Third-party components (Weka) are depicted with light grey background color. The SBA submits its log traces (service bindings plus input messages) to the Logging Interface and provides a Notification Interface to receive fault localization updates. The Trace Log Store receives trace data and forwards them to the Trace Converter. The Domain Partition Manager maintains the customizable value partitions for input messages. For instance, if a trace contains an integer input parameter $x = -173$ and the chosen domain partition for x is $\{negative, zero, positive\}$ then the Trace Converter transforms the input to $x = negative$. Transformed traces are put to the Weka Instances Store. The Decision Tree Pool utilizes Weka's J48 Classifier to maintain the set of trees. The Statistics Calculator determines quality measures for the trained classifiers, and the Training Scheduler triggers adaptation of the tree pool to changing environments.

4.6 Evaluation

To evaluate different performance aspects of the proposed approach, we have set up multiple experimental evaluations in a private Cloud Computing environment, managed by an installation of *Eucalyptus*[7].

Our experiments focus on five aspects: first, the effect of the k-node coverage criterion on the size of the test case set (Section 4.6.1); second, the performance of testing WS-BPEL instances of different sizes (Section 4.6.2); third, the performance of testing long-running and event-based compositions in WS-Aggregation (Section 4.6.3); fourth, the effectiveness of identifying incompatible service assignments and the evolution of the respective indicators over time

[7] `http://www.eucalyptus.com/`

(Section 4.6.4); fifth, the performance of fault localization under noisy trace data, including transient faults and changing fault conditions (Section 4.6.5).

The experiments in Sections 4.6.1 and 4.6.2 have been run on a single machine with Quad Core 2.8GHz CPU, 3GB memory, running Ubuntu Linux 9.10 (kernel version 2.6.31-22). For the distributed performance tests in Sections 4.6.2.1 and 4.6.3, multiple cloud VM instances with slightly weaker performance characteristics have been used. Each VM is equipped with 2GB of main memory and one virtual CPU core with 2.33 GHz (comparable to the *small* instance type in Amazon EC2[8]).

4.6.1 Effect of k-Node Coverage Criterion on the Number of Test Cases

First, we investigate the effect of the k-node data flow coverage criterion on the number of generated test cases. Consider three imaginative composition scenarios (S=small/M=medium/L=large), see Table 4.9. S/M/L have, respectively, 6/10/20 abstract services, with 5/10/20 concrete services per abstract service, and different data flows of length 2/3/4.

	Small		**Medium**		**Large**	
Abstract Services	6		10		20	
Concr. S./Abstr. S.	5		10		20	
2-Node Data Flows	2		5		10	
3-Node Data Flows	1		2		5	
4-Node Data Flows	-		1		2	
Min. $\|T\|$ for $k = 4$	125		10000		160000	
Min. $\|T\|$ for $k = 3$	125		1000		8000	
Min. $\|T\|$ for $k = 2$	25		100		400	
Min. $\|T\|$ for $k = 1$	5		10		20	
FoCuS Results	min	max	min	max	min	max
Execution Time (seconds)	0.15	0.48	2.91	3.84	742.9	939.3
Test Cases ($\|T\|$)	125	125	10K	10K	160003	160008
Occurrence Differences	35	48	557	623	5148	5935

Table 4.9: Optimization of Different Model Sizes

The lower bound of generated test cases, $min(|T|)$, varies with the value of k. For the special case of $k = 1$, $min(|T|)$ equals the maximum number of concrete services per one abstract service, $max(|s(a_x)|), a_x \in A$. In general, $min(|T|) = max(|s(a_x)|)^k$, provided that the composition contains any data flow of length k. Note that these bounds are only valid if 1) the composition is free of constraints (e.g., that some abstract services must or must not bind to a specific concrete service), and 2) no previous execution traces exist. We applied the FoCuS optimization to the three scenario sizes (see Table 4.9). The FoCuS optimization was repeated 20 times and the table lists the minimum and maximum execution times (seconds), the number of generated test cases and the total service occurrence differences ($\sum_{a \in A}(max(a) - min(a))$). $|T|$ of the FoCuS result is optimal for S and M, and close to $min(|T|)$ for L.

[8] `http://aws.amazon.com/ec2`

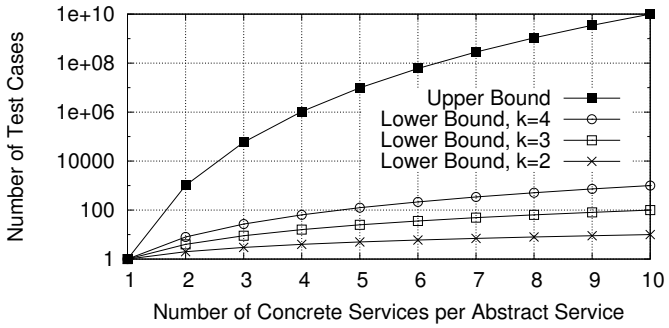


Figure 4.12: Concrete Service Combinations in Medium Scenario (cf. Table 4.9)

Figure 4.12 illustrates the upper and lower bound of $|T|$ in the Medium scenario with increasing number of concrete services. The upper bound represents the number of all possible combinations, which reaches the (infeasible) value of 10^{10} (note the logarithmic scale of the y-axis). Applying the k-node data flow coverage goal ($k \in \{2, 3, 4\}$) drastically decreases the lower bound of the number of test cases. For instance, 10 individual tests need to be executed for a coverage of k=1. Covering k=2 requires 100 test cases, and for k=3 we need data from at least 1000 tests. Arguably, the reduction in the lower bound on the number of test cases delivered by the k-node approach is not surprising, because that approach limits data flow path length. Although this argument is valid, it does not invalidate our approach and its importance, as the k-node approach is very effective in practice, as we demonstrate across this section.

4.6.2 Performance of Testing WS-BPEL Service Compositions

To evaluate the end-to-end performance of our testing approach, we have implemented and tested the scenario presented in Section 4.2. The scenario WS-BPEL process is deployed in a *Glassfish*[9] server. To simulate the services, we make use of *Genesis* [170], a test bed generator for Web services. We generated (pseudo-)random data for the composition input (*date* and *city* parameters in the scenario) as well as for the output of the individual services. For each input and output parameter, values are chosen randomly from a set of predefined possible values. However, in our setup the actual values are only relevant in those cases where data incompatibilities are injected, e.g., different formats for the geographic coordinates, as discussed in Section 4.2.1.

Table 4.10 lists the execution time of each step in the end-to-end testing lifecycle (all values are in milliseconds). Firstly, creating the data flow view from the composition definition (WS-BPEL file and WS-Aggregation query) took 151ms. This duration depends mainly on the number of `Assign` tasks contained in the WS-BPEL process. Finding the candidate services took 1205ms. This value depends on the number of services in the registry. We had 30 registered services, and 13 were identified as candidates (see scenario). Converting the composition model to FoCuS took 13ms, and the FoCuS algorithm terminated after 644ms. Initialization of WS-Aggregation finished after 353ms. The largest part of the preparation is WS-BPEL instrumentation and deployment (around 10 seconds).

[9] `https://glassfish.dev.java.net/`

Test Preparation		#	Dynamic EPR	Process Logic	Total
Data Flow View	151	c_4	240.0	408.0	648.0
Find Candidates	1205	c_{12}	280.0	390.0	670.0
Convert Model	13	c_{13}	410.0	331.0	741.0
FoCuS Solver	644	c_{17}	250.0	487.0	737.0
Initialize WS-Aggregation	353	c_{21}	320.0	303.0	623.0
Deploy WS-BPEL	10275	c_{20}	230.0	386.0	616.0
Sum	12641	Avg	288.3	384.2	672.5

Table 4.10: Performance of Test Scenario (Durations in Milliseconds)

The other values on the right side of the table are time measurements of the six test cases generated. To compute the WS-BPEL overhead caused by the instrumented code, we slightly extended the scenario process to have it measure the timestamps before and after the execution of each service invocation. These timestamps are sent to the Test Manager Service at the end of the execution to calculate test case duration end-to-end. The total time is comprised of two elements: setting dynamic EPR according to the test case, which took 288.3ms on average; executing the actual business logic, which depends on the domain, and in our case was on average 384.2ms. The average total time per test case was 672.5ms. Note that the "Process Logic" execution time is scenario-specific, whereas the "Dynamic EPR" time is near-constant and does not depend on the process logic.

4.6.2.1 Large-Scale Execution of WS-BPEL Tests

Whereas the six test cases of our scenario (see $c_4, c_{12}, c_{13}, c_{17}, c_{20}, c_{21}$ in Section 4.6.2) can easily be executed sequentially, larger test suites require a parallel execution strategy. In the following we evaluate parallel execution of WS-BPEL composition tests in TeCoS. Again, we use our scenario composition definition, but this time we leave out the WS-Aggregation part (a_4 and a_6 are implemented in WS-BPEL and the composition does not wait for 20 seconds), and we provide 10 concrete services per abstract service (to increase the size of the testing problem). Moreover, we drop the requirement that a_3 and a_4 must always bind to the same concrete service. FoCuS generated a minimum of 1000 test instances for this problem (this value corresponds to all combinations of the 3-node data flow in the scenario). We executed these tests in different distributed settings with varying number of w parallel test worker clients ($w \in \{10, 30, 50, 100, 150, 200\}$). A worker is a client program that invokes the service composition under test. The clients are distributed among multiple compute nodes in our Cloud Computing environment; each node hosts 10 clients. Table 4.11 breaks down the test processing times for deployment and actual execution of each test case. The total duration of a test case consists of the actual WS-BPEL process logic plus the overhead for dynamically exchanging EPRs with the TeCoS test manager service. For space reasons, we have only included the values for $w \in \{10, 30, 50, 100\}$, but the remaining results for $w \in \{150, 200\}$ are plotted in Figure 4.13.

As expected, the average deployment time of the process is almost the same regardless of the number of active clients. The fluctuation in the duration of the process logic is application dependent, but in our case the services used by the process are implemented to facilitate a high

	10 Clients			30 Clients			50 Clients			100 Clients		
	min.	avg.	max.	min.	avg.	max.	min.	avg.	max.	min.	avg.	max.
Deploy WS-BPEL	10103	10103.0	10103	10112	11009.7	12723	9961	10500.0	12313	9362	10196.4	12326
Process Logic	367	795.8	4699	316	1292.0	4120	396	1429.3	2736	390	1830.1	3089
Dynamic EPRs	50	278.4	490	40	447.8	1490	40	469.1	1290	50	580.8	1270
Test Case Total	607	1074.2	4749	356	1739.8	4600	446	1898.4	3246	460	2411.0	3789
Test Suite Total	157057			110760			78334			58375		

Table 4.11: Performance of Distributed Test Execution (Durations in Milliseconds)

level of concurrency. That is, a single execution of the process takes roughly the same amount of time in all settings (around 800-1800 milliseconds on average). The slight fluctuations of processing time in the business logic mostly depend on the caching/optimization strategy of the WS-BPEL engine (the first request usually takes longer than the requests to follow). The assignment of dynamic EPRs is fastest in the case of 10 clients, because of the synchronization overhead required when multiple clients access the TeCoS test manager service for EPR information. However, this overhead does not seem to grow strongly (the difference between 30, 50 and 100 clients is negligible).

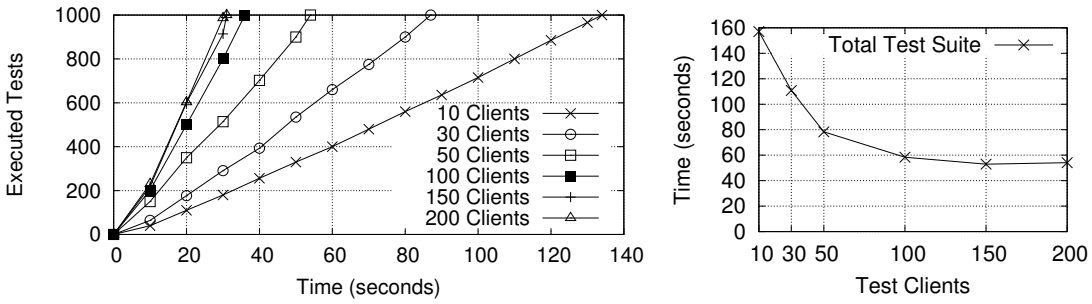


Figure 4.13: Performance of Distributed Test Execution

A core observation in Table 4.11 and Figure 4.13 is the degree by which the execution becomes faster as additional test clients are added. The left plot in Figure 4.13 depicts the number of executed tests against the time required for execution, for different numbers of parallel test clients. We can observe that deploying additional clients is especially beneficial when only few clients are active. For instance, executing the 1000 tests took 133 seconds for 10 clients, but only 87 seconds for 30 clients and 54 seconds for 50 clients. However, the marginal time benefit of adding another client decreases as the total number of clients increases, up to the point where additional clients have no effect at all, which is the case in our experiment when moving from 150 to 200 clients. The reason for this is that the composed services allow only a certain level of parallel requests. Besides faster execution of the total test suite, parallelized test execution has the benefit that the composition is, to a certain degree, stress tested (or load tested), which is a key issue for service compositions and applications that serve multiple concurrent users. However, this is not the core focus of our approach, and systematic approaches for stress testing have been discussed elsewhere (see, e.g., [89]).

4.6.3 Performance of Testing Event-Based Applications with WS-Aggregation

In the following, we discuss characteristics of testing composite services with continuous and event-based processing using the WS-Aggregation framework. The continuous event-based composition results in WS-Aggregation illustrate nicely the way in which the test indicators evolve over time. To that effect, we implemented the scenario composition discussed in Section 4.2, but with two minor modifications: firstly, to increase the size of the problem search space (from 24 to 4096 possible composition instantiations), 4 candidate services are provided for each of the 6 abstract services, and the requirement that a_3 and a_4 need to bind to the same concrete service is dropped; secondly, to illustrate more long-running compositions, the WS-Aggregation queries are *not* terminated after 20 seconds, but keep running until the complete test suite is finished. We then deliberately inject faults and incompatible service bindings, execute the test cases, and measure various test indicators, which are discussed in the following. The following results are obtained from a composition with two fault combinations along data flow paths of length 2 ($\{a_5 \mapsto s_2, a_6 \mapsto s_7\}$ and $\{a_1 \mapsto s_1, a_2 \mapsto s_3\}$) and one fault combination along a data flow path of length 3 ($\{a_1 \mapsto s_1, a_3 \mapsto s_1, a_4 \mapsto s_1\}$).

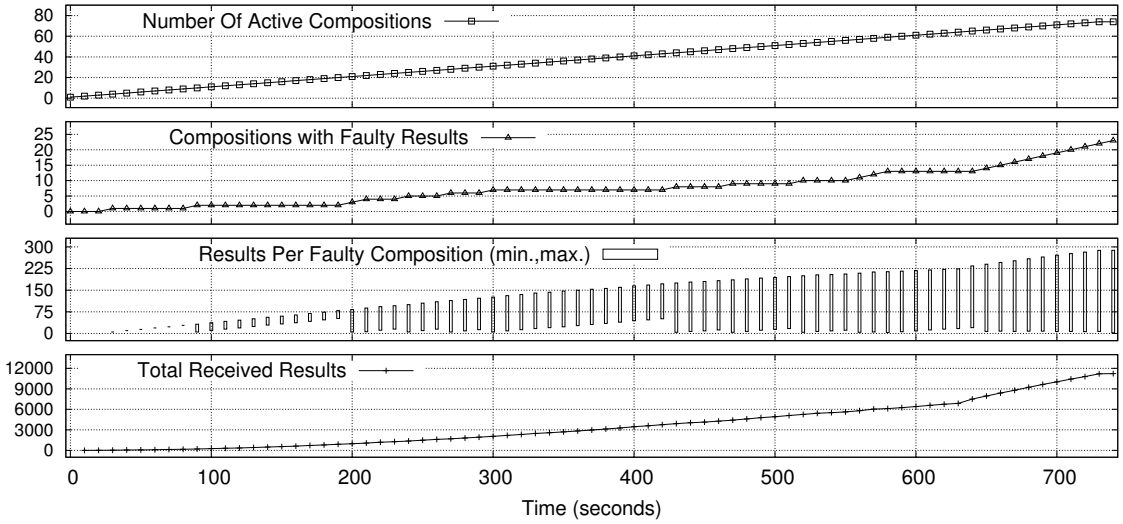


Figure 4.14: Test Results for the Event-Based Scenario Service Composition

The minimal test case set computed by FoCuS contains 64 combinations, and our experiment successively starts one test case after the other, leaving 10 seconds time between each two test cases. On the one hand, this interval gives WS-Aggregation time to initialize and distribute the execution to the available computing nodes, and on the other hand this repeated procedure allows us to take a snapshot after each added test case and to compare the values. Figure 4.14 shows how the number of compositions and results evolves over time. The values presented in the figure are taken from one typical and representative end-to-end test run. We do not use averaged values over multiple distinct and isolated test runs because the combinations computed by FoCuS are nondeterministic and the results discussed here depend on the order in which the specific test cases are added to the system.

Figure 4.14 contains four plots that illustrate the characteristics of a typical test execution of an event-based continuous service composition. The uppermost plot shows how the number of active compositions increases as one instance is added every 10 seconds. Note that the test suite executes more than the minimal set of 64 compositions, because we have added 10 additional instances (i.e., in total 74 test cases are executed) to evaluate the effectiveness of test result analysis later in Section 4.6.4. The lowermost plot in Figure 4.14 draws a trendline of the total number of results that have been received as event notification messages from the composition. This curve climbs slowly in the beginning, but the slope gets steeper as more compositions are active. At time point 740 roughly 11000 result events have been received in total. The second plot from the top of the figure contains the number of compositions from which the TeCoS framework has received faulty results, i.e., results with unexpected output data. The third plot from the top depicts the range of results per faulty composition. This value is zero at the beginning (until second 20) and starts to rise as the first faulty composition is activated at time point 30. Up to time point 130, the minimum and maximum are equal (because there is only one faulty composition), and from second 90 onwards the minimum and maximum span up an actual interval range (printed as a box in the figure).

Computing and analyzing the number of received composition results (both correct and faulty) is important to obtain a measure for performance-related QoS data. Not shown in the figure are the QoS metrics that are contained in the TeCoS test report, some of which are: *event throughput* (average number of received events per time unit), *regularity* (fluctuations in the interval of received events) or *duplication* (whether or not duplicate events haven been received). These metrics allow a composition tester not only to identify incompatible service combinations, but to favor one composition configuration over another for QoS reasons. A detailed discussion of QoS aspects is out of scope of this thesis, and more details can be found, e.g., in [187, 373].

4.6.4 Measures for Determining Incompatible Service Assignments

After having discussed the composition test generation and execution in the previous sections, we now evaluate the mechanism for actually detecting faulty and incompatible service assignments in TeCoS. Again, we take the test setup of the event-based composition scenario of Section 4.6.3 (composition with 3 faulty service combinations along data flow paths of length 2 and 3, respectively). In the following, T denotes the set of composition test cases (same notation as in Section 4.3.5), which contains 74 test cases in our experiment (64 cases in the minimal set computed by FoCuS, plus 10 additional tests).

Recall from Section 4.3.5 that a service assignment x is defined as $x : A \rightarrow S$, and let $X \in \mathcal{P}([A \rightarrow S])$ be the set of all service assignments for which the fault contribution evaluates to 1. More formally, $\forall x \in X : cont(x, T) = 1$. The general idea is that, given the knowledge of succeeded and failed test compositions at any point in time, the service assignments in X which have always led to a fault are deemed incompatible. However, we will see that, as new test results become available, a service assignment which is assumed to be incompatible at some point in time does not necessarily stay in this state.

Figure 4.15 plots the number of incompatible service assignments over time. The three curves in the figure represent the number of incompatible service bindings along data flow paths of length 1, 2 and 3 (denoted curve I, curve II and curve III). The special case of a 1-node data

flow (curve I) actually represents a single-service binding. Up to time point 20, there are no faulty compositions, hence no incompatible service assignments exist. At second 30, the first faulty composition is recorded, and curve I takes value 2, curve II jumps to a value of 3, and curve II reaches value 1. From second 30 onwards, all three curves show a rising trend until they reach a peak value (2 for curve I, 7 for curve II, and 13 for curve III) and from then on start to decrease. Curve I drops to 0 at time 150, which means that after the 15th test case we detect that no single service is solely responsible for the faulty behavior (i.e., the fault has to involve a combination of 2 or more services).

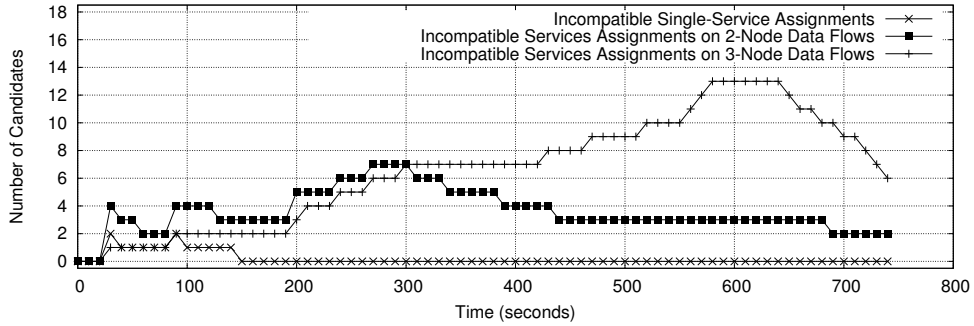


Figure 4.15: Incompatible Service Assignments Along Data Flows of Different Lengths. The minimal set (64 tests in this example) ensures that all desired combinations are covered; the 10 additional tests (seconds 650-740) further narrow down the fault localization result.

We observe that the curves in Figure 4.15 rise and drop over time. The reason for a rise is that a new test case has failed which contains one or more bindings which have not been tested before. Conversely, a drop happens if some binding that previously failed in all test cases is now contained in a successful test case. Overall, it is desirable to narrow down the faulty service combinations as far as possible, i.e., we aim to see the curves at a low level. After executing the minimal set of 64 test cases, at time point 640, curve II has a value of 3, and curve III stands at value 13. We see that the following additional 10 test cases contribute further to the fault localization. This aspect is discussed in Section 4.6.4.1. Another aspect to consider is the number of incompatible service combinations. In this section we so far considered a composition scenario with 3 injected data flow faults; Section 4.6.4.2 discusses evaluation results with different faulty data flow combinations.

4.6.4.1 Additional Tests Beyond the k-Coverage Minimal Test Set

In our approach we distinguish between fault detection and fault localization. Generally, the test framework first executes all test cases computed by FoCuS, because they are the minimum requirement to meet the data flow based coverage goal, which ensures that faults are *detected*. To *localize* the origins of faults, the minimal test set may not provide sufficiently precise results (depending on the configuration). The reason for this is as follows. Since the k-node coverage goal attempts to minimize the number of test cases, the longest data flows within a composition are usually represented by only a single test case per possible service combination. That is,

within the minimal test set (64 test cases in our experiment) each service combination along the maximum length data flow (3-node data flow in our experiment) is necessarily considered faulty if it is being used in a faulty composition. We can observe this by comparing Figure 4.14 and Figure 4.15: up to time point 640, the number of compositions with faulty results is equal to the number of incompatible service assignments on 3-node data flows in Figure 4.15.

Hence, the tester can decide to include additional tests in the test set T (for illustration, 10 additional tests were run in our experiment). As outlined in Section 4.3.5, the selection of additional tests is crucial. At this point we generally strive for a reduction of the size of X, i.e., identifying service combinations that *so far* only participated in failed test, but do not necessarily *always* lead to a fault. Therefore, we heuristically construct a new test composition $c \in C$ by picking a (so far deemed to be faulty) service assignment $x \in X$ and filling in concrete services (from previously observed successful compositions) for the remaining abstract services not included in x. By continuously executing such additional test cases, we expect to find a successful composition instance that contains x, in which case x is removed from X. For our example, this effect is shown between time points 650 and 740 where ten additional test cases are executed that bring down curve II and curve III to a value of 2 and 6, respectively. While curve-II has now reached its minimum (i.e., the actual expected value), curve-III may be further reduced by executing additional test cases.

4.6.4.2 Coping with Multiple Faulty Service Combinations

To illustrate how our approach copes with multiple faulty service combinations, we have run five versions of the experiment mentioned in this section, with increasing number of injected fault combinations (1,2,3,4,5) along (non-overlapping) data flow paths of length 2 and 3.

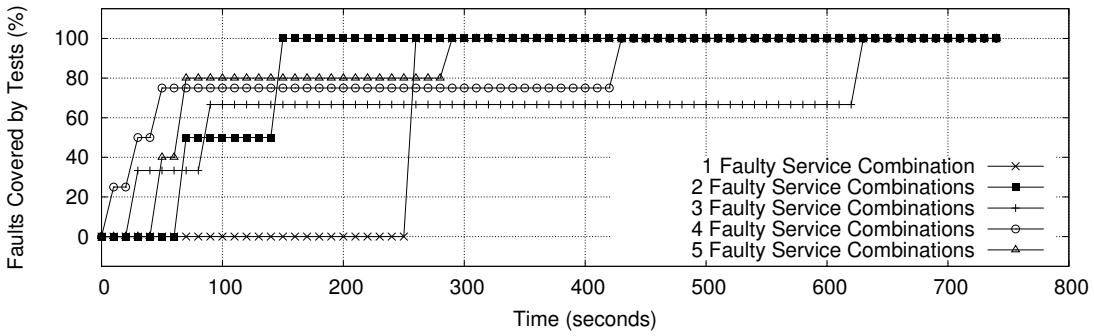


Figure 4.16: Fault Combination Test Coverage Over Time

Figure 4.16 illustrates the percentage of faults (i.e., faulty concrete service combinations) that are covered by the tests over time. The percentage increases as new test cases are executed. For instance, the single fault combination is covered after time point 250 and the percentage jumps from 0% to 100%; for 2 faulty service combinations, the first combination is tested at time point 70, and the second fault is covered by the test case at time point 150. The core observation in Figure 4.16 is that eventually all curves reach 100% on or before time point 640,

where the last test case of the minimal test set is executed. This is deterministically reproducible because k-node coverage ensures that all relevant data flow paths are covered by the tests.

4.6.5 Performance of Fault Localization Approach

In the following we evaluate different aspects of our proposed fault localization approach. We have set up a comprehensive evaluation framework as part of *Indenica*[10], a research project aiming at developing a virtual platform for service computing. The framework provides traces of large SBAs, against which we run our fault detection algorithms.

4.6.5.1 Evaluation Setup

The test traces are generated randomly, with assumed uniform distribution of the underlying random generator. Table 4.12 shows six different SBA instances with corresponding parameter settings that are considered for evaluation. $|A|$ denotes the number of abstract services, $|s(a)|$ is the number of concrete services of each abstract service $a \in A$, $|p(a)|$ represents the number of input parameters per abstract service, $|d(p)|$ is the domain size for a parameter $p \in P$, and $|E_I|$ is the number of incompatibilities (cf. Section 4.4.3) which are injected to cause runtime faults. The table also lists for each setting the probability that a fault occurs in a random execution.

ID	$\|\mathbf{A}\|$	$\|\mathbf{s(a)}\|, \mathbf{a} \in \mathbf{A}$	$\|\mathbf{p(a)}\|, \mathbf{p(a)} \in \mathbf{P}$	$\|\mathbf{d(n)}\|, \mathbf{n} \in \mathbf{N}$	$\|\mathbf{e}\|, \mathbf{e} \in \mathbf{E_I}$	**Fault Probability**
$S1$	5	5	10	20	$\{1\}$	$4 * 10^{-2}$
$S2$	5	5	10	20	$\{2\}$	$2 * 10^{-3}$
$S3$	5	5	10	20	$\{3\}$	$1 * 10^{-4}$
$S4$	5	5	10	20	$\{3, 3, 3\}$	$3 * 10^{-4}$
$S5$	10	10	10	100	$\{3, 4\}$	$1.001 * 10^{-6}$
$S6$	10	10	10	100	$\{4\}$	$1 * 10^{-12}$

Table 4.12: Fault Probabilities for Exemplary SBA Model Sizes

The tests have been performed on machines with two Intel Xeon E5620 quad-core CPUs, 32 GB RAM, running Ubuntu Linux 11.10 with kernel version 3.0.0-16.

4.6.5.2 Training Duration

First, we evaluate how many fault traces are required by the J48 classifier to pass the threshold for reliable fault detection. The scenario SBAs $S3, S2, S1$ (cf. Table 4.12) were used in Figure 4.17, 20 iterations of the test were executed, and the figure contains three boxes representing the range of minimum and maximum values. As shown in Figure 4.17, the number of traces required to successfully detect a faulty configuration depends mostly on the complexity (i.e., probability) of the fault with regard to the total scenario size.

A single fault in configuration $S1$ was on average detected after observing between 90 and 190 traces. Multiplying these values with the fault probability of $4 * 10^{-2}$, we get a range of 4 to

[10] `http://www.indenica.eu/`

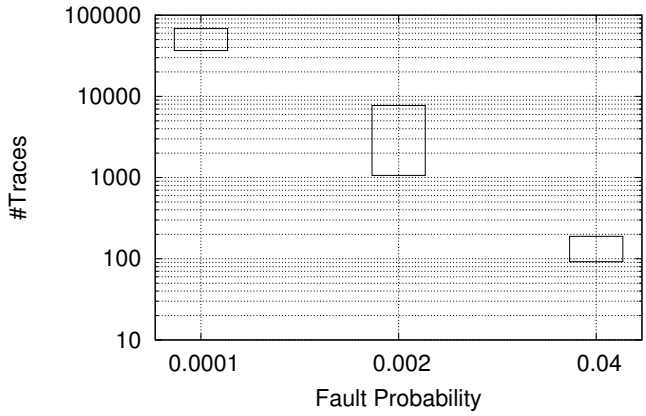


Figure 4.17: Number of Traces Required to Detect Faults of Different Probabilities

8 fault traces required for localization. Also with more complex (unlikely) faults the relative figures do not appear to change considerably. With fault probabilities of $2*10^{-3}$ and $1*10^{-4}$ faults are detected after observing 3/16 and 4/7 minimum/maximum fault traces, respectively. The data suggest a strong relationship between number of required fault traces and fault probability.

4.6.5.3 Coping with Transient Faults

As discussed in Section 4.4.4.2, our fault localization approach is designed to cope with changing environments, which is evaluated here.

Figure 4.18 shows the performance in the presence of changing faults. The evaluation setup is as follows: Initially a fault combination $FC1$ (e.g., $\langle t_x(premium) = false, t_x(a_3) = s_8 \rangle$) is active. At trace 33000, the implementation that causes the fault $FC1$ is repaired, but the fix introduces a new fault $FC2$ that is fixed at trace 66000. At trace 66000, another fault $FC3$ occurs, and an attempted fix at trace 88000 introduces fault $FC4$, while $FC3$ remains active. At trace 121000, both $FC3$ and $FC4$ are fixed, but two new faults $FC5$ and $FC6$ are introduced to the system. The occurrence probability for all six fault combinations is set to $2 * 10^{-3}$ (corresponding to scenario $S2$ in Table 4.12).

This scenario is designed to mimic a realistic situation, but serves mainly to highlight several aspects of our solution. After about 4000 observed execution traces the localizer provides a first guess as to the cause of the fault, but the classification is not yet correct. After around 5200 observed execution traces, the localizer was able to analyze enough error traces to provide an accurate localization result. Note that at that time, only about 6 error traces have been observed, yet the algorithm already produces a correct result. At trace 33000, the previously detected fault $FC1$ disappears and is replaced by $FC2$. Due to the pool of decision trees maintained by our localizer, $FC2$ can again be accurately localized roughly 6000 traces later. Similarly, after $FC2$ disappears, $FC3$ is localized roughly 5000 traces after its introduction.

The decision tree pool allows for the effective localization of new faults introduced to the system at any time. At trace 88000 in Figure 4.18, $FC4$ is introduced, and can again be accurately localized after observing around 5000 traces. $FC3$ and $FC4$ disappear at trace 121000

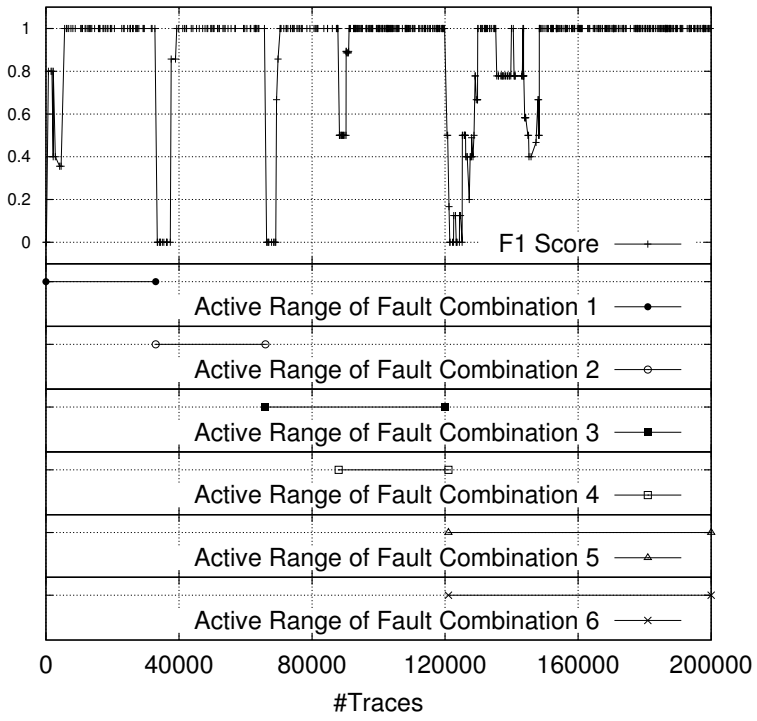


Figure 4.18: Fault Localization Accuracy for Dynamic Environment with Transient Faults

and are replaced by simultaneously occurring errors $FC5$ and $FC6$. This situation is more challenging for our approach, as seen in the rightmost 80000 traces in Figure 4.18. The spikes between trace 121000 and 150000 represent different localization attempts that are later invalidated by contradicting execution traces. Finally, however, the localization stabilizes and both faults $FC5$ and $FC6$ are accurately detected.

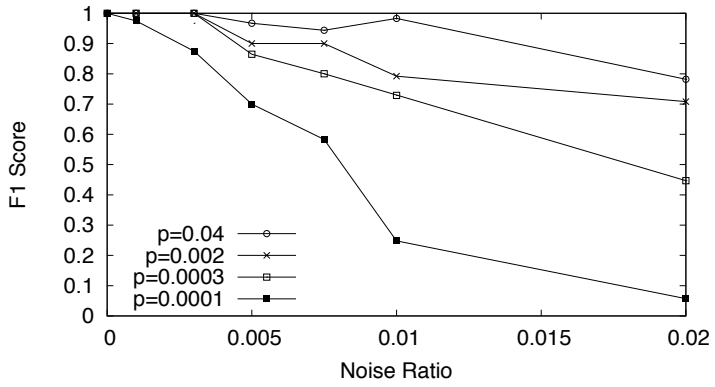


Figure 4.19: Noise Resilience – Accuracy in the Presence of Noisy Data

We also evaluated the performance of our approach using different noise levels in the trace

logs. Figure 4.19 analyzes how F1 develops with increasing noise ratio. The figure contains four lines for the scenario settings $S1 - S4$. To ensure that the algorithm actually obtained enough traces for fault localization, we executed the localization run after 200000 observed traces.

4.6.5.4 Runtime Considerations

Due to the nature of the tackled problem, as well as the usage of C4.5 decision trees to generate rules, there are some practical limitations to the number of traces and scenario sizes that can be analyzed using our approach within a reasonable time. In the following we provide insights into the runtime performance in different configurations and discuss strategies for fine-tuning.

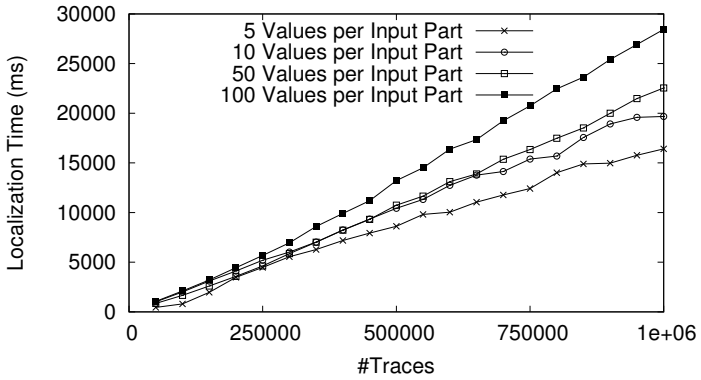


Figure 4.20: Localization Time for Different Trace Window Sizes ($S5$, $|d| = \{5, 10, 50, 100\}$)

Figure 4.20 shows the time needed for to localize faults for various trace window sizes for the base scenario $S5$, for input sizes $|d| = \{5, 10, 50, 100\}$. The figure illustrates that the time needed for a single localization run increases roughly linearly with increasing window sizes. Larger trace windows allow the algorithm to find more complex faults. If fast localization results are needed, the window size must be kept adequately small, at the cost of the system not being able to localize faults above a certain complexity.

Furthermore, the frequency of localization runs must be considered when implementing our approach in systems with very frequent incoming traces (in the area of hundreds or thousands of traces per second). Evidently, there is a natural limit to the number of traces that can be processed per time unit. Figure 4.21 shows the localization speed as number of traces processed per second compared to different fault localization intervals (i.e., number of traces after which fault localization is triggered periodically) for different window sizes ($|T|$).

The data in Figure 4.21 can be seen as a performance benchmark for the machine(s) on which the fault localization is executed. Executing this test on different machines will result in different performance footprints, which serves as a decision support for configuring window size and localization interval. For instance, if our application produces 1500 traces per second (i.e., processes 1500 requests per second), a localization interval greater than 400 should be used. Currently, the selection happens manually, but as part of our future work we investigate means to fine-tune this configuration automatically.

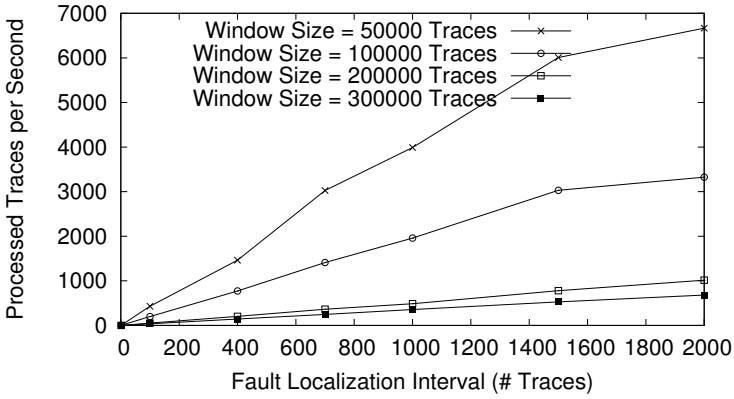


Figure 4.21: Fault Localization Performance for Different Intervals and Window Sizes ($S5$)

4.6.6 Discussion of Assumptions, Weaknesses and Limitations

In this section we summarize the main assumptions, weaknesses and current limitations of the approach discussed in this chapter. This summary also points to open research questions which may be of interest for future investigation.

- Long-running business processes with humans: The focus of the presented testing approach is on data-centric, technical Web services and processes with short waiting time between activities. Our approach is arguably not well suited for lengthy service compositions and business processes possibly involving human tasks (e.g., credit application process, product assembly process).

- Testing with real services: The presented framework aims at integration testing of compositions with real (production use) services. The approach is hence best suited for technical services that are free of charge or only associated with a small fee. Integration of expensive services from external providers is not the intended scope. We point out that related approaches which perform *online testing* of services [35, 54, 119, 127] use similar assumptions. For instance, [119] utilizes online testing for service discovery, binding, and composition. Critical security features like authentication and authorization are also best tested in the services' real execution environment [35, 134]. A possible extension to our approach would be to utilize mock testing services which simulate or proxy the actual services. However, some faults and side effects may only be reliably discovered on the basis of the real services.

- Long transitive data dependencies: The k-node data flow coverage criterion is an instrument to limit the size of the test case set. However, the effectiveness of the measure depends on the composition structure. Whereas our approach is well-suited for most real-life data flow graphs (e.g., Composition 1 on the left side of Figure 4.22), for certain corner cases k-node coverage cannot be applied as effectively (e.g., Composition 2 on the right side of Figure 4.22). As part of our future work we are further investigating the impact of structural features.

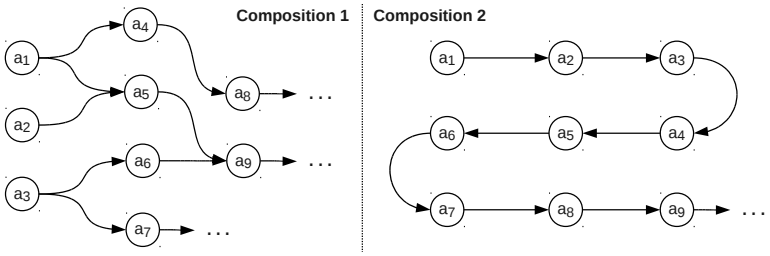


Figure 4.22: Examplary Data Flow of Structurally Different Service Compositions

* Implicit data dependencies: Moreover, our test model considers only data dependencies on the service composition level. Our approach is currently not able to detect implicit dependencies in the environment, e.g., caused by two services reading/writing from/to the same distributed file system. However, in Section 3.5 we have proposed techniques for capturing such system-level changes (e.g., write operations on the filesystem), and in future work we plan to integrate these implicit dependencies into the testing approach discussed here.

* Lack of semantic information: Currently, our approach lacks semantic information such as knowledge about the internal implementation of services. In particular, a model of the service-internal data flow would help to generate even more accurate test cases. To achieve this task we envision the use of testable services [24, 94] which expose metadata without discovering the actual service internals.

4.7 Related Work

Testing of SBAs and service compositions has been intensively studied in the previous years. In this section, we discuss some of the related work in these areas in detail.

4.7.1 Testing of Service-Based Applications

Earlier works have placed the research areas of service-oriented computing and software testing into perspective, and identified the characteristics and challenges that apply to testing single services and services in combination [44, 51]. An overview and timeline of different testing approaches for service compositions has recently been given by Hazlifah et al. [282]. Our approach is also influenced by early works that introduced data flow oriented program testing (e.g., by Laski/Korel [183] or Rapps/Weyuker [275]), as well as by authors who have proposed data flow analysis as a suitable means to generate test cases for testing Web services (e.g., Heckel and Mariani [124]). Also Liu et al. [200] have addressed the problem of data-flow based testing of Web applications, although their focus is more on analyzing Hypertext Markup Language (HTML)/XML documents and navigation relations across HTTP requests.

In [111], a method to generate test case specifications for WS-BPEL compositions is introduced. Whereas their approach attempts to cover all transitions of (explicit) links between

invocations, we focus on direct and indirect data dependencies and ensure k-node data flow coverage.

The data flow-based validation of Web services compositions presented by Bartolini et al [25] has similarities with our approach. The paper names categories of data validation problems (e.g., redundant or lost data) and their relevance for Web service compositions. Data flows in Web service compositions are modeled using Data Flow Diagrams (DFDs). In addition, the authors propose the usage of a data fault model to seed faults (e.g., some value that is out of its domain range) into the data flow model and to establish fault coverage criteria. The DFD can be used either stand-alone to measure structural coverage along data flow paths, or in combination with a WS-BPEL description for checking whether the composition conforms to the data flow requirements. This is contrary to our method: whereas DFDs are defined manually to statically validate the WS-BPEL process, we auto-generate the data flow view and create test cases for dynamic integration testing.

Mei et al. [222] propose a WS-BPEL data flow testing approach which aims at identifying defects in service compositions that are caused by faulty (or ambiguous) XPath expressions selecting a different XML element at runtime than the one that the composition developer intended to be selected. The paper introduces the XPath rewriting graph (XRG) to model XPath on a conceptual level. Based on XRG, different test coverage criteria are defined, which mandate that all possible variants of the XPaths in a WS-BPEL process shall be tested. Other than our work, the paper does not consider dynamic binding or indirect data dependencies in the form of k-node data flows.

In [82], a method for testing orchestrated services is shown. Service orchestrations, e.g., expressed in WS-BPEL, are transformed into an abstracted graph model. Testers define *trap properties* expressed as LTL formulas, which indicate impossible execution traces that should never occur. Model checking is used to generate a counter-examples tree, containing all paths that violate a certain trap property. Testers can further specify which traces are more relevant, which helps pruning the counter-examples tree. This approach is different to ours, as it aims at covering invocation traces of compositions with fixed concrete services. Additionally, it strongly involves the human tester and requires domain knowledge and more manual adjustments in preparation of the test.

Tarhini et al. [322] present an approach for testing Web service based applications, which involves test case generation for several testing steps. Firstly, candidate services are identified based on boundary value testing analysis [30]. Secondly, the candidates are individually tested, making use of a state machine based model of the service internals. Finally, at the integration level, service compositions are tested by covering all possible paths defined in the composition model. A major difference to our work is that services are selected during development time and dynamic binding is not considered.

The authors of [381] present another minimum coverage method for composite services. A description model is introduced, which defines both individual Web services and relationships among these services. Other approaches use Petri nets [83] or extended types of finite state machines [182] to model Web service compositions (or, more generally component and process interactions [87]) in order to generate tests based on these models.

Testing of dynamic service compositions is related to integration testing in component-based

systems [213, 214, 352]. The comprehensive survey in [164] discusses various issues in integration testing of component based systems, some of which are also closely related to testing service compositions. The work of Piel et al. [265] is largely centered around realizing and testing virtual components in component models. Virtual components enclose a set of real components that are to be tested in combination. Certain structures of virtual components have similarity with service combinations defined by the k-node data flow criterion, but virtual components suffer from problems such as ill-formed empty data flows, which cannot occur in our approach.

The nature of event-based systems and service compositions poses difficult challenges to testing and debugging [29, 295]. For instance, event correlation is necessary to ensure that incoming event messages are associated with the correct processing element, but this mechanism has proven to be complex and often error-prone (e.g., [23, 201]). Additionally, event-based compositions evolve over time and require a means to dynamically initiate new or terminate existing event streams, which is another potential point of failure. Additionally, QoS characteristics are a key concern, particularly when the event processing platform operates under high load. While some of the above mentioned points have previously been tackled in isolation, we utilize TeCoS to test dynamic event-based service compositions end-to-end, as a whole.

Finally, there is a large field of related work in the areas of test result analysis, fault diagnosis [63], and fault localization [358]. Tu et al. [333] discuss fault diagnosis in large graph-based systems and present efficient algorithms for finding potential failure sources from the set of graph nodes that report an alarm. Our approach to analyzing test results is related to their work, as we also consider the dependency graph to determine faulty service assignments that have caused a set of compositions to fail. Software fault localization (e.g., [149, 169, 276, 358]) is a research area whose primary target is to find faults or bugs on the source code level, and the techniques are partly applicable to dynamic service compositions as well. Other seminal work in the area of fault localization has been recently presented by Masri [216]. The technique attempts to identify faulty program statements based on information flow coverage data of historical program executions. In essence, the likelihood of a statement being faulty is determined by contrasting the percentage of failed to passed executions. We build on this approach and, in order to eliminate false positives and false negatives, additionally define the metrics fault contribution and fault participation which are based on precision and recall known from information retrieval [18].

4.7.2 Fault Detection and Fault Localization Techniques

Fault detection and localization is an active research field that continues to produce sophisticated ongoing results. For instance, approaches in the area of software fault localization (e.g., [149, 169, 276, 358]), whose primary target are faults/bugs on the software source code level, are partly applicable to service compositions as well. In particular, the metrics termed *hue* and *suspiciousness* in [169] are related to the fault participation and fault contribution metrics defined here. Another related fault localization method has recently been applied in [216], which, similar to our approach, uses coverage metrics based on information flow. The approach ranks source code statements regarding their likelihood of being faulty, and this ranking is *"primarily determined by contrasting the percentage of failing runs to the percentage of passing runs that induced it"* [216].

Software fault localization helps to identify bugs in software on the source code level. Oftentimes a two-phase procedure is applied: 1) finding suspicious code that may contain bugs and 2) examining the code and deciding whether it contains bugs with the goal of fixing them. Research mainly focused on the former, the identification of suspicious code parts with prioritization based on its likelihood of containing bugs [358]. The seminal paper by Hutchins et al. [149] introduces an evaluation environment for fault localization (often referred to as the *Siemens suite*), consisting of seven base programs (in different versions) that have been seeded with faults on the source code level. Renieres et al. [276] present a fault localization technique for identifying suspicious lines of a program's source code. Based on the existence of a faulty run of the program and many correct runs they select the correct run that is most similar to the faulty one. Proximity is defined based on the program dependence graph. Then, they compare the two runs and produce a report of suspicious program lines. This general functionality is very common in software fault localization. Guo et al. [120] propose a different similarity metric based on control flow. The metric takes into account the sequence of statement instances rather than just the according set. Our work differs from traditional software fault localization in that we do not analyze program code but assume to only be able to observe the external behavior of services. We also assume that the environment or service implementations may change during runtime, in contrast to the analysis of static code.

Monitoring and fault detection are key challenges for implementing reliable distributed systems, including SBAs. Fault detectors are a general concept in distributed systems and aim at identify faulty components. In asynchronous systems it is in fact impossible to implement a perfect fault detector [60], because faults cannot be distinguished with certainty from lost or delayed messages. Heartbeat messages can be used for probabilistic detection of faulty components; in this case a monitored component or service has the responsibility to send heartbeats to a remote entity. The fault detector presented in [287] considers the heartbeat inter-arrival times and allows for a computation of a component's faulty behavior probability based on past behavior. Lin et al. [199] describes a middleware architecture called *LIama* that advocates a service bus that users can install on existing service-based infrastructures. It collects and monitors service execution data which enable to incorporate fault detection mechanisms using the data. Such a service bus can be used to collect the data necessary for our analysis. The major body of research in the area of monitoring and fault detection in SBAs deals with topics like SLAs and service compositions rather than compatibility issues [259].

Fault analysis derives knowledge from faults that have been experienced. Adaptation tries to leverage this knowledge to reconfigure the system to overcome faults. Zhou et al. [378] have proposed GAUL, an problem analysis technique for unstructured system logs. Their approach is based on enterprise storage systems, whereas we focus on dynamic service-based applications. At its core, GAUL uses a fuzzy match algorithm based on string similarity metrics to associate problem occurrences with log output lines. The aim of GAUL differs from our approach since we assume the existence of structured log files and focus on the localization of faulty configuration parameters. Control of SOAs mostly relies on static approaches, such as predefined policies [264]. Techniques from artificial intelligence can be used to improve management policies for SBAs during runtime. Markov decision processes, for instance, represent a possible way for modeling the decision-making problems that arise in controlling SBAs. Markov deci-

sion processes and algorithms to solve them have been shown effective in reducing the impact of defects in service implementations by adapting the SBA at runtime [154]. In this work we focus on fault localization rather than on how to react in the face of faults.

4.8 Conclusions

In this chapter we have discussed the problem of testing and fault localization for data-centric and event-based applications, and presented a suitable mechanism which is implemented as part of the TeCoS framework.

The first major contribution of this chapter is about a systematic testing method. The key problem addressed here is that applications combining services and data from different providers may result in unforeseen incompatibilities at runtime, hence requiring thorough integration testing. We use an abstracted view of applications that takes into account the data flow occurring between individual service invocations. Based on an illustrative scenario, we presented our model of data-centric compositions and formalized the k-node data flow coverage criterion. The data flow view is suited to abstract from the actual composition technology, and the framework provides a plug-in mechanism to implement test adapters for concrete target platforms. Two such adapters have been implemented and evaluated, one for the de-facto standard service composition language WS-BPEL and the other for the data processing platform WS-Aggregation, which has been introduced in Chapter 3 of this thesis.

The second contribution of this chapter is a fault localization technique that is able to identify which combinations of service bindings and input data cause problems in SBAs. The analysis is based on log traces, which accumulate during runtime of the SBA. A decision tree learning algorithm is employed to construct a tree from which we extract rules, describing which configurations are likely to lead to faults. For providing a fine-grained analysis we do not only consider the service bindings but also data on message level. This allows to find incompatibilities that go beyond "service A is incompatible with service B" leading to rules of the form "service A has incompatibility issues with service B for messages of type C". Such rules can help to safely use partial functionality of services. We present extensions to our basic approach that help to cope with dynamic environments and changing fault patterns. We have conducted experiments based on a real-world industry scenario of realistic size. The results provide evidence that the employed approach leads to successful fault localization for dynamically changing conditions, and is able to cope with the large amounts of data that accumulate by considering fine-grained data on message level.

As part of our ongoing work on TeCoS we strive to unify the orthogonal fields of interface-based service testing and service composition testing. Furthermore, we envision potential future research directions by integrating the concept of testable services into our approach. It will be interesting to see whether different coverage data (e.g., line/branch coverage) exposed by testable services can be utilized to further narrow down the search for faults in dynamic data-centric compositions. Moreover, we are extending the framework to support further service composition techniques, particularly focusing on emerging fields such as data-centric service mashups [32] and interactions in mixed service-oriented systems [290]. We also plan to improve the test generation algorithm and to provide the tester with augmented control over the

characteristics of the test optimization and execution. Finally, we are currently integrating the presented testing approach with our work on fault modeling in event-based systems [136] (see Section 3.2). The core idea is to obtain a more complete picture of the processing logic of event-based service compositions (including event correlation, input-output functions, or deployment topologies), in order to perform systematic tests which aim at identifying common sources of faults.

Additionally, as future work we plan to extend our approach beyond the pure fault localization aspects. In particular, we will use the extracted rules for guiding automated reconfiguration when a fault occurs. Furthermore, we intend to integrate test coverage mechanisms that help to actively investigate faults. This can be used for systematic test execution of insightful configurations and input requests which further narrow down the search space of possible fault reasons.

CHAPTER 5

SeCoS: Automated Enforcement of Access Constraints in Business Processes

5.1 Introduction

The SOA metaphor has been elaborated by different communities to address different problem areas such as enterprise application integration or business process management (see, e.g., [88, 132, 259]). Today, Web services [364] are a commonly used technology which serves as a foundation of SOAs, as well as distributed business processes. A distributed business process is an intra-organizational or cross-organizational business process executed in a distributed computing environment (such as SOA). Mission- or safety-critical business processes often require the definition and enforcement of process-related security policies. For example, such requirements result from internal business rules of an organization, or service-level agreements (SLAs) [187] with customers. Particularly for applications which process potentially sensitive data and operate in Cloud environments with multiple tenants, enforcement of security and access control is imperative. In addition, numerous regulations and Information Technology (IT) standards exist that pose compliance requirements for the corresponding systems. In particular, IT systems must comply with laws and regulations such as the Basel II/III Accords, the International Financial Reporting Standards (IFRS), or the Sarbanes-Oxley Act (SOX). For instance, one important part of SOX compliance is to provide adequate support for definition and enforcement of process-related security policies (see, e.g., [53, 80, 232]).

Role-based access control (RBAC) [101,284] is a de-facto standard for access control in both research and industry. In the context of RBAC, roles are used to model different job positions and scopes of duty within an information system. These roles are equipped with the permissions to perform their respective tasks. Human users and other active entities (subjects) are assigned to roles according to their work profile [313, 314]. A process-related RBAC model [316, 345]

enables the definition of permissions and entailment constraints for the tasks that are included in business processes. A *task-based entailment constraint* places some restriction on the subjects who can perform a task x given that a certain subject has performed another task y. Entailment constraints are an important means to assist the specification and enforcement of compliant business processes [33, 40, 69, 315, 320, 357].

Mutual exclusion and binding constraints are typical examples of entailment constraints. Mutual exclusion constraints can be subdivided in *Static Mutual Exclusion (SME)* and *Dynamic Mutual Exclusion (DME)* constraints. An SME constraint defines that two tasks (e.g. "Order Supplies" and "Approve Payment") must never be assigned to the same role and must never be performed by the same subject (to prevent fraud and abuse). This constraint is global with respect to *all process instances* in an information system. In contrast, DME refers to individual process instances and can be enforced by defining that two tasks must never be performed by the same subject in the *same process instance*.

In contrast to mutual exclusion constraints, binding constraints define that two bound tasks must be performed by the *same* entity. In particular, a *Subject-Binding* constraint defines that the same individual who performed the first task must also perform the bound task(s). Similarly, a *Role-Binding* constraint defines that bound tasks must be performed by members of the same role but not necessarily by the same individual.

5.1.1 Motivation

As outlined above, RBAC and entailment constraints are an important means to assist the specification of business processes and control their execution. Yet, the runtime enforcement of such constraints in distributed SOA business processes is complex, and currently there is still a lack of straightforward solutions to achieve this task. The complexity arises from the fact that the tasks of distributed business processes are performed on independent, loosely coupled nodes in a network. The advantage of loose coupling is that the different nodes (i.e., services) can execute their tasks independently of other nodes. However, the enforcement of access constraints in a distributed system often requires knowledge that is not available to a single node.

Moreover, to enforce access control policies in a software system, the resulting policy models must also be mapped to the implementation level. To account for different platforms and implementation styles, it is important to first establish the enforcement on a generic and conceptual level, in order to map it to concrete platforms (e.g., SOA, as in our case).

Evidently, enforcement of RBAC policies and constraints has an impact on the execution time of business processes. Depending on the complexity of the constraints and the amount of data that needs to be evaluated, the impact will be more or less severe. While the theory behind RBAC and entailment constraints in business processes has been intensively studied in the past, less attention has been devoted to the runtime enforcement, including performance impacts, of such constraints.

With respect to the rising importance of process-aware information systems, paired with an ever-increasing trend to move applications into Cloud environments, the correct and efficient implementation of consistency checks in these systems is an important issue. Therefore, the consistency and runtime performance needs to be evaluated thoroughly in order to ensure the efficient execution of business processes that are subject to access constraints.

5.1.2 Approach Synopsis

In general, distributed business processes involve stakeholders with different background and expertise. A technical RBAC model may be well-suited for software architects and developers, but for non-technical domain experts an abstracted view is desirable. In the context of model-driven development (MDD) [296, 299, 309], a systematic approach for DSL development has emerged in recent years (see, e.g., [228, 308, 318, 376]). A DSL is a tailor-made (computer) language for a specific problem domain. To ensure compliance between models and software platforms, models defined in a DSL are mapped to code artifacts via automated model-transformations (see, e.g., [227, 300, 375]). In our approach, the use of a DSL for RBAC constraints allows us to abstract from technical details and involve domain experts in the security modeling procedure.

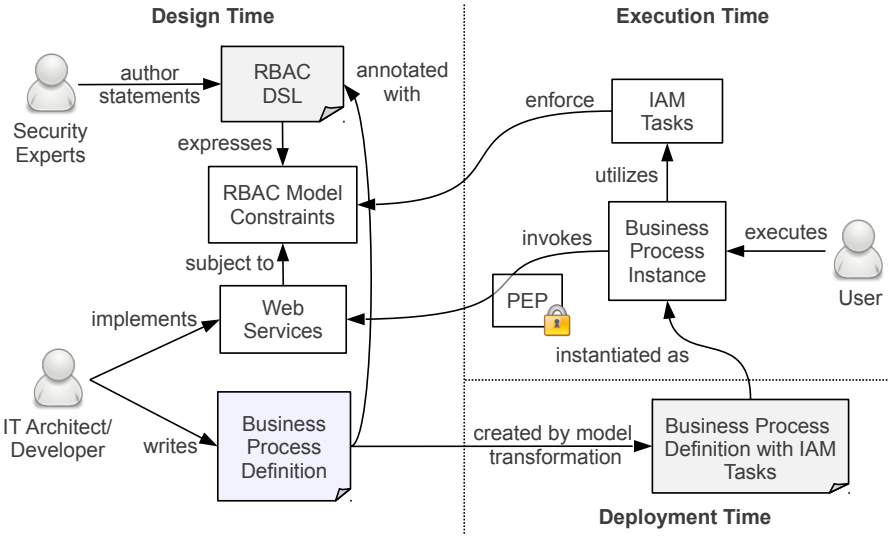


Figure 5.1: Overview of Access Constraint Enforcement Approach

Figure 5.1 depicts a high-level overview of our approach, including the involved stakeholders, system artifacts, and relationships between them. At design time, the security experts author RBAC DSL statements to define the RBAC model and entailment constraints. IT specialists implement Web services and define business processes on top of the services. At deployment time, the process definition files are automatically enriched with tasks for IAM (identity and access management) that conform to the corresponding entailment constraints. The business process is instantiated and executed by human individuals, and the IAM tasks ensure that the process conforms to the constraints defined in the RBAC model. A Policy Enforcement Point (PEP) component intercepts all service invocations to block unauthorized access (see also [134]).

For the sake of platform independence, we model business processes using UML activity diagrams [249]. In particular, we use the *BusinessActivities* extension [316], which enables the definition of process-related RBAC models via extended UML activity models. Based on the generic solution, we discuss a concrete instantiation and show how the approach is mapped to the Web services technology stack, including WS-BPEL [246].

The remainder of this chapter is structured as follows. In Section 5.2, we present a motivating scenario. Section 5.3 introduces a generic metamodel for specification of process-related RBAC models including entailment constraints. Section 5.4 describes the transformation procedure that enriches the process definitions with IAM tasks to enforce runtime-compliance. In Section 5.5, we present a concrete WS-BPEL based instantiation of our approach. Implementation-related details are given in Section 5.6, and in Section 5.7 we evaluate different aspects of our solution. Section 5.8 discusses related work, and Section 5.9 concludes with an outlook for future work.

5.2 Scenario

We illustrate the concepts of this chapter based on a scenario taken from the e-health domain. The scenario models the workflow of orthopedic hospitals which treat fractures and other serious injuries. The hospitals are supported by an IT infrastructure organized in a SOA, implemented using Web services. The SOA provides Web services for patient data, connects the departments of different hospitals, and facilitates the routine processes. Because the treatment of patients is a critical task and the personal data constitute sensitive information, security must be ensured and a tailored domain-specific RBAC model needs to be enforced. Task-based entailment constraints in the form of mutual exclusion and binding constraints are a crucial part of the system.

5.2.1 Patient Examination Business Process

A core procedure in the hospital is the patient examination, illustrated in Figure 5.2 as a *Business Activity* [316] model. We assume that this procedure is implemented using a business process engine and that the actions (tasks) represent the invocations of services. The arrows between the actions indicate the control flow of the process. All tasks are backed by technical services, however, part of the tasks are not purely technical but involve some human labor or interaction.

The top part of the figure shows the BusinessActivity model of the process, and the bottom part contains an excerpt of the RBAC definitions that apply to the scenario. We define three types of roles (Staff, Physician, Patient), each with a list of tasks they are permitted to execute, and four subjects (John, Jane, Bob, Alice), each with roles assigned to them. The names of permitted tasks of a role are displayed after the string *"Task:"*, following the grahical notation in [316]. Role inheritance hierarchies are modeled using the role-to-role assignment (*rrAssign*) relationship (senior-roles inherit the permissions of junior-roles, e.g., Physician inherits from Staff). The role-to-subject assignment (*rsAssign*) association is used to assign roles to subjects.

The first step in the examination process (see Figure 5.2) is to retrieve the personal data of the patient. To demonstrate the cross-organizational character of this scenario, suppose that the patient has never been treated in our example hospital (H1) before, but has already received medical treatment in a partner hospital (H2). Consequently, H1 obtains the patient's personal data via the Web services of H2. Secondly, the patient is assigned to a physician. After the patient has been assigned, the physician requests an x-ray image from the responsible department. The physician then decides whether additional data are required (e.g., information about similar fractions or injuries in the past). If so, the business process requests historical data from partner hospitals which also participate in the SOA. For privacy reasons, the historical data are only

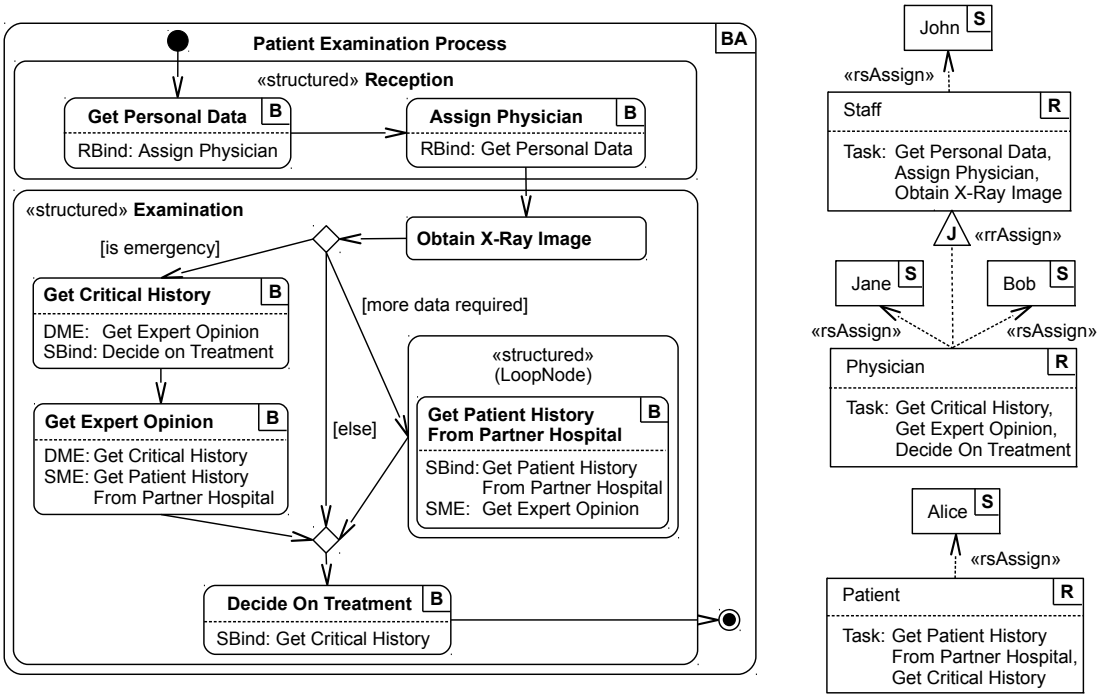


Figure 5.2: Patient Examination Scenario Modeled as UML Business Activity

disclosed to the patient herself, and the *Get Patient History* service task has to execute under the role *Patient* (see Figure 5.2). Another situation that requires additional data is the case of an emergency. If the emergency demands for immediate surgery, it is important to determine historical data about any critical conditions or diseases that might interfere with the surgery (task *Get Critical History*). To avoid that a single physician takes wrong decisions in an emergency, it is mandatory to get the opinion of a second expert. Finally, the task *Decide On Treatment* completes the examination and triggers the (physical) treatment.

5.2.2 Entailment Constraints

We support four types of entailment constraints which we briefly discuss in the following. The scenario process in Figure 5.2 contains examples for each type of constraint.

- **Static Mutual Exclusion** (SME): The SME constraint between *Get Expert Opinion* and *Get Patient History from Partner Hospital* defines that the two tasks must never be executed by the same subject or role, across all process instances. This constraint is reasonable as we need to explicitly separate the permissions of patients and physicians.
- **Dynamic Mutual Exclusion** (DME): The DME constraint for *Get Critical History* and *Get Expert Opinion* requires that, for each instance of the process, these two tasks are executed by different subjects. This ensures that the treatment decision in an emergency clearly depends on the medical assessment of two individual physicians.

- **Subject Binding** (SBind): An example SBind constraint is the *Get Patient History From Partner Hospital* task, which executes multiple times in a loop. To ensure that each iteration is done by the same subject, the *SBind* attribute reflexively links to the same task. A second subject binding exists between *Get Critical History* and *Decide on Treatment*.
- **Role Binding** (RBind): The process defines a role-binding constraint which demands that the *Get Personal Data* and *Assign Physician* are performed by the same role (although potentially different subjects).

5.3 Metamodel for Specification of Entailment Constraints in Business Processes

This section gives an overview of the generic metamodel for specification of process-related RBAC models including entailment constraints. To provide a self-contained view, Section 5.3.1 repeats the core definitions from [316], which form the basis for our approach. In Section 5.3.2, we introduce the textual RBAC DSL which allows to define entailment constraints in a simple textual syntax and enables a seamless mapping of UML-based process-related RBAC models (see [316]) to the implementation level.

5.3.1 Business Activity RBAC Models

Definition 8 (Business Activity RBAC Model) A Business Activity RBAC Model $BRM = (E, Q, D)$ where $E = S \cup R \cup P_T \cup P_I \cup T_T \cup T_I$ refers to pairwise disjoint sets of the metamodel, $Q = rh \cup tra \cup rsa \cup ptd \cup pi \cup ti \cup es \cup er$ to mappings that establish relationships, and $D = sb \cup rb \cup sme \cup dme$ to binding and mutual exclusion constraints, such that:

* For the sets of the metamodel:
  - An element of S is called *Subject*. $S \neq \emptyset$.
  - An element of R is called *Role*. $R \neq \emptyset$.
  - An element of P_T is called *Process Type*. $P_T \neq \emptyset$.
  - An element of P_I is called *Process Instance*.
  - An element of T_T is called *Task Type*. $T_T \neq \emptyset$.
  - An element of T_I is called *Task Instance*.

In the list below, we iteratively define the partial mappings of the Business Activity RBAC Model and provide corresponding formalizations ($\mathcal{P}$ refers to the power set, for further details see [316]):

1. The mapping $rh : R \rightarrow \mathcal{P}(R)$ is called ***role hierarchy***. For $rh(r_s) = R_j$ we call r_s *senior role* and R_j the set of direct *junior roles*. The transitive closure rh^* defines the inheritance in the role hierarchy such that $rh^*(r_s) = R_{j^*}$ includes all direct and transitive junior roles that the senior role r_s inherits from. The role hierarchy is cycle-free, i.e. for each $r \in R : rh^*(r) \cap \{r\} = \emptyset$.

2. The mapping $tra : R \rightarrow \mathcal{P}(T_T)$ is called ***task-to-role assignment***. For $tra(r) = T_r$ we call $r \in R$ *role* and $T_r \subseteq T_T$ is called the set of *tasks assigned to* r. The mapping $tra^{-1} : T_T \rightarrow \mathcal{P}(R)$ returns the set of roles a task is assigned to (the set of roles owning a task).
   This assignment implies a mapping ***task ownership*** $town : R \rightarrow \mathcal{P}(T_T)$, such that for each role $r \in R$ the tasks inherited from its junior roles are included, i.e. $town(r) = \bigcup_{r_{inh} \in rh^*(r)} tra(r_{inh}) \cup tra(r)$. The mapping $town^{-1} : T_T \rightarrow \mathcal{P}(R)$ returns the set of roles a task is assigned to (directly or transitively via a role hierarchy).

3. The mapping $rsa : S \rightarrow \mathcal{P}(R)$ is called ***role-to-subject assignment***. For $rsa(s) = R_s$ we call $s \in S$ *subject* and $R_s \subseteq R$ the set of *roles assigned to this subject* (the set of roles owned by s). The mapping $rsa^{-1} : R \rightarrow \mathcal{P}(S)$ returns all subjects assigned to a role (the set of subjects owning a role).
   This assignment implies a mapping ***role ownership*** $rown : S \rightarrow \mathcal{P}(R)$, such that for each subject $s \in S$ all direct and inherited roles are included, i.e. $rown(s) = \bigcup_{r \in rsa(s)} rh^*(r) \cup rsa(s)$. The mapping $rown^{-1} : R \rightarrow \mathcal{P}(S)$ returns all subjects assigned to a role (directy or transitively via a role hierarchy).

4. The mapping $ptd : P_T \rightarrow \mathcal{P}(T_T)$ is called ***process type definition***. For $ptd(p_T) = T_{p_T}$ we call $p_T \in P_T$ process type and $T_{p_T} \subseteq T_T$ the set of task types associated with p_T.

5. The mapping $pi : P_T \rightarrow \mathcal{P}(P_I)$ is called ***process instantiation***. For $pi(p_T) = P_i$ we call $p_T \in P_T$ *process type* and $P_i \subseteq P_I$ the set of process instances instantiated from process type p_T.

6. The mapping $ti : (T_T \times P_I) \rightarrow \mathcal{P}(T_I)$ is called ***task instantiation***. For $ti(t_T, p_I) = T_i$ we call $T_i \subseteq T_I$ set of *task instances*, $t_T \in T_T$ is called *task type* and $p_I \in P_I$ is called *process instance*.

7. Because role-to-subject assignment is a many-to-many relation (see Def. 8.3), more than one subject may be able to execute instances of a certain task type. The mapping $es : T_I \rightarrow S$ is called ***executing-subject*** mapping. For $es(t) = s$ we call $s \in S$ the *executing subject* and $t \in T_I$ is called *executed task instance*.

8. Via the role hierarchy, different roles may posses the permission to perform a certain task type (see Def. 8.1 and Def. 8.2). The mapping $er : T_I \rightarrow R$ is called ***executing-role*** mapping. For $er(t) = r$ we call $r \in R$ the *executing role* and $t \in T_I$ is called *executed task instance*.

9. The mapping $sb : T_T \rightarrow \mathcal{P}(T_T)$ is called ***subject-binding***. For $sb(t_1) = T_{sb}$ we call $t_1 \in T_T$ the *subject binding task* and $T_{sb} \subseteq T_T$ the set of *subject bound tasks*.

10. The mapping $rb : T_T \rightarrow \mathcal{P}(T_T)$ is called ***role-binding***. For $rb(t_1) = T_{rb}$ we call $t_1 \in T_T$ the *role binding task* and $T_{rb} \subseteq T_T$ the set of *role bound tasks*.

11. The mapping $sme : T_T \rightarrow \mathcal{P}(T_T)$ is called ***static mutual exclusion***. For $sme(t_1) = T_{sme}$ with $T_{sme} \subseteq T_T$ we call each pair $t_1 \in T_T$ and $t_x \in T_{sme}$ *statically mutual exclusive tasks*.

12. The mapping $dme : T_T \rightarrow \mathcal{P}(T_T)$ is called ***dynamic mutual exclusion***. For $dme(t_1) = T_{dme}$ with $T_{dme} \subseteq T_T$ we call each pair $t_1 \in T_T$ and $t_x \in T_{dme}$ *dynamically mutual exclusive tasks*.

5.3.2 RBAC Modeling for Business Processes

Figure 5.3 depicts the core RBAC metamodel and its connection with the core elements of the BusinessActivity metamodel. In particular, Figure 5.3 outlines how we extended our DSL from [134] to include process-related RBAC entailment constraints (see [316]). The different model elements are described below.

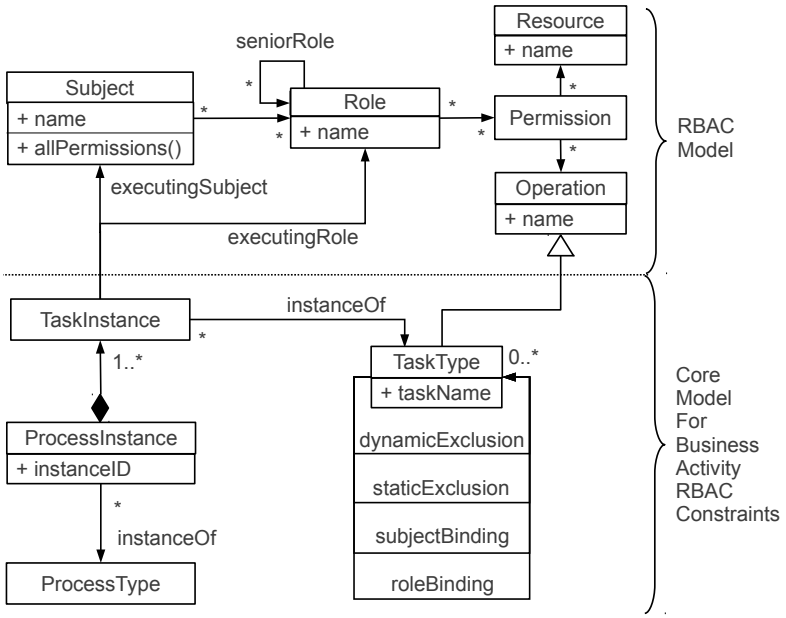


Figure 5.3: Excerpt of RBAC Metamodel and Business Activity Metamodel

A `ProcessInstance` has a unique `instanceID`, a `ProcessType`, and is composed of multiple `TaskInstance` objects which are again instances of a certain `TaskType`. The class `TaskType` has a `name` and four reflexive associations that define mutual exclusion and binding constraints. `Subjects` are identified by a `name` attribute and are associated with an arbitrary number of `Roles`, which are themselves associated with `Permissions` to execute certain `Operations`. A `TaskType` in the BusinessActivity metamodel corresponds to an `Operation` in the RBAC metamodel. Roles may inherit permissions from other roles (association `seniorRole`). In our approach, we directly associate Web service instances with `Resources`. That is, a subject that attempts to invoke a Web service operation *op* on a service resource *res* must be associated with a role that holds a permission to execute *op* on *res*. A detailed description of the BusinessActivity metamodel and corresponding Object Constraint Language (OCL) constraints can be found in [316]. We utilize the core parts of this model

and focus on the mapping of the RBAC constraints to a textual DSL and to business process execution platforms, as illustrated for WS-BPEL in Section 5.5.

5.3.3 RBAC DSL Statements

Our RBAC DSL is implemented as an embedded DSL [376] and is based on the scripting language *Ruby* as host programming language. We now briefly discuss how the model elements are mapped to language constructs provided by the DSL (see also Section 5.3.1 and Figure 5.3). Table 5.1 lists the basic DSL statements (left column) and the corresponding effect (right column). In the table, keywords of the DSL syntax are printed in **`bold typewriter`** font, and placeholders for custom (scenario-specific) expressions are printed in *italics*.

RBAC DSL Statement	Effect
`RESOURCE` *name* [*description*]	Define new resource
`OPERATION` *name* [*description*]	Define new operation
`SUBJECT` *name* [*description*]	Define new subject
`ROLE` *name* [*description*]	Define new role
`ASSIGN` *subject role*	Assign role to subject
`INHERIT` *juniorRole seniorRole*	Let senior role inherit a junior role
`PERMIT` *role operation resource*	Allow a role to execute a certain operation on a specific resource
`TASK` *name operation resource*	Define operation-to-task mapping
`DME` *task*1 *task*2	Define dynamic mutual exclusion (DME)
`SME` *task*1 *task*2	Define static mutual exclusion (SME)
`RBIND` *task*1 *task*2	Define role-binding (RBind)
`SBIND` *task*1 *task*2	Define subject-binding (SBind)

Table 5.1: Semantics of RBAC DSL Statements

The RBAC DSL statements `RESOURCE`, `OPERATION`, `SUBJECT` and `ROLE` are used to create resources, operations, subjects and roles with the respective $name$ and optional $description$ attributes. `ASSIGN` creates an association between a subject and a role. `INHERIT` takes two parameters, a junior-role and a senior-role name, and causes the senior-role to inherit all permissions of the junior-role. `PERMIT` expresses the permission for a role to execute a certain operation on a resource. DME and SME allow the specification of dynamically or statically mutual exclusive operations. Using `RBIND` and `SBIND`, two operations are subjected to role-binding or subject-binding constraints. Finally, the `TASK` statement is used to establish a mapping from our RBAC DSL to implementation level artifacts. More precisely, operations are mapped to concrete WS-BPEL tasks (see Section 5.5.2). The complete access control configuration for the patient examination scenario, expressed via RBAC DSL statements, is printed in Appendix A.1.

5.4 Process Model Transformations for Runtime Constraint Enforcement

To enforce the RBAC constraints at runtime, the business process needs to follow a special procedure. If the process executes a secured task, it needs to provide a valid authentication

token for the active user. For instance, this token contains information which subject (e.g., "Jane") executes an operation, and under which role (e.g., "Staff") this individual operates. In this section, we discuss our approach for automatically obtaining these authentication tokens to enforce security at runtime.

Figure 5.4 illustrates which artifacts are utilized by the instances of the business process. We follow the concepts of the Security Assertion Markup Language (SAML) framework [245] and provide the authentication data with the aid of an *Identity Provider (IdP)* service. An IdP is a service provider that maintains identity information for users and provides user authentication to other services. The IdP is a reusable standard component; its sole responsibility is to authenticate the user and to issue an *AuthData* document which asserts the user's identity (subject and role). As such, the IdP has no knowledge about the process structure and RBAC constraints. Hence, we utilize the decoupled RBAC Manager Service which keeps track of the state of the process instances. The RBAC Manager Service knows the process structure and decides, based on the RBAC constraints, which subject or role is responsible for the next task (see also [315]).

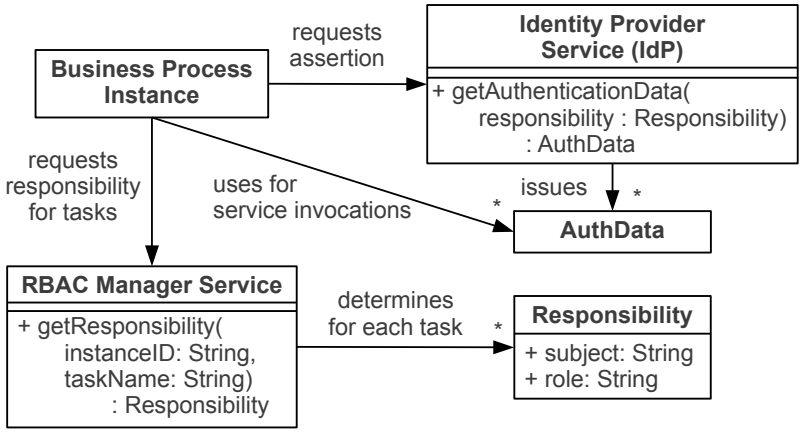
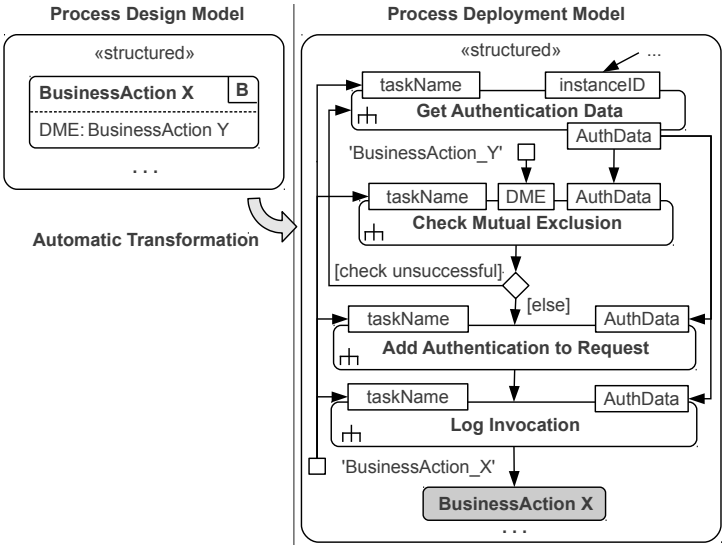


Figure 5.4: Relationship Between Business Process Instance and Security Enforcement Artifacts

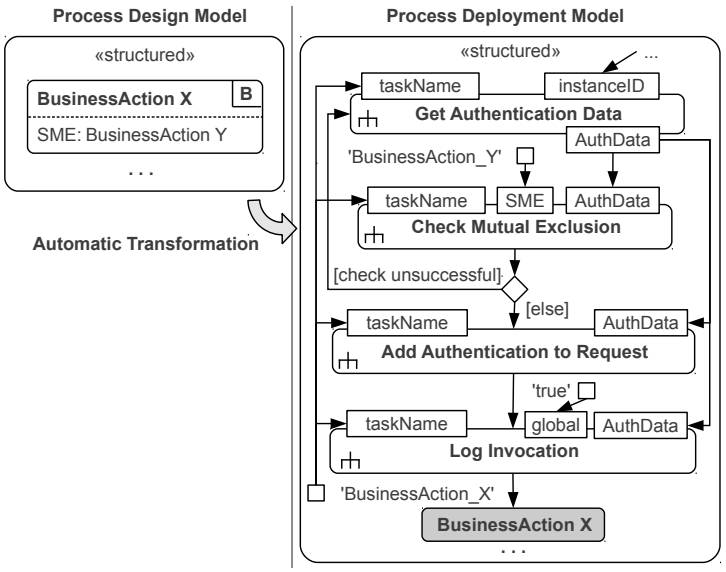
Combining the functionality of `getResponsibility` and `getAuthenticationData` (see Figure 5.4) constitutes the core protocol for obtaining authentication tokens that enable the enforcement of task-based entailment constraints in a BusinessActivity. This recurring protocol is executed for each secured task; hence, it need not be implemented manually, but should ideally be generated automatically on top of the business process model that is defined by the developer. We therefore aim at providing automatic transformations to convert the domain-specific extensions for mutual exclusion and binding constraints in BusinessActivity models into regular activity models which perform the required IAM tasks. This transformation is required as an intermediate step towards the generation of corresponding definitions that are directly deployable and executable (e.g., by WS-BPEL engines). In the following, we describe the transformation procedure in detail and discuss different implementation and runtime aspects.

5.4.1 Model Transformations to Enforce Mutual Exclusion Constraints

Here we discuss the detailed procedure for runtime enforcement of mutual exclusion constraints. The design-time BusinessActivity models are transformed into deployable standard activity models that comply with this procedure. The necessary transformations are illustrated in Figure 5.5. Tasks representing invocations to external Web services are printed in grey, while structured activities and tasks with local processing logic are depicted with a white background.



(a) Transformation for DME Constraints



(b) Transformation for SME Constraints

Figure 5.5: Process Transformations to Enforce Mutual Exclusion Constraints

The transformed activity models with mutual exclusion constraints in Figure 5.5 contain four additional tasks. All four tasks are UML *CallBehaviorAction*s [249] (indicated by the rake-style symbol) which consist of multiple sub-tasks. The internal processing logic depends on the concrete target platform; later in Section 5.5.1 we discuss the detailed logic for WS-BPEL.

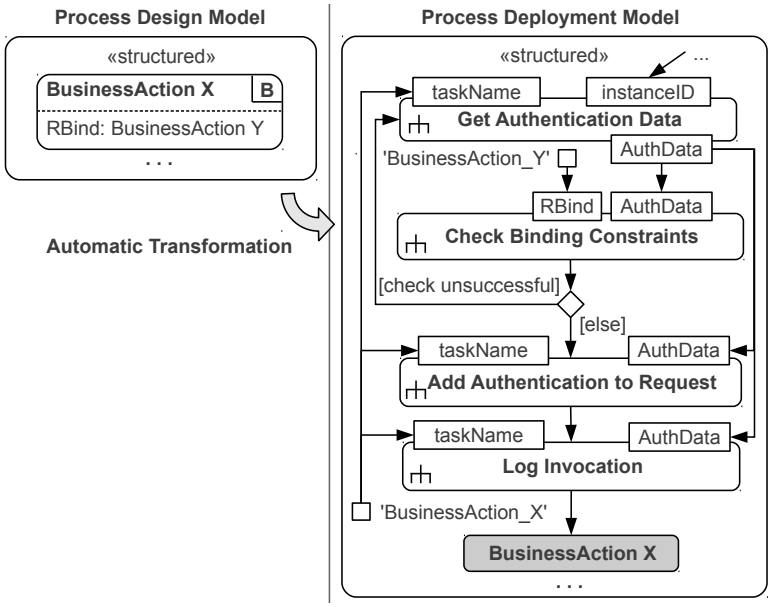
The task *Get Authentication Data* invokes the IdP service to obtain the authentication data token (*AuthData*) to be used for later invocation of the BusinessAction. The second inserted task is *Check Mutual Exclusion*, which is responsible for checking whether the provided authentication data are valid with respect to the mutual exclusion constraint. A UML value pin [249] holding the name of the corresponding task provides the input for the pin *DME* (Figure 5.5(a)) or the pin *SME* (Figure 5.5(b)), respectively. Additionally, the *Check Mutual Exclusion* task receives as input the name of the task to-be-executed (*taskName*, which is known from the original process definition), and the *AuthData* (received from the IdP service). The decision node is used to determine whether *Check Mutual Exclusion* has returned a successful result. If the result is unsuccessful (i.e., a constraint violation has been detected) the control flow points back to *Get Authentication Data* to ask the IdP again for a valid authentication data token. Otherwise, if the result is successful, the task *Add Authentication to Request* appends the user credentials in *AuthData* to the request message for the target Web service operation. The fourth inserted task is *Log Invocation*, which adds a new log record that holds the name of the task (*taskName*) and the *AuthData* of the authenticated user. The input pin *global* determines whether the log entry is stored in a local variable of the process instance (value *null*) or in a global variable accessible from all process instances (value *'true'*).

5.4.2 Model Transformations to Enforce Binding Constraints

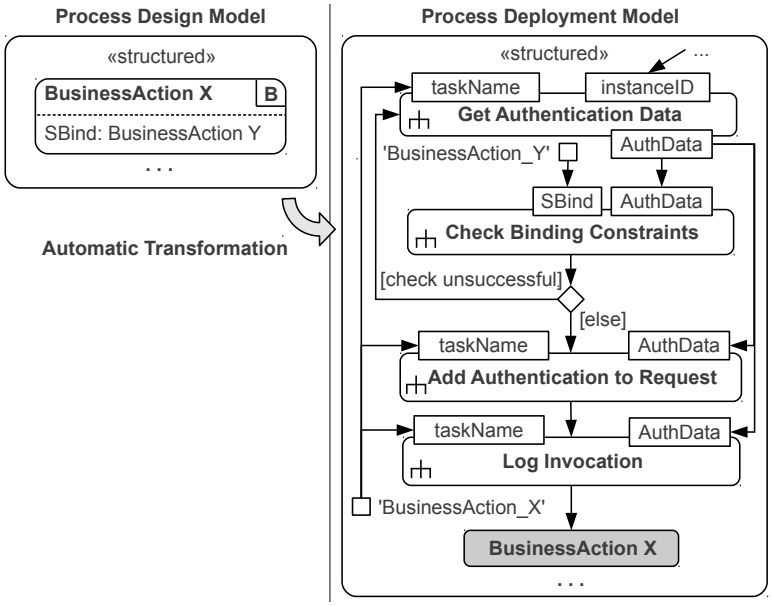
The approach for transforming binding constraints in BusinessActions (illustrated in Figure 5.6) is similar to the transformation for mutual exclusion constraints presented in Section 5.4.1. The transformed process model first requests authentication data from the IdP service. The task *Check Binding Constraints* then checks the resulting *AuthData* with respect to role-bindings (*RBind*, Figure 5.6(a)) and subject-bindings (*SBind*, Figure 5.6(b)). The process asks for new user credentials and repeats the procedure if the binding constraint is not fulfilled.

Note that the entailment constraints are checked directly inside the process, not by the IdP service. Even though the AuthData (subject, role) obtained from the IdP is trusted and assumed to properly represent the user executing the process, the AuthData may be invalid with respect to entailment constraints. Hence, the branch "check unsuccessful" indicates that the process instance asks for a different user to login and execute the task. As the log of previous tasks is stored locally by each process instance (except for SME constraints, where log entries are also stored globally), the *Check Binding* and *Check Mutual Exclusion* tasks are required directly inside the process logic and are not outsourced to external services. This approach is able to deal with deadlock situations (evaluated in Section 5.7.2).

In certain deployments, the platform providers (e.g., hospital management) may be interested in tracking failed authorizations. For brevity, such mechanisms are not included in Figures 5.5 and 5.6, but extending the approach with notifications is straight-forward.



(a) Transformation for Role Binding



(b) Transformation for Subject Binding

Figure 5.6: Process Transformations to Enforce Binding Constraints

5.4.3 Transformation Rules for Combining Multiple Constraints

So far, the transformation rules for the four different types of entailment constraints in BusinessActivities (role-binding, subject-binding, SME, DME) have been discussed in isolation. How-

ever, as the scenario in Section 5.2 illustrates, BusinessActions can possibly be associated with multiple constraints (e.g., *Get Critical History*). Therefore, we need to analyze how the transformation rules can be combined while still maintaining the constraints' semantics. A simple approach would be to successively apply the atomic transformations for each BusinessAction and each of the constraints associated with it. However, this approach is not suited and may lead to incorrect results. For instance, if we consider the task *Get Critical History* with the associated DME and SBind constraints, the process might end up requesting the authentication data twice, which is not desired. Therefore, multiple constraints belonging to the same task are always considered as a single unit (see also [315]).

Figure 5.7 depicts the transformation template for a generic sample BusinessAction X with multiple constraints $c_1, c_2, \ldots, c_n$.

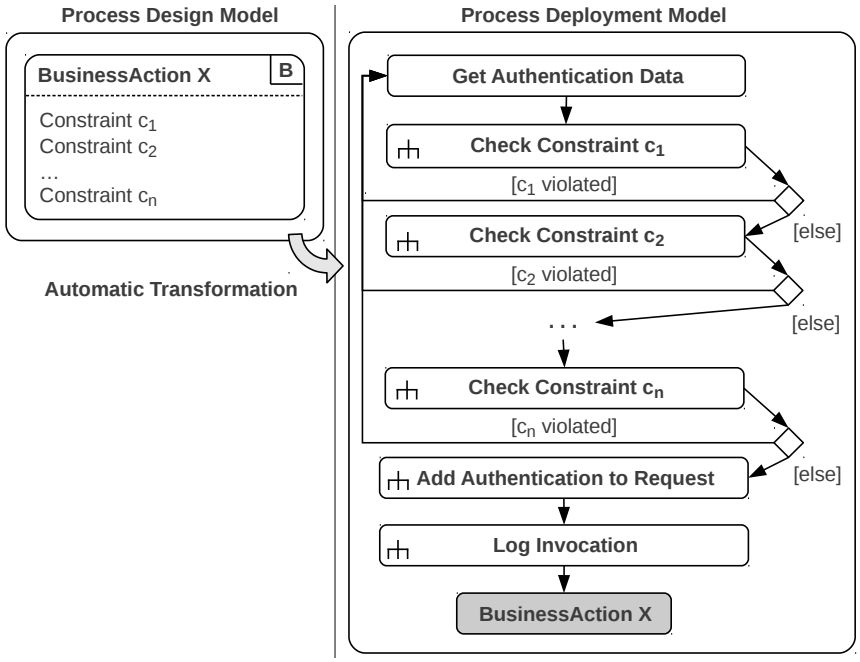


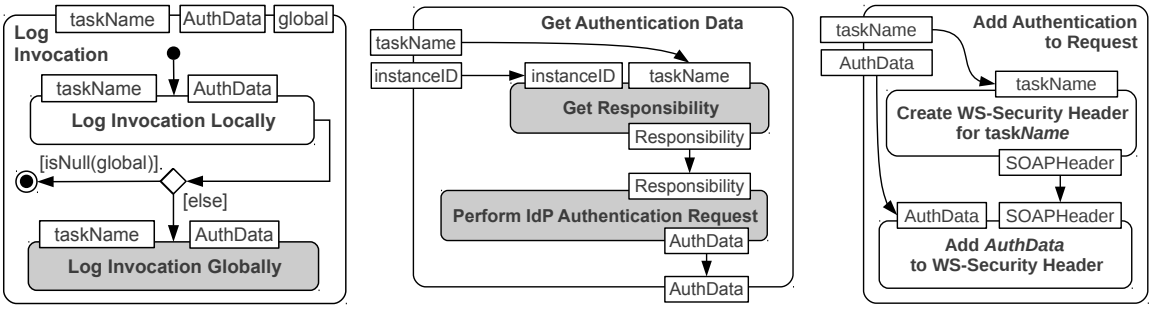
Figure 5.7: Generic Transformation Template for Business Action With Multiple Constraints

5.5 Application to SOA and WS-BPEL

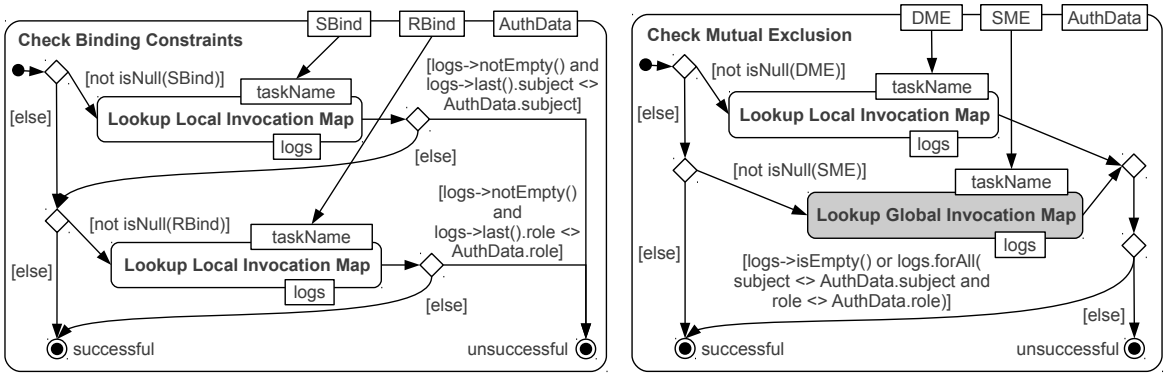
This section discusses details of the process transformation from Section 5.4 and illustrates how the approach is applied to SOA, particularly WS-BPEL and the Web services framework.

5.5.1 Supporting Tasks for IAM Enforcement in WS-BPEL

In the following we discuss the internal logic of the five supporting IAM tasks used in the transformed activity models for the enforcement of mutual exclusion (Section 5.4.1) and binding constraints (Section 5.4.2).



(a) Log Invocation (b) Get Authentication Data (c) Add Authentication to Request



(d) Check Binding Constraints (e) Check Mutual Exclusion

Figure 5.8: Supporting Tasks for IAM Enforcement in WS-BPEL

Task **Log Invocation**: In general, process-related RBAC constraints rely on knowledge about historical task executions (see also [316]). Therefore, a mechanism is required to store data about previous service invocations. One conceivable approach is that the process execution engine keeps track of the invocation history. To that end, invocation data can be stored either in a local variable of the process instance (for DME constraints) or in a global variable that is accessible from all process instances (for SME constraints). Unfortunately, WS-BPEL does not support global variables, but we can overcome this issue by using an external logging Web service. Figure 5.8(a) shows the *Log Invocation* activity, which stores data about service calls, including the name of the invocation and the *AuthData* of the user under which the process executes. The invocation is first stored in a local array variable of WS-BPEL. If the input pin named *global* is not null, the data is also stored with the external logging service (*Log Invocation Globally*). Currently, our framework relies on a central logging service. In our future work, we tackle advanced challenges such as privacy, and timing issues that come with decentralized logging.

Task **Get Authentication Data**: This supporting IAM task is used to obtain authentication tokens, see Figure 5.8(b). The identifier of the affected process task is provided as a parameter *taskName*. For instance, in the case of WS-BPEL, the `name` attribute of the corresponding `invoke` statement can be used to determine this value. As outlined in Section 5.4, the procedure is split up between the RBAC Manager service and the IdP. First, the invocation *Get Respon-*

sibility asks the RBAC Manager for the role or subject responsible for executing the next task. All combinations of values are possible, i.e., either subject or role, or both, or none of the two may be specified. The subject/role responsibility information is used to execute an *IdP Authentication Request*. The authentication method performed by the IdP is transparent; for instance, it may perform smartcard based authentication or ask for username/password. The *AuthData* output pin provided by this invocation contains the definite subject and role name of the user.

Task **Add Authentication to Request**: The activity in Figure 5.8(c) illustrates how authentication data are appended to the invocation of Business Actions. First, the *AuthData* information is used to request a SAML assertion from the IdP service. This token contains the subject and role with a trusted signature that ensures the integrity of the assertion content. The assertion is then added to the request message for the target service operation (the name is specified via the input pin *taskName*) using the SOAP header mechanism [366]. Note that this activity leaves room for optimization. If many tasks in the process are executed by the same subject and role, it is advantageous to cache and reuse the SAML tokens in a local variable of the process instance. However, caching security tokens carries the risk of inconsistencies if the RBAC policies change.

Task **Check Binding Constraints**: Figure 5.8(d) contains the activity *Check Binding Constraints*, whose internal logic is to check the logged invocations with role-binding and subject-binding against the *AuthData* information. If the *SBind* parameter is set, the activity looks up the last corresponding log entry (the *taskName* of the log entry needs to be equal to *SBind*) in the local invocation map of the WS-BPEL process instance. If the returned array (named *logs*) is not empty, then the subject stored in the last log entry needs to be identical to the subject in *AuthData*. Analogously, if the *RBind* parameter is set, then the role of the last log entry with *taskName* equal to *RBind* must be equal to the role in *AuthData*. If and only if all conditions hold true, the activity returns a success status.

Task **Check Mutual Exclusion**: Similarly, the *Check Mutual Exclusion* activity in Figure 5.8(e) uses the log data to check the *AuthData* against the previously performed invocations. If the input parameter *DME* is set, WS-BPEL looks up the log entries from the local invocation map. Otherwise, if an *SME* parameter is provided, the corresponding logs are received from the external logging service (global invocation map). The activity returns a successful result if either the *logs* sequence is empty or all log entries have a different subject and role than the given *AuthData*. Due to the possibly large number of entries in the *logs* sequence, it is crucial where these conditions are evaluated (by the process or the logging service directly). To avoid transmitting log data over the network, we recommend the implementation variant in which the logging service itself validates the conditions. To that end, *AuthData* is sent along with the request to the logging service and the service returns a boolean result indicating whether the constraints are satisfied.

5.5.2 RBAC DSL Integration with WS-BPEL

The `TASK` statement of the RBAC DSL realizes a mapping from operations to concrete WS-BPEL tasks (`invoke` activities). This corresponds to the model in Figure 5.3, where `TaskType` in the Business Activities metamodel is mapped to `Operation` in the RBAC metamodel. Using this mapping, we are able to automatically apply all Business Activity entailment constraints to the corresponding WS-BPEL `invoke` activities.

DSL Statement	WS-BPEL DSL Statement
DME $task1$ $task2$	<invoke name="$task1$" **rbac:dme=**"$task2$" ../>
SME $task1$ $task2$	<invoke name="$task1$" **rbac:sme=**"$task2$" ../>
SBIND $task1$ $task2$	<invoke name="$task1$" **rbac:sbind=**"$task2$" ../>
RBIND $task1$ $task2$	<invoke name="$task1$" **rbac:rbind=**"$task2$" ../>

Table 5.2: Mapping of RBAC DSL Statements to WS-BPEL DSL Statements

In our approach, WS-BPEL `invoke` activities are constrained using specialized DSL statements. The DSL uses the extension mechanism of WS-BPEL and introduces new XML attributes `rbac:dme`, `rbac:sme`, `rbac:sbind` and `rbac:rbind` (the prefix `rbac` refers to the corresponding XML namespace). These attributes are directly annotated to the `invoke` activities in WS-BPEL. Table 5.2 illustrates how the relevant RBAC DSL statements are mapped to the corresponding WS-BPEL DSL statements. For instance, the `DME` statement is mapped to a `rbac:dme` attribute. The parameters of the DSL statements in Table 5.2 refer to the task types defined using the `TASK` statement (see Section 5.3.3). The `rbac:`$\star$ attributes can contain multiple valued separated by commas, e.g., a task that is dynamically mutual exclusive to $task1$ and $task2$ can be annotated with a `rbac:dme="task1,task2"` attribute.

5.5.3 Automatic Transformation of WS-BPEL Definition

At deployment time, the business process model is automatically transformed to ensure correct enforcement of identity and access control policies at runtime. The transformation can happen on different abstraction levels, either based on the platform-independent model (PIM) or on the platform-specific model (PSM) (see, e.g., [356]). On the PIM level, model transformation languages such as *Query/View/Transformation* (QVT) [248] can be used to perform UML-to-UML transformation of process activity models. Our approach proposes a transformation directly on the PSM model, i.e., the WS-BPEL process definition file.

Algorithm 4 gives a simplified overview of which WS-BPEL code fragments are injected, and where. Variable names are printed in *italics*, and XML markup and XPath expressions are in `typewriter` font. The input is a WS-BPEL document $bpel$ with security annotations. First, various required documents (e.g. the XSD files of SAML and WS-Security) are imported into the WS-BPEL process using `import` statements. Then the `partnerLink` declarations for the needed services (such as the IdP service) are added to $bpel$, and `variable` declarations are created (e.g., input/output variables for `getAuthenticationData` operations). Using `assign` statements, some variables (such as `ProcessInstanceID`) are initialized. Next, the algorithm loops over all `invoke` elements that have an attribute from the `rbac` namespace assigned. For every matching `invoke` several WS-BPEL code injections and transformations have to be conducted. First, an `invoke` statement (`authInvoke`) is created. At runtime, this statement calls the IdP's `getAuthenticationData` operation. Next, an empty set (`constraintChecks`) is created. Afterwards, the algorithm iterates over all constraints (e.g., `rbac:sbind`) that have been defined for this particular `invoke` statement. The values of every `constraint` are split by commas. For instance, in the case of an

Algorithm 4 WS-BPEL Transformation Algorithm

Input: WS-BPEL document $bpel$, Fragment Templates $tmpl$
Output: transformed WS-BPEL document

add <import ../>, <partnerLink ../>, and <variable ../> statements to $bpel$
add <assign ../> statements to initialize $ProcessInstanceID$ and $InvocationLogs$ variables
for all $bpel$//invoke as inv **do**
  if inv/@rbac:* **then**
    $authInvoke \leftarrow$ create <invoke ../> for operation *getAuthenticationData* and partnerLink *IdP*
    $constraintChecks \leftarrow \emptyset$
    for all inv/@rbac:* as $constraint$ **do**
      $tasks \leftarrow$ split value of $constraint$ by commas
      for all $tasks$ as $task$ **do**
        $check \leftarrow$ create <if>..</if> which checks outcome of $authInvoke$ for RBAC entailment constraint $constraint$ and task $task$
        $constraintChecks \leftarrow constraintChecks \cup check$
      end for
    end for
    enforcementBlock $\leftarrow$ wrap sequence $authInvoke||constraintChecks$ in new <while> block
    insert $enforcementBlock$ before inv
    if inv/@rbac:sme **then**
      $logInvoke \leftarrow$ create <invoke ../> for operation *logInvocation* via partnerLink *LoggingService*
      insert $logInvoke$ after inv
    end if
  end if
end for

rbac:dme="task1,task2" annotation, constraint is rbac:dme and tasks is a set with two elements (task1 and task2). For every task an if-block (check) is created. At runtime, this if-block checks if there is a violation of the entailment constraint constraint regarding another task task. Next, a new <while>-block (enforcementBlock) is created. This block envelopes the previously created authInvoke statement and all checks contained in constraintChecks. Finally, this enforcementBlock is inserted directly before the secured invoke statement. Just in case the latter is also annotated using a rbac:sme attribute, an additional invocation is injected right after the actual invoke element. This one calls the logInvocation operation via the LoggingService PartnerLink.

5.6 Implementation

The implementation of the proposed approach is integrated in the SeCoS[1] (*Se*cure *Co*llaboration in *S*ervice-based systems) framework, which is discussed in the following. This section is divided into four parts: firstly, we outline the architecture of the system and the relationship between the individual services and components in Section 5.6.1; secondly, the SAML-based SSO mechanism is described in Section 5.6.2; in Section 5.6.3 we present the algorithm for automatic transformation of WS-BPEL definitions containing security annotations from our DSL; finally, Section 5.6.4 discusses the implementation for checking constraints over the log data.

[1] http://www.infosys.tuwien.ac.at/prototype/SeCoS/

5.6.1 System Architecture

Figure 5.9 sketches the high-level architecture and relationships between the example process and the system components.

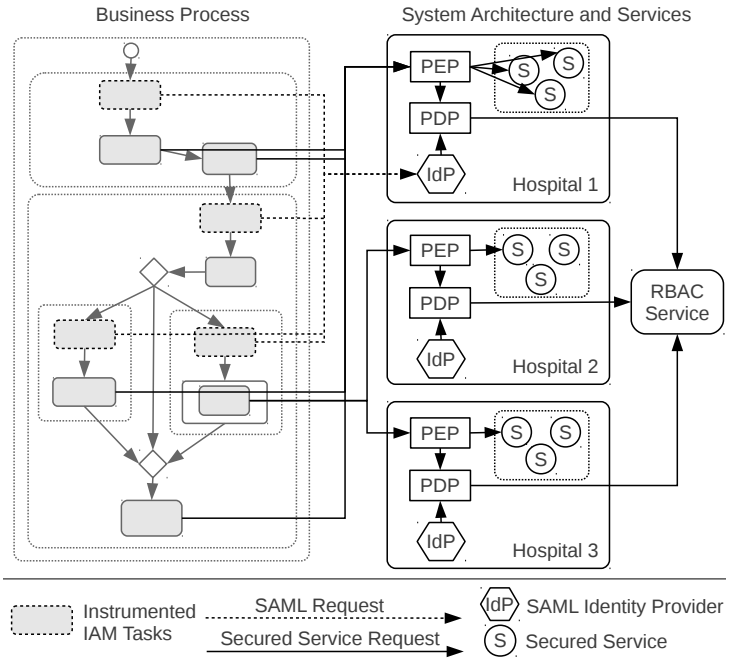


Figure 5.9: Example Process in System Architecture

The patient examination scenario from Section 5.2 is implemented using WS-BPEL and deployed in a Glassfish[2] server. The scenario involves three hospitals, which host the protected services for patient management and examination. All service invocations are routed through a PEP, which acts as a central security gateway, intercepts every incoming service request and either allows or disallows its invocation. It is important that the PEP operates transparently and as close to the protected resources (i.e., services) as possible. Using the Java API for XML Web services (JAX-WS), the PEP has been implemented as a SOAP message handler (interface `SOAPHandler`). This handler can be plugged into the Web service's runtime engine in a straightforward manner. Once activated, the interceptor is able to inspect and modify inbound and outbound SOAP messages and to deny service invocations.

Each hospital runs a SAML IdP service, which is used to issue the SAML assertions that are required in the WS-BPEL process. The IdP's responsibility is twofold: firstly, it authenticates users; secondly, the IdP assures the identity of a subject and its associated attributes (e.g., roles) by issuing a SAML assertion SOAP header which is used in subsequent service invocations. For the sake of an easy integration into the given system environment, we decided to use the JAX-WS API for implementing the Login Web service. This SOAP Web service offers a

[2] `https://glassfish.dev.java.net/`

`login` method. It requires a username/password pair and returns a SAML assertion. Internally, we utilize the Java Architecture for XML Binding (JAXB) for parsing and creating SAML assertions. Additionally, the Apache XML Security for Java[3] library is used for digitally signing XML documents (i.e., the SAML assertions).

The actual decision whether an invocation should be prevented or not is typically delegated to another entity, the PDP. When deciding over the access to a service resource the PDP has to make sure that the subject attempting to access the resource has the permission to do so. This decision is based on the policy information stored in the RBAC repository (which is based on the DSL statements authored by domain experts). In our implementation, the core functionality of the PDP is embedded into the RBAC DSL (see Section 5.3.2). That is, the DSL offers an `access` method that can be used to determine whether the requesting subject is permitted to access the target resource (service) under the specified context and role (see Figure 5.9). In order to make this functionality accessible to the outside of the DSL's interpreter, we developed a RESTful Web service, that bridges HTTP requests to the interpreter. More precisely, the PDP service uses the Bean Scripting Framework (BSF)[4] to access the interpreter. The Java API for RESTful Web Services (JAX-RS) is used to realize the PDP service's RESTful Web interface.

5.6.2 SAML-based Single Sign-On

Figure 5.10 depicts an example of the access control enforcement procedure modeled in UML. To illustrate the SSO aspect of the scenario, we assume that a patient with subject name "Alice" (cf. Figure 5.3), who is registered in hospital 2 (H2), is examined in hospital 1 (H1) and requests her patient history from previous examinations in hospital 3 (H3). The procedure is initiated by the WS-BPEL process which requests the execution of a protected Web service.

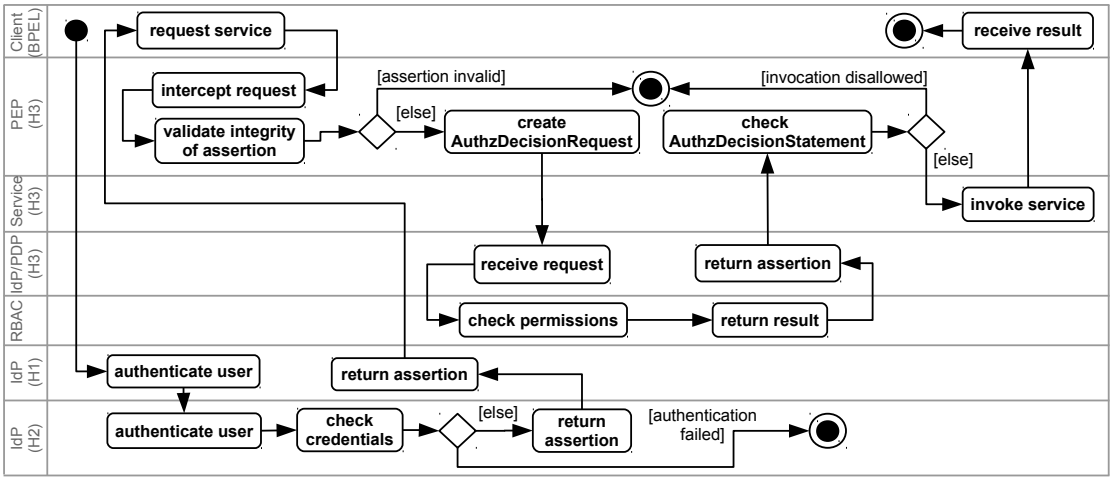


Figure 5.10: Identity and Access Control Enforcement Procedure

[3] `http://santuario.apache.org/`
[4] `http://commons.apache.org/bsf/`

Prior to issuing the actual service request, the user has to authenticate using the SAML IdP. The IdP queries the user database to validate the credentials provided by the client. As the credentials of user *Alice* are not stored in the Data Base (DB) of H1, the IdP contacts the IdP of H2, which validates the credentials.

If the user credentials could not be validated, the process is terminated prematurely and a SOAP fault message is returned. In our example scenario, the business process receives the fault message and activates corresponding WS-BPEL fault handlers. Otherwise, if the credentials are valid, the IdP creates a signed assertion similar to the one shown in Listing 5.1 and passes it back to the WS-BPEL process (see Figure 5.10). The business process attaches the assertion to the actual service request.

```
<Assertion>
 <Issuer>http://h2.com/IdP</Issuer>
 <ds:Signature>...</ds:Signature>
 <Subject><NameID>Alice</NameID></Subject>
 <Conditions NotBefore="2013-05-17T09:48:36.171Z"
             NotOnOrAfter="2013-05-17T10:00:36.171Z"/>
 <AttributeStatement>
   <Attribute Name="role">
     <AttributeValue>staff</AttributeValue>
   </Attribute>
 </AttributeStatement>
</Assertion>
```

Listing 5.1: Exemplary SAML Assertion Carrying Subject and Role Information

The example SAML assertion in Listing 5.1 illustrates the information that is encapsulated in the header token when the scenario process invokes the `getPatientHistory` operation of the patient Web service of H3. The assertion states that the subject named `Alice`, which has been successfully authenticated by the IdP of the hospital denoted by the `Issuer` element (H2), is allowed to use the the role `staff`. The included XML signature element ensures the integrity of the assertion, i.e., that the assertion content indeed originates from the issuing IdP (H2) and has not been modified in any way. When the PEP of H3 intercepts the service invocation with the SAML SOAP header, its first task is to verify the integrity of the assertion. The signature verification requires the public key of the IdP that signed the assertion; this key is directly requested from the corresponding IdP (under `http://h2.com/IdP`) using SAML Metadata [244]. The implementation uses the Apache XML Security for Java library to conduct the signature verification.

```
<Assertion>
  <Issuer>http://h3.com/IdP</Issuer>
  <ds:Signature>...</ds:Signature>
  <Subject>
    <NameID>Alice</NameID>
  </Subject>
  <AuthzDecisionStatement Decision="Permit"
    Resource="http://h3.com/patient">
    <Action>getPersonalData</Action>
  </AuthzDecisionStatement>
</Assertion>
```

Listing 5.2: Exemplary SAML Authorization Decision

After the PEP of H3 has verified the message integrity, it needs to determine whether the subject is authorized to access the requested service operation. This is achieved by the PDP service of H3 that allows the PEP to post a SAML Authorization Decision Query. The PDP answers this query by returning an assertion containing a SAML Authorization Decision Statement. Listing 5.2 shows an example SAML assertion which informs the PEP that our staff member is allowed to invoke the action (operation) `getPersonalData` of the resource (Web service) `http://h1.com/patient`. Analogously to the IdP service, we also used the JAX-WS API to implement the SOAP-based interface of the PDP service. The PDP offers the method `query`, which takes an Authorization Decision Query message as argument and returns an Authorization Decision Statement. Again, we leverage JAXB for parsing the SAML documents.

5.6.3 Automatic Transformation of WS-BPEL Definition

Since both WS-BPEL and SAML are XML based standards, we are able to reuse and utilize the broad line-up of existing XML tooling. The transformation procedure of WS-BPEL process definitions is hence based on XSLT (Extensible Stylesheet Language Transformations) [367], a language for arbitrary transformation and enrichment of XML documents.

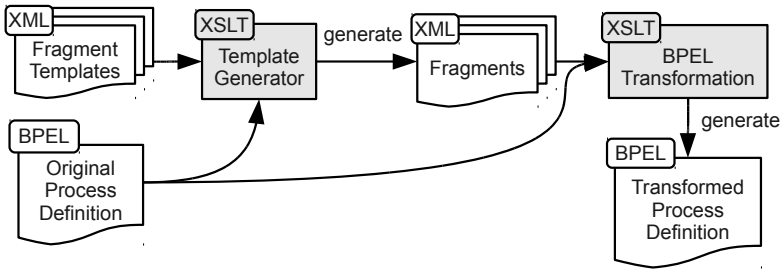


Figure 5.11: Artifacts of the Transformation Process

In general, the original WS-BPEL process is transformed by enriching the process definition file with code fragments that perform the IAM tasks (cf. Section 5.5.1). In principle, these fragments are generic and static, i.e., for arbitrary WS-BPEL processes nearly the same fragments can be injected. However, some fragments contain volatile elements that are specific to every single WS-BPEL process. As these fragments need to be adapted to fit a specific WS-BPEL process, we propose a two-stage transformation process. Figure 5.11 depicts an overview of the document artifacts involved in the transformation process, as well as the flow relations between them. The leftmost part of the figure indicates how the original WS-BPEL process definition file and various XML fragment files serve as input for the *Template Generator* XSLT file. This Template Generator constitutes the first transformation step and turns the generic fragment templates into fragments tailored to the target process definition. The last transformation step injects the generated fragments into the original WS-BPEL process file.

5.6.4 Checking Business Activity Constraints

The process transformation approach presented in Section 5.4 ensures runtime enforcement of Business Activity entailment constraints. For highly business- or security-critical systems we propose log analysis to additionally monitor that the process instances behave as expected (see, e.g., [129]). To check whether all constraints are fulfilled in the log data, we require an engine capable of querying the state of historical invocation data. As our framework is operating in a Web Services environment, XML is the prevalent data format and we focus mainly on XML tooling. We hence utilize XQuery [361] to continuously perform queries over the invocation logs stored in XML. To facilitate the handling of these queries, we use WS-Aggregation [139], a platform for event-based distributed aggregation of XML data.

```
1 <log taskName="Get_Personal_Data"subject="john"
2       role="staff" instanceID="i1" time="1316423654600"/>
3 <log taskName="Assign_Physician" subject="john"
4       role="staff" instanceID="i1" time="..."/>
5 <log taskName="Get_Personal_Data" subject="john"
6       role="staff" instanceID="i2" time="..."/>
7 <log taskName="Get_Critical_History" subject="bob"
8       role="physician" instanceID="i1" time="..."/>
9 ...
```

Listing 5.3: Format of Invocation Data Logged as Events

Listing 5.3 prints exemplary log data that are emitted by the transformed business process and handled by WS-Aggregation. Each *log* element in the listing represents one invocation event. The detailed constraint enforcement queries, expressed as XQuery assertion statements, are printed and discussed in Appendix A.2.

5.7 Evaluation and Discussion

In this section, we evaluate various aspects to highlight the benefits, strengths, and weaknesses of the presented solution. Five representative business processes with entailment constraints were selected to conduct the evaluation, including our example process from Section 5.2 and four additional processes from existing literature. The examples represent typical processes from different domains and cover all constraint types supported by our approach. The key properties of the evaluated processes are summarized in Table 5.3: *ID* identifies the process ($P1$ is our sample process), $|T_T|$ is the total number of task types per process, $|CT_T|$ is the number of task types associated with entailment constraints[5], $|R|$ is the number of roles defined in the scenario, $|S|$ is the number of subjects used for the test, and $|HR|$ is the number of senior-junior relationships in the role hierarchy[6].

Although not all results of our evaluation are fully generalizable, they are arguably valid for a wide range of scenarios and SOA environments in general. An evident observation is that runtime enforcement of security constraints is computationally intensive, and therefore performance

[5] $CT_T = \{\, t \in T_T \mid sb(t) \neq \emptyset \vee rb(t) \neq \emptyset \vee sme(t) \neq \emptyset \vee dme(t) \neq \emptyset \,\}$

[6] $HR = \{\, (s, j) \in R \times R \mid j \in rh(s) \,\}$

ID	Name	$\lvert\mathbf{T_T}\rvert$	$\lvert\mathbf{CT_T}\rvert$	$\lvert\mathbf{R}\rvert$	$\lvert\mathbf{S}\rvert$	$\lvert\mathbf{HR}\rvert$
P1	Patient Examination	7	6	3	4	1
P2	Purchase Order [75]	6	4	2	3	1
P3	Paper Review [316]	5	4	3	5	0
P4	Tax Refund [33]	5	4	2	5	0
P5	Credit Application [316]	5	3	2	4	1

Table 5.3: Characteristics of Business Processes Used in the Evaluation

effects need to be taken into account. We also show that the proposed DSL greatly simplifies development of security-enabled WS-BPEL processes, which becomes apparent when comparing the number of code artifacts before and after automatic transformation. However, the approach also has certain limitations which we also want to document explicitly. Overall, our evaluation is organized in four parts: first, we evaluate the runtime performance in Section 5.7.1; second, in Section 5.7.2 we verify the behavior of secured processes when provided with valid and invalid authentication data[7]; third, Section 5.7.3 evaluates the WS-BPEL transformation procedure; fourth, in Section 5.7.4 we discuss current limitations in the framework and general threats to validity. The experiments in Sections 5.7.1, 5.7.2 and 5.7.3 were executed on a machine with Quad Core 2.8GHz CPU, 8GB RAM, running Ubuntu Linux 9.10 (kernel 2.6.31-23).

5.7.1 Performance and Scalability

To evaluate the scalability we have deployed and executed different process instantiations (based on the scenario in Section 5.2) in a Glassfish server (version 2.1.1) with WS-BPEL engine (version 2.6.0). Here, we are interested in the net processing time of the Web service invocations, the duration of human tasks is not considered. Therefore, the execution of business operations (e.g., *Obtain X-Ray Image* or *Decide On Treatment*) has zero processing time in our testbed.

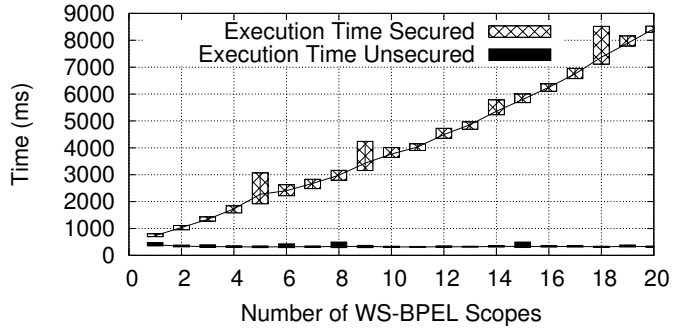


Figure 5.12: Process Execution Times – Secured vs Unsecured

[7]Note that all processes from Table 5.3 where implemented and evaluated with the same rigor. However, we do believe that certain parts of our evaluation are best explained in detail based on a single process. Therefore, Sections 5.7.1 and 5.7.2 exemplarily discuss the results from the patient examination example. This discussion applies analogously to the other processes from Table 5.3. The aggregated results for all processes are discussed in Section 5.7.2.3.

The WS-BPEL process has been deployed in different sizes (multiple `scopes`, one `invoke` task per scope), once with enforced security (i.e., annotated with security attributes, automatically transformed at deployment time), and once in an unsecured version. The deployed processes were executed 100 times and we have computed the average value to reduce the influence of external effects. Figure 5.12 plots the execution time (minimum, maximum, average) for both the secured (top line) and the unsecured version (bottom line). The top/bottom of each box represents the maximum/minimum, respectively, and a trendline is drawn for the average value[8]. We observe that a single BusinessAction invocation in the unsecured version is very fast, whereas the secured version incurs a considerable overhead. The overhead is hardly surprising considering that for each business logic service the process needs to invoke the IdP and RBAC services, as well as apply and check XML signatures. However, the measured results indicate that the current implementation has potential room for additional optimization.

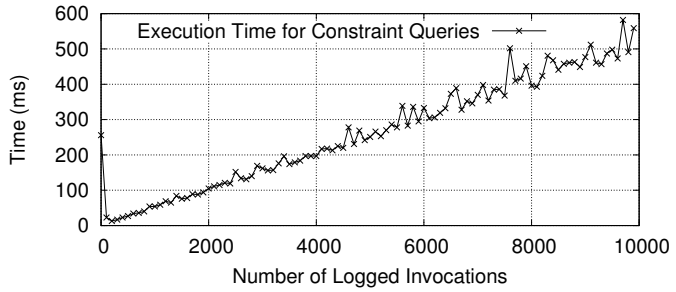


Figure 5.13: Execution Time of Constraint Queries for Increasing Log Data

In addition to the end-to-end performance of the secured WS-BPEL process, we also evaluated the performance of enforcing the BusinessActivity constraints using the XQuery based querying approach. To that end, we stored 10000 entries with SME, DME, SBind and RBind constraints to the invocation log and measured the time required to execute the four constraint queries in Listing A.2 (see Appendix A.2). The results are illustrated in Figure 5.13, which plots the time for every 100th invocation over time. As the testbed started cleanly from scratch, the first logged invocation(s) took longer ($\sim$250ms) because of internal initialization tasks in the log store and the WS-Aggregation query engine. Starting from the second data point (invocation 100), we see the query time increasing by around 6ms per 100 queries. To provide an insight about resource consumption, the CPU utilization and Java heap space usage are plotted in Figure 5.14. The slight fluctuations in heap space are due to Java's garbage collection procedure. The four constraint queries are executed in parallel, but since they access a shared data structure with log data, internal thread synchronization is applied. Hence, CPU utilization reaches only a peak value of $\sim$70% (i.e., 3 of the 4 cores).

The increase of time is inherent to the problem of querying growing log data. We argue that query performance is feasible for medium-sized to even large scenarios. Firstly, as evidenced in

[8]The standard deviation was in the range of 39.21 to 413.69 ms (lowest and highest values are for processes with 1 scope and 18 scopes, respectively) for the secured version, and in the range of 10.38 to 58.78 ms (for 13 scopes and 8 scopes, respectively) for the unsecured version.

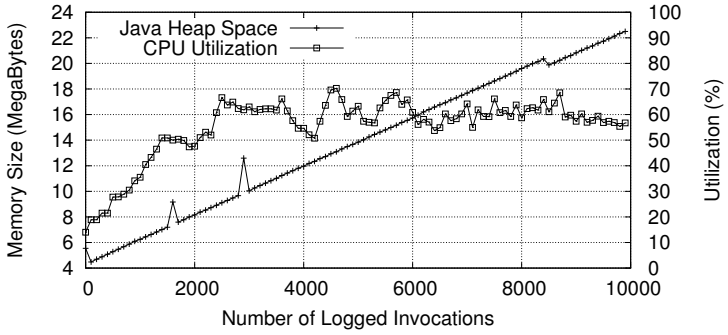


Figure 5.14: Resource Consumption for Constraint Queries

Figure 5.13, the execution time appears to grow only linearly (we have also performed a linear regression which showed almost perfect fit for $y = 20 + 0.06x$). The reason is that the queries are formulated in a way that always only the last added log entry needs to be compared to the other entries (hence, the queries are executed for each new log entry). Secondly, even for large logs (tens of thousands of entries) the execution time is still in a range of only a few seconds. If we extrapolate the test values for very huge logs (millions of entries), however, the current approach would take in the order of minutes, which may not be feasible for real-time processes. Hence, additional optimizations will be required for such very-large scale situations – a problem we actively tackle in our future work.

5.7.2 Reaction of the Secured Process to Valid and Invalid Authentication Data

In the second experiment, we utilize the five evaluation processes (see Section 5.7) to evaluate how our approach deals with authentication data of authorized and unauthorized users provided by the IdP service. As outlined in Section 5.4, the task of the IdP is solely to authenticate users, but the authorization in terms of RBAC constraints is enforced by the process instance (and, additionally, by the log data queries from Section 5.6.4). Hence, the reason for performing this experiment is to test the ability of the transformed business process to cope with unauthorized users who attempt to execute restricted process tasks. Moreover, we are interested in evaluating under which circumstances the RBAC rules may become overconstrained such that the process ends in a deadlock and is unable to continue. Our methodology in this experiment is to execute all possible instances of the test processes with respect to user authorization (given a set of subjects and process tasks, try each combination of subjects performing a task; see Section 5.7.2.1 for details). The chosen scenario processes have a feasible size to perform this full enumeration. We discuss detailed results based on the patient examination process (P1) in Section 5.7.2.2, and aggregated results over all five processes (P1-P5) in Section 5.7.2.3.

5.7.2.1 Permutation of RBAC Assignments

We define the domain $[T_T \rightarrow (S \times R)]$ of RBAC assignment functions, where T_T is the set of BusinessAction task types, S is the set of subjects and R is the set of roles (cf. Section 5.3.1).

The function defines which authentication data should be used for each task type. We then consider all possible permutations of function assignments in this domain, with the restriction that for each pair $(s, r) \in S \times R$ the subject s is directly associated with role r. To keep the domain small, inherited roles are not considered. For instance, in our scenario the pair (*Bob*,*Physician*) is included, but (*Bob*,*Staff*) is not considered, although *Bob* inherits the role *Staff* through *Physician*. Furthermore, note that SME constraints are checked at design-time when defining a process-related RBAC model. The static correctness rules ensure the consistency of the corresponding RBAC models at any time (see [316]). Hence, it is not possible to define an inconsistent RBAC model where, for example, a subject or role possesses the right to execute two SME tasks. The respective RBAC model is then applied to make access decisions and to perform task allocations for all process instances. In other words, because the allocation of task instances is based on a consistent process-related RBAC model, it is not necessary to check the fulfillment of SME constraints again at runtime (see also [315]).

For each permutation one process instance is executed, and the IdP in the test environment is configured to return the authentication data that correspond to the respective permutation. The IdP keeps track of *getAuthenticationData* requests and registers the number of duplicate requests in each process instance. Recall that a duplicate request is always issued if the IdP provides authentication data of a non-authorized user. Thus, each duplicate *getAuthenticationData* request represents a blocked execution of a restricted task (which is the desired/expected behavior).

The purpose of this experiment setup is to empirically evaluate 1) whether the secured process correctly allows/denies access for valid/invalid provided credentials, respectively, and 2) how the platform deals with unresolvable conflicts (if the process deadlocks due to mutual exclusions). For instance, when *Get Personal Data* in our scenario is invoked with (*Bob*,*Physician*) and the IdP provides (*John*,*Staff*) for *Assign Physician*, then re-authentication is necessary because of role-binding violation. In this case, the IdP simply provides the next available authentication data, simulating the real-life situation that a new subject logs in after an unauthorized subject has been denied access. This procedure is repeated as long as new (subject,role) combinations can be provided; if the process has unsuccessfully attempted to invoke a task with *all* possible combinations, the entire process terminates with a fault message. Note that this method of deadlock detection is suitable for our scenario with only a small number of subjects; for more advanced detection of deadlocks and unsatisfiable constraints we refer to related work [76, 349].

5.7.2.2 Detailed Discussion for the Patient Examination Process

In our scenario, the domain $(S \times R)$ consists of the four pairs ((*John,Staff*), (*Jane,Physician*), (*Bob,Physician*), (*Alice,Patient*)), and six task types exist ($|T_T| = 6$). Hence, the total number of possible assignment function permutations is 4^6 =4096. However, the process structure allows to reduce this number because the decision node (whether it is in an emergency situation) splits the process into two possible execution paths (one path with 5 tasks and the other path with 4 tasks). The decision node is simulated to uniformly use both of the two possible conditional branches. Therefore, in total only $4^5 + 4^4$ =1280 process instances have to be executed.

Figure 5.15 illustrates the number of blocked authorization requests for each process instance. Considering the procedure of security enforcement (cf. Section 5.4), a blocked request means that the authentication data provided by the IdP violate any constraints (which is expected

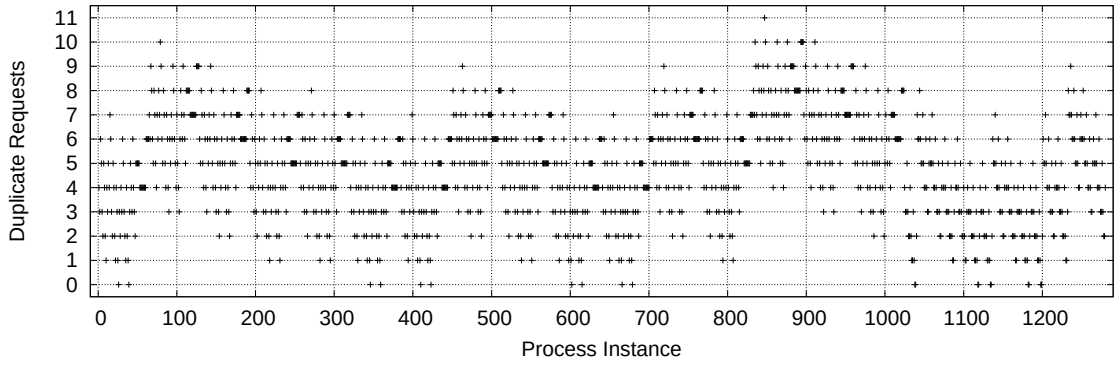


Figure 5.15: Blocked Task Executions per Test Process Instance (Patient Examination Scenario)

in many cases, since all permutations are tested). Table 5.4 summarizes the aggregated values: 20 of the 1280 generated RBAC assignments were completely valid from the start and no blocked requests were necessary. The remaining instances required between 1 and 11 blocked requests until a final state (successful or unsuccessful) is reached.

Result Outcome	**Instances**	**Result Outcome**	**Instances**
No Blocked Requests	20	7 Blocked Requests	140
1 Blocked Request	56	8 Blocked Requests	80
2 Blocked Requests	108	9 Blocked Requests	32
3 Blocked Requests	163	10 Blocked Requests	10
4 Blocked Requests	228	11 Blocked Requests	1
5 Blocked Requests	232	Successful Execution	1024
6 Blocked Requests	210	Failed (Deadlocked)	256
		Total Instances	1280

Table 5.4: Process Executions with Permutations of $T_T \rightarrow (S \times R)$ Assignments

While there have been 1024 successful executions of the process, 256 failed instances had to be aborted because of deadlock situations. Deadlocks can result from the complex inter-dependencies of BusinessActivity access rules (see, e.g., [293, 320]). For instance, consider the operation sequence in Table 5.5. The deadlock is caused by the subject-binding between *Get Critical History* and *Decide On Treatment*, combined with the fact that both tasks can be executed by different roles (the former by *Patient* and *Physician*, and the latter only by *Patient*). In fact, all process executions in which the patient *Alice* executes *Get Critical History* lead to this conflicting situation. Note that the focus of our work is to enforce RBAC constraints and to *detect* deadlocks[9]. In our future work we also investigate techniques to check the satisfiability of a certain process and *avoid* deadlocks in advance (see, e.g., [76, 292, 320, 349]).

[9]Note that the deadlocks in our evaluation result from the fact that we automatically generate and execute all possible process instances (see Section 5.7.2). Because our process-related RBAC models adhere to the static and dynamic consistency requirements defined in [315, 316] the resulting RBAC models are always consistent. However, even though we always have consistent models, it is still possible that a certain process is not satisfiable [76, 349].

Task	Sub.	Role	Effect
Get Personal Data	*John*	*Staff*	Role *Staff* must *Assign Physician*; *John* must *Assign Physician*
Assign Physician	*John*	*Staff*	-
Obtain X-Ray Image	*Bob*	*Physician*	-
Get Critical History	*Alice*	*Patient*	*Alice* must not *Get Expert Opinion*; *Alice* must *Decide On Treatment*
Get Expert Opinion	*Jane*	*Physician*	-
Decide On Treatment	*?*	*?*	Deadlock, because the bound subject *Alice* is not permitted

Table 5.5: Operation Sequence Leading to a Constraint Conflict (Deadlock)

The same experiment setup has been used to measure the execution time of the secured process instances over time (Figure 5.16). Again, we see a slight upwards trend in the processing time. The reasons for this trend are twofold. First, the more instances have executed, the more log data must be checked for constraint conflicts. Second, particularly for SME constraints an increasing number of log data increases the likelihood that the blocked requests need to be issued because the provided test authentication data are in a conflict with one or more previous invocations. The spikes in Figure 5.16 indicate different execution times of instances with few versus many blocked requests (see also Figure 5.15). Notice that the execution time shows a certain pattern between roughly 0 and 1000, and a different pattern between 1000 and 1280. These patterns are a direct result of the experiment design, because we first execute 1024 instances that follow the "emergency" path in the scenario process, and afterwards 256 instances that follow the "non-emergency" path.

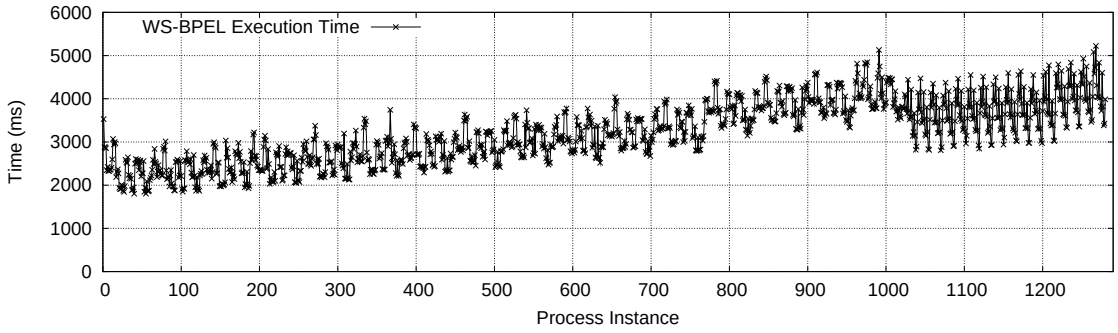


Figure 5.16: Execution Time of Secured BPEL Process Instances Over Time

5.7.2.3 Aggregated Results for All Test Processes

Table 5.6 summarizes the test results for the five test processes. The table contains the process *ID* that refers back to Table 5.3, the total number of executed instances which were generated from the RBAC assignment permutations, the number of deadlocks that occurred, the blocked requests (minimum/maximum/average) per process instance, and the aggregated execution time per instance. In general, the number of instances corresponds to $|S|^{|T_T|}$, except in cases where we can take advantage of the process structure to reduce the number of instances (i.e., 1280 instead of 4096 instances for P1). Process P4 has the highest number of instances (3125). The

ID	Inst-	Dead-	Blocked Requests			Execution Time (ms)		
	ances	locks	min	avg	max	min	avg	max
P1	1280	256	0.0	4.8	11.0	1802.0	3199.6	5222.0
P2	729	243	0.0	3.3	7.0	3990.0	5009.0	8881.0
P3	625	0	0.0	3.6	8.0	3444.0	5464.8	8057.0
P4	3125	0	0.0	6.9	16.0	2984.0	8356.6	14363.0
P5	64	0	0.0	1.8	4.0	2799.0	3070.1	5530.0

Table 5.6: Aggregated Test Execution Results of the Five Evaluated Processes

aggregated values are computed over all process instances; for example, the average number of blocked requests over all 1280 instances of process P1 is 4.8. The difference between minimum and maximum execution time depends on the executed tasks, and hence correlates strongly with the number of blocked requests. The maxium execution time was roughly 14 seconds (for an instance of process P4), and the shortest instance (of P1) executed within less than 2 seconds. Depending on the process definition and the chosen subjects, either all generated process instances were able to execute successfully (P3, P4, P5), or some instances deadlocked (P1, P2). Some process definitions are prone to deadlocking (e.g., 20% of P1's possible instances lead to a deadlock), whereas in other processes deadlocks are not even possible. For instance, the tax refund process [33] (P4) was run with the smallest possible number of subjects (at least 2 clerks and 3 managers are required), but out of the 3125 instances (each subject tries to access each of the five task types, $5^5 = 3125$) not a single instance deadlocks. Even though satisfiability of access constraints at different points of the process execution can be determined algorithmically (see, e.g., [320]), we argue that it is equally important to test the running system, and to empirically verify the number of successful and blocked requests, as shown in this evaluation.

5.7.3 WS-BPEL Transformation Algorithm

Concerning the evaluation of the WS-BPEL transformation algorithm, we consider the same twenty test process definitions with different sizes described earlier in Section 5.7.1.

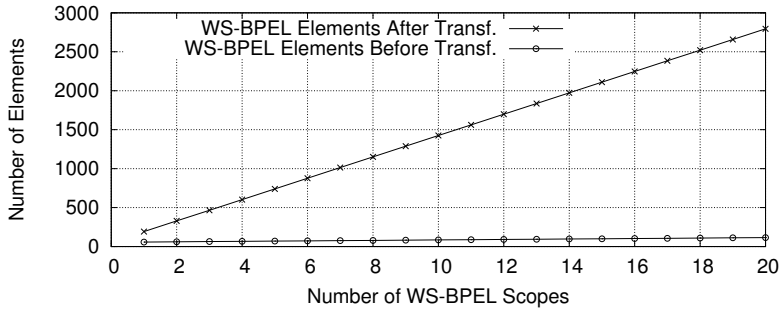


Figure 5.17: Different Sizes of WS-BPEL Processes Before and After Transformation

Figure 5.17 shows the number of WS-BPEL elements of the process definition before and after the automatic transformation. The results indicate that the size of the WS-BPEL def-

inition rises with increasing number of scopes. While our test process with a single scope contains 33/115 WS-BPEL elements before/after transformation, the process definition for 10 scopes grows to 60/484 WS-BPEL elements before/after transformation, respectively. These numbers are determined by counting all XML (sub-)elements in the WS-BPEL file using the XPath expression `count(//*)`. At the beginning of the transformation, 41 elements are added (`import`, `partnerLink` and `variable` declarations), and for each new scope 41 elements are added for the IAM task definitions (note that both values are 41 coincidentally). We observe that the ability to define security annotations in WS-BPEL keeps the process definition clear at design time. In fact, the additional code for security enforcement in WS-BPEL is often larger than the actual business logic. This can be seen as an indicator that our approach can reduce the development effort as compared to manual implementation, although we did not empirically evaluate this aspect in detail.

5.7.4 Discussion of Limitations

In this section, we discuss the current limitations and weaknesses of our approach and the corresponding Web service technology projection. We also propose possible mechanisms and future work to mitigate the consequences and risks associated with these limitations.

* Parallel Process Flows: WS-BPEL provides the `flow` command for concurrent execution of tasks. Security enforcement of tasks that execute in parallel poses a challenge for various reasons. Firstly, if two tasks are started with mutually exclusive access rights, a race condition is created with respect to the first task to access the authentication token. Secondly, since we make use of "global" (process-instance-wide) variables, the injected IAM tasks for each single WS-BPEL `invoke` action are supposed to execute atomically and should not access these variables concurrently. To handle parallel execution, we hence propose to extend the injected IAM tasks with two additional tasks to acquire and release an exclusive lock when entering and leaving the critical region, respectively. Since WS-BPEL does not provide a corresponding language construct, an external Web service is used to acquire the exclusive lock on a semaphore. For brevity and clarity, these additional synchronization tasks have not been added to the transformation in Section 5.4. In future work, we further plan to introduce more sophisticated synchronization using the WS-BPEL `link` mechanism.

* Deadlocking: If the RBAC policies are conflicting, the procedure for obtaining and checking user authentication data can end up in a deadlock that is unable to terminate with a successful result. To mitigate the effect of policy conflicts, it is therefore required to perform timely satisfiability checks. In Section 5.8 we discuss related work that focuses on this topic, in particular we refer to previous work in [292, 293, 315, 320].

* Single Point of Failure: Our Web service technology projection builds on the assumption that the IdP and Logging services operate reliably and continuously. An outage of any of these services would imply that the access control procedure cannot be performed in its entirety or that certain log data cannot be stored. Depending on the process definition at hand, the consequences can be more or less severe. The IdP service is the key

component that provides the basis for user authentication. If it is unavailable, the secured execution fails. A possible strategy for certain application scenarios would be to define *break-the-glass* (BTG) rules (see, e.g., [102, 215, 294]) which allow to temporarily access the protected resources with fallback security settings, in order to provide for continuous operation. An outage of the Logging service is less severe, because it is strictly only required to perform a posteriori conformance checks of global constraints that may affect all (or at least multiple) process instances (see, e.g., [28]). Instance-specific constraints are local to a certain process instance and can be enforced by means of instance-specific log data stored in WS-BPEL variables (see Section 5.5).

* Security Token Hijacking: Malicious users may attempt to gain access to services they are not entitled to. Consider an attacker who intentionally does not follow the processing logic of the transformed process but invokes the target Web services directly. The attacker may obtain a SAML token by executing *getAuthenticationData*, which asserts its subject and role. Assume that the token is used in combination with the *instanceID* of an active process instance to invoke the *Decide On Treatment*; this situation must be avoided under any circumstances. To enforce the subject-binding with *Get Critical History* and other RBAC rules it is imperative that all access constraints are validated on the service side. In our architecture we hence require a PEP which intercepts and analyzes all invocations.

* Invalid WS-BPEL Modification: For the approach to work reliably, it is important that the WS-BPEL definition should not be modified after the automated code transformation step. We therefore propose the use of a trusted deployment component which provides exclusive access to the business process execution engine. As part of transformation process the WS-BPEL file is signed with an XML signature [368], which is then checked by the deployment component to enforce integrity.

* Human Factors: In the end, a business process involving human labor can only be as safe and reliable as the persons who perform it. That is, control mechanisms such as mutual exclusion (e.g. to enforce the four-eyes principle) can provide a strong instrument for improving quality and reliability, but human errors can never be fully ruled out.

5.8 Related Work

This section provides a discussion of related work in the area of model-driven IAM and their application to SOA business processes. Our analysis focuses on three main research areas: security modeling for Web service based systems, DSL-based security modeling, and techniques for incorporating runtime enforcement of security constraints into business processes.

5.8.1 Security Modeling for Web Service Based Systems

Jensen and Feja [166] discuss security modeling of Web service based business processes, focusing on access control, confidentiality and integrity. Their approach is based on Event-driven Process Chains (EPC) [291] and defines different security symbols that the process definitions

are annotated with. Their implementation is integrated into the ARIS SOA Architect software, which is also able to transform the EPC model into an executable SOA business process. The paper describes the generation of WS-SecurityPolicy [247] policies, but does not discuss mutual exclusion and binding constraints in process-related RBAC models, nor does it discuss in detail how the process engine enforces the policies and constraints at runtime, which in contrast is a core part in our work.

Kulkarni et al. [179] describe an application of context-aware RBAC to pervasive computing systems. As the paper rightly states, model-level support for revocation of roles and permissions is required to deal with changing context information. The approach has a strong focus on dynamically changing context (e.g., conditions measured by sensors) and the associated permission (de-)activation. In our framework, context information is part of the RBAC model definitions (more details can be found in [134]). In this thesis, the context information in the RBAC model has been abstracted from, but as part of our future work we plan to integrate the Business Activity model in [316] with context information (see also [317]).

Although our model does not directly build on the notion of trust, access control policies can also be established dynamically by deriving trust relationships among system participants [86]. Skoksrud et al. present Trust-Serv [305], a solution for model-driven trust negotiation in Web service environments. Similar to our approach, the policy enforcement is transparent to the involved Web services. Another similarity is that trust credentials (such as user identifier, address or credit card number) are exchanged iteratively throughout the process, which is also the case for the authentication credentials in our approach. However, trust-based policies in [305] are *monotonic* in the sense that additional trust credentials always add access rights and never remove existing ones, which is in contrast to access control in this thesis, where the execution of tasks can activate entailment constraints which progressively narrow down the set of valid access control configurations.

Our approach was also influenced by Foster et al. [105] who present an integrated workbench for model-based engineering of service compositions. Their approach supports service and business process developers by applying formal semantics to service behavior and configuration descriptors, which can then be analyzed and checked by a verification and validation component. The policies enforced by the workbench are quite generally applicable and hence require developers to perform application specific modeling, whereas our proposed DSL and WS-BPEL annotations are tailored to the domain of RBAC and entailment constraints and arguably straight-forward to apply.

Seminal contributions in the context of modeling support for Web service based business processes are provided within the Web Services Modeling Framework (WSMF) by Fensel et al. [100], and the modeling ontologies that emerged from this project. For instance, security requirements can be modeled in WSMF by declaring the subject and role as input data and defining pre-conditions for all operations that require certain authentication data. In the previous years, the Semantic Web community has been pushing forward various ontologies to draw an ever more exact picture of the functionality exposed by Web services, in order to allow for sophisticated discovery, execution, composition and interoperation [221]. In fact, although not very frequently used in practice, semantically annotated Web services also allow for a more fine-grained definition of access control policies, from the interaction level down to the message

level. Whereas annotations in semantic Web services are used mostly for reasoning purposes, the WS-BPEL annotations used in our approach are utilized as metadata for runtime access control enforcement. Such business process model abstractions, which are the underpinning of semantic equivalence and structural difference, have been empirically studied in [306], and our approach can be seen as the reverse operation of abstraction (i.e., concretization) for the specific application domain of task-based entailment constraints.

Various other papers have been published that are related to our work or have influenced it, some of which are mentioned in the following. The platform-independent framework for Security Services named SECTISSIMO has been proposed by Memon at al. [225]. The conceptual novelty of this framework is the three-layered architecture which introduces an additional layer of abstraction between the models and the concrete implementation technologies. In contrast, our prototype only considers two layers (i.e. modeling of RBAC constraints and transformation of WS-BPEL code). However, the presented modeling concepts (see Section 5.3) as well as the model transformations (see Section 5.4) are independent from concrete implementation technologies too.

Lin et al. [198] propose a policy decomposition approach. The main idea is to decompose a global policy and distribute it to each collaborating party. This ensures autonomy and confidentiality of each party. Their work is particularly of relevance for cross-organizational definition of RBAC policies, as performed in our multi-hospital use case scenario. Currently, our prototypical implementation relies on a single, global RBAC Web service. However, we plan to adopt this complementary policy decomposition approach, which will allow each hospital to employ its own dedicated RBAC Web service.

5.8.2 DSL-Based Security Modeling

An integrated approach for Model Driven Security, that promotes the use of Model Driven Architectures in the context of access control, is presented by Basin et al. [27]. The foundation is a generic schema that allows creation of DSLs for modeling of access control requirements. The domain expert then defines models of security requirements using these languages. With the help of generators these models are then transformed to access control infrastructures. However, compared to our approach, [27] does not address the definition of task-based entailment constraints.

The approach by Wolter et al. [356] is concerned with modeling and enforcing security goals in the context of SOA business processes. Similar to our approach, their work suggests that business process experts should collaboratively work on the security policies. They define platform independent models (PIM) which are mapped to platform specific models (PSM). At the PIM level, eXtensible Access Control Markup Language (XACML) and *AXIS 2*[10] security configurations are generated. Whereas their approach attempts to cover diverse security goals including integrity, availability and audit, we focus on entailment constraints in service-based business processes.

A related access control framework for WS-BPEL is presented by Paci et al. in [257]. It introduces the *RBAC-WS-BPEL* model and the authorization constraint language *BPCL*. Similar

[10] `http://axis.apache.org/axis2/java/core/`

to our approach, the WS-BPEL activities are associated with required permissions (in particular, we associate permissions for `invoke` activities that try to call certain service operations). However, one main difference is related to the boundaries of the validity of user permissions: RBAC-WS-BPEL considers pairs of adjacent activities (a_1 and a_2, where a_1 has a control flow link to a_2) and defines rules among them, including separation of duty (a_1 and a_2 must execute under different roles) and binding of duty (a_1 and a_2 require the same role or user). As elaborated in previous work [134], our approach also allows to annotate scopes (groups of `invoke` tasks) in BPEL processes and hence to apply RBAC policies in a sequential, but also in a hierarchical manner.

XACML [243] is an XML-based standard to describe RBAC policies in a flexible and extensible way. Our DSL could be classified as a high-level abstraction that implements a subset of XACML's feature set. Using a transformation of DSL code to XACML markup, it becomes possible to integrate our approach with the well-established XACML environment and tools for policy integration (e.g., [219]).

5.8.3 Runtime Enforcement of Constraints in Business Processes

Various approaches have been proposed to incorporate extensions and cross-cutting concerns such as security features into business process models. Most notably, we can distinguish different variants of model transformation [78, 300] and approaches that use aspect-oriented programming [172].

A dynamic approach for enforcement of Web services Security is presented in [235] by Mourad et al. The novelty of the approach is mainly grounded by the use of AOP in this context, whereby security enforcement activities are specified as *aspects* that are dynamically woven into WS-BPEL processes at certain *join points*. Charfi and Mezini presented the AO4BPEL [61] framework, an aspect-oriented extension to WS-BPEL that allows to attach cross-cutting concerns. The aspect-oriented language Aspects for Access Control (AAC) by Braga [41] is based on the same principle and is capable of transforming SecureUML [203] models into aspects. A main difference is that AAC does not operate on WS-BPEL, but on Java programs, and can hence be applied directly to Java Web service implementations to enforce access control.

Essentially, our approach can be regarded as a variant of AOP: the weaved aspects are the injected IAM tasks, and join points are defined by security annotations in the process. A major advantage of our approach is the built-in support for SSO and cross-organizational IAM. An interesting extension could be to decouple security annotations from the WS-BPEL definition, to store them in a separate repository and to dynamically adapt to changes at runtime.

A plethora of work has been published on transformations and structural mappings of business process models. Most notably, our solution builds on work by Saquid/Orlowska [283], and Eder/Gruber [93] who presented a meta model for block structured workflow models that is capable of capturing atomic transformation actions. These transformation building blocks are important for more complex transformations, as in our case when multiple process fragments for enforcement of entailment constraints are combined for a single action in WS-BPEL. While this work focuses mainly on deployment time model transformations, other research also investigates runtime changes of service compositions. For instance, automatic process instrumentation and runtime transformation have previously been applied in the context of functional testing [143] of

service-based business processes. Weber et al. [350] investigate security issues in adaptive process management systems and claim that such dynamicity increases the vulnerability to misuse. Our approach is adaptive in that it allows the "environment" (e.g., access policies) to change at runtime. However, we currently assume that the process definition itself does not change. In our ongoing research, we are complementing our approach with support for online structural process adaptation.

An important aspect of security enforcement is the way how constraint conflicts are handled at runtime. Consequently, our approach is related to a recent study on handling conflicts of binding and mutual exclusion constraints in business processes [292, 293]. Based on a formalization of process-related RBAC, this work proposes algorithms to detect conflicts in constraint definitions, as well as strategies to resolve the conflicts that have been detected. In our evaluation (see Section 5.7), we illustrated an example constraint conflict that lead to a deadlock and discussed how the platform is able to detect such conflicts. In order to anticipate and avoid deadlocks altogether, we will eventually integrate these algorithms with our RBAC DSL.

Although not necessarily concerned with security (i.e., access control) in the narrower sense, the area of Web service transaction processing [289, 344] and conversational service protocols [31, 138] is related to our work on secured business processes. Put simply, a transactional protocol is a sequence of operations with multiple participants that have a clearly defined role and need to collaboratively perform a certain task. Analogously, BusinessActivities are performed by subjects with clearly defined roles and limited permissions. One could argue that while the responsibility of transaction control is to ensure that all participants actually *do* perform their task, the main purpose of access control is to ensure that subjects *do not* perform tasks they are not authorized to. Amongst others, our approach was influenced by von Riegen et al. [344] who model distributed Web service transactions with particular focus on complex interactions where participants are restricted to only possess limited local views on the overall process. These limited views are comparable to our access control enforcement. Our approach also detects if a process instance is about to break the required conversational protocol (i.e., access control policies), in which case we apply a sequence of compensation actions [289] (e.g., repeat authentication or terminate instance due to deadlock).

5.9 Conclusions

Enforcement of security and access constraints is a key concern for reliable application provisioning, particularly in Cloud environments where multiple tenants and users collectively utilize shared infrastructure and platform services. For any mission- or safety-critical application it is imperative that restricted resources be protected from unauthorized access. While the theoretical foundations of security mechanisms like RBAC or task-based entailment constraints are well-understood, development support for runtime enforcement has so far received less attention. In lack of a systematic development approach, the business logic code may get mixed up with tailor-made security enforcement procedures, resulting in often recurring manually written boilerplate code that becomes hard to maintain and validate. To overcome this situation, developers should be provided with suitable abstractions and convenient tools for reliable enforcement of access constraints.

In this chapter, we have tackled these challenges and presented an integrated, model-driven approach for the enforcement of access control policies and task-based entailment constraints in distributed service-based business processes. The approach is centered around the DSL-driven development of RBAC policies and the runtime enforcement of the resulting policies and constraints in Web services based business processes. Our work fosters cross-organizational authentication and authorization in service-based applications, and facilitates the systematic development of secured business environments. From the modeling perspective, the solution builds on the BusinessActivity extension – a native UML extension for defining entailment constraints in activity diagrams. We provided a detailed description of the procedure to transform design-time BusinessActivity models into standard activity models that enforce the access constraints at runtime. Based on a generic transformation procedure, we discussed our implementation which is based on WS-BPEL and the Web services framework.

Our approach based on BusinessActivities allows to abstract from the technical implementation of security enforcement in the design time view of process models. The detailed evaluation of the process transformation has shown that process definitions with injected tasks for security enforcement grow considerably large. In fact, the additional code for security enforcement in WS-BPEL is often larger than the actual business logic. This can be seen as an indicator that our approach can reduce the development effort as compared to manual implementation, although we did not empirically evaluate this aspect in detail.

Our extensive performance evaluation has illustrated that the proposed runtime enforcement procedures operate with a slight overhead that scales well up to the order of several ten thousand logged invocations. We can conclude that the overhead consists of three main parts: 1) the approach builds on digital signatures for ensuring message integrity, 2) the process determines the role and permissions of the currently executing user, which results in additional requests and increased execution time, and 3) the enforcement of entailment constraints requires querying the log traces of previous executions of the process. Note that the overhead for 1) and 2) does not increase over time (with rising number of process executions), whereas the overhead for 3) inherently rises because the log traces are accumulating over time, and more data have to be evaluated.

The implementation of our prototype still has limitations, and we discussed strategies to improve some of these limitations in future work. For instance, advanced synchronization mechanisms are required for business processes with highly parallel processing logic. Moreover, the query mechanism that checks security constraints for validity needs to be further optimized for very large log data sets (in the order of millions of invocations). We envision advanced data storage and compression techniques, as well as optimized query mechanisms to further reduce this increase of overhead over time. In our ongoing work we also investigate the use of additional security annotations and an extended view of context information. Finally, we plan to shift from a process-centric to a more data-centric view and seek for a stronger integration of entailment constraints into our ongoing work on reliability in event-based data processing (see Section 3.2) and collaborative Web applications [113–115].

CHAPTER 6

Conclusions

This final chapter summarizes the main results achieved within this thesis. Section 6.1 provides a summary of the contributions with a focus on how the work has advanced the scientific state of the art. In Section 6.2, the research questions formulated in Section 1.1.2 are revisited and put into perspective with the provided contributions. Finally, Section 6.3 concludes the thesis with a discussion of ongoing trends in related research areas, as well as open topics for future research which can build on the contributions achieved here.

6.1 Summary of Contributions

Throughout this thesis, we have elaborated novel techniques for reliable provisioning of data-centric and event-based applications in the Cloud. The thesis takes a holistic viewpoint and tackles the problem from different angles, ranging from reliable deployment on the infrastructure layer, to optimized processing and load distribution on the middleware platform layer, to functional testing and access control enforcement on the application level.

The discussion of the individual contributions is aligned along three distinct, yet strongly interconnected, frameworks that have been developed in the course of this thesis:

- First, in the context of the WS-Aggregation platform, a novel model and execution approach for reliable event-based data processing applications has been introduced. The platform is tightly integrated with Cloud computing techniques, focusing on dynamic resource allocation, elastic scaling, and adaptation to workload fluctuations. Our comprehensive survey and taxonomy of faults in event-based systems provides guidance for developers of fault tolerant event processing platforms. Motivated by the fact that resource elasticity depends on reliable infrastructure provisioning (e.g., deployment and configuration of new VMs), a novel methodology for testing idempotence and convergence of Infrastrucure-as-Code (IaC) automation scripts has been developed. The comprehensive evaluation based on roughly 300 publicly available IaC automations (Chef scripts) reveals that a large number ($> 30\%$) of the selected automations exhibit non-idempotent behavior, compromising the stability and repeatability of

deployments. Reliable infrastructure provisioning allows us to elastically scale and reconfigure the WS-Aggregation platform at runtime. A formally defined target function achieves a tradeoff along three dimensions: balanced load among nodes, low data transfer, and reduction of duplicate event buffering. Our work is the first to study the effect of optimizing these particular dimensions. Moreover, our heuristic optimization algorithm, based on VNS, also takes the cost for migrating to a new configuration into account.

- Second, the TeCoS framework introduces a novel approach for testing and fault localization in dynamic composite Service-Based Applications (SBAs). The testing goal is to instantiate SBAs in various configurations, systematically combining concrete service implementations, in order to identify incompatibilities and integration issues. Based on a generic model of SBAs, we define k-node data flows, a coverage metric aiming at limiting the size of generated test suites. The generic model is mapped and applied to two concrete technologies for SBAs, namely WS-Aggregation and WS-BPEL. We leverage techniques from combinatorial test design to generate near-minimal test suites, utilizing the *FoCuS* tool developed by IBM. The instantiations of the SBA, both during testing and in production mode, are monitored and stored as execution traces. The collected execution traces are analyzed to localize faults resulting from incompatible services. Our fault localization approach utilizes machine learning techniques, and is capable of dealing with transient faults and changing fault conditions.
- Third, the SeCoS framework provides identity and access management (IAM) for service-based business processes deployed in multi-tenant Cloud environments. We build on the well-established concept of RBAC to assign roles and permissions to subjects participating in the business process. Task-based entailment constraints allow for fine-grained assignment of responsibilities, in particular separation and binding of duties. We propose a declarative DSL for defining access constraints as annotations in the business process definition. At deployment time, systematic transformations are applied to convert the process definition into an executable format which enforces the constraints at runtime. The prototype implementation is integrated with the WS-BPEL framework and supports single sign-on (SSO) across multiple organizations. Our comprehensive evaluation has demonstrated that the approach correctly ensures access constraint consistency.

Throughout the entire thesis, we have striven to achieve a fair balance between systematic problem abstractions with formal underpinnings on the one hand, and concrete technology mappings with runnable prototype implementations on the other hand. The majority of the developed implementation code has been provided as open source tools to the community. Each contribution has been rigorously evaluated based on the prototype implementations and representative experimentation scenarios.

6.2 Research Questions Revisited

In Section 1.1.2, we have formulated three core research questions which have guided the research of this thesis. In the following, we briefly summarize how these questions have been addressed within our work, and which limitations remain in the current solution.

Q1: What are suitable methods and a supporting system to reliably execute event-based data processing applications in the Cloud, leveraging elasticity and dynamic resource allocation?

Chapter 3 has introduced WS-Aggregation, a self-adaptive platform for event-based data processing. WS-Aggregation is designed for reliability and elastic runtime adaptation, providing consistent QoS in the face of workload fluctuations and runtime faults (e.g., node outages). A comprehensive taxonomy for faults in event-based systems has been derived, which builds the foundation for reliable processing and fault tolerance mechanisms (e.g., redundant processing). The platform leverages dynamic resource allocation in the Cloud to provide runtime optimization and elastic scaling. Since elasticity requires reliable resource provisioning on the infrastructure level, a systematic testing method for IaC automations is employed, which is able to efficiently determine faulty automations and issues concerning idempotence and repeatability. The method for elastic reconfiguration is based on the M-A-P-E cycle from Autonomic Computing [171], with the recurring phases of *monitoring* (capturing the workload and data traffic between nodes), *analysis* (detecting whether the system is about to enter a critical state and requires reconfiguration), *planning* (finding a new configuration based on a well-defined optimization target), and *execution* (putting the plan into action, reconfiguring the system and migrating processing elements between nodes).

Some limitations and shortcomings of the presented approach still remain. Our solution currently only considers resource elasticity, that is, the number of computing resources is adapted based on external factors such as changing workloads. In addition, quality elasticity and cost elasticity have to be considered; the former takes an existing resource allocation and dynamically adjusts the QoS, e.g., varying the level of data consistency or precision of results; the latter means to dynamically adjust the budget limit devoted to operation of the platform, while tolerating variable QoS [91]. Moreover, the approach needs to be extended to systematically handle and detect all fault types of the taxonomy. To that end, the taxonomy needs to be encoded in a generic runtime model which is kept in sync with specific target platforms, reflecting all platform changes in the model, and enforcing all changes in the model to trigger reconfigurations in the platform. By means of such a synchronization mechanism between the model and real platforms, systematic fault detection and fault injection techniques can be implemented by generic manipulations on the model, in order to determine whether the platform gracefully handles different types of runtime faults.

Q2: Which testing and fault localization techniques can be applied to ensure reliable provisioning of data-centric and event-based applications in the Cloud?

Testing is an integral activity for ensuring reliability and detecting potential runtime problems up front. The testing technique discussed in Chapter 4 tackles reliable provisioning on the application level and has been shown to successfully detect integration issues with incompatible service implementations and data incompatibilities. The k-node data flow coverage metric is tailored to data-centric services and allows to keep the testing effort low by limiting the size of test suites. The approach is generic and applicable to different types of application frameworks, which is evidenced by the demonstrated mappings to WS-BPEL and WS-Aggregation. Since

systems are typically evolving and not all situations can be anticipated up front, the system execution is additionally monitored for runtime faults. The collected execution traces are utilized for fault localization based on machine learning techniques. The evaluation has shown that the approach successfully deals with transient faults as well as changing fault conditions.

Among the current limitations, we identify that the testing approach is tailored to short-running technical processes with integration of real (production use) services. Long running processes, possibly with human interaction or long waiting times in the activities, are currently not well supported, because of the typically large number of process instantiations required for testing. In addition, if the real services are not available for testing (e.g., for cost reasons), a possible extension would be to utilize mock testing services which simulate or proxy the actual services. Moreover, the presented fault localization approach provides useful insights for the system maintainers, but currently does not perform further processing of the derived results. To improve this aspect, the extracted rules could be used for guiding automated reconfiguration when a fault occurs. Furthermore, future work will focus on integrating test coverage mechanisms that help to actively investigate faults. Guiding the test execution with identified runtime faults could be used to execute insightful configurations and input requests which further narrow down the search space of possible fault reasons.

Q3: How can security and access control policies be enforced in the context of data processing applications and workflows?

Provisioning of security-critical applications, particular in Cloud environments with multiple tenants, requires systematic enforcement of access constraints to avoid unauthorized access. Chapter 5 has tackled this issue, with a focus on two well-established types of constraints: RBAC constraints which allow fine-grained assignment of roles and permissions, and entailment constraints which regulate the interdependencies between individual processing steps, for instance binding of duty or separation of duty. The logic for security enforcement is often considered "boilerplate" code which tends to blow up the application code and becomes error-prone if tailor made. Hence, we propose to define access constraints declaratively, separating the application logic from the security enforcement code. Our approach takes process models annotated with security annotations and automatically transforms them into process models which strictly follow the required enforcement procedure at runtime. Our implementation and evaluation is integrated with WS-BPEL and the Web services framework, although the approach is general.

The presented method provides a solid foundation and covers the crucial aspects of research question Q3, yet some extensions and optimizations still have to be tackled. One central issue for practical applicability is the performance of the query mechanism that checks security constraints for consistency, given the fact that the log data of past process invocations grows indefinitely. We envision advanced data storage and compression techniques, combined with query optimization to further reduce the increasing overhead over time. Moreover, our approach currently leads to the situation of possible process deadlocks (see evaluation in Section 5.7.2), which is desired from a security perspective (if none of the subjects is permitted to perform an action, no action should be performed), but may be critical from a practical perspective. Hence, advanced satisfiability and conflict prediction techniques have to be integrated with the approach, in order to avoid deadlocking in the first place.

6.3 Future Work

Within this thesis, different solutions for reliable application provisioning have been proposed, covering the infrastructure, platform, and application layer in Cloud environments. Yet, a number of important challenges remain which were out of scope of this thesis. In the following we conclude the thesis by summarizing these open issues for future work.

- It is expected that future research will build on and possibly extend the fault taxonomy for EBS derived in this thesis. The generic model for EBS (Section 3.2.2) will be encoded and instantiated for concrete event processing platforms, thereby providing a consistent mapping between the model state and the real state of the platform. This mapping will allow to utilize the taxonomy as the basis for two orthogonal goals: fault detection (or fault monitoring) and fault injection. Using this two-way synchronization, faults can be injected into the platform by generic manipulations on the model, and faults in the platform can be detected by monitoring changes in the model. This way, the fault detection and injection mechanisms can be implemented in a generic fashion, and only the model mapping needs to be implemented to integrate new platforms.
- The approach for testing idempotence of infrastructure automation scripts (Section 3.5) should be extended to also consider systematic testing of convergence. Particularly in multi-node environments where automations are continuously executed on interdependent machines, it must be ensured that the overall system eventually converges to the desired state. Moreover, the approach should be generalized to support arbitrary automation frameworks. The current evaluation is focused on Chef where tasks have total ordering, but other frameworks like Puppet use partial task ordering, which increases the complexity of the testing problem space.
- Elasticity is becoming a key feature for applications in the Cloud (e.g., elastic scaling in WS-Aggregation), which raises the demand for systematic engineering and validation mechanisms. Implementing elastic systems is a complex endeavor that is associated with a number of risks and potential faults. For instance, the system may fail to properly allocate sufficient resources on time, it may become *plastic* in the sense that it becomes unable to scale down after a scale out, or it may fail entirely due to the complex internal reconfigurations which may break the system. Future work will tackle this problem and propose systematic methods for testing of elastic computing systems. Initial work and research directions have been defined in [108–110], but there are still a number of open challenges related to modeling of elastic behavior, suitable coverage metrics, generation of test suites, and efficient test execution.
- In future work, we envision a tighter mutual integration of the proposed techniques for testing and fault localization in data-centric applications (Section 4). The information about detected faults at runtime could be used to trigger regression test suites, guiding the testing process towards the *proximity* (similar inputs and configurations) of the identified fault, and potentially redefining the test coverage criteria. Moreover, as an alternative to generating the entire test set up front, fault localization could be used for iterative test generation, combined with adaptive test case prioritization [168]. To that end, the execution starts with a set of random tests, which is then iteratively extended and refined based on the faulty test cases.
- Finally, the security policies supported in Chapter 5, currently covering RBAC with mutual exclusion and binding of duty constraints, will be extended with advanced schemes to support

a larger set of complex real-world situations, including extended notions of execution context (e.g., different end user devices), exceptional policy settings (e.g., break-glass rules [102, 215]), or dynamic runtime changes in the access constraint model. Moreover, future research is expected to explore the concepts established in this thesis for novel application areas, such as entailment constraints in the context of collaborative Web applications [115] and constraint enforcement in offline scenarios [114].

List of Acronyms

AOP	Aspect Oriented Programming. 55, 68, 167, 171
API	Application Programming Interface. 7, 24, 69, 88, 108, 151, 152, 154, 171
AQC	Active Query Coordinator. 44, 65, 66, 171
BPM	Business Process Management. 1, 171
CC	Cloud Computing. 17, 171
CEP	Complex Event Processing. 27, 29, 34, 39–41, 171
COP	Combinatorial Optimization Problem. 87, 171
CPU	Central Processing Unit. 70, 115, 156, 157, 171
CT	Combinatorial Testing. 24, 171
CTL	Computational Tree Logic. 25, 171
DAG	Directed Acyclic Graph. 79, 171
DB	Data Base. 152, 171
DBMS	Data Base Management System. 13, 171
DME	Dynamic Mutual Exclusion. 134, 137, 141, 143–148, 157, 171, 178
DSL	Domain-Specific Language. 7, 135, 138, 140, 141, 148, 149, 151, 152, 156, 164–169, 171, 177
DSMS	Data Stream Management System. 12, 171
EBNF	Extended Backus-Naur Form. 66, 171
EBS	Event-Based System. 1, 7, 8, 11, 22, 27–30, 35, 171
EC2	Elastic Compute Cloud. 17, 69, 70, 171
EDBPM	Event-Driven Business Process Management. 30, 42, 171
EDIP	Event-Driven Interaction Paradigms. 29, 38, 171
EFSM	Extended Finite State Machine. 25, 171
EPA	Event Processing Agent. 12, 31–35, 39–43, 171
EPN	Event Processing Network. 12, 13, 31, 32, 35, 40–42, 80, 81, 171
EPR	Endpoint Reference. 95, 96, 108, 112, 113, 117, 118, 171
ES	Elastic System. 18, 171
ESP	Event Stream Processing. 29, 39, 40, 171

GPS	Global Positioning System. 14, 89, 171
GUID	Globally Unique Identifier. 113, 171
HTTP	Hypertext Transfer Protocol. 14, 171
IaaS	Infrastructure as a Service. 13, 17, 171
IaC	Infrastructure as Code. 8, 9, 19, 54–57, 62, 64, 78, 81, 82, 171
IAM	Identity and Access Management. 7, 135, 136, 142, 146, 147, 154, 163, 164, 167, 171
IdP	Identity Provider. 142, 144, 147–149, 151–154, 157–159, 163, 171
IP	Internet Protocol. 1, 171
IR	Information Retrieval. 99, 100, 171
IT	Information Technology. 133, 135, 136, 171
JMX	Java Management Extensions. 70, 171
LXC	Linux Containers. 18, 55, 82, 171
MAPE	Monitor-Analyze-Plan-Execute. 80, 171
MBT	Model Based Testing. 23, 24, 171
NH	Neighborhood. 53, 54, 171
NIST	National Institute of Standards and Technology. 17, 171
OCL	Object Constraint Language. 140, 171
OS	Operating System. 8, 18, 55, 59, 171
PaaS	Platform as a Service. 17, 18, 171
PDP	Policy Decision Point. 7, 152, 154, 171
PEP	Policy Enforcement Point. 135, 151, 153, 154, 164, 171
PFT	Partial Fault Tolerance. 81, 171
POSIX	Portable Operating System Interface. 82, 171
Pub/Sub	Publish/Subscribe. 4, 171
QoS	Quality of Service. 2, 6, 16, 18, 19, 109, 171
RBAC	Role-Based Access Control. 2, 5, 7, 133–136, 138, 140–142, 146–149, 152, 157–161, 163–169, 171, 177
REST	Representational State Transfer. 14, 171
RPC	Remote Procedure Call. 4, 171
SaaS	Software as a Service. 17, 18, 171
SAML	Security Assertion Markup Language. 142, 148–154, 164, 171

SBA Service-Based Application. 6, 7, 13, 15, 16, 24, 85–88, 94, 102–104, 106, 108, 114, 123, 128, 171

SLA Service Level Agreement. 1, 2, 22, 133, 171

SME Static Mutual Exclusion. 134, 137, 141, 143–145, 147, 148, 157, 159, 161, 171, 178

SOA Service-Oriented Architecture. 13, 45, 81, 85, 113, 133, 134, 136, 146, 155, 164–166, 171

SOAP Simple Object Access Protocol. 14, 109, 111, 148, 151–154, 171

SOC Service-Oriented Computing. 13, 171

SSH Secure Shell. 68, 171

SSO Single Sign-On. 7, 150, 152, 167, 171

STG State Transition Graph. 8, 25, 55, 57, 59, 62, 63, 171

SUT System Under Test. 23–25, 55, 171

UML Unified Modeling Language. 23, 32, 109, 135, 138, 144, 149, 152, 169, 171

URL Uniform Resource Locator. 14, 108, 171

VM Virtual Machine. 4, 5, 8, 17, 18, 28, 43, 54, 55, 64, 82, 115, 171

VNS Variable Neighborhood Search. 53, 54, 83, 171

VRESCo Vienna Runtime Environment for Service-oriented Computing. 16, 45, 46, 171

WAQL Web services Aggregation Query Language. 5, 28, 171

WS Web Service. 1, 14, 171

WS-BPEL Web Services Business Process Execution Language. 7, 14, 16, 85, 87–92, 95, 108–110, 112–114, 116–118, 128, 129, 131, 135, 136, 141, 142, 144, 146–149, 151, 152, 154, 156, 157, 162–167, 169, 171

WSDL Web Services Description Language. 14, 15, 86, 110, 171

WSN Wireless Sensor Network. 22, 29, 32, 34, 40–42, 171

WWW World Wide Web. 1, 171

XACML eXtensible Access Control Markup Language. 166, 167, 171

XML Extensible Markup Language. 5, 14, 43, 71, 85, 86, 149, 151–155, 157, 163, 164, 167, 171

XPath XML Path Language. 14, 89, 90, 95, 113, 129, 171

XQuery XML Query Language. 5, 66, 155, 171, 178

XSD XML Schema Definition. 14, 149, 171

Bibliography

[1] Daniel J. Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Çetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag Maskey, Alex Rasin, Esther Ryvkina, Nesime Tatbul, Ying Xing, and Stanley B. Zdonik. The Design of the Borealis Stream Processing Engine. In *2nd Biennial Conference on Innovative Data Systems Research (CIDR)*, 2005. → pages: 31, 41, and 42

[2] Divyakant Agrawal, Amr El Abbadi, Sudipto Das, and Aaron J Elmore. Database scalability, elasticity, and autonomy in the cloud. In *Database Systems for Advanced Applications*, pages 2–15. Springer, 2011. → pages: 2

[3] Jagrati Agrawal, Yanlei Diao, Daniel Gyllstrom, and Neil Immerman. Efficient Pattern Matching Over Event Streams. In *SIGMOD Int. Conference on Management of Data*, 2008. → pages: 82

[4] David W. Aha. Tolerating noisy, irrelevant and novel attributes in instance-based learning algorithms. *Int. Journal of Man-Machine Studies*, 36(2):267–287, 1992. → pages: 110

[5] Ian F. Akyildiz, Weilian Su, Yogesh Sankarasubramaniam, and Erdal Cayirci. Wireless sensor networks: a survey. *Computer Networks*, 38(4):393–422, Elsevier, 2002. → pages: 31, 43, and 44

[6] Open Mashup Alliance. Enterprise Mashup Markup Language (EMML). `http://www.openmashup.org/omadocs/v1.0/index.html`. → pages: 113

[7] P.E. Ammann, P.E. Black, and W. Majurski. Using model checking to generate tests from specifications. In *2nd International Conference on Formal Engineering Methods*, pages 46–54, 1998. → pages: 99

[8] ANSI/IEEE. Standard glossary of software engineering terminology. STD-729-1991, 1991. → pages: 2

[9] Krzysztof R. Apt. *Principles of Constraint Programming*. Cambridge University Press, 2003. → pages: 115

[10] Michael Armbrust et al. Above the clouds: A berkeley view of cloud computing. Technical Report UCB/EECS-2009-28, University of California at Berkeley, 2009. → pages: 1 and 17

[11] Ellen M. Arruda and Mary C. Boyce. A three-dimensional constitutive model for the large stretch behavior of rubber elastic materials. *Journal of the Mechanics and Physics of Solids*, 41(2):389 – 412, 1993. → pages: 18

[12] Taimur Aslam, Ivan Krsul, and E Spafford. Use of a taxonomy of security faults. *19th National Information Systems Security Conference (NISSC)*, pages 551–560, 1996. → pages: 6 and 38

[13] Algirdas Avizienis, Jean Laprie, Brian Randell, and Carl Landwehr. Basic Concepts and Taxonomy of Dependable and Secure Computing. *IEEE Transactions on Dependable and Secure Computing*, 1:11–33, 2004. → pages: 2, 6, 20, 21, 22, 36, 37, and 38

[14] Ahmed Ayad and Jeffrey Naughton. Static optimization of conjunctive queries with sliding win- dows over infinite streams. In *SIGMOD Int. Conf. on Management of Data*, 2004. → pages: 83

[15] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. Models and issues in data stream systems. In *21st Symposium on Principles of Database Systems (PODS)*, pages 1–16, 2002. → pages: 1, 11, and 31

[16] Shivnath Babu, Utkarsh Srivastava, and Jennifer Widom. Exploiting k-constraints to reduce memory overhead in continuous queries over data streams. *ACM Transactions on Database Systems*, 29:545–580, 2004. → pages: 83

[17] Shivnath Babu and Jennifer Widom. Continuous queries over data streams. *ACM SIGMOD International Conference on Management of Data*, 30:109–120, 2001. → pages: 1, 31, and 49

[18] Ricardo Baeza-Yates and Ribeiro-Neto Berthier. *Modern information retrieval*. ACM Press, Addison-Wesley, 1999. → pages: 103, 104, 110, and 134

[19] Guruduth Banavar, Tushar Chandra, Robert Strom, and Daniel Sturman. A case for message oriented middleware. In *SDC*, 1999. → pages: 31

[20] Luciano Baresi and Sam Guinea. Towards dynamic monitoring of ws-bpel processes. In *International Conference on Service-Oriented Computing (ICSOC)*, pages 269–282. Springer, 2005. → pages: 14

[21] Roger S. Barga, Jonathan Goldstein, Mohamed H. Ali, and Mingsheng Hong. Consistent streaming through time: A vision for event stream processing. In *3rd Biennial Conference on Innovative Data Systems Research (CIDR)*, pages 363–374, 2007. → pages: 31

[22] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. *ACM SIGOPS Operating Systems Review*, 37(5):164–177, 2003. → pages: 17

[23] Alistair Barros, Gero Decker, Marlon Dumas, and Franz Weber. Correlation patterns in service-oriented architectures. In *Fundamental Approaches to Software Engineering*, volume 4422 of *Lecture Notes in Computer Science*, pages 245–259. Springer, 2007. → pages: 134

[24] Cesare Bartolini, Antonia Bertolino, Sebastian Elbaum, and Eda Marchetti. Bringing white-box testing to service oriented architectures through a service oriented approach. *Journal of Systems and Software*, 84(4):655–668, 2011. → pages: 90 and 132

[25] Cesare Bartolini, Antonia Bertolino, Eda Marchetti, and Ioannis Parissis. Data Flow-Based Validation of Web Services Compositions: Perspectives and Examples. *Architecting Dependable Systems V*, 2008. → pages: 133

[26] Cesare Bartolini, Antonia Bertolino, Eda Marchetti, and Andrea Polini. WS-TAXI: A WSDL-based Testing Tool for Web Services. In *2nd IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 326–335, 2009. → pages: 14 and 90

[27] David Basin, Jürgen Doser, and Torsten Lodderstedt. Model driven security: From UML models to access control infrastructures. *ACM Transactions on Software Engineering Methodology*, 15:39–91, 2006. → pages: 172

[28] Anne Baumgrass, Thomas Baier, Jan Mendling, and Mark Strembeck. Conformance Checking of RBAC Policies in Process-Aware Information Systems. In *BPM'11 Workshop on Workflow Security Audit and Certification (WfSAC)*. Springer, 2011. → pages: 170

[29] Armin Beer and Matthias Heindl. Issues in testing dependable event-based systems at a systems integration company. In *2nd IEEE International Conference on Availability, Reliability and Security (ARES)*, pages 1093–1100, 2007. → pages: 134

[30] Boris Beizer. *Software testing techniques (2nd ed.)*. Van Nostrand Reinhold Co., New York, USA, 1990. → pages: 2, 23, and 133

[31] Boualem Benatallah, Fabio Casati, and Farouk Toumani. Web Service Conversation Modeling: A Cornerstone for E-Business Automation. *IEEE Internet Computing*, 8(1):46–54, 2004. → pages: 174

[32] Djamal Benslimane, Schahram Dustdar, and Amit Sheth. Services Mashups: The New Generation of Web Applications. *IEEE Internet Computing*, 12(5):13–15, 2008. → pages: 113 and 136

[33] Elisa Bertino, Elena Ferraria, and Vijay Atluri. The specification and enforcement of authorization constraints in workflow management systems. *ACM Transactions on Information and System Security*, 2(1):65–104, 1999. → pages: 140, 162, and 168

[34] Antonia Bertolino. Software testing research: Achievements, challenges, dreams. In *Future of Software Engineering (FOSE)*, pages 85–103. IEEE, 2007. → pages: 23 and 24

[35] Antonia Bertolino, Guglielmo De Angelis, Sampo Kellomaki, and Andrea Polini. Enhancing service federation trustworthiness through online testing. *IEEE Computer*, 45(1):66–72, 2012. → pages: 131

[36] Robert V Binder. Testing object-oriented software: a survey. *Software Testing Verification and Reliability (STVR)*, 6(3-4):125–252, 1996. → pages: 6

[37] Christian Böhm, Beng Chin Ooi, Claudia Plant, and Ying Yan. Efficiently processing continuous k-nn queries on data streams. In *Int. Conf. on Data Engineering*, pages 156–165, 2007. → pages: 83

[38] Boris J. Bonfils and Philippe Bonnet. Adaptive and decentralized operator placement for in-network query processing. *Telecommunication Systems*, 26:389–409, 2004. → pages: 82

[39] Irina Botan, Donald Kossmann, Peter M. Fischer, Tim Kraska, Dana Florescu, and Rokas Tamosevicius. Extending xquery with window functions. In *33rd international Conference on Very Large Data Bases (VLDB)*, pages 75–86, 2007. → pages: 11 and 13

[40] Reinhardt A. Botha and Jan H.P. Eloff. Separation of duties for access control enforcement in workflow environments. *IBM Systems Journal*, 40(3):666–682, 2001. → pages: 140

[41] Christiano Braga. A transformation contract to generate aspects from access control policies. *Software and Systems Modeling*, 10:395–409, 2011. → pages: 173

[42] Yuriy Brun, Giovanna Di Marzo Serugendo, Cristina Gacek, Holger Giese, Holger Kienle, Marin Litoiu, Hausi Müller, Mauro Pezzè, and Mary Shaw. Engineering self-adaptive systems through feedback loops. In *Software Engineering for Self-Adaptive Systems*, pages 48–70. Springer, 2009. → pages: 82

[43] Stefan Bruning, Stephan Weissleder, and Miroslaw Malek. A Fault Taxonomy for Service-Oriented Architecture. In *10th IEEE High Assurance Systems Engineering Symposium (HASE)*, pages 367–368, 2007. → pages: 6 and 84

[44] Antonio Bucchiarone, Hernán Melgratti, and Francesco Severoni. Testing service composition. In *8th Argentine Symposium on Software Engineering*, 2007. → pages: 132

[45] Stefan Bucur, Vlad Ureche, Cristian Zamfir, and George Candea. Parallel symbolic execution for automated real-world software testing. In *ACM EuroSys Conference*, pages 183–198, 2011. → pages: 85

[46] Mark Burgess. Testable system administration. *Communications of the ACM*, 54(3):44–49, 2011. → pages: 19, 56, and 84

[47] Zack Butler and Daniela Rus. Event-based motion control for mobile-sensor networks. *Pervasive Computing*, 2(4):34–42, 2003. → pages: 31, 34, and 36

[48] Rajkumar Buyya, Chee Shin Yeo, Srikumar Venugopal, James Broberg, and Ivona Brandic. Cloud computing and emerging it platforms: Vision, hype, and reality for delivering computing as the 5th utility. *Future Generation computer systems*, 25(6):599–616, 2009. → pages: 1 and 17

[49] C. Cadar, P. Godefroid, S. Khurshid, C.S. Pasareanu, K. Sen, N. Tillmann, and W. Visser. Symbolic execution for software testing in practice: preliminary assessment. In *33rd International Conference on Software Engineering (ICSE)*, pages 1066 –1071, 2011. → pages: 85

[50] George Candea, Stefan Bucur, and Cristian Zamfir. Automated software testing as a service. In *1st ACM Symposium on Cloud Computing (SoCC)*, pages 155–160, 2010. → pages: 85

[51] Gerardo Canfora and Massimiliano Di Penta. Testing Services and Service-Centric Systems: Challenges and Opportunities. *IT Professional*, 8(2):10–17, 2006. → pages: 90 and 132

[52] Gerardo Canfora, Massimiliano Di Penta, Raffaele Esposito, and Maria Luisa Villani. A framework for qos-aware binding and re-binding of composite web services. *Journal of Systems and Software*, 81(10):1754–1769, 2008. → pages: 8 and 16

[53] J.C. Cannon and Marilee Byers. Compliance Deconstructed. *ACM Queue*, 4(7):30–37, Septemper 2006. → pages: 139

[54] Tien-Dung Cao, P. Felix, R. Castanet, and I. Berrada. Online testing framework for web services. In *3rd International Conference on Software Testing, Verification and Validation (ICST)*, pages 363–372, 2010. → pages: 131

[55] Barbara Carminati, Elena Ferrari, and Patrick C.K. Hung. Security conscious web service composition. In *International Conference on Web Services (ICWS)*, pages 489–496, 2006. → pages: 95

[56] Antonio Carzaniga, David Rosenblum, and Alexander Wolf. Design and evaluation of a wide-area event notification service. *ACM Transactions on Computer Systems*, 19(3), 2001. → pages: 31

[57] Giuliano Casale, Amir Kalbasi, Diwakar Krishnamurthy, and Jerry Rolia. Automatic stress testing of multi-tier systems by dynamic bottleneck switch generation. In *10th ACM/IFIP/USENIX International Middleware Conference*, pages 20:1–20:20, 2009. → pages: 85

[58] Fabio Casati, Ski Ilnicki, LiJie Jin, Vasudev Krishnamoorthy, and Ming-Chien Shan. Adaptive and dynamic service composition in eflow. In *Advanced Information Systems Engineering*, pages 13–31. Springer, 2000. → pages: 16

[59] K. S. Chan, Judith Bishop, Johan Steyn, Luciano Baresi, and Sam Guinea. A fault taxonomy for web service composition. In *International Conference on Service-Oriented Computing (ICSOC) - Workshops*, pages 363–375, 2009. → pages: 6, 37, and 84

[60] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM (JACM)*, 43(2):225–267, 1996. → pages: 135

[61] Anis Charfi and Mira Mezini. AO4BPEL: An Aspect-oriented Extension to BPEL. *World Wide Web Journal - Special Issue: Recent Advances in Web Services*, 10:309–344, 2007. → pages: 173

[62] Jianjun Chen, David DeWitt, Feng Tian, and Yuan Wang. NiagaraCQ: a scalable continuous query system for Internet databases. In *ACM SIGMOD International Conference on Management of Data*, pages 379–390, 2000. → pages: 83

[63] Jie Chen and Ron J. Patton. *Robust Model-based Fault Diagnosis for Dynamic Systems*. Kluwer Academic Publishers, 1999. → pages: 134

[64] Liming Chen and Algirdas Avizienis. N-version programming: A fault-tolerance approach to reliability of software operation. In *8th IEEE International Symposium on Fault-Tolerant Computing*, pages 3–9, 1978. → pages: 23

[65] Qiming Chen and Meichun Hsu. Data stream analytics as cloud service for mobile applications. In *Int. Symp. on Distributed Objects, Middleware, and Applications (DOA)*, 2010. → pages: 82

[66] Christine T. Cheng. The test suite generation problem: Optimal instances and their implications. *Discrete Applied Mathematics*, 155(15):1943–1957, 2007. → pages: 101

[67] Michal R. Chmielewski and Jerzy W. Grzymala-Busse. Global discretization of continuous attributes as preprocessing for machine learning. *International Journal of Approximate Reasoning*, 15(4):319 – 331, 1996. → pages: 108

[68] Calin Ciordas, Twan Basten, Andrei Rădulescu, Kees Goossens, and Jef Van Meerbergen. An event-based monitoring service for networks on chip. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 10(4):702–723, 2005. → pages: 32

[69] David D. Clark and David R. Wilson. A Comparison of Commercial and Military Computer Security Policies. In *IEEE Symp. on Security and Privacy*, April 1987. → pages: 140

[70] Alberto Coen-Porisini, Giovanni Denaro, Carlo Ghezzi, and Mauro Pezzé. Using symbolic execution for verifying safety-critical systems. In *ACM SIGSOFT Software Engineering Notes*, volume 26, pages 142–151. ACM, 2001. → pages: 85

[71] David M. Cohen, Siddharta R. Dalal, Michael L. Fredman, and Gardner C. Patton. The AETG system: an approach to testing based on combinatorial design. *IEEE Transactions on Software Engineering*, 23(7):437–444, 1997. → pages: 25 and 101

[72] European Commission. General Data Protection Regulation. http://eur-lex.europa.eu/LexUriServ/LexUriServ.do?uri=COM:2012:0011:FIN:EN:PDF, January 2012. Accessed: 2013-12-10. → pages: 6

[73] Microsoft Corporation. Windows workflow foundation. http://msdn.microsoft.com/en-us/vstudio/jj684582. Visited: 2013-10-15. → pages: 113

[74] Alva Couch and Yizhan Sun. On the algebraic structure of convergence. In *14th International Workshop on Distributed Systems: Operations and Management (DSOM)*, pages 28–40, 2003. → pages: 8, 19, 56, 57, 61, 62, and 84

[75] Jason Crampton. A reference monitor for workflow systems with constrained task execution. In *10th ACM Symposium on Access Control Models and Technologies (SACMAT)*, pages 38–47, 2005. → pages: 162

[76] Jason Crampton, Gregory Gutin, and Anders Yeo. On the parameterized complexity of the workflow satisfiability problem. In *19th ACM Conference on Computer and Communications Security (CCS)*, pages 857–868. ACM, October 2012. → pages: 165 and 166

[77] Gianpaolo Cugola and Alessandro Margara. TESLA: a Formally Defined Event Specification Language. In *International Conference on Distributed Event-Based Systems*, 2010. → pages: 82

[78] Krzysztof Czarnecki and Simon Helsen. Feature-based survey of model transformation approaches. *IBM Systems Journal - Model-driven software development*, 45:621–645, July 2006. → pages: 173

[79] Frank Dabek, Nickolai Zeldovich, Frans Kaashoek, David Mazières, and Robert Morris. Event-driven programming for robust software. In *10th ACM SIGOPS (workshops)*, 2002. → pages: 31

[80] Marios Damianides. How does SOX change IT? *Journal of Corporate Accounting & Finance*, 15(6):35–41, 2004. → pages: 139

[81] Abhinandan Das, Johannes Gehrke, and Mirek Riedewald. Approximate join processing over data streams. In *ACM SIGMOD International Conference on Management of Data*, pages 40–51, 2003. → pages: 41

[82] Francesco de Angelis, Andrea Polini, and Guglielmo de Angelis. A counter-example testing approach for orchestrated services. In *3rd IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 373–382, 2010. → pages: 133

[83] Giovanni Denaro and Mauro Pezze. Petri nets and software engineering. In *Lectures on Concurrency and Petri Nets*, pages 439–466. Springer, 2004. → pages: 133

[84] Massimiliano Di Penta, Raffaele Esposito, Maria Luisa Villani, Roberto Codato, Massimiliano Colombo, and Elisabetta Di Nitto. Ws binder: a framework to enable dynamic binding of composite web services. In *International Workshop on Service-Oriented Software Engineering*, pages 74–80. ACM, 2006. → pages: 16

[85] Massimiliano Di Penta, Raffaele Esposito, Maria Luisa Villani, Roberto Codato, Massimiliano Colombo, and Elisabetta Di Nitto. Ws binder: a framework to enable dynamic binding of composite web services. In *International Workshop on Service-Oriented Software Engineering (SOSE)*, pages 74–80. ACM, 2006. → pages: 94

[86] Nathan Dimmock, András Belokosztolszki, David Eyers, Jean Bacon, and Ken Moody. Using trust and risk in role-based access control policies. In *9th ACM Symposium on Access Control Models and Technologies (SACMAT)*, 2004. → pages: 171

[87] Wen-Li Dong, Hang Yu, and Yu-Bing Zhang. Testing BPEL-based Web Service Composition Using High-level Petri Nets. In *10th IEEE International Enterprise Distributed Object Computing Conference (EDOC'06)*, pages 441–444, 2006. → pages: 14 and 133

[88] Dirk Draheim. The Service-Oriented Metaphor Deciphered. *Journal of Computing Science and Engineering (JCSE)*, 4(4):253–275, 2010. → pages: 139

[89] Dirk Draheim, John Grundy, John Hosking, Christof Lutteroth, and Gerald Weber. Realistic Load Testing of Web Applications. In *IEEE Conference on Software Maintenance and Reengineering (CSMR)*, pages 57–70, 2006. → pages: 122

[90] Joao A. Duraes and Henrique S. Madeira. Emulation of software faults: A field data study and a practical approach. *IEEE Transactions on Software Engineering*, 32(11):849–867, 2006. → pages: 84

[91] Schahram Dustdar, Yike Guo, Benjamin Satzger, and Hong-Linh Truong. Principles of elastic processes. *Internet Computing, IEEE*, 15(5):66–71, 2011. → pages: 5, 18, and 179

[92] Schahram Dustdar and Wolfgang Schreiner. A Survey on Web Services Composition. *International Journal of Web and Grid Services*, 1(1):1–30, 2005. → pages: 1 and 89

[93] Johann Eder and Wolfgang Gruber. A meta model for structured workflows supporting workflow transformations. In *6th East European Conference on Advances in Databases and Information Systems (ADBIS'02)*, pages 326–339. Springer-Verlag, 2002. → pages: 173

[94] Marcelo Medeiros Eler, Marcio Eduardo Delamaro, Jose Carlos Maldonado, and Paulo Cesar Masiero. Built-in structural testing of web services. In *Brazilian Symposium on Software Engineering (SBES)*, pages 70–79, 2010. → pages: 90 and 132

[95] Elmootazbellah Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys*, 34(3), 2002. → pages: 31

[96] Thomas Erl. *Service-oriented architecture*. Prentice Hall Englewood Cliffs, 2004. → pages: 1 and 13

[97] Opher Etzion and Peter Niblett. *Event Processing in Action*. Manning Publications, 2010. → pages: 1, 11, 29, 31, and 32

[98] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys*, 35:114–131, 2003. → pages: 32 and 40

[99] Mohamed Fayad and Douglas C Schmidt. Object-oriented application frameworks. *Communications of the ACM*, 40(10):32–38, 1997. → pages: 16

[100] Dieter Fensel and Christoph Bussler. The web service modeling framework wsmf. *Electronic Commerce Research and Applications*, 1(2):113 – 137, 2002. → pages: 171

[101] David Ferraiolo, Richard Kuhn, and Ramaswamy Chandramouli. *Role-Based Access Control*. Artech House, second edition, 2007. → pages: 139

[102] Ana Ferreira, Ricardo Cruz-Correia, Luis Antunes, Pedro Farinha, Ernesto Oliveira-Palhares, David W. Chadwick, and Altamira Costa-Pereira. How to break access control in a controlled manner. In *19th IEEE International Symposium on Computer-Based Medical Systems (CBMS)*, pages 847–854, 2006. → pages: 170 and 182

[103] Ludger Fiege, Felix Gartner, Oliver Kasten, and Andreas Zeidler. Supporting Mobility in Content-Based Publish/Subscribe Middleware. In *Middleware Conference*, pages 103–122. Springer, 2003. → pages: 32

[104] Roy Fielding, Jim Gettys, Jeffrey Mogul, Henrik Frystyk, Larry Masinter, Paul Leach, and Tim Berners-Lee. Hypertext transfer protocol – HTTP/1.1. `https://tools.ietf.org/html/rfc2616`, 1999. → pages: 14

[105] Howard Foster, Sebastian Uchitel, Jeff Magee, and Jeff Kramer. An integrated workbench for model-based engineering of service compositions. *IEEE Transactions on Services Computing*, 3(2):131–144, 2010. → pages: 171

[106] Chris Fowler and Behrang Qasemizadeh. Towards a Common Event Model for an Integrated Sensor Information System. In *Workshop on the Semantic Sensor Web*, 2009. → pages: 83

[107] Martin Gaedke, Johannes Meinecke, and Martin Nussbaumer. A modeling approach to federated identity and access management. In *Special interest tracks and posters of the 14th international conference on World Wide Web*, pages 1156–1157. ACM, 2005. → pages: 9

[108] Alessio Gambi, Waldemar Hummer, and Schahram Dustdar. Automated testing of cloud-based elastic systems with autocles. In *28th IEEE/ACM International Conference on Automated Software Engineering (ASE), Demo track*, pages 714–717, 2013. → pages: 18, 181, and 226

[109] Alessio Gambi, Waldemar Hummer, and Schahram Dustdar. Testing Elastic Systems with Surrogate Models. In *ICSE Workshop on Combining Modelling and Search-Based Software Engineering (CMSBSE)*, 2013. → pages: 18, 85, 181, and 226

[110] Alessio Gambi, Waldemar Hummer, Hong-Linh Truong, and Schahram Dustdar. Testing elastic computing systems. *IEEE Internet Computing*, 17(6):76–82, 2013. → pages: 18, 85, 181, and 225

[111] José García-fanjul, Javier Tuya, and Claudio De La Riva. Generating Test Cases Specifications for BPEL Compositions of Web Services Using SPIN. In *WS-MaTe 2006*, pages 83–94, 2006. → pages: 132

[112] David Garlan and David Notkin. Formalizing design spaces: Implicit invocation mechanisms. In *4th International Symposium of VDM Europe on Formal Software Development*, pages 31–44, 1991. → pages: 31

[113] Patrick Gaubatz, Waldemar Hummer, Uwe Zdun, and Mark Strembeck. Supporting Customized Views for Enforcing Access Control Constraints in Real-time Collaborative Web Applications. In *International Conference on Web Engineering (ICWE)*, 2013. → pages: 175 and 227

[114] Patrick Gaubatz, Waldemar Hummer, Uwe Zdun, and Mark Strembeck. Enforcing Entailment Constraints in Offline Editing Scenarios for Real-time Collaborative Web Documents. In *29th ACM Symposium On Applied Computing (SAC)*, 2014. → pages: 175, 182, and 227

[115] Patrick Gaubatz and Uwe Zdun. Supporting entailment constraints in the context of collaborative web applications. In *28th Symposium On Applied Computing (SAC)*. ACM, 2013. → pages: 175, 182, and 227

[116] David Gelernter. Multiple tuple spaces in Linda. *Parallel Architectures and Languages Europe*, 366:20–27, 1989. → pages: 31

[117] Ioana Giurgiu, Claris Castillo, Asser Tantawi, and Malgorzata Steinder. Enabling efficient placement of virtual infrastructures in the cloud. In *13th ACM/IFIP/USENIX International Middleware Conference*, pages 332–353, 2012. → pages: 84

[118] John B Goodenough and Susan L Gerhart. Toward a theory of test data selection. *IEEE Transactions on Software Engineering*, (2):156–173, 1975. → pages: 26

[119] Michaela Greiler, Hans-Gerhard Gross, and Arie van Deursen. Evaluation of online testing for services: a case study. In *2nd International Workshop on Principles of Engineering Service-Oriented Systems (PESOS)*, pages 36–42, 2010. → pages: 131

[120] Liang Guo, Abhik Roychoudhury, and Tao Wang. Accurately choosing execution runs for software fault localization. In *Compiler Construction*, pages 80–95. Springer, 2006. → pages: 135

[121] Vassos Hadzilacos and Sam Toueg. A modular approach to fault-tolerant broadcasts and related problems. Technical report, Cornell University, Ithaca, NY, USA, 1994. → pages: 83

[122] Jiawei Han, Micheline Kamber, and Jian Pei. *Data mining: concepts and techniques*. Morgan Kaufmann, 2006. → pages: 1

[123] Pierre Hansen and Nenad Mladenović. *Handbook of metaheuristics*, chapter Variable Neighborhood Search. Springer, 2003. → pages: 55

[124] Reiko Heckel and Leonardo Mariani. Automatic conformance testing of web services. *Fundamental Approaches to Software Engineering*, pages 34–48, 2005. → pages: 132

[125] Pat Helland. Idempotence is not a medical condition. *ACM Queue*, 10(4), 2012. → pages: 84

[126] Pat Helland and David Campbell. Building on quicksand. In *Conference on Innovative Data Systems Research (CIDR)*, 2009. → pages: 84

[127] Julia Hielscher, Raman Kazhamiakin, Andreas Metzger, and Marco Pistore. A framework for proactive self-adaptation of service-based applications based on online testing. In Petri Mähönen, Klaus Pohl, and Thierry Priol, editors, *Towards a Service-Based Internet*, pages 122–133. Springer, 2008. → pages: 131

[128] Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David Culler, and Kristofer Pister. System architecture directions for networked sensors. *SIGPLAN Notices*, 35:93–104, 2000. → pages: 32

[129] Bernhard Hoisl and Mark Strembeck. A UML Extension for the Model-driven Specification of Audit Rules. In *2nd International Workshop on Information Systems Security Engineering (WISSE)*. Springer Verlag, 2012. → pages: 161

[130] Hyoung Seok Hong, Insup Lee, Oleg Sokolsky, and Hasan Ural. A temporal logic based theory of test coverage and generation. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 327–341. Springer, 2002. → pages: 26

[131] George Hripcsak and Adam S. Rothschild. Agreement, the f-measure, and reliability in information retrieval. *Journal of the American Medical Informatics Association*, 12(3):296–298, 2005. → pages: 110

[132] Michael N. Huhns and Munindar P. Singh. Service-Oriented Computing: Key Concepts and Principles. *IEEE Internet Computing*, 9:75–81, January 2005. → pages: 139

[133] Jez Humble and David Farley. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Addison-Wesley Professional, 2010. → pages: 84

[134] Waldemar Hummer, Patrick Gaubatz, Mark Strembeck, Uwe Zdun, and Schahram Dustdar. An integrated approach for identity and access management in a SOA context. In *16th ACM Symposium on Access Control Models and Technologies (SACMAT)*, pages 21–30, 2011. → pages: xxii, 9, 95, 131, 141, 146, 171, 173, and 226

[135] Waldemar Hummer, Patrick Gaubatz, Mark Strembeck, Uwe Zdun, and Schahram Dustdar. Enforcement of Entailment Constraints in Distributed Service-Based Business Processes. *Information and Software Technology (IST)*, 55(11):1884–1903, 2013. → pages: xxi, 9, and 224

[136] Waldemar Hummer, Christian Inzinger, Philipp Leitner, Benjamin Satzger, and Schahram Dustdar. Deriving a unified fault taxonomy for distributed event-based systems. In *6th ACM International Conference on Distributed Event-Based Systems (DEBS)*, pages 167–178, 2012. → pages: xxi, 6, 137, and 225

[137] Waldemar Hummer, Philipp Leitner, and Schahram Dustdar. A Step-by-Step Debugging Technique To Facilitate Mashup Development and Maintenance. In *4th International Workshop on Web APIs and Services Mashups, co-located with ECOWS'10*, 2010. → pages: xxii, 7, and 226

[138] Waldemar Hummer, Philipp Leitner, and Schahram Dustdar. SEPL – a domain-specific language and execution environment for protocols of stateful Web services. *Distributed and Parallel Databases*, 29(4):277–307, 2011. → pages: 174 and 225

[139] Waldemar Hummer, Philipp Leitner, and Schahram Dustdar. WS-Aggregation: Distributed Aggregation of Web Services Data. In *ACM Symposium On Applied Computing (SAC)*, pages 1590–1597, 2011. → pages: xxi, 7, 50, 83, 100, 114, 161, and 226

[140] Waldemar Hummer, Philipp Leitner, Anton Michlmayr, Florian Rosenberg, and Schahram Dustdar. *VRESCo - Vienna Runtime Environment for Service-oriented Computing*, pages 299–324. Service Engineering. European Research Results. Springer, 2010. → pages: 16, 47, and 229

[141] Waldemar Hummer, Philipp Leitner, Benjamin Satzger, and Schahram Dustdar. Dynamic migration of processing elements for optimized query execution in event-based systems. In *1st International Symposium on Secure Virtual Infrastructures (DOA-SVI'11), OnTheMove Federated Conferences*, pages 451–468, 2011. → pages: xxii, 7, 44, and 226

[142] Waldemar Hummer, Orna Raz, and Schahram Dustdar. Towards Efficient Measuring of Web Services API Coverage. In *3rd International Workshop on Principles of Engineering Service-Oriented Systems (PESOS), co-located with ICSE'11*, pages 22–28, 2011. → pages: xxii, 9, 112, and 226

[143] Waldemar Hummer, Orna Raz, Onn Shehory, Philipp Leitner, and Schahram Dustdar. Test coverage of data-centric dynamic compositions in service-based systems. In *4th International Conference on Software Testing, Verification and Validation (ICST'11)*, pages 40–49, 2011. → pages: xxii, 9, 173, and 226

[144] Waldemar Hummer, Orna Raz, Onn Shehory, Philipp Leitner, and Schahram Dustdar. Testing of Data-Centric and Event-Based Dynamic Service Compositions. *Software Testing, Verification and Reliability (STVR)*, 23(6):465–497, 2013. → pages: xxi, 9, 85, and 225

[145] Waldemar Hummer, Florian Rosenberg, Fábio Oliveira, and Tamar Eilam. Automated testing of chef automation scripts. In *ACM/IFIP/USENIX Middleware Conference (tool demo track)*, 2013. → pages: xxi, 8, and 225

[146] Waldemar Hummer, Florian Rosenberg, Fábio Oliveira, and Tamar Eilam. Testing idempotence for infrastructure as code. In *14th ACM/IFIP/USENIX Middleware Conference*, pages 368–388, 2013. **Best Student Paper Award**. → pages: xxi, 8, and 225

[147] Waldemar Hummer, Benjamin Satzger, and Schahram Dustdar. Elastic Stream Processing in the Cloud. *Wiley Interdisciplinary Rewiews: Data Mining and Knowledge Discovery*, 3(5):333–345, 2013. → pages: xxi, 7, and 225

[148] Waldemar Hummer, Benjamin Satzger, Philipp Leitner, Christian Inzinger, and Schahram Dustdar. Distributed Continuous Queries Over Web Service Event Streams. In *7th IEEE International Conference on Next Generation Web Services Practices (NWeSP)*, pages 176–181, 2011. → pages: xxii, 7, 52, 100, and 226

[149] Monica Hutchins, Herb Foster, Tarak Goradia, and Thomas Ostrand. Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *16th International Conference on Software Engineering (ICSE)*, pages 191–200, 1994. → pages: 134 and 135

[150] Michael Hüttermann. *DevOps for Developers*. Apress, 2012. → pages: 8 and 19

[151] IBM alphaWorks. Focus code and functional coverage tool. `http://alphaworks.ibm.com/tech/focus`. → pages: 91 and 115

[152] Cisco Systems Inc. Visual networking index. `http://www.cisco.com/web/go/vni`. Accessed: July 5, 2013. → pages: 1

[153] Christian Inzinger, Waldemar Hummer, Ioanna Lytra, Philipp Leitner, Huy Tran, Uwe Zdun, and Schahram Dustdar. Decisions, models, and monitoring a lifecycle model for the evolution of service-based systems. In *17th IEEE International EDOC Conference*, 2013. → pages: 227

[154] Christian Inzinger, Waldemar Hummer, Benjamin Satzger, Philipp Leitner, and Schahram Dustdar. Towards identifying root causes of faults in service-based applications. In *31st International Symposium on Reliable Distributed Systems (poster paper)*, pages 404–405, 2012. → pages: xxii, 9, 136, and 227

[155] Christian Inzinger, Waldemar Hummer, Benjamin Satzger, Philipp Leitner, and Schahram Dustdar. Identifying incompatible service implementations using pooled decision trees.

In *28th ACM Symposium on Applied Computing (SAC), DADS Track*, pages 485–492, 2013. → pages: xxii, 9, and 227

[156] Christian Inzinger, Waldemar Hummer, Benjamin Satzger, Philipp Leitner, and Schahram Dustdar. Generic Event-Based Monitoring and Adaptation Methodology for Heterogeneous Distributed Systems. *Software: Practice and Experience*, 2014. → pages: 225

[157] Christian Inzinger, Benjamin Satzger, Waldemar Hummer, and Schahram Dustdar. Specification and deployment of distributed monitoring and adaptation infrastructures. In *Workshop on Performance Assessment and Auditing in Service Computing, co-located with ICSOC'10*, pages 167–178, 2012. → pages: 227

[158] Christian Inzinger, Benjamin Satzger, Waldemar Hummer, Philipp Leitner, and Schahram Dustdar. Non-intrusive policy optimization for dependable and adaptive service-oriented systems. In *27th Annual ACM Symposium on Applied Computing (SAC)*, pages 504–510, 2012. → pages: 227

[159] Christian Inzinger, Benjamin Satzger, Philipp Leitner, Waldemar Hummer, and Schahram Dustdar. Model-based adaptation of cloud computing applications. In *International Conference on Model-Driven Engineering and Software Development*, pages 351–355, 2013. → pages: 227

[160] Yannis E. Ioannidis. Query optimization. *ACM Computing Surveys*, 28:121–123, 1996. → pages: 82

[161] Rolf Isermann. Model-based fault-detection and diagnosis - status and applications. *Annual Reviews in Control*, 29(1):71–85, 2005. → pages: 6

[162] Sadeka Islam, Kevin Lee, Alan Fekete, and Anna Liu. How a consumer can measure elasticity for cloud platforms. In *International Conference on Performance Engineering (ICPE)*, pages 85–96, 2012. → pages: 18

[163] Gabriela Jacques-Silva, Bugra Gedik, Henrique Andrade, Kun-Lung Wu, and Ravishankar Iyer. Fault injection-based assessment of par- tial fault tolerance in stream processing applications. In *5th International Conference on Distributed Event-Based Systems (DEBS)*, 2011. → pages: 6, 42, and 84

[164] Muhammad Jaffar-ur Rehman, Fakhra Jabeen, Antonia Bertolino, and Andrea Polini. Testing software components for integration: a survey of issues and techniques. *Software Testing, Verification and Reliability (STVR)*, 17:95–133, 2007. → pages: 134

[165] Pankaj Jalote. *Fault tolerance in distributed systems*. Prentice Hall, 1994. → pages: 2 and 6

[166] Meiko Jensen and Sven Feja. A security modeling approach for web-service-based business processes. In *16th Annual IEEE International Conference on the Engineering of Computer Based Systems (ECBS'09)*, pages 340–347, 2009. → pages: 170

[167] Zbigniew Jerzak and Christof Fetzer. Bloom filter based routing for content-based publish/subscribe. In *2nd ACM International Conference on Distributed Event-Based Systems (DEBS)*, pages 71–81, 2008. → pages: 31 and 40

[168] Bo Jiang, Zhenyu Zhang, Wing Kwong Chan, and TH Tse. Adaptive random test case prioritization. In *24th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 233–244, 2009. → pages: 181

[169] James A. Jones and Mary Jean Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *20th IEEE/ACM International Conference on Automated software engineering (ASE)*, pages 273–282, 2005. → pages: 134

[170] Lukasz Juszczyk and Schahram Dustdar. Script-Based Generation of Dynamic Testbeds for SOA. In *International Conference on Web Services (ICWS)*, pages 195–202, 2010. → pages: 120

[171] Jeffrey O. Kephart and David M. Chess. The vision of autonomic computing. *Computer*, 36(1), 2003. → pages: 82 and 179

[172] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *European Conference on Object-Oriented Programming (ECOOP'97)*, pages 220–242, 1997. → pages: 173

[173] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. kvm: the linux virtual machine monitor. In *Proceedings of the Linux Symposium*, volume 1, pages 225–230, 2007. → pages: 17

[174] Michael Klein and Birgitta Konig-Ries. Combining query and preference-an approach to fully automatize dynamic service binding. In *IEEE International Conference on Web Services*, pages 788–791. IEEE, 2004. → pages: 16

[175] Donald Kossmann, Tim Kraska, and Simon Loesing. An evaluation of alternative architectures for transaction processing in the cloud. In *ACM SIGMOD International Conference on Management of Data*, pages 579–590, 2010. → pages: 18

[176] Bhaskar Krishnamachari, Deborah Estrin, and Stephen Wicker. The impact of data aggregation in wireless sensor networks. In *22nd International Conference on Distributed Computing Systems - Workshops*, pages 575–578, 2002. → pages: 31, 34, and 36

[177] Christopher Krügel, Thomas Toth, and Clemens Kerer. Decentralized event correlation for intrusion detection. In *4th International Conference on Information Security and Cryptology (ICISC)*, pages 114–131, 2002. → pages: 31 and 34

[178] D. Richard Kuhn, Dolores R. Wallace, and Albert M. Gallo. Software fault interactions and implications for software testing. *IEEE Transactions on Software Engineering*, 30(6):418–421, 2004. → pages: 25

[179] Devdatta Kulkarni and Anand Tripathi. Context-aware role-based access control in pervasive computing systems. In *13th ACM Symposium on Access Control Models and Technologies (SACMAT)*, pages 113–122, 2008. → pages: 171

[180] Puppet Labs. Cucumber-puppet. `http://projects.puppetlabs.com/projects/cucumber-puppet`. Accessed: 2013-02-03. → pages: 84

[181] Geetika T. Lakshmanan, Yuri G. Rabinovich, and Opher Etzion. A stratified approach for supporting high throughput event processing applications. In *3rd ACM International Conference on Distributed Event-Based Systems (DEBS)*, pages 5:1–5:12, 2009. → pages: 43

[182] Mounir Lallali, Fatiha Zaidi, and Ana Cavalli. Timed modeling of web services composition for automatic testing. In *3rd IEEE International Conference on Signal-Image Technologies and Internet-Based Systems (SITIS'07)*, pages 417–426, 2007. → pages: 133

[183] Janusz W. Laski and Bogdan Korel. A data flow oriented program testing strategy. *IEEE Transactions on Software Engineering*, SE-9(3):347–354, 1983. → pages: 132

[184] George Lawton. Developing software online with platform-as-a-service technology. *Computer*, 41(6):13–15, 2008. → pages: 18

[185] Philipp Leitner, Johannes Ferner, Waldemar Hummer, and Schahram Dustdar. Data-driven and automated prediction of service level agreement violations in service compositions. *Distributed and Parallel Databases*, 31(3):447–470, 2013. → pages: 225

[186] Philipp Leitner, Waldemar Hummer, and Schahram Dustdar. A monitoring data set for evaluating qos-aware service-based systems. In *ICSE Workshop on Principles of Engineering Service Oriented Systems (PESOS)*, pages 67–68, 2012. → pages: 228

[187] Philipp Leitner, Waldemar Hummer, and Schahram Dustdar. Cost-based optimization of service compositions. *IEEE Transactions on Services Computing (TSC)*, 6(2):239–251, 2013. → pages: 124, 139, and 225

[188] Philipp Leitner, Waldemar Hummer, Benjamin Satzger, and Schahram Dustdar. Stepwise and asynchronous runtime optimization of web service compositions. In *12th International Conference on Web Information System Engineering (WISE)*, pages 290–297. Springer, 2011. → pages: 228

[189] Philipp Leitner, Waldemar Hummer, Benjamin Satzger, Christian Inzinger, and Schahram Dustdar. Cost-Efficient and Application SLA-Aware Client Side Request Scheduling in an Infrastructure-as-a-Service Cloud. In *5th IEEE International Conference on Cloud Computing*, pages 213–220, 2012. → pages: 228

[190] Philipp Leitner, Christian Inzinger, Waldemar Hummer, Benjamin Satzger, and Schahram Dustdar. Application-level performance monitoring of cloud services based on the complex event processing paradigm. In *5th IEEE International Conference on Service-Oriented Computing and Applications (SOCA)*, pages 1–8, 2012. → pages: 228

[191] Philipp Leitner, Benjamin Satzger, Waldemar Hummer, Christian Inzinger, and Schahram Dustdar. Cloudscale: a novel middleware for building transparently scaling cloud applications. In *27th Annual ACM Symposium on Applied Computing (SAC)*, pages 434–440. ACM, 2012. → pages: 228

[192] Philipp Leitner, Branimir Wetzstein, Dimka Karastoyanova, Waldemar Hummer, Schahram Dustdar, and Frank Leymann. Preventing SLA violations in service compositions using aspect-based fragment substitution. In *8th International Conference on Service-Oriented Computing*, pages 365–380. Springer, 2010. → pages: 228

[193] Marek Leszak, Dewayne E. Perry, and Dieter Stoll. A case study in root cause defect analysis. In *22nd International Conference on Software Engineering (ICSE)*, pages 428–437, 2000. → pages: 38

[194] Guoli Li, Vinod Muthusamy, and Hans-Arno Jacobsen. A distributed service-oriented architecture for business process execution. *ACM Transactions on the Web*, 4:2:1–2:33, 2010. → pages: 44

[195] Xiaogang Li and Gagan Agrawal. Efficient evaluation of XQuery over streaming data. In *International Conference on Very Large Data Bases*, pages 265–276, 2005. → pages: 29

[196] Zhong Jie Li, Wei Sun, and Bin Du. BPEL4WS unit testing: Framework and implementation. *International Journal of Business Process Integration and Management*, 3(2):131–143, 2008. → pages: 14

[197] Christoph Liebig, Mariano Cilia, and Alejandro Buchmann. Event composition in time-dependent distributed systems. In *International Conference on Cooperative Information Systems (CoopIS)*, pages 70–78, 1999. → pages: 43

[198] Dan Lin, Prathima Rao, Elisa Bertino, Ninghui Li, and Jorge Lobo. Policy decomposition for collaborative access control. In *13th ACM SACMAT*, pages 103–112, 2008. → pages: 172

[199] Kwei-Jay Lin, M. Panahi, Yue Zhang, Jing Zhang, and Soo-Ho Chang. Building accountability middleware to support dependable soa. *Internet Computing, IEEE*, 13(2):16 –25, 2009. → pages: 135

[200] Chien-Hung Liu, David C. Kung, Pei Hsia, and Chih-Tung Hsu. Structural testing of web applications. In *11th IEEE International Symposium on Software Reliability Engineering (ISSRE)*, 2000. → pages: 132

[201] Guangtian Liu, Aloysius K. Mok, and Eric J. Yang. Composite events for network event correlation. In *Sixth IFIP/IEEE International Symposium on Integrated Network Management*, pages 247–260, 1999. → pages: 134

[202] Ling Liu, Calton Pu, and Wei Tang. Continual queries for Internet scale event-driven information delivery. *IEEE Trans. on Knowledge and Data Engineering*, 11(4), 1999. → pages: 83

[203] Torsten Lodderstedt, David A. Basin, and Jürgen Doser. SecureUML: A UML-Based Modeling Language for Model-Driven Security. In *5th International Conference on The Unified Modeling Language (UML)*, pages 426–441. Springer, 2002. → pages: 173

[204] Mike Loukides. *What is DevOps?* O'Reilly Media, 2012. → pages: 19

[205] Daniel Lübke, Leif Singer, and Alex Salnikow. Calculating BPEL Test Coverage Through Instrumentation. In *ICSE Workshop on Automation of Software Test (AST)*, pages 115–122, 2009. → pages: 113

[206] David Luckham and Roy Schulte. Event processing glossary v1.1. *Event Processing Technical Society*, 2, 2008. → pages: 11

[207] David C. Luckham. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison-Wesley Longman, 2001. → pages: 1, 11, 31, and 82

[208] David C. Luckham and Brian Frasca. Complex Event Processing in Distributed Systems. *Analysis*, 28, 1998. → pages: 31

[209] David C. Luckham, John J. Kenney, Larry M. Augustin, James Vera, Doug Bryan, and Walter Mann. Specification and analysis of system architecture using Rapide. *IEEE Transactions on Software Engineering*, 21(4), 1995. → pages: 31

[210] Shruti P. Mahambre, Madhu Kumar, and Umesh Bellur. A Taxonomy of QoS-Aware, Adaptive Event-Dissemination Middleware. *IEEE Internet Computing*, 11(4):35–44, 2007. → pages: 31

[211] Yashwant K. Malaiya, Michael Naixin Li, James M. Bieman, and Rick Karcich. Software reliability growth with test coverage. *Reliability, IEEE Transactions on*, 51(4):420–426, 2002. → pages: 2 and 26

[212] Sam Malek, Marija Mikic-Rakic, and Nenad Medvidovic. A style-aware architectural middleware for resource- constrained, distributed systems. *IEEE Transactions on Software Engineering*, 31, 2005. → pages: 31

[213] Leonardo Mariani, Sofia Papagiannakis, and Mauro Pezze. Compatibility and regression testing of cots-component-based software. In *Software Engineering, 2007. ICSE 2007. 29th International Conference on*, pages 85–95. IEEE, 2007. → pages: 134

[214] Leonardo Mariani and Mauro Pezzè. Dynamic detection of cots component incompatibility. *Software, IEEE*, 24(5):76–85, 2007. → pages: 134

[215] Srdjan Marinovic, Robert Craven, Jiefei Ma, and Naranker Dulay. Rumpole: a flexible break-glass access control model. In *16th ACM Symposium on Access Control Models and Technologies (SACMAT)*, 2011. → pages: 170 and 182

[216] Wes Masri. Fault localization based on information flow coverage. *Software Testing, Verification and Reliability (STVR)*, 20:121–147, 2010. → pages: 134

[217] Mark T. Maybury. Generating Summaries From Event Data. *International Journal on Information Processing and Management*, 31:735–751, September 1995. → pages: 82

[218] Philip Mayer and Daniel Lübke. Towards a BPEL unit testing framework. In *Workshop on Testing, Analysis, and Verification of Web Services and Applications*, pages 33–42. ACM, 2006. → pages: 14

[219] Pietro Mazzoleni, Bruno Crispo, Swaminathan Sivasubramanian, and Elisa Bertino. XACML Policy Integration Algorithms. *ACM Transactions on Information System Security*, 11:4:1–4:29, February 2008. → pages: 173

[220] Dennis McCarthy and Umeshwar Dayal. The architecture of an active database management system. In *SIGMOD'89*. → pages: 31 and 36

[221] Sheila A. McIlraith, Tran Cao Son, and Honglei Zeng. Semantic web services. *IEEE Intelligent Systems*, 16(2):46–53, 2001. → pages: 171

[222] Lijun Mei, W.K. Chan, and T.H. Tse. Data flow testing of service-oriented workflow applications. In *30th International Conference on Software Engineering (ICSE)*, 2008. → pages: 14 and 133

[223] René Meier and Vinny Cahill. Taxonomy of Distributed Event-Based Programming Systems. *Computer Journal*, 48(5):602–626, 2005. → pages: 83

[224] Peter Mell and Timothy Grance. The nist definition of cloud computing (draft). *NIST special publication*, 800:145, 2011. → pages: 13 and 17

[225] Mukhtiar Memon, Michael Hafner, and Ruth Breu. SECTISSIMO: A Platform-independent Framework for Security Services. In *Modeling Security Workshop at MODELS '08*, 2008. → pages: 172

[226] Daniel A. Menascé. QoS Issues in Web Services. *IEEE Internet Computing*, 6(6):72–75, 2002. → pages: 94

[227] Tom Mens and Pieter Van Gorp. A Taxonomy of Model Transformation. *Electronic Notes in Theoretical Computer Science*, 152:125–142, 2006. → pages: 141

[228] Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and How to Develop Domain-Specific Languages. *ACM Computing Surveys*, 37(4):316–344, December 2005. → pages: 141

[229] Anton Michlmayr, Florian Rosenberg, Philipp Leitner, and Schahram Dustdar. Advanced event processing and notifica- tions in service runtime environments. In *2nd International Conference on Distributed Event-Based Systems (DEBS)*, pages 115–125, 2008. → pages: 32

[230] Anton Michlmayr, Florian Rosenberg, Philipp Leitner, and Schahram Dustdar. End-to-End Support for QoS-Aware Service Selection, Binding and Mediation in VRESCo. *IEEE Transactions on Services Computing*, 2010. → pages: 16, 47, and 117

[231] Anton Michlmayr, Florian Rosenberg, Christian Platzer, Martin Treiber, and Schahram Dustdar. Towards recovering the broken soa triangle: a software engineering perspective. In *2nd International Workshop on Service Oriented Software Engineering*, pages 22–28. ACM, 2007. → pages: 13

[232] Sushma Mishra and Heinz Roland Weistroffer. A Framework for Integrating Sarbanes-Oxley Compliance into the Systems Development Process. *Communications of the Association for Information Systems (CAIS)*, 20(1):712–727, 2007. → pages: 139

[233] Oliver Moser, Florian Rosenberg, and Schahram Dustdar. Non-intrusive monitoring and service adaptation for ws-bpel. In *17th International Conference on World Wide Web (WWW)*, pages 815–824. ACM, 2008. → pages: 14

[234] Rajeev Motwani, Jennifer Widom, Arvind Arasu, Brian Babcock, Shivnath Babu, Mayur Datar, Gurmeet Manku, Chris Olston, Justin Rosenstein, and Rohit Varma. Query processing, approximation, and resource management in a data stream management system. In *Conference on Innovative Data Systems Research (CIDR)*, 2003. → pages: 83

[235] A. Mourad, S. Ayoubi, H. Yahyaoui, and H. Otrok. New approach for the dynamic enforcement of Web services security. In *8th International Conference on Privacy Security and Trust*, pages 189–196, 2010. → pages: 173

[236] Catherine Moxey, Mike Edwards, Opher Etzion, Mamdouh Ibrahim, Sreekanth Iyer, Hubert Lalanne, Mweene Monze, Marc Peters, Yuri Rabinovich, Guy Sharon, and Kristian Stewart. A conceptual model for event processing systems. *IBM Redguide publication*, 2010. Accessed: 2013-04-02. → pages: 6, 11, 31, 32, 33, and 34

[237] Gero Mühl, Ludger Fiege, and Peter Pietzuch. *Distributed event-based systems*. Springer, 2006. → pages: 1, 11, and 29

[238] Glenford J. Myers, Corey Sandler, and Tom Badgett. *The art of software testing*. Wiley, 2011. → pages: 23

[239] Luis Daniel Navarro, Rémi Douence, and Mario Südholt. Debugging and testing middleware with aspect-based control-flow and causal patterns. In *9th ACM/IFIP/USENIX International Middleware Conference*, pages 183–202, 2008. → pages: 85

[240] Stephen Nelson-Smith. *Test-Driven Infrastructure with Chef*. O'Reilly, 2011. → pages: 8 and 19

[241] Sandvine Intelligent Broadband Networks. Global Internet Phenomena Report, 2013. → pages: 18

[242] Changhai Nie and Hareton Leung. A survey of combinatorial testing. *ACM Computing Surveys*, 43:11:1–11:29, 2011. → pages: 25, 65, and 101

[243] OASIS. eXtensible Access Control Markup Language, 2005. → pages: 173

[244] OASIS. Metadata for the OASIS Security Assertion Markup Language (SAML). `http://docs.oasis-open.org/security/saml/v2.0/saml-metadata-2.0-os.pdf`, 2005. → pages: 159

[245] OASIS. Security Assertion Markup Language. `http://docs.oasis-open.org/security/saml/v2.0/saml-core-2.0-os.pdf`, March 2005. → pages: 148

[246] OASIS. Web Services Business Process Execution Language. `http://docs.oasis-open.org/wsbpel/2.0/OS`, 2007. → pages: 6, 14, 89, and 141

[247] OASIS. WS-SecurityPolicy 1.3. `http://docs.oasis-open.org/ws-sx/ws-security\discretionary{-}{}{}policy/v1.3/os/`, 2009. → pages: 171

[248] Object Management Group. Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, January 2011. → pages: 155

[249] Object Management Group. UML 2.4.1 Superstructure, August 2011. → pages: 24, 141, and 150

[250] Object Management Group (OMG). Business Process Model and Notation (BPMN). `http://www.omg.org/spec/BPMN/`, 2011. → pages: 93

[251] Jeff Offutt and Wuzhi Xu. Generating Test Cases for Web Services Using Data Perturbation. *ACM SIGSOFT Software Engineering Notes*, 29(5), 2004. → pages: 90 and 99

[252] Jefferson Offutt, Shaoying Liu, Aynur Abdurazik, and Paul Ammann. Generating test data from state-based specifications. *Software Testing, Verification and Reliability*, 13(1):25–53, 2003. → pages: 26 and 64

[253] Opscode. Chef. `http://www.opscode.com/chef/`. Accessed: 2013-02-03. → pages: 8, 19, 57, and 84

[254] Opscode. Test Kitchen. https://github.com/opscode/test-kitchen. Visited: 2013-02-06. → pages: 84

[255] Opscode Community. http://community.opscode.com/. Accessed: 2013-02-03. → pages: 20 and 57

[256] Chun Ouyang, Eric Verbeek, Wil van der Aalst, Stephan Breutel, Marlon Dumas, and Arthur HM ter Hofstede. Wofbpel: A tool for automated analysis of bpel processes. In *International Conference Service-Oriented Computing (ICSOC)*, pages 484–489. Springer, 2005. → pages: 14

[257] Federica Paci, Elisa Bertino, and Jason Crampton. An Access-Control Framework for WS-BPEL. *International Journal of Web Services Research*, 5(3):20–43, 2008. → pages: 172

[258] Michael P. Papazoglou, Paolo Traverso, Schahram Dustdar, and Frank Leymann. Service-Oriented Computing: State of the Art and Research Challenges. *Computer*, 40, 2007. → pages: 13 and 89

[259] Mike P Papazoglou, Paolo Traverso, Schahram Dustdar, and Frank Leymann. Service-Oriented Computing: State of the Art and Research Challenges. *IEEE Computer*, 40(11):38–45, 2007. → pages: 135 and 139

[260] Lilia Paradis and Qi Han. A survey of fault management in wireless sensor networks. *Journal of Network and Systems Management*, 15(2):171–190, 2007. → pages: 23

[261] Kostas Patroumpas and Timos Sellis. Window specification over data streams. In *Current Trends in Database Technology (EDBT)*, pages 445–464, 2006. → pages: 11 and 12

[262] Cesare Pautasso, Olaf Zimmermann, and Frank Leymann. Restful web services vs. "big"' web services: making the right architectural decision. In *17th International Conference on World Wide Web (WWW)*, pages 805–814. ACM, 2008. → pages: 14

[263] Marshall Pease, Robert Shostak, and Leslie Lamport. Reaching agreement in the presence of faults. *Journal of the ACM (JACM)*, 27(2):228–234, 1980. → pages: 23

[264] Tan Phan, Jun Han, J.-G. Schneider, T. Ebringer, and T. Rogers. A survey of policy-based management approaches for service oriented systems. In *19th Australian Conference on Software Engineering*, pages 392–401, 2008. → pages: 135

[265] Éric Piel, Alberto Gonzalez-Sanchez, and Hans-Gerhard Gross. Built-in data-flow integration testing in large-scale component-based systems. In *22nd IFIP International Conference on Testing Software and Systems (ICTSS)*, pages 79–94. Springer, 2010. → pages: 134

[266] Peter Pietzuch, Jonathan Ledlie, Jeffrey Shneidman, Mema Roussopoulos, Matt Welsh, and Margo Seltzer. Network-aware operator placement for stream-processing systems. In *International Conference on Data Engineering (ICDE)*, 2006. → pages: 82

[267] Peter R. Pietzuch and Jean M. Bacon. Hermes: a distributed event-based middleware architecture. In *22nd International Conference on Distributed Computing Systems (ICDCS) Workshops*, pages 611–618, 2002. → pages: 31

[268] Alexander Pretschner. Model-based testing. In *27th International Conference on Software Engineering (ICSE)*, pages 722–723, 2005. → pages: 85

[269] Puppet Labs. Puppet. `http://puppetlabs.com/`. Accessed: 2013-02-03. → pages: 19 and 84

[270] John Ross Quinlan. Induction of decision trees. *Machine Learning*, 1:81–106, 1986. → pages: 108

[271] John Ross Quinlan. *C4.5: programs for machine learning*. Morgan Kaufmann Publishers Inc., 1993. → pages: 117

[272] Peter J. Ramadge and W. Murray Wonham. The control of discrete event systems. *Proceedings of the IEEE*, 77(1), 1989. → pages: 32

[273] Chittoor V. Ramamoorthy, Siu-Bun Ho, and Wen-Tsuen Chen. On the automated generation of program test data. *IEEE Transactions on Software Engineering*, SE-2(4):293–300, 1976. → pages: 99

[274] Brian Randell. System structure for software fault tolerance. *Software Engineering, IEEE Transactions on*, (2):220–232, 1975. → pages: 23

[275] Sandra Rapps and Elaine J. Weyuker. Selecting software test data using data flow information. *IEEE Transactions on Software Engineering*, 11:367–375, 1985. → pages: 132

[276] Manos Renieris and Steven P. Reiss. Fault localization with nearest neighbor queries. In *18th IEEE International Conference on Automated Software Engineering (ASE)*, pages 30–39, 2003. → pages: 134 and 135

[277] Debra J. Richardson, Stephanie Leif Aha, and T. Owen O'Malley. Specification-based test oracles for reactive systems. In *14th International Conference on Software Engineering (ICSE)*, pages 105–118, 1992. → pages: 96, 97, and 117

[278] Prasan Roy, S. Seshadri, S. Sudarshan, and Siddhesh Bhobe. Efficient and extensible algorithms for multi query optimization. In *ACM SIGMOD International Conference on Management of Data*, pages 249–260, 2000. → pages: 82

[279] Anne Rozinat and Wil van der Aalst. Conformance testing: Measuring the fit and appropriateness of event logs and process models. In *4th International Conference on Business Process Management - Workshops*, pages 163–176. Springer, 2006. → pages: 44

[280] Szabolcs Rozsnyai, Aleksander Slominski, and Geetika T. Lakshmanan. Discovering event correlation rules for semi-structured business processes. In *5th DEBS*, 2011. → pages: 32 and 34

[281] Linnyer Beatrys Ruiz, Isabela G. Siqueira, Leonardo B. e Oliveira, Hao Chi Wong, José Marcos S. Nogueira, and Antonio A. F. Loureiro. Fault management in event-driven wireless sensor networks. In *7th ACM International Symposium on Modeling Analysis and Simulation of Wireless and Mobile Systems (MSWiM)*, pages 149–156, 2004. → pages: 23 and 31

[282] Hazlifah Mohd Rusli, Suhaimi Ibrahim, and Mazidah Puteh. Testing web services composition: A mapping study. *Communications of the IBIMA*, pages 34–48, 2011. → pages: 132

[283] Wasim Sadiq and Maria Orlowska. On business process model transformations. In Alberto Laender, Stephen Liddle, and Veda Storey, editors, *Conceptual Modeling — ER 2000*, volume 1920, pages 47–104. Springer Berlin / Heidelberg, 2000. → pages: 173

[284] Ravi S. Sandhu, Edward J. Coyne, Hal L. Feinstein, and Charles E. Youman. Role-based access control models. *Computer*, 29(2):38–47, 1996. → pages: 2 and 139

[285] Benjamin Satzger, Waldemar Hummer, Christian Inzinger, Philipp Leitner, and Schahram Dustdar. Winds of change: From vendor lock-in to the meta cloud. *IEEE Internet Computing*, 17(1):69–73, 2013. → pages: 225

[286] Benjamin Satzger, Waldemar Hummer, Philipp Leitner, and Schahram Dustdar. Esc: Towards an elastic stream computing platform for the cloud. In *IEEE International Conference on Cloud Computing (CLOUD)*, pages 348–355, 2011. → pages: 228

[287] Benjamin Satzger, Andreas Pietzowski, Wolfgang Trumler, and Theo Ungerer. A new adaptive accrual failure detector for dependable distributed systems. In *ACM Symposium on Applied Computing (SAC)*, pages 551–555, 2007. → pages: 135

[288] Andreas Schaefer, Marc Reichenbach, and Dietmar Fey. Continuous Integration and Automation for Devops. *IAENG Transactions on Engineering Technologies*, 170:345–358, 2013. → pages: 19

[289] Michael Schäfer, Peter Dolog, and Wolfgang Nejdl. An environment for flexible advanced compensations of web service transactions. *ACM Transactions on the Web*, 2(2):14:1–14:36, 2008. → pages: 174

[290] Daniel Schall, Florian Skopik, and Schahram Dustdar. Expert discovery and interactions in mixed service-oriented systems. *IEEE Transactions on Services Computing*, 99(PrePrints), 2011. → pages: 136

[291] August-Wilhelm Scheer, Oliver Thomas, and Otmar Adam. *Process Modeling using Event-Driven Process Chains*, pages 119–145. John Wiley & Sons, Inc., 2005. → pages: 170

[292] Sigrid Schefer, Mark Strembeck, and Jan Mendling. Checking satisfiability aspects of binding constraints in a business process context. In *Workshop on Workflow Security Audit and Certification (WfSAC)*. Springer, August 2011. → pages: 166, 169, and 174

[293] Sigrid Schefer, Mark Strembeck, Jan Mendling, and Anne Baumgrass. Detecting and resolving conflicts of mutual-exclusion and binding constraints in a business process context. In *19th International Conference on Cooperative Information Systems (CoopIS'11)*. Springer, 2011. → pages: 166, 169, and 174

[294] Sigrid Schefer-Wenzl and Mark Strembeck. A UML Extension for Modeling Break-Glass Policies. In *5th International Workshop on Enterprise Modelling and Information Systems Architectures (EMISA)*, 2012. → pages: 170

[295] Josef Schiefer, Gerd Saurer, and Alexander Schatten. Testing event-driven business processes. *Journal of Computers*, 1(7):69–80, 2006. → pages: 134

[296] Douglas C. Schmidt. Guest editor's introduction: Model-driven engineering. *IEEE Computer*, 39(2):25–31, 2006. → pages: 141

[297] Schultz-Møller, Nicholas Poul and Migliavacca, Matteo and Pietzuch, Peter. Distributed Complex Event Processing with Query Rewriting. In *International Conference on Distributed Event-Based Systems (DEBS)*, pages 4:1–4:12. ACM, 2009. → pages: 31, 42, and 43

[298] Toby Segaran. *Programming Collective Intelligence*. O'Reilly Media, 2007. → pages: 109

[299] Bran Selic. The Pragmatics of Model-Driven Development. *IEEE Software*, 20(5):19–25, 2003. → pages: 141

[300] Shane Sendall and Wojtek Kozaczynski. Model Transformation: The Heart and Soul of Model-Driven Software Development. *IEEE Software*, 20(5), 2003. → pages: 141 and 173

[301] Sangeetha Seshadri, Vibhore Kumar, and Brian F. Cooper. Optimizing multiple queries in distributed data stream systems. In *Int. Conference on Data Engineering, Workshops*, 2006. → pages: 82

[302] Sangeetha Seshadri, Vibhore Kumar, Brian F. Cooper, and Ling Liu. Optimizing multiple distributed stream queries using hierarchical network partitions. In *IEEE International Parallel and Distributed Processing Symposium*, pages 1–10, 2007. → pages: 82

[303] Mehul A. Shah, Joseph M. Hellerstein, and Eric Brewer. Highly available, fault-tolerant, parallel dataflows. In *ACM SIGMOD International Conference on Management of Data*, pages 827–838, 2004. → pages: 42

[304] Guy Sharon and Opher Etzion. Event-processing network model and implementation. *IBM Systems Journal*, 47(2):321–334, 2008. → pages: 6, 11, and 32

[305] Halvard Skogsrud, Boualem Benatallah, and Fabio Casati. Model-Driven Trust Negotiation for Web Services. *IEEE Internet Computing*, 7:45–52, 2003. → pages: 171

[306] Sergey Smirnov, Hajo A. Reijers, and Mathias Weske. A semantic approach for business process model abstraction. In *23rd International Conference on Advanced information Systems engineering (CAiSE)*, pages 497–511, 2011. → pages: 172

[307] Stephen Soltesz, Herbert Pötzl, Marc E Fiuczynski, Andy Bavier, and Larry Peterson. Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 275–287. ACM, 2007. → pages: 17

[308] D. Spinellis. Notable design patterns for domain-specific languages. *Journal of Systems and Software*, 56(1):91–99, 2001. → pages: 141

[309] Thomas Stahl and Markus Völter. *Model-Driven Software Development*. John Wiley & Sons, 2006. → pages: 141

[310] William Stallings. *Cryptography and Network Security*. Pearson Education India, 2006. → pages: 6

[311] Malgorzata Steinder and Adarshpal S. Sethi. A survey of fault localization techniques in computer networks. *Science of Computer Programming*, 53(2):165–194, 2004. → pages: 38 and 84

[312] Robert Stephens. A survey of stream processing. *Acta Informatica*, 34:491–541, 1997. → pages: 1, 31, 35, and 36

[313] Mark Strembeck. A Role Engineering Tool for Role-Based Access Control. In *3rd Symposium on Requirements Engineering for Information Security*, 2005. → pages: 139

[314] Mark Strembeck. Scenario-driven Role Engineering. *IEEE Security & Privacy*, 8(1):28–35, January 2010. → pages: 139

[315] Mark Strembeck and Jan Mendling. Generic Algorithms for Consistency Checking of Mutual-Exclusion and Binding Constraints in a Business Process Context. In *18th International Conference on Cooperative Information Systems (CoopIS)*, October 2010. → pages: 9, 140, 148, 152, 165, 166, and 169

[316] Mark Strembeck and Jan Mendling. Modeling Process-related RBAC Models with Extended UML Activity Models. *Information and Software Technology*, 53(5):456–483, May 2011. → pages: 139, 141, 142, 144, 146, 153, 162, 165, 166, and 171

[317] Mark Strembeck and Gustaf Neumann. An Integrated Approach to Engineer and Enforce Context Constraints in RBAC Environments. *ACM Transactions on Information and System Security (TISSEC)*, 7(3):392–427, 2004. → pages: 171

[318] Mark Strembeck and Uwe Zdun. An Approach for the Systematic Development of Domain-Specific Languages. *Software: Practice and Experience*, 39(15), October 2009. → pages: 141

[319] Ya-Yunn Su, Mona Attariyan, and Jason Flinn. AutoBash: improving configuration management with operating system causality analysis. In *21st ACM Symposium on Operating Systems Principles*, 2007. → pages: 85

[320] Kaijun Tan, Jason Crampton, and Carl A. Gunter. The Consistency of Task-Based Authorization Constraints in Workflow Systems. In *17th IEEE Workshop on Computer Security Foundations (CSFW)*, pages 155–169, June 2004. → pages: 140, 166, 168, and 169

[321] Andrew S Tanenbaum and Maarten Van Steen. *Distributed Systems: Principles and Paradigms*, volume 2. Prentice Hall, 2002. → pages: 22

[322] Abbas Tarhini, Hacène Fouchal, and Nashat Mansour. A simple approach for testing web service based applications. In *International Workshop on Innovative Internet Community Systems*, 2005. → pages: 133

[323] Richard N. Taylor, Nenad Medvidovic, Kenneth M. Anderson, E. James Whitehead, Jr., Jason E. Robbins, Kari A. Nies, Peyman Oreizy, and Deborah L. Dubrow. A Component- and Message-Based Architectural Style for GUI Software. *IEEE Transactions on Software Engineering*, 22(6):390–406, 1996. → pages: 31

[324] TeleManagement Forum. Case study handbook, December 2009. → pages: 90

[325] Gerard J. Tellis. The price elasticity of selective demand: A meta-analysis of econometric models of sales. *Journal of Marketing Research*, pages 331–341, 1988. → pages: 18

[326] Jean-Yves Tigli, Stephane Lavirotte, Gaëtan Rey, Vincent Hourdin, Daniel Cheung-Foo-Wo, Eric Callegari, and Michel Riveill. WComp middleware for ubiquitous computing: Aspects and composite event-based Web services. *Annales des Télécommunications*, 64(3-4):197–214, 2009. → pages: 31

[327] Sameer Tilak, Nael B. Abu-Ghazaleh, and Wendi Heinzelman. A taxonomy of wireless micro-sensor network models. *ACM SIGMOBILE Mobile Computing and Communications Review*, 6(2):28–36, 2002. → pages: 43

[328] Torry Harris Business Solution. SOA Test Methodology. `http://thbs.com/pdfs/SOA_Test_Methodology.pdf`, 2007. Accessed: 2012-01-14. → pages: 89

[329] Steve Traugott. Why order matters: Turing equivalence in automated systems administration. In *16th Conference on Systems Administration (LISA)*, pages 99–120, 2002. → pages: 84

[330] Hong-Linh Truong, Schahram Dustdar, Georgiana Copil, Alessio Gambi, Waldemar Hummer, Duc-Hung Le, and Daniel Moldovan. CoMoT - A Platform-as-a-Service for Elasticity in the Cloud. In *IEEE International Workshop on the Future of PaaS*, 2014. → pages: 228

[331] Wei-Tek Tsai, Yinong Chen, Zhibin Cao, Xiaoying Bai, Hai Huang, and Ray Paul. Testing Web Services Using Progressive Group Testing. In *Content Computing*, pages 314–322. Springer, 2004. → pages: 16 and 90

[332] Wei-Tek Tsai, Yinong Chen, Ray Paul, Ning Liao, and Hai Huang. Cooperative and Group Testing in Verification of Dynamic Composite Web Services. In *28th International Computer Software and Applications Conference (COMPSAC)*, pages 170–173, 2004. → pages: 14, 16, and 90

[333] Fang Tu, K.R. Pattipati, S. Deb, and V.N. Malepati. Computationally efficient algorithms for multiple fault diagnosis in large graph-based systems. *IEEE Transactions on Systems, Man and Cybernetics*, 33(1):73–85, 2003. → pages: 134

[334] Mark Utting and Bruno Legeard. *Practical model-based testing: a tools approach*. Morgan Kaufmann, 2010. → pages: 24

[335] Mark Utting, Alexander Pretschner, and Bruno Legeard. A taxonomy of model-based testing approaches. *Software Testing, Verification and Reliability*, 22(5):297–312, 2012. → pages: 24, 25, and 58

[336] Wil van der Aalst. Formalization and verification of event-driven process chains. *Information & Software Technology*, 41(10):639–650, 1999. → pages: 45

[337] Wil van der Aalst and Kristian Bisgaard Lassen. Translating unstructured workflow processes to readable bpel: Theory and implementation. *Information and Software Technology (IST)*, 50(3):131–159, 2008. → pages: 14

[338] Wil van der Aalst, Arthur HM Ter Hofstede, and Mathias Weske. *Business process management: A survey*. Springer, 2003. → pages: 1

[339] Wil van der Aalst, Ton Weijters, and Laura Maruster. Workflow mining: discovering process models from event logs. *IEEE Transactions on Knowledge and Data Engineering*, 16(9):1128–1142, 2004. → pages: 32, 34, and 44

[340] Sander van der Burg and Eelco Dolstra. Automating system tests using declarative virtual machines. In *21st Int. Symposium on Software Reliability Engineering*, 2010. → pages: 85

[341] Luis M. Vaquero, Luis Rodero-Merino, Juan Caceres, and Maik Lindner. A break in the clouds: towards a cloud definition. *ACM SIGCOMM Computer Communication Review*, 39(1):50–55, 2008. → pages: 17

[342] Seth Vargo. Chefspec. `http://code.sethvargo.com/chefspec/`. Accessed: 2013-02-03. → pages: 84

[343] Vitria. Complex Event Processing for Operational Intelligence. `http://www.club-bpm.com/Documentos/DocProd00015.pdf`, 2010. Accessed: 2013-10-05. → pages: 82

[344] Michael von Riegen, Martin Husemann, Stefan Fink, and Norbert Ritter. Rule-based coordination of distributed web service transactions. *IEEE Transactions on Services Computing*, 3(1):60–72, 2010. → pages: 174

[345] Jacques Wainer, Paulo Barthelmess, and Akhil Kumar. W-RBAC - A Workflow Security Model Incorporating Controlled Overriding of Constraints. *International Journal of Cooperative Information Systems*, 12(4):455–485, December 2003. → pages: 139

[346] Carl A Waldspurger. Memory resource management in vmware esx server. *ACM SIGOPS Operating Systems Review*, 36(SI):181–194, 2002. → pages: 17

[347] Fusheng Wang, Shaorong Liu, Peiya Liu, and Yijian Bai. Bridging Physical and Virtual Worlds: Complex Event Processing for RFID Data Streams. In *10th International Conference on Extending Database Technology (EDBT)*, pages 588–607, 2006. → pages: 31 and 43

[348] Nanbor Wang, Douglas C Schmidt, Aniruddha Gokhale, Christopher D Gill, Balachandran Natarajan, Craig Rodrigues, Joseph P Loyall, and Richard E Schantz. Total quality of service provisioning in middleware and applications. *Microprocessors and Microsystems*, 27(2):45–54, 2003. → pages: 3

[349] Qihua Wang and Ninghui Li. Satisfiability and resiliency in workflow authorization systems. *ACM Transactions on Information and System Security (TISSEC)*, 13(4):40:1–40:35, December 2010. → pages: 165 and 166

[350] Barbara Weber, Manfred Reichert, Werner Wild, and Stefanie Rinderle. Balancing flexibility and security in adaptive process management systems. In *On the Move to Meaningful Internet Systems 2005: CoopIS, DOA, and ODBASE*, volume 3760, pages 59–76. Springer Berlin / Heidelberg, 2005. → pages: 174

[351] Utz Westermann and Ramesh Jain. Toward a common event model for multimedia applications. *IEEE MultiMedia*, pages 19–29, 2007. → pages: 6, 32, and 83

[352] Elaine J. Weyuker. Testing component-based software: a cautionary tale. *IEEE Software*, 15(5):54–59, 1998. → pages: 134

[353] Elaine J. Weyuker and Bingchiang Jeng. Analyzing partition testing strategies. *IEEE Transactions on Software Engineering (TSE)*, 17(7):703–711, 1991. → pages: 108

[354] Andrew Whitaker, Richard Cox, and Steven Gribble. Configuration debugging as search: finding the needle in the haystack. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 6–6, 2004. → pages: 85

[355] Matthias Wieland, Daniel Martin, Oliver Kopp, and Frank Leymann. SOEDA: A Methodology for Specification and Implementation of Applications on a Service-Oriented Event-Driven Architecture. In *12th International Conference on Business Information Systems (BIS)*. Springer, 2009. → pages: 32

[356] Christian Wolter, Michael Menzel, Andreas Schaad, Philip Miseldine, and Christoph Meinel. Model-driven business process security requirement specification. *Journal of Systems Architecture*, 55:211–223, 2009. → pages: 155 and 172

[357] Christian Wolter, Andreas Schaad, and Christoph Meinel. Task-based entailment constraints for basic workflow patterns. In *13th ACM Symposium on Access Control Models and Technologies (SACMAT)*, pages 51–60. ACM, June 2008. → pages: 2, 6, 9, and 140

[358] W. Eric Wong and Vidroha Debroy. Software fault localization. *IEEE Reliability Society 2009 Annual Technology Report*, 2009. → pages: 134 and 135

[359] World Wide Web Consortium (W3C). Web Services Addressing. `http://www.w3.org/Submission/WS-Addressing/`. → pages: 49 and 99

[360] World Wide Web Consortium (W3C). Web Services Eventing. `http://www.w3.org/Submission/WS-Eventing/`. → pages: 68

[361] World Wide Web Consortium (W3C). XQuery 3.0: An XML Query Language. `http://www.w3.org/TR/xquery-30/`. → pages: 66 and 161

[362] World Wide Web Consortium (W3C). XML Path Language (XPath). `http://www.w3.org/TR/xpath/`, 1999. → pages: 14 and 93

[363] World Wide Web Consortium (W3C). Web Services Description Language (WSDL) 1.1. `http://www.w3.org/TR/wsdl`, 2001. → pages: 99

[364] World Wide Web Consortium (W3C). Web Services Activity. `http://www.w3.org/2002/ws/`, 2002. → pages: 1, 6, 14, 20, 68, 89, and 139

[365] World Wide Web Consortium (W3C). Web Services Description Language (WSDL) Version 2.0 Part 0: Primer. `http://www.w3.org/TR/2006/CR-wsdl20-primer-20060327/`, 2006. → pages: 90

[366] World Wide Web Consortium (W3C). SOAP Messaging Framework. `http://www.w3.org/TR/soap12-part1/`, 2007. → pages: 154

[367] World Wide Web Consortium (W3C). XSL Transformations (XSLT) Version 2.0. `http://www.w3.org/TR/xslt20/`, 2007. → pages: 160

[368] World Wide Web Consortium (W3C). XML Signature Syntax and Processing. `http://www.w3.org/TR/xmldsig-core/`, 2008. → pages: 170

[369] Franz Wotawa, Marco Schulz, Ingo Pill, Seema Jehan, Philipp Leitner, Waldemar Hummer, Stefan Schulte, Philipp Hoenisch, and Schahram Dustdar. Fifty Shades of Grey in SOA Testing. In *Workshop on Advances in Model Based Testing, co-located with ICST'13*, pages 154–157, 2013. → pages: 228

[370] Eugene Wu, Yanlei Diao, and Shariq Rizvi. High-performance complex event processing over streams. In *SIGMOD International Conference On Management of Data*, 2006. → pages: 42, 43, and 83

[371] Wuzhi Xu, Jeff Offutt, and Juan Luo. Testing Web services by XML perturbation. In *16th IEEE International Symposium on Software Reliability Engineering (ISSRE)*, 2005. → pages: 14

[372] Lamia Youseff, Maria Butrico, and Dilma Da Silva. Toward a unified ontology of cloud computing. In *Grid Computing Environments Workshop, 2008. GCE'08*, pages 1–10. IEEE, 2008. → pages: 13 and 17

[373] Tao Yu, Yue Zhang, and Kwei-Jay Lin. Efficient algorithms for Web services selection with end-to-end QoS constraints. *ACM Transactions on the Web*, 1(1), 2007. → pages: 124

[374] Diego Zamboni. *Learning CFEngine 3: Automated system administration for sites of any size*. O'Reilly Media, Inc., 2012. → pages: 84

[375] Uwe Zdun and Mark Strembeck. Modeling Composition in Dynamic Programming Environments with Model Transformations. In *5th Int. Symposium on Software Composition*, 2006. → pages: 141

[376] Uwe Zdun and Mark Strembeck. Reusable Architectural Decisions for DSL Design: Foundational Decisions in DSL Projects. In *14th European Conference on Pattern Languages of Programs (EuroPLoP)*, 2009. → pages: 141 and 147

[377] Yongyan Zheng, Jiong Zhou, and Paul Krause. Analysis of BPEL Data Dependencies. In *33rd Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, 2007. → pages: 94

[378] Pin Zhou, Binny Gill, Wendy Belluomini, and Avani Wildani. Gaul: Gestalt analysis of unstructured logs for diagnosing recurring problems in large enterprise storage systems. In *29th IEEE Symposium on Reliable Distributed Systems (SRDS)*, pages 148–159, 2010. → pages: 135

[379] Hong Zhu, Patrick AV Hall, and John HR May. Software unit test coverage and adequacy. *ACM Computing Surveys*, 29(4):366–427, 1997. → pages: 2, 25, and 26

[380] Yali Zhu, Elke Rundensteiner, and George Heineman. Dynamic plan migration for continuous queries over data streams. In *SIGMOD Int. Conf. on Management of Data*, 2004. → pages: 82

[381] Zhengdong Zhu, Yahong Hu, Xuehan Dong, and Zengzhi Li. A minimum coverage method for web service composition. In *5th IEEE International Conference on Fuzzy Systems and Knowledge Discovery (FSKD)*, pages 468–472, 2008. → pages: 133

APPENDIX A

Code Listings

A.1 RBAC DSL Statements for Patient Examination Process

Listing A.1 contains the complete access control configuration of the Patient Examination scenario process (two involved hospitals), expressed using RBAC DSL statements.

```
1 RESOURCE   PatientService1 "http://hospital1.com/patients"
2 RESOURCE   PatientService2 "http://hospital2.com/patients"
3 OPERATION retrieveData
4 OPERATION makeAssignment
5 OPERATION getHistory
6 OPERATION getOpinion
7 OPERATION queryPartner
8 OPERATION makeDecision
9 ROLE       Staff
10 ROLE      Physician
11 ROLE      Patient
12 INHERIT   Staff Physician
13 SUBJECT   John
14 SUBJECT   Jane
15 SUBJECT   Bob
16 SUBJECT   Alice
17 ASSIGN    John Staff
18 ASSIGN    Jane Physician
19 ASSIGN    Bob Physician
20 ASSIGN    Alice Patient
21  # Web service operation permissions (hospital 1)
22 PERMIT    Staff      retrieveData    PatientService1
23 PERMIT    Staff      makeAssignment  PatientService1
24 PERMIT    Physician  getHistory      PatientService1
25 PERMIT    Patient    getHistory      PatientService1
26 PERMIT    Physician  getOpinion      PatientService1
27 PERMIT    Patient    queryPartner    PatientService1
28 PERMIT    Physician  makeDecision    PatientService1
29  # Web service operation permissions (hospital 2)
30 PERMIT    Staff      retrieveData    PatientService2
31 PERMIT    Staff      makeAssignment  PatientService2
32 PERMIT    Physician  getHistory      PatientService2
33 PERMIT    Patient    getHistory      PatientService2
```

```
34 PERMIT     Physician getOpinion      PatientService2
35 PERMIT     Patient   queryPartner    PatientService2
36 PERMIT     Physician makeDecision    PatientService2
37  # 'task' to 'service operation' bindings (hospital 1)
38 TASK       GetPersonalData retrieveData    PatientService1
39 TASK       AssignPhysician makeAssignment PatientService1
40 TASK       GetCriticalHistory getHistory  PatientService1
41 TASK       GetExpertOpinion getOpinion     PatientService1
42 TASK       GetPartnerHistory queryPartner PatientService1
43 TASK       DecideOnTreatment makeDecision PatientService1
44  # 'task' to 'service operation' bindings (hospital 2)
45 TASK       GetPersonalData retrieveData    PatientService2
46 TASK       AssignPhysician makeAssignment PatientService2
47 TASK       GetCriticalHistory getHistory  PatientService2
48 TASK       GetExpertOpinion getOpinion     PatientService2
49 TASK       GetPartnerHistory queryPartner PatientService2
50 TASK       DecideOnTreatment makeDecision PatientService2
51  # task-based entailment constraints
52 RBIND      GetPersonalData AssignPhysician
53 DME        GetCriticalHistory GetExpertOpinion
54 SBIND      GetCriticalHistory DecideOnTreatment
55 SBIND      GetPartnerHistory GetPartnerHistory
56 SME        GetExpertOpinion GetPartnerHistory
```

Listing A.1: Exemplary RBAC DSL Statements for Hospital Scenario

A.2 XQuery Assertion Expressions for Enforcing Access Constraints

Listing A.2 prints the constraint enforcement queries, expressed as XQuery assertion statements that are expected to always yield a boolean *true* value. Lines 1-7 contain an excerpt of the constraint definitions of the scenario introduced in Section 5.2.1. For instance, the two tasks named *Get_Personal_Data* and *Assign_Physician* are in a role-binding relationship and hence combined in an element `rbind`. Moreover, the code binds the log elements from Listing 5.3 to the variable `$logs` (line 8). Finally, Listing A.2 contains the four XQuery expressions used for enforcing constraints concerning SME tasks (lines 11-16), DME tasks (lines 19-23), subject-bindings (lines 26-30) and role-bindings (lines 33-37).

The four expressions use universal quantification (`every...in...satisfies`) to express assertions about pairs of tasks defined in the constraints list. The variables $*t1* and $*t2* refer to the names of the respective tasks. The query for SME loops over all pairs of SME tasks and ensures that the logs do not contain invocations for both tasks that use the same subject or the same role. The DME query tasks is similar, with the difference that only the subject is queried and additionally the *instanceID* attribute of the log entries is considered. Subject-binding is checked by ensuring that for all log entries of a particular process instance two tasks $*t1* and $*t2* are executed by the same subject. The role-binding query works analogously, but instead of using the *subject* attribute, here we require the *role* attribute to match for all *rbind*-connected tasks that occur in the same process instance.

```
let $cons := <constraints>
  <rbind><task>Get_Personal_Data</task> <task>Assign_Physician</task></rbind>
  <sbind><task>Decide_On_Treatment</task> <task>Get_Critical_History</task></sbind>
  <dme><task>Get_Critical_History</task> <task>Get_Expert_Opinion</task></dme>
  <sme><task>Get_Expert_Opinion</task> <task>Get_Patient_History</task></sme>
  ...
</constraints>
let $logs := //log

SME Tasks:
every $sme in $cons/sme, $t1 in $sme/task, $t2 in $sme/task satisfies (
  $t1 = $t2 or
  (every $i in $logs[@taskName=$t1] satisfies (
    not(exists($logs[@taskName=$t2][@role=$i/@role]))
      and
    not(exists($logs[@taskName=$t2][@subject=$i/@subject])))))

DME Tasks:
every $dme in $cons/dme, $t1 in $dme/task, $t2 in $dme/task satisfies (
  $t1 = $t2 or
  (every $i in $logs[@taskName=$t1] satisfies (
    not(exists(
      $logs[@taskName=$t2][@subject=$i/@subject][@instanceID=$i/@instanceID])))))

Subject-Binding:
every $sbind in $cons/sbind, $t1 in $sbind/task, $t2 in $sbind/task satisfies (
  $t1 = $t2 or
  (every $i in $logs[@taskName=$t1] satisfies (
    every $j in $logs[@taskName=$t2][@instanceID=$i/@instanceID]
      satisfies $i/@subject = $j/@subject)))

Role-Binding:
every $rbind in $cons/rbind, $t1 in $rbind/task, $t2 in $rbind/task satisfies (
  $t1 = $t2 or
  (every $i in $logs[@taskName=$t1] satisfies (
    every $j in $logs[@taskName=$t2][@instanceID=$i/@instanceID]
      satisfies $i/@role = $j/@role)))
```

Listing A.2: XQuery Assertion Expressions for Enforcing Access Constraints

APPENDIX B

Curriculum Vitae

Personal Information

Name	Waldemar Hummer
Address	Josefstädter Straße 14/2/58, 1080 Wien, Austria
Email	hummer@dsg.tuwien.ac.at
Web	http://dsg.tuwien.ac.at/staff/hummer
Date & Place of Birth	April 12, 1986, Hall in Tirol, Austria
Citizenship	Austrian

Work Experience

October 2013 – ongoing	Researcher within the **Pacific Controls Cloud Computing Lab**
January 2010 – ongoing	Research Assistant at **Vienna University of Technology**
Oct. 2010 – Oct. 2013	Researcher in the FP7 Project **INDENICA** (http://indenica.eu)
June – September 2012	Research Internship at **IBM T.J. Watson** Research Center, NY, US
July – September 2010	Visiting Researcher at **IBM Haifa** Research Labs, Haifa, Israel
June – September 2008	R&D Internship at **Bloomberg L.P.**, London, UK
2008 – 2009	Teaching Assistant at **Vienna University of Technology**
June 2007 – Sept. 2007	Student Researcher at **University of Innsbruck**, Quality Engineering Group (head: Univ.-Prof. Dr. Ruth Breu)
Oct. 2006 – March 2007	Software Engineer for **Arctis GmbH**, subcontractor of **Bayerische Landesbank** (BayernLB), Germany

Education and Training

2010 - 2013	Ph.D. in Computer Science, Vienna University of Technology
2007 - 2009	M.S. in Computer Science, Vienna University of Technology
2007 - 2010	B.S. in Business Administration, Vienna University of Economics and Business
2004 - 2007	B.S. in Computer Science, University of Innsbruck

Teaching

Bachelor Level Courses	• Distributed Systems Lab (184.167 - UE 2.0 - *Verteilte Systeme*) • Project Lab Work (184.230 - PR 4.0 - *Projektpraktikum*)
Master Level Courses	• Distributed Systems Technologies (184.260 - VU 4.0) • Software Architectures (184.159 - VU 2.0) • Distributed Systems Engineering (184.159 - VU 2.0) • Project Lab Software Engineering & Internet Computing (184.715 - PR 6.0)
Co-Supervised Theses	• Eduard Szente: *Efficient Computation of Web Services API Coverage Metrics* • Andrea Floh: *Dependable Event Processing over High Volume Data Streams* • Michael Strasser: *Cloud-Based Monitoring and Simulation for Fault-Tolerant Event Processing Platforms*

Professional Activities

Reviewer, Program Committee Member (Excerpt)	• 9th International Conference on Internet and Web Applications and Services (ICIW) 2014 • 1st International Workshop on Monitoring and Prediction of Cloud Services (MoPoC), co-located with ICSOC 2012 • 2nd IEEE International Conference on Computer & Communication Technology (ICCCT) 2011 • 3rd ServiceWave Conference 2010 • 8th IEEE European Conference on Web Services (ECOWS) 2010 • IEEE Transactions on Services Computing • IEEE Transactions on Software Engineering • ACM Transactions on the Web • IEEE Software • IEEE Internet Computing • Engineering Applications of Artificial Intelligence (Elsevier)

Publications

Journal Papers

1. Waldemar Hummer, Patrick Gaubatz, Mark Strembeck, Uwe Zdun, and Schahram Dustdar. Enforcement of Entailment Constraints in Distributed Service-Based Business Processes. *Information and Software Technology (IST)*, 55(11):1884–1903, 2013

2. Waldemar Hummer, Orna Raz, Onn Shehory, Philipp Leitner, and Schahram Dustdar. Testing of Data-Centric and Event-Based Dynamic Service Compositions. *Software Testing, Verification and Reliability (STVR)*, 23(6):465–497, 2013

3. Waldemar Hummer, Benjamin Satzger, and Schahram Dustdar. Elastic Stream Processing in the Cloud. *Wiley Interdisciplinary Rewiews: Data Mining and Knowledge Discovery*, 3(5):333–345, 2013

4. Waldemar Hummer, Philipp Leitner, and Schahram Dustdar. SEPL – a domain-specific language and execution environment for protocols of stateful Web services. *Distributed and Parallel Databases*, 29(4):277–307, 2011

5. Alessio Gambi, Waldemar Hummer, Hong-Linh Truong, and Schahram Dustdar. Testing elastic computing systems. *IEEE Internet Computing*, 17(6):76–82, 2013

6. Christian Inzinger, Waldemar Hummer, Benjamin Satzger, Philipp Leitner, and Schahram Dustdar. Generic Event-Based Monitoring and Adaptation Methodology for Heterogeneous Distributed Systems. *Software: Practice and Experience*, 2014

7. Philipp Leitner, Waldemar Hummer, and Schahram Dustdar. Cost-based optimization of service compositions. *IEEE Transactions on Services Computing (TSC)*, 6(2):239–251, 2013

8. Philipp Leitner, Johannes Ferner, Waldemar Hummer, and Schahram Dustdar. Data-driven and automated prediction of service level agreement violations in service compositions. *Distributed and Parallel Databases*, 31(3):447–470, 2013

9. Benjamin Satzger, Waldemar Hummer, Christian Inzinger, Philipp Leitner, and Schahram Dustdar. Winds of change: From vendor lock-in to the meta cloud. *IEEE Internet Computing*, 17(1):69–73, 2013

Conference and Workshop Papers

1. Waldemar Hummer, Florian Rosenberg, Fábio Oliveira, and Tamar Eilam. Testing idempotence for infrastructure as code. In *14th ACM/IFIP/USENIX Middleware Conference*, pages 368–388, 2013. **Best Student Paper Award**

2. Waldemar Hummer, Florian Rosenberg, Fábio Oliveira, and Tamar Eilam. Automated testing of chef automation scripts. In *ACM/IFIP/USENIX Middleware Conference (tool demo track)*, 2013

3. Waldemar Hummer, Christian Inzinger, Philipp Leitner, Benjamin Satzger, and Schahram Dustdar. Deriving a unified fault taxonomy for distributed event-based systems. In *6th ACM International Conference on Distributed Event-Based Systems (DEBS)*, pages 167–178, 2012

4. Waldemar Hummer, Orna Raz, Onn Shehory, Philipp Leitner, and Schahram Dustdar. Test coverage of data-centric dynamic compositions in service-based systems. In *4th International Conference on Software Testing, Verification and Validation (ICST'11)*, pages 40–49, 2011

5. Waldemar Hummer, Philipp Leitner, and Schahram Dustdar. WS-Aggregation: Distributed Aggregation of Web Services Data. In *ACM Symposium On Applied Computing (SAC)*, pages 1590–1597, 2011

6. Waldemar Hummer, Patrick Gaubatz, Mark Strembeck, Uwe Zdun, and Schahram Dustdar. An integrated approach for identity and access management in a SOA context. In *16th ACM Symposium on Access Control Models and Technologies (SACMAT)*, pages 21–30, 2011

7. Waldemar Hummer, Benjamin Satzger, Philipp Leitner, Christian Inzinger, and Schahram Dustdar. Distributed Continuous Queries Over Web Service Event Streams. In *7th IEEE International Conference on Next Generation Web Services Practices (NWeSP)*, pages 176–181, 2011

8. Waldemar Hummer, Philipp Leitner, Benjamin Satzger, and Schahram Dustdar. Dynamic migration of processing elements for optimized query execution in event-based systems. In *1st International Symposium on Secure Virtual Infrastructures (DOA-SVI'11), OnTheMove Federated Conferences*, pages 451–468, 2011

9. Waldemar Hummer, Orna Raz, and Schahram Dustdar. Towards Efficient Measuring of Web Services API Coverage. In *3rd International Workshop on Principles of Engineering Service-Oriented Systems (PESOS), co-located with ICSE'11*, pages 22–28, 2011

10. Waldemar Hummer, Philipp Leitner, and Schahram Dustdar. A Step-by-Step Debugging Technique To Facilitate Mashup Development and Maintenance. In *4th International Workshop on Web APIs and Services Mashups, co-located with ECOWS'10*, 2010

11. Alessio Gambi, Waldemar Hummer, and Schahram Dustdar. Automated testing of cloud-based elastic systems with autocles. In *28th IEEE/ACM International Conference on Automated Software Engineering (ASE), Demo track*, pages 714–717, 2013

12. Alessio Gambi, Waldemar Hummer, and Schahram Dustdar. Testing Elastic Systems with Surrogate Models. In *ICSE Workshop on Combining Modelling and Search-Based Software Engineering (CMSBSE)*, 2013

13. Patrick Gaubatz, Waldemar Hummer, Uwe Zdun, and Mark Strembeck. Enforcing Entailment Constraints in Offline Editing Scenarios for Real-time Collaborative Web Documents. In *29th ACM Symposium On Applied Computing (SAC)*, 2014

14. Patrick Gaubatz, Waldemar Hummer, Uwe Zdun, and Mark Strembeck. Supporting Customized Views for Enforcing Access Control Constraints in Real-time Collaborative Web Applications. In *International Conference on Web Engineering (ICWE)*, 2013

15. Patrick Gaubatz and Uwe Zdun. Supporting entailment constraints in the context of collaborative web applications. In *28th Symposium On Applied Computing (SAC)*. ACM, 2013

16. Christian Inzinger, Waldemar Hummer, Benjamin Satzger, Philipp Leitner, and Schahram Dustdar. Identifying incompatible service implementations using pooled decision trees. In *28th ACM Symposium on Applied Computing (SAC), DADS Track*, pages 485–492, 2013

17. Christian Inzinger, Benjamin Satzger, Philipp Leitner, Waldemar Hummer, and Schahram Dustdar. Model-based adaptation of cloud computing applications. In *International Conference on Model-Driven Engineering and Software Development*, pages 351–355, 2013

18. Christian Inzinger, Waldemar Hummer, Ioanna Lytra, Philipp Leitner, Huy Tran, Uwe Zdun, and Schahram Dustdar. Decisions, models, and monitoring a lifecycle model for the evolution of service-based systems. In *17th IEEE International EDOC Conference*, 2013

19. Christian Inzinger, Benjamin Satzger, Waldemar Hummer, and Schahram Dustdar. Specification and deployment of distributed monitoring and adaptation infrastructures. In *Workshop on Performance Assessment and Auditing in Service Computing, co-located with ICSOC'10*, pages 167–178, 2012

20. Christian Inzinger, Waldemar Hummer, Benjamin Satzger, Philipp Leitner, and Schahram Dustdar. Towards identifying root causes of faults in service-based applications. In *31st International Symposium on Reliable Distributed Systems (poster paper)*, pages 404–405, 2012

21. Christian Inzinger, Benjamin Satzger, Waldemar Hummer, Philipp Leitner, and Schahram Dustdar. Non-intrusive policy optimization for dependable and adaptive service-oriented systems. In *27th Annual ACM Symposium on Applied Computing (SAC)*, pages 504–510, 2012

22. Philipp Leitner, Benjamin Satzger, Waldemar Hummer, Christian Inzinger, and Schahram Dustdar. Cloudscale: a novel middleware for building transparently scaling cloud applications. In *27th Annual ACM Symposium on Applied Computing (SAC)*, pages 434–440. ACM, 2012

23. Philipp Leitner, Waldemar Hummer, Benjamin Satzger, Christian Inzinger, and Schahram Dustdar. Cost-Efficient and Application SLA-Aware Client Side Request Scheduling in an Infrastructure-as-a-Service Cloud. In *5th IEEE International Conference on Cloud Computing*, pages 213–220, 2012

24. Philipp Leitner, Christian Inzinger, Waldemar Hummer, Benjamin Satzger, and Schahram Dustdar. Application-level performance monitoring of cloud services based on the complex event processing paradigm. In *5th IEEE International Conference on Service-Oriented Computing and Applications (SOCA)*, pages 1–8, 2012

25. Philipp Leitner, Waldemar Hummer, and Schahram Dustdar. A monitoring data set for evaluating qos-aware service-based systems. In *ICSE Workshop on Principles of Engineering Service Oriented Systems (PESOS)*, pages 67–68, 2012

26. Philipp Leitner, Waldemar Hummer, Benjamin Satzger, and Schahram Dustdar. Stepwise and asynchronous runtime optimization of web service compositions. In *12th International Conference on Web Information System Engineering (WISE)*, pages 290–297. Springer, 2011

27. Philipp Leitner, Branimir Wetzstein, Dimka Karastoyanova, Waldemar Hummer, Schahram Dustdar, and Frank Leymann. Preventing SLA violations in service compositions using aspect-based fragment substitution. In *8th International Conference on Service-Oriented Computing*, pages 365–380. Springer, 2010

28. Benjamin Satzger, Waldemar Hummer, Philipp Leitner, and Schahram Dustdar. Esc: Towards an elastic stream computing platform for the cloud. In *IEEE International Conference on Cloud Computing (CLOUD)*, pages 348–355, 2011

29. Hong-Linh Truong, Schahram Dustdar, Georgiana Copil, Alessio Gambi, Waldemar Hummer, Duc-Hung Le, and Daniel Moldovan. CoMoT - A Platform-as-a-Service for Elasticity in the Cloud. In *IEEE International Workshop on the Future of PaaS*, 2014

30. Franz Wotawa, Marco Schulz, Ingo Pill, Seema Jehan, Philipp Leitner, Waldemar Hummer, Stefan Schulte, Philipp Hoenisch, and Schahram Dustdar. Fifty Shades of Grey in SOA Testing. In *Workshop on Advances in Model Based Testing, co-located with ICST'13*, pages 154–157, 2013

Book Chapters

1. Waldemar Hummer, Philipp Leitner, Anton Michlmayr, Florian Rosenberg, and Schahram Dustdar. *VRESCo - Vienna Runtime Environment for Service-oriented Computing*, pages 299–324. Service Engineering. European Research Results. Springer, 2010