University of Zagreb

FACULTY OF ELECTRICAL ENGINEERING AND COMPUTING

Goran Delač

RELIABILITY MANAGEMENT OF COMPOSITE CONSUMER APPLICATIONS

DOCTORAL THESIS

Zagreb, 2014

University of Zagreb

FACULTY OF ELECTRICAL ENGINEERING AND COMPUTING

Goran Delač

RELIABILITY MANAGEMENT OF COMPOSITE CONSUMER APPLICATIONS

DOCTORAL THESIS

Supervisor: Professor Siniša Srbljić, PhD

Zagreb, 2014

Sveučilište u Zagrebu

FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

Goran Delač

UPRAVLJANJE POUZDANOŠĆU KOMPOZITNIH POTROŠAČKIH PRIMJENSKIH PROGRAMA

DOKTORSKI RAD

Mentor: Prof. dr. sc. Siniša Srbljić

Zagreb, 2014.

Doctoral thesis was made at the University of Zagreb,

Faculty of Electrical Engineering and Computing,

Department of Electronics, Microelectronics, Computer and Intelligent Systems

Supervisor:

Professor Siniša Srbljić, PhD

Doctoral thesis contains: 228 pages.

Doctoral thesis number: _____

About the Supervisor

Siniša Srbljić was born in Velika Gorica in 1958. He received B.Sc. degree in EE and M.Sc. and Ph.D. degrees in CS from the University of Zagreb, Faculty of Electrical Engineering and Computing (FER), in 1981, 1985, and 1990, respectively. From February 1982, he is working at the Department ZEMRIS at FER. He was working with Prvomajska, R&D Dep., Croatia (1984-86) and he was visiting scientist at the University of Toronto, Canada (1993-1995), at the AT&T Labs, USA (1995-99), at the UC, Irvine, USA (2000,08-09), at the GlobalLogic Inc, USA (2011), and at the US Huawei, USA (2011). In March 2007, he was promoted to tenured professor. He coordinated 1 scientific program, led 1 scientific project, 1 technological project, and participated in 9 scientific projects. He led 3 research projects financed by companies from Croatia and USA, led 2 projects in USA, and participated in 3 projects in USA and Canada. He coordinates scientific program "Distributed Systems, Methods, and Applications" and leads scientific project "Computing Environments for Ubiquitous Distributed Systems" financed by the MZOS RH. He is author of 2 textbooks, more than 60 papers in journals and conference proceedings, and two 2 patents in USA in the area of distributed computing systems and consumer computing. Prof. Srbljić is a member of IEEE, ACM, and HATZ. He received Silver medal "Josip Lončar" from FER for Ph.D. thesis in 1990 and Vratislav Bedjanič Award, Iskra, Ljubljana, for M.Sc. thesis in 1985.

O mentoru

Siniša Srbljić rođen je u Velikoj Gorici 1958. godine. Diplomirao je u polju elektrotehnike, a magistrirao i doktorirao u polju računarstva na Sveučilištu u Zagrebu Fakultetu elektrotehnike i računarstva (FER), 1981., 1985. odnosno 1990. godine. Od veljače 1982. godine radi u Zavodu ZEMRIS FER-a. Bio je zaposlen u tvornici Prvomajska, odjel Istraživanje i razvoj (1984.-86.), a gostujući znanstvenik bio je na University of Toronto, Kanada (1993.-95.), u AT&T Labs, SAD (1995.-99.), na UC, Irvine, SAD (2000., 2008.-09.), u GlobalLogic Inc (2011.) i u US Huawei, SAD (2011.). U ožujku 2007. godine izabran je u trajno znanstveno-nastavno zvanje redovitog profesora. Koordinirao je 1 znanstveni program, vodio 1 znanstveni projekt, 1 tehnologijski projekt i sudjelovao na 9 znanstvenih projekta MZOS RH. Vodio je 3 istraživačka projekta financirana od kompanija iz Hrvatske i SAD, vodio 2 projekta u SAD i sudjelovao na 3 projekta u SAD i Kanadi. Koordinira znanstveni program "Raspodijeljeni sustavi, metode i primjene" i vodi znanstveni projekt "Računalne okoline za sveprisutne raspodijeljene sustave" koje financira MZOS RH. Autor je 2 udžbenika, više od 60 radova u časopisima i zbornicima konferencija i 2 patenta u SAD u području raspodijeljenih računalnih sustava i potrošačkog računarstva. Prof. Srbljić član je IEEE, ACM i HATZ. Primio je srebrnu plaketu "Josip Lončar" FER-a za doktorsku disertaciju (1990.) i nagradu Vratislav Bedjanič, Iskra Ljubljana, za magistarski rad (1985.).

To my grandparents
Milica, Milka, Božo and Vjeko

Acknowledgements

As any Ph.D. student knows and the future one is bound to find out, getting your degree in many ways resembles a difficult and long journey. Down the path, one is most likely to encounter joy and sadness, relief and frustration, pride and shame, hope and fear. So, in order to get to the goal, you have to rely on the help of others, some you have known for the most of your life, other extraordinary people you get to meet along the way. In order to thank them all properly, I would probably need to double this dissertation in size. Since this pages can bear only so many words, I hope that those that I unwittingly left out will have understanding.

First of all, I would like to express my deep gratitude to Professor Siniša Srbljić, my supervisor. When Professor Srbljić accepted me as a Ph.D. student at the *Consumer Computing Laboratory* back in 2008, he gave me an opportunity to advance towards my goals. He also took on himself the responsibility to guide my just freshly started scientific career to success, a feat that no doubt requires a leap of faith. With many scientific, and also life advices, he helped me focus my research. It shouldn't be forgotten that many of the discussions we had took place in a variety of moving vehicles. Luckily, we stopped by some really valuable research results, and for that, I will be forever grateful. I would also like to thank my committee members Professors Domagoj Jakobović, Darko Huljenić and Goran Martinović for accepting the responsibility of evaluating the research presented in this thesis.

Getting to this point surely wouldn't be possible without unconditional love and support I was getting from my family. It is fare to say that they shared all the good and the bad moments with me and had absolute understanding for my, sometimes lengthy, absence. To my parents Goranka and Darko, and my brother Davor, I owe everything and this thesis is as much their success as it is mine.

In some sense, each doctoral thesis is a team effort and I was lucky enough to be surrounded by extraordinary lab mates. I would especially like to thank my friend and colleague Marin Šilić with whom I spent many hours discussing the research challenges and shaping the most important contributions. Marin's help was invaluable and his input is visible throughout this thesis. I would also like to express my deepest gratitude to the rest of my lab mates that really helped me to keep going: Ivan Budiselić, Ivan Žužak, Klemo Vladimir, Jakov Krolo, Dejan Škvorc, Miroslav Popović and Zvonimir Pavlić. Thank you guys!

Although I am thanking them last, this is not where friends come in my life. I am fortunate and privileged to have friends that are always around when I need them. Some of them put up with me for over 20 years and I know they will remain by my side for many more. *Prvi prijatelji* thank you so much!

Abstract

Reliability Management of Composite Consumer Applications

Consumer computing is a research field focused on empowering ordinary application consumers, especially those that are not educated programmers, to build applications adapted to their specific needs. Consumer applications are component-based systems, constructed by combining the existing applications into new added value workflows. They are usually run in dynamic environments where building components are accessed over a shared medium, like the Internet, and often are not under supervision of the developer. Furthermore, component reusability is an important consumer computing concept as it simplifies application development, but also enables development of very complex consumer applications. Therefore, one of the principal challenges in sustainable consumer application development is to maintain a proper level of application's non-functional properties. The goal of this doctoral thesis is to produce a methodology for development of reliable consumer applications. To support development of complex consumer applications, the method needs to scale with the increased number of building components and workflow complexity. The proposed approach is a design-time iterative reliability management method that consists of consecutive steps of reliability estimation, detection of architectural weaknesses, and application improvement. In order to achieve scalability, an application analysis approach based on heuristic algorithms that leverage graphical structure of the reliability model is presented. Feasibility of the proposed method is extensively evaluated both on artificial and real-world data sets.

Keywords: Consumer computing, component-based systems, dependability properties, reliability management method.

Sažetak

Upravljanje pouzdanošću kompozitnih potrošačkih primjenskih programa

Potrošačko računarstvo je područje istraživanja čiji je cilj omogućiti izgradnju primjenskih programa korisnicima koji nemaju formalnog obrazovanja ili praktičnog iskustva u programskom inženjerstvu. Potrošački primjenski programi su sustavi zasnovani na komponentama, izgrađeni povezivanjem postojećih primjenskih programa u složene tijekove izvođenja. Pritom se smatra da su postojeći primjenski programi često izvan nadzora graditelja programa te im se pristupa korištenjem dijeljenih komunikacijskih kanala, poput globalne mreže internet. Osim toga, ključno svojstvo potrošačkog računarstva jest mogućnost ponovnog korištenja postojećih kompozitnih potrošačkih programa kao gradivnih komponenti novih programa. Takav pristup omogućava izgradnju vrlo složenih potrošačkih programa u čiji je tijek izvođenja ugrađen veliki broj gradivnih komponenti. S obzirom na dinamičko okruženje izvođenja i složenost potrošačkih programa, nužna je primjena postupaka ostvarivanja pouzdanosti kako bi se osigurala odgovarajuća nefunkcijska svojstva. Tradicionalni pristupi u rješavanju sličnih problema zahtijevaju široku primjenu zalihosti ili složenih modela pouzdanosti i postupaka optimizacije tijeka izvođenja programa. Međutim, navedeni pristupi ne zadržavaju svojstvo razmjernog rasta s obzirom na porast broja gradivnih komponenti i složenosti tijeka izvođenja primjenskog programa. S ciljem rješavanja navedenih istraživačkih izazova, u sklopu doktorske disertacije predložena je metoda za upravljanje pouzdanošću kompozitnih potrošačkih primjenskih programa. Metodu čine postupci otkrivanja arhitekturnih slabosti i poboljšanja pouzdanosti primjenskog programa. Otkrivanje arhitekturnih slabosti provodi se uz očuvanje svojstva razmjernog rasta primjenom skupa heurističkih algoritama. S ciljem ugradnje predložene metode u okolinu potrošačkog računarstva, definirani su odgovarajući programirljivi elementi i pomoćnik za potporu izgradnji pouzdanih potrošačkih primjenskih programa.

U prvom poglavlju (1 "*Introduction*"), predstavljena je motivacija istraživanja i hipoteze na kojima se zasniva rješenje postavljenih istraživačkih ciljeva. U zaključku poglavlja naveden je pregled sadržaja doktorskog rada.

U drugom poglavlju (2 "*Consumer Computing*"), opisano je istraživačko područje potrošačkog računarstva te motivacija za njegovu primjenu. Definirani su osnovni elementi okoline potrošačkog računarstva (gradivne komponente, programirljive komponente i pomoćnici) te su dani primjeri njihovog ostvarenja primjenom modernih tehnoloških i istraživačkih dostignuća.

U trećem poglavlju (3 "*Dependable Computer Systems*"), predstavljen je iscrpan pregled postojećih metoda za ostvarivanje svojstava oslonjivosti kompozitnih računalnih sustava. Predstavljena je taksonomija oslonjivog računarstva s posebnim naglaskom na metode ostvarivanja oslonjivosti. Pored tradicionalnih postupaka, opisana je metoda dinamičkog odabira komponenti, koja je često korištena u sustavima zasnovanim na uslugama.

U četvrtom poglavlju (4 "*Dependable Consumer Computing*"), predstavljen je pregled metode upravljanja pouzdanošću potrošačkih primjenskih programa. Metodu sačinjavaju tri osnovna procesa: procjena pouzdanosti (procjena ukupne pouzdanosti kompozitnog potrošačkog primjenskog programa zasnovana na analizi tijeka izvođenja), predlaganje arhitekturnih slabosti (otkrivanje najznačajnijih arhitekturnih slabosti u primjenskom sustavu) te poboljšanje pouzdanosti (primjena postupaka ostvarivanja oslonjivosti na najznačajnije arhitekturne slabosti primjenskog sustava). U zaključku poglavlja, predstavljena metoda je smještena u okvire taksonomije oslonjivih računalnih sustava.

U petom poglavlju (5 "*Reliability Model*"), opisan je model pouzdanosti primijenjen u sklopu procesa procjene pouzdanosti potrošačkog programa. Model se zasniva na preslikavanju tijeka izvođenje primjenskog programa u puteve utjecaja mreže vjerojatnosti. Na taj način izbjegnute su poteškoće prisutne u modelima zasnovanim na stanju, posebice kombinatorna eksplozija u rastu broja stanja kod složenijih potrošačkih primjenskih programa. Predstavljena je formalna definicija modela pouzdanosti, kao i algoritam pretvorbe UML dijagrama aktivnosti u zapis modela.

U šestom poglavlju (6 "*Weak Point Recommendation Method*"), predstavljen je skup algoritama preporuke arhitekturnih slabosti primjenskog programa. Pored izrazito preciznog računalno zahtjevnog algoritma zasnovanog na metodi analize osjetljivosti pouzdanosti (*WP-Influence*), definirana su dva manje računalno zahtjevna heuristička algoritma (*WP-WeakestPath* i *WP-WeightedPath*). Iako su manje preciznosti, heuristički algoritmi ostvaruju bolje svojstvo razmjernog rasta. Heuristički algoritmi su pogodni u uvjetima smanjenih računalnih sredstava potrebnih za provođenja postupka poboljšanja pouzdanosti. Provođenje većeg broja manje preciznih, ali računalno učinkovitijih koraka poboljšanja pouzdanosti uzrokuje manju ukupnu potrošnju računalnih sredstava.

U sedmom poglavlju (7 "*Consumer-Defined Reliability Improvement*"), opisan je proces poboljšanja pouzdanosti potrošačkih primjenskih programa. Predstavljena je arhitektura pomoćnika za potporu izgradnji pouzdanih kompozitnih potrošačkih primjenskih programa. Na

osnovi arhitekture, opisano je programsko ostvarenje pomoćnika zasnovano na udomljenicima. Osim toga, okolina potrošačkog računarstva proširena je dodatnim programirljivim komponentama. Opisane programirljive komponente podržavaju tradicionalne postupke oporavka od pogreške.

U osmom poglavlju (8 "*Evaluation*"), predstavljeno je vrednovanje metode upravljanja pouzdanošću potrošačkih primjenskih programa. Preciznost i računalna učinkovitost algoritama za preporuku arhitekturnih slabosti vrednovana je na umjetno stvorenom i stvarnom skupu podataka. Sažetak rezultata te zaključci koji potvrđuju hipoteze provedenog istraživanja, predstavljeni su na kraju poglavlja.

U devetom poglavlju (9 "*Conclusion*"), predstavljen je zaključak doktorskog rada s naglaskom na rezultate provedenog vrednovanja i ostvarene izvorne znanstvene doprinose.

Zaključno, rješenje predstavljeno u doktorskoj disertaciji primjereniji je pristup postizanja odgovarajućih nefunkcijskih svojstava vrlo složenih kompozitnih sustava od tradicionalnih postupaka ostvarivanja oslonjivosti. Predstavljena metoda omogućava ciljano poboljšanje arhitekturnih slabosti primjenskih programa uzimajući u obzir svojstvo pouzdanosti. Kako bi se osiguralo svojstvo razmjernog rasta postupka otkrivanja arhitekturnih slabosti, primijenjen je skup heurističkih algoritama preporuke čija su svojstva vrednovana na umjetnom i stvarnom skupu modela pouzdanosti primjenskih sustava. Navedeni postupak ostvaren je u okolini potrošačkog računarstva u skladu s čim su definirani odgovarajući programirljivi elementi i pomoćnik za potporu izgradnji pouzdanih potrošačkih primjenskih programa.

Ključne riječi: Potrošačko računarstvo, sustavi zasnovani na komponentama, svojstva oslonjivosti, postupak upravljanja pouzdanošću.

Contents

Chapter 1

Introduction

Consumer computing is a novel research field in computer science focused on including consumers into process of application development. Specifically, the goal of consumer computing is to empower ordinary application consumers, especially those that are not educated programmers, to create ubiquitous sociotechnical applications adapted to their needs. Motivation for this inclusion comes from the needs of the contemporary digital society that developed through proliferation of modern information and communication systems. Today's digital society can be regarded as a group of consumers, connected through state-of-the-art communication technologies, that utilize modern applications and devices in some of their every day activities. For instance, consumers can be connected through the Internet using modern mobile devices (e.g. smartphones and tablets) that support applications useful in their home or work environments. Since its creation, the digital society is constantly expanding, prompting the need for new content, as well as functionalities. The problem of content demand has been addressed by including consumers into its creation through various Web 2.0 systems such as blogs, social networks, wikis and others. However, having in mind the ever increasing number of various devices and their capabilities, it is not realistic to expect that professional developers will be able to meet all the application demands put forward by the modern digital society. Therefore, the motivation behind consumer computing is that sustainable expansion of digital society can be supported by including consumers into application development in a similar way they were included into content generation.

Consumer computing relies on a premise that consumers can construct applications in much the same way they use them. By automating their application consumption knowledge—knowledge on how the applications are used—consumers can construct new applications by

combining the existing ones. Therefore, consumer applications are in fact component-based systems constructed out of other applications (building components) that provide some basic functionality. Examples of possible building components in consumer computing are web widgets and mobile applications. Moreover, by utilizing capabilities of modern handheld devices on which these applications can be run, it is possible to crate information-rich situational applications that interface with other cyber and physical systems, like smarthome environments or social networks. In addition, consumer applications are considered to be hierarchical, meaning that other composite consumer applications can be used as building components of a new consumer application. Having that in mind, it can be expected that with adoption of consumer computing, the number of consumer applications and their complexity would increase rapidly, supporting sustainable development of the digital society.

However, the increase in number of available building components, as well as their complexity, introduces additional challenges that impact consumer's abilities to meet both the desired functional and non-functional properties of the constructed application. For instance, selecting components based on their functional properties can become intractable for a consumer due to an overwhelming number of possible choices. Thus, consumer computing entails necessity for assistants that can help identify components that best meet the consumer's design. On the other hand, non-functional application properties are another strong concern, namely application's dependability properties. This is due to the fact that consumer applications are component-based, meaning that each incorporated building component introduces its dependability threats. With the increase in the number of building components and complexity of the constructed workflow, combining building components with inadequate dependability properties can render the constructed application unusable, consequently preventing its further reuse. The focus of this thesis is to provide means of constructing complex consumer applications that retain adequate dependability, while focusing on reliability as one of the most pronounced dependability properties.

A possible solution to constructing reliable consumer applications can be found by observing similar component-based systems, like web services or other cloud-based systems in general. For instance, both the consumer applications and web services are run in dynamic environments where reliability values change rapidly and unexpectedly. This is due to the fact that such applications are often accessed over a shared medium, e.g. the Internet, and are developed and maintained by third parties over which a consumer has no influence. In such dynamic

environments, the commonly used solution is to introduce redundancies into the application's workflow with the aim of securing a certain degree of reliability. Redundancies entail using multiple functionally equivalent components over which redundant workflow execution paths are constructed. However, implementing and running such redundant systems can be resource demanding and no comprehensible solution exists that would enable their construction on a larger scale.

The solution proposed in this thesis is a novel design time reliability management method that enables focusing reliability improvements to building components (weak points) that have the most influence on the overall application reliability. That way, it is possible to achieve an acceptable level of reliability by selectively introducing redundancy or making other application improvements, e.g. replacing specific components with more reliable ones, while overall spending less resources during both design and run times.

To ensure the method is applicable to large consumer applications, it has to scale well with the increase in the number of building components as well as workflow complexity. For that reason, an adequate reliability model for consumer applications is defined. In addition, the weak point recommendation method is designed to scale with regard to the size of reliability model and additional computational requirements that stem from implementing the application improvements. The introduced method is based on a suite of algorithms that leverage a trade-off between accuracy in recommending most influential components and consumption of system resources. Specifically, it is based on a hypothesis that a suite of heuristic algorithms that detect most influential components based solely on the graph properties of the reliability model can be applied. That way it would possible to decrease the dependence of the proposed weak point recommendation method on the number of consumer application's building components. Furthermore, by taking into account additional computational overheads of the reliability management method, better computational performance can be achieved through proper selection of the recommendation algorithms. The stated hypotheses are confirmed by the results of the conducted evaluation.

Furthermore, the thesis gives insights into how the proposed reliability management method is integrated into the consumer computing environment. Specifically, the presented research shows how the consumer computing environment is extended with both programmable and assistant elements in order to support the proposed reliability management method. While the assistant element enables consumers to to perform the reliability management method,

programmable components enable introduction of redundancies into the consumer application workflow.

The rest of the thesis is organized as follows. Chapter 2 introduces the consumer computing research field. The underlying motivation for its adoption is presented, along with a definition of a general consumer computing environment that supports automation of application consumption knowledge. Basic elements of the consumer computing environment are described, along with their possible implementations using contemporary technologies and research findings. Finally, a consumer programming tool Geppeto that utilizes web widgets as application building components is presented.

Chapter 3 gives an overview of methods commonly used to ensure dependability of computer systems. The taxonomy of dependable computing is presented, defining dependability properties, threats and attainment means. Each of the dependability attainment mans (fault prevention, removal, forecasting and tolerance) is described in a greater detail. Finally, the chapter is concluded by discussing the dynamic component selection, a method commonly used to improve dependability properties of service-oriented systems.

Chapter 4 introduces the basics of dependable consumer computing. The reliability management method for consumer applications is defined, along with its main processes and actors. The proposed method is iterative and consists out of three main processes: *reliability estimation*, *weak point recommendation* and *reliability improvement*. These processes are conducted by actors, either a consumer or a computer system. Finally, the presented reliability management method is placed within the general dependable computing taxonomy.

Chapter 5 focuses on the *reliability estimation* process of the reliability management method. A formal definition of the reliability model for consumer applications based on belief networks is given. Furthermore, a model generator algorithm that converts workflow definitions, represented as UML activity diagrams, into consumer application reliability models is presented. The chapter is concluded with an example that illustrates how reliability models are constructed for Geppeto consumer applications.

Chapter 6 defines a suite of weak point recommendation algorithms that constitute the *weak point recommendation process* of the reliability management method. A reliability sensitivity analysis algorithm *WP-Influence* for accurate estimation of weak points is presented. In addition, two heuristic algorithms *WP-WeakestPath* and *WP-WeightedPath* that make recommendations based on graphical structure of reliability model are defined. These algorithms are

less accurate, but also less resource demanding. Finally, a computational complexity analysis for the introduced weak point recommendation algorithms is presented.

Chapter 7 addresses the *reliability improvement* process. An architecture for a consumer assistant based on the introduced reliability management method is presented. Based on the presented architecture an example of consumer assistant for development of reliable applications, implemented as a web widget, is presented. Apart form the assistance in analyzing an application, consumer is provided with a set of programmable elements that enable introduction of redundant constructs into the application workflow. Architectures of programmable elements that implement some of the common reliability attainment methods are presented. Finally, examples of programmable components implemented as web widgets are given.

Chapter 8 presents evaluation of the weak point recommendation algorithms. The evaluation is conducted on a set of artificially generated reliability models, as well as on a set of real-world models generated out of Yahoo Pipes composite applications. Furthermore, the evaluation is performed for two distinct experimental setups with aim of confirming properties of the weak point recommendation algorithms regarding accuracy and computational efficiency. The results of evaluation are summed up and discussed at the end of the chapter.

Chapter 9 outlines the achieved scientific contributions and concludes the thesis.

Chapter 2

Consumer Computing

Consumer computing is a methodology introduced at the UniZg, FER-CCL[1] whose aim is to empower ordinary application consumers, specifically those that are not educated programmers, to create ubiquitous sociotechnical applications adapted to their needs. Such applications are considered to be composite in their nature, constructed by consumers out of a number of building components. These building components are in fact other applications that provide some basic functionality. That way, consumer's innovation potential is expressed through interaction of building components, whose individual functionalities are well understood by its designer. The premise for such an application development environment is that an adequate number of consumers exist, accompanied by an appropriate number of building components that can be consumed using well established and intuitive interfaces. For that reason, the consumer computing methodology is mainly built around the environment of state-of-the-art web and mobile applications that provide intuitive graphical user interfaces (GUI) and are accessible to an ever increasing number of consumers. In fact, by utilizing modern devices that provide rich sensory data and connection to the Internet, e.g. smartphones and tablets, it is possible to build complex consumer applications that interact with contemporary technical and social systems. More on motivation behind the consumer computing methodology and on the challenges of designing a consumer computing environment is discussed throughout the rest of this chapter.

The remainder of the chapter is organized as follows. Motivation for consumer computing focused on enabling a sustainable digital society is presented in Section 2.1. The main concept behind consumer computing—development through automation of application consumption knowledge—is presented in Section 2.2. Based on the concept of consumption knowledge

[1]University of Zagreb, Faculty of Electrical Engineering and Computing, Consumer Computing Laboratory, http://ccl.fer.hr

automation, the consumer computing environment is defined in Section 2.3. The chapter is concluded by Section 2.4 that describes a consumer programming tool Geppeto which enables construction of web widget compositions.

2.1 Motivation: A Sustainable Digital Society

Motivation behind consumer computing can be expressed through the needs of the modern digital society that developed over the last couple of decades due to breakthroughs in information and communication technologies. Contemporary digital society can be regarded as a group of individuals connected through state-of-the-art communitarian technologies, mostly the Internet, that utilize some of the modern devices or applications in their every day activities. Since its creation, digital society is constantly expanding, which can be best observed by the increasing number of Internet users. According to the conducted surveys [1], over 2.7 billion people worldwide have access to the Internet, which is about 40% of the entire population. This means that the number of Internet users has increased roughly four times over the last 10 years. In order to sustain such growth, apart from providing the necessary infrastructure, a large amount of content had to be made available to the users. This task due to its sheer size could not be solved by a smaller group of people, e.g. professional developers. Therefore, with the emergence of Web 2.0 [2] Internet users were included in generation of online content through social networking sites, wikis, blogs and other platforms. That way it was possible to generate substantial amount of online content and support expansion of the digital society.

However, apart form content consumption, modern digital society relies on consumption of services. Consumption of services emerged with appearance of web applications supported by cloud computing platforms, but it really gained traction through wide adoption of mobile platforms. With proliferation of various handheld devices (e.g. smartphones, tablets) and other platforms, there is a huge demand for new applications, custom designed to the consumer's needs. For instance, latest reports indicate that the two largest mobile application stores *Google Play*[2] and *App Store*[3] have reached a combined number of approximately 2 million applications [3]. The data indicates that the number of applications in this two sores has increased about 60% in the last year alone. In the same time, the number of applications installed and used by consumers is on the increase. The data presented in [4] shows that the average number of

[2]Google Play, https://play.google.com/
[3]Apple App Store, http://www.apple.com/itunes/

installed applications by smartphone users in the US has increased form 32 in 2011 to 41 in the year 2012 (increase of 32%). Facilitating this growth is becoming an ever increasing problem as the gap between user demands and developer capabilities gets wider. As new platforms become available (e.g. *Google Glass*[4]) putting forward new application demands, this issue will only get exacerbated. In essence, sustainability of this segment of digital society will be brought into question as it will not be possible to fully address all the consumer demands that stem from capabilities provided by modern technology.

One obvious way to solve this issue is to include consumers into application development process, in much the same way consumers were included into content generation [5]. Some steps have already been taken in that direction with introduction of development tools that can be used by consumers to construct simple mobile applications (e.g. Appsbar[5] and App Inventor[6]). However, research goals adopted by FER-CCL indicate that consumer computing should be defined in a more comprehensive way by identifying all its actors, their relationships and roles within a general consumer computing environment. More details on how the consumer computing environment is defined as part of FER-CCL research activities are provided in the following sections.

2.2 Application Development Through Automation of Consumption Knowledge

One of the main guiding principles behind consumer computing is to provide development methods that would enable consumers to create new applications in the same way they use them, e.g. by utilizing GUI-level operations. All the knowledge collected during application usage on both application's functionality and possible interactions with other applications is defined as application consumption knowledge. For instance, such knowledge includes semantics of particular functionality subsets an application exposes, as well as use cases in which that particular application can be utilized in combination with other applications, e.g. to pre-process or post-process results. In essence, consumers should be able to innovate by directly translating previously obtained application consumption knowledge into new composite consumer applications. Thus, consumer applications effectively represent the automated application consump-

[4]Google Glass, http://www.google.com/glass
[5]Appsbar, http://www.appsbar.com/
[6]MIT App Inventor, http://appinventor.mit.edu/

tion knowledge and are built by designing workflows through composition of multiple building components. A simplified process of application consumption knowledge automation is presented in Figure 2.1. Although the process is not described thoroughly, it provides sufficient information to identify key elements of a general consumer computing environment.

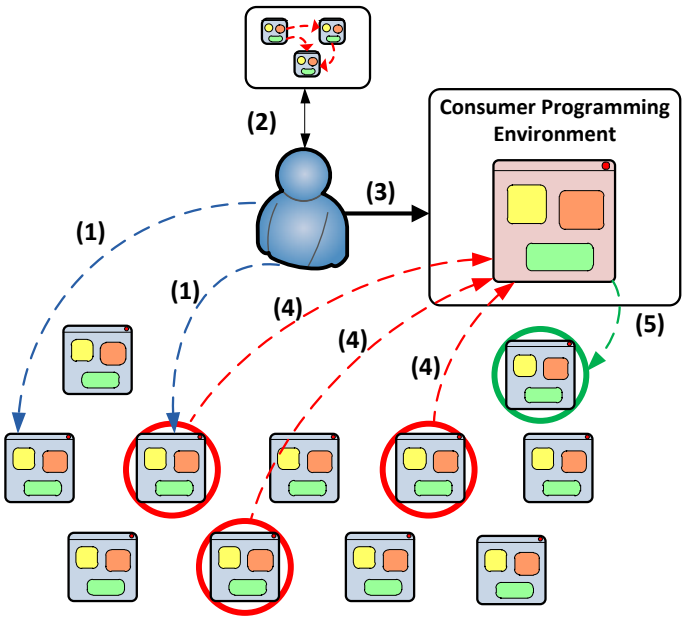


Figure 2.1: Automation of application consumption knowledge.

The process of application consumption knowledge automation is initiated by a consumer that uses a certain set of applications (1). By using the applications, consumer effectively constructs the application consumption knowledge (2) that consists both of application semantics and possible application interactions. The process of knowledge accumulation proceeds until it yields common application usage patterns, e.g. usage patterns that are often repeated and, thus, could be automated. In order to achieve the task of automating the acquired knowledge, consumer utilizes a consumer programming environment that enables construction of application compositions (3). Consumer first constructs a workflow on a functional level by translating the acquired knowledge into an abstract application implementation and then selects the appropriate building components from the set of known applications (4). It should be noted that components are selected based both on their functional and non-functional properties (e.g. quality of service). Components are then merged into a workflow resulting in a new application that effectively represents the automated application consumption knowledge. The constructed application can then be exposed as a new building block in the application set (5), supporting

further application development. For this reason, consumer applications are considered to be hierarchical, i.e. constructed applications can serve as building components for other consumer applications. Based on the described simplified knowledge automation process, it is possible to identify key elements of the consumer computing environment as described in the following section.

2.3 Consumer Computing Environment

Consumer computing environment is defined based on the requirements stemming from the process of application consumption knowledge automation. By taking into account the process described in Figure 2.1, three basic elements of consumer computing environment emerge defined as application-specific components, generic programmable components and consumer assistants. The first two elements are straightforward to identify as consumers are required to have a set of application-specific components, i.e. components that provide some basic functionality, at their disposal and use them in order to devise a more complex workflow. Moreover, when such a workflow is designed, consumers need to have a set of intuitive elements that would enable them to connect the selected application-specific components. Such consumer computing environment elements are referred to as generic programmable components. In essence, consumers use the generic programmable components to create composite (complex) components by combining the existing application-specific components. As stated previously, such composite components can then be reused as new application-specific components, extending the existing set of application-specific components. In fact, it can be concluded that expansion of the application-specific component set is dominated by two causes. The first cause stems from the increase in the number of application-specific components deployed by professional programmers (e.g. increase in the number of web or mobile applications). The second and more substantial cause for expansion of the application-specific component set comes from the potentially large number of applications developed exclusively by consumers. Since such application development process is continuously bootstrapped by newly emerging application-specific components, it can be concluded that the set of application-specific components rapidly expands.

However, with the increase in the number of available application-specific components issue of sustainable application development comes into question as the process of selecting appro-

priate building components becomes less tractable. Specifically, having a substantially large number of application-specific components it can become resource demanding to identify components that best meet the consumers' requirements. To mitigate this issue, consumer assistant elements are introduced into the consumer computing environment. Such elements secure the sustainability of application development by helping consumers to select the appropriate application-specific components based both on their functional and non-functional properties. Here functional properties encompass all the properties that are required to construct a certain workflow form the standpoint of the exposed component's functionality, e.g. if arithmetic operations are to be performed, a calculator component is required. On the other hand, non-functional properties define how well a selected component performs, e.g. how precise or reliable are the performed computations. Both properties are fundamental when it comes to supporting sustainable development as applying components with inadequate non-functional properties would eventually render the newly developed application-specific components unusable, preventing reusability and further development.

An overview of the consumer computing environment, defined by its three fundamental elements is shown in Figure 2.2. Each of the environment elements is described more thoroughly in the following subsections.

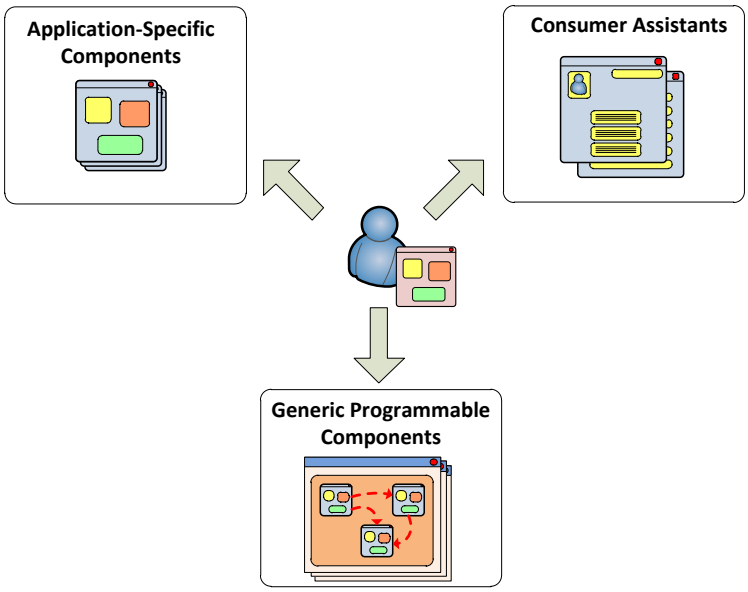


Figure 2.2: Consumer computing environment.

2.3.1 Application-Specific Components

Application-specific components represent application building blocks form various domains. Such components expose a specific functionality to consumers through an intuitive, well-established interface, like GUI. In essence, application-specific components can be build and made accessible using various contemporary technologies, like web widgets, web applications or mobile applications. For instance, a good example of application-specific component sets are large mobile applications stores. In such stores, applications are made available in various categories, ranging from social applications, games, business applications and others.

In their essence, application-specific components can be designed to run on a variety of devices that have different computational capabilities. Specifically, they can be designed as thin clients that provide user interface, while the actual computation is offloaded to other computer systems. For instance, components may access underlaying web services (e.g. through SOAP [6] or REST [7]) or other cloud computing services (platform and infrastructure as a service) [8]. Apart from that, application-specific components can be used to control the physical world through employment of cyber-physical systems [9]. Access to both physical and digital worlds enables construction of context-aware applications [10]. Such applications can be adapted to both physical parameters (e.g. user location) and virtual parameters (e.g. data retrieved from social services). In conclusion, application-specific components provide the consumer with a rich set of functionalities that can be combined into innovative workflows, well adapted to specific functional needs.

2.3.2 Generic Programmable Components

Generic programmable components are specialized applications whose purpose is to merge application-specific components into a new, added-value workflow that can be redeployed as a single application-specific component. Thus, the most basic functionality of programmable components is to provide means of establishing control and data flows between the building components. Considerable research efforts were conducted at FER-CCL with aim to facilitate data and control flow construction. Early research directions considered web service compositions using a spreadsheet-based programming language *Husky* [9]. Following research approaches were focused on application-specific components more suited to the consumer. Consumer programing model defined in [11, 12] considers constructing compositions out of web widgets. The model leverages a widget-oriented architecture, i.e. all elements of the consumer

computing environment are exposed as widgets, including the programmable components. In this specific case, composite applications are constructed using a GUI-level language and are represented by a semi-graphical tabular representation. Based on the defined consumer programming model, a tool for widget composition *Geppeto* was constructed. Since Geppeto applications are used as a motivating example for consumer applications in this thesis, the tool is described in a greater detail in Section 2.4.

Apart form providing basic data and control follow construction capabilities, generic programming elements need to be extended with other general computer system properties to make the applications more expressive and usable. Such properties include: concurrent execution, support for event-driven architectures, security, fault tolerance and others. Previous research activities at FER-CCL were aimed at developing support for event-driven consumer applications within the Geppeto framework [12]. Furthermore, a series of synchronization mechanisms were developed to enable concurrent execution of consumer applications [13]. Future research efforts are focused on extending the consumer computing environment with support for other properties. In accordance with that aim, one of the goals of this thesis is to enable implementation of reliable consumer applications through introduction of fault tolerance. More details on how the generic programming components are extended to support fault tolerant programming constructs are presented in Chapter 7.

2.3.3 Consumer Assistants

Previously presented arguments indicate that a set of application-specific components within a consumer computing environment can be expected to be significantly large in size. In such conditions, the task of constructing applications can become intractable as consumers would have substantial difficulties when searching for appropriate building components, both from their functional and non-functional perspectives. Therefore, with an escalating number of building components, consumer's innovation would be limited by an overwhelming number of choices, making the whole process of application development unsustainable. This would in turn cause a decrease in the number of new applications and stall the growth of the application-specific component sets. In order to avoid the stated issue, consumer computing environment is extended with assistant elements. Consumer assistants are specialized elements designed to ensure sustainability of application development by guiding the design and implementation process. In essence, consumer assistants provide a set of recommendations both on design

decisions and component selections with aim of achieving satisfactory functional and non-functional properties of the constructed application. In contrast to the application development process, the increase in application-specific data set size, accompanied by growing meta data on application-specific components' past usage is beneficiary towards consumer assistants as it enables them to make more accurate recommendations. Thus, accurate assistant recommendations make possible for consumers to create applications with appropriate quality parameters that enable their reuse, effectively securing sustainable increase of the application number and workflow complexity. Constructing such recommender systems that leverage very large data sets, also named *big data*, is feasible as a lot of ongoing research efforts are focused on making *big data* manageable [14], e.g. by enabling its capture, storage, search, sharing, analysis and visualization.

Having sufficient application-related data at disposal, there are two dominant ways consumer assistants can aid application development. The first approach comes strictly form leveraging the data through analysis procedures conducted by a computer system with aim of automatically extracting relevant recommendations. Since in that case analysis operations are conducted by a machine, such consumer assistants are named *digital assistants*. On the other hand, consumer assistants can be used to leverage human help by accessing application consumption knowledge of other consumers. Such systems, as they rely on human inputs, are named *human-based assistants*. The rest of this section shows that building both types of consumer assistants is feasible using contemporary research findings and technology.

Digital assistants

Digital assistants are designed to automatically process data on application-specific components and their usage with purpose of making recommendations that would aid the ongoing development efforts. Thus, recommendations are strictly made by a computer system based on the available application data by leveraging data mining, artificial intelligence or some other application-specific methods. In essence, digital assistants can be designed like contemporary recommender systems, e.g. systems used to predict ratings or preferences of Internet users. Generally, such recommender systems can be implemented using various approaches out of which two stand out in the literature defined as collaborative filtering and content-based filtering.

The collaborative filtering approach [15] enables recommendation on big data sets by esti-

mating similarity between particular entities. For instance, a data set may be constructed out of consumers and components they use, having information on how a particular consumer uses any given component. Based on that two dimensional matrix, similarity between users can be calculated based on components they used in the past. Such estimation is usually performed by applying classification methods (e.g. k-nearest neighbor algorithm) or leveraging statistical measures (e.g. Pearson Correlation Coefficient). When similar users are predicted, components can be recommended based on the assumption that similar users will use similar components. A clear advantage of collaborative filtering approach is that no specific knowledge on data set items (e.g., specific user preference or component properties) is needed to make a recommendation. However, since the recommendation is based solely on the collected data, the data set needs to be sufficiently large to facilitate accurate recommendations. Thus, collaborative filtering is sensitive to cold start issues as data quantity impacts the accuracy. On the other hand, having too large data sets may raise issues of scalability. Furthermore, data sparsity is another important issue, as collected data may not be equally representative for all of data set items, i.e. some items may have significantly larger and more representative inputs.

On the other hand, content-based filtering [16] derives recommendations by leveraging specific information on data set items. In the previously used example where users need to be recommended components, each component would have to be described by a set of features. Similarly, users are described by weighted feature vectors that contain the same item features, where weights indicate how important a particular feature is to the consumer. In general, weights can be calculated based on consumer's actions using various methods, including machine learning techniques such as: Bayesian classifiers, neural networks, decision trees and others. Although content-based filtering approaches depend less on the properties of the collected data set, their accuracy is mostly influenced by precision in weight factor estimation.

The above described methods clearly indicate that it is possible to construct digital assistants for consumer computing environment using state-of-the-art research findings as solutions to common problems in recommender systems can be applied to aid the consumer application development process. One of the goals of this thesis is to extend the the consumer computing environment with a digital assistant that would enable construction of reliable consumer applications. The assistant is based on the reliability management method described in Chapter 4. More details on its properties are given in Chapter 7.

Human-based assistants

Human-based assistants aid the application development process by leveraging direct human help. Specifically, using human-based assistants, consumers help other consumers by directly disseminating their application consumption and design knowledge. In essence, assistants that rely on human-made input can be implemented by relying on collective intelligence—knowledge that was generated through interaction and competition of a large number of individuals [17]. The subsets of collective intelligence, as defined by authors in [18] and presented in Figure 2.3, encompasses several fields including crowdsourcing, human computation, social computing and data mining. First three fields are directly applicable to human-based assistants and are more thoroughly described in the rest of the section. Data mining in its essence belongs to the sphere of digital assistants, although it can be applied to detect consumers that are best matches to provide help. Such techniques are similar to the previously described collaborative filtering approach and are not discussed further.

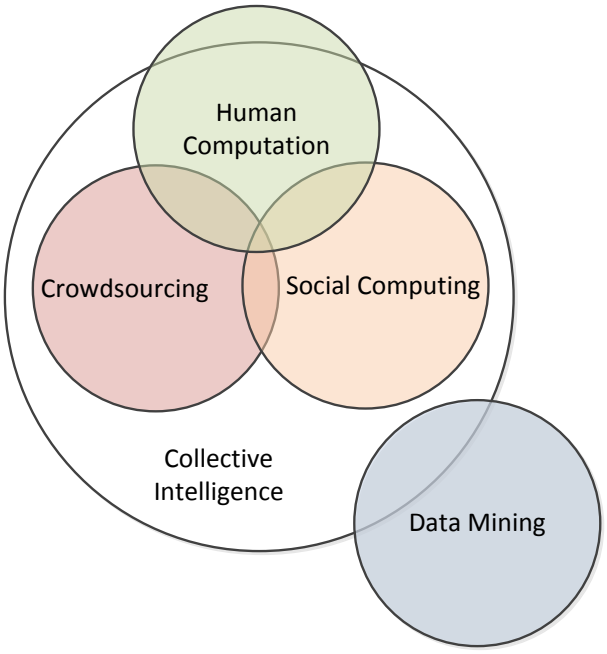


Figure 2.3: Collective intelligence: relation to crowdsourcing, social computing, and data mining.

Crowdsourcing [19] is a method of obtaining ideas, services or content by utilizing collective effort of a large group of people (consumers). The main premise of crowdsourcing is that a large group of people, working on the same problem, will through their collective effort yield a solution of a greater quality or solve problems that individuals cannot solve in the first place. Although general in its definition, crowdsourcing is mostly applicable to the online community as modern information and communication technologies enable nearly effortless interaction.

Crowdsourcing is currently a well established principle used in many spheres of problem solving. One of its most notable applications are the open source communities that produced a lot of high quality and widely used software solutions, e.g. *Linux* operating system. Other applications include content creation out of which is most notable *Wikipedia*[7], an online encyclopedia generated by its users. Apart form that a variety of idea incubators and problem solving frameworks are in wide use, like *Innocentive*[8], *Ideaken*[9] and *Marblar*[10]. All of the above mentioned concepts are easily modifiable to solve problems in the sphere of consumer computing, from collaborative application design to development.

Human computation [18] also applies human help to solve problems as crowdsourcing with one crucial difference. In human computation the focus is not on harnessing human creativity, but rather to use human's cognitive capabilities to efficiently solve the tasks that are difficultly solved by computer systems. The term is best defined in [20]:

Human computation is a paradigm for utilizing human processing power to solve problems that computers cannot yet solve.

In essence, humans are given a well-defined set of instructions to solve problems using their cognition, as opposed to identifying problems and deriving solutions in crowdsourcing approaches. In other words, such problems and their solutions can be easily defined but are not easily solvable by computer systems. For instance, common problems in that field include text translations and generation of various metadata. A wide variety of human computation tools is in use today. One of the most notable examples of a general purpose problem solving platform is the *Amazon Mechanical Turk*[11]. Other more specialized platforms include text translations (e.g. *Gengo*[12]), text recognition (e.g. *reCAPTCHA*[13]) and metadata generation (e.g. image recognition with *ESP game* (*GWAP*)[14]). The presented principles can be applied to construct human-based assistants especially to aid those parts of development that are are easy to describe, but hard to automate (e.g. finding equivalent GUI elements in semantically equivalent components).

Social computing [21] is a term that encompasses all the research activities aimed at un-

[7] Wikipedia, http://en.wikipedia.org/
[8] Innocentive, http://www.innocentive.com/
[9] Ideaken, http://www.ideaken.com/
[10] Marblar, http://marblar.com/
[11] Amazon Mechanical Turk, https://www.mturk.com
[12] Gengo, http://gengo.com/
[13] reCAPTCHA, http://www.google.com/recaptcha
[14] Gwap, http://www.gwap.com/

derstanding phenomena at the intersection of human social behavior and computer systems. In essence, social computing is focused on principles that enable human social interactions through use of computer systems. The examples of such systems are omnipresent and include blogs, instant messaging, social networks and others. Most notable representatives of social computing technologies are social networks that today have an extremely large number of users, like *Facebook*[15], *Google+*[16] and *Twitter*[17]. Such systems offer information on human behavior and relationships (e.g. *social graphs* [22]) that can be leveraged in human-based assistants. For instance, a consumer may direct development issues to a specific social circle that might provide best assistance. Thus, the principles of social computing are well-adapted to solve problems in the consumer computing domain.

2.4 Geppeto: A Widget Composition Tool

Geppeto [12] is an implementation of the consumer computing environment developed by researchers at UniZg, FER-CCL. More specifically, Geppeto is a programming tool that focuses on web widgets as basic elements of the consumer computing environment, i.e. application-specific components, as well as programmable components and assistants are implemented as web widgets. Web widgets (e.g. *Google Gadgets*[18]) are essentially miniature web pages loaded into smaller frames. Multiple such pages can than be loaded onto a single page, called a container page, and used simultaneously. Although due to size constraints widgets usually provide basic functionality, their combined usage can yield interesting and diverse composite applications that span various domains.

An example of widgets from various functional domains and their combined usage is shown in Figure 2.4. Specifically, three application-specific widgets are shown entitled *Bestsellers*, *Simple currency converter* and *Google translate*. *Bestsellers* is a widget that lists popular books based on their category by accessing services of an on-line book store. *Simple currency converter* is another widget that provides functionality of converting between currencies based on the exchange rate obtained by underlaying financial services. Finally, *Google Translate* is a widget used to translate text between languages by utilizing underlaying translation services. Although each of the presented widgets has a well-defined stand-alone functionality, their com-

[15] Facebook, https://www.facebook.com/
[16] Google+, https://plus.google.com/
[17] Twitter, https://twitter.com/
[18] Google Gadgets, https://developers.google.com/gadgets/

bined usage can yield a new workflow. For instance, a consumer may want to display the book price in another currency and read the book synopsis in another language. To achieve this task consumer needs to perform several operations. First, to show the price in another currency, the content of the *Price* field in *Bestsellers* widget needs to be copied into the *Amount* text box of the *Simple currency converter* widget (2). After the content has been copied, consumer needs to click on the *Convert* button in the *Simple currency converter* widget (2) and the result is displayed. Similarly, to get the translated synopsis, text is copied form the *Bestsellers* widget into the input field of the *Google translate* widget (3) and then *Translate* button is clicked to display the result. Although the presented example is simple, user performs two data flow ((1), (3)) and two control flow operations ((2), (4)) that can be automated using generic programmable elements.

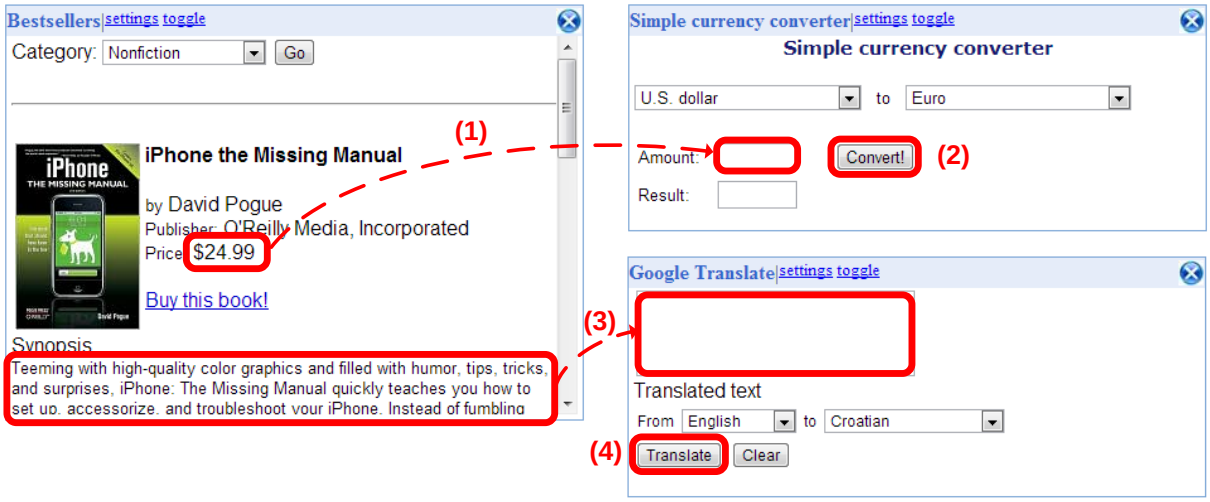


Figure 2.4: Widgets: example of combined usage.

The main generic programming component of Geppeto environment is the *Geppeto TouchMe* widget. *Geppeto TouchMe* implements the consumer programming methodology, as defined in [11]. The programming methodology consists of defining both data flow and control flow operations by performing GUI-level actions that are stored using a semi-graphical tabular representation. Essentially, *Geppeto Touch* me is a widget, with an empty canvas, whose functionality has to be constructed. This is done by adding GUI elements to the canvas, as shown in Figure 2.5. The added elements are used as inputs and outputs for the constructed application, as well as control elements that initiate workflow execution. The elements are added to the widget canvas by performing GUI-level actions using a context menu (*add* option). In a similar manner, by using the same context menu, both data flow (*copy* and *paste* options) and control flow actions are defined (*click* and *double click* options).

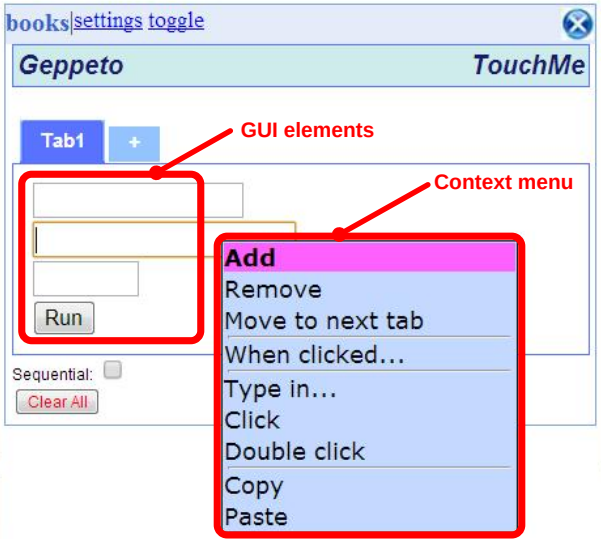


Figure 2.5: Geppeto TouchMe generic programming component.

To initiate application development, a control element (e.g. button) is selected using the *when clicked* option form the context menu. At that point data flow and control flow operations are stored in a tabular form as presented in Figure 2.6. Each cell in the presented spreadsheet represents a single GUI-level operation. Operations stored in the cells are executed sequentially, from left to right and from top to bottom as shown in the image. In case a cell has two adjacent cells defined on the bottom and right sides, both of the adjacent cells are run in parallel. For instance in the presented example, after *wait for click* cell has been executed, both *copy* cells are run in parallel. The shown program is in fact the representation of stored GUI-level operations for the previously discussed example, defined in Figure 2.4. To make the application complete, additional operations would have to be defined in order to copy results from *Simple currency converter* and *Google translate* widgets to text fields of the *books* widget. For reasons of simplicity, this actions are not shown.

wait for click Run at books	copy Synopsis at Bestsellers to element9 at GoogleTranslate	click Translate at GoogleTranslate
copy Price at Bestsellers to Amount at Simplecurrencyconverter1		
click Convert! at Simplecurrencyconverter1		
Execution order		

Figure 2.6: Tabular representation of Geppeto applications

Other programming components apart form *Geppeto TouchMe* are used within the environ-

ment. Their purpose is to extend the environment with additional programming constructs to enable development of various types of applications. Since focus here is not on data flow operations, such components usually have predefined interfaces and mostly enable only control flow operations. Two examples of such programmable widgets are *Geppeto TickMe* and *Geppeto TriggerMe* used to support development of event driven applications. Specifically, *Geppeto TickMe* enables introduction of events that occur periodically in time and *Geppeto TriggerMe* enables introduction of generic events published by an external system. Interfaces of both widgets are presented in Figure 2.7. *Geppeto TickMe* has a predefined interface that enables defining initial event time, its recurrence interval and a click operation executed at each event occurrence. Similarly, *Geppeto TriggerMe* enables subscription to a series of event channels. When an event is published to a certain event channel, widget executes the corresponding click operation, transferring the control flow to other parts of the application.

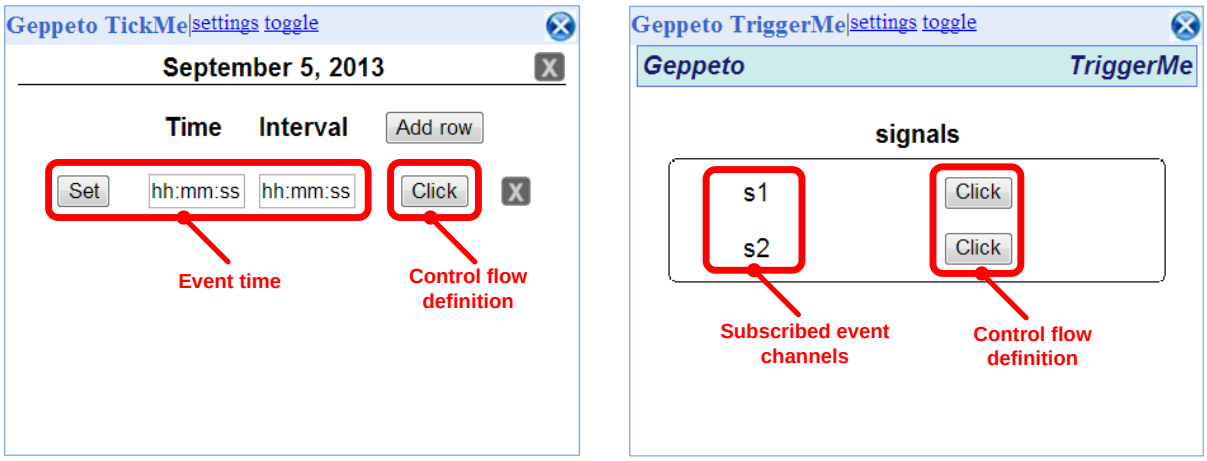


Figure 2.7: Programming components for event driven applications.

As stated previously, other programming elements have been developed as part of the Geppeto environment, for example widgets that enable concurrent programming. However, due to their similarity with the presented concepts, they are not discussed further.

Chapter 3

Dependable Computer Systems

This chapter presents an overview of the research work related to this thesis from the general standpoint of dependable computer systems. Basic definitions of dependability and dependable computer systems are presented in Section 3.1. The section also introduces basic dependability taxonomy that consists of dependability properties, threats and attainment means. More focus on dependability attainment means is given in Sections 3.2, 3.3, 3.4, and 3.5. Finally, Section 3.6 presents state-of-the-art approaches in the dynamic component selection principle, as such built systems are closely related to consumer applications addressed by this thesis.

3.1 Basic Definitions

A commonly used definition for dependability of computer systems in general was given by Avižienis and Laprie in [23]:

"Dependability is that property of a computer system that allows reliance to be justifiably placed on the service it delivers."

The authors define service as behavior of a computer system observed by a user, while the user is some other system that interacts with the afore mentioned service through a well-defined service interface, i.e. the user can be a computer system or a human. In its essence, the presented definition stresses out the need to establish an adequate level of confidence between the service provider and service user entities. Confidence here means that the user can expect to receive a service that meets a well-defined set of properties. To that end, an alternative definition is provided by the authors in [24] with emphasis on how to establish the criteria for a dependable service (system behavior):

"Dependability of a system is the ability to avoid service failures that are more frequent and more severe than it is acceptable."

In the presented definition a *service failure* or simply *failure*, as denoted in the rest of the section, is considered to be a specific behavior of a computer system in which the provided service deviates from what is considered to be correct service. Therefore, dependability can be simply defined as the ability of a system to provide correct service. To further clarify the meaning of dependability, taxonomy of dependable computing, as defined in [24, 25], incorporates three elements: properties, threats and means. Properties are used to quantify the level of dependability a particular system displays, e.g. quantify failure severity and frequency. Threats influence the properties as they degrade the dependability of a system. On the other hand, means define procedures that can be applied to increase the dependability by alleviating the aforementioned threats. Each of the dependable computing taxonomy elements is shortly explained throughout the rest of this section.

3.1.1 Dependability Properties

Commonly used dependability properties to describe a computer system are: availability, reliability, safety, integrity and maintainability [24, 26]. A computer system, or specifically the service it provides, is deemed to be dependable if it satisfies the stated properties to the extent acceptable to its user. Each of the dependability properties is briefly introduced in the following sections.

Availability

Two basic definitions for availability exist in the literature [27, 28]. Availability can be defined as the probability a particular system will be ready to provide service during a specific time interval. In order to support this definition, a concept of instantaneous availability is introduced. Instantaneous availability is a function that denotes the probability of a service being available at a given moment in time $a(t)$. Thus, the availability (or average availability) for a computer system in time interval $[t_1, t_2]$ is defined as:

$$A(T) = \frac{1}{T} \int_{t_1}^{t_2} a(t)dt \tag{3.1}$$

where $T = t_2 - t_1$. The expression is divided by T in order to normalize it to the interval

$[0, 1]$, since it is a probability measure.

Other commonly used definition states that average availability can be estimated as the ratio of uptime, i.e. time interval in which a computer system is ready to provide service, and the total running time:

$$A(T) = \frac{T_{up}}{T}, \quad T = T_{up} + T_{down} \tag{3.2}$$

where T_{up} is service uptime and T_{down} service downtime, i.e. the interval in which the service is unavailable.

Reliability

Reliability is defined as the probability that a computer system will provide correct service for a specific time period [29, 30]. Although definitions of reliability and availability are sometimes used interchangeably in the literature, reliability explicitly stands for providing correct service. On the other hand, availability stands for readiness to provide any kind of service, including that in presence of a failure. Thus, a necessary condition for a system to be reliable is for it to be available. Available systems, on the other hand, are not necessarily reliable as they can provide incorrect service, e.g. return erroneous results.

Formally, reliability is defined using random variable T that represents a unit of measurement in which a system provides correct service. This unit of measurement can be defined as time, e.g. time to system failure, or as a number of operations, e.g. number of successful service invocations until an unsuccessful one. By its definition, T is considered to be a continuous variable and can only take positive values. If statistical distribution of random variable T is given by the probability density function $f(x)$ (failure probability density function), reliability is defined as follows:

$$R(t) = Pr(T > t) = \int_t^\infty f(x)dx \tag{3.3}$$

Thus, reliability is probability that the system will operate according to its specifications if the value of random variable T is greater than t. It should be noted that $T(0) = 1$ and $T(\infty) = 0$.

Safety

Safety is a property that quantifies absence of catastrophic consequences for a system user [24, 31]. In general, computer systems should prevent harmful events that are direct results of their incorrect behavior, like loss of data or other harmful consequences to the user's environment. Such system events that cause irreparable damage are called *catastrophic failures*. In that context, safety is usually measured as continuation of *safeness*, i.e. time to a catastrophic failure. For that reason, safety can be expressed as a measure of reliability with respect to occurrence of catastrophic failures.

Integrity

Integrity encompasses all the characteristics of a system that prevent improper alternation of information, such as unauthorized amendment or deletion of information, or alterations to the system state [31–33]. This type of faulty behavior can be introduced intentionally (malevolently) or unintentionally (accidentally). In general, integrity can be quantified in a similar way as safety, as time to compromising of system information.

Maintainability

Maintainability is defined as capability of a system to undergo repairs or alterations. Alternations consider both the adaptive maintenance — adjustments of a system to environmental changes and perfective maintenance — improvements to the provided service in accordance with user-defined changes. It should be noted that it is not possible to clearly distinguish repairs and alteration tasks as user-requested change may be aimed at repairing a certain specification fault [34]. Maintainability can be expressed as the measure of time elapsed between a failure occurrence or alteration demand and service restoration [25].

The degree to which dependability properties are to be taken into account depends on the expected computer system characteristics, i.e. the degree of reliability safety, integrity and maintainability can be adapted to the application's specifications. In fact, some of the properties can prove to be contradictory (e.g. availability and safety) [35]. Therefore, the goal in designing a dependable system is to achieve a proper balance between the stated dependability properties.

Apart from dependability, security is an additional system characteristic described in literature that impacts the reliance which can be placed upon a given service. This characteristic has

a great importance for the systems accessible through the network infrastructure, e.g. systems exposed on the Web. In literature, security is described as a property of dependability [36] or, more commonly, as a stand-alone complementary characteristic [25]. Security encompasses the properties of availability, integrity and an additional property confidentiality. Confidentiality is defined as the property that guarantees absence of unauthorized disclosure of information. In that respect, availability is defined as readiness to provide service to *authorized* users and integrity stands for absence of *unauthorized* modifications of information. Since issues of security are not the focus of this thesis, security properties are not discussed further.

3.1.2 Threats to Dependability

Threats to dependability, as stated by the taxonomy of dependable and secure computing [24], are defined as faults, errors and failures. Failures are run time events that transition a computer system from the state of providing correct service to providing incorrect service. Failures are manifestations of errors observed by an user accessing a service through a well-defined interface. Thus, errors are internal states of the system that have led to deviation form what is considered to be correct service. The cause of an error, i.e. software or hardware issue that causes the system to transit into an erroneous state, is called a fault. The relationship between faults, errors and failures is summarized in Figure 3.1. In short, faults activate errors, errors propagate through the system and cause failures at user interfaces. It should be noted that not every fault will cause an error. This particularly applies to software as faults that are only activated by a specific user input can remain dormant in the system. In cases when the number of possible user input combinations is large, such faults may remain dormant throughout the system's life cycle. Furthermore, not every error will cause a failure. Errors propagate through the system state, but they do not necessarily reach the user interface. It may be necessary for multiple errors to occur before a system starts to provide incorrect service. Finally, failures propagate to users, causing further faults along the chain in case the user is another computer system. In the rest of this section, faults errors and failures are discussed in a greater detail.

Faults

Fault is an adjudged logical or physical cause of an error. According to the taxonomy defined in [24], faults are divided into three partially overlapping groups: physical , development and interaction faults.

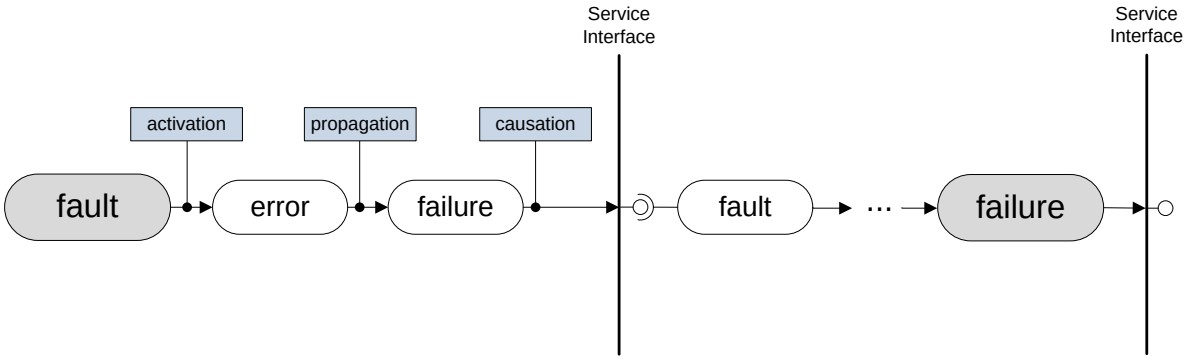


Figure 3.1: Faults, errors and failures.

Physical faults are faults in hardware that occur of natural causes, i.e. without direct human participation [32, 37–46]. They can either be internal — caused by physical deterioration of the hardware during its normal operation, or external — caused by interference of the environment with the system operation.

Development faults encompass human-made actions that deviated the system's functionality from its expected specification. According to their objective, development faults can be classified as non-malicious or malicious. Non-malicious faults can be either deliberate or non-deliberate. Deliberate non-malicious faults [47–51] are caused by bad design decisions that usually stem from introducing design compromises, like performance to cost trade-offs. Non-deliberate faults [52–60] are results of developer-made mistakes that remained undetected during the design and implementation process. On the other hand, malicious faults [61–65] are introduced intentionally to disrupt the system by causing denial of service, access of information without authorization or modification of system's functionality. Such faults are implemented as malicious logic in form of *Trojan horses*, *logic bombs*, *viruses*, *worms* and other software constructs.

Interaction faults [66–69] are a class of faults that result form system's interaction with the environment, i.e. the cause of a fault is of an external origin. In order for an external condition to impact the operation of a system, internal faults need to be present. For that reason, interaction faults are considered to be complementary to physical and development faults. For instance, physical faults whose origin is external, e.g. room temperature, belong to the class of interaction faults. Furthermore, interaction faults encompass a wide set of human-induced faults classified as reconfiguration faults. Such faults are caused by reconfiguration of a system during its operation either through change of operational parameters, or through maintenance and upgrade procedures.

Errors

An error is a system state, triggered by a fault, that may induce a system failure. The presented definition implies that system as a whole will fail only if the error reaches user interface, i.e. a set of states directly observed by the service user. Thus, errors propagate through the system from their point of origin ether internally, within building components, or between the building components. Since errors are in effect logical or physical manifestations of faults, rather than adjudged or hypothesized causes, they can be identified and mitigated during operation by directly observing system states. In that regard, errors can be classified as detected — their presence in the system is known, or latent — the errors are present but remain undetected. A wider classification of errors is given by the type of failure they produce, as presented in the following section.

Failures

Failures are events that occur when a system deviates from its intended behavior (correct service). Formally stated, a failure is triggered when the following condition is satisfied:

$$Q_E \subset Q_{Ext} \neq \varnothing \quad | \quad Q_{Ext} \subseteq Q_S \tag{3.4}$$

where Q_E is the set of erroneous states, Q_{Ext} set of external system states that are observed by the user and Q_S a set of all system states.

Failures are classified into groups according to the viewpoints defined in [24]. Commonly applied viewpoints are failure domain, detectability of failures, consistency of failures and severity (consequences) of failures.

Failure domain viewpoints are defined as content and timing failures [70–74]. In case of content failures, information provided at the service interface deviates from the one defined by the system specification, while in case of timing failures the information is not provided at the service interface within the expected time interval.

The detectability viewpoint addresses whether the occurred failure has been detected by the system [75–78]. In case a failure is detected, the system performs mitigation actions or informs the user on the failure occurrence. Successfully detected failures are called signaled (detected) failures, while the undetected failures are called unsignaled (latent) failures. It should be noted that failure detection mechanisms can fail themselves, causing an nonexistent failure

to be signaled or inducing an unsignaled failure.

In case multiple users are invoking a single service, failures are classified as consistent or inconsistent, according to the user perception. In case a failure occurs and it is perceived by all the service users, the failure is consistent [24, 25]. On the other hand, in case a failure exists but is only visible to a subset of users, the failure is considered to be inconsistent (Byzantine failure) [79–84].

Finally the failures can be classified according to their impact on the user or the environment. In general, a set of failure severity levels is defined along with the acceptable probabilities that a fault of a certain severity level will occur. In broadest sense, failures can be classified as minor and catastrophic [24,85–88]. Minor failures are those whose harmful consequences are of similar significance as the benefits achieved through correct service. On the other hand, catastrophic failures are accompanied by harmful consequences with orders of magnitude higher significance than benefits of a correct service.

A computer system that fails in a predictable way, i.e. its failure is described in the specification according to the defined viewpoints and its consequences are somewhat acceptable, is called a *fail-controlled system*. In case a system experiences timing failures that are acceptable to a certain degree, such system is called a *fail-stop system*. Finally, if a system fails to an acceptable extent through minor failures, such system is classified as a *fail-safe system*.

3.1.3 Means of Attaining Dependability

The primary goal of designing a dependable computer system is to assure that the values of dependability properties, as defined in Section 3.1.1, are at an acceptable level for proper service consumption. As stated previously, dependability properties are negatively impacted by the dependability threats defined in Section 3.1.2. In short, the fault, error and failure propagation chain, as presented in Figure 3.1, needs to be broken before a failure occurs if the computer system's dependability properties are to be improved. To address this issue, four means of attaining dependability are presented in the literature [24, 25] defined as fault prevention, fault tolerance, fault removal and fault forecasting. Fault prevention encompasses all the procedures that can be used to prevent introduction of faults into the system at design time. Fault tolerance considers a set of means used to mitigate errors (fault occurrences) at run time by introducing redundancies, i.e. provide correct service in the presence of faults. On the other hand, fault removal focuses on methods that are used to reduce the number of faults that are already present

in the system (have been introduced during system design and implementation), as well as their severity. Finally, fault forecasting considers means to estimate the number of faults in the system, as well as their possible future occurrence and impact. Depending on their goal, the described means can be classified as dependability provisioning and analysis. Dependability provisioning means are fault prevention and fault tolerance as their goal is to ensure such system design that will provide proper delivery of service. On the other hand, fault removal and fault forecasting are dependability analysis means as their goal focuses on analyzing the existing system to isolate particular faults and, thus, support their removal or avoidance. Furthermore, the means can be divided by the phase of their application. For instance, fault prevention means are strictly applied at design time, while fault tolerance is applied at run time. On the other hand, fault removal and fault forecasting can be applied both at design and run times. Since means of attaining dependability are of close interest to this thesis, they are described more thoroughly in in the following separate sections.

3.1.4 Taxonomy Summary

An overview of the dependability taxonomy is summarized in Figure 3.2. The presented taxonomy is fully applicable to composite consumer applications addressed in this thesis. However, definitions of dependably properties can be further rained to better suit the nature of consumer applications exposed on the web. In that regard, dependability properties are redefined to address the discrete nature of consumer applications as presented in the following subsection. It should be noted that the focus of this thesis is on observing reliability properties of composite consumer applications and, thus, other dependability properties are not addressed further. The sections in the rest of this chapter address the means of attaining dependability in a greater depth.

Dependability of Service-Oriented Systems

The so far presented definitions for dependability properties best conform to continuous systems, i.e. systems that are expected to provide correct service during a particular time interval (*never ending systems*). However, such definitions can be adapted to better suit dependability issues present in component based systems that are accessed on demand over the web infrastructure. Such composite systems are built upon the client-server architectural style [89,90], e.g. the components are exposed using some of the existing architectures based on the service-oriented

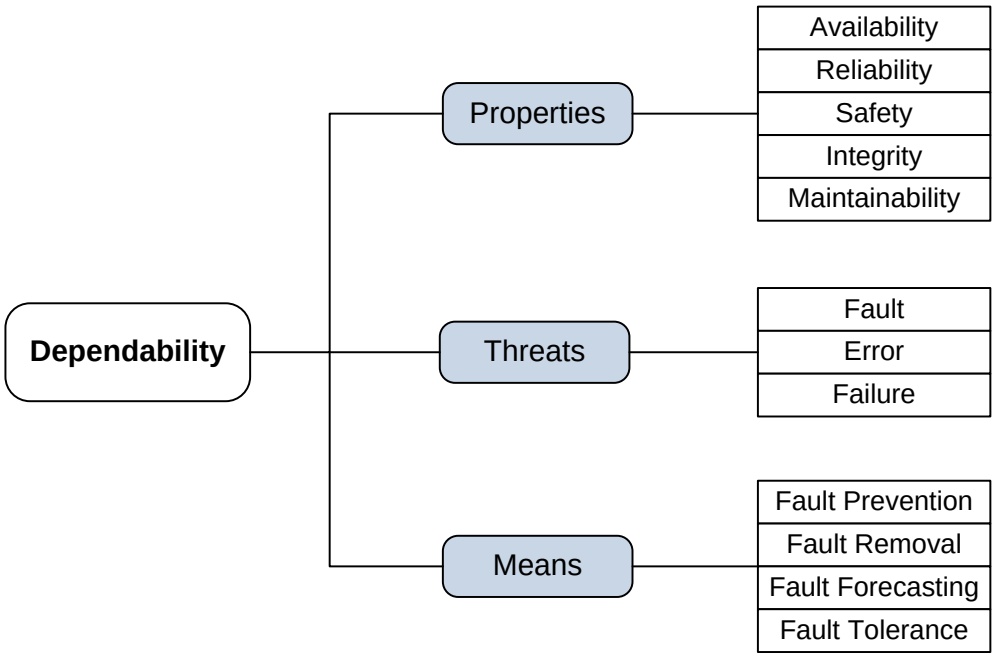


Figure 3.2: Taxonomy of dependable computing.

architectural style (SOA) [8, 91] or some other *Software as a Service* (SaaS) [92] principle. In contrast to the continuous systems, the operation of such a composite system can be regarded as a discrete event. This is due to the fact that it is only necessary for the component invocation to complete successfully, i.e. return a correct response, rather than to provide continuous service over a specific time period.

Taking into account the discrete nature of component invocations, availability and reliability are usually defined as a percentage of failures over the system's invocation history [93–95]. Taking that into account, the availability of a discrete system is defined as follows:

$$A(T) = \frac{N_{up}}{N_{tot}}, \quad N_{tot} = N_{up} + N_{down} \tag{3.5}$$

where N_{up} is the total number of invocations for which a component has provided an answer and N_{tot} is the total number of invocations over a time period T. In a similar way, reliability of a discrete system is defined by the following expression:

$$R(T) = \frac{N_{corr}}{N_{tot}}, \quad N_{tot} = N_{corr} + N_{fail} \tag{3.6}$$

where N_{corr} is the total number of invocations for which a component has provided correct

service, N_{fail} is the number of times a failure occurred and N_{tot} is the total number of invocations over a time period T. It should be noted that since correct invocations are a subset of the total number of invocations, the following condition always holds: $N_{corr} \leq N_{up} \Rightarrow R(T) \leq A(T)$.

Availability and reliability defined in such a way are also known as *availability and reliability on demand* [96,97]. Other dependability properties that quantify correctness of service as a probability that a certain disruptive event will occur, can be defined in a similar manner. For example, integrity can be defined as the ratio of component invocations without the undesirable changes to system data and the total number of component invocations.

3.2 Fault Prevention

Fault prevention encompasses design and development methodologies that prevent or reduce introduction of faults during system's design and implementation. In the context of software engineering, fault prevention improves dependability through the use of system requirement specifications, structured design and programming methods, formal methods (mathematically tractable languages and development tools) and principle of software reusability. Each of the fault prevention methods is briefly described in the rest of the section.

System Requirement Specification. From the service consumer's point of view a system may fail as a consequence of introduced faulty logic that does not conform to the expected system specifications. Such faults are generally introduced as a lack of understanding between service developers and users. Requirements engineering is a fault prevention method that encompasses all the techniques of collecting user requirements and translating them into system requirements [98–100]. Having a correct and detailed system specification is a prerequisite for fault free system design.

Structured Design and Programming Methods. To translate a system requirement into a fault-free implementation, systematic and structured design and development process needs to be applied. For instance, design methods can be used to achieve structured design by compartmentalizing functionalities into separate modules and, thus, enabling *information hiding* between different development activities [24, 101, 102]. In addition, design methods can be used to further reduce complexity and interdependence of particular modules. For instance,

structured design can be accomplished by applying design patterns [103, 104] — architectural schemes that appear frequently in application design. By applying well-known architectural schemes, the design decisions are better understood by all the involved parties and, thus, cause less frequent fault introductions. On the other hand, systematic approach to development can be achieved by utilizing various life cycle models [105–108] that define the order (priority) in which design and development tasks are to be executed, e.g. *V model*, *spiral model*, or *cluster model*.

Formal Methods. Formal methods in fault prevention are used to improve dependability during system design and implementation by harnessing mathematically tractable means (e.g. languages, tools) to develop and enforce formal system specifications [31, 101, 109–111]. Such means include mathematical verifications for correctness and appropriateness of each aspect of system development (e.g. verifying test case specifications), development procedures with incremental refinement, and proof mechanisms for software validation and verification. Although formal system specifications can greatly aid the development process, as they can be used to infer useful design information, their greatest disadvantage lies in development costs. In fact, formal methods of deriving a system specification often tend to match the actual system in its size causing significant development costs increases. For that reason, formal methods in fault prevention are dominantly used to improve dependability of a critical system subset.

Software Resuabiliy. Finally, software reusability is a common method of fault prevention. By utilizing previously constructed and thoroughly tested components, less faults get to be introduced during development cycles [101]. For that reason, reused components (i.e. reliable or trusted components) improve dependability properties of a constructed system. A strong advantage of this approach is decrease in development cost as only parts of the system need to be originally developed. However, changes in system requirements may cause the components to become inadequate, requiring redesign accompanied by a possible introduction of faults. Most commonly used design methodology that supports fault prevention is the *object-oriented programming* paradigm [112–114].

3.3 Fault Removal

Fault removal is a mean of attaining dependability though detection and removal of faults that have already been introduced into the system implementation. Since the introduction of faults cannot be completely avoided by fault prevention methods, fault removal is commonly used during both the system's design and run time. A typical fault removal cycle is defined by the diagram in Figure 3.3. The first step of any fault removal process is system analysis performed utilizing validation and verification methods. Verification is an internal system assessment whose goal it to check whether the system conforms to the given set of properties named verification conditions. In essence, verification methods assess whether the system has been constructed in a proper way. On the other hand, validation considers testing the system as a whole against a given specification or some other external system. In other words, validation is used to assess whether the system displays expected behavior, without taking into account its internal structure or other properties. If a verification or validation step detects deviations form the intended system design, a further step is conducted to analyze the fault, i.e. localize the part of the system responsible for incorrect service. The fault is then corrected and a new verification or validation step is conducted (non-regression test). If no further faults are detected, the fault removal cycle is completed, otherwise additional diagnosis and correction steps are undertaken.

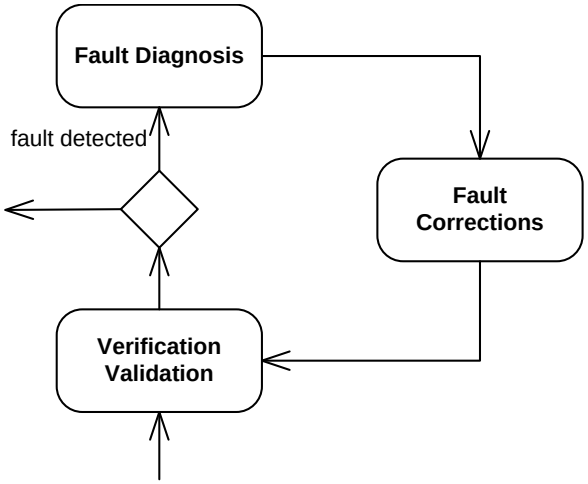


Figure 3.3: Fault removal cycle.

In general, verification and validation methods are classified by the way they are performed [24]. Verifying (validating) a system without the need of its execution is defined as a static verification (validation). On the other hand, if a system is to be run during the verification (validation), such a method is regarded as dynamic verification (validation).

3.3.1 Static Verification and Validation

Static verification (validation) can be performed on the system itself or on a system model. In case static verification (validation) is performed on a system itself, it can be conducted as *static analysis* or *theorem proving*. On the other hand, if the analysis is performed using a system model, *model checking* methods are applied.

Static Analysis and Theorem Proving. *Static analysis* incorporates code inspections [115–117], open-source development processes [118, 119], data flow analysis [120, 121], complexity analysis [122, 123], compiler checks [124, 125], abstract interpretation [126, 127], concurrency control [128, 129] and other methods. *Theorem proving* is a formal method used to mathematically prove that a certain system satisfies a given set of properties. This is particularly applicable to computer programs as they are written using programming languages that provide similar constructs as mathematical propositions. Thus, by using techniques present in theorem proving, it should be possible to prove or disprove properties of a program (e.g. correctness, security) [130]. There are two approaches to *theorem proving* defined as analytical and constructive methods. The analytical methods take into account the entire program and extend it with assertions, i.e. conditions that need to hold at a particular step. The assertions are then proven manually or by automated theorem provers [131, 132]. On the other hand, constructive methods are integrated into the development process itself [133, 134]. Such methods enable successive refinements by gradually translating a program specifications into an implementation. In doing so, each consecutive transformation step preserves correctness of the entire program.

Model Checking. The other approach in static verification is to test the behavior of a system using a model [24]. Such models are usually state based (e.g. finite state automata, Petri nets) and the process of their evaluation is called *model checking* [135–137]. *Model checking* provides means to exhaustively check the correctness of a system. However, a clear disadvantage of this approach is the escalating number of states that need be used in order to provide an accurate enough system representation. State explosions can easily occur as, for example, a single 32-bit integer variable can be assigned 2^{32} possible values. Thus, a common research topic in the field of *model checking* is to provide means of reducing system model complexity, e.g. using *predicate abstractions* [130] to reduce the state space by replacing all the expressions with boolean values. Although such simplified models are less accurate, they prove to be

an efficient fault removal method and are thus commonly used in circuit and communication protocols design.

3.3.2 Dynamic Verification and Validation

Dynamic verification (validation) methods are done by executing the actual system and are usually termed in literature simply as testing methods [24, 130, 138]. Since the number of input combinations for evaluated systems can be extremely high and, thus, can introduce considerable development and run time costs, testing is done on a reduced set of parameters. Therefore, testing methods can be used to detect particular system faults, rather than to prove that no faults at all exist. For that reason, a crucial step in test design is to provide an appropriate coverage of the input parameters in order to encompass the system's use cases as thoroughly as possible. Generally, tests are designed and executed as presented by the diagram in Figure 3.4.

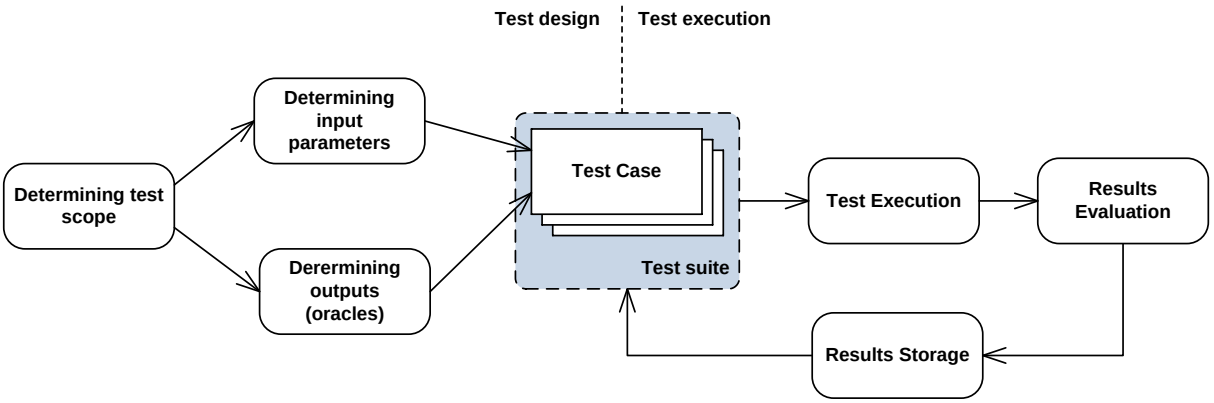


Figure 3.4: Test design and execution.

Test Cases. When designing a test, the first step is to determine which part of the system is to be tested. After focusing on a certain service, a reduced set of input parameters needs to be generated. The input parameters can be generated manually by a human or automatically. In certain cases manually derived test inputs that focus on critical issues are needed [139, 140]. However, such test inputs generally do not cover a wide enough set of input parameters. Thus, the inputs are also generated automatically [130, 141] in a deterministic or random way. The deterministic test generation [142, 143] considers input generation following a certain predefined pattern or specification. On the other hand, random (statistical) input generation [144, 145] creates test inputs by following an appropriate probability distribution. In order to provide as wide test coverage as possible, test inputs can be further strengthened by introducing invalid inputs.

The goal of this method is to observe how the system behaves in an unspecified situation. Such methods are known in literature as *fault injection* [24, 146].

A further issue in designing tests is to define the expected test outputs (oracles) for the given inputs. This problem is also known in the literature as the *oracle problem* [130, 147]. One way to construct oracles is to apply the *design by contract* [102, 130] methodology, which enables each functional property of a system to be described as a routine post condition.

Test inputs and oracles together form a test case. A set of test cases is called a test suite. During the test execution, defined input parameters are fed into the system. The gathered results are then tested against the oracles to determine if a fault has been detected. The results are then stored in case a further testing is required. For instance, *regression tests* [24, 138, 148] are conducted upon system changes to check if an old error has reemerged due to service functionality alterations.

Test Types. According to the architectural level at which they are applied, tests are commonly defined in literature as: *unit*, *integration*, *system* and *acceptance tests* [130, 149, 150]. *Unit tests* have the greatest granularity as they are conducted on selected parts of code or system components. At the higher architectural level, *integration tests* are conducted on component interfaces to determine if the system's building components interact properly. On the other hand, *system tests* are conducted exclusively on the system's interfaces to determine if the specified service as a whole is delivered properly. Finally, *acceptance tests* consider the interaction of a system with its intended user, i.e. determine how appropriate is the provided service to its user.

3.4 Fault Forecasting

Fault forecasting encompasses a set of methods that are used to evaluate system behavior with regard to activation of faults and their impact on dependability properties, mostly reliability [24, 101]. Such methods can give an answer to the question whether a system will achieve the expected dependability parameters or further corrective actions are needed to improve dependability. There are two main evaluation approaches in fault forecasting defined as qualitative (ordinal) evaluation and quantitative (probabilistic) evaluation.

3.4.1 Qualitative Evaluation

Qualitative evaluation has a task of identifying and classifying faults, and ranking them by their severity. In other words, a sequence of events leading to a system failure needs to be established. Such qualitative evaluation tasks are usually done by methods of *failure mode and effect analysis* [151, 152]. The results of such analysis yield a list of possible failures and their potential cause. Each failure is described by its probabilities of occurrence, probability of detection during system operation and impact on a provided service (severity). Additional information may be included to propose failure mitigation procedures or other actions that can be used to reduce the severity of particular faults.

3.4.2 Quantitative Evaluation

Quantitative evaluation aims at determining to what extent in terms of probability are the dependability properties satisfied, e.g. estimation of the systems reliability. Quantitative analysis usually relies on formal probabilistic models, like Markov chains and Petri nets [153–156]. The results of quantitative evaluation are numerical estimates of a particular dependability property. Additional methods such as *reliability block diagrams* [157, 158] and *fult-trees* [159–161] can be used to facilitate both the quantitative and qualitative evaluations.

Two dominant approaches in terms of quantitative fault forecasting are estimation (modeling) and testing (measurements) [24]. These approaches are complementary since measured system performance data is required to make estimations of dependability parameters. Thus, the accuracy of fault forecasting depends both on validity of the applied system model as well as on the accuracy of measured data. To reduce the need of testing, prediction methods can be employed to aid dependability estimation.

Estimation Approach. Quantitative estimation approaches are usually applied to assess system reliability and thus they are also known in the literature as reliability modeling methods. In general, reliability modeling is done by performing the following steps: 1) constructing a reliably model form stochastic processes that describe the system operation regarding failure and service restoration, and 2) processing the model to gain numerical estimations for system reliability (or other dependability properties). In terms of software, reliability models are usually constructed by analyzing a system as a whole or by analyzing system's architecture. A series of traditional reliability models that consider system as a whole (*black box* approach)

have been suggested in the literature [29, 31, 162]. For instance, statistical testing methods can be used to fit a system to a certain reliability (or probability of failure occurrence) distribution curve. In case of the latter approach, systems are decomposed into building components. The interaction of building components is then taken into account (white box approach) to estimate how do particular components impact the overall system reliability. Such approach is known in the literature as the architecture-based software reliability analysis [163–167]. Architecture-based reliability models are divided into path- [168, 169] and state-based [31, 170, 171] models. Path-based models are designed to take into account system's execution paths and aggregate their influence, while the state-based approaches map the system control-flow graph to a state space model (e.g. discrete- and continuous-time Markov chains [170, 172, 173], semi-Markov processes [174]).

Testing Approach. Testing approaches in quantitative fault forecasting differ from the previously described fault removal testing methods as they are focused on estimating system dependability parameters, rather then detecting specific faults. The main goal of the tests is to gather past system performance data. Such data can then be used to estimate reliability or availability of a system, e.g. using expressions (3.5) and (3.6). A variety of system performance and measuring approaches are in use [175–179]. The accuracy of testing approaches depends on the testing frequency as well as the testing scope. Regarding the testing scope, a major concern is to define the test input parameters so that they are a good representation of the system's operational profile. Operational profiles [180, 181] are statistical models that characterize how the system is going to be used, i.e. probabilities of execution are assigned to specific operations. Thus, the testing approaches in fault forecasting are usually named *operational testing* [24]. On the other hand, the more frequent the tests are, the collected data represents the system more accurately and, thus, the dependability properties are estimated with a greater precision. However, a trade-off between the testing frequency and accuracy needs to be established as too frequent tests can cause performance degradations of the monitored systems, resulting possibly in denial of service.

Prediction Approach. Prediction methods can be used in order to reduce the frequency and extensiveness of the tests that put an additional strain on the system. In case of the black box statistical models, if the reliability distribution curve is known and it properly represents the system, future reliability values may be extrapolated from the distribution curve. Although in

such cases the prediction process is reasonably computationally efficient, it is valid as long as the distribution curve remains a good representation of the system. If system's dependability properties change frequently, a new probability distribution curve would have to be fitted to keep the model valid. This is a demanding task as often substantial measurements need to be performed to obtain a valid sample needed to fit the model. One way to achieve a more efficient reliability prediction that emerged recently in the literature is to use models based on collaborative filtering [15, 182]. Such approaches employ statistical methods to discover similarities between the systems and how they are used. Therefore, the data set gathered for all the systems can then be used to estimate dependability properties of each individual system, reducing the need for testing [183–185]. Apparent disadvantage of this approach is that it requires a large enough data set of measurements in order to statistically match systems and provide accurate reliability estimations. For that reason, it is mostly applicable to web based services. However, it should be noted that with the increase in size of data sets, collaborative filtering approaches get more computationally demanding. In order to further improve their computational performance, data set reduction (aggregation) methods can be used as described in [186].

3.5 Fault Tolerance

Fault tolerance is a run time mean of attaining dependability that prevents occurrence of failures when faults are present in the system, i.e. a system is designed to prevent faults from disrupting its normal operation. The reasoning behind fault tolerance methods, as presented in Figure 3.5, is to prevent errors form propagating towards the service interfaces and consequently cause failures. For that reason, fault tolerance methods are divided into steps of *error detection and containment* and *system recovery* [24, 31, 101]. The *error detection and containment* step is designed to identify erroneous system states and contain their impact until proper mitigation actions can be taken. Containing the errors can also imply temporary degradation of system performance or service functionality in a safe way for its user. After the error is detected and its impact on the user minimized, *system recovery* step is performed. The goal of *system recovery* is to compensate for the detected error by utilizing built-in redundancies. In other words, the recovery process is dependent on existence of redundancy introduced during system development. If the recovery process is successful, system resumes its normal operation, and

fault activation remains transparent for the service user. In case the recovery is not successful, system should fail in a safe way and notify its user.

The rest of the section describes types of redundancy that can be integrated into a system, along with error detection and system recovery steps. Finally, the section is concluded by most common implementations of fault tolerance methods for single- and multi-version systems.

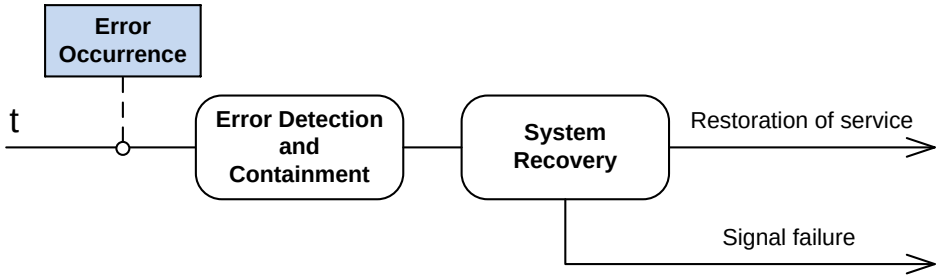


Figure 3.5: Fault tolerance principle.

3.5.1 Redundancy

Redundancy is essential to the concept of fault tolerance as it represents additional resources that are leveraged in order to compensate for the detected errors. These resources are redundant as they are not essential to the system's functionality, but rather improve its dependability properties. Thus, introducing redundancy generates both design and run time overheads. In general, redundant resources are classified as hardware, software, information and temporal redundancies.

Hardware redundancy is well explored and widely used in a variety of technical systems (e.g. embedded systems [187, 188]). Its goal is to provide a stable software execution environment through replication of hardware. Such replicated (redundant) circuits can be used to mitigate random malfunctions caused by component aging or the environment. Since hardware reliability is not the focus of this thesis, methods for implementing hardware-specific redundancy and fault tolerance will not be discussed further.

Software Redundancy

Software redundancy is based on existence of multiple, redundant software components (modules) that implement the same functionality. Although this concept is borrowed from hardware redundancy, significant differences exist. Hardware faults are random in their nature, caused by internal malfunctions or external (environmental) influences. Software faults,

on the other hand, cannot be considered as purely random events. As previously stated, they are caused by inadequacies in specification and design, as well as by mistakes done during the implementation. Therefore, it is not possible to tolerate software faults by mere replication as these functional level faults wold be replicated as well. A commonly used solution to this issue is addressed in literature as *design diversity* [189, 190]. The goal of *design diversity* is to assign the same functional specification of a software component to be developed by different programming teams. That way it is possible to avoid similar or common-cause failures, as well as decrease the probability that the components will fail on the same subsets of the input space. Software components designed in such fashion are called versions, variants or alternates. An apparent disadvantage to this approach is the increase in spent resources both at system design and run time. Furthermore, it should be noted that usage of diverse software components implies that means of deciding whether the variants have returned acceptable results need to be established. More on this issue is discussed in Section 3.5.2.

Data Redundancy

Similarly like in software redundancy, data redundancy is defined around the concept of *data diversity* [191]. Diverse data can be utilized to tolerate faults as fault activation in software is usually dependent on the data input sequences. For instance, the data re-expression algorithm [191] can be used to produce semantically equivalent, but syntactically different representations of the input data. Thus, changes at the system input may be used to avoid faults, but in certain cases may also require post-processing of the results if two input sequences are not entirely equivalent.

Temporal Reundancy

Temporal redundancy focuses on using time as a redundant resource. More specifically, the additional or redundant time is used to perform operations related to tolerating system faults. Temporal redundancy usually consists of repeating a system execution when an error is detected using the same initial hardware and software conditions (execution context). The main goal of temporal redundancy is to mitigate the failures that may not reoccur at different moments in time. For instance, temporal redundancy can be used to mitigate failures caused by intermittent [192–194] and transient system faults [195–197]. Intermittent faults are permanent faults that are usually activated periodically, possibly at irregular intervals. They may be caused by a

variety of factors including hardware malfunctions (e.g. due to temperature variations, regular maintenance) or software development faults (e.g. memory leaks, failures to properly initialize the program). On the other hand, transient faults are not periodical, but rather consequences of usually complex interactions that lead a system into a erroneous state (e.g. interactions between hardware, operating system and software). For instance, an error may occur because a certain system module is currently undergoing error recovery. If the execution is retried at a different moment, the fault may not be present any more and the system could continue its normal operation. Due to their nature, transient faults are difficult to detect, duplicate or diagnose and are, therefore, also regarded in literature as *Heisenbugs* [198, 199]. The main advantage of temporal redundancy is that it can be used to mitigate intermittent and transient faults without the need for software or hardware redundancy. Thus, the overhead of implementing temporal redundancy is significantly lower as such procedures do not require fault-specific information. However, use of additional time—and consequently system resources—may not be fully applicable to systems with hard real-time constraints [101]. Furthermore, temporal redundancy in ineffective in mitigating permanent faults, i.e. those that would be activated for repeated service invocations.

3.5.2 Error Detection

Error detection is the first step of any fault tolerance method. Essential to error detection design is to enable discovery of the broadest possible set of errors, regardless of the available recovery capabilities, as unhanded errors may lead to catastrophic failures. Its design depends on type of the applied redundancy as it can be focused on detecting errors within a single component, or outputs of multiple components may be used to argue about error occurrences. Both approaches are described in the rest of this section.

Component Checking

Component checks may be performed either internally or externally. In case a system component itself has a built-in error detection, it is required to have two basic properties: self-checking and self-protection [24, 200, 201]. The self-checking property considers that a component must be able to detect internal errors and take appropriate actions to prevent further error propagation. On the other hand, self-protection property means that a component must be able to withstand externally generated errors by detecting inadequacies in the information passed by other interacting components. On the other hand, component checks can be performed ex-

ternally by a separate component or subsystem. This approach is particularly applied when information required to perform a check is not fully available to the component in operation, but can rather be accessed by an other part of the system. For example, floating point or integer overflows can be detected by the operating system and passed to an appropriate error recovery routine, e.g. exception handling mechanism [202, 203].

A common way to implement component checking is to apply the dynamic verification based fault removal methods described in Section 3.3. Thus, fault removal and fault tolerance dependability means intersect in the field of error detection. Usually, acceptance tests are designed to check on the results of a program execution. These tests can incorporate timing checks to determine whether a system adheres to expected timing constraints, reversal checks used to establish whether the computed outputs appropriately match the given inputs, and reasonableness checks applied to confirm syntax and semantic properties of data, e.g. structural properties [204] and data ranges.

Selection Algorithms

In case software reliability is applied and variants are run in parallel, multiple equivalent outputs are available. Since the output set is generated by functionally equivalent components, it can be used to argue about each individual output's correctness. A common approach to detect errors in sets of equivalent outputs is by using selection algorithms (adjudication mechanisms) [31, 101, 205]. Most common selection algorithms are voting and consensus algorithms. In general, selection algorithms are designed to detect correct outputs if at least m out of N components produce identical outputs. If the number of agreeing components m is less than N, a system failure is signaled. The size of m depends on the particular selection algorithm implementation. Some of the voting strategies are described below.

Majority voting. In case a total of N component variants is available, majority voting [35, 206] is successful if at least $m = \lceil (N+1)/2 \rceil$ components return an identical result. In that case, the given subset of components with cardinality $|m|$ is considered to function properly and their result is selected as the correct one. Other components that differ in their outputs are considered to have experienced a failure.

Two-out-of-N voting. A commonly used voting mechanism two-out-of-N voting requires that at least 2 components agree on the result delivered at output ($m = 2$). It is shown in [207] that

this mechanism is effective is the output space if sufficiently large and variant failures can be considered statistically independent. Thus, if the output space is large enough, there is a low probability of two components failing in the exactly same way. The precondition for this assumption is that component failures are statistically independent, thus the principle of *design diversity* needs to be strictly upheld.

Consensus voting. In its essence, consensus voting is a generalization of the majority voting algorithm [31, 208]. It is defined by the following algorithm:

1. If a majority agreement can be reached ($m \geq \lceil (N+1)/2 \rceil$), the result of the majority is selected as output.

2. Otherwise, if a unique maximum agreement exists $m < \lceil (N+1)/2 \rceil$, the output of this agreeing group is selected as system output.

3. Otherwise, if a tie in the maximum agreement number exists, the correct output (group) is chosen randomly.

Many approaches to consensus algorithms can be applied, like the widely used *Paxos* algorithm family [209].

In general, selection algorithms have an advantage over component checking when it comes to design and implementation overhead. Specifically, selection algorithms do not require implementation of acceptance tests, that can be specific to a particular system component, as the error detection result is inferred from the available outputs. However, selection algorithms also have disadvantages, mainly regarding resource consumption and correctness. Resource consumption is one of the main disadvantages of selection algorithms as they require multiple variants to be run in parallel (or consecutively), which can in turn cause significant increases in spent resources (e.g. paid web services). On the other hand, correctness is another significant issue as selection algorithms can provide false positive and false negative solutions. For example, false positive solutions can occur in cases when the component output space is small [31]. In worst case, the output can have two possible values which means that if more than m components fail, the algorithm will indicate an erroneous output as correct. False negatives can occur when more than a single correct output exists causing dispersion of component outputs and preventing the consensus to be reached. In cases more than a single correct output exists, two-out-of-N voting

can prove to be more efficient than the majority voting. Consensus voting mechanism is flexible as it adapts to the output cardinality. In cases the output cardinality is small (2), the algorithm behaves as majority voting, and in cases output cardinality is large (tends to infinity), the algorithm behaves as two-out-of-N voting [208]. Finally, false negatives can also be caused by a different issue entirely that stems form inadequate output comparison (e.g. issue of comparing numerical values [210]).

3.5.3 Error Recovery

Once an error has been detected and its effect isolated, error recovery step can be taken. Its goal is to compensate for the activated fault or if that is not possible, to make sure system fails in a safe way. Two basic types of recovery exist named backward and forward recovery [24, 101]. Other recovery principles are derivatives of the two mentioned types.

Backward Recovery

Backward recovery compensates for errors by reverting a system *back* into its previous, error-free, state. Therefore, this error recovery type is based on leveraging temporal redundancy. The principle of backward recovery is presented in Figure 3.6. After an error has been detected, the first step of backward recovery is to revert the system state. The state is reverted into a previously recorded snapshot of system's execution context. These snapshots are referred to as checkpoints and the process of their storing is called checkpointing. If a checkpointed state is stable (error-free), system can restore service and resume its normal operation. However, if the error reoccurs, other fault handling procedures need to be taken (e.g. fault removal methods) before the system can resume normal operation.

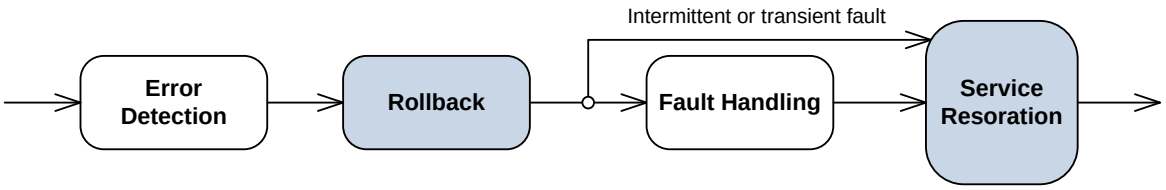


Figure 3.6: Backward recovery model.

Advantages of backward recovery are aligned with advantages of temporal redundancy. Such an approach is particularly suited to handle intermittent and transient faults. It is straightforward to implement as no knowledge of a specific fault is needed. This makes it a fre-

quently used error recovery principle in distributed systems that incorporate software fault tolerance [101, 211, 212]. However, backward recovery also retains disadvantages of applying temporal redundancy. As stated previously, one of the main disadvantages is the additional consumption of system resources, i.e. the system needs to be rerun and can remain unavailable for the duration of recovery. On the other hand, storing a system state (checkpoint) is an additional concern. A widely researched issue in backward recovery is to determine a proper checkpointing frequency [213, 214]. Checkpointing itself can be a computationally demanding task, depending on the size of system state and can require substantial storage capacities. Thus, too frequent checkpointing can lead to significant resource consumption. However, too rare checkpointing can increase recovery time as the system needs to be rerun form the time of checkpoint to the moment an error occurred, i.e. recovery is longest in cases an error occurs right before a scheduled checkpoint. Additional issues can arise if two systems (or subsystems) are interacting. For example, rollback to a previous checkpoint of one system may initiate a rollback in the other system consequently causing a new rollback in the first system. In such cases rollbacks can possibly chain back to the initial state causing great overheads.

Forward Recovery

Forward recovery is designed to mitigate errors by transitioning a system *forward* into a new state from which the system can resume normal operation. An overview of the forward recovery principle is presented in Figure 3.7. After an error has been detected, the compensation or rollforward step is taken. Compensation step leverages software redundancy to mitigate the error's effects. On the other hand, rollforward step sets a system into a new, usually degraded state form which the system can resume its operation. If compensation or rollforward steps fail, fault handling procedures (e.g. fault removal methods) need to be used before a service can be restored.

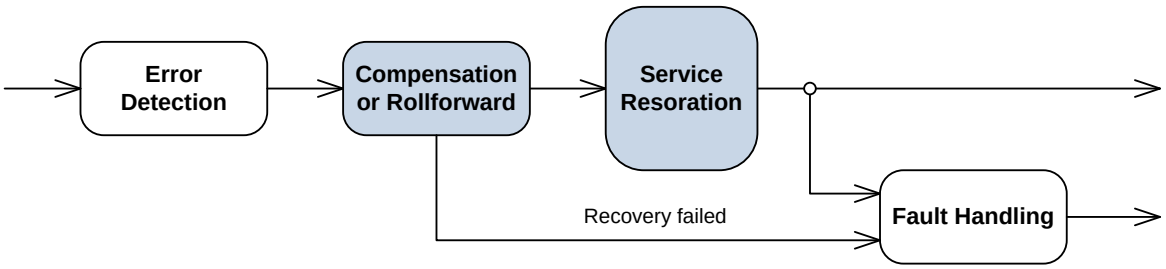


Figure 3.7: Forward recovery model.

Since the compensation step leverages, software redundancy, it can be implemented in two

basic ways. The first way is to design redundant components to be on standby until an error is detected. Only if an error is detected, they are run—in sequence or in parallel—to compensate for its effects. On the other hand, different approach is taken if multiple redundant components are run in parallel. In that case a selection (voting) algorithm, as described in Section 3.5.2, can be used to decide on a correct output. That way there is no need for a separate error detection mechanism as error detection and recovery are effectively merged into a single process. Compensation step designed in such a way is called *fault masking* [24, 101].

A notable advantage of forward recovery is short time required for restoration of service. In case of the stand by approach, this time is equal to rerunning a redundant component, or in case of *fault masking* service is practically restored instantly. Similarly, rollforward is computationally less demanding than rollback as it does not require checkpointing. In addition, forward recovery can be used to handle other faults then intermittent and transient [101, 215]. Most common disadvantages associated with forward recovery are related to application of software redundancy. For instance, fault-specific knowledge needs to be integrated into error detection procedures, i.e. it may be necessary to design error detection separately for each system component. Although this disadvantage does not apply to *fault masking*, compensation through selection algorithms requires parallel execution of multiple variants, leading to increase in spent system resources. Thus, in practice forward recovery is usually used when there is no time to perform backward recovery [101] or backward recovery is not adequate to mitigate failures.

3.5.4 Single-Version Fault Tolerance Techniques

This section presents the most commonly used single-version fault tolerance technique checkpoint and restart, along with its extension process pairs.

Checkpoint and Restart

Checkpoint and restart technique [200, 216] is a direct implementation of backward recovery, as presented by the diagram in Figure 3.8. In essence, this technique extends the software component execution environment with a checkpoint storage module. Checkpoint storage is used to store system state, following the principle of temporal redundancy defined in Section 3.5.1. At each execution or at predefined intervals (if the service is continuous), the error detection step checks the component state for errors. If an error is detected, recovery process is triggered and the system is rolled back to a error-free state loaded from checkpoint storage.

After the rollback is completed, system restarts and attempts to provide correct service. There are two ways to store error-free system states into checkpoint storage defined as static and dynamic. Static checkpointing is based on storing predetermined known stable states to which a system can safely revert. This is procedure can be performed at design time. On the other hand, dynamic checkpointing is done at run time by storing the actual system state at predefined intervals. Although dynamically stored states can be used to perform a faster recovery, additional circumstances need to be taken into account. For instance, checkpointing can happen after an error has occurred but before it has been detected, invalidating the stored state. In addition, checkpointing frequency needs to be adapted to system's needs, as discussed in Section 3.5.3. The presented checkpoint and restart model can be further extended by using data redundancy [200]. In that case, the recovery mechanism first attempts to recover by reverting to a stable system state. Then in case the recovery fails, execution is attempted with a different, semantically equivalent input.

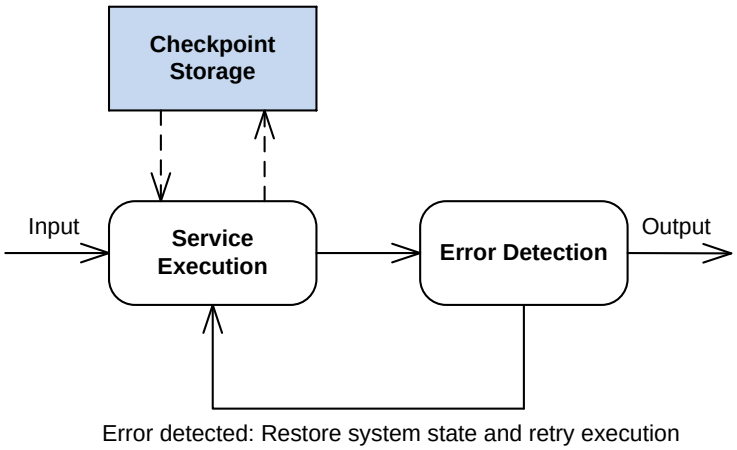


Figure 3.8: Checkpoint and restart model.

Process Pairs

Process pairs [216] are an extension of checkpoint and restart technique. The goal of process pairs technique is to avoid failures caused by operating system or hardware by running the same version of software component in two distinct processes or on two distinct processors. This way an additional software or hardware redundancy is applied alongside the temporal redundancy. The process pairs technique is defined by the diagram in Figure 3.9. Out of two processes one is considered to be primary and it is executed at service invocation. In case an error is detected, secondary process is started from a previously stored checkpoint. The service is momentarily

resorted and the primary process can undergo error recovery without causing additional down time.

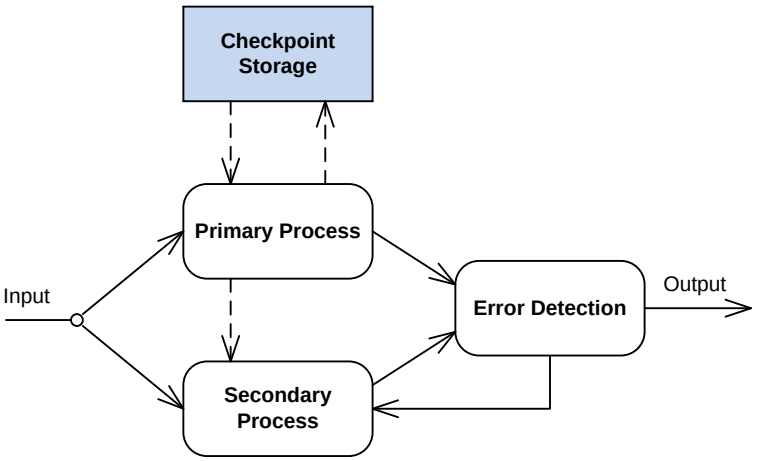


Figure 3.9: Process pairs model.

3.5.5 Multi-Version Fault Tolerance Techniques

Multi-version fault tolerance is based on applying multiple versions of components that can be executed either in sequence or in parallel. The goal of using multiple functionally equivalent components is to make the system more reliable than each individual component version. Thus, multi-version fault tolerance techniques are based on leveraging software (or hardware) redundancy. This section describes some of the most commonly applied multi-version fault tolerance implementations. Two basic techniques described in literature are recovery blocks and n-version programming. Other techniques, like n self-checking programming and consensus recovery blocks, are usually combinations of basic multi-version fault tolerance techniques.

Recovery Blocks

Recovery blocks [31, 217–219] are a fault tolerance technique that incorporates checkpoint and restart principle (temporal redundancy) with software redundancy. The model of recovery blocks is defined by the diagram in Figure 3.10. Recovery blocks leverage a set of functionally equivalent software components (alternate versions), out of which one is considered to be the primary component (*primary version*). Other components are ordered by their priority of execution. Prior to each execution, the system state is stored into checkpoint storage, similarly like in the checkpoint and restart technique. The input data is then passed to the primary version

and results of execution are evaluated by an acceptance test (*acceptance test 1*). In case the acceptance test fails, recovery step is performed by rolling back the system into a prior error-free state. The inputs are then passed to the next alternate version according to execution priority, e.g. after the primary version has been invoked the next in line is *alternate version 1*. This process is then repeated until the system recovers or the final component (*alternate version n*) fails triggering a system failure.

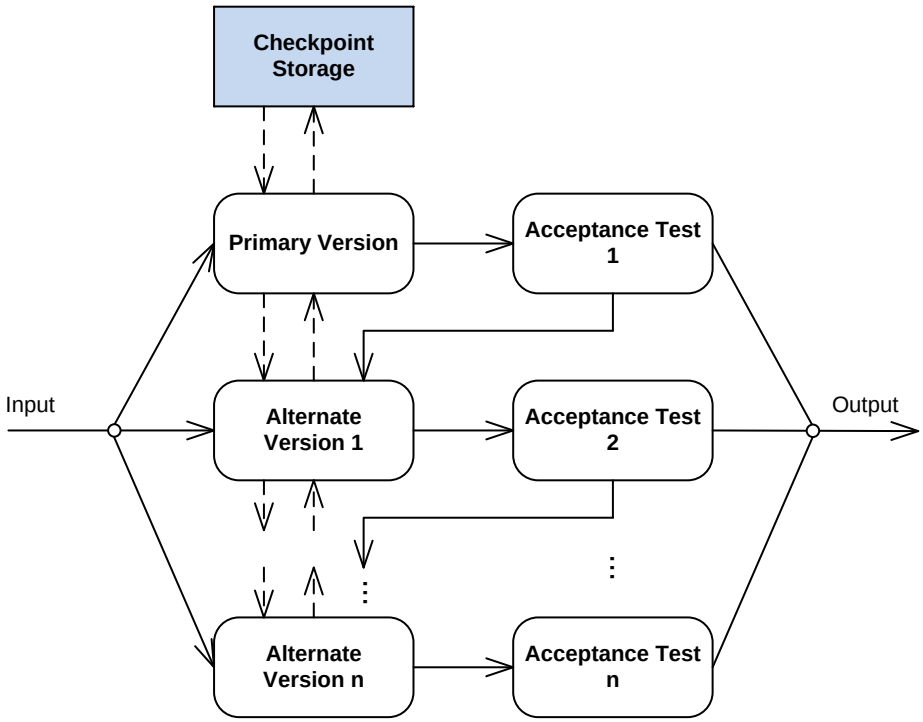


Figure 3.10: Recovery blocks model.

Component alternate versions can share the same acceptance test, but it is also possible that a separate acceptance test needs to be designed for each particular component, causing increase in design time. Acceptance tests can be designed to monitor the internal state of a component, rather than simply test its output. Although the recovery block technique considers sequential component executions, in reality components and tests can be executed concurrently (distributed recovery block [220]), depending on the available system resources. That way, it is possible to achieve faster error recoveries as outputs of other components are instantly available.

Reliability estimation. To simplify reliability analysis of recovery block system constructs, it will be assumed that all the acceptance tests perform correctly, i.e. probability of correct error detection is assumed to be 1. If the stated conditions apply, reliability can be estimated by observing how do particular components behave. Let system S_3 implement recovery block

technique using 3 component alternates and let r_1, r_2, and r_3 be the alternate component's reliabilities. Since reliability is defined as probability of correct service, probability of failure for any component is defined as $1 - r_i$. According to the recovery block technique all the alternate components need to fail in order for the whole system to fail. Therefore, the overall reliability of system $R(S_3)$ can be estimated as probability that all the components will not fail simultaneously: $R(S_3) = 1 - (1 - r_1) \cdot (1 - r_2) \cdot (1 - r_3)$. For the general case of n alternate components, the reliability of a system can be estimated as:

$$R(S_n) = 1 - \prod_{i=1}^{n} (1 - r_i) \tag{3.7}$$

It is visible form the presented equation that by increasing the number of components, the product on the right hand side will get smaller as all its factors are less than 1 ($r_i < 1$) and, thus, the overall system reliability will increase.

N-Version Programming

N-version programming [31, 200, 221, 222] is an implementation of the forward recovery principle. Like recovery block technique it leverages functionally equivalent components (software or hardware redundancy) to compensate for system errors. As shown by the model in Figure 3.11, in n-version programming technique alternate components are run concurrently and their outputs are processed by a voting algorithm, as described in Section 3.5.2. The advantage of this approach is that no additional acceptance tests are needed, reducing the design time and making the recovery process faster. However n-version programming entails increased resource consumption as components need to be run in parallel, rather than sequentially like in the recovery block technique. In actual implementations, resource consumption can be lowered by running component subsets sequentially, i.e. an appropriate trade-off between running components in sequence and parallel can be established [101, 223]. In that case it is necessary to add temporal redundancy into the design, e.g. checkpointing.

Reliability estimation. Reliability of a system that implements n-version programming depends on the number of alternate components and the applied selection algorithm. For the purposes of this analysis a simplification is introduced that subsystems executing selection algorithms are completely reliable. Let system S_3 be implemented using 3 component alternates and let r_1, r_2, and r_3 be the alternate component's reliabilities. In addition, let the selection algo-

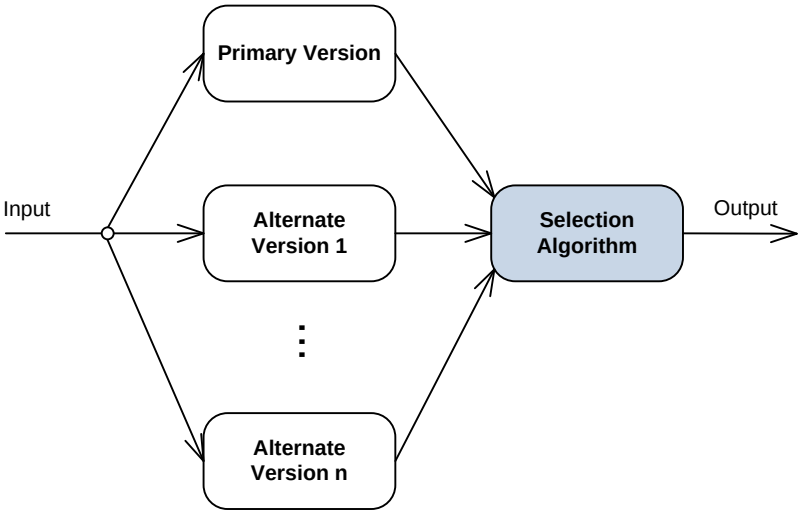


Figure 3.11: N-version programming model.

rithm be 2-out-of-n majority voting. This means that at least two components need to agree on the result for the system to function properly. Thus, at any time one component may fail and the error will be compensated. Since the probability of failure can be expressed as $1-r_i$, the overall system reliability can be estimated as: $R(S_3) = r_1r_2r_3+(1-r_1)r_2r_3+r_1(1-r_2)r_3+r_1r_2(1-r_3)$.

More details on estimating reliability of n-version programming constructs can be found in [101, 224]. If a final simplification is introduced, defining all alternate component's reliabilities as equal ($r = r_1 = \cdots = r_n$), a general expression for reliability estimation of a n-version programming system is given by the following expression:

$$R(S_n) = \sum_{i=m}^{n} \binom{n}{n-i} r^i (1-r)^{n-i} \tag{3.8}$$

where m is the lower bound on the number of agreeing components in an m-out-of-n majority vote.

The remainder of this section describes complex multi-version fault tolerance techniques that are constructed to adopt advantages of both the recovery block and n-version programming approaches.

N Self-Checking Programming

N self-checking programming [31, 200, 225] is a combination of recovery blocks and n-version programming, as defined by the diagram in Figure 3.12. The first step of this technique is to apply the recovery block principle. Each alternate component has an assigned acceptance test which is run to check for erroneous behavior. If the acceptance test is passed, component's

result is sent to the next phase, otherwise if an error is detected, the result is discarded. In the next phase, a selection algorithm processes the results. The selection algorithm may use results of the acceptance tests, e.g. rank the results by test determined quality. It can also implement some of the previously described selection algorithms. Generally, alternate components can be run in parallel, sequentially or in subsets (e.g. pairs) depending on the system configuration. Sequential execution requires the use of checkpoints, while parallel execution requires application of voting or consensus algorithms.

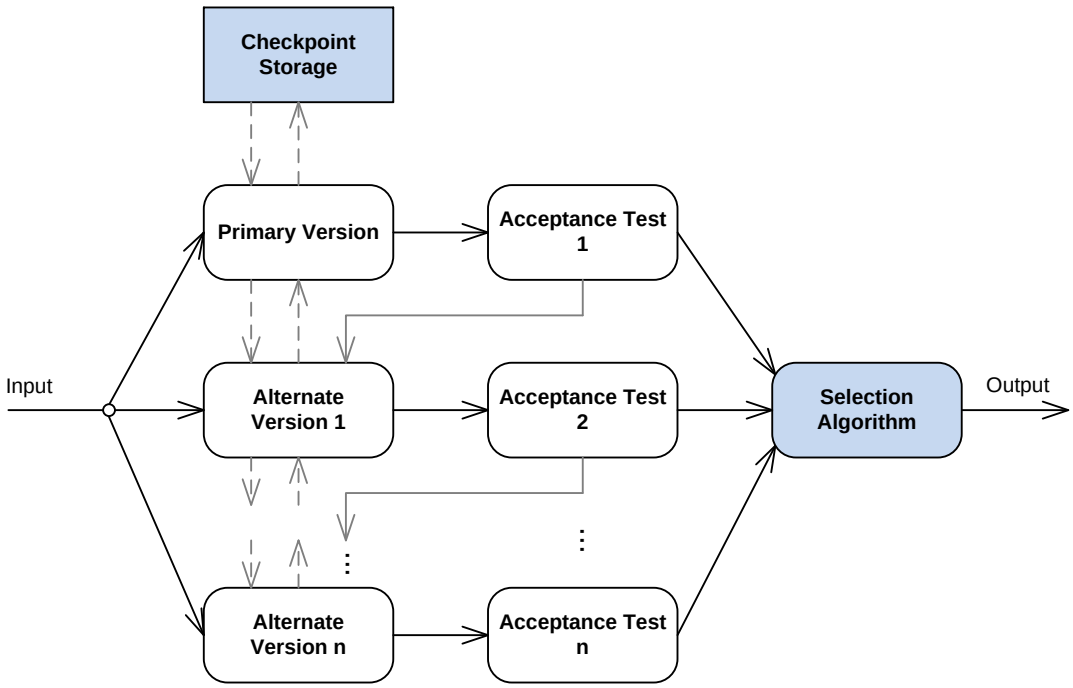


Figure 3.12: N self-checking programming model.

Consensus Recovery Blocks

Consensus recovery blocks [31, 200, 207] are also a combination of recovery blocks and n-version programming, but unlike in n self-checking programing, the n-version programming phase is run first. The model of consensus recovery blocks is presented by the diagram in Figure 3.13. The components are first run in parallel and their execution results are evaluated by a selection algorithm. If the selection algorithm can derive the correct output, it is passed to the system interface bypassing the recovery block phase. On the other hand, in case the applied selection algorithm fails, recovery block phase is triggered. That way, the consensus recovery block approach can be used to avoid the issue of false negative results in the n-version

programming step (e.g. multiple correct answers are present). At the recovery block step, components can be first rerun if an intermittent or transient fault is suspected or alternatively, previously obtained results can be directly applied. If the acceptance test is passed, the error has been mitigated, otherwise the system experiences a failure.

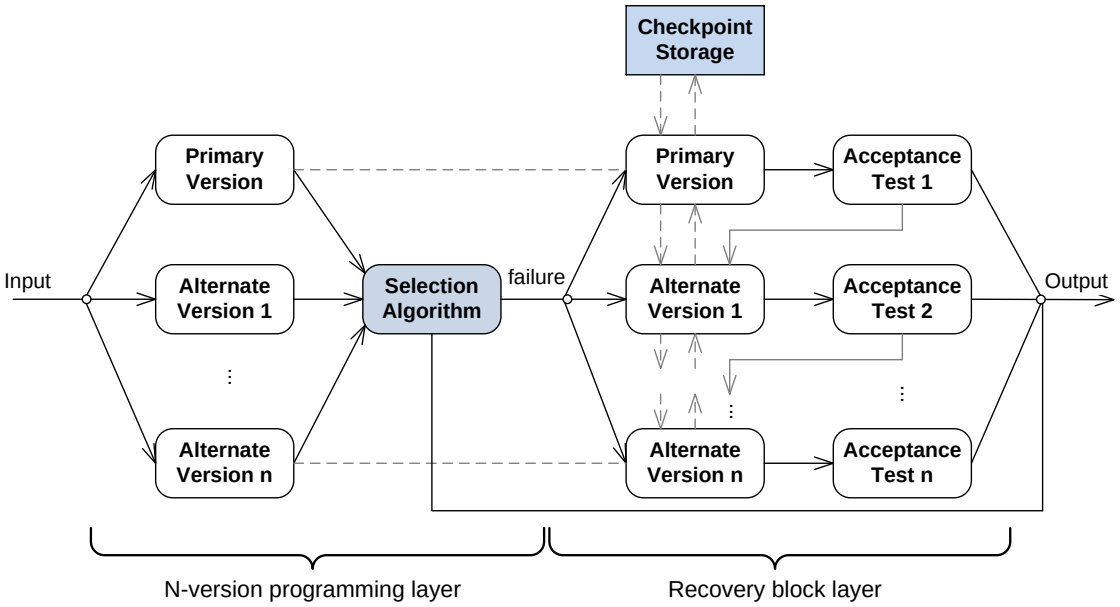


Figure 3.13: Consensus recovery blocks model.

3.6 Dynamic Component Selection

As described in the previous sections, attaining dependability during run time can be achieved by introducing some type of redundancy. This generally means that a fault tolerance mechanism is to be applied. In practice, fault tolerance mechanisms are often avoided in favor of design time dependability attainment due to their high and inefficient resource consumption. However, for systems whose dependability properties change often and unpredictably (e.g. composite systems exposed on the Web), applying some form of fault tolerance is essential. In order to achieve appropriate dependability properties and reduce resource consumption, methods can be designed to combine the presented dependability attainment means. One such, commonly used, method that combines both fault forecasting and fault tolerance is dynamic component selection. The method is conducted in two phases defined as design and execution phase, as presented in Figure 3.14.

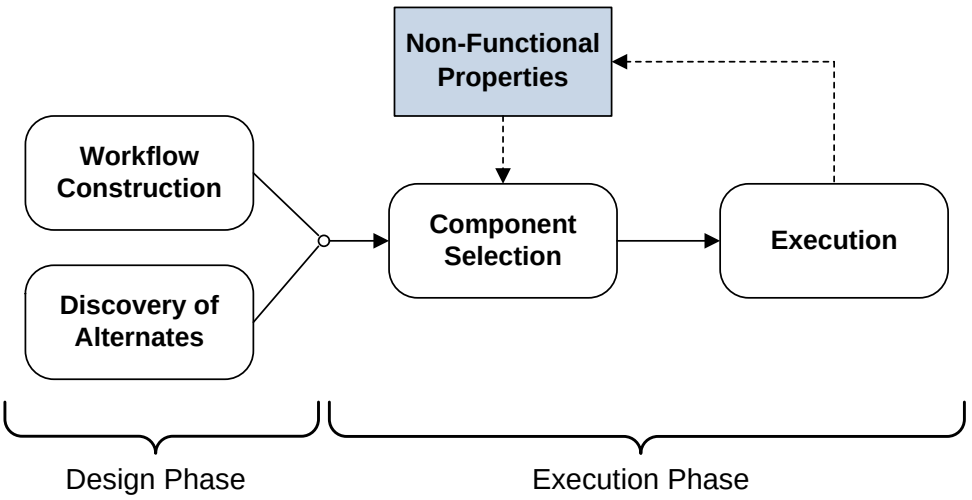


Figure 3.14: Dynamic component selection method.

In the design phase, workflow of a composite application is constructed based on the application's functional properties. After constructing a workflow, a set of functionally equivalent components is procured (constructed or located as a resource on the Web). This step can be done after the worflow has been constructed on a functional level or concurrently with the workflow construction. In effect, by attaining a set of functionally equivalent components, software redundancy is introduced into the application design. In the execution phase, components are selected so that each functionality is performed by a single component, i.e. an execution plan is constructed, based on components' non-functional properties. Specifically, a fault forecasting model is used to construct an execution plan in a way it has the greatest probability of a failure-free execution. The process of component selection is further clarified by Figure 3.15. The presented workflow defines 3 component groups S_1, S_2, and S_3 each containing a certain number of functionally equivalent components. The workflow states that groups S_1 and S_2 are to be run in sequence, while the group S_3 is to be run concurrently. At execution, based on the current non-functional properties, components $S_{1,1}$, $S_{2,5}$, and $S_{3,3}$ are selected generating the following execution plan $(S_{1,1} - S_{2,5}), S_{3,3}$.

The presented method implies that not all of the redundant components are run during execution, but only those whose combined effect has the greatest probability to provide correct service, thus, reducing resource consumption. The results of execution can be used to update the non-functional parameters or some other monitoring method may be applied. Non-functional properties encompass the dependability properties but are deliberately stated differently as they can be expressed through technical properties (e.g. throughput, latency, jitter or others). These values are in a direct relation with the presented dependability properties.

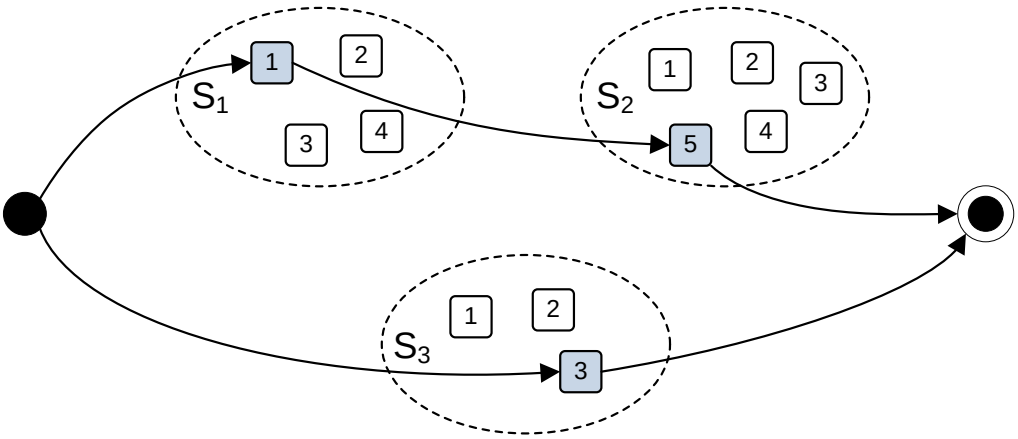


Figure 3.15: Dynamicaly generated execution paths.

The method of dynamic component selection has been widely applied and researched in the context of service-oriented computing [226–237]. For that reason, the state-of-the-art research efforts will be presented in the context of dynamic service compositions.

The first dominant research effort is focused on the design phase, specifically on procurement of functionally equivalent services (components). In the context of service-oriented systems, the problem of service procurement can be solved through discovery of service alternates deployed on the Web, rather than by constructing and deploying new service instances or versions. Sets of service alternates maintained in such a way are also referred to as homogeneous *service communities* [228, 238]. However, discovering and maintaining *service communities* is a non trivial task [239–241] that can become resource demanding when constructing service compositions on a larger scale or in pervasive computing environments [242]. This issue can become more pronounced when *service community* discovery is performed in environments with possibly lacking or inaccurate service meta-data [239, 243]. Therefore, in such cases the level of redundancy assumed by dynamic service composition strategies can be difficult to achieve or may be reached at a high expense of system resources.

The other research effort targets properties of the service selection process. Generally, service selection can be performed periodically or at each execution. In case an execution plan is to be computed frequently, it has to be computationally efficient. One way to tackle the service selection problem is to regard it as an optimization problem. However, QoS (*Quality of Service*) [244] inspired dynamic service compositions often rely on computationally demanding optimization strategies. A promising approach to improve the scalability of QoS driven dynamic composition is presented in [235]. The authors propose a hybrid approach to dynamic service selection by combining global optimization with local service selection methods. The

presented approach displays scalability improvement at a cost of accuracy (i.e. best execution plan is not always selected). Other optimization strategies, such as hybrid genetic algorithms have been reported in the literature [245]. Again, their greatest shortcoming is in substantial resource consumption that can escalate in dynamic environments.

The complexity of deriving the most reliable execution path stems from the need to reconfigure service compositions at run time. Such reconfigurations introduce additional issues, like statefulness and composability [226]. Both statefulness and composability introduce restrictions in execution plan construction by limiting component selections in certain composition execution phases. In case of composability, the selection is limited by service's functionalities. For instance, let a service composition run operations a and b, both provided by service S_1. Also let service S_2 provide a single operation b. In case service S_1 is to be replaced by service S_2 the resulting configuration would not be composable, i.e. operation a would not be supported by service S_2. In that case it is necessary to add an additional service that provides operation a, possibly increasing costs of execution. On the other hand, in case of statefulness the limitations are conditioned by internal state of the application. For instance, a certain atomic service could be used to acquire a resource needed for the proper composition execution, e.g. a credential in an e-banking application. It may not be possible to replace such a service at all or more complex workflow interventions could be needed, e.g. *checkpointing* [246].

Because of the impacts of composability and statefulness, reliability models used to handle dependability of service compositions at run time are dominantly state-based. One of the principal disadvantages of state-based models is their proneness to state explosion for more complex composite workflows. A promising approach to handle state explosions was given by Mansour and Dillon in [247]. The authors propose a reliability model based on the bounded set technique. By applying the suggested approach the number of states in a model can be significantly reduced. However, it is necessary to classify a subset of all possible states as either operational or failure states—some services need to be operational in certain states, while other services are are not important. The classification is preformed by performing observations, i.e. monitoring whether the composition is in a failure state if only a certain subset of atomic services is operational. Such approach can prove be impractical when performing monitoring leads to a significant decrease in performance of underlying atomic services. By applying the method proposed in this paper, monitoring and state related issues could be reduced by focusing on a subset of atomic services.

In conclusion, it should be noted that dynamic component selection is actively applied in service-oriented computing and is currently being intensively researched. However, many research questions in this field remain unanswered and the current state-of-the-art dynamic service composition models are not very well suited for supporting sustainable development of large-scale hierarchical composite applications.

Chapter 4

Dependable Consumer Computing

As discussed in Chapter 2, consumer applications are considered to be composite systems constructed out of a certain set of building components, namely entities that encapsulate a specific functionality. Apart from that, architecture of consumer applications is devised to be hierarchical, meaning that application's building components can be atomic (standalone, indivisible applications) or composite (applications constructed out of other composite or atomic applications). An illustration of such an architecture is presented in Figure 4.1, having the composite application at the top and its building components throughout levels $1, \ldots, n$. In essence, the presented composite architecture enables component reusability, which is a pivotal concept in consumer computing. Reusability enables creation of complex and highly customized workflows as applications are not built from the ground up, but rather using other specialized workflows, thus, reducing the required development effort. Although reusability is crucial to enabling sustainable application development, this concept also theoretically entails that applications would get increasingly depended on a large number of atomic components, as the nesting factor n in Figure 4.1 increases.

One of the major obstacles in freely applying the reusability principle is that dependability properties of each individual atomic component impact the overall dependability of a composite application [248]. In fact, with the increase in the number of atomic components, dependability properties generally decrease, as each component introduces its own threats. In extreme cases, decrease in dependability can render the constructed applications unusable in future designs, effectively preventing their reusability. Therefore, it is necessary to provide design principles that would enable construction of composite application through proper combination of building components' dependability properties. That way, a sufficient level of overall composition

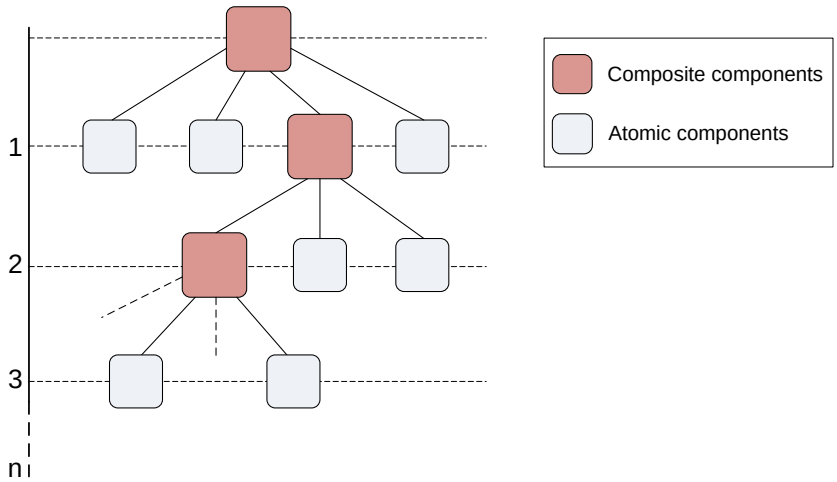


Figure 4.1: Architecture of composite based applications.

dependability can be achieved in order to ensure possibility of composite application's reuse.

The goal of this thesis is to put forward a method that would enable design and implementation of dependable applications within the consumer computing environment. The proposed method focuses on reliability as the primary dependability property, although the presented concepts are applicable to other properties, as well as general QoS attributes. In essence, the proposed method enables development of consumer applications whose building components and their interactions ensure an adequate level of reliability.

The following sections briefly sum up the state-of-the-art challenges in attaining dependability of component-based systems similar to consumer applications and present the proposed reliability management method.

4.1 Challenges in Solving Similar Problems

Since consumer computing applications have a component-based architecture, the problem of attaining dependability can be considered similar to the approaches taken in general purpose component-based systems. To support this claim, Figure 4.2 shows that a level of equivalence exists between consumer computing applications and general component-based systems, both in their architecture and design process. Regarding the design process, in both cases consumers and engineers construct applications (a) by combining the building components that are available in component repositories (b). For instance, in case of the Geppeto consumer computing environment, applications are constructed by combining web widgets using GUI-level operations. On the other hand, in the general context of composite-based systems, applications are

constructed by linking components using their exposed APIs, that are mostly understandable to engineers, e.g. by applying SOA [91] or CORBA [249] principles. In both cases the constructed applications can be submitted to repositories for further reuse (c). Regarding the architecture, both consumer applications and component-based systems in general foster application reuse, enabling design of hierarchical composite systems. Moreover, consumer applications can be considered as an extension to the existing component-based systems. This is due to the fact that atomic consumer components may be designed to access the functionality of underlaying components, e.g. web services, through their well-defined APIs. That way, a consumer can be effectively creating an underlaying composite-based system while in actuality constructing a consumer application by combining intuitive building components (d).

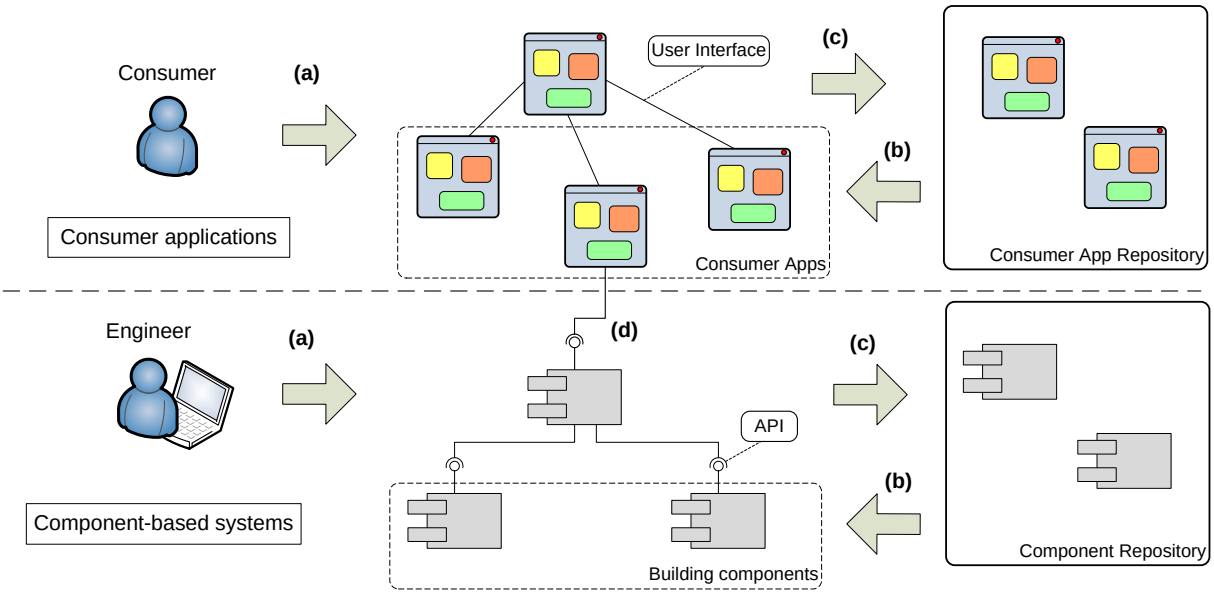


Figure 4.2: Similarity of consumer and component-based applications.

For the stated reasons, research efforts in component-based systems can be used to give insight into solving problems in the consumer computing domain. However, there is an additional characteristic that needs to be taken into account when considering dependability of consumer applications. It can be expected that consumer applications are accessed through a communication infrastructure (e.g. over the Internet), while being executed remotely or locally. For instance, in case of the Geppeto consumer computing environment, widgets are loaded from a remote server and executed locally in the consumer's browser. In such dynamic environments where multiple users can access the same resource over a shared medium, dependability parameters can change rapidly and unexpectedly. For that reason, applying only fault removal, prevention or forecasting dependability means will not necessarily secure an adequate level of

dependability parameters, namely reliability. To address the issue of attaining dependability in such dynamic environments a common method includes applying redundancies, i.e. the fault tolerance method. However, applying redundancies to support either forward of backward recovery requires additional resources and, thus, in some cases is not a viable solution (e.g. payed services).

One possible way to address dependability issues in such dynamic environments is to observe solutions applied in SOA-based systems. Since they are accessed over a shared communication medium by multiple users, web services are composite-based systems with characteristics most similar to consumer applications. To reduce the resource requirements necessary for utilization of traditional fault tolerance methods, a principle of dynamic service compositions is applied. Dynamic service compositions consider generating execution plans over a set of functionally equivalent services. That way, at execution only the services with best non-functional parameters are invoked, securing better overall dependability while consuming less system resources. However, such an approach entails runtime reconfigurations that in turn require state based dependability models or complex optimization strategies to determine viable execution plans that retain appropriate dependability properties. The main drawback of these approaches, as stated in Section 3.6, is that they do not scale well with the increase in application complexity, namely with the increase in the number of building components and workflow constructs. For instance, using state based models leads to state explosions, potentially making the models intractable, or alternatively, the problem space for optimization methods can extend rapidly. Since a comprehensive solution to solve this issue does not exist, dynamic service composition approach is not well suited to enable development of large-scale composite consumer applications.

4.2 Reliability Management Method for Consumer Applications

In order to support development of reliable consumer applications, it is necessary to avoid the issues of high resource consumption present in traditional fault tolerance and dynamic component selection methods, e.g. solutions that entail reliability models prone to state explosions. To meet these demands, a design time iterative reliability management method for consumer applications, based on research findings in [250, 251], is introduced. Since it is performed at

design time, the method applies a stateless reliability model that is focused on analyzing the impact individual application's execution paths have on the overall reliability, rather than assessing the probability a particular erroneous state will occur. By avoiding the need to model application's states, the applied reliability model scales much better with regard to increase in the number of building components and workflow complexity. The other benefit of the suggested method comes from its iterative nature. Since it is not possible to avoid introduction of fault tolerance to guarantee reliability in dynamic environments, iterative modifications to the application are used to focus the redundancies in a way they achieve highest reliability increase. Since that way it is possible to improve only the most relevant parts of an application, the desired reliability properties can be reached by spending less additional resources.

The introduced reliability management method is in fact an iterative application improvement method, motivated by the fault removal cycle presented in Figure 3.3. The difference is that instead of detecting and mitigating faults at each iteration, most influential components based on the impact on the overall reliability are detected and then improved. The component improvement step is conducted by replacing a component with a more reliable one or by applying some of the existing dependability attainment means, e.g. fault tolerance methods or dynamic component selection. Therefore, it should be noted that the presented approach does not replace the existing reliability attainment means, but rather enhances sustainable development of very large applications by focusing their use and, subsequently, containing resource consumption.

An overview of the reliability management method for consumer applications is presented in Figure 4.3. The method consists of three main processes defined as *reliability estimation*, *weak point recommendation*, and *reliability improvement*. The method is initiated by the *analysis requester* process (1) indicating that reliability of an application should be assessed. First the overall reliability for the given application is estimated (2) by the *reliability estimation* process. Based on the estimated reliability value, the *improvement requester* process decides if the application should be improved (3a) or if no further improvement actions are necessary (3b), ending the cycle. In case improvement is necessary, *weak point recommendation* process is executed, yielding a list of application's weak points (4). List of weak points is defined as a list of application's building components ordered by their influence on the application's overall reliability. The goal of the subsequent *reliability improvement* process is to improve most influential weak points to gain the greatest reliability increase. Upon performing the necessary improvements,

new application design is reevaluated (5) and the whole iteration is repeated until the desired reliability value has been reached or there are no viable improvement options.

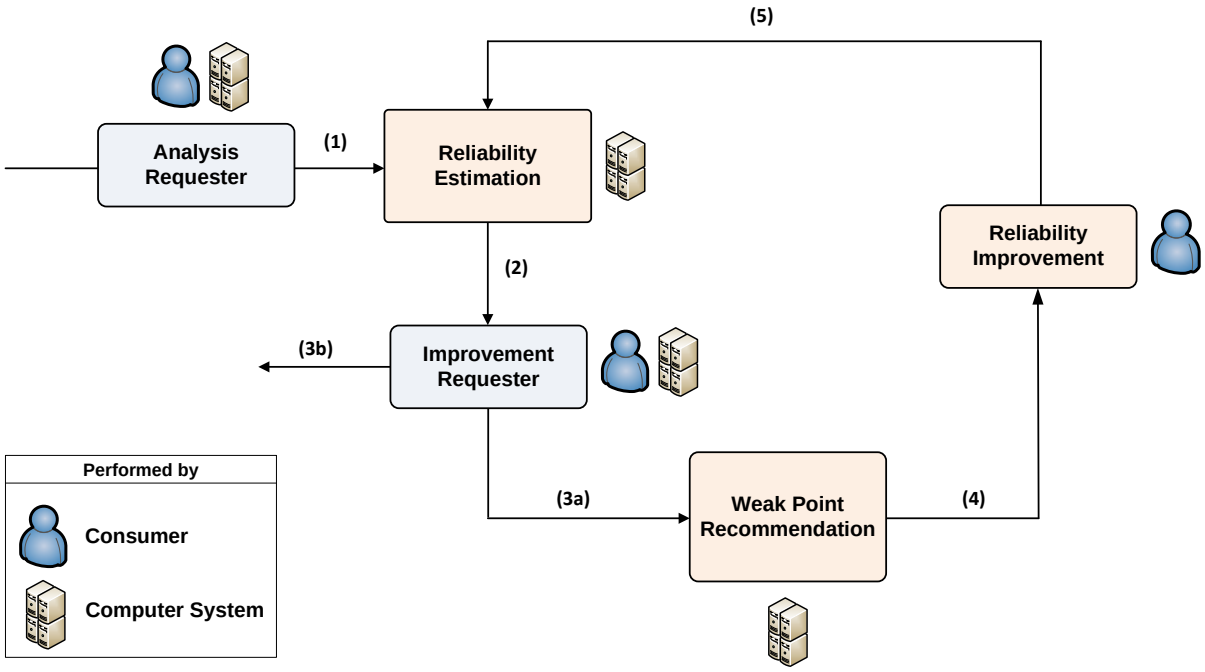


Figure 4.3: Reliability management method for composite consumer applications.

In addition to the presented method definition, Figure 4.3 defines what method processes are executed by a consumer or a computer system. Although in a general sense, each of the processes could be performed by consumers or computer systems, the method is designed to be in alignment with currently available technologies. For instance *analysis requester* and *improvement requester* processes can be performed by both consumers and computer systems. A consumer can initiate the analysis on his own request, while a computer system can be configured or trained to trigger the analysis if certain conditions are met. Similarly, consumer or a computer system can terminate the reliability management method once an appropriate level of reliability has been reached. On the other hand, steps of *reliability estimation* and *weak point recommendation* are performed by computer systems as they are purely computational tasks. On the other hand, the process of *reliability improvement* (implementation of improvements) is performed by a consumer. This is due to the fact that consumers, unlike computer systems, posses exact semantic knowledge needed to properly incorporate new components or fault tolerance constructs into an existing workflow. Although such a process could be theoretically performed by automated composition frameworks, e.g. automated service compositions, state-of the-art research results do not provide a solution to fully exclude humans from the de-

velopment process [252]. In fact, the problem is exacerbated when it comes to components that do not have well-defined interface representations (e.g. widget GUI elements opposed to web service interface described by WSDL [253]).

In the rest of the section, each reliability management method process is described in a greater detail.

4.2.1 Reliability Estimation

The *reliability estimation* process, as presented in Figure 4.4, is triggered by the *analysis requester* process (1), that can be ether a consumer or a computer system. Central to the *reliability estimation* is the *reliability estimator* process. The *reliability estimator* process requires two pieces of information in order to estimate the overall reliability of a composite application. First of all, a reliability model of the evaluated composite application is needed. If such a model does not exist or is outdated (application's structure has changed), *reliability model generator* process translates the application's definition (2) into representation of the corresponding reliability model (3). Apart from the reliability model itself, in order to estimate the overall reliability of a composite application, it is necessary to obtain reliability values for each of its building components (4). By having all the required information, the *reliability estimator* process is performed by feeding the atomic component reliability values into the model (5). The calculated composite application reliability value is then passed to the *improvement requester* process that that decides whether further improvement actions are needed.

Two crucial issues stand out in the described process. The first issue is related to obtaining reliability values for atomic building components. As stated previously in Section 3.4.2, determining reliabilities of atomic building components in environments where dependability properties frequently change is not a trivial task. A separate research activity was conducted at FER-CCL with aim of deriving an accurate and scalable method for reliability prediction of atomic building components [185]. Therefore, it is considered that atomic components' reliability values are available and this issue is not further addressed by this thesis.

The other issue is related to the applied reliability model. As stated previously, it is not feasible to describe large applications using state-based models in order to fully analyze their execution scenarios. Rather than trying to identify error states that lead to failures, a path based model is applied with aim of determining how much each building component impacts (decreases) the overall composite application reliability. As part of the presented solution a

reliability model for composite consumer applications based on belief networks is introduced. A formal definition of the applied reliability model is given in Chapter 5. Apart from that, a generalized implementation of the *reliability model generator* that creates reliability models out of UML activity diagrams is presented. Finally an example of *reliability model generator* for the Geppeto consumer applications is presented.

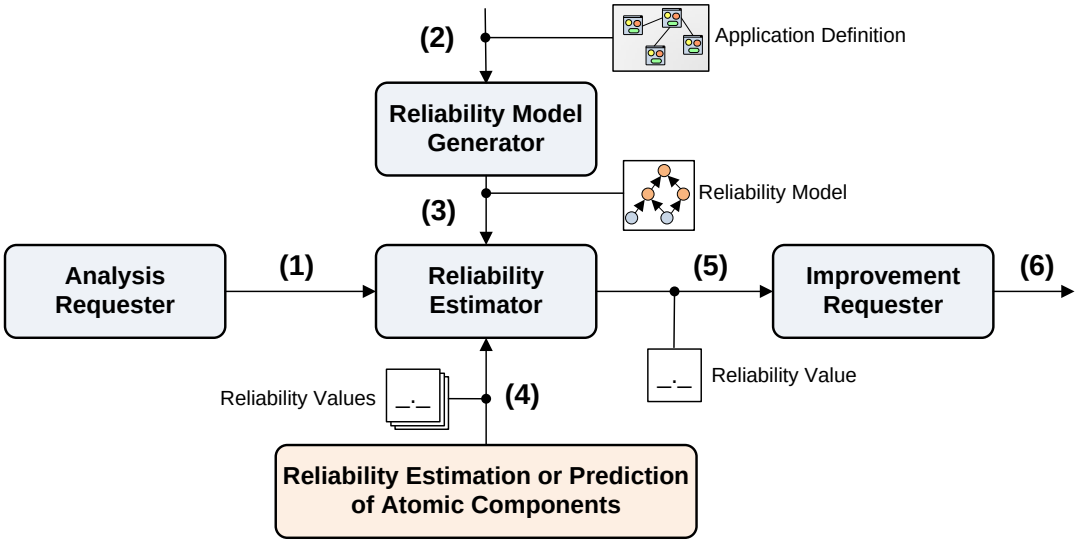


Figure 4.4: Reliability estimation process.

4.2.2 Weak Point Recommendation

In order to facilitate focused application improvement, as defined by the reliability management method, it is necessary to determine how do particular building components impact the overall composite application's reliability. To that end, the term of weak point list is defined as follows:

Definition 1 ***Weak point list*** *is a list (sequence) of composite application's building components ordered by the priority in which they are to be improved:*

$$WpL = (s_1, s_2, \ldots, s_n) \quad | \quad \textit{if } i < j, \;\; s_i \textit{ is to be improved before } s_j. \tag{4.1}$$

Definition 2 ***Optimal weak point list*** *is a list (sequence) of composite application's building components ordered by their exact influence on the composition's overall reliability, i.e. application of this list yields the highest reliability growth in each improvement step:*

$$OWpL = (s_1, s_2, \ldots, s_n) \quad | \quad \textit{if } i < j, \;\; RelImp_k(s_i) \geq RelImp_k(s_j) \tag{4.2}$$

where $RelImp_k(s_i)$ is the overall application reliability achieved if component s_i is improved in step k.

Having in mind the stated definitions, the goal of *weak point recommendation* process is to recommend a composite application's weak point list taking into account both its accuracy and computational performance. The *weak point recommendation* process, presented in Figure 4.5, is initiated when further improvements to the application are necessary, as specified by the *improvement requester* process (1). Similar to *reliability estimator*, the *weak point recommender* process requires both the data on reliability model of the evaluated application and its building component's reliabilities. In case the application has not been modified since *reliability estimation*, the same reliability model can be utilized (2). Otherwise, it is necessary to generate a new model using the *model generator* process. Similarly, atomic building components' reliability values can be reused or reacquired if their values are no longer valid (3). Finally, by analyzing the reliability model along with building components' reliability values, the *weak point recommender* process produces an ordered list of weak points (4).

The issues of reliability model and attainment of atomic building component reliability values have been previously discussed. One issue that remains to be addressed is related to computational performance of the *weak point recommender*. Although the applied reliability models are designed to avoid state explosions, their increase in size can still lead to scalability issues. To tackle this problem, *weak point recommender* process incorporates a suite of weak point recommendation algorithms with varying accuracy and computational complexity. In essence, heuristic algorithms are applied to improve the performance of the recommendation process, while reducing the accuracy of the recommended solution (suboptimal weak point lists are recommended). That way it is possible to achieve better computational performance when performing a higher number of less accurate application improvement steps is less resource demanding than performing a lower number of highly accurate improvement steps. More details on the proposed weak point recommendation methods are presented in Chapter 6.

In the context of a general consumer computing environment, both the *reliability estimation* and *weak point recommendation* processes in effect constitute a digital assistant. It is visible from Figure 4.3 that this assistant can be controlled by a consumer through *analysis requester* and *consumer requester* processes. Therefore, an implementation of such an assistant is needed within the consumer computing environment. More on the proposed consumer assistant can be found in Chapter 7.

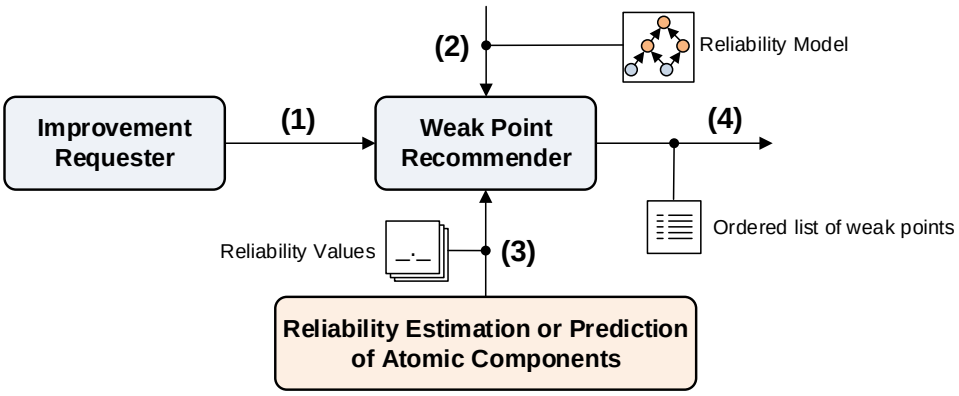


Figure 4.5: Weak point recommendation process.

4.2.3 Reliability Improvement

The final step of the reliability management method is the *reliability improvement* process, presented in Figure 4.6. Its goal is to perform actions that lead to increase in reliability of the recommended weak points. This is done by traversing the weak point list to find the first entry for which a valid reliability improvement solution exists. To that end, the improvement solution is defined as follows:

Definition 3 ***Improvement solution*** *is a modification to the composite application that results in a higher estimated overall reliability. Thus, a valid improvement solution is considered to be the one that has higher reliability than the recommended weak point.*

In general, improvement solution can constitute of a simple component replacement, i.e. a component is replaced with a more reliable one, or by introducing fault tolerance strategies that are at developer's disposal. More on the subject of implementing reliability improvements in consumer computing environments can be found in Section 7.2.

The *reliability improvement* process is separated in two segments: *weak point selector* and *application improver*. The *weak point selector* process first filters the received list of weak points (1) and selects the most viable improvement candidates (e.g. components with the highest influence factor) (2). This step is necessary as if improvement solutions are unknown, the *application improver* needs to first find appropriate replacements for the selected weak points or suggest other structural solutions that would increase the reliability. Since in general discovering similar (or functionally equivalent components) is a resource demanding task, as discussed in Section 3.4.2, it is necessary to first reduce the list of weak points to most viable candidates. Thus, based on the reduced list, the *similar component recommender* process returns the list of

possible replacements or other improvement solutions (3). In order to argue about the improvement solutions' appropriateness, it is also necessary to determine their individual reliability (4). Finally, based on the available improvement solutions, the *application improver* makes changes to the application's workflow and outputs the new application definition (5).

As issues of determining reliability of building components have been discussed previously, the challenge of similar component recommendation remains to be addressed. Although similar component recommendation is not the focus of this thesis, extensive research efforts in that direction have been conducted at FER-CCL. Specifically, two types of recommenders have been constructed implementing both digital and human-based assistants. The digital assistant [254] detects similar components by observing structures of other available composite applications. Its presumption is that similar components are used in similar application contexts, which can be ascertained form the application's morphology. On the other hand, human-based assistant [255] suggests other consumers that have used the same or similar components in similar application development efforts. Such consumers are deemed to be capable of providing help in discovering other useful components or by suggesting structural changes to the application that would lead to higher overall reliability.

As stated previously, the step of reliability improvement is conducted by consumers as they best understand the semantics of building components needed to implement the improvements. Therefore, as part of this thesis, a set of programmable consumer computing elements has been designed to support introduction of redundancies into the application's workflow. More on the programmable consumer computing elements can be found in Chapter 7.

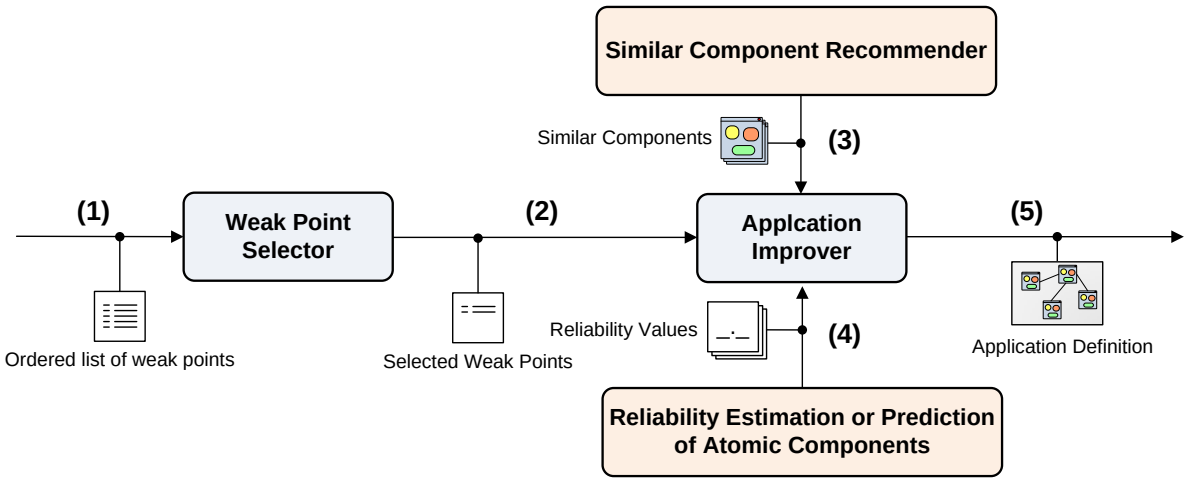


Figure 4.6: Reliability improvement process.

4.3 Taxonomy of Dependable Consumer Computing

Taxonomy of dependable consumer computing with focus on dependability attainment means is presented in Table 4.1. The goal of the presented table is to show how do concepts of dependability attainment in consumer computing relate to the general taxonomy of dependable computer systems, as presented in Chapter 3. The presented means are grouped into four common classes: fault prevention, fault removal, fault forecasting and fault tolerance. The table shows whether a specific dependability attainment mean exists within the consumer computing and whether it is performed (implemented or executed) by a consumer or performed (executed) by a computer system. Finally, the column *thesis contribution* outlines how the presented reliability management method advances the state-of-the-art in dependable consumer computing.

Regarding fault prevention, it should be noted that these means are performed by developers, as their goal is to prevent or reduce introduction of faults during design time. However, it is not conceivable that consumers will produce system specifications, use structured approaches to system design and development, as well as use formal methods, i.e. use mathematically tractable means to validate the development process. For the presumed complexity of consumer application development, these methods do not contribute to dependability. However, consumers do utilize the concept of software reusability. In fact, this concept is inherit to the consumer computing environment, as already existing applications are used as building blocks. By using applications that have been tested and used extensively, it can be expected that less faults will be introduced into the constructed workflow.

Fault removal means consider methods used to remove faults once a system has been developed. They are divided into static and dynamic verification and validation methods. Static methods evaluate the system without the need to run it. It is not probable that consumers will use formal methods, like theorem proving or model checking as they require extensive engineering knowledge. On the other hand, some aspects of static analysis are applicable to consumer computing. For instance, processes that stem form the open-source development can be utilized as other consumers might inspect shared applications and suggest improvements. Regarding dynamic verification and validation methods, the dependability attainment means require system to be run while its outputs are tested for failures. Similarly like for software reusability, this is an inherent process of consumer computing. The consumers construct applications while applying the *trial and error* approach. As consumers construct the applications, they run it several times to make sure it performs properly. These actions are equivalent to running tests in

traditional development environments.

In case of fault forecasting, two classes of dependability attainment means can be identified. Qualitative means are performed by developers and they require classifying system faults by their severity, impact and probability. It is not probable that consumers would be willing to perform such analysis. On the other hand, quantitative fault forecasting means are widely covered by consumer computing. The estimation approach implies estimation of dependability properties based on the available collected data. If a consumer application is described by a dependability model, computer system can compute an estimation of its dependability properties. Acquiring the actual values of dependability parameters can be performed in consumer computing both by computer systems and humans. Computer systems can implement a variety of monitoring processes used to collect dependability data. On the other hand, consumers can also contribute to the values of dependability parameters by expressing their satisfaction with applications' quality through rating systems. In effect, these actions can be considered as tests, similar like in fault removal, but with emphasis on application quality rather than functional properties. Finally, in cases the actual data for the dependability parameters is hard to collect, prediction methods can be run by computer systems to predict dependability property values, as described in [256].

Fault tolerance defines a set of means used to ensure that systems perform properly during run time. Such methods imply introduction of a redundancy into the workflow execution. As this is a necessary way of ensuring dependability in dynamic environments like consumer computing, consumers need to implement the redundancy using either backward or forward recovery principle. The implemented redundancy is then executed automatically by a computer system at system run time.

As defined by the *thesis contribution* column the presented reliability management method is in fact a combination of fault forecasting and fault tolerance dependability attainment means. Quantitative fault forecasting (estimation) is used to determine the overall reliability of a composite application based on reliability of its building components. Furthermore, estimation means are used to determine the impact the individual building components have on the overall composite application dependability. These actions can be entirely performed by a computer system and are, thus, implemented as a digital consumer assistant. Furthermore, to implement the necessary redundancies, consumers need to have specialized programmable elements at their disposal. These elements are incorporated into the workflow and automatically secure

a desired level of dependability properties. For that reason both the *consumer* and *computer system* columns are marked in the table. The thesis introduces programming elements that enable both backward and forward recovery through checkpoint and restart, recovery blocks and n-version programming principles. Since in order to implement backward recovery methods it is necessary to provide acceptance tests, these tests need to be defined by a consumer. The tests are then run by a computer system in order to determine if an application displays erroneous behavior. For that reason, the testing field of fault forecasting is also marked in the table as this is where fault tolerance and forecasting methods intersect.

Table 4.1: Taxonomy of dependable consumer computing

Dependability Attainment Mean			Performed By		Thesis contribution
			Consumer	Computer System	
Fault Prevention	System requirement specification				
	Structured system design				
	Structured development process				
	Formal methods				
	Software reusability		+		
Fault Removal	Static verification and validation	Static analysis	+		
		Theorem proving			
		Model checking			
	Dynamic verification and validation	Testing	+		
Fault Forecasting	Qualitative	Failure effect analysis			
	Quantitative	Estimation		+	+
		Testing	+	+	+
		Prediction		+	
Fault Tolerance	Backward recovery	Checkpoint and restart	+	+	+
		Recovery blocks	+	+	+
	Forward recovery	N-version programming	+	+	+

Chapter 5

Reliability Model

This chapter presents key concepts required to implement the *reliability estimation* process whose goal is to assess the overall reliability of a composite consumer application. Central to *reliability estimation* is the *reliability estimator* process. Its purpose is to process the graphical reliability model, and based on its defined rules compute the overall application reliability. Since the reliability model has to be able to describe very complex consumer applications, i.e. applications composed out of a large number of atomic building components, a path based reliability model founded on belief networks is selected. This approach is chosen in favor of the state based reliability models, prone to state explosions that limit their usability. A formal definition of reliability model for composite consumer applications is presented in Section 5.1. Furthermore, the model extensions that take into account mutual dependence of atomic components are presented in Section 5.2. The introduced reliability model is generated out of a consumer application workflow representation, e.g. Geppeto spreadsheet, by the *reliability generator* process. Since consumer applications can be stored using various representations, to maintain generality, an algorithm that generates reliability models form UML activity diagrams is presented in Section 5.3. Finally, the chapter is concluded with an example in Section 5.4 showing how can reliability models be generated for Geppeto consumer applications.

5.1 Formal Definition

In this section a formal definition of the reliability model for composite applications based on the principle of belief networks (Bayesian networks) is presented. The model is defined as a directed acyclic graph $M = G(V, E)$, where V is a finite set of nodes (vertices) representing

random variables and E a finite set of edges representing the dependency between random variables. For instance, a directed edge pointing from random variable Y_j to X_i denotes that random variable X_i is conditionally dependent on the random variable Y_j. Throughout the rest of the section the terms *model node* and *random variable* will be used interchangeably. An illustrative example of the reliability model for composite applications introduced in this section is given in Figure 5.1. Each reliability model has multiple input random variables representing the reliability of atomic components or a specific fault tolerance construct and a single output variable denoting the overall composite application reliability. A layer of hidden nodes is constructed between input and output nodes based on the composite application's workflow. Specifically, the nodes in the hidden layer define how a particular input random variable, i.e. atomic component, impacts the reliability value denoted by the output random variable based on workflow scenarios in which the given atomic component is used. The paths leading form input random variables to the single output random variable are called influence paths throughout the rest of the thesis.

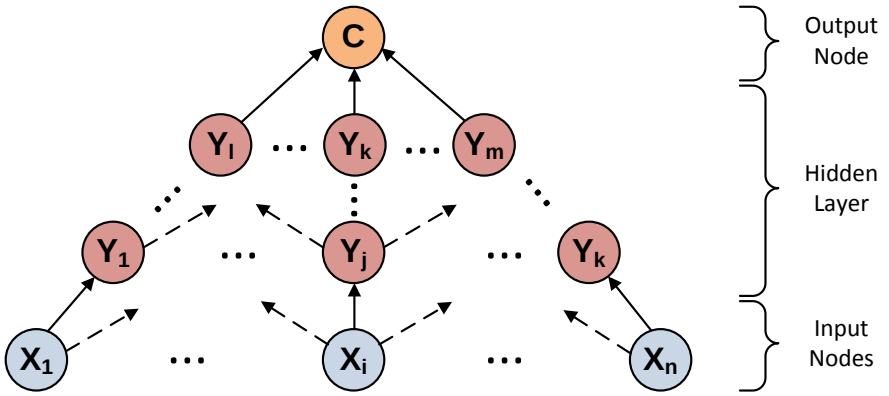


Figure 5.1: Reliability model for composite applications.

A set of possible states Ω is defined for each random variable $X \in V$ as:

$$\Omega = \{\text{normal operation}, \text{fault occurrence}\} \tag{5.1}$$

Therefore, each random variable in the model specifies the likelihood that the composite application or a subset of its workflow is functioning according to the specification. Based on the state space Ω, we define the random variable X_i as follows:

$$X_i(\gamma) = \begin{cases} 1, & \gamma = \text{normal operation} \\ 0, & \gamma = \text{fault occurrence} \end{cases} \tag{5.2}$$

Since random variables X_i are discrete, we introduce a probability mass function as defined in (5.3). The probability of an outcome in which a random variable X_i is assigned the value 1, $p(X_i = 1)$, is in fact equal to the reliability of the composite application or its subset represented by the random variable X_i. Similarly, the probability of the outcome $X_i = 0$ is equal to the probability of fault occurrence in the composite application or its subset represented by the random variable X_i. For space considerations, the outcomes $X_i = 1$ and $X_i = 0$ will be denoted X_i and $\neg X_i$, respectively throughout the rest of the section.

$$\rho_{X_i}(x) = \begin{cases} p(X_i = 1), & x = 1 \\ p(X_i = 0) = 1 - p(X_i = 1), & x = 0 \end{cases} \tag{5.3}$$

A set of all possible outcomes P_O for n random variables is defined as follows:

$$P_O = \{(e_1, e_2, \ldots, e_n)\} \, | X_i = e_i, e_i \in \{0, 1\} \tag{5.4}$$

where e_i is an outcome assigned to random variable X_i. Since there are 2 possible outcomes for each random variable, a set of all possible outcomes for n random variables contains 2^n elements.

By applying the law of total probability, we can calculate the reliability of a composite application or its subset represented by a random variable X_i by summing out all the possible outcomes for random variables $Y_1, \ldots, Y_n$. Therefore, $P(X_i)$ can be calculated as in (5.5):

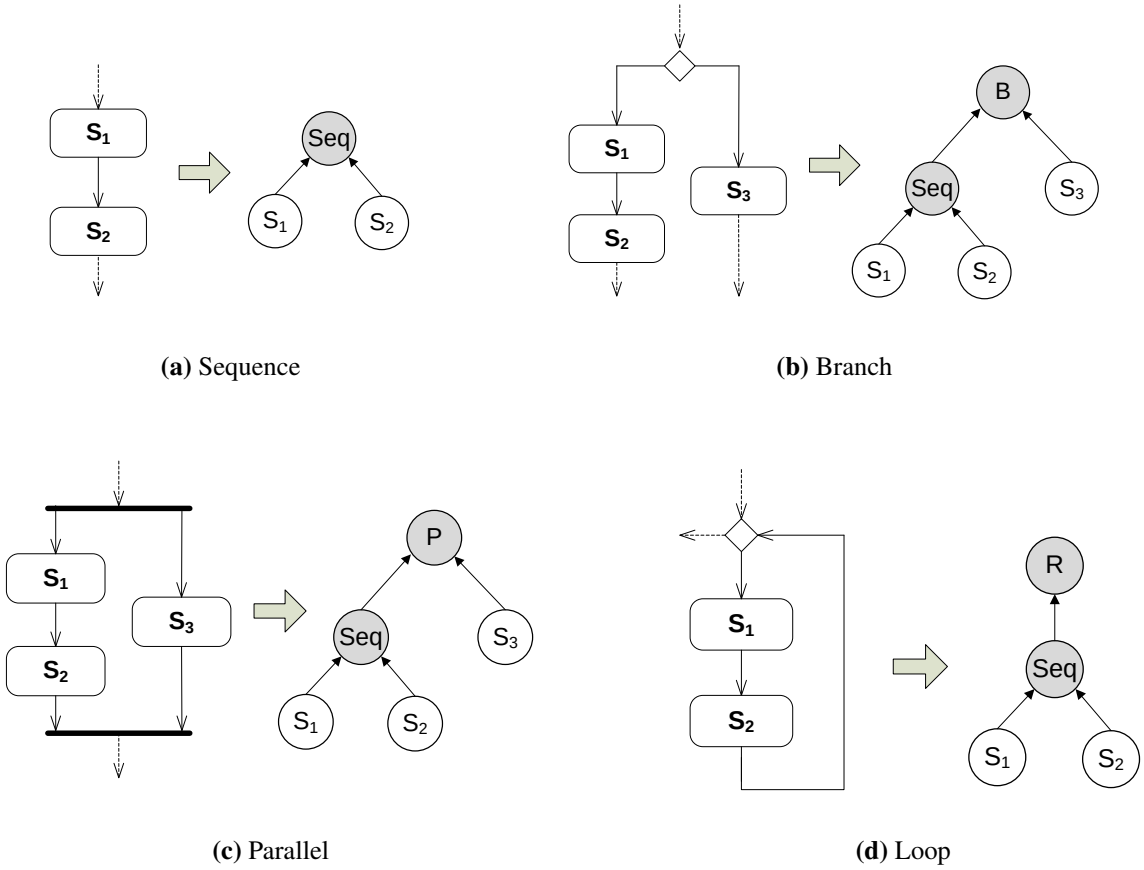
$$P(X_i) = \sum_{j=1}^{2^n} P(X_i | o_j) P(o_j), \; o_j \in P_O \tag{5.5}$$

where $P(X_i|o_j)$ is the conditional dependence of the variable X_i on the outcome o_j and $P(o_j)$ the probability that the outcome o_j will occur. If the random variables $Y_1 \ldots Y_n$ are conditionally independent the probability $P(e_1, \ldots e_n)$ is calculated as follows:

$$P(e_1, \ldots, e_n) = \prod_{i=1}^{n} P(e_i) \tag{5.6}$$

As stated previously, hidden network nodes specify influence paths from input random vari-

ables to the single model's output random variable. Therefore, the model supports several types of random variables used to model various workflow constructs present in the composite application's activity diagram, as shown in Fig. 5.2. In effect, the type of the random variable specifies how the conditional dependencies to other random variables are defined. Throughout the rest of this section we define which types of random variables are defined by the presented model.



(a) Sequence **(b)** Branch

(c) Parallel **(d)** Loop

Figure 5.2: Nodes supported by the reliability model.

Component

Component nodes are random variables that represent the reliability of atomic components used to construct a composite application. More specifically, a random variable $S_{i,j}$ represents the reliability of an operation j exposed by the component i. Since it is presumed that all atomic components are independent entities, they are considered to be conditionally independent in the model. As an implication of this hypothesis, all atomic component nodes in the proposed

reliability model have no incoming edges. Thus, the atomic component nodes are the input nodes for the presented reliability model.

Composition

Composition node C_i is the single output node for the model and it represents the overall reliability of a composite application. The node has no outgoing arcs and is therefore the only leaf in the network. Conditional probabilities for the variables associated with composition node's incoming arcs are calculated the same way as for the sequence node described in the following subsection.

Sequence

A sequence node is used to model a sequence of events present in the composition's workflow that can comprise of component executions or more complex constructs like parallel workflow paths, branch constructs and loops. An example of a sequence node modeling the sequential invocation of two components is presented in Fig. 5.2a. In order for a sequence of component executions to be successful, each components invocation has to be completed successfully, i.e. without fault occurrences in all of the sequence sub elements. Therefore, the only outcome in which the sequence is completed without causing a failure of the entire composition is $(e_1 = 1, e_2 = 1, \ldots, e_n = 1)$. Thus, the conditional probabilities associated with the sequence node's incoming arcs are calculated as in (5.7).

$$P(Seq|o_i) = \begin{cases} 1, & o_i \in P_O|e_1, \ldots, e_n = 1 \\ 0, & \text{otherwise} \end{cases} \tag{5.7}$$

By plugging (5.7) into (5.5) the reliability of a sequence node can be calculated as follows:

$$P(Seq) = \prod_{i=1}^{n} P(X_i) \tag{5.8}$$

For example, the reliability of a sequence node in Fig. 5.2a can be calculated as $P(Seq) = P(S_1)P(S_2)$.

Loop and Repetition

To model repetitive workflow structures a repeat node R is used. The repeat node has a single incoming arc for which the conditional probability is calculated as follows:

$$\begin{aligned} P(R|X_j) &= P(X_j)^{k-1} \\ P(R) &= P(X_j)^{k-1}P(X_j) = P(X_j)^k \end{aligned} \tag{5.9}$$

where k is the expected number of repetitions. The repeat node is used to model both loop constructs, as well as multiple component invocations within a sequence. Specifically, multiple invocations of the same component in a sequence can be modeled by a single repeat node with the parameter k set to the number of repetitions. For example, if the component S_1 is invoked 4 times in a certain composition scenario, the reliability of repeat node can be calculated as: $P(R) = P(S_1)^{4-1}P(S_1) = P(S_1)^4$. It should be noted that for loop constructs parameter k can not be determined from the composition's workflow, but rather it can be estimated at run time in order to fine-tune the model. As already discussed in Chapter 3, such information can be made available through the composite application's operational profile [180]. If the application's operational profile is not known, the parameter k can initially be set to 1. On the other hand, repetition through sequential executions of the same component can be calculated from the workflow representation, since k is equal to the number of repeated executions. An example of how the repeat node is generated for a loop construct in the activity diagram is presented in Fig. 5.2d.

Parallel

A composite application workflow can consist of multiple execution paths that are run concurrently. A successful invocation of a composite application generally depends on fault free completion of a subset of the parallel workflow execution paths. This means that in case there is a build-in redundancy in execution paths, e.g. access to multiple data sources, failure in one or more execution path may not cause the failure of the entire composite application. Therefore, the conditional probabilities for a parallel node P_i are calculated as follows:

$$P(P^{(k)}|o_i) = \begin{cases} 1, & if\ E_o \subset o_i | \forall e_j \in E_o, e_j = 1 \wedge |E_o| \geq k \\ 0, & \text{otherwise} \end{cases} \tag{5.10}$$

where $k \in [1, n]$ indicates how many of the n parallel workflow execution paths have to be successfully completed in order for the composition to display fault-free behavior. In other words, at least k of n random variables representing parallel execution paths have to assume value 1 for the normal composition operation. In case that $k = n$ all the parallel execution paths must be completed properly not to cause composition failure. The overall reliability for the parallel node can be calculated as:

$$P(P^{(k)}) = \sum_{i=1}^{m} P(P^{(k)}|o_i)P(o_i), \quad m = 2^n - \sum_{j=0}^{k-1} \begin{pmatrix} n \\ j \end{pmatrix} \tag{5.11}$$

where m is the total number of summation factors, i.e the total number of combinations in which k random variables are assigned value 1.

An example of a parallel node constructed out of an activity diagram is presented on Fig. 5.2c. If we assume that at least one branch has to be completed for a fault-free composite application behavior $k = 1$, the overall reliability for the parallel node P is equal to: $P(P) = P(Seq)P(S_3) + P(\neg Seq)P(S_3) + P(Seq)P(\neg S_3)$.

Branch

Branch constructs introduce uncertainty that a specific workflow sequence will be executed upon a composite application invocation. Such uncertainty reduces the impact a particular workflow sequence has on the overall reliability since it is not executed at each composite application invocation. Therefore, each branch i of the branch construct is assigned the probability of execution p_i. Similarly like for the repeat node, it is not possible to determine the probabilities p_i form the composition workflow, but they can be estimated at run-time to fine-tune the model. However if application's operational profile is not available, execution probabilities can initially be set evenly for all the branches. Specifically, the following condition must hold for the branch construct with n branches: $\sum_{i=1}^{n} p_i = 1$. Therefore, the conditional probabilities for a branch node with n incoming arcs can be calculated as follows:

$$P(B|o_i) = \begin{cases} 1, & o_i \in P_O | e_1, \ldots, e_n = 1 \\ (1 - \sum_{a \in I} p_a), & o_i \in P_O | \exists e_x = 1 \wedge \exists e_y = 0 \\ & , x \neq y \\ & I = \{a | e_a \in o_i \wedge e_a = 0\} \\ 0, & \text{otherwise} \end{cases} \tag{5.12}$$

The uncertainty of execution is modeled in a way that the outcomes in which faulty branches exist are multiplied by the probability that the branches in question will not be executed. An example of a branch construct with two incoming arcs is presented in Fig. 5.2b. If p_{Seq} and p_{S_3} are the probabilities of execution for nodes Seq and S_3 respectively, the overall reliability for the presented branch node can be calculated as follows: $P(B) = P(Seq)P(S_3) + (1 - p_{Seq})P(\neg Seq)P(S_3) + (1 - p_{S_3})P(Seq)P(\neg S_3)$.

5.2 Atomic Component Dependency

The presented reliability model assumes that all atomic components are conditionally independent of each other. Although this is a common assumption when modeling reliability of composite systems, particularly those similar to SOA applications [96], it is generally not a valid one. However, assumption of interdependency is often applied as collecting information on atomic component dependency can prove to be difficult, as such information is usually available only implicitly in process descriptions and service level agreements (SLA) [257]. Moreover, building a dependency model often requires a deeper insight into the component-based system's implementation, e.g. component (e.g. web service) deployment details. However, if information on atomic component dependency is available, it can be used to make the presented reliability model more accurate as it is well suited to take into account such influences. In the rest of this section three basic atomic service dependency cases are described along with the corresponding model constructs.

Functional Dependency

A component S_j can be functionally dependent on component S_i if during its operation S_j invokes S_i. However, in such a case S_j cannot be considered to be an atomic component, but rather a composite one. Thus, S_j is modeled using a composition node and subsequent

constructs (e.g. a sequence node) that describe how S_i is incorporated into its functionality. This case is shown in Figure 5.3. Specifically, component S_j is modeled by composition node C_j that has incoming edges from nodes S_i and S'_j, where random variable S'_j is related to internal properties of component S_j. In this case it is considered that S_i is invoked once sequentially for each invocation of S_j. Other model nodes can be added between S_i and C_j, and S'_j and C_j to take into account more complex interaction cases.

Furthermore, the model can be extended to support circular functional dependencies. However if two components are mutually recursive, their impact needs to be unrolled, similarly like for the loop construct. In that case an additional repeat node R_k needs to be added as a child of the corresponding composition node C_j.

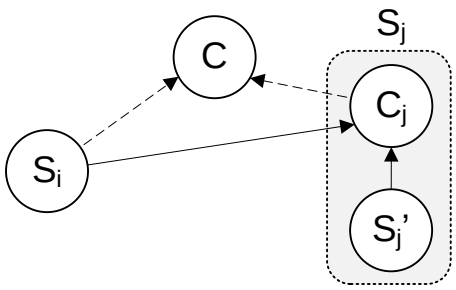


Figure 5.3: Functional atomic component dependency.

Unidirectional Dependency

In case inclusion of component S_i into composition impacts the component S_j, but S_j does not influence the component S_i, an unidirectional dependency exists. For example, such dependency can occur if components S_i and S_j share a certain system resource, but component S_i has the absolute priority in consuming it. Thus, an unidirectional dependency can be modeled as an edge leading from node S_i to node S_j, as shown in Figure 5.4. In that case, the reliability of component S_j is calculated by summing out all the outcomes of random variable S_i. If the dependency needs to be described using a random variable that can assume more than two values, an additional modifier variable m can be introduced between S_i and S_j. In that case, the reliability of S_j is calculated as follows:

$$P(S_j) = \sum_{k=1}^{l} P(S_j|m^k)P(m^k) \tag{5.13}$$

where m^k is one of the l possible outcomes for random variable m. Consequently, the probability that random variable m will assume value m^k is equal to:

$$P(m^k) = P(m^k|S_i)P(S_i) + P(m^k|\neg S_i)P(\neg S_i) \tag{5.14}$$

Since both the $P(m^k|S_i)$ and $P(S_j|m^k)$ depend on a particular dependency case, it is not explicitly stated how they are defined.

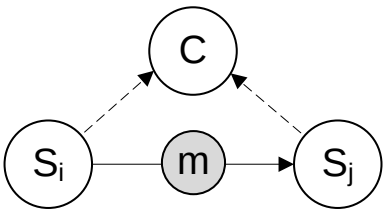


Figure 5.4: Unidirectional atomic component dependency.

Bidirectional Dependency

If including component S_i into composition impacts the reliability of S_j and vice versa, a bidirectional dependency between the components exists, i.e. the components are interdependent. For example, this scenario can occur in cases when two components share a system resource, but no component has the absolute priority in consuming it. Therefore, in such a case, by increasing reliability of component S_i, reliability of component S_j is reduced. In order to avoid model loops, such shared properties are expressed as separate random variables p_a, as presented in Figure 5.5. For instance, in the given example if component S_i changes its reliability, probability distribution of p_a can be recalculated and then in turn reliability of S_j. Thus, the reliability of a random variable S_i is calculated as follows:

$$P(S_i) = \sum_{k=1}^{l} P(S_i|p_a^k)P(p_a^k) \tag{5.15}$$

where p_a^k is one of the l possible outcomes for random variable p_a. The same expression can be applied to variable S_j. It should be noted that if probability distributions for variables

p_a are known, those variables can be used as input model random variables, instead of the atomic component variables. As it was the case for unidirectional dependencies, since $P(S_i|p_a^k)$ depends on a particular dependency case, its definition is not explicitly stated.

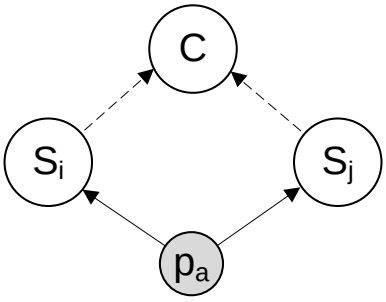


Figure 5.5: Bidirectional atomic component dependency.

5.3 Reliability Model Generation from UML Activity Diagrams

In order to provide a general context, not taking into account any particular consumer application notation, reliability model generation out of UML activity diagram models is considered. Since UML activity diagrams are a general model for representing dynamic behavior of a system, namely activities and actions that constitute a workflow, they can be applied to component-based consumer applications. Activity diagrams are built using 6 types of elements, as presented in Figure 5.6. Each activity diagram has a single *initial* and *final state*. All the workflow activities start at *initial state* and terminate at *final state* element. Furthermore, activities consist of actions that represent a certain operation, e.g. invocation of a consumer application or web service, modeled using the *action* elements. Each action can be split into multiple parallel actions by applying the *split* element. Complementary, multiple parallel actions are joined into a single action using the *join* element. Finally, certain workflow actions can be executed only if specific conditions are met. Such occurrences are modeled using the *decision* element.

An example of a consumer application represented using UML activity diagram is shown in Figure 5.7. In the presented example, activities represent operations performed by 6 distinct application-specific building components ($S_1, \ldots S_6$). From the presented model, it is visible the workflow is initiated by invoking component S_1, followed by subsequent invocation of

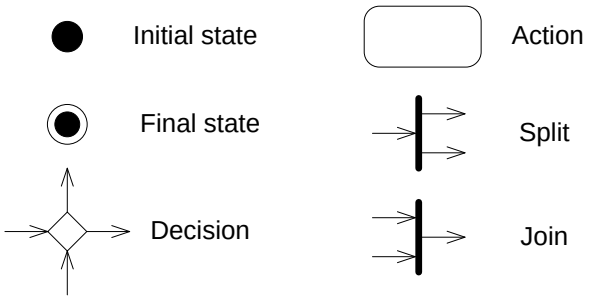


Figure 5.6: UML activity diagram elements.

components S_2. Following completion of component S_2 the activity is split into two parallel execution paths, having S_4 and S_5 invoked sequentially in the first path and S_3 invoked in the second path. The parallel activities are then joined after which the component S_2 is invoked. At this point, the activity branches into two possible execution path, out of which only on at a time is executed. The first path comprises of sequential invocations of components S_1 and S_6, while in the second path only component S_1 is invoked. Both paths lead into the *final state* element, indicating workflow completion.

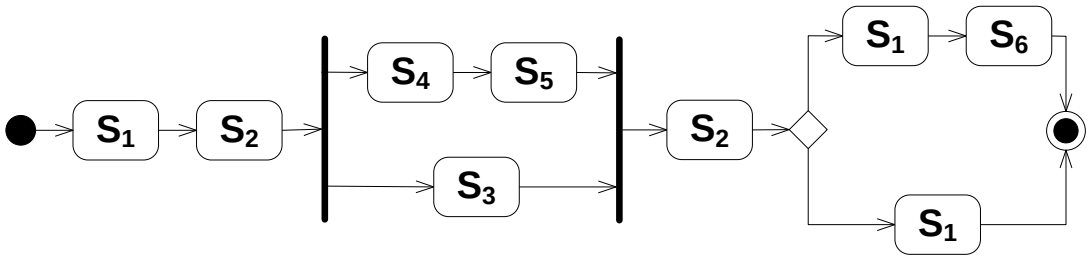


Figure 5.7: Example of a composite application

The presented UML activity diagram workflow model can be translated into reliability model introduced in the previous section. To that end, a reliability model generator can be implemented as defined by pseudocode Algorithm 5.1. At its input the algorithm receives the starting UML model node and a list of processed UML elements, required for its recursive subroutine. Initially, this list is empty and gets populated as UML elements are processed. The algorithm first processes the *initial state* node adding a new composition node for the composite application (lines $2-6$). Then, elements of the model are read sequentially until one of the stopping conditions is met. In case more than a single sequential element is read (loop is in its second iteration), a sequence node is added as a child of both elements to combine their

influence (lines $10 - 13$). Then, a new reliability model node is generated based on the read UML element.

In case an *action* element is read (line 16), first it is checked if the corresponding reliability model node already exists (line 15). If it exists, it is retrieved (line 16), otherwise a new composition or component node is created (lines $18 - 19$). Specifically, a new node is created based on the defined action. For instance, in case the action is invocation of an atomic building component, a component node is added. On the other hand, if the performed action invokes a complex component, a composition node is added and its reliability model retrieved.

When a *split* element is read (line 21), multiple parallel execution paths need to be processed. For each of the parallel paths, defined by their starting elements, the algorithm is run recursively (lines $25 - 29$). The resulting reliability model subsets are added to the parallel node and their conditional probabilities set according to the join conditions. Finally, the current UML element reference is advanced to the join element that corresponds to the processed split element.

In case of the *decision* element three cases need to be taken into account. Firstly, if the element has been already visited along the presently analyzed path (line 33), a loop in the activity diagram exists and it needs to be unrolled. This is done by placing a repeat node as a child to the existing branch node, indicating that the entire portion of the workflow can be repeated during execution. The value k of the repeat node can be set according to the composite application's operational profile. Secondly, if a branch node for the read *decision* element exist in the reliability model, but the element has not been visited along the currently analyzed path (line 38), the existing branch node is retrieved (line 39). Finally, if a *decision* element is analyzed for the first time, a new branch node needs to be created (lines $41 - 42$). Similarly like in case of a *split* element, for all the paths that lead from the *decision* element the algorithm is run recursively (line 45). Each of the generated submodels is then added to the branch node (line 46) and its conditional probabilities set according to the available application's operational profile.

Finally, the generated node is integrated into the reliability model. If multiple arcs are to be added between the two nodes in the model, to aggregate the paths a repeat node is added and its k parameter set accordingly (lines $52 - 54$). The algorithm terminates, i.e. returns the generated model, when a *decision*, *join* or *final state* UML element is read (lines $57 - 58$).

Using the model generator defined by Algorithm 5.1, reliability model for the application example in Figure 5.7 is constructed as presented in Figure 5.8. The model is a network structure

```
1: function GENERATEMODELFROMUML(e, lU)  ▷ Generate reliability model starting at the e element of the activity diagram
Require: lU  ▷ A list of processed UML elements
Require: lR  ▷ A global list of generated model elements and their links
2:   if e.type == "initialstate" then
3:     m ← new composition node
4:     lU.add(e); lR.add(m, e)
5:     e ← getNextEl()  ▷ Retrieves next UML element
6:   end if
7:   useSeq ← false  ▷ Check if a sequence node is required
8:   while true do
9:     lU.add(e)
10:    if useSeq == true and m.type! = "composition" then  ▷ Check if a sequence node is required
11:      Add a sequence node s between m and its single parent node
12:      m ← s
13:    end if
14:    if e.type == "action" then
15:      if e exists in lR then
16:        n ← get node from lR
17:      else
18:        n ← new composition or component node
19:        lR.add(n, e)
20:      end if
21:    else if e.type == "split" then
22:      n ← new parallel node
23:      lR.add(n, e)
24:      elemList ← get starting elements in all parallel paths
25:      for all p in elemList do
26:        subMod ← GENERATEMODELFROMUML(p, lU)
27:        Get the top node tn from subMod and add arc tn → n to lR
28:        Append subMod to lR
29:      end for
30:      e ← advance to join element
31:      Set conditional probabilities according to the join condition
32:    else if e.type == "decision" then
33:      if e exists in lU then  ▷ A loop exists and needs to be unrolled
34:        Get branch node b for element e from lR
35:        ch ← get child node of b
36:        Create a new repeat node r and add it to lR
37:        Remove arc b → ch and add arcs b → r and r → ch
38:      else if e exists in lR then  ▷ Element exists but this is not a loop
39:        n ← get node from lR
40:      else
41:        n ← new branch node
42:        lR.add(n, e)
43:        elemList ← get starting elements in all branches
44:        for all p in elemList do
45:          subMod ← GENERATEMODELFROMUML(p, lU)
46:          Get the top node tn from subMod and add arc tn → n to lR
47:          Append subMod to lR
48:        end for
49:        Set conditional probabilities according to the branching condition
50:      end if
51:    end if
```

52: **if** arc exists from n to m **then**
53: If an R node is in between the two nodes, increase its param k, otherwise add an R node and set param k to 1
54: **else**
55: add arc $n \rightarrow m$ to lR
56: **end if**
57: **if** $e.type ==$ "$decision$" or "$join$" or "$finalstate$" **then**
58: **return** lR ▷ Return the generated model
59: **else**
60: $e \leftarrow getNextEl()$ ▷ Retrieves next UML element
61: $useSeq \leftarrow true$
62: **end if**
63: **end while**
64: **end function**

Algorithm 5.1: UML to reliability model.

with a single output node C, representing reliability of the entire composite application, and 6 input nodes standing for atomic components $S_1, ..., S_6$. Since according to the UML activity diagram, component S_2 is invoked 2 times during each composition execution, it is placed as a child node of a repeat node with parameter k set to 1. Furthermore, it can be observed that the component S_1 is invoked on three different occasions: once per each composition execution and in two separate decision paths. For that reason a direct arc exists between C and S_1 along with two paths through the node B. Therefore, component S_1 impacts the overall application reliability through 3 influence paths. Furthermore, since S_3 is executed concurrently to S_4 and S_5 a parallel node with two parent elements is generated.

Finally, to fine-tune the presented model, an application operational profile is needed. For the purposes of this particular example, let probability of execution for both branches of element B be set equally ($p_1 = 0.5$, $p_2 = 0.5$). In addition, it is assumed that only one path of two parallel workflow paths has to complete for the composition to operate properly, thus the parallel node parameter k is set to 1.

5.4 Geppeto: Reliability Model Example

This section outlines an example how can a reliability model be constructed for Geppeto consumer applications. As previously discussed in Section 2.4, Geppeto applications are constructed using a generic programmable component *Geppeto TouchMe* that enables defining both data and control flow operations. The programmable component stores applications in a tabular representation, as shown in Figure 2.6. The actions stored in cells represent GUI-level operations that are to be executed in a strictly defined order, from top to bottom and from left to right.

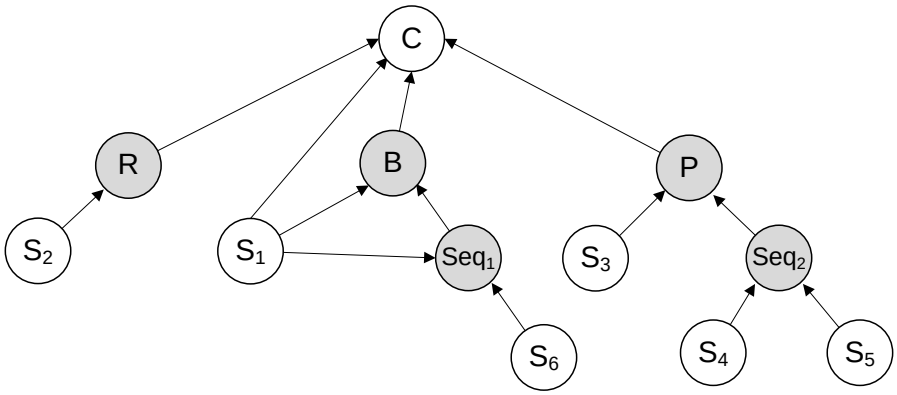


Figure 5.8: Reliability model for the composite application example in Fig. 5.7.

Although the presented representation can be directly transformed into a reliability model, to maintain generality, an algorithm that translates the tabular representation into an UML activity diagram is presented. The algorithm *GeppetoToUml* is defined by pseudocode presented in Algorithm 6.2. The generated UML diagram can be converted into a reliability model using the previously defined Algorithm 5.1. The rest of this section describes the *GeppetoToUml* algorithm and presents reliability model generation example for a Geppeto consumer application.

The *GeppetoToUml* is a recursive algorithm that takes at its input the top left cell of application's tabular representation. Depending on the cell type, two distinct operations can be performed. In case a *wait for click* cell has been read, a new *initial state* UML element is generated (line 6). Otherwise, if a *click* or *double click* cell is read, a new *action* UML element is created. This is done as *click* and *double click* GUI operations in fact represent control flow operations that are used to invoke components in the application's workflow. Depending on if a composite or atomic component is invoked, a corresponding action is generated. For instance, a composite component can represent any programmable component in the workflow, including other generic programmable components. However, different programmable components can yield distinctions in generating UML activity diagrams, e.g. the *Geppeto TriggerMe* widget implies generating a *decision* UML element. These specific workflow constructs will not be discussed in a greater detail, as programmable components can greatly vary in their purpose. Furthermore, it can be observed that *copy/paste* data flow operation are ignored in the algorithm definition. This is due to the fact that these operations can be considered to be highly reliable, as they are executed locally in consumer's browser. In case a more precise model is needed, data flow operations can also be modeled as *actions* elements of the UML activity diagram.

Moreover, each data flow operation can be considered to be a separate *action* or multiple data flow operations can be modeled as the same *action*.

```
1:  function GEPPETOTOUML(c)                          ▷ c - Geppeto spreadsheet cell
2:    m ← create new UML model
3:    n - current UML node
4:    while true do
5:      if c.type == "waitforclick" then
6:        n ← create initialstate node
7:      else if c.type == "click" or "doubleclick" then
8:        n ← create action node
9:        Set action node type composition or component
10:     end if
11:     Add n to m (link n to last element in m)
12:     b ← get bottom cell of c
13:     r ← get right cell of c
14:     if b and r exist then                         ▷ Split operation is required
15:       Add split node s to m
16:       bmod = GEPPETOTOUML(b)
17:       rmod = GEPPETOTOUML(r)
18:       Add bmod and rmod to s
19:       Add a join node to close s
20:       Add finalstate node to m
21:       return m
22:     else if b or r exists then
23:       c ← existing neighbor cell (b or r)
24:     else if no neighbor cells exist then
25:       Add finalstate node to m
26:       return m
27:     end if
28:   end while
29: end function
```

Algorithm 5.2: Geppeto to UML.

After the current cell has been processed, it is added to the UML model (line 11). The next cell in tabular representation is then selected, depending on three possible cases. In case both the bottom and right cells exist (line 14), the workflow is to be split into two parallel execution paths. First, a *split* element is added to the UML model (line 15). Then, for each of the execution paths, the algorithm is called recursively, having bottom and right cells as input (lines $15 - 16$). The resulting submodels are added to the *split* element (line 18) that is then closed with a *join* element (line 19). Finally, the UML model is concluded with a *final state* element and returned.

In the second case, when either a bottom or right cell exists, the next cell is set to the existing one. Finally, if no bottom or right cells exist, the UML model is concluded with a *final state*

node and the algorithm returns.

An example of a Geppeto consumer application defined by its tabular representation is shown in Figure 5.9. From the given tabular representation, it is visible that application *app1* is constructed out of three distinct widgets (*wid1*, *wid2*, and *wid3*). In addition, a total of 5 distinct *click* operations are performed on the widgets, i.e. operations are performed on distinct GUI elements, meaning that the resulting UML diagram also needs to have at least five *action* elements. Furthermore, the presented application definition indicates that the workflow is branched into two distinct execution paths, after the third cell has been executed. This entails, that the generated UML model needs to have a *split* element.

wait for click elem1 at app1			
click elem1 click at wid1			
click elem1 click at wid2	copy elem2 at app1 to elem2 at wid1	copy elem3 at wid1 to elem1 at wid3	
copy elem2 at app1 to elem1 at wid2		click elem1 click at wid3	
click elem2 click at wid2		click elem3 click at wid1	

Figure 5.9: Geppeto application example.

Based on the tabular application representation given in Figure 5.9, using the *GeppetoToUml* algorithm (Algorithm 6.2), a UML activity diagram is constructed as shown in Figure 5.10.

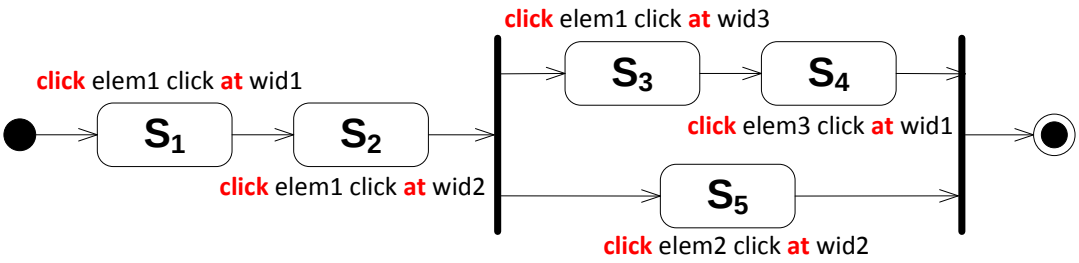


Figure 5.10: UML model for Geppeto application example in Fig. 5.9.

The presented UML activity diagram indicates that upon workflow initialization, first widgets *wid1* and *wid2* are invoked sequentially through *click* operations on their *elem1* controls, as denoted by *action* elements S_1 and S_2. After the action S_2 is completed, the workflow splits into two parallel paths, as indicated by the *split* element. In the first path *wid3* (through *elem1* con-

trol) and *wid1* (through *elem2* control) are invoked sequentially, as denoted by *action* elements S_3 and S_4. In the second execution path, *wid2* is invoked through *elem2* control, as defined by *action* element S_5. Since there are no further operations defined in the tabular representation, the workflow execution is concluded at that point.

Finally, the generated UML activity diagram is translated into a reliability model, using Algorithm 6.1, as presented in Figure 5.11. It is visible that the *action* elements map to the model's input variables and the *split* element maps to the parallel model node, while sequence nodes are used to model sequential widget execution.

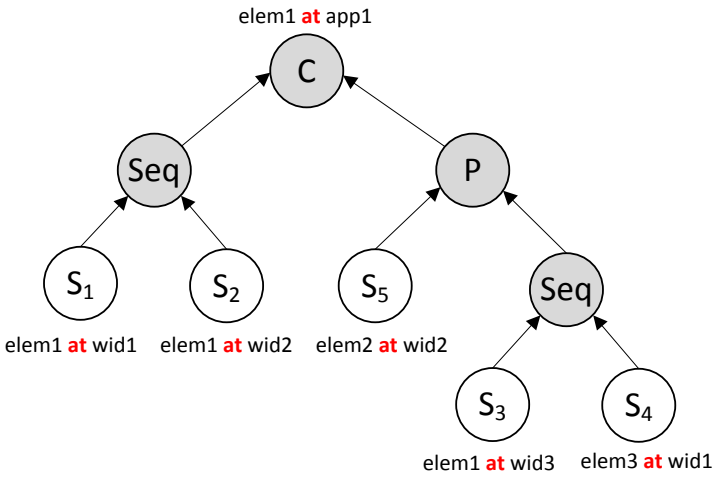


Figure 5.11: Reliability model for Geppeto application example in Fig. 5.10.

Chapter 6

Weak Point Recommendation Method

The goal of the *weak point recommendation* process is to order the composite application's building components by their influence on its overall reliability. More specifically, the probabilistic graphical model presented in Chapter 5 is leveraged to calculate a list of weak points, i.e. a list of atomic building components ordered by the priority in which they are to be replaced. In the following subsections, a suite of weak point recommendation algorithms is presented. The suite consists of three algorithms: *WP-Influence*, and heuristic algorithms *WP-WeakestPath* and *WP-WeightedPath*. While the *WP-Influence* algorithm accurately calculates the influence a particular atomic building component has on the overall composite application reliability, it is more resource intensive than the less accurate heuristic algorithms *WP-WeakestPath* and *WP-WeightedPath*. Distinction also exist between the heuristic algorithms as the *WP-WeightedPath* is designed to be on average more accurate than the *WP-WeakestPath* algorithm at the cost of computational performance. As it has the greatest accuracy in recommending application weak points, the *WP-Influence* algorithm will always reach the desired reliability improvement threshold in lower or equal number of steps then the heuristic algorithms. However, the heuristic algorithms prove to be more applicable in cases when performing a greater number of computationally efficient but less accurate improvement steps achieves the targeted reliability improvement while overall spending less system resources. In addition, the accuracy of the weak point recommendation is impacted by the existing improvement solutions, e.g. replacement components. Specifically if data on improvement solution's reliability is available, it can be used to make the recommendation algorithms more accurate. To that end, we introduce *WP-Influence-R* and *WP-WeightedPath-R* algorithms, extensions to the *WP-Influence* and *WP-WeightedPath* algorithms, that take into account the reliability of existing improvement solutions. More on the

subject of accuracy and computational performance of the presented algorithm suite is discussed in Chapter 8 that gives an extensive evaluation of the algorithm's properties.

6.1 Reliability Sensitivity Analysis

Since in the presented reliability model (Section 5) the influence of each atomic component is bounded to a single random variable, a simple algorithm defined by pseudocode Algorithm 6.1, named *WP-Influence*, can be used to calculate an ordered list of weak points. At each step of the algorithm the reliability of an atomic component being tested, i.e. input random variable, is set to 1 and the overall reliability of the composite application is recalculated. Thus, the reliability of a composite application is estimated for the case that a particular atomic building component is perfectly reliable. Specifically, the algorithm is defined by the following steps performed for each atomic building component, i.e. input random variable in the reliability model. First, the reliability of the evaluated atomic component is temporarily stored (line 4). The reliability of the atomic component is then set to 1 simulating the case that the given atomic component is perfectly reliable, i.e. the atomic component does not reduce the overall reliability of the composite application (line 5). The overall reliability of the composite application is then recalculated (line 6). Finally, the newly calculated reliability is stored into the sorted list of weak points (line 8) and the reliability of the evaluated atomic component is restored to its initial value (line 7).

The atomic components whose increase in reliability causes a greater increase in the overall composite application's reliability are placed closer to the beginning of the list of weak points as more suitable improvement candidates. Although this approach can be leveraged to recommend highly accurate weak point lists, its performance strongly depends on the number of atomic components in the reliability model. This is due to the fact that the reliability of the whole composite application is recalculated for each input random variable. Specifically, the recursive routine to recalculate the composite application's reliability needs to be called kn times, where n is the number of atomic components and $k \leq n$ the number of performed component improvements. In the worse case, when all the components need to be improved ($k = n$), the performance of the algorithm depends on the square of the number of atomic components (n^2). More on the analysis of computational complexity of the algorithm can be found in Section 6.4.

```
1:  function WP-INFLUENCE(model)                          ▷ Returns a list of weak points
2:    wpList ← {}                                         ▷ List of weak points sorted by reliability growth
3:    for all node in inNodes do                          ▷ inNodes - list of input nodes
4:      relTmp ← node.Rel
5:      node.Rel ← 1
6:      finRel ← model.calcRel()                          ▷ Returns composition reliability
7:      node.Rel ← relTmp
8:      wpList.insert({finRel, node})                     ▷ Inserts (finRel,node) pair into the list sorted by
    finRel
9:    end for
10:   model.calcRel()                                     ▷ Resets compostion reliability
11:   return wpList
12: end function
```

Algorithm 6.1: *WP-Influence*.

WP-Influence-R

As stated previously, the presented algorithm is not guaranteed to produce an optimal list of weak points at all times. This is due to the fact that the actual reliability growth is impacted by the increase in building component's reliability achieved as the result of implementing the available improvement solution. For example, a better improvement solution, i.e. one that yields higher overall reliability, can be available for an atomic service that is not recommended as the most significant weak point. Therefore, if known, reliability of the available improvement solutions should be taken into account to make the algorithm more accurate. To that end, the *WP-Influence-R* algorithm is defined as a simple extension to the *WP-Influence* algorithm. Specifically, when performing the sensitivity test, rather then by 1, reliability of the tested service is replaced with the actual reliability of the improvement solution (line 5). By taking into account the presented modification, the *WP-Influence-R* algorithm is guaranteed to produce optimal weak point lists.

6.2 Heuristic Algorithms

The motivation behind heuristic algorithms for weak point detection of composite applications is to reduce the computational complexity at the cost of solution accuracy. This implies that heuristic algorithms require on average more improvement steps to reach a certain reliability threshold than the *WP-Influence* algorithm. However, depending on the working conditions, it may be less resource intensive to perform reliability improvement by committing a greater

number of improvement steps.

The premise used to construct the heuristic algorithms is that the graphical structure of the presented reliability model can be leveraged to make the weak point recommendation process less dependent on the number of atomic components (model input random variables) at the expense of accuracy. This section presents two heuristic algorithms for weak point detection that rely solely on the graphical structure of the reliability model. The first algorithm, presented in Section 6.2.1 (*WP-WeakestPath*), is a simplistic approach that detects weak points by traversing the network in a non-causal direction following the path of random variables with lowest reliability. The second algorithm defined in Section 6.2.2 (*WP-WeightedPath*) is an improvement of the *WP-WeakestPath* path algorithm which takes into account multiple influence paths leading to the output random variable. Thus, the *WP-WeightedPath* algorithm is on average more accurate than the *WP-WeakestPath* algorithm, but also more computationally demanding. More on the analysis of computational complexity of the presented heuristic algorithms can be found in Section 6.4. Finally, an extension to the *WP-WeightedPath* algorithm that takes into account the reliability of available improvement solutions is presented.

6.2.1 Weakest Path

WP-WeakestPath is a heuristic algorithm designed to detect weak points by traversing the reliability model (network) in a non-causal direction, starting at the leaf node, i.e. the composition node (output variable). The algorithm is defined recursively and its definition is given by pseudocode Algorithm 6.2. At each recursive call the algorithm first sorts the parent nodes of the currently visited node by their reliability (line 8). A simple invariant holds for each call: the next node to be visited is the node in the parent's list that has not been previously visited and has the lowest reliability (lines $7-8$). The recursive calls continue until the termination condition has been met, i.e. an atomic component node (input random variable) has been reached (line 3). The atomic component node is then added into the weak point list in the order it was visited (line 4). In essence, the algorithm identifies the most suitable improvement candidate by advancing from the composition node towards input random variables while following the path of lowest reliabilities.

This approach is more lightweight than the previously described *WP-Influence* since it is only necessary to traverse the graphical structure of the model once. In fact, to detect a single improvement candidate, it is necessary to perform a single descend through the reliability

```
1:  function WP-WeakestPath(node)                    ▷ Returns a list of weak points
2:    wpList ← {}                                    ▷ List of weak points
3:    if node in inNodes then                        ▷ inNodes - list of input nodes
4:      wpList.append(node)
5:      return wpList
6:    else
7:      nextNodes ← node.getParentNodes()
8:      nextNodes.sortByReliability()
9:      for all cNode in nextNodes do
10:       if cNode not visited then
11:         cNode.markAsVisited()
12:         wpList.extend(WP-WeakestPath(cNode) )
13:       end if
14:     end for
15:   end if
16:   return wpList
17: end function
```

Algorithm 6.2: *WP-WeakestPath*.

model. However, the proposed algorithm introduces simplifications that make it less accurate than weak point recommendation by observing the influence of each individual atomic component. For instance, the *WP-WeakestPath* algorithm does not take into account that a single input random variable can impact the reliability of the composition through multiple influence paths. An example of such an influence can be seen in the composite application example shown in Figure 5.7. Specifically, the atomic component S_1 influences the reliability of the composite application C directly and over nodes B and Seq_1. By ignoring the existence of multiple influence paths, the *WP-WeakestPath* algorithm recommends weak point solutions with a reduced accuracy, i.e. the atomic building components in the recommended weak point list are not necessarily ordered by their actual influence.

6.2.2 Weighted Path

To reduce the impact of multiple influence paths present in the *WP-WeakestPath* algorithm, a heuristic algorithm *WP-WeightedPath* is introduced. The basic premise behind algorithm's design is that by aggregating all the influence paths for a given input random variable, the random variable's impact on the composite application's reliability is better observed then in the case of *WP-WeakestPath* algorithm. Since determining the exact impact of a particular influence path would in effect make the algorithm as computationally complex as the *WP-Influence* algorithm, a simple heuristic is applied to calculate influence path weights. The weight of each influence

path leading from the composition node to an input random variable is calculated as follows:

$$w_{i,j} = \prod_{k=1}^{n} (1 - r_k)/r_k \tag{6.1}$$

where $w_{i,j}$ is the weight of the j^{th} path leading to input random variable i and r_k the reliability of the k^{th} node on the path to variable i starting at the output random variable. The path from the composition node to the random variable i contains n nodes. In effect, at each algorithm step the influence a parent node (r_{pr}) has on the reliability of its child node (r_{ch}) is roughly estimated using this expression: $r_{ch}/r_{pt} - r_{ch}$.

The overall influence of an input random variable i is calculated by summing all the weights for the paths leading to that variable:

$$inf_i = \sum_j w_{i,j} \tag{6.2}$$

where $w_{i,j}$ is the influence path weight defined by expression (6.1). Atomic components modeled by random variables with a higher cumulative weight factor inf_i are considered to be better improvement candidates.

Using the described heuristic, the *WP-WeightedPath* algorithm is defined recursively by pseudocode Algorithm 6.3. Weights for all the influence paths in the reliability model are firs calculated recursively (line 6). The algorithm advances from the output node (composition node) towards the input nodes of the reliability network in a similar manner as in *WP-WeakestPath* algorithm (line 21). Unlike in the case of *WP-WeakestPath* algorithm, the order in which the nodes are visited is not relevant. Partial weight value of a particular influence path is calculated for each node along the path by applying the equation (6.1) (line 22). As in the *WP-WeakestPath* algorithm, the termination condition for the recursion is when an atomic component node (input random variable) has been reached (line 16). The weight value is then stored into a map, using atomic component node as the key (line 17). Next, all the weights for each particular random variable are summed (lines $7 - 9$) to get the inf_i value. Finally, a list of atomic building components ordered by descending factor inf_i is generated (line 11). The generated list is in effect the recommended list of weak points for the given composite application.

The presented approach is on average more accurate and computationally demanding than *WP-WeakestPath* algorithm, but still less accurate than the *WP-Influence* algorithm. This is due

```
1: function WP-WeightedPath(node)                    ▷ Returns a list of weak points
2:   wpHmap ← {}          ▷ Hash map of weak points: Key - component, Value - weight
3:   for all entry in inNodes do                      ▷ inNodes - list of input nodes
4:     wpHmap.add(entry, 0.0)
5:   end for
6:   wPaths ← CalculatePathWeights(node, 1.0)
7:   for all entry in wPaths do
8:     weight ← wpHmap.get(entry.node) + entry.weight
9:     wpHmap.update(entry.node, weight)
10:  end for
11:  wpList = wpHmap.generateDescendingSortedList()
12:  return wpList
13: end function
14: function CalculatePathWeights(node, weight)
15:  pathWeights ← {}                                 ▷ List of path weights
16:  if node in inNodes then
17:    pathWeights.append({node, weight})
18:    return pathWeights
19:  else
20:    nextNodes ← node.getParentNodes()
21:    pWeight ← weight * ((1 − node.reliability)/node.reliability)
22:    for all cNode in nextNodes do
23:      pathWeights.extend(CalculatePathWeights(cNode,
     pWeight) )
24:    end for
25:  end if
26:  return pathWeights
27: end function
```

Algorithm 6.3: *WP-WeightedPath*.

to the fact that the heuristic presented in (6.1) is only a rough estimation of the component's influence on its child node. It should also be noted that the algorithm has to completely traverse the entire model in order to produce a solution, as opposed to the *WP-WeakestPath* algorithm. Specifically, in case of the *WP-WeakestPath* algorithm, the search can be aborted after a first viable improvement solution is found, while in case of *WP-WeightedPath*, the entire model has to be traversed to calculate cumulative influence factors inf_i.

WP-WeightedPath-R

Similarly like in case of the *WP-Influence* algorithm, if reliability of the available improvement solutions is known, it could be used to make the presented heuristic more accurate. To that end, a simple solution that considers weighting the influence factor inf_i with reliabilities of available improvement solutions is presented. Specifically, the modified cumulative influence

factor is calculated as follows:

$$inf_i' = inf_i \cdot r_i \tag{6.3}$$

where r_i is the reliability of available improvement solution for building component i. Throughout the rest of the thesis, this modification is referred to as *WP-WeightedPath-R* algorithm.

6.3 Reliability Improvement Example

As stated previously, the list of recommended composite application's weak points calculated utilizing the presented probabilistic model and algorithms in previous subsections is used to prioritize component improvements. Hence, the atomic components that are closer to the beginning of the weak point list are considered to be better improvement candidates, as their improvement would yield a greater increase in overall composite application reliability. An example of the reliability improvement process using the *WP-Influence*, *WP-WeakestPath* and *WP-WeightedPath* algorithms for the composite application example in Figure 5.7 is given in Figure 6.1. Initially, all the atomic components $S_1, ..., S_6$ are considered to have an equal reliability of 0.85. It is assumed that each atomic component can be replaced by a single more reliable—functionally equivalent—component with reliability value of 0.99. Since there are 6 possible replacement choices, one for each atomic component, there is a total of 6 improvement steps, as denoted on the x-axis. The overall composite application's reliability is given on the y-axis of the graph. For the presented example it can be observed that the *Wp-Influence* and *WP-WeightedPath* algorithms gave better replacement recommendations than the *WP-WeakestPath* algorithm since they both resulted in a higher overall reliability for replacement lines $2 - 4$. A detailed evaluation of the presented weak point recommendation algorithms can be found in Section 8.

6.4 Computational Complexity Analysis

This section presents the computational complexity analysis for the weak point recommendation algorithms defined in this chapter. The complexity is studied in turn for the *WP-Influence* algorithm, and heuristic algorithms *WP-WeakestPath* and *WP-WeightedPath*. For the purposes

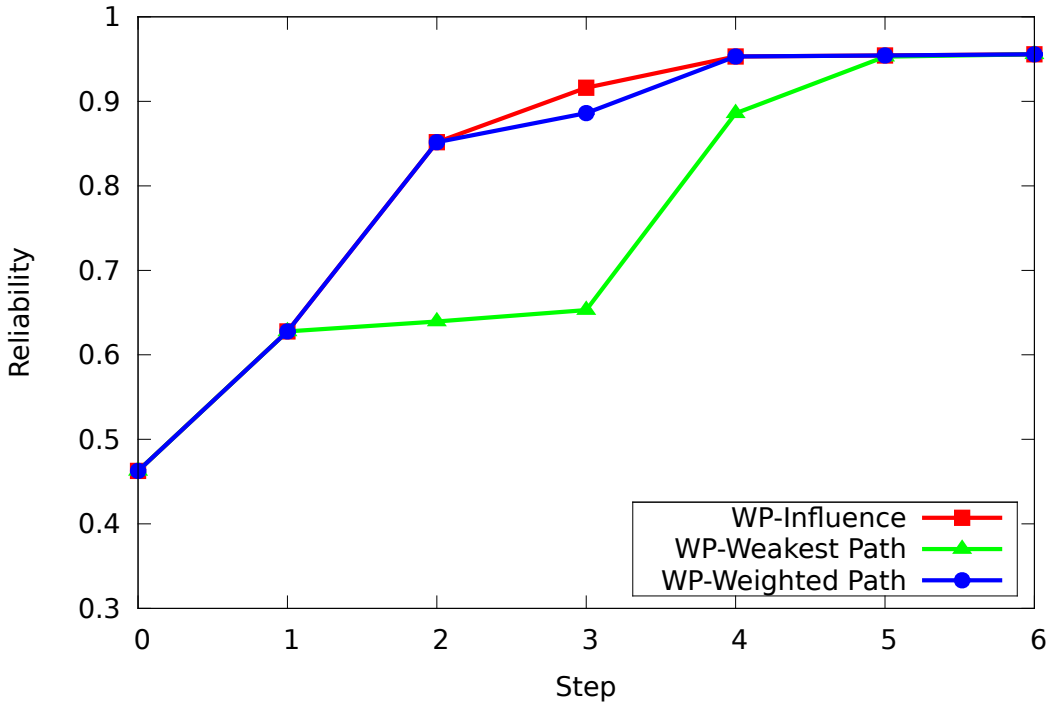


Figure 6.1: Reliability improvement results for the composition example in Fig. 5.7.

of analysis, the *big O* notation, i.e. limiting behavior of the algorithm, is studied. The behavior of algorithm is described by function $T(x)$, denoting the dependence of algorithm's execution time and variable x. Formally, the *big O* notation is defined as follows:

$$T(x) = O(f(x)) \mid x \to \infty \tag{6.4}$$

if and only if a constant M exists for a sufficiently large value of $x = x_0$ such that the following condition holds:

$$|T(x)| \leq M \cdot |f(x)|, \ \forall \ x \mid x > x_0 \tag{6.5}$$

WP-Influence

In case of the *WP-Influence* algorithm, the impact of each atomic component on the overall reliability needs to be estimated separately. Thus, it is clear that the computational complexity of the algorithm depends on the number of atomic components, which will be denoted as variable n. In addition, it also depends on the complexity of the model, namely the number of variables along the influence path l and number of influence paths per input variable i. Thus, the computational complexity of *WP-Influence* is defined by function $T_{inf}(n, l, i)$.

Since the influence for each input random variable is computed separately (loop defined by lines $3-8$), the complexity is defined by the following function: $T_{inf}(n,l,i) = n \cdot T_l(n,l,i)$, where $T_l(n,l,i)$ is the inner complexity of the loop. Lines 4, 5 and 7 that modify and restore the reliability of the tested input random variable have all complexity $O(1)$. In line 6, the reliability of the composite application is recalculated recursively. Since the it is only necessary to update the reliability values along the influence paths leading form a given input random variable, the complexity depends on the number of random variables along the path and on the number of influence paths. Thus, the complexity for reliability computation in line 6 is defined as: $T_{rCal}(l,i) = l \cdot i$. In order to construct a list of weak points, the resulting reliability gained from the estimation of influence needs to be stored into a sorted list (line 8). Computational complexity of inserting a value into a sorted list $T_{ins}(n)$ can be estimated as $\log(n)$. This is due to the fact that the temporary list does not have to be implemented as a linked list, but rather as a binary tree or heap with insertion complexity of $\log(n)$. Thus, the inner loop complexity can be estimated as $T_l(n,l,i) = T_{rCal}(l,i) + T_{ins}(n) = l \cdot i + \log(n)$.

By plugging in all the values into T_{inf} and adding an additional single reliability computation in line 10, the following complexity is estimated: $T_{inf}(n,l,i) = n \cdot (l \cdot i + \log(n)) + l \cdot i$. Finally, the upper bound on the computational complexity of the *WP-Influence* algorithm is defined as:

$$T_{inf} = O(n \cdot l \cdot i + n \cdot \log(n)) \tag{6.6}$$

WP-WeakestPath

WP-WeakestPath is a recommendation algorithm that traverses the reliability model to generate a list of weak points without the need to recalculate composite application's reliability. Since an additional condition not to visit the already visited nodes is introduced in line 11, the reliability model is effectively transformed into a tree. Thus, its computational complexity can be estimated in a similar manner as for tree traversal and it is equal to $O(E)$, where E is the total number of nodes in the model. More specifically, the computational complexity for *WP-WeakestPath* can be defined as:

$$T_{wkp}(n,k) = O(n+k) \tag{6.7}$$

where n is the number of input random variables and k is the number of hidden model nodes

including the output variable.

In order to study the algorithm's properties in a greater detail, each recursive call can be separated into two segments: the stopping condition (lines $3-5$) and the recursion step (lines $6-12$). The presented complexity analysis is organized the same way. Thus, the complexity of the *WP-WeakestPath* algorithm is defined as $T_{wkp}(n,k,c) = n \cdot T_{st}(n) + k \cdot T_{rs}(k,c)$, where $T_{st}(n)$ and $T_{rs}(k,c)$ are the functions representing the complexity of stopping condition and recursion step respectively. The stopping condition is executed for each of n input random variables. Computational complexity for inserting a new entry into an empty weak point list is $O(1)$. Therefore, the computational complexity of the stopping condition equals to $T_{st}(n) = n$.

On the other hand, the recursion step of the algorithm is executed k times. At each execution, all the parent nodes are sorted by their reliability (line 8). In case there is an upper bound of c parents per child node, the computational complexity of the sorting procedure can be estimated to $c \cdot \log(c)$. It should be noted that $k \geq c$ and usually $k >> c$.

Insertion into the weak point list in line 12 is executed c times per recursive call. In case the list is implemented as a stack or a dynamic array passed into the function as a reference (this is not shown in pseudocode), the merging complexity for lists in line 12 is O(1). This is due to the fact that insertion is then performed only in line 4 and its complexity for a stack and dynamic array is $O(1)$ (amortized constant time in case of the dynamic array). Having in mind the stated conclusions, computational complexity of the recursive step is estimated as: $T_{rs}(k,c) = k \cdot (c \cdot \log(c) + c)$. Thus, the overall algorithm computational complexity is estimated as: $T_{wkp}(n,k,c) = n + k \cdot (c \cdot \log(c) + c)$. Thus, the upper bound on the computational complexity is defined as:

$$T_{wkp}(n,k,c) = O(n + k \cdot c \cdot \log(c)) \tag{6.8}$$

It should be noted that the presented algorithm has an additional favorable property as it can be easily modified to recommend weak points one at the time. This can be achieved by stopping the recursion once the first input random variable has been reached. In that case the computational complexity of the algorithm is reduced to:

$$T_{wkp}(l) = O(l) \tag{6.9}$$

where l is the number of nodes along the influence path, i.e. the depth of the tree the algo-

rithm is descending.

WP-WeightedPath

The *WP-WeightedPath* algorithm achieves on average greater accuracy then the *WP-WeakestPath* algorithm due to the fact that it takes into account the existence of multiple influence paths leading form output to input random variables. Thus, the algorithm can be broken down into following three tasks: calculating path weights (line 6), aggregating path weights (lines $7-10$) and preparing a sorted list of weak points (line 11). The presented complexity analysis is organized the same way and, thus, the computational complexity of *WP-WeightedPath* algorithm is defined by function $T_{whp}(n,k,c,s,l) = T_{whc}(k,c,s,l) + T_{agg}(k,c,s) + T_{prep}(n)$, where n is the number of input random variables, k is the number of hidden model nodes including the output variable, c is the number of parents per model node, s the number of children nodes per model node, and l length of the influence path.

The path weights are computed by a recursive function *calculatePathWeights* (lines $14-27$). The recursion is similar to the one applied in the *WP-WeakestPath* algorithm and consists of a stopping condition (lines $16-18$) and recursion step (lines $20-23$). Thus, the computational complexity of the recursive function is defined as: $T_{whc}(k,c,s,l) = T_{st}(k,c,s) + T_{rs}(k,c,s,l)$. Adding a single element to a list, or rather a stack, in the stopping condition (line 17) is a $O(1)$ operation. However, this operation can be executed more than once for each input random variable as multiple influence path can exist. It should be noted that nodes are not marked as visited as it was the case in the *WP-WeakestPath* algorithm. Thus, the operation in line 17 is executed once for each influence path. The total number of influence paths can be estimated using values k, c and s. Since each node has at most c parents, there are c influence paths branching form the output random variable. Furthermore, each random variable in the hidden layer of the model ($k-1$ variables) can add additional $c-1$ influence paths for each path leading into the variable. Finally, any random variable, except the output variable, can have s children (s influence paths leading into the variable). The total number of influence paths, that also represents computational complexity of the stopping condition, is $T_{st}(k,c,s) = s((c-1)(k-1)+c) = s(c \cdot k + 1)$.

The body of the recursion step contains two operations. The operation in line 22 has complexity $O(1)$, as does the operation is line 23 for identical reasons as in case of *WP-WeakestPath* algorithm. For the reasons of simplicity, computational complexity of the recursion step is

brought into relation with the length of influence path l. It is estimated that the recursion step will be executed for each node along the influence path, except for the output random variables. Given the number of influence paths, computational complexity of the recursion step can be estimated as: $T_{rs}(k, c, s, l) = (l - 1) \cdot s \cdot (c \cdot k + 1)$. Is should be noted that T_{rs} is an overestimation of the complexity as the parameter l will be lower due to branching conditions. Finally, the computational complexity of the recursive function is estimated by function: $T_{whc}(k, c, s, l) = l \cdot s \cdot (c \cdot k + 1)$.

In the task of aggregating influence path weights (line $7 - 10$), a hash map is used to store the aggregated values. The computational complexity of accessing and updating the values in the map is $O(1)$. The procedure is preformed for each of the influence paths, as there are so many weights in the list. Thus, the computational complexity of the entire task is $T_{agg}(k, c, s) = s(c \cdot k + 1)$.

Furthermore, a sorted list of weak points needs to be generated from the hash map of aggregated influence path weights (lines 11). Generating a sorted list of n elements, where n is the number of atomic components has the complexity of $n \cdot \log(n)$. Therefore, the computational complexity of task is $T_{prep}(n) = n \cdot \log(n)$.

Finally, the overall computational complexity of the *WP-WeightedPath* algorithm is represented by function: $T_{whp}(n, k, c, s, l) = (l + 1) \cdot s \cdot (c \cdot k + 1) + n \cdot \log(n)$. Thus, the upper bound on the computational complexity is defined as:

$$T_{whp}(n, k, c, s, l) = O(l \cdot s \cdot c \cdot k + n \cdot \log(n)) \tag{6.10}$$

The computational complexity is further reduced if the algorithm is modified to return one weak point recommendation at a time. Although the influence path weights still need to be calculated and aggregated, the weak point list preparation step can be reduced to locating the most influential component. Since in the worst case this task can be achieved in n steps, the overall computational complexity is defined as:

$$T_{whp}(n, k, c, s, l) = O(l \cdot s \cdot c \cdot k + n) \tag{6.11}$$

Chapter 7

Consumer-Defined Reliability Improvement

In order to make the reliability management method applicable to composite consumer applications, it is necessary to extend the consumer computing environment with appropriate elements. Specifically, there are two major processes defined by the reliability management method: application analysis and application improvement. The application analysis process consists of estimating the overall application reliability and recommending application's weak points. If it is necessary to attain higher reliability, application's weak points are improved, by modifying the workflow through introduction of redundancies or more reliable components. As stated in Section 4 the process of application analysis can be fully performed by a computer system and, thus, the consumer computing environment needs to be extended with a digital assistant. On the other hand, workflow modifications need to be performed by consumers. Consequently, in order to introduce workflow redundancies, consumers need to have a set of programmable elements at their disposal. Therefore, in order to support the proposed reliability management method within the consumer computing environment, it is necessary to define a digital consumer assistant and a set of programmable elements.

The rest of this chapter introduces architectures of digital consumer assistant and programmable components. The architecture of consumer assistant addresses both the consumer actions implied by the reliability management method, as well as requirements for efficient execution within the distributed environment. Furthermore, architecture definitions for both the forward and backward recovery based programmable components are presented. Finally, the chapter introduces consumer interface examples that stem from the described architectures.

7.1 Consumer Assistant for Development of Reliable Consumer Applications

Consumer application reliability assistant is an element of the consumer computing environment that aids consumers in development of reliable composite applications. Its design stems from the processes defined by the reliability management method presented in Section 4.2. Specifically, consumer assistant implements the entire iterative method, except for the *application improver* segment of the *reliability improvement* process. Based on the consumer interaction points defined within the *analysis requester*, *improvement requester* and *reliability improvement* processes, the following basic consumer operations need to be supported by the consumer assistant:

- **Attach / detach application** - Consumer needs to be provided with ability to associate an application with a particular consumer assistant. Operation of attaching an application is conducted as part of the *analysis requester* process, while applications are detached in the *improvement requester* process when no further improvements are needed.
- **Estimate reliability** - As part of the *analysis requester* process, consumer requests reliability estimation for an associated composite application.
- **Get weak point list** - If further application improvements are needed, consumer requests application analysis as part of the *improvement requester* process. As result of this operation, an ordered list of application's weak points is generated.
- **Get replacement components** - Prior to introducing workflow improvements, as part of the *reliability improvement* process, consumer requests a set of similar (functionally equivalent) components for a selected subset of weak points.

Based on the identified operations that stem from the reliability management method, an architecture of the consumer application reliability assistant is defined as presented in the following section.

7.1.1 Consumer Assistant Architecture

Architecture of the proposed consumer application reliability assistant is presented in Figure 7.1. The architecture consists out of four main modules: *application reliability manager*, *reli-*

ability model generator, *reliability estimator* and *weak point recommender*. Out of the defined modules, the *application reliability manager* module is specific to each individual consumer assistant because it contains data characteristic to the attached applications. On the other hand, *reliability model generator*, *reliability estimator* and *weak point recommender* modules are designed to be generic and can be shared between different instances of consumer assistants. Functionality and interoperability of each module is described through the supported consumer operations.

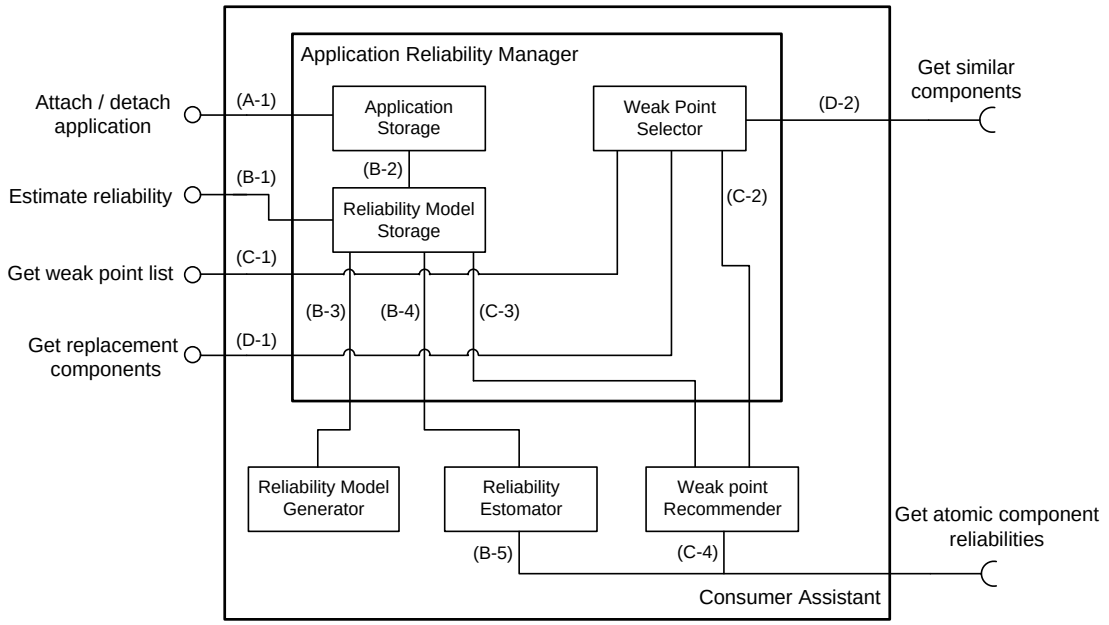


Figure 7.1: Architecture of consumer application reliability assistant.

Application reliability manager is a complex module comprised of *application storage*, *reliability model storage* and *weak point selector* submodules. It serves as an interface towards the consumer by exposing a set of basic consumer operations. Each of the operations is briefly described below.

The *attach / detach application* operation (A-1) is used to add or remove application definitions from the *application storage* module. Application definitions are workflow definitions stored in some representation, e.g. Geppeto spreadsheet language.

Once an application has been attached, it can be analyzed using the *estimate reliability* operation (B-1). The *estimate reliability* operation accesses the *reliability model storage* that contains the data necessary to perform reliability estimations, i.e. composite applications' reliability models. In case a reliability model does not exist for the analyzed application, application

definition is retrieved from the *application storage* module (B-2) and passed to the *reliability model generator* module. The *reliability model generator* module produces a reliability model for the given application definition and stores it into the *reliability model storage* (B-3). Once a reliability model is available, the *reliability estimator* module is invoked (B-4). If the passed reliability model does not contain atomic components' reliability values or the values are considered to be outdated, the *reliability estimator* module retrieves new reliability values by invoking the *get atomic component reliabilities* operation (B-5). This operation produces atomic component reliabilities using other consumer computing environment elements, as discussed in Section 4.2. Once all the necessary data is available, reliability estimation is performed and the estimated value stored into *reliability model storage*.

Afrer a reliability estimate is available in the *reliability model storage*, this value can be presented to the consumer. Further actions can then be taken as part of the *improvement requester* process. To that end, the *get weak point list* operation invokes the *weak point selector* module that is responsible for managing application's weak point recommendations (C-1). First, the *weak point selector* module invokes the *weak point recommender* module responsible for generating ordered lists of weak points. The *weak point recommender* module then retrieves the reliability model and the available atomic components' reliability values from the *reliability model storage* module (C-3). If valid atomic components' reliabilities are available, the recommendation process is performed and results presented to the consumer. Otherwise, the reliability values need to be renewed using the *get atomic component reliabilities* operation (C-4).

Finally, as part of the *reliability improvement* step, consumer selects the most pronounced weak points and gets a set of replacement components through the *get replacement components* operation (D-1). This operation invokes the *weak point selector* module that contains data on weak point list recommendations. Form the presented list of weak points, consumer selects a subset of components that are considered to be best candidates for improvement. The *weak point selector* module then retrieves components similar to selected weak points through the *get similar components* operation (D-2). This operation invokes external consumer computing elements, as discuses in Section 4.2, and therefore it is not discussed further.

Efficient execution

Achieving efficient execution of the presented consumer assistant in distributed environments relies on two properties. The first property regards possibilities of computation offload-

ing into the cloud infrastructure. This principle is particularly useful when it comes to mobile devices, as it enables execution of computationally demanding operations while saving vital system resources, including battery power [258]. The presented consumer assistant architecture enables computational offloading of most demanding operations, performed by *reliability model generator*, *reliability estimator* and *weak point recommender* modules. All other operations performed by the *application reliability manager* are lightweight in comparison, making it possible to deploy that module onto devices with less computational resources. Therefore, the presented consumer assistant architecture enables usage of cloud infrastructure to achieve efficient execution within a distributed environment.

Apart from the possibilities of computational offloading, efficient execution can be achieved by optimizing the underlaying modules. This thesis specifically focuses on the *weak point recommender* module, as its operations are most computationally demanding. Specifically, the proposed weak point recommendation method leverages algorithms with varying performance and accuracy that are used at different environment conditions. In cases when reliability management method steps induce large computational overheads, e.g. estimated resources to detect similar components are high, it is more computationally efficient to use more accurate recommendation algorithms and, thus, perform less improvement steps. On the other hand, if stepwise resource overheads are low, performance of the *weak point recommender* module can be improved by using less accurate, but also less computationally demanding weak point recommendation algorithms. Details on the weak point recommendation algorithms introduced in this thesis are presented in Chapter 6.

One possible way to determine what recommendation algorithm is best to use at a particular application improvement step is to analyze dependence of spent resources and recommendation algorithm accuracy on a sufficiently large reliability model data set. To that end, the architecture in Figure 7.1 would have to be extended with a method that allows the *weak point recommender* module to access the global repository of reliability modules. In addition, *reliability model storage* would have to be extended with a method invocation that enables storing reliability models into the global model repository. More details on the perspective of capturing dependence of computational overhead and recommendation algorithm accuracy can be found in Chapter 8.

7.1.2 Consumer Assistant Interface

Example of an application reliability consumer assistant implemented as a web widget is shown in Figure 7.2. The presented interface is implemented by the *application reliability manager* module as it supports all the defined consumer operations. The figure illustrates how do particular consumer operations map to the graphical interface. The operations are performed by consumers as defined in the following use case example.

Consumer starts using the widget by adding composite applications through the *attach /detach* operations implemented as part of the applications panel. Specifically, applications can be added using the *attach* control (1) and detached using the remove symbol next to the application name (2). Once an attached application is selected (3), an additional panel is displayed, implementing the *estimate reliability*, *get weak point list*, and *get replacement components* operations. To estimate the selected application's reliability, consumer uses the *estimate* control (4). Based on the estimation result, consumer can proceed with application analysis by using the *recommend* control (5), or abort the reliability management method. The recommended weak points are presented as a list of components ordered by their influence on the overall reliability. Once consumer selects a weak point (6), its possible replacement candidates can be retrieved by using the *get replacement* control (7). The recommended replacements can than be accessed by the consumer and incorporated into the improved application. Upon modifying the application's workflow, consumer can reestimate the reliability (4) and, if necessary, repeat the process until an adequate reliability level has been reached.

7.2 Programmable Components for Reliability Improvement

In order to enable introduction of fault tolerant constructs into the application's workflow, as part of the *reliability improvement* process, consumer computing environment is extended with a set of programmable elements. Programmable elements are used to implement both backward and forward recovery methods, specifically through implementation of *checkpoint and restart*, *recovery blocks* and *n-version programming* principles. Architectures of programmable elements, along with their consumer interface implementations, are described in the following sections.

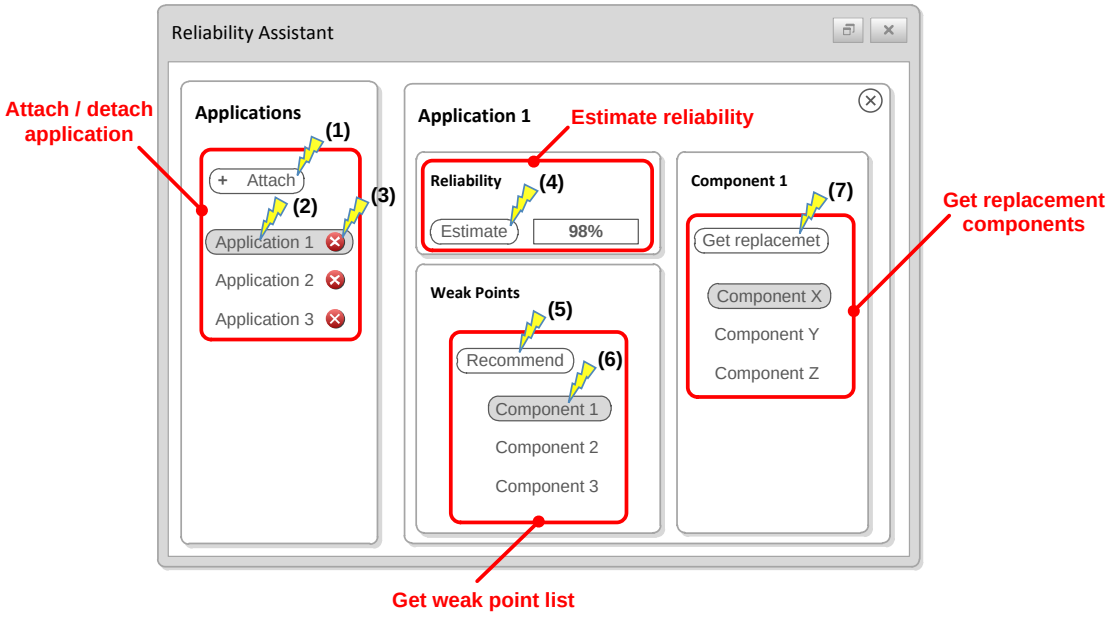


Figure 7.2: Consumer assistant interface.

7.2.1 Backward Recovery Programmable Component

In order to enable consumer programmable elements based on the backward recovery model, as defined in Section 3.5.3, consumer has to be able to perform the following operations:

- **Add / remove components** - Consumer associates a set of semantically equivalent (redundant) components with the programmable component. Consumer can either add or remove components from the fault tolerant construct.

- **Define workflow actions** - Redundant components need to be incorporated into the application workflow. Required workflow actions for various components may differ, e.g. due to differences in user interfaces and usage. For that reason, consumer needs to define workflow actions on a component basis. Moreover, after a programmable component has completed its operation, it needs to transfer the control flow to other parts of the consumer application. This is a specialized workflow action defined by the consumer.

- **Define unified inputs** - In some cases it might be necessary for all components to be provided with exactly the same input. Therefore, rather than retrieving data from the data sources at each component invocation, inputs can be stored at the beginning of redundant construct execution. This is done by performing specialized consumer-defined data flow actions that prepare parameters prior to component execution.

- **Define state restoration actions** - In case an error is detected, consumer-defined actions might be needed to restore the system into an error-free state before other execution attempts can be committed. These restoration actions can differ depending on the component, thus, need to be defined on a component basis.

- **Define acceptance tests** - In case of backward recovery method, it is necessary to explicitly define error detection mechanisms, e.g. in a form of an acceptance test. As in case of workflow and state restoration actions, consumer defines testing actions that can differ for each component.

- **Run** - Consumer uses this control flow operation to runs the redundant workflow.

In accordance with the stated consumer operations, architecture of backward recovery programmable components is defined, as presented in Figure 7.3. The architecture consists of *component storage*, *workflow storage*, *unified input storage*, *state restoration storage*, *acceptance test storage*, *workflow execution engine*, and *test execution engine* modules. Module functionality and interoperability at architectural level is described by observing the individual consumer operations.

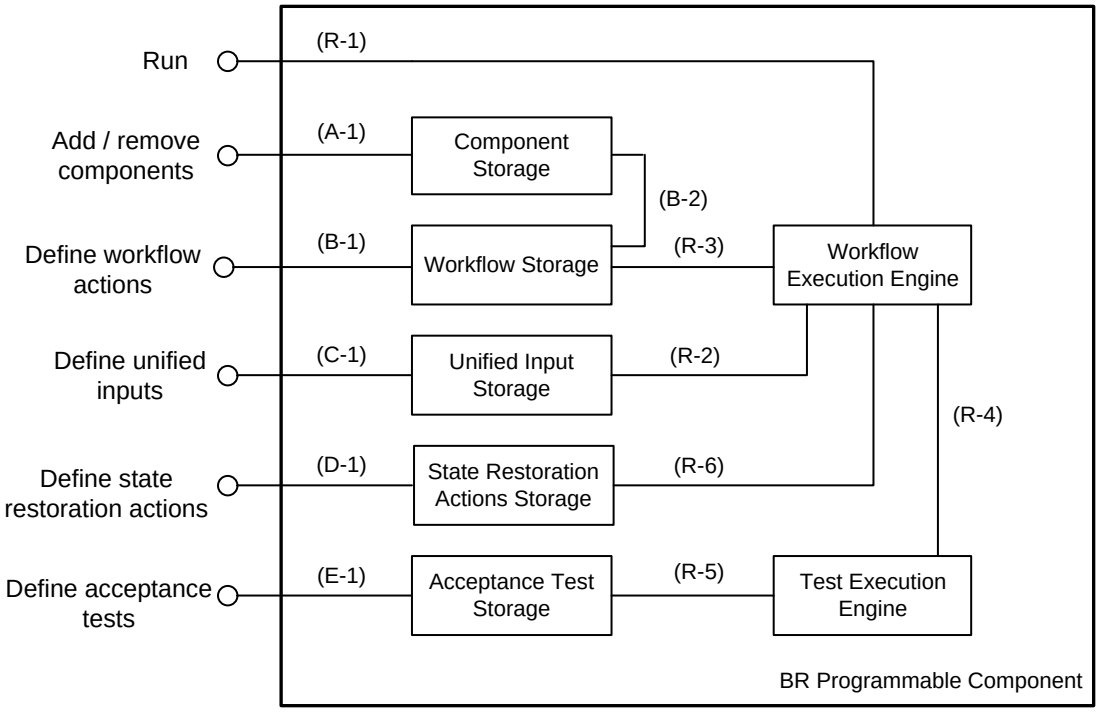


Figure 7.3: Architecture of backward recovery programmable component.

By performing the *add / remove components* operation (A-1), consumers store or remove component data from the *component storage* module. The data contains all the information on how to access the associated component, e.g. its web location or reference to executable code.

Through the *define workflow actions* operation (B-1), consumer states the operations required to functionally integrate a component into the application's workflow. For example, in case of the Geppeto tool, such operations are defined by storing the GUI-level operations performed on the component's interface, e.g. copy/paste, click, double click operations. The operations are stored as consumer programs into the *workflow storage* module. The module associates all the actions with corresponding component data stored in *component storage* module (B-2) to provide the complete execution context.

By utilizing *define unified inputs* operation consumer defines how are the redundant construct's inputs retrieved and prepared. These actions are performed prior to workflow execution to ensure all the components have the same input. The actions are defined similarly like workflow actions and are stored into the *unified input storage* module (C-1).

The *define state restoration actions* enables consumer to define a set of actions needed to restore the application into an error-free state. These actions are similar to workflow actions and are stored into the *state restoration actions storage* module (D-1).

To define the acceptance tests, consumer uses the *define acceptance tests* operation. This operation is used to store actions that define the test cases, i.e. against what values are particular component outputs tested. These actions are stored into the *acceptance test storage* module (E-1).

Finally, when all the actions are defined, the *run* operation can be used by consumer or computer system to execute the redundant workflow (R-1). Upon receiving the execution request, *workflow execution engine* first runs the operations needed to prepare the component inputs (R-2). After preparing the inputs, redundant components are run in sequence or in parallel using the stored workflow operations (R-3). The execution results are then passed to *test execution engine* (R-4) that performs the tests based on the defined acceptance test operations (R-5). In case of an error, the *workflow execution engine* is triggered to perform the state restoration actions (R-6) and then execute other redundant components, if they are available.

Checkpoint and Restart

The *retry* programmable component implements static checkpoint and restart principle, i.e. checkpoints are not generated dynamically, but rather defined explicitly by the consumer. It is based on the architecture of backward recovery programmable components with two distinctions. First, since this is a single version fault tolerance method, it is not necessary to support the *add / remove components* operations and subsequently implement the *component storage* module. The one component associated with the *retry* component is attached through workflow operations. The second distinction regards the restart operation as consumers need to be able to provide an acceptable number of retries before a component is considered to fail. The number of retries is set using the additional *define number of retries* operation.

The interface of the *retry* programmable component implemented as a web widget is presented in Figure 7.4. The image shows how the particular consumer operations relate to the user interface constructs. The consumer operations are performed utilizing the presented interface, as described in the following use case example.

In case it is necessary for each component execution to have exactly the same input, consumer can utilize the *define unified inputs* method. Consumer adds (1) or removes (2) input fields using controls in the *inputs* panel. Then, consumer defines data flow operations to link the input fields with the data sources (3). These data flow operations are performed prior to component invocation. Apart from the unified inputs, consumer needs to define acceptance tests to assess if the component operates properly. Tests cases can be added (4) and removed (5) using the controls in the *test* panel. Each test case contains two fields that are connected by data flow operations with component outputs (6). In order to compare values of two fields, an operator is selected (7). Defined test cases are performed after the component invocation has been completed. To define how the component is invoked, i.e. how it is incorporated into the application's workflow, *define execution* control is used (8). Apart from the specific workflow actions, additional control flow transfer functions are defined. The *on success* control (9) is used to define workflow actions that transfer control flow to other parts of application if no errors have been detected by the test cases. The *on failure* (10) control is used to define state restoration actions that need to be performed between the successive component invocations. The maximum number of component invocation retries is defined using the *retries* control (11). Finally, the defined fault tolerant construct can be executed using the *run* control (12).

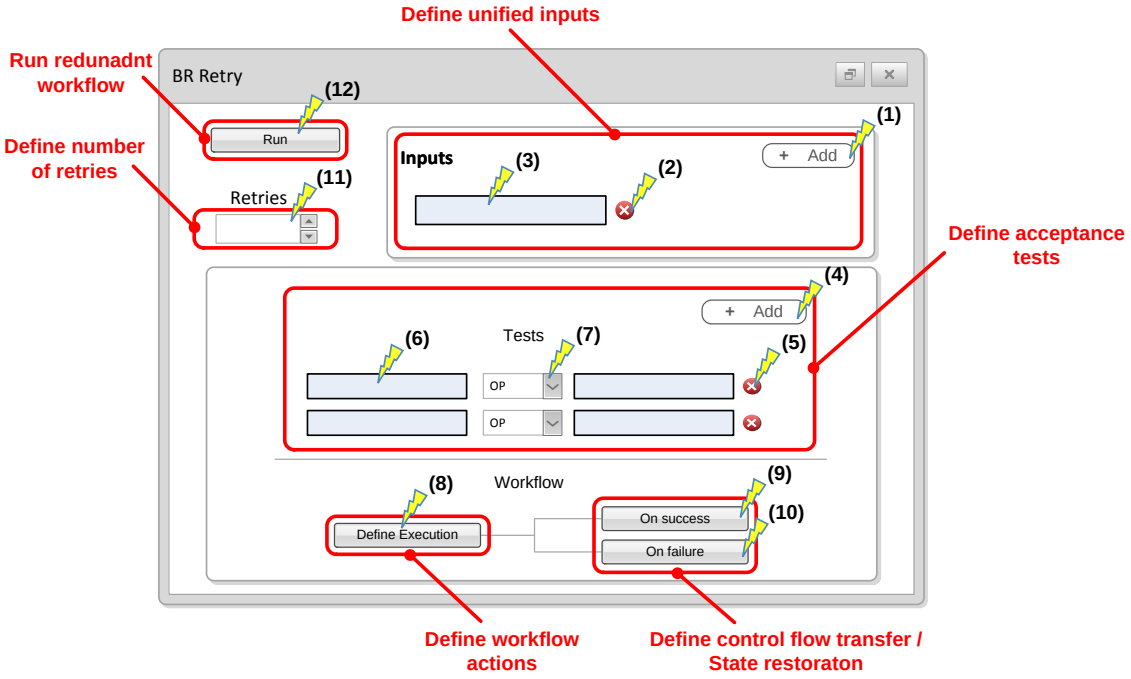


Figure 7.4: Retry programmable component interface.

Recovery Blocks

The *recovery blocks* programmable component is based on the recovery blocks fault tolerance method. Since this is a multi-version fault tolerance method, multiple redundant, semantically equivalent, components need to be included into the application workflow, as opposed to the previously described *retry* programmable component. Specifically, the *recovery blocks* programmable component enables sequential execution of redundant components. After a component is invoked, its outputs are tested for errors. If errors are detected, the next component in line is executed, otherwise the execution chain is discontinued.

The interface of the *recovery blocks* programmable component stems from the architecture of backward recovery programmable components and is implemented as a web widget, presented in Figure 7.5. The image shows how the particular consumer operations relate to the graphical interface constructs. The consumer operations are performed using the presented interface as described in the following use case example.

The main difference between *retry* and *recovery blocks* programmable elements is that the latter supports software redundancy. Thus, consumer needs to be provided with the ability to associate semantically equivalent components with the programmable widget. This is performed through the *components panel* where consumer can use controls to add (1) or remove (2) com-

ponents form the redundant component list. By selecting a component in the list (3), consumer displays the *component details* panel. Through this panel consumer can define separate tests, workflow, and state restoration actions for each component. Tests cases can be added (4) and removed (5) using the controls in the *test* panel. Each test case contains two fields that are connected by data flow operations with component outputs (6). In order to compare values of two test case fields, consumer selects an appropriate operator (7). Tests for a particular component are performed after its invocation has been completed and the outputs are available. To define how each particular component is incorporated into the application workflow, consumer uses the *define execution* control (8). An additional control *on success* (9) is used when acceptance tests have have detected no errors to transfer the control flow to other parts of the consumer application. The *on failure* control (10) is used to set the state restoration actions that are to be performed after a particular component has displayed erroneous behavior. State restoration actions of the last component in the redundant component list are used to transfer control to other failure mitigation mechanisms or to notify the consumer that a failure has occurred.

Apart from the presented functionalities, an additional panel for defining component inputs is available. If components need to be provided with the identical inputs upon their execution, consumer secures this property by performing the *define unified inputs* operation. Consumer adds (11) or removes (12) input fields using controls in the *inputs* panel. Then, consumer defines data flow operations to link the input fields with the data sources (13). The stored data flow operations are performed once, prior to invocation of the first component in the list.

Finally, the fault tolerant construct that incorporates all the associated semantically equivalent components can be executed using the *run* control (14).

7.2.2 Forward Recovery Programmable Component

To support forward recovery programmable components, as defined in Section 3.5.3, consumer has to be able to perform *add / remove component*, *define workflow actions*, *define unified inputs*, and *run* operations, as defined in Section 7.2.1. Since correct output are determined by comparing outputs of multiple components, no explicit error detection or state restoration actions are needed, like it is the case with backward recovery programmable components. However, an additional consumer operation is required, defined as follows:

- **Define outputs** - Error detection and mitigation mechanisms for forward recovery programmable components depend on selecting the correct output among all the redundant

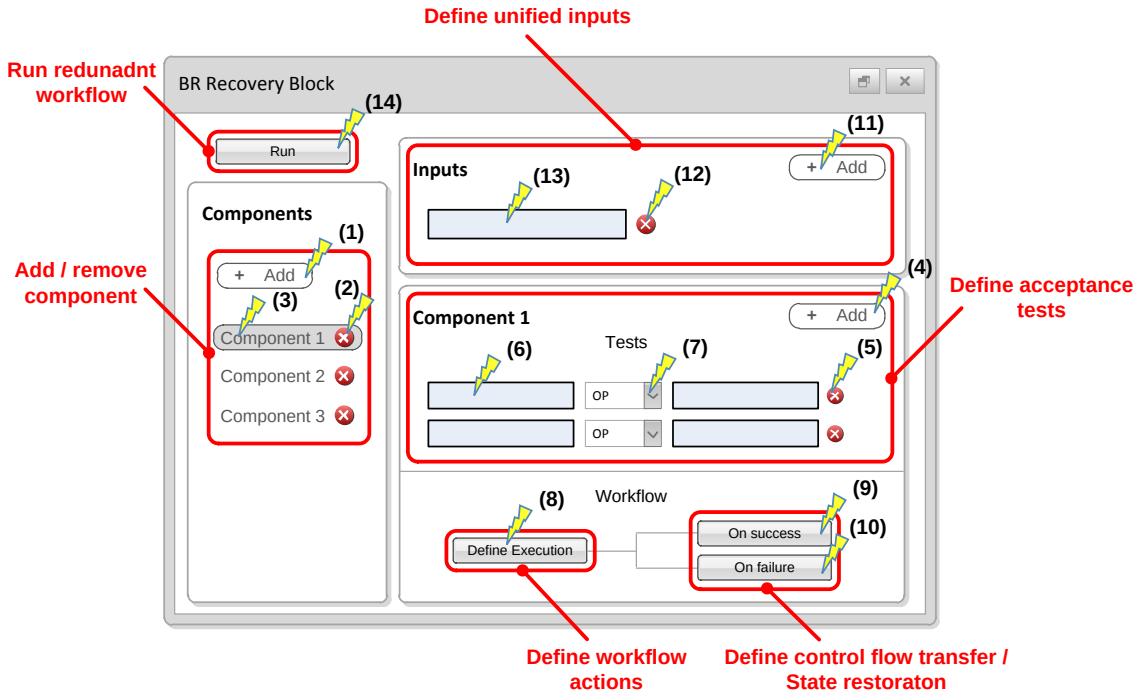


Figure 7.5: Recovery blocks programmable component interface.

components' outputs. To enable output selection algorithms, consumer defines which particular output fields in different redundant components match semantically. In addition, if outputs are not syntactically equivalent, consumer needs to define appropriate transformations that are to be executed before the output selection algorithms.

In accordance with the stated consumer operations, architecture of forward recovery programmable components is defined, as presented in Figure 7.6. The architecture consists of *component storage*, *workflow storage*, *unified input storage*, *output storage*, *workflow execution engine*, and *output selector* modules. Module functionality and interoperability at architectural level is described by observing the individual consumer operations.

Consumer operations (A-1) - (C-1) are performed in the same way as defined by the architecture of backward programmable components, presented in Section 7.2.1. The only difference is the *define output* operation (D-1). By executing this operation, consumer stores the matching rules between different component outputs into the *output definition storage* module.

As defined previously, the *run* operation is used by a consumer or a computer system to execute the redundant workflow (R-1). Upon received execution request, the *workflow execution engine* runs the actions defined in the *unified input storage* module to prepare the unified inputs (R-2). Then the stored workflow action are loaded from the *workflow storage* module (R-3) and

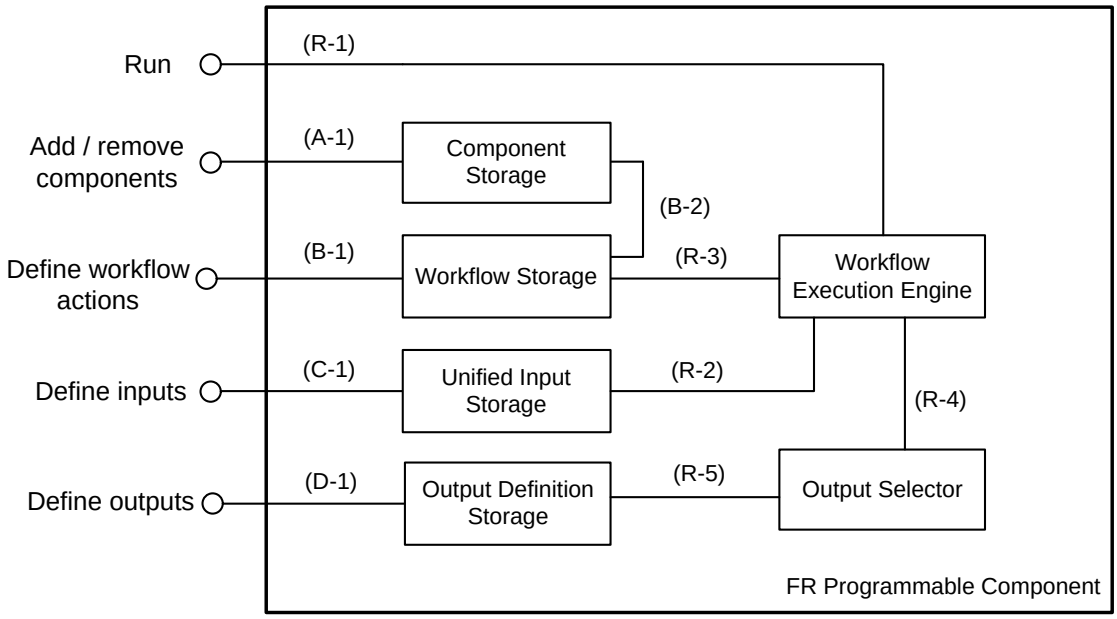


Figure 7.6: Architecture of forward recovery programmable component.

executed for all the redundant components in parallel. Upon the completed workflow execution, the *output selector* module is invoked. The module loads the output preparation actions form the *output definition storage* and executes them (R-5). Once the outputs have been prepared, the *output selection module* runs an implementation of selection algorithm and sets the final output values. If the selection algorithm is able to detect errors, workflow execution engine is prompted to perform additional actions.

N-Version Programming

The *n-version programming* programmable component is based on the n-version programming fault tolerance method. Similarly like in the case of recovery blocks principle, multiple semantically equivalent, components are included into the application workflow. Unlike in the recovery blocks principle, the components are run in parallel, making it possible to detect errors by observing component outputs.

The interface of the *n-version programming* programmable component stems from the architecture of forward recovery programmable components and is implemented as a web widget, presented in Figure 7.7. The image shows how the particular consumer operations relate to the user interface constructs. Consumer operations are performed using the presented interface as described in the following use case example.

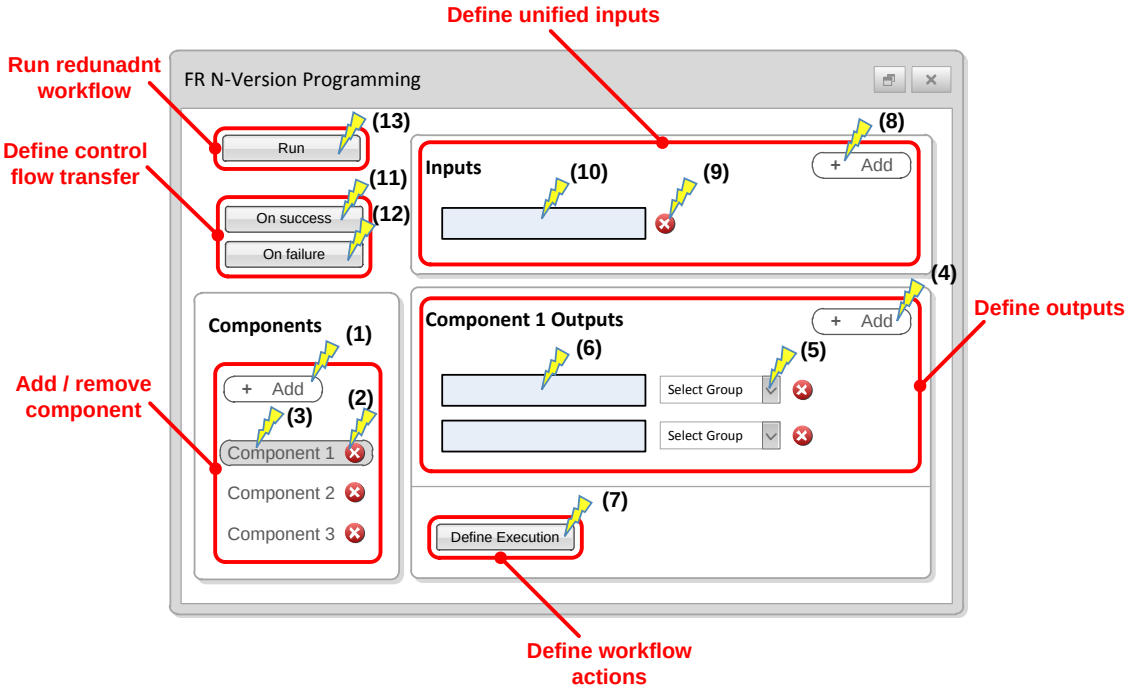


Figure 7.7: N-version programming programmable component interface.

Since the programmable component supports a multi-version fault tolerance method, consumer needs to be provided with ability to associate semantically equivalent components with the programmable widget. This is performed through the *components panel* where consumer can use controls add (1) or remove (2) components form the redundant component list. By selecting a component in the list (3), consumer displays the *component details* panel. In the *component details* panel consumer can configure outputs and workflow actions for each particular component. The outputs are configured by adding output fields (4). Each output field can be marked with an identifier using the *select group control* (5). Outputs with the same identifier are considered to be semantically equivalent and will be processed by the selection algorithm. Apart from selecting the identifier, consumer also defines data flow actions that copy the component outputs into appropriate fields (6). Except for output mapping, consumer needs to define workflow actions for each individual component using the *define execution* control (7).

If components need to be provided with the same input upon execution, consumer secures this property performing the *define unified inputs* operation. Consumer adds (8) or removes (9) input fields using controls in the *inputs* panel. Then, consumer defines data flow operations to link the input fields with the data sources (10). The stored data flow operations are performed once, prior to invocation of the first component in the list.

Additional constructs to define control flow transfer are available. Unlike in the *recovery blocks* programmable component, these actions are unique for all the components and are defined only once. As defined previously, the *on success* control (11) is used to transfer control flow if no errors have been detected. On the other hand, the *on failure* control (12) is used to transfer the control flow when an error has been detected, i.e. no output can be produced by a selection algorithm.

Finally, redundant workflow that incorporates all the associated semantically equivalent components is executed using the *run* control (13).

Chapter 8

Evaluation

This chapter presents evaluation of the weak point recommendation algorithms introduced in this thesis. The goal of the presented evaluation is to determine how the algorithms perform for both artificial and real-world data sets with regard to the recommended solution accuracy and computational performance. Section 8.1 describes two basic evaluation scenarios used to assess the algorithms named application-wise and data-set-wise component improvement. Computational performance and accuracy measures used in the presented evaluation are described in Section 8.2. Exact experimental setups applied in both evaluation scenarios are described in Section 8.3. The evaluation considers two data sets, an artificial data set and a real-world data set based on a collection of Yahoo Pipes composite applications (web mashups). The results of evaluation for the artificial data set and Yahoo Pipes are presented in Section 8.4 and Section 8.5 respectively. Furthermore, impact of the atomic component's mutual dependence on reliability model accuracy is evaluated in Section 8.6. Finally, the evaluation results are summarized in Section 8.7.

8.1 Evaluation Scenarios

For the purposes of evaluating the weak point recommendation algorithms presented in this thesis, two evaluation scenarios are considered. In application-wise evaluation scenario, atomic component improvements are performed separately for each composite application based on its own weak point recommendation results. On the other hand, data-set-wise scenario considers component improvements based on weak point recommendation results aggregated from all composite applications in the data set. Both evaluation scenarios are described in a greater

detail in the following subsections.

8.1.1 Application-Wise Component Improvement

In application-wise component improvement, the component to be improved is strictly chosen based on composite application's own weak point list. Such an evaluation scenario best describes the use case in which an application developer manages the reliability of a particular composite application by focusing on its specific workflow weaknesses. A conceptual overview of the described evaluation scenario is presented in Figure 8.1. The evaluation scenario is performed on a data set of composite application reliability models. For each composite application reliability model in the data set, a list of weak points ordered by the influence on the overall application reliability is recommended. Based on the recommended list of weak points, the reliability improvement step is performed for each individual composite application, i.e. the most significant recommended weak point is replaced by a more reliable atomic component. This process is repeated until all the atomic components of the composite application have been improved. For each step of the evaluational scenario k, a set of evaluational measures is observed. The applied evaluation measures are described in Section 8.2.

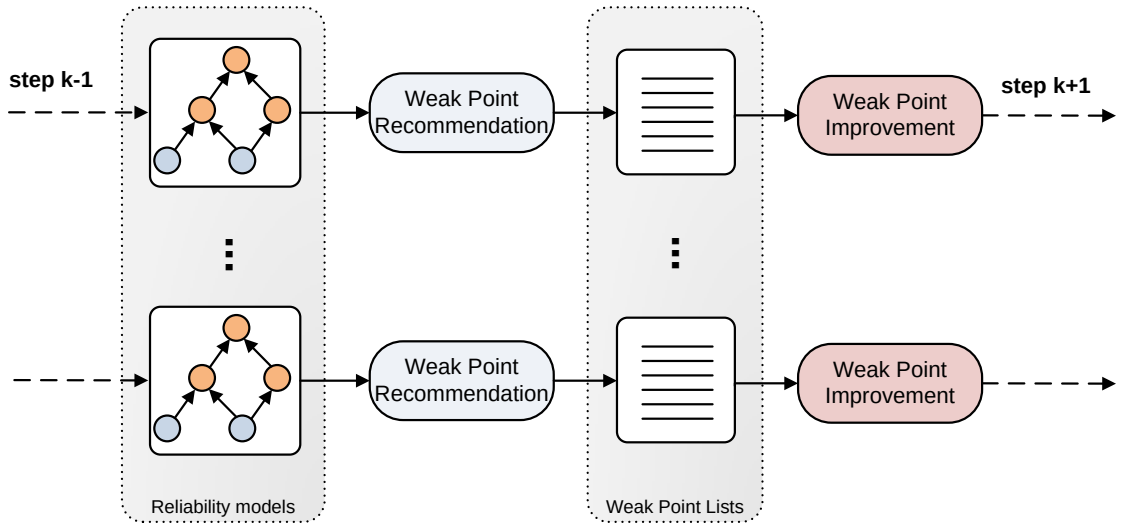


Figure 8.1: Application-wise component improvement.

The application-wise component improvement scenario is formally described by Algorithm 8.1. For each individual model in the data set, the described evaluation process is performed until all the atomic building components have been improved (lines $4 - 12$). At each step, the component to be improved is chosen from the recommended weak point list (line 7). It

is possible that a recommendation algorithm could place the same component at the top of the recommended list in two consecutive improvement steps. Since the presented scenario considers that each component can be improved strictly once, the one with lowest index in the recommended list that still has not been improved, i.e. the unimproved component with most influence on the overall composite application reliability, is chosen (line 7). The component improvement is performed by increasing the component's reliability by a fixed or proportional amount. This step is equivalent to performing a component replacement or introducing fault tolerance mechanisms into the application workflow. Finally, the reliability of the improved application is recalculated along with the other improvement results (lines $8-10$). The results are stored and analyzed using evaluation measures defined in Section 8.2.

```
 1: function AppWiseComp(dataSet)
 2:   for all models in dataset do
 3:     stepNumber ← 0
 4:     repeat
 5:       initRel ← calculate model reliability
 6:       wpList ← calculate a list of weak points
 7:       improve component with lowest index in wpList
 8:       finRel ← calculate model reliability
 9:       relGr ← finRel − initRel
10:       stepNumber ← stepNumber + 1
11:       store results: finRel, relGr, stepNumber
12:     until all components are improved
13:   end for
14: end function
```

Algorithm 8.1: Improve most significant application-wise component.

8.1.2 Data-Set-Wise Component Improvement

In contrast to the previously described evaluation scenario, data-set-wise component improvement approach applies component selection based on a weak point list aggregated from weak point lists of each individual composite application in the data set. The premise of such an evaluation scenario is that composite applications in the data set share building components, e.g. services within the cloud infrastructure. In that case it would be possible to analyze which individual component improvements could be made to achieve best average reliability growth for the whole data set. Such an approach can be leveraged by the owner of the application set or administrator of the cloud infrastructure to properly allocate computational resources with the

aim of achieving highest average application reliability increase. In contrast to the application-wise improvement, such an increase would be facilitated by improving a single component, rather than a set of atomic components as it best suites each individual composite application. An overview of the described evaluation scenario is presented in Figure 8.2. The evaluation is performed on a data set of composite application reliability models. For each composite application reliability model, an ordered list of weak points by the influence on the overall application reliability is recommended. The set of weak point lists is then aggregated into a single list using appropriate aggregation methods for each particular weak point recommendation algorithm. Based on the aggregated weak point list, reliability improvement step is performed and evaluated for each individual application. It should be noted that in case the recommended component is not a building block of a certain application, the reliability of the application in question is not improved in that particular improvement step. Although such an approach is expected to yield a lower overall step-wise reliability increase, it is useful in determining how the computational resources should be allocated between the atomic components.

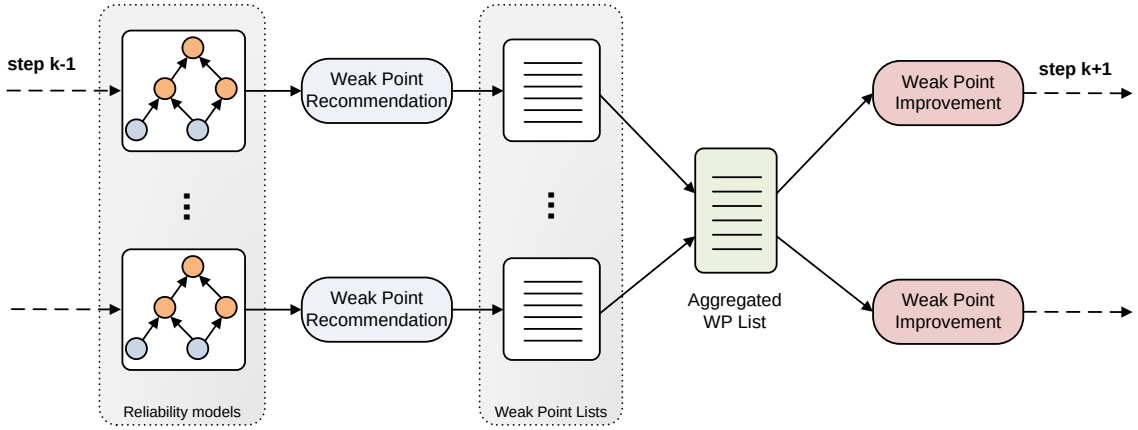


Figure 8.2: Data-set-wise component improvement.

The data-set-wise component improvement scenario is formally described by Algorithm 8.2. The evaluation process is performed until all the atomic building components have been improved (lines $4-17$). At each step, an extended list of weak points is first calculated for each application in the data set (lines $5-8$). Along with the order of components, list stores the estimated weights a particular component has on the overall composite application reliability. The set of weak point lists is then aggregated using methods described in Section 8.1.3 (line 9). The component to be improved in all composite applications is chosen from the aggregated weak point list. In case the selected component is not a building block of a given application,

that particular application remains unaffected. Since the scenario considers that each component can be improved strictly once, the one with lowest index in the recommended list that still has not been improved, i.e. the unimproved component with most influence on the overall composite application reliability, is chosen (line 7). As in case of application-wise scenario, the component improvement is performed by increasing the component's reliability by a fixed or proportional amount. Finally, step-wise reliability improvement results are stored and analyzed using measures defined in Section 8.2 (lines $13 - 15$).

```
1:  function BESTAVGCOMP(dataSet)
2:    stepNumber ← 0
3:    repeat
4:      wpListArray ← [ ]
5:      for all models in dataset do
6:        wpList ← calculate a list of weak points
7:        wpListArray.add(wpList)
8:      end for
9:      aggregatedList ← aggregate lists in wpListArray
10:     for all models in dataset do
11:       initRel ← calculate model reliability
12:       improve component with lowest index in aggregatedList
13:       finRel ← calculate model reliability
14:       relGr ← finRel − initRel
15:       store results: finRel, relGr, stepNumber
16:     end for
17:   until all components are improved
18:   stepNumber ← stepNumber + 1
19: end function
```

Algorithm 8.2: Improve most significant data-set-wise component.

8.1.3 Weak Point List Aggregation Methods

This section describes weak point list aggregation methods used in data-set-wise component improvement evaluation scenario. A separate aggregation measure is applied to the reliability sensitivity analysis algorithm *Wp-Influence* and the heuristic algorithms *WP-WeakestPath*, *WP-WeightedPath* described in Chapter 6.

In general, the aggregated weak point list is defined as a sequence of ordered pairs:

$$AggWPList = [(c_j, w_j)_i] \mid 0 < i, j \leq n \tag{8.1}$$

where n is the number of atomic components, i is the position of the ordered pair within the

sequence, c_i is the component identifier and w_i aggregated influence factor for component c_i. The aggregated weak point list is ordered by the descending influence factor w_i. Therefore, the following condition holds:

$$\forall (c_j, w_j)_i \wedge (c_l, w_l)_k \mid i \neq k, \text{ if } w_j > w_l \implies i < k \tag{8.2}$$

Thus, components corresponding to ordered pairs positioned closer to the beginning of the aggregated list are presumed to have greater influence on the average application reliability within the data set. The aggregated influence factors w_i are calculated differently for specific recommendation algorithms.

WP-Influence

$$w_i = \sum_{j=1}^{n} relGr_{i,j} \tag{8.3}$$

where n is the number of composite applications in the data set and $relGr_{i,j}$ reliability growth achieved performing sensitivity analysis for component i in composite application j using Algorithm 6.1.

WP-WeakestPath and WP-WeightedPath

In case of *Wp-WeakestPath* and *Wp-WeightedPath* algorithms the influence weight factor w_i is calculated based solely on the position of the component c_i in weak point lists of individual applications.

$$w_i = \sum_{j=1}^{n} N_{max} - wpList_j^{-1}(c_i)) \tag{8.4}$$

where N_{max} is the total number of distinct atomic components and $wpList_j^{-1}(c_i)$ index of component c_i in $wpList_j$.

8.2 Evaluation Measures

The following computational performance and reliability improvement evaluation measures are used throughout the rest of the chapter.

8.2.1 Computational Performance Measure

Execution time has been selected as the computational performance measure used to evaluate the weak point recommendation experiments. The measure represents average execution time needed to calculate a single list of weak points, i.e. perform a single component improvement. The measure is defined as follows:

$$AvgExec = \frac{\sum_{i=1}^{N} t_i}{N} \tag{8.5}$$

where t_i is the execution time for the i^{th} application in the data set and N is the size of data set. All the performance experiments described in this chapter were conducted on an Intel Core 2 Quad 2.66GHz CPU with 8GB of memory, running Ubuntu 11.10 operating system.

8.2.2 Recommendation Accuracy Measures

The presented measures are used to evaluate the accuracy of weak point recommendation algorithms, i.e. the accuracy of recommended weak point lists. Since a weak point list is in effect a sequence in which the components are to be improved, the measures show how the differences in recommended sequences impact the increase in reliability.

Average reliability

The average composition reliability is defined as the arithmetic mean of all composite applications' reliabilities in the data set. It is given by the following equation:

$$AvgRel = \frac{1}{N} \sum_{i=1}^{N} rel_i \tag{8.6}$$

where rel_i is the reliability of the composite application i and N is the total number of applications in the data set.

Average reliability growth

The average reliability growth is defined as the average reliability increase for all composite applications achieved between two subsequent reliability improvement steps. The measure is

defined as follows:

$$RelGr_k = \frac{1}{N}\sum_{i=1}^{N}(rel_{i,k} - rel_{i,k-1}) \tag{8.7}$$

where $rek_{i,k}$ is the reliability of composite application i in step k and N is the total number of compositions in the data set.

Total reliability growth

The total reliability growth is defined as the increase in reliability between the composite application's initial reliability and reliability achieved after k component improvements:

$$TotRelGr_k = rel_{i,k} - rel_{i,0} \tag{8.8}$$

where $rel_{i,0}$ is the initial reliability of the composite application i, $rel_{i,k}$ the reliability after k component improvements.

Average number of improvement steps

The average number of improvement steps required to reach a reliability threshold t is defined as follows:

$$AvgSt_t = \frac{1}{N}\sum_{i=1}^{N} st_i \quad | \ \forall i \ \ TotRelGr_{st_i} \geq t \tag{8.9}$$

where st_i is the number of improvement steps required for application i to reach the reliability threshold t and N the total number of applications that reached t.

Average reliability growth rate

The average reliability growth rate brings into correlation achieved reliability growth and spent system resources. Specifically, the measure correlates average total reliability growth and the execution time required to achieve it. The average reliability growth rate measure is defined as follows:

$$AvgGrRate_i = \frac{1}{N}\sum_{j=1}^{N}\left(\frac{TotRelGr_{i,j}}{t_{i,j}}\right) \tag{8.10}$$

where $TotRelGr_{i,j}$ is the total reliability growth for the application j after i improvement

steps and $t_{i,j}$ total execution time from initial to the i^{th} reliability improvement step for application j.

Overheads other than the weak point list recommendation process, like locating a replacement component or estimating the reliability of an atomic component, impact the overall performance of the proposed reliability improvement method. For the purposes of the experiments, all the additional overhead is defined as fixed execution time t_o for all the improvement steps performed for all composite applications. Following this simplification the total execution time in (8.10) is redefined as:

$$t'_{i,j} = t_{i,j} + i \cdot t_o \tag{8.11}$$

where i is the number of performed improvement steps, and $t_{i,j}$ weak point list recommendation execution time for composite application j.

8.2.3 Error Measures

In order to evaluate the difference in accuracy between recommendation algorithms, four standard error measures are applied. *Mean square error* (*MAE*) and *root mean square error* (*RMSE*) measures are respectively defined by the following equations:

$$MAE = \frac{1}{N} \sum_{i=1}^{N} |x_{1,i} - x_{2,i}| \tag{8.12}$$

$$RMSE = \sqrt{\frac{1}{N} \sum_{i=1}^{N} (x_{1,i} - x_{2,i})^2} \tag{8.13}$$

where $x_{1,i}$ and $x_{2,i}$ are two compared values on a set of entries $i \in [1, N]$. Both measures are positively oriented and range from 0 to infinity.

In addition, two percentage error measures *mean absolute percentage error* (*MAPE*) and *root mean square percentage error* (*RMSPE*) are used, defined as follows:

$$MAPE = \frac{1}{N} \sum_{i=1}^{N} \left| \frac{x_{1,i} - x_{2,i}}{x_{2,i}} \right| \tag{8.14}$$

$$RMSPE = \sqrt{\frac{1}{N} \sum_{i=1}^{N} \left(\frac{x_{1,i} - x_{2,i}}{x_{2,i}} \right)^2} \tag{8.15}$$

where the value of $x_{1,i}$ is compared against $x_{2,i}$ on a set of entries $i \in [1, N]$. Both measures are positively oriented and range from 0 to infinity.

8.3 Experimental Setup

As stated previously, one of the key metrics in evaluating the presented weak point recommendation algorithms is reliability growth, i.e. the increase in reliability achieved after replacing the weak points in the recommended order. Specifically, the choice of weak points should facilitate as great reliability growth as possible, while requiring as least component improvement steps as possible. Two factors influence the achieved reliability growth: the structure of the composite application, along with the initial and final reliability of its atomic building components. In order to evaluate how structure of applications impacts the reliability improvement method, two distinct data sets are considered in the experimental setup: the artificial data set described in Section 8.4.1 and the collected real-world data set consistent of Yahoo Pipes composite applications described in Section 8.5.1.

Two experiments are conducted on the artificial and the Yahoo Pipes real-world data set. The goal of the conducted evaluation is to asses how the proposed weak point recommendation algorithms behave when atomic components are very similar or equal in reliability and when there is a more significant difference in the atomic components' reliability. In the first experiment (*experiment A*) the initial reliability of all components is equal and set to 0.99. For each atomic component a single replacement component of reliability 0.9999 is available (10% greater reliability). Such an experimental setup was chosen as it closely matches reliability values expected for systems based on high-availability architecture [259]. In the second experiment (*experiment B*), the reliability of the components varies and it is randomly chosen from the interval $[0.7, 0.9]$ with a uniform distribution. Such an experimental setup more closely matches systems that do not have a prearranged level of QoS, e.g. no service level agreements exist. Similarly like in the first experiment, a single replacement component is available for each initial atomic component of the composite application. The replacement components have a 20% greater reliability than the ones initially built into the workflow, i.e. they have the final reliability from the interval $[0.84, 1.0]$. The experiments for both data sets and experimental setups are separately performed for each evaluation scenario described in Section 8.1. The results are presented throughout the remainder of this chapter.

8.4 Artificial Data Set

This section presents the results of the evaluation for the artificial data set. Details on how the artificial data set was generated are given in Section 8.4.1. Results of the computational performance experiments are presented in Section 8.4.2. Finally, Sections 8.4.3 and 8.4.4 present the results of evaluation for application-wise and data-set-wise component improvement scenarios respectively.

8.4.1 Data Set Generation

The generated data set consists of 5000 randomly generated reliability models of composite applications. Each application is built out of 30 independent atomic components, i.e. no conditional dependence exists between the individual atomic components. Generated models have the maximum nesting depth, i.e. the maximum number of nodes between input and output random variables (length of influence path) set to 3. Furthermore, the number of parents and children nodes can not exceed the number of input random variables (atomic components). However, each input random variable has at least one child node, each hidden layer variable has at least one child and one parent node and the output random variable has at least one parent node. At each model level, the *repetition*, *branch*, *sequence* and *parallel* nodes are generated with an equal probability. The number of repetitions defined by the *repeat* nodes is randomly chosen form the interval $[1, 4]$, while all the *parallel* nodes require that at least one workflow sequence completes without a fault ($k = 1$). Finally, the probabilities of execution for all workflow branches defined by the *branch* element are equally set, i.e. the probability of execution for a single branch is $1/n$, where n is the total number of branches. Reliability of the atomic components, i.e. input random variables, is set according to the particular experimental setup applied in the evaluation process.

8.4.2 Performance Evaluation

In order to determine how the size of the model, i.e. the number of atomic components within a composite application, impacts the resource requirements for weak point recommendation algorithms defined in Section 6, a series of execution time measurements were performed. The measured value t_i was the time required by the algorithm to recommend a single weak point list for composite application i. Execution time t_i is a value computed by averaging out

the results of 10 independent consecutive measurements. The overall performance of the algorithm for the entire data set $AvgExec$ is calculated by applying the expression (8.5). Two experimental setups focusing on specific working conditions are presented.

For the purposes of first experiment setup, 10 data sets have been generated. Each data set consists out of a 1000 applications built using a different number of atomic components, as defined in Section 8.4.1. The number of components varies from 5 atomic components in the first data set to 50 atomic components in the final data set with a step of 5 components. This scenario is consistent with contemporary composite applications, that are usually built from from the beginning, using atomic building components rather than reusing complex ones. The results of the experiment are given by the graph in Figure 8.3. It should be noted that separate experiments are not conducted for the *WP-Influence-R* and *WP-WeightedPath-R* algorithms. As they are designed to be minor modifications, that do not entail increase in computational complexity, results for *WP-Influence* and *WP-WeightedPath* represent also the extended algorithms.

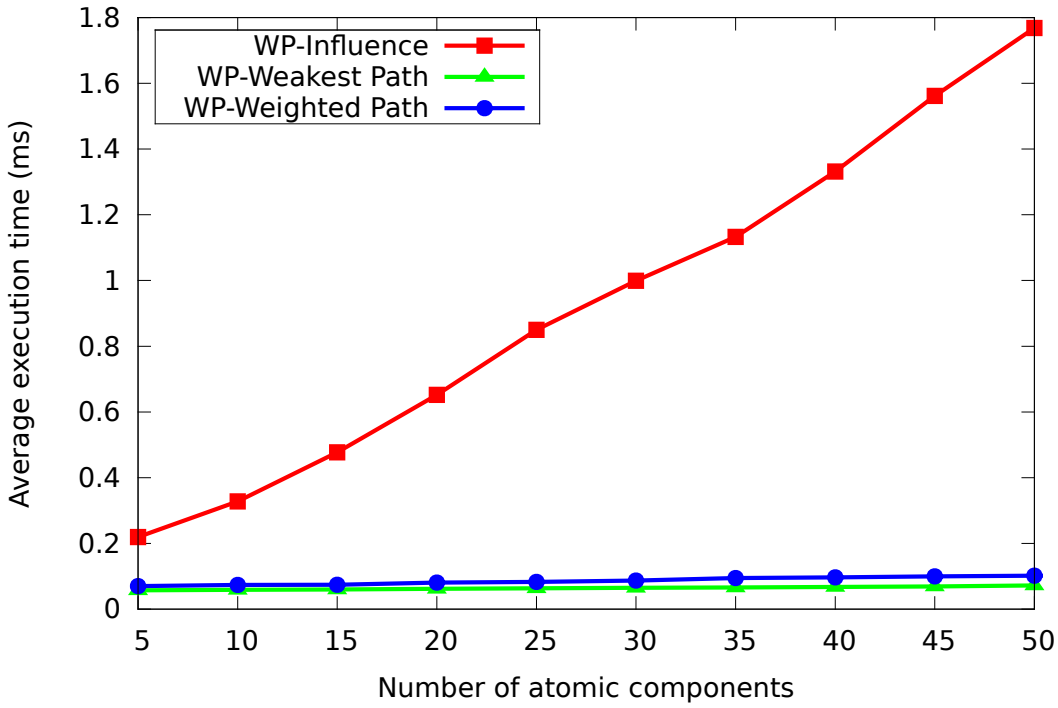


Figure 8.3: Execution times for artificial data set - 50 components.

The results indicate that the average execution time for a single weak point list recommendation in case of the *WP-Influence* algorithm is linearly dependent on the number of atomic components in a composite application. This is expected for a lower number of atomic components as recalculating reliabilities of influence paths is more computationally demanding than inserting into a sorted list ($T_{rCal}(l,i) >> T_{ins}(n)$). Thus, the expression (6.6) is reduced to

$O(n \cdot l \cdot i)$. In addition, for the generated data set l is fixed to 3 as the maximum nesting depth (maximum length of an influence path). Although the number of influence paths per input random variable is not bounded, it grows modestly in relation to the number of input random variables, and thus, can be regarded constant. The graph in 8.3 confirms these assumptions as average execution time of the *WP-Influence* algorithm grows linearly with the increasing number of input random variables in the generated reliability models.

In case of the heuristic algorithms *WP-WeakestPath* and *WP-WeightedPath* linear growth of execution time with the number of atomic components can also be observed in Figure 8.3. However, the execution time for heuristic algorithms grows at a less steeper angle than for the *WP-Influence* algorithm. Specifically, the average execution time for *WP-Influence* algorithm increases by a 105.3 and 48.4 times higher gradient then for the *WP-WeakestPath* and *WP-WeightedPath* algorithms respectively. This is due to the fact that the *WP-Influence* algorithm performs a sensitivity test for each atomic component separately, requiring multiple reliability recalculations for the entire composite application. On the other hand, heuristic algorithms *WP-WeakestPath* and *WP-WeightedPath* calculate a list of weak points by traversing the model only once, performing less complex computations. This makes them less dependent on the number of atomic component nodes in the model, as the presented results confirm.

The results can be further explained by studying the computational complexity analysis of the heuristic algorithms. The computational complexity of *WP-WeakestPath* algorithm is defined by expression (6.8). Since the nesting level of the generated models is bounded to 3, for the given number of input variables n there is a low number of influence paths per input random variable. Thus, the number of parents per the model node c is low and the expression (6.8) is reduced to $O(n+k)$. Therefore, the computational complexity of *WeakestPath* algorithm for the given experimental setup is linear. In case of the *WP-WeightedPath* algorithm, computational complexity is given by expression (6.10). For the given range of input random variable number n the following condition applies: $T_{whc}(k, c, s, l) + T_{agg}(k, c, s) >> T_{prep}(n)$. Furthermore, the length of influence paths l, i.e. the nesting depth of the model is constant (3 levels) and, thus, the computational complexity is reduced to: $O(s \cdot c \cdot k)$. Again, the number of parent and child nodes c and s can be regarded constant for the given experimental setup. Since there is a low number of influence paths per input random variable with a constant nesting depth l, k grows linearly with n.

In the second experimental setup, a wider range of the number of input random variables is

considered. For the purposes of experiment, 30 additional data sets have been generated. Each data set consists out of a 1000 applications built using a different number of atomic components. The number of components varies from 100 atomic components in the first data set to 3000 atomic components in the final data set with a step of 100 components. This scenario is more aligned with the future composite applications which will be built by reusing previously constructed applications (complex building components). In such cases the total number of atomic building components at the lowest level could escalate. The results of the experiment are given by the graphs in Figure 8.4 and Figure 8.5.

The results in Figure 8.4 demonstrate how the logarithmic component of the computational complexity expression (6.6) for the *WP-Influence* algorithm ($T_{ins}(n) = n \cdot \log(n)$) gains more significance with the increase of n. The presented results indicate logarithmic dependence of average execution time and the number of input random variables n. The results also indicate that the average execution time for *WP-Influence* grows much more rapidly than for heuristic algorithms. Therefore, the results for heuristic algorithms are presented separately in Figure 8.5 to give a better insight into their behavior. Results for both algorithms indicate a logarithmic dependence between average execution time and the number of atomic components in the model. In case of *WP-WeakestPath* algorithm, this behavior is due to the increase of the number of parent nodes c in the model (expression 6.8). This increase is expected as the total nesting depth of the models is fixed to 3. On the other hand, in case of the *WP-WeightedPath* algorithm, the logarithmic behavior is due to factor $T_{prep}(n) = n \cdot \log(n)$ in computational complexity expression (6.10). Due to more complex operations, namely computation of heuristics and aggregation of path weights, the average execution time for *WP-WeightedPath* grows more rapidly with the number of atomic components than in the case of *WP-WeakestPath* algorithm.

In conclusion, the presented results confirm that the *WP-Influence* algorithm is the most computationally demanding one. Furthermore, the *WP-WeightedPath* algorithm is conclusively more computationally demanding than the *WP-WeakestPath* algorithm. Finally, it can be stated that the heuristic algorithms are less dependent on the number of input random variables, i.e. atomic components, in the model.

8.4.3 Application-Wise Component Improvement Evaluation

The results for average reliability ($AvgRel$) in both experiments are given by the graphs in Fig. 8.6a and Fig. 8.7a. From the graphs it is visible that the *WP-Influence* algorithm achieves

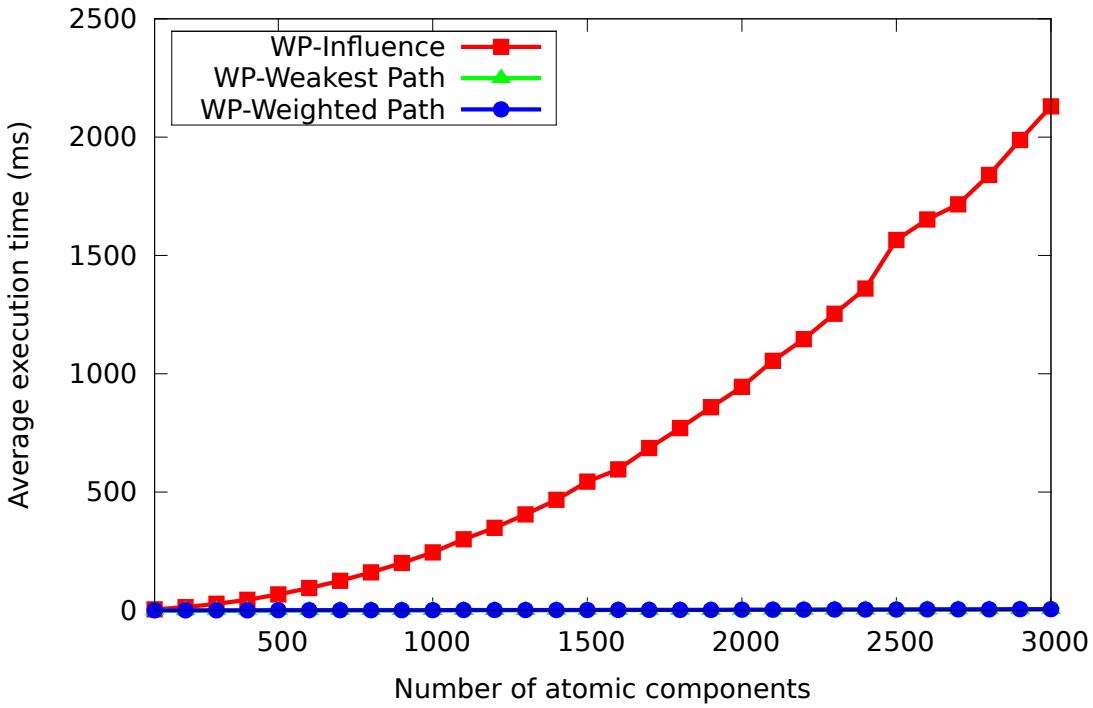


Figure 8.4: Execution times for artificial data set - 3000 components.

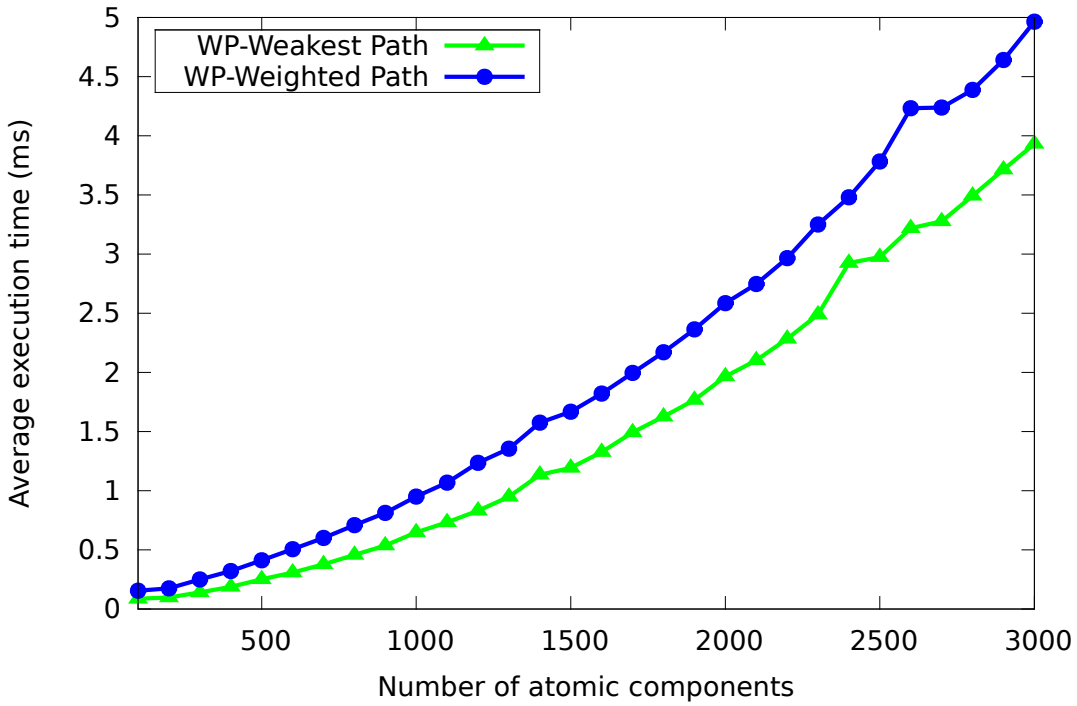


Figure 8.5: Execution times for artificial data set - 3000 components - heuristic algorithms.

higher $AvgRel$ then the heuristic algorithms at each composition improvement step. In addition, it can be concluded that the *WP-WeightedPath* algorithm, which takes into account multiple influence paths, achieves higher $AvgRel$ than the *WP-WeakestPath* algorithm in both experiments. Finally, it can be observed that *WP-Influence-R* and *WP-WeightedPath-R* algorithms expectedly

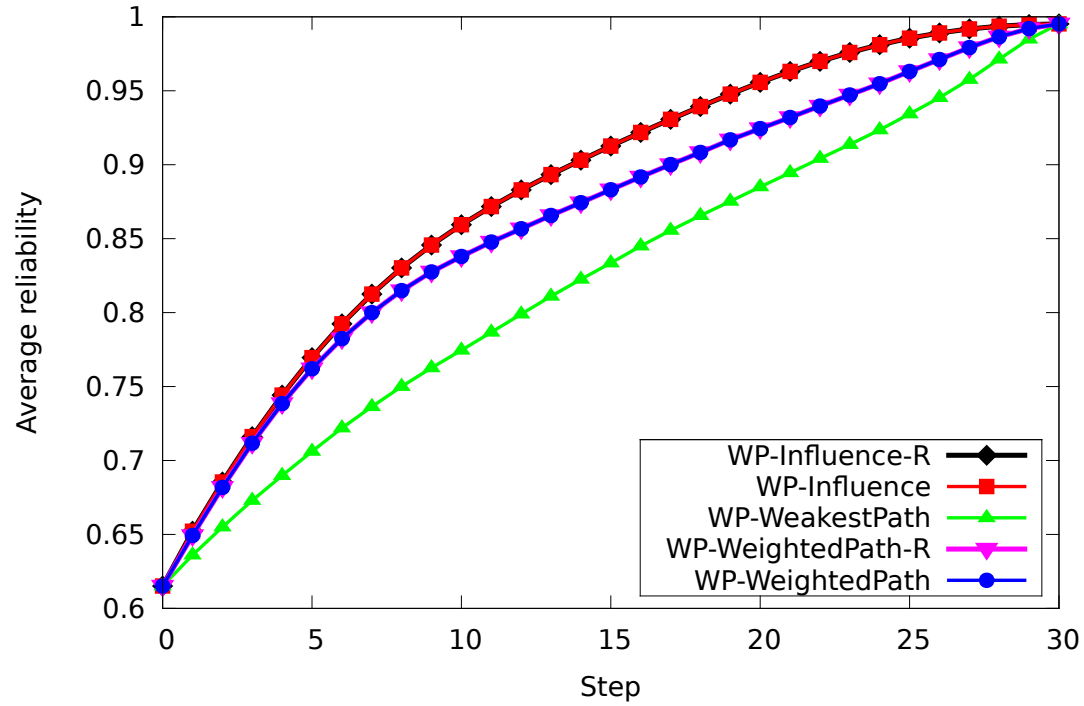
achieve higher $AvgRel$ than *WP-Influence* and *WP-WeightedPath* respectively as they take into account the reliability of replacement components.

To give a more precise insight, standard error measures are calculated to evaluate accuracy of recommendation algorithms against the optimal solution provided by the *WP-WeightedPath-R*. Thus, the value $x_{2,i}$ in expressions (8.12), (8.13), (8.14), and (8.15) stands for the average reliability ($AvgRel$) achieved by *WP-Influence-R* algorithm, the value $x_{1,i}$ stands for the average reliability achieved by the rest of the algorithms and N is the number of improvement steps (30).

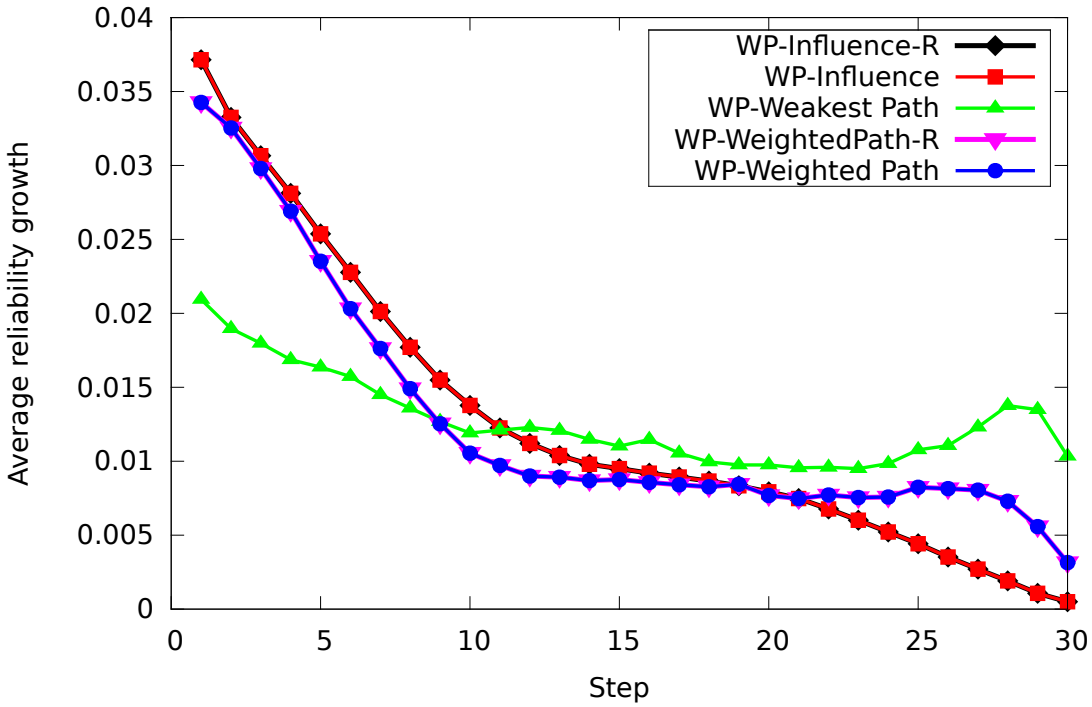
The results are presented in Table 8.1. As expected, *WP-Influence* has lower error measure values than the heuristic algorithms in both experiments. Specifically, the error measures are for several orders of magnitude lower than for the heuristic algorithms. Furthermore, *WP-WeightedPath* path is more accurate than *WP-WeakestPath*, e.g. it achieves up to 2.95 times lower *RMSE* in *experiment A*. Finally, it is confirmed that improvements incorporated into *WP-WeightedPath-R* yield higher accuracy, e.g. *RMSE* in *experiment B* is 1.56% lower than for *WP-WeightedPath*. Expectedly due to expression (6.3), in *experiment A* *WP-WeightedPath-R* performs the same as *WP-WeightedPath* since all replacement components have the same reliability. Finally, the results for percentage error measures indicate that all the algorithms achieve higher accuracy in *experiment A*. Since reliability of replacement components (0.9999) is close to 1, the *WP-Influence* achieves several orders of magnitude lower error measures in *experiment A* when compared to *experiment B*. Similar results hold for the heuristic algorithms as *WP-WeakestPath*, *WP-WeightedPath* and *WP-WeightedPath-R* algorithms achieve respectively 8.87, 20.76, and 20.24 times lower *RMSPE* in *experiment A* when compared to *experiment B*.

In order give a better insight into accuracy for the initial improvement steps and to analyze the trend of growth, the average reliability growth ($RelGr$) for both experiments is presented in Fig. 8.6b (*experiment A*) and Fig. 8.7b (*experiment B*). When comparing two weak point recommendation algorithms, a better performing algorithm should achieve higher initial average reliability growth and a lower average reliability growth at the final improvement steps. The results presented for both experiments confirm that the *WP-Influence-R* algorithm achieves highest average reliability growth during the initial improvement steps. It can also be concluded that the *WP-WeightedPath* and *WP-WeightedPath-R* algorithms outperform the *WP-WeakestPath* algorithm in both experiments.

Finally, Figure 8.8 presents the average number of composition improvement steps ($AvgSt$) required to reach a certain reliability threshold—denoted on the x-axis as the total reliability



(a) Average reliability



(b) Average reliability growth

Figure 8.6: Reliability growth results: initial 0.99, increase 0.0099.

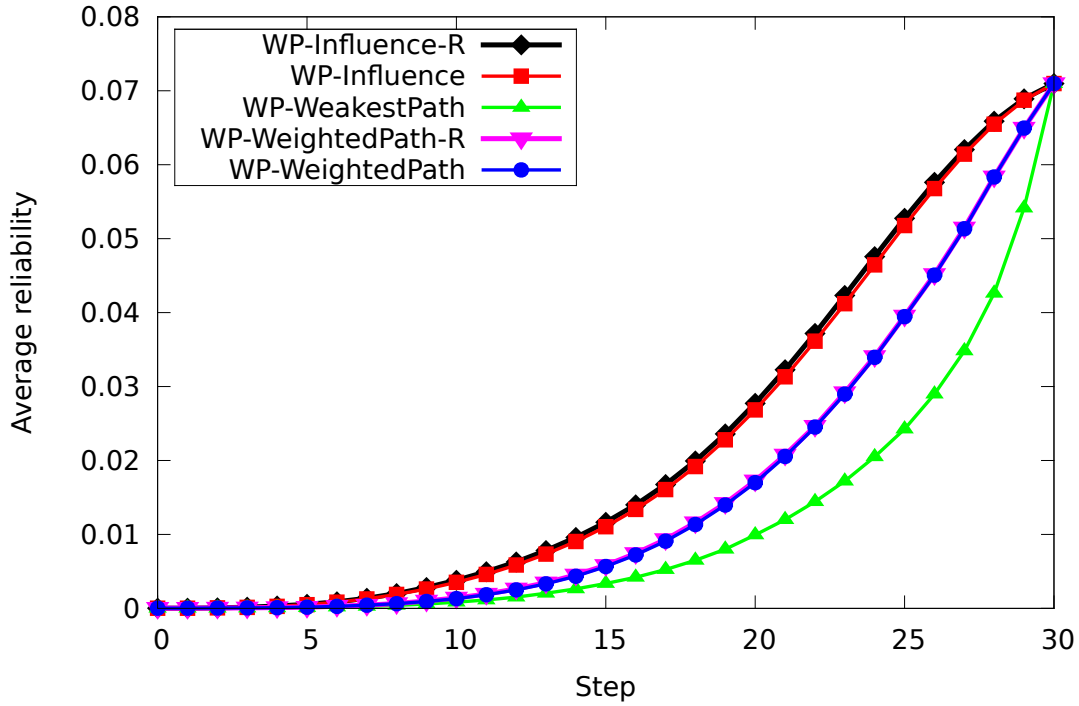
growth ($TotRelGr$). It should be noted that the arithmetic mean of the component improvement steps was calculated on the reduced data set, i.e. on a subset of the available composite applications that can achieve the given total reliability growth. Results for both experiments

Table 8.1: Difference in $AvgRel$ between recommendation algorithms and *WP-Influence-R*: artificial data set, application-wise improvement

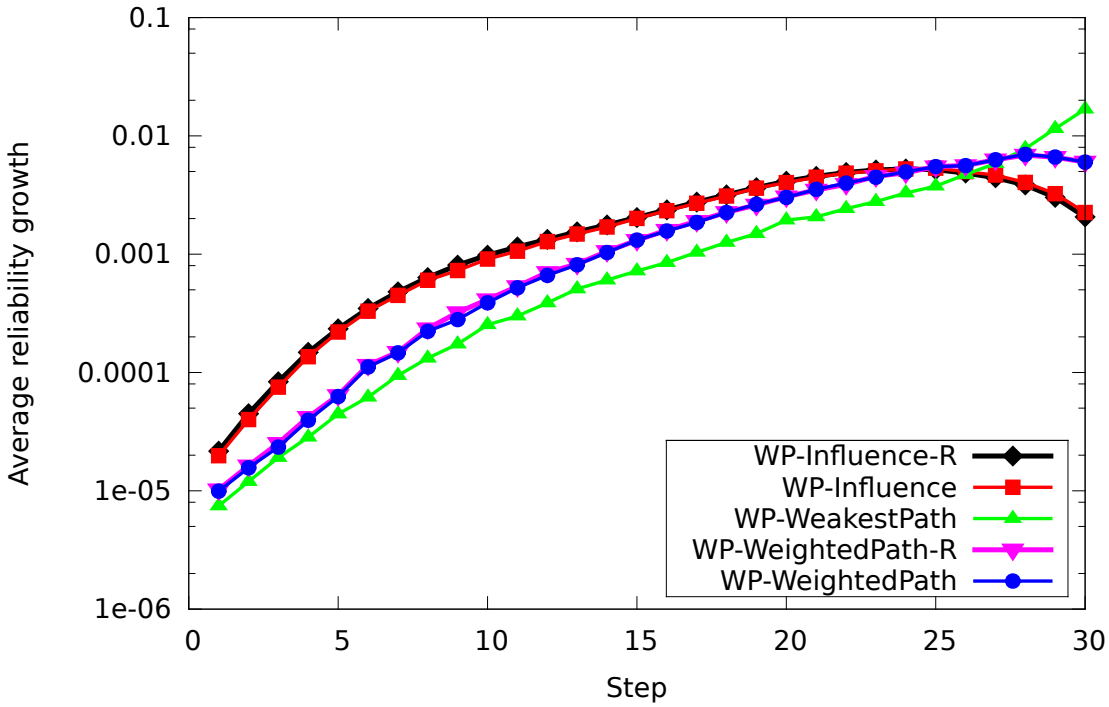
Algorithm	Measure	0.99, inc. 0.0099	0.7 - 0.9, inc. 20%
WP-Influence	*MAE*	0.000000001403482	0.000474431127909
	RMSE	0.000000007367383	0.000601585203192
	MAPE	0.000000002039215	0.046556422960173
	RMSPE	0.000000010747492	0.053636586671875
WP-WeakestPath	*MAE*	0.059908631114824	0.010822164214565
	RMSE	0.064505312660963	0.014829473572233
	MAPE	0.067967853476540	0.623348330011113
	RMSPE	0.073214212661829	0.649471059395158
WP-WeightedPath-R	*MAE*	0.019015265587158	0.005677979257328
	RMSE	0.021836964047474	0.007387740169723
	MAPE	0.020843018143821	0.436557019115393
	RMSPE	0.023625890217850	0.478267596098688
WP-WeightedPath	*MAE*	0.019015265587158	0.005781190143887
	RMSE	0.021836964047474	0.007503325530043
	MAPE	0.020843018143821	0.447353987984990
	RMSPE	0.023625890217850	0.490608737267258

in Figure 8.8a (*experiment A*) and Figure 8.8b (*experiment B*) expectedly indicate that the *WP-Influence* algorithm on average requires less improvement steps to reach a threshold than the heuristic algorithms. Specifically, in *experiment A*, the *WP-Influence* algorithm achieves the defined reliability threshold while performing 13.45% and 10.01% less improvement steps then algorithms *WP-WeakestPath* and *WP-WeightedPath* respectively. Similarly, in *experiment B*, the *WP-Influence* algorithm reaches the reliability threshold in 26.53% and 12.34% less improvement steps then *WP-WeakestPath* and *WP-WeightedPath* respectively. The results also expectedly indicate that *WP-Influence-R* and *WP-WeightedPath-R* perform more accurately than their unmodified versions.

The impact of the overhead t_o on average growth rate is analyzed for two distinct experimental setups (Figure 8.9). In both experimental setups the initial reliability is set equal to 0.9 for all the atomic components and the final reliability is set to 0.99 (10% increase). In the first setup we assume that replacement components are available at the time a list of weak points is calculated and that no other execution overheads for improving an atomic component exist ($t_o = 0ms$). The results shown in Figure 8.9a indicate that both *WP-WeakestPath* and *WP-WeightedPath* algorithms have a higher average reliability growth rate then the *WP-Influence* for all improvement steps. Furthermore, the *WP-WeightedPath* has a greater initial average growth rate than the *WP-WeakestPath*, but lower average growth rate for later improvement steps. The presented



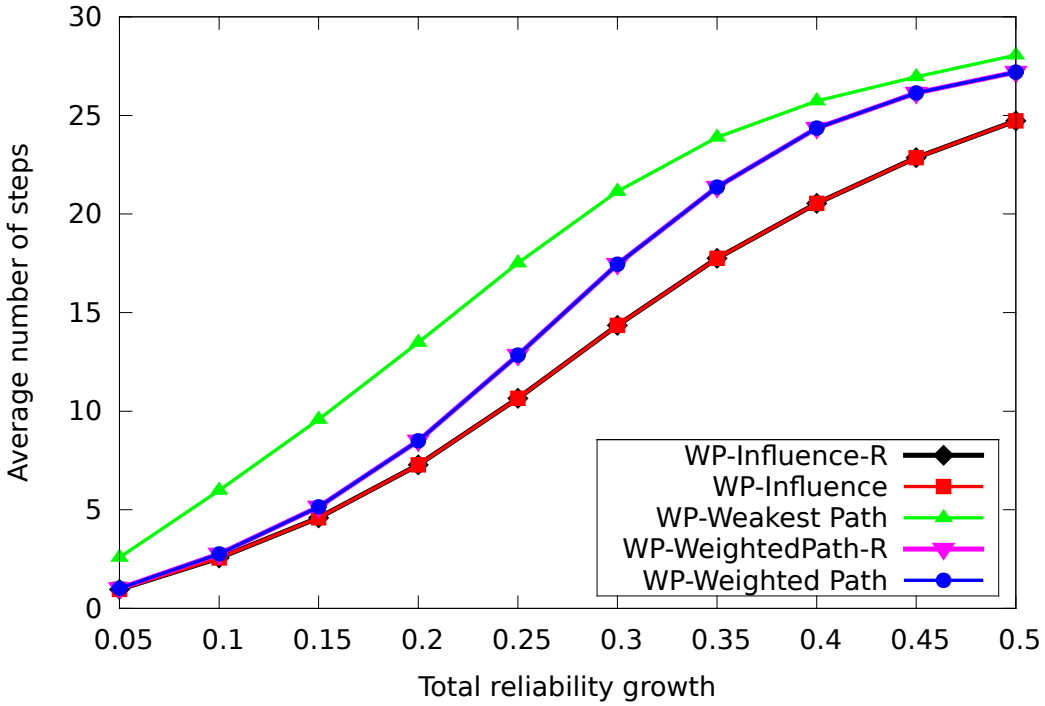
(a) Average reliability



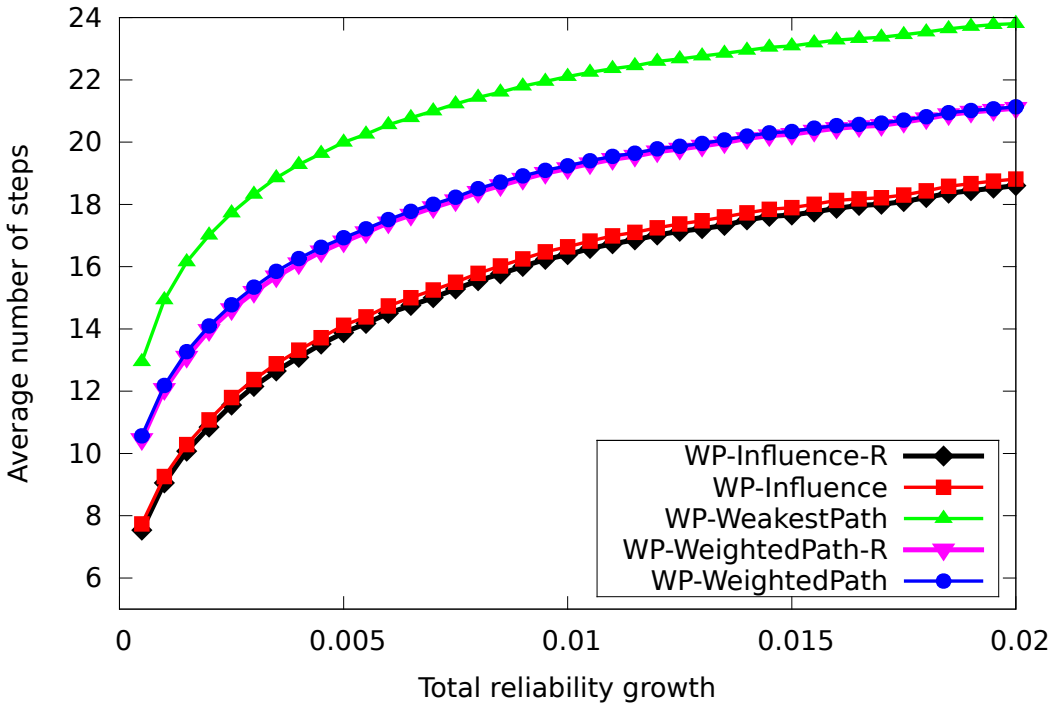
(b) Average reliability growth

Figure 8.7: Reliability growth results: initial 0.7 - 0.9, increase 20%.

results lead to a conclusion that better performance for the given data set could be achieved if initial replacement steps are done by utilizing *WP-WeightedPath* algorithm and the final steps by using *WP-WeakestPath* algorithm. Thus, the solution space can first be reduced using a more



(a) Average number of steps: initial 0.99, increase 0.0099



(b) Average number of steps: initial 0.7 - 0.9, increase 20%

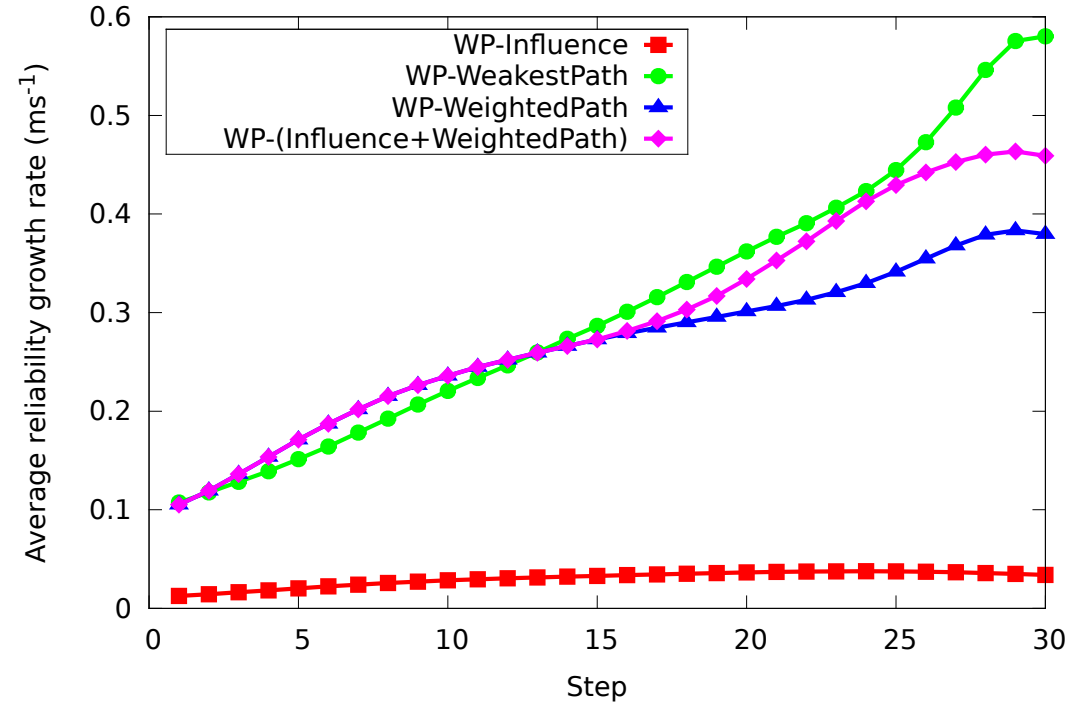
Figure 8.8: Average number of steps to reach a reliability threshold.

accurate but also more computationally intensive algorithm. Then a less accurate algorithm can be applied on the reduced solution space to achieve better average reliability growth rate. To support this claim, experimental results are presented for a hybrid algorithm in which the first

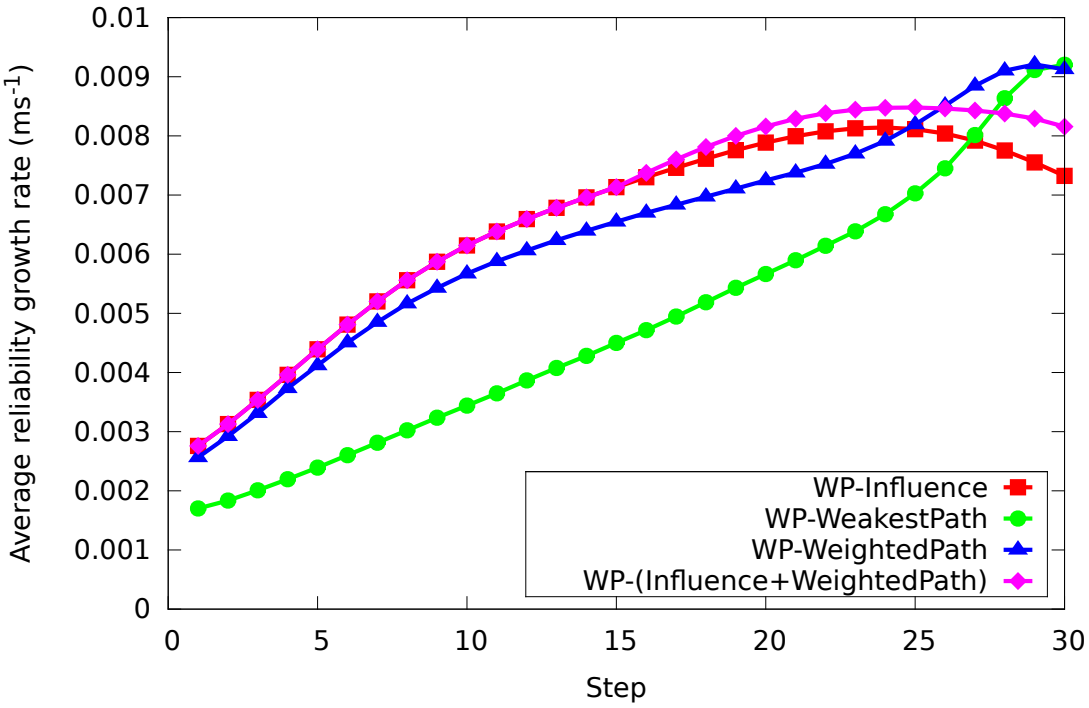
half of improvement steps (15 steps) are conducted by the *WP-WeightedPath* algorithm and the last half of improvement steps by the *WP-WeakestPath* algorithm. The results show that the hybrid algorithm achieves better average reliability growth rate in the final replacement steps than the *WP-WeightedPath* algorithm. However, the hybrid algorithm shows lower average reliability growth rate in the final replacement steps than the *WP-WeakestPath* algorithm. This is due to the fact that the *WP-WeightedPath* component of the hybrid algorithm has exhausted better weak point recommendations in earlier steps.

In the second experimental setup, shown in Figure 8.9b, an additional fixed overhead per step of $t_o = 3.5ms$ was added. In this case, the *WP-Influence* algorithm shows best initial average reliability growth since it achieves higher total reliability growth by performing less replacement steps. Again, it can be concluded that it is suitable to use the *WP-Influence* to perform weak point recommendation in reliability improvement steps for which it outperforms the other algorithms. Similarly like in the previous experimental setup, we show the results for the hybrid algorithm in which first half of replacement steps is conducted by the *WP-Influence* algorithm and the last half by the *WP-WeightedPath* algorithm. The results indicate an increase in average reliability growth rate of the hybrid algorithm when compared to the *WP-Influence* algorithm.

In order to further evaluate the impact of the additional overhead t_o on the applicability of the weak point recommendation algorithms, results of average growth rate for $t_o \in [0ms, 5ms]$ are shown in Figure 8.10. The average growth rate, shown on the z-axis in logarithmic scale, is calculated for each of the 30 possible improvement steps, as denoted by the y-axis. The results indicate that for different values of t_o, different algorithms achieve highest reliability growth rate given the number of conducted replacement steps. To give a better overview which algorithm performs best for the given t_o and the conducted number of replacement steps, a projection of graph in Figure 8.10a onto the x-y plane is shown in Figure 8.10b. It can be concluded that for the presented experimental setup, the number of steps in which the *WP-Influence* and *WP-WeightedPath* algorithms achieve the highest *AvgGrRate* grows rapidly with the overhead t_o in a step-like manner followed by a region of more moderate growth.The presented results can be used to reason about the design of a hybrid algorithm which would achieve better performance by utilizing the best performing weak point recommendation algorithm.



(a) Average growth rate ($t_o = 0ms$).

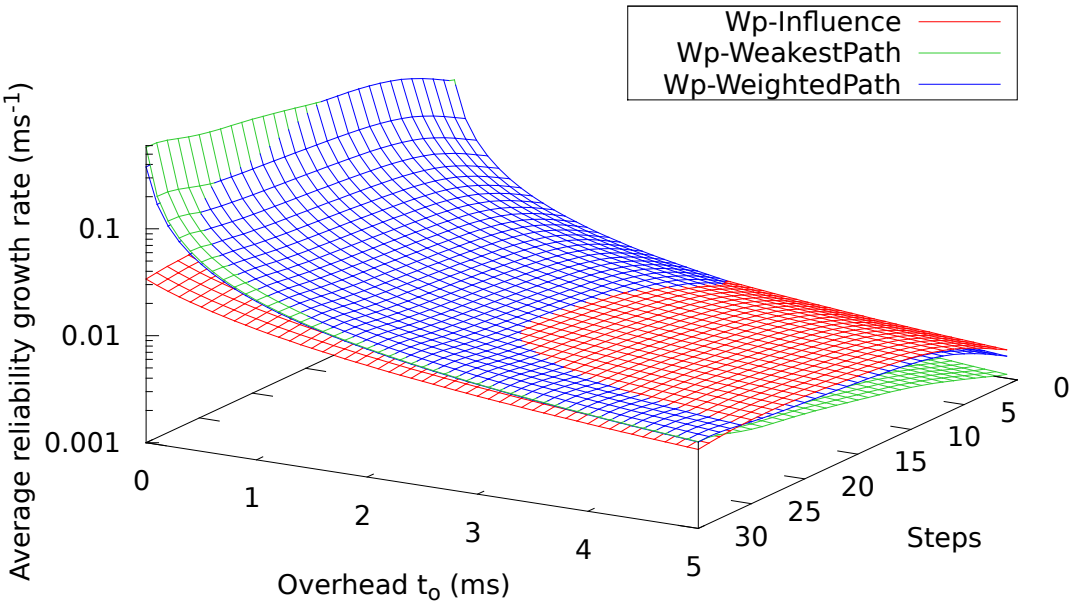


(b) Average growth rate ($t_o = 3.5ms$).

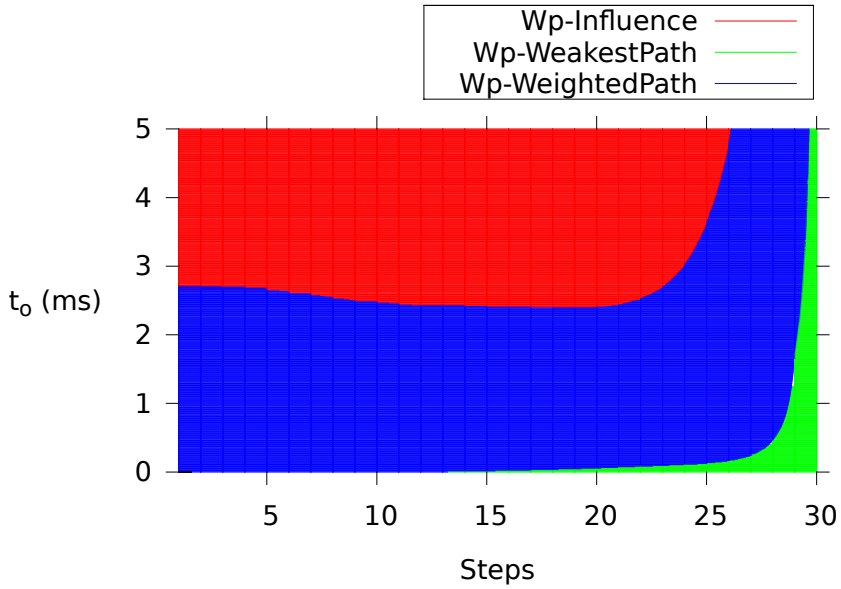
Figure 8.9: Average reliability growth rate: initial 0.9, increase 10%.

8.4.4 Data-Set-Wise Component Improvement Evaluation

The results for average reliability measure for both experimental setups of data-set-wise component improvement are given by the graphs in Figure 8.11a and Figure 8.12a. Although



(a) Average growth rate for t_o $0ms - 5ms$



(b) Best performing algorithm in relation to conducted improvement steps and overhead t_o

Figure 8.10: Impact of the overhead t_o on the $AvgGrRate$: initial 0.9, increase 10%

the experiments are conducted on the same data set as in Section 8.4.3, the difference between the recommendation algorithms is much less conclusive than for the application-wise improvement experiments. In addition a lower increase in reliability is achieved than in experiments

described in Section 8.4.3. This outcome is expected since at each replacement step the best replacement candidate is averaged out over the entire data set. Consequently, the best improvement candidate is not selected for each composite application at each improvement step causing a lower reliability increase.

The presented graphs indicate that the *WP-Influence-R* algorithm achieves on average higher reliability than the *WP-WeakestPath* and *WP-WeightedPath* algorithms. However, this is not the fact for the entire improvement interval as it was in application-wise improvement experiments, since the heuristic algorithms can at certain improvement steps perform as good as the *WP-Influence-R* and *WP-Influence* algorithms. Similarly, the *WP-WeightedPath* algorithm displays better average performance than the *WP-WeakestPath* algorithm. However, at certain improvement steps the *WP-WeakestPath* algorithm can outperform the *WP-WeightedPath* algorithm.

To further analyze distinctions in accuracy between the *WP-Influence-R* and the rest of the recommendation algorithms, error measures for average reliability ($AvgRel$) on the whole improvement interval are presented in Table 8.2. The results conclusively indicate that *WP-Influence* is more accurate than the rest of the recommendation algorithms as it has lower error measure values. In fact, in *experiment A*, *WP-Influence* matches the *WP-Influence-R* in accuracy. Furthermore, the results in both experimental setups confirm that *WP-WeightedPath* path is on average more accurate than *WP-WeakestPath*. Finally, by observing the percentage error measures, it can be concluded that all algorithms are more accurate in *experiment A* where reliability values of atomic components are higher. It should be noted that in both experiments the differences in accuracy are much less pronounced than in the application-wise evaluation scenario.

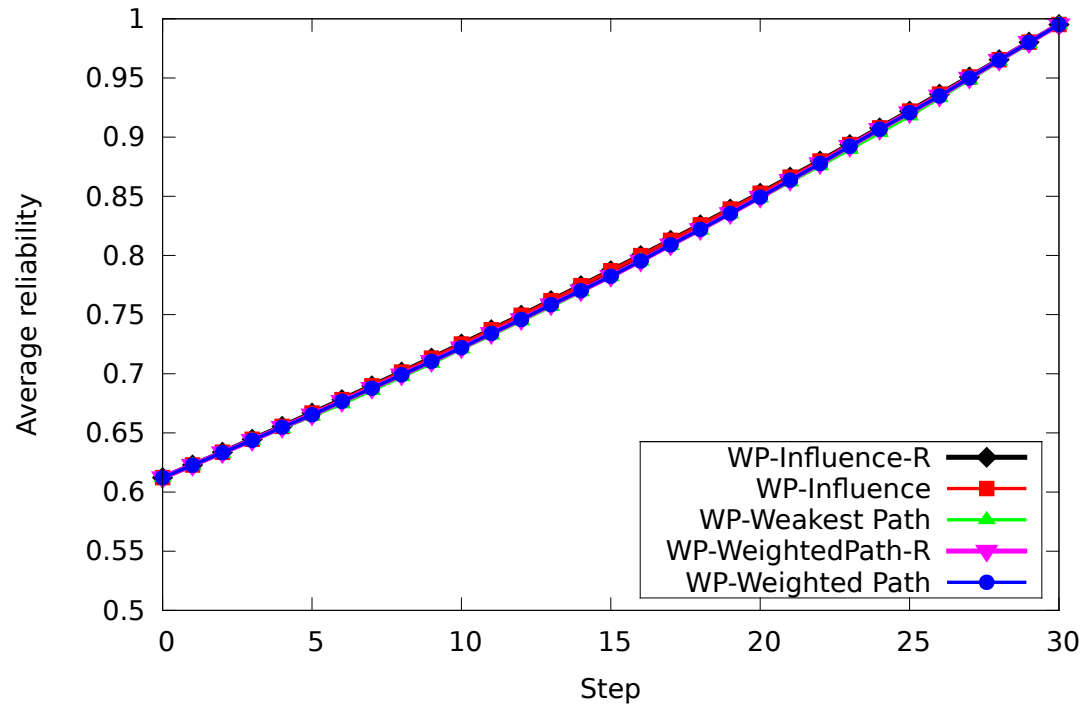
To further analyze the behavior of the recommendation algorithms, the average reliability growth is presented for both experiments in Figure 8.11b and Figure 8.12b. The presented results confirm that the heuristic algorithms closely match the performance of *WP-Influence*. It is visible that unlike in the application-wise improvement scenario, the reliability growth curves intersect *WP-Influence* more than once. This behavior indicates that the heuristic algorithms manage to correct the error in component recommendation and maintain the quality of solution close to the *WP-Influence* algorithm. The results for both experimental setups also indicate that the *WP-WeightedPath* displays a more stable behavior than the *WP-WeakestPath* algorithm since it deviates less from the curve of more accurate *WP-Influence* algorithm. Finally, the results confirm that *WP-Influence-R* and *WP-WeightedPath-R* are more accurate than their un-

Table 8.2: Difference in $AvgRel$ between recommendation algorithms and *WP-Influence-R*: artificial data set, DS-wise improvement

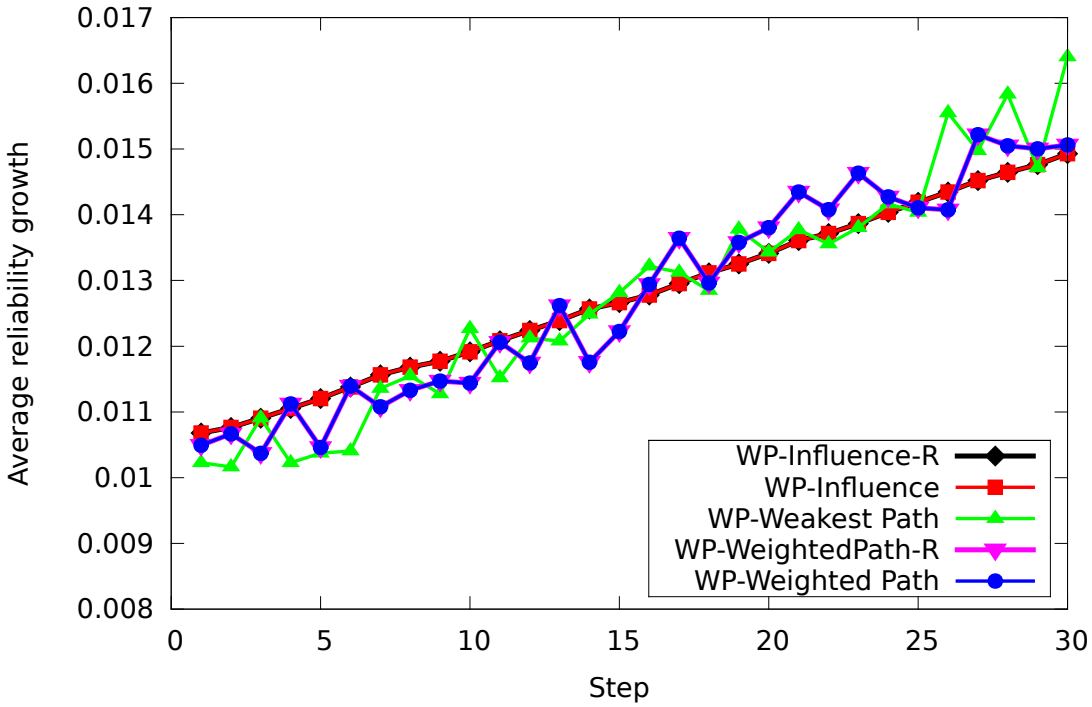
Algorithm	Measure	0.99, inc. 0.0099	0.7 - 0.9, inc. 20%
WP-Influence	*MAE*	0.0	0.000257128057820
	RMSE	0.0	0.000367429200260
	MAPE	0.0	0.202888450510625
	RMSPE	0.0	0.254300770278976
WP-WeakestPath	*MAE*	0.003473436834555	0.000817379769043
	RMSE	0.003780353641522	0.001294318605322
	MAPE	0.004401942231719	0.225760979549925
	RMSPE	0.004804381347203	0.259428704092128
WP-WeightedPath-R	*MAE*	0.002134985106179	0.000306291941399
	RMSE	0.002558979486279	0.000435801200143
	MAPE	0.002728392807155	0.208785254706772
	RMSPE	0.003268205480683	0.258326509146711
WP-WeightedPath	*MAE*	0.002134985106179	0.000306256945261
	RMSE	0.002558979486279	0.000435797119083
	MAPE	0.002728392807155	0.208521303757330
	RMSPE	0.003268205480683	0.257934563181939

modified versions. For instance, difference in accuracy for *WP-Influence* is expectedly more pronounced in *experiment A* since reliability of replacement components greatly differs from 1 than in *experiment B*.

Figure 8.13 shows the average number of composition improvement steps required to reach a certain reliability threshold—denoted on the x-axis as the total reliability growth. Since the $AvgSt$ measure, defined by expression (8.9), depends on the total reliability growth ($TotRelGr$) of individual applications, rather than the ($TotRelGr$) averaged out on the entire data set, lower values of $AvgSt$ do not necessarily imply higher accuracy. Specifically, higher average reliablity growth can be achieved by performing improvements that better suite a smaller number applications that have a large impact on the data set's average reliability. In that case, improvement solutions of lower accuracy would be applied to other applications in the data set, yielding a higher $AvgSt$ value. An example of such an occurence can be seen in Figure 8.13b for the initial reliability thresholds. For instance, to reach the reliability threshold 0.5×10^{-3}, the most accurate *Wp-Influence-R* requires 3.19% more improvement steps than the *WP-Influence* algorithm. Similar results hold for the other recommendation algorithms. On the other hand, to reach the 0.05 threshold, the *WP-Influence* requires 0.54% less improvement steps than the *WP-Influence*. This indicates that the *WP-Influence-R* algorithm provides highly accurate weak point recommendations for most of the applications in the data set.



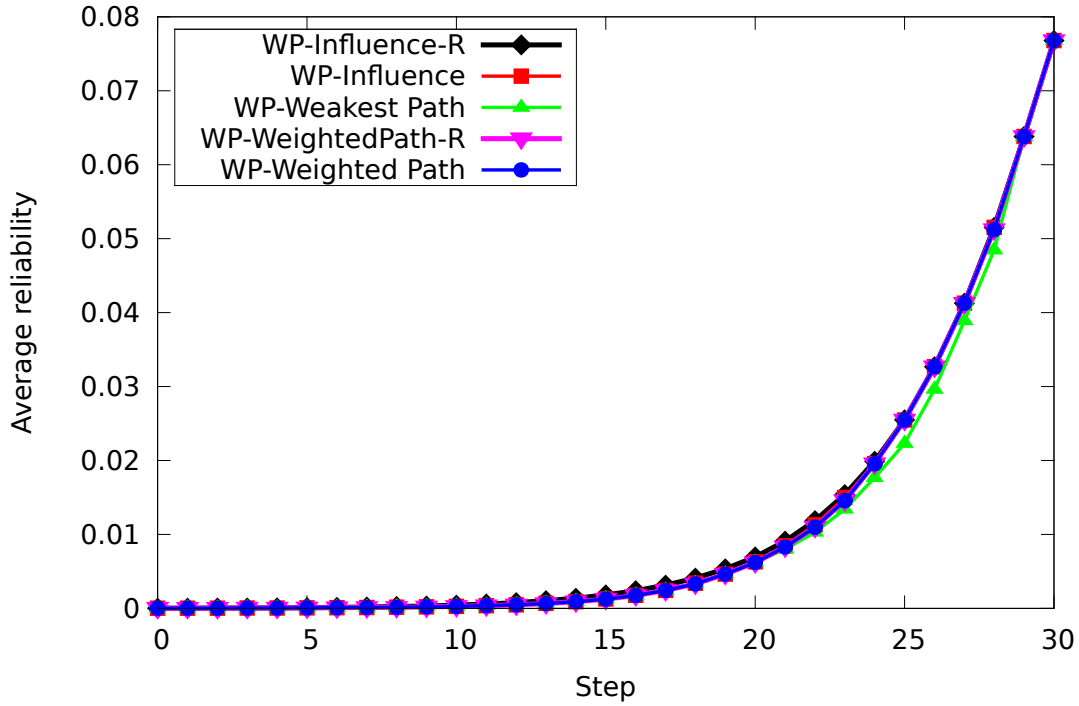
(a) Average reliability



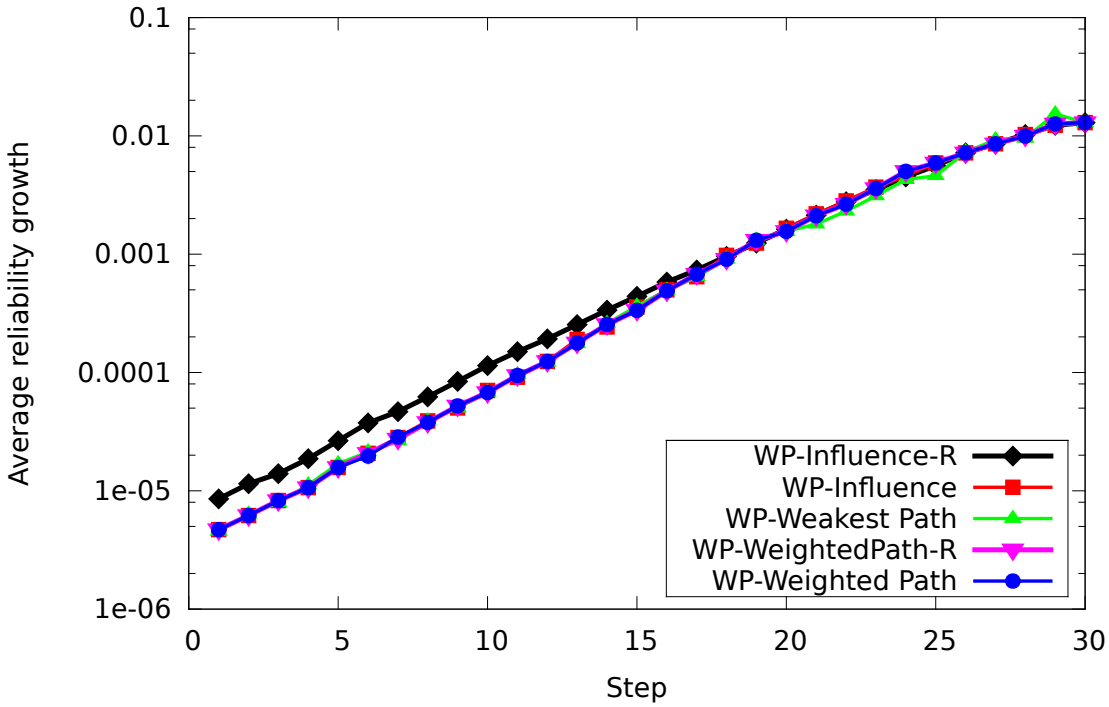
(b) Average reliability growth

Figure 8.11: Reliability growth results: initial 0.99, increase 0.0099.

In order to further evaluate the impact of the additional overhead t_o on the applicability of the weak point recommendation algorithms, results of average growth rate for $t_o \in [0ms, 500ms]$ are shown in Figure 8.14. In experimental setup the initial reliability of all components is set



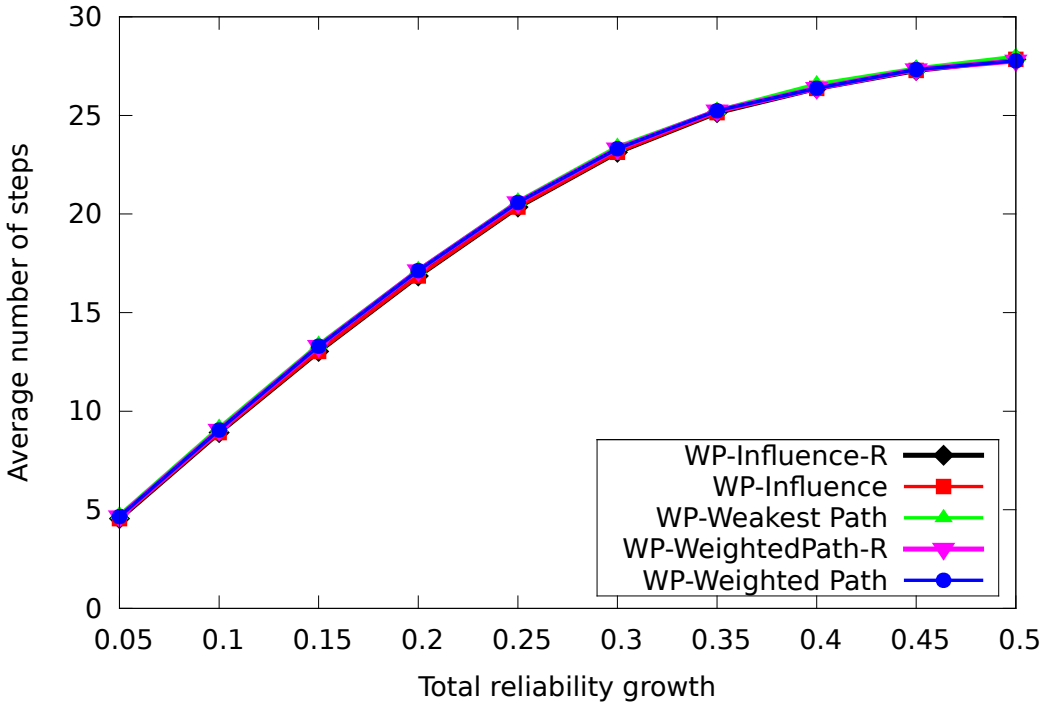
(a) Average reliability



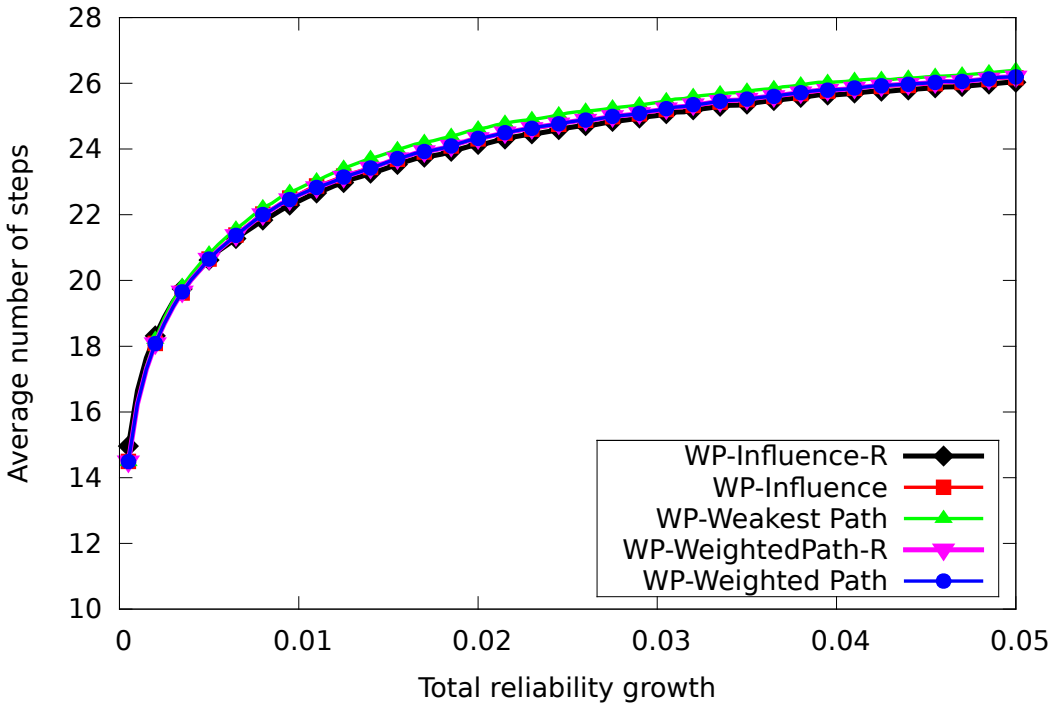
(b) Average reliability growth

Figure 8.12: Reliability growth results: initial 0.7 - 0.9, increase 20%.

equally to 0.9 and the final reliability is set to 0.99 (10% increase). The average growth rate, shown on the z-axis in logarithmic scale, is calculated for each of the 30 possible improvement steps, as denoted by the y-axis. The results indicate that a significantly larger overhead t_o



(a) Average number of steps: initial 0.99, increase 0.0099



(b) Average number of steps: initial 0.7 - 0.9, increase 20%

Figure 8.13: Average number of steps to reach a reliability threshold.

than in the application-wise scenario is needed in order to differentiate between the algorithms' performance. This is an expected result due to the fact that the heuristic algorithms closely match the performance of the *WP-Influence* algorithm. With regard to the applicability of weak

point recommendation algorithms, the results are less conclusive than in the application-wise scenario since the surfaces intersect on multiple occasions, causing discontinuities regardless of the increase in t_o. In fact, the heuristic algorithms oscillate around the best solution provided by the *WP-Influence* algorithm.

This behavior can be better observed in Figure 8.14b which shows a projection of graph in Figure 8.14a onto the x-y plane. For instance, a discontinuity independent of t_o between *WP-Influence* and *Wp-WeightedPath* can be observed in the 7^{th} improvement step. Since the *WP-WeightedPath* matches the performance of the *WP-Influence* algorithms in the 7^{th} improvement step, the curves will not intersect for any value of t_o. To prove this claim, it is necessary to observe the reliability growth rate measure defined by expression (8.10). If a single application in the data set is observed, the curves will intersect in case the following condition is met:

$$\frac{TotRelGr_{Inf,i}}{k \cdot (t_{Inf,i} + t_o)} = \frac{TotRelGr_{Wh,i}}{k \cdot (t_{Wh,i} + t_o)} \tag{8.16}$$

where $TotRelGr_{Inf,i}$ and $TotRelGr_{Wh,i}$ are total reliability growths achieved by application i when applying *Wp-Influence* and *WP-WeightedPath* algorithms respectively and k is the number of applied reliability improvement steps. From the expression (8.16) the value of t_o in which the surfaces intersect can be calculated as follows:

$$t_o = \frac{t_{Inf,i} \cdot TotRelGr_{Wh,i} - t_{Wh,i} \cdot TotRelGr_{Inf,i}}{TotRelGr_{Inf,i} - TotRelGr_{Wh,i}} \tag{8.17}$$

The presented equation has no solutions in case $TotRelGr_{Inf,i} = TotRelGr_{Wh,i}$. Thus, the only case in which the curves do not intersect is when the performance of both algorithms is matched. This occurs in cases when algorithms perform the same replacement steps but not necessarily in the same order. Since in the data-set-wise improvement scenario the same sequence of improvement steps is applied to all composite applications the following condition holds:

$$TotRelGr_{Inf,i} = TotRelGr_{Wh,i}, \quad \forall i \in [1, N] \tag{8.18}$$

where N is the total number of composite applications in the data set. Thus, the claim that the curves will not intersect regardless of the value of t_o can be extended to the expression for average reliability growth rate defined in expression (8.10). In addition, since $t_{Wh,i} < t_{Inf,i}$, $\forall i \in [1, N]$, the average growth rate achieved using *WP-WeightedPath* path

will be greater than then the one achieved using *WP-Influence* algorithm for any value of t_o. The projection presented in Figure 8.14b indicates that discontinuities appear in other reliability improvement steps as well, thus indicating the cases when the heuristic algorithms match the performance of *WP-Influence* algorithm for the given experimental setup. Expectedly, the *WP-WeakestPath* algorithm emerges as the best performing algorithm in the final reliability improvement steps.

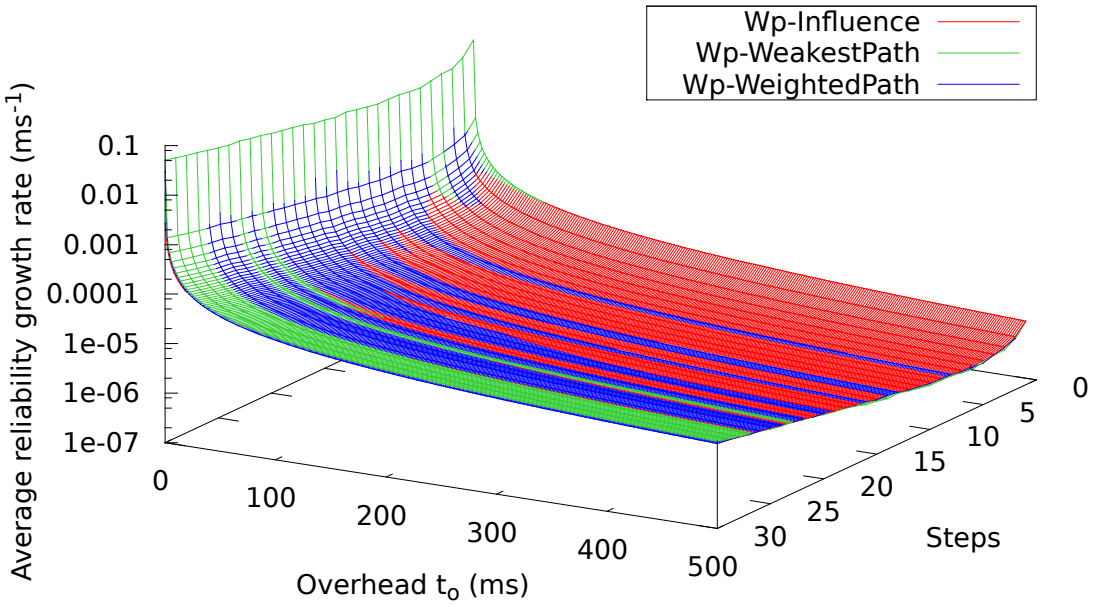
8.5 Yahoo Pipes Data Set

This section presents the results of the evaluation for the Yahoo Pipes real-world data set. Details on how Yahoo Pipes data set was generated from the mashup definitions are given in Section 8.5.1. Results of the computational performance experiments are presented in Section 8.5.2. Finally, Sections 8.5.3 and 8.5.4 present the results of evaluation for application-wise and data-set-wise component improvement scenarios respectively.

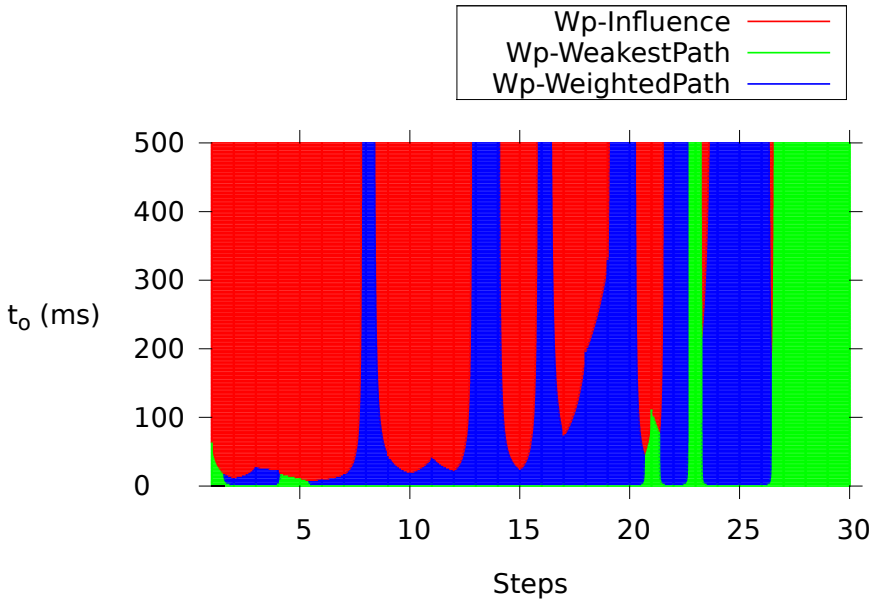
8.5.1 Data Set Generation

The input data set for Yahoo Pipes was obtained as part of the research findings preseted in [260]. The data set is prepared by collecting and translating the JSON mashup definitions into the reliability model defined in Section 5. A simplified JSON structure for Yahoo Pipes mashup definitions is presented in Figure 8.15. Generally, Yahoo Pipes mashup definitions contain a set of metadata describing a particular mashup. The metadata contains values such as: unique mashup identifier, name, description, list of domains, list of users and time of creation. The object layout defines how the graphical representations of components are placed on the canvas. The modules object defines which atomic building blocks are used to construct a particular mashup. Since the Yahoo Pipes framework supports nesting of mashups, i.e. a particular mashup can be a building component for another mashup, modules also define a set of complex building components. Finally, the wires object describes how particular components in modules are binded together into a workflow.

A high level overview of the reliability model generation process is given in Figure 8.16. In the first step (1), a list of used building components is extracted from modules of each mashup definition. The components are stored into atomic components and complex components tables. It should be noted that duplicates are not stored into tables: the key in atomic components is



(a) Average growth rate for t_o $0ms - 500ms$



(b) Best performing algorithm in relation to conducted improvement steps and overhead t_o

Figure 8.14: Impact of the overhead t_o on the $AvgGrRate$

the atomic component name, and the key in complex components is mashup id. The tables are maintained throughout the model generation process in order to detect incomplete mashup definitions, i.e. mashups constructed using complex components for which a definition does not

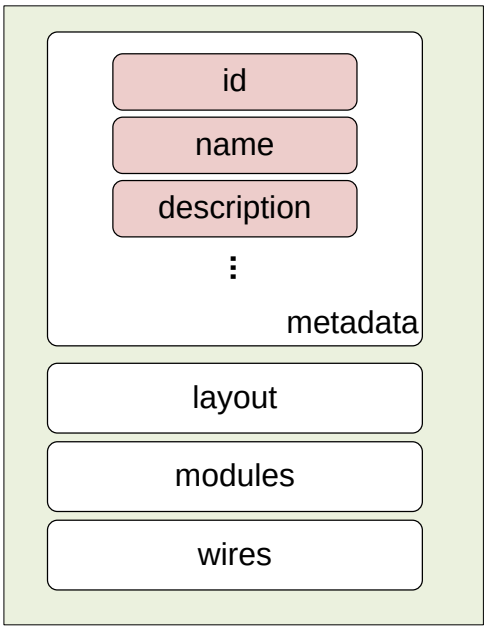


Figure 8.15: Simplified JSON structure for Yahoo Pipes Mashup Definition.

exist. Such entries are discarded form the data set.

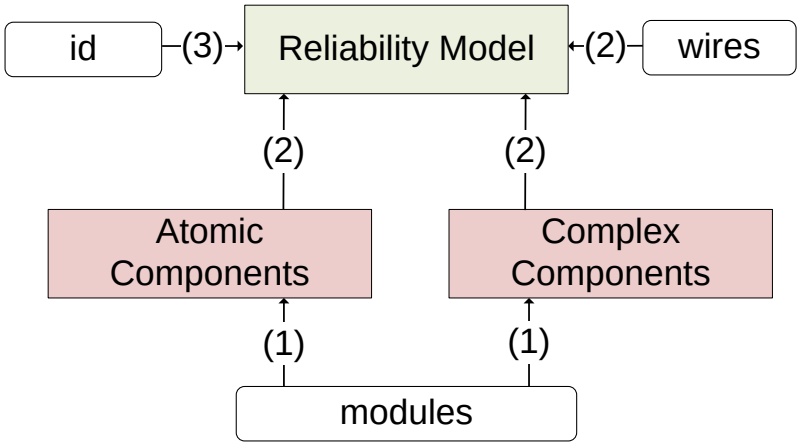


Figure 8.16: Conceptional overview of the data set generator.

In the second step (2), the mashup reliability models are constructed by combining data from modules and wires objects. Since the Yahoo Pipes mashup framework does not support complex workflow constructs, like branching and redundant execution paths, only composition, repetition and sequence nodes are used in the model. In general, the links defined by the wires object are translated into arcs between the composition node and atomic component node in the model. If more than a single link is defined for a certain atomic component, an additional repetition node is added and its repetition factor is set accordingly. Thus, most generated models have

maximum nesting depth of 3 nodes, while additional model layers are added by introduction of complex components. Finally, the generated model is stored into the data set using mashup id as the key (3).

Data Set Overview

The generated data set contains 8569 reliability models for the collected Yahoo Pipes mashup definitions. In addition, the data set contains 1391 (16.2%) mashups that are built using complex components, i.e. other mashups form the data set. To give an insight into the structure of mashups in the data set, histogram in Figure 8.17 shows how many mashups were built using a certain number of distinct atomic or complex components. The data shows that most mashups were built using 5 components (1354 mashups) following an exponential drop of the number of mashups with the increase of the number of building components. The presented results include the obligatory component *out*, present in each mashup. Thus, 8 entries in the data set are empty mashups containing only the output component. From the total number of mashups, 66.97% were built using 3-7 components. Apart from the data presented in Figure 8.17, the mashup definitions contained 2 mashups build using 24 components, one using 26 components and one using 47 components.

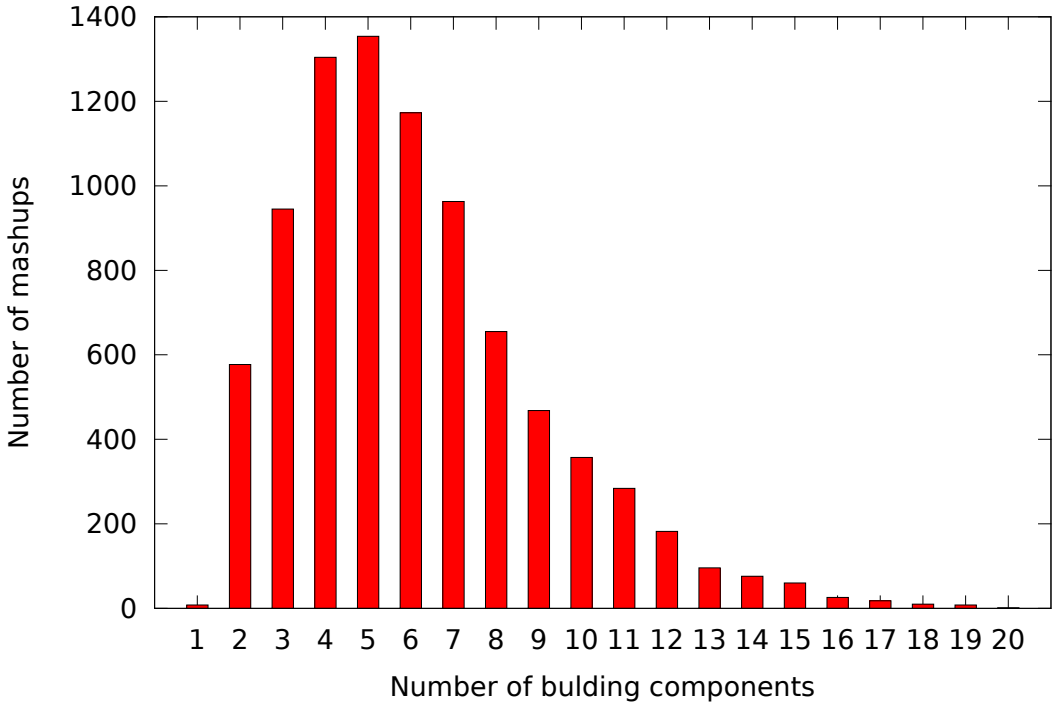


Figure 8.17: Number of building components used to construct mashups.

To further analyze the usage of the atomic building components, the total number of execu-

tions across the entire data set is presented in Figure 8.18. The presented data contains values for the 30 most executed components. The results indicate that the most executed component is *fetch* from the sources component category (on average 1.53 times per mashup) followed by the *textinput* component from the user inputs category (on average 1.02 times per mashup). Since it is an obligatory component, *out* is executed once per mashup. Other commonly used components include *urlbuilder* from the url category (on average 0.704 times per mashup) which is commonly used alongside the *fetch* component. The rest of the most commonly executed components are mostly used to manipulate string entries.

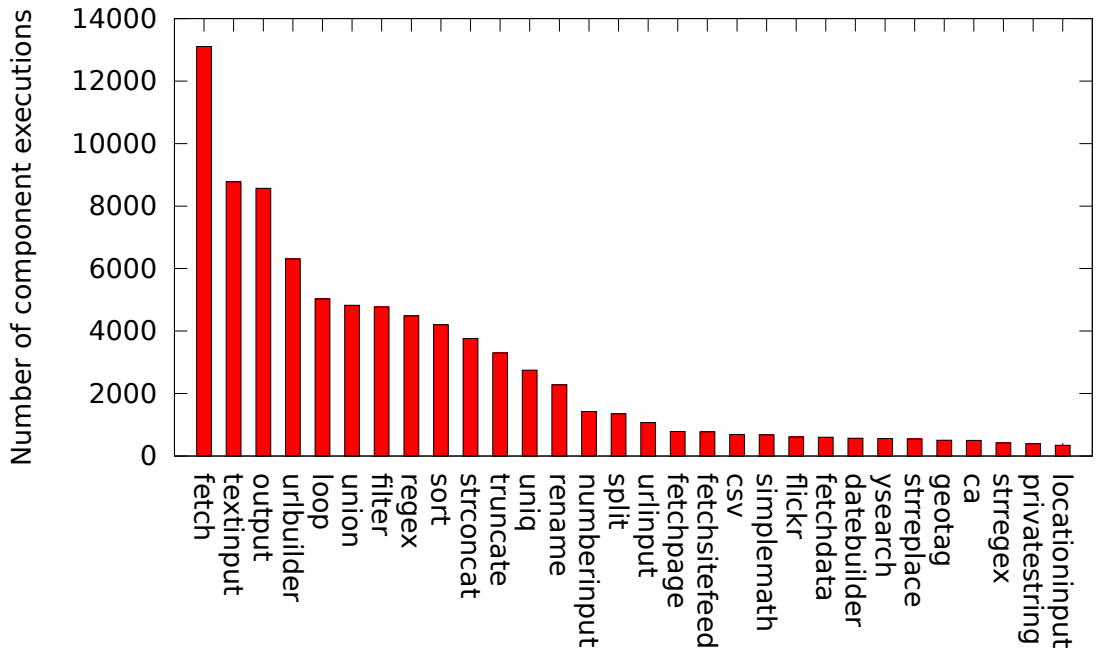


Figure 8.18: Number of component executions.

8.5.2 Performance Evaluation

Performance evaluation for the Yahoo Pipes data set is conducted in two experimental setups. In the first setup, average execution time (time required to calculate a single list of weak points), as defined in expression (8.5), is measured for all the composite applications in the data set. Execution time t_i for each composite application is an average value calculated on a set of 10 consecutive measurements. The results of the experiment for *WP-Influence* and heuristic algorithms are shown in Table 8.3. For the given set of Yahoo Pipes mashups, the results indicate that the *WP-Influence* algorithm has the highest average execution time, while the

Table 8.3: Average execution time for Yahoo Pipes data set

Algorithm	$AvgExec\ [ms]$	$\sigma_{AvgExec}\ [ms]$
WP-Influence	0.07159423585015753	0.10983113627126237
WP-WeakestPath	0.00242772995682118	0.00176407266635176
WP-WeightedPath	0.01299486929630072	0.05699716872007603

WP-WeightedPath algorithm has higher execution time than the *WP-WeakestPath* algorithm. Specifically, the *WP-Influence* algorithm has 5.51 and 29.49 times higher average execution time than the *WP-WeightedPath* and *WP-WeakestPath* algorithms respectively. Along with the results are standard deviations for the measured average execution time ($\sigma_{AvgExec}$). Expectedly, standard deviations are high due to significant differences between the composite applications in the data set, namely variations in the number of atomic building components.

In the second experimental setup, the applications are grouped by the number of atomic building components. The measurements are conducted for each group separately. It should be noted that the number of composite applications is not equal for all the groups. The number of components per application group is defined by histogram presented in Figure 8.17. The results of average execution time for *WP-Influence* and heuristic algorithms are shown in Figure 8.19. The presented results are limited to application groups with up to 10 atomic components. This is due to a low number of applications in the groups with a higher number of atomic components. As in the case of artificial data set, the average execution time grows linearly with the increase of the number of atomic building components for all the weak point recommendation algorithms. Expectedly, the execution time for *WP-Influence* algorithm grows more steeply than for the heuristic algorithms. Specifically, the average execution time for *WP-Influence* algorithm increases by a 46.47 and 31.3 times higher gradient then for the *WP-WeakestPath* and *WP-WeightedPath* algorithms respectively. Furthermore, the results confirm that the average execution time of *WP-WeightedPath* grows more steeply with the increase of atomic components than for the *WP-WeakestPath* algorithm. In conclusion, the performance evaluation results for Yahoo Pipes data set are in alignment with the results measured for the artificial data set.

8.5.3 Application-Wise Component Improvement Evaluation

The results for average reliability measure for both experimental setups are given by the graphs in Figure 8.20a and Figure 8.21a. The presented results are limited to the first 30 out of 55 possible component improvements. This is due to the fact that in the final reliability

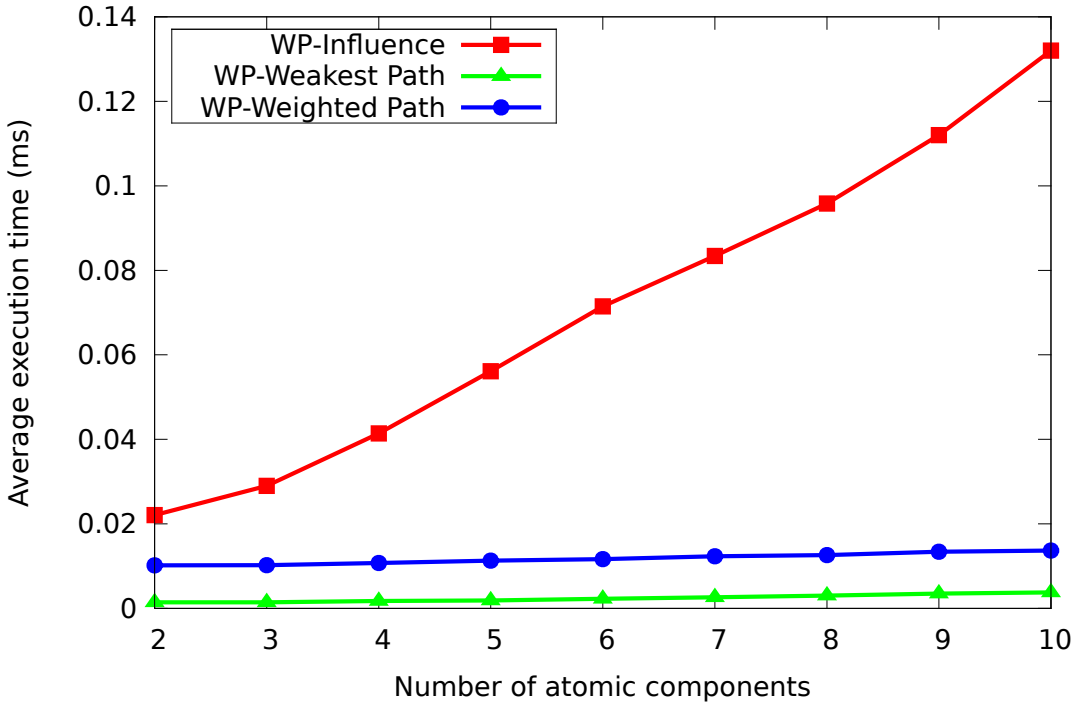


Figure 8.19: Execution times for Yahoo Pipes data set.

improvement steps, the achieved average increase in reliability is small due to the decrease of applications built using a certain number of atomic building components, as can be seen in Figure 8.17.

As in case of the artificial data set, presented results confirm that the *WP-Influence* algorithm is more accurate than the heuristic algorithms, as it achieves highest average reliability ($AvgRel$) at each composition improvement step. Furthermore, the results indicate that the *WP-WeightedPath* algorithm, which takes into account multiple influence paths, achieves better average results than the *WP-WeakestPath* algorithm for both the experiments on the given data set. Finally, it can be observed that *WP-Influence-R* and *WP-WeightedPath-R* algorithms achieve higher $AvgRel$ than *WP-Influence* and *WP-WeightedPath* respectively as they take into account the reliability of replacement components.

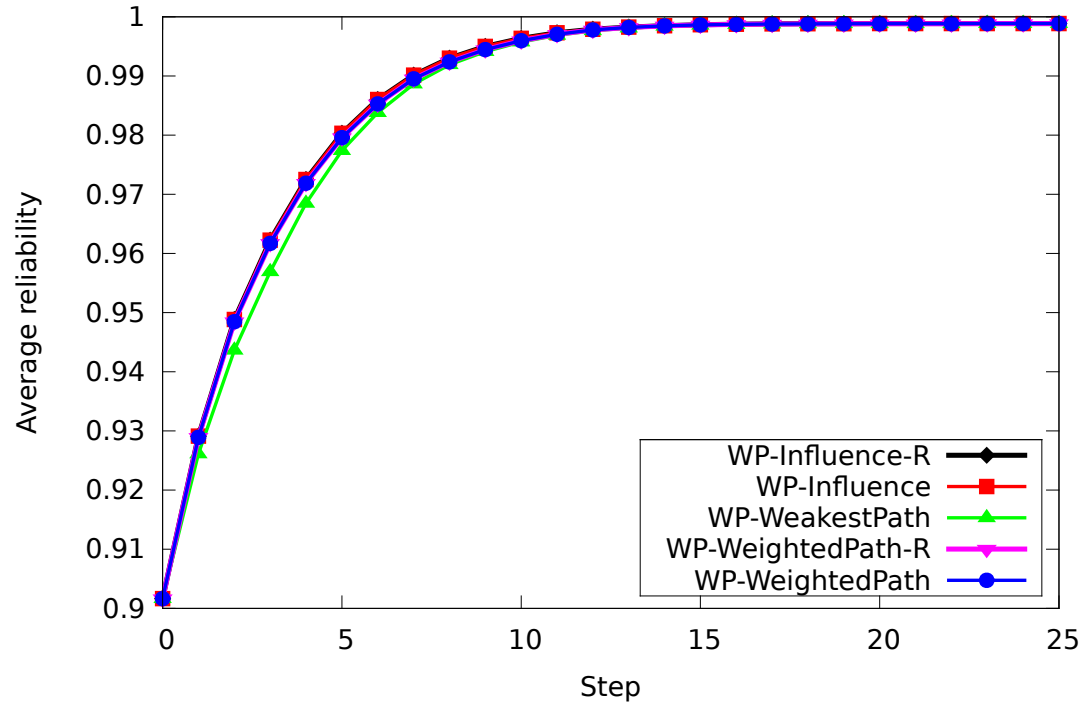
The presented results are conclusive with the results computed for the artificial data set and they confirm the differences in the accuracy of the weak point recommendation algorithms. However, the performance of the algorithms is less differentiated than in the case of the artificial data set. This is mostly due to the fact that Yahoo Pipes do not support complex workflows, namely workflow branching and redundant execution paths. In such conditions, the presented heuristic weak point detection algorithms better match the *WP-Influence* algorithm by their performance.

Table 8.4: Difference in $AvgRel$ between recommendation algorithms and *WP-Influence-R*: Yahoo Pipes application-wise improvement

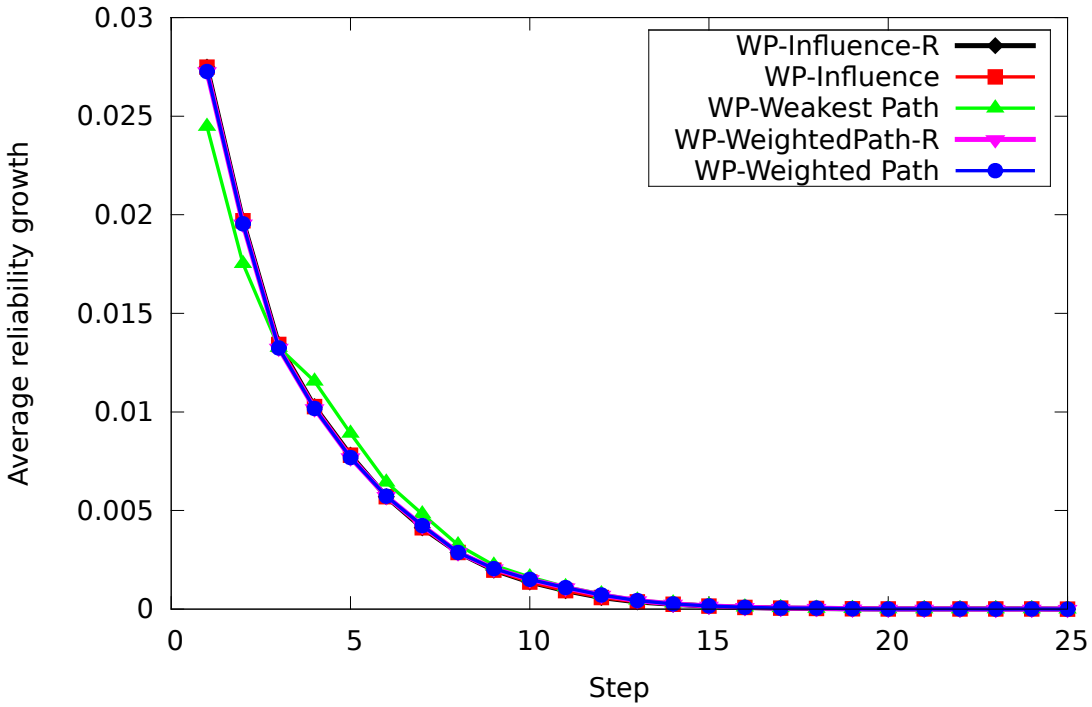
Algorithm	Measure	0.99, inc. 0.0099	0.7 - 0.9, inc. 20%
WP-Influence	*MAE*	0.0	0.000044691663694
	RMSE	0.0	0.000098576641700
	MAPE	0.0	0.000107624361092
	RMSPE	0.0	0.000259839327776
WP-WeakestPath	*MAE*	0.000939707159147	0.001614467357183
	RMSE	0.001818329901584	0.003094995611902
	MAPE	0.000971532517476	0.003613212667699
	RMSPE	0.001891845798050	0.007535013845068
WP-WeightedPath-R	*MAE*	0.000211260629097	0.000590927730020
	RMSE	0.000335563552720	0.001224383925877
	MAPE	0.000215201801221	0.001336575543312
	RMSPE	0.000341833937596	0.002929653782866
WP-WeightedPath	*MAE*	0.000211260629097	0.000650487367530
	RMSE	0.000335563552720	0.001361560540816
	MAPE	0.000215201801221	0.001476999559615
	RMSPE	0.000341833937596	0.003273562565117

To give a more precise insight, standard error measures are calculated to evaluate accuracy of recommendation algorithms against the optimal solution provided by the *WP-WeightedPath-R*. The results calculated for the the whole improvement interval $[1 \dots 55]$, are presented in Table 8.4. It is confirmed that *WP-Influence* has lower error measure values than the rest of the heuristic in both experiments. It should be noted that in *experiment A*, *WP-Influence* matches the *WP-Influence-R* in accuracy. Furthermore, *WP-WeightedPath* path is more accurate than *WP-WeakestPath*, e.g. it achieves up to 5.41 times lower *RMSE* in *experiment A*. Finally, it is confirmed that improvements incorporated into *WP-WeightedPath-R* yield higher accuracy, e.g. *RMSE* in *experiment B* is 11.21% lower than for *WP-WeightedPath*. It sholud be noted that since in *experiment A* all replacement components have the same reliability, *WP-WeightedPath-R* performs the same as *WP-WeightedPath*. Finally, the results for percentage error measures indicate that all the algorithms achieve higher accuracy in *experiment A*. As stated previously, in *experiment A*, *WP-Influence* matches the *WP-Influence-R* in accuracy as its *RMSPE* value is 0. Similar results hold for the heuristic algorithms as *WP-WeakestPath*, *WP-WeightedPath* and *WP-WeightedPath-R* algorithms achieve respectively 3.92, 9.57, and 8.57 times lower *RMSPE* in *experiment A* when compared to *experiment B*.

In order give a better insight into accuracy for the initial improvement steps and to analyze the trend of growth, the average reliability growth ($RelGr$) for both experiments is presented in



(a) Average reliability



(b) Average reliability growth

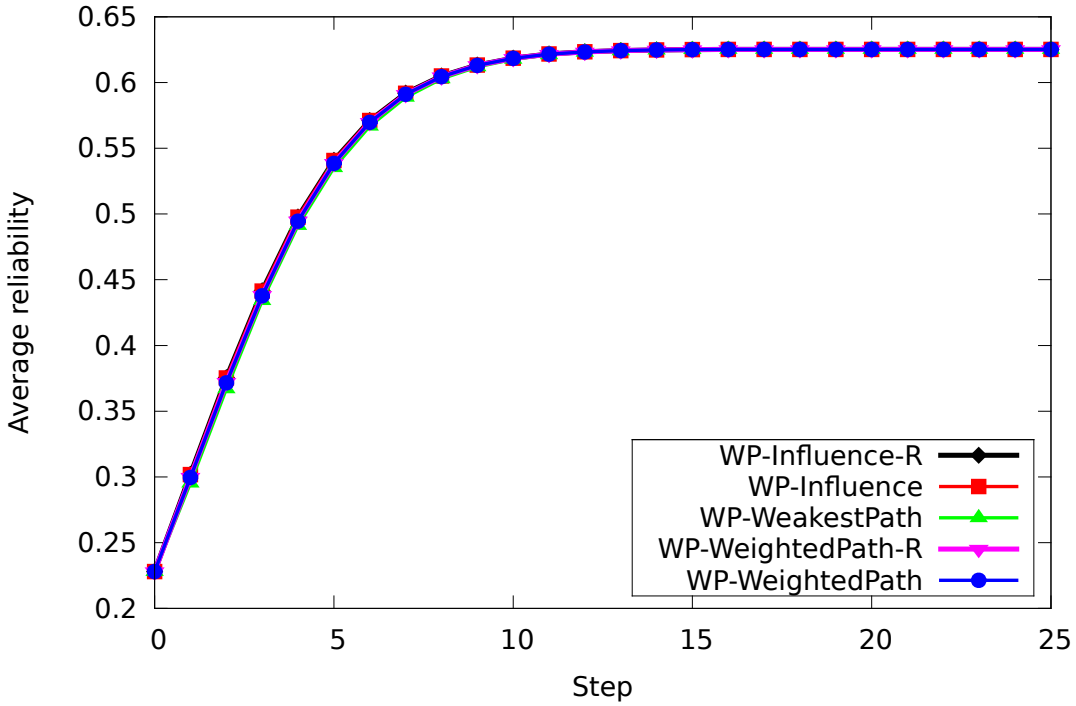
Figure 8.20: Reliability growth results: initial 0.99, increase 0.0099.

Fig. 8.20b (*experiment A*) and Fig. 8.21b (*experiment B*).As in previous experimental setups, a better performing of two weak point recommendation algorithms should achieve higher initial average reliability growth and a lower average reliability growth at the final improvement

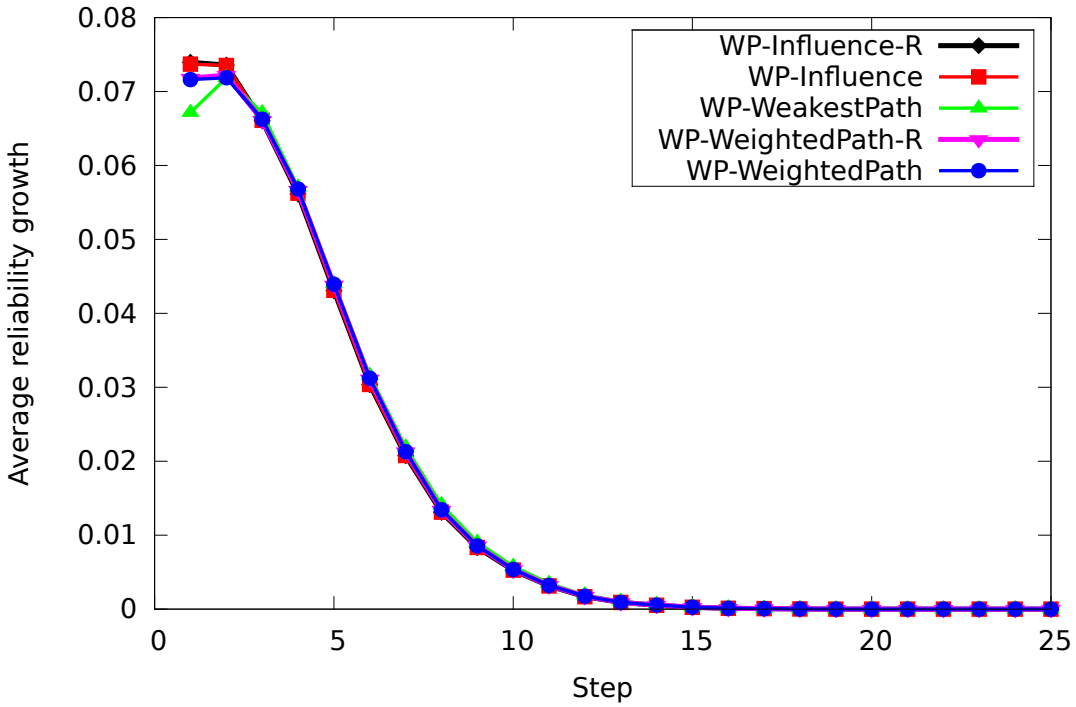
steps. The results presented for both experiments confirm that the *WP-Influence-R* algorithm achieves highest average reliability growth during the initial improvement steps. It can also be concluded that the *WP-WeightedPath* and *WP-WeightedPath-R* algorithms outperform the *WP-WeakestPath* algorithm in both experiments.

Figure 8.22 shows the average number of composition improvement steps ($AvgSt$) required to reach a certain reliability threshold—denoted on the x-axis as the total reliability growth. As in the case of the artificial data set, the results for both experiments in Figure 8.22a and Figure 8.22b indicate that the *WP-Influence* algorithm achieves greatest total reliability growth while on average requiring least reliability improvement steps. However, the difference between the *WP-Influence* algorithm and the heuristic algorithms in both experimental setups is expectedly lower than in the application-wise improvement scenario. In addition, the results show that the *WP-WeightedPath* algorithm has better performance for initial reliability improvement steps, closer to the *WP-Influence* algorithm. Specifically, in *experiment A* (Figure 8.22a), the *WP-Influence* algorithm achieves the defined reliability threshold 0.25 while performing 18.41% and 6.01% less improvement steps then algorithms *WP-WeakestPath* and *WP-WeightedPath* respectively. Similarly, in *experiment B* (Figure 8.22b), the *WP-Influence* algorithm reaches the reliability threshold 0.2 in 4.26% and 1.99% less improvement steps then *WP-WeakestPath* and *WP-WeightedPath* respectively. Furthermore, the results confirm that *WP-Influence-R* and *WP-WeightedPath-R* perform more accurately than their unmodified versions. For instance, in *experiment B* *WP-Influence-R* and *WP-WeightedPath-R* achieve reliability threshold 0.2 in respectively 0.1% and 1.65% less improvement steps than *WP-Influence* and *WP-WeightedPath*.

In order to further evaluate the impact of the additional overhead t_o on the applicability of the weak point recommendation algorithms, results of average growth rate for $t_o \in [0ms, 5ms]$ are shown in Figure 8.23a.In experimental setup the initial reliability of all components is set equally to 0.9 and the final reliability is set to 0.99 (10% increase). The average growth rate, shown on the z-axis in logarithmic scale, is calculated for each improvement step, as denoted on the y-axis. Since for the simple Yahoo Pipes workflows the heuristic algorithms closely match the performance of the *WP-Influence* algorithm, the *WP-WeakestPath* algorithm becomes the best performing algorithm after performing a small number of improvement steps. Therefore, in order to make the results easily readable, the number of improvement steps is reduced to the interval $[1, \dots 6]$. The results indicate that for the given experimental setup, $t_o > 1ms$ and the number of replacements steps less than 4, the *WP-WeightedPath* outperforms the *WP-*



(a) Average reliability

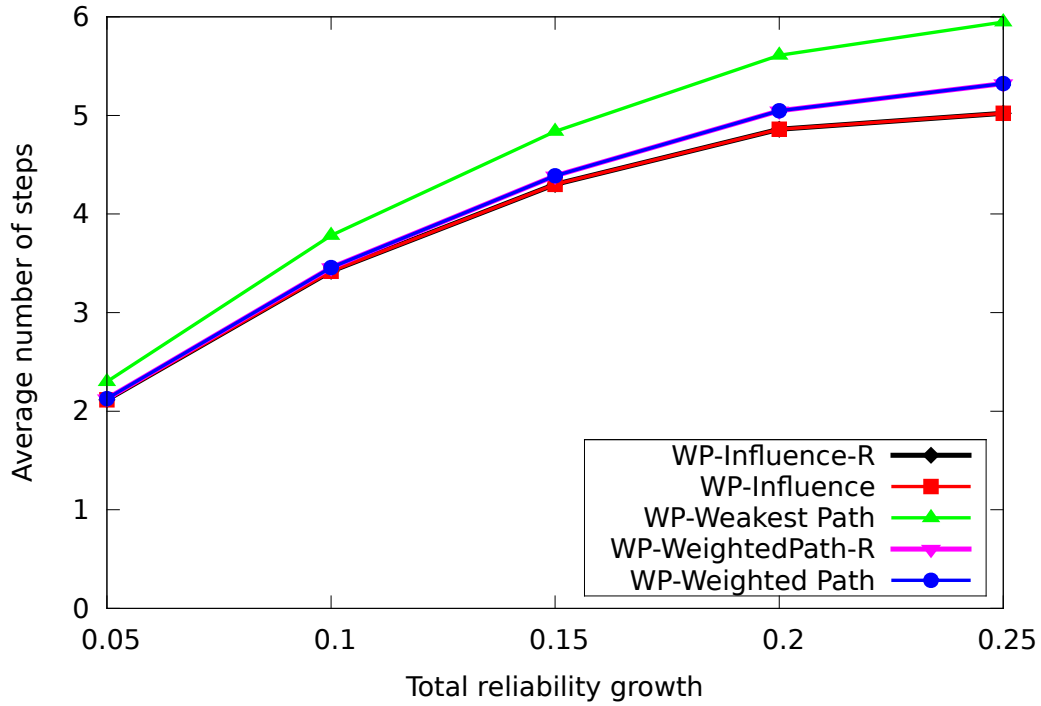


(b) Average reliability growth

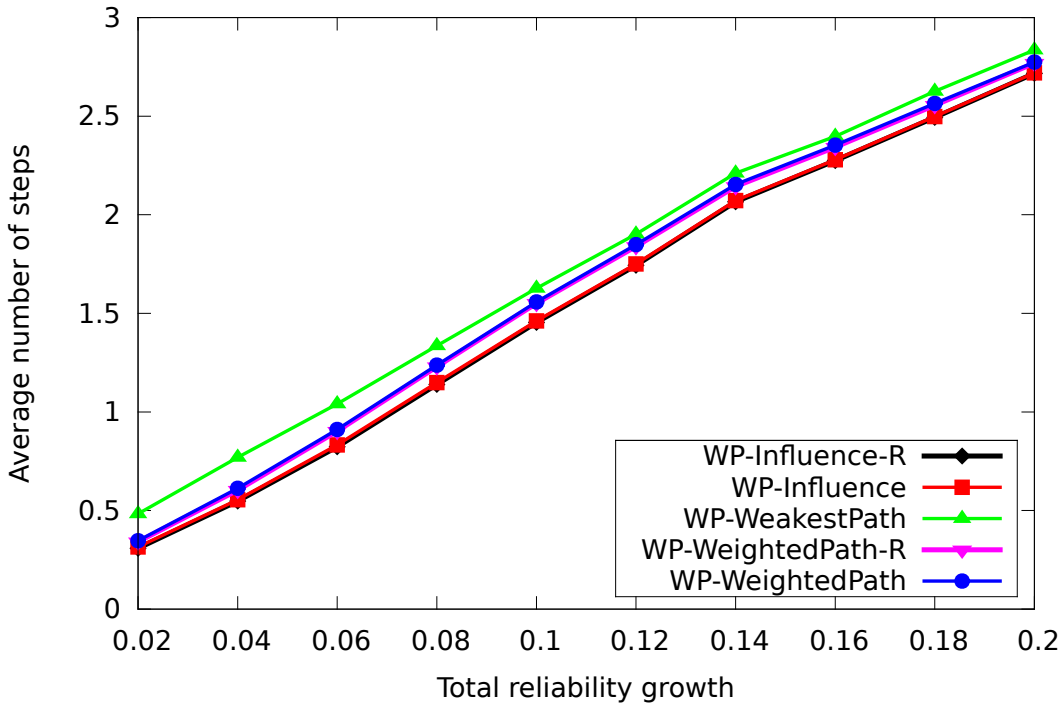
Figure 8.21: Reliability growth results: initial 0.7 - 0.9, increase 20%.

WeakestPath algorithm. Since the performance of the algorithms is closely matched, the *WP-Influence* does not outperform heuristic algorithms for the given t_o interval.

To give a wider overview of the impact of the additional overhead t_o on the applicability of



(a) Average number of steps: initial 0.99, increase 0.0099



(b) Average number of steps: initial 0.7 - 0.9, increase 20%

Figure 8.22: Average number of steps to reach a reliability threshold.

the weak point recommendation algorithms, results of average growth rate for a wider overhead interval $t_o \in [0ms, 500ms]$ are shown in Figure 8.23b. The presented plot is a projection of graph in Figure 8.10a onto the x-y plane. As in the case of the artificial data set, there are

no discontinuities in the presented graph indicating clear separation of conditions in which a particular algorithm performs best.

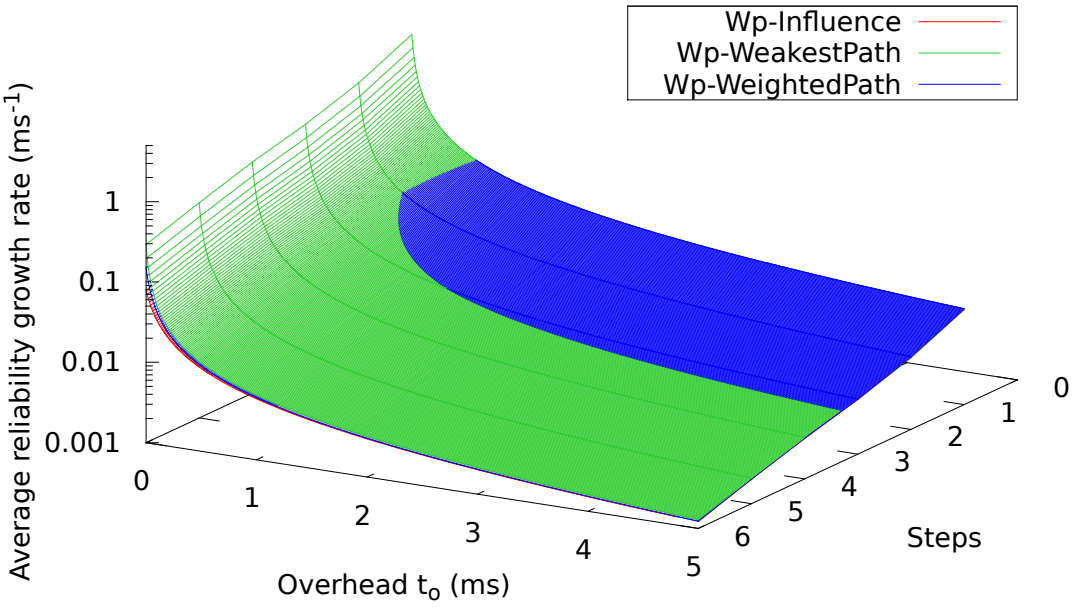
It can be concluded that for the presented experimental setup, the *WP-Influence* and *WP-WeightedPath* algorithms achieve highest *AvgGrRate* for a sufficiently large t_o. As the number of improvement steps grows, the *WP-WeakestPath* rapidly becomes the best performing algorithm. This is due to the properties of the data set. Firstly, for the latter improvement steps, the performance of the the *WP-WeakestPath* and *WP-WeightedPath* algorithms closely matches the performance of *WP-Influence* algorithm, as can be seen in Figures 8.20 and 8.21. Secondly, the number of the applications constructed using a certain number of atomic components drops exponentially after 4 atomic components, as presented in Figure 8.17.

8.5.4 Data-Set-Wise Component Improvement Evaluation

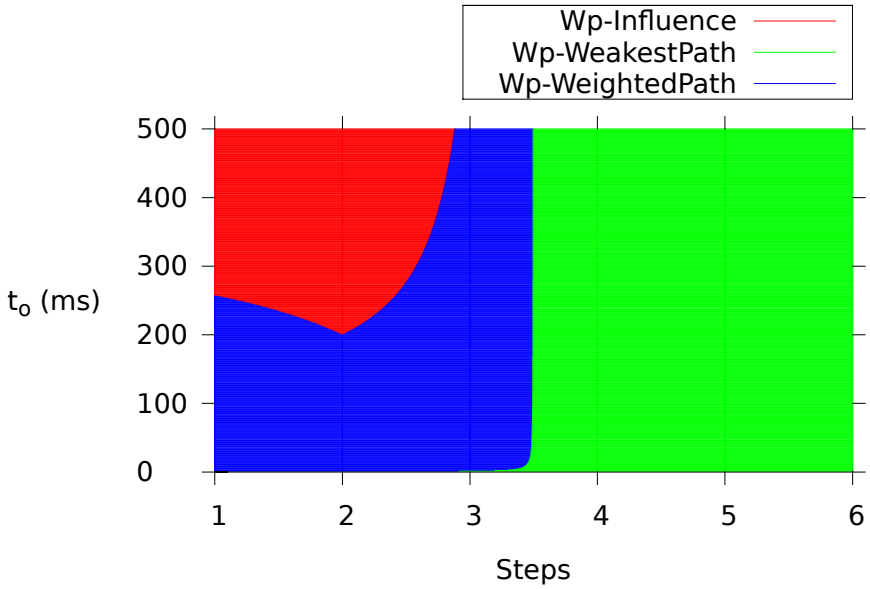
The results for average reliability measure ($AvgRel$) for both Yahoo Pipes experimental setups of data-set-wise component improvement are given by the graphs in Figure 8.24a and Figure 8.25a. As in the case of the artificial data set (Section 8.4.4), the difference between the recommendation algorithms is much less pronounced than for the application-wise improvement experiments. Also, a more graduate increase in reliability is achieved than in application-wise improvement experiments described in Section 8.5.3. These results are in accordance with the results computed for the artificial data set.

The presented graphs indicate that the *WP-Influence-R* algorithm expectedly achieves higher average reliability ($AvgRel$) than the *WP-Influence*, *WP-WeakestPath*, and *WP-WeightedPath* algorithms for the given data set and experimental setup. However, this conclusion does not hold for the entire improvement interval since the heuristic algorithms can at certain improvement steps perform as good as the *WP-Influence-R* algorithm due to aggregated solution space. In addition, although it is less apparent from the graphs, the *WP-WeightedPath* algorithm displays better average performance than the *WP-WeakestPath* algorithm. However, it should be noted that at certain improvement steps the *WP-WeakestPath* algorithm can outperform the *WP-WeightedPath* algorithm, while both heuristic algorithms can at best match the performance of the *WP-Influence-R* algorithm. Moreover, in this evaluation scenario, heuristic algorithms outperform the *WP-Influence* algorithm for certain experimental setups. This effect was not visible in previously described experiments.

To further analyze distinctions in accuracy between the *WP-Influence-R* and the rest of the



(a) Average growth rate for t_o $0ms - 5ms$



(b) Best performing algorithm in relation to conducted replacement steps and overhead t_o

Figure 8.23: Impact of the overhead t_o on the $AvgGrRate$

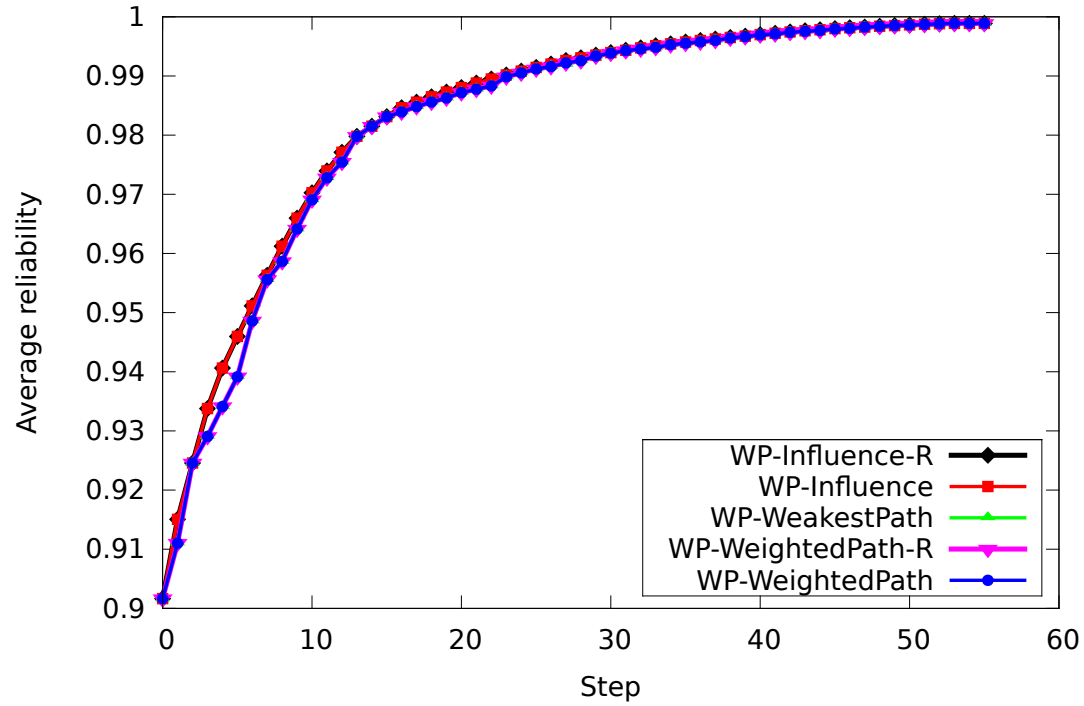
recommendation algorithms, error measures for average reliability ($AvgRel$) on the whole improvement interval are presented in Table 8.5. In *experiment A*, where the reliability of replacement services is close to 1, *WP-Influence* algorithm is highly accurate. In fact, it matches the

Table 8.5: Difference in $AvgRel$ between recommendation algorithms and *WP-Influence-R*: Yahoo Pipes DS-wise improvement

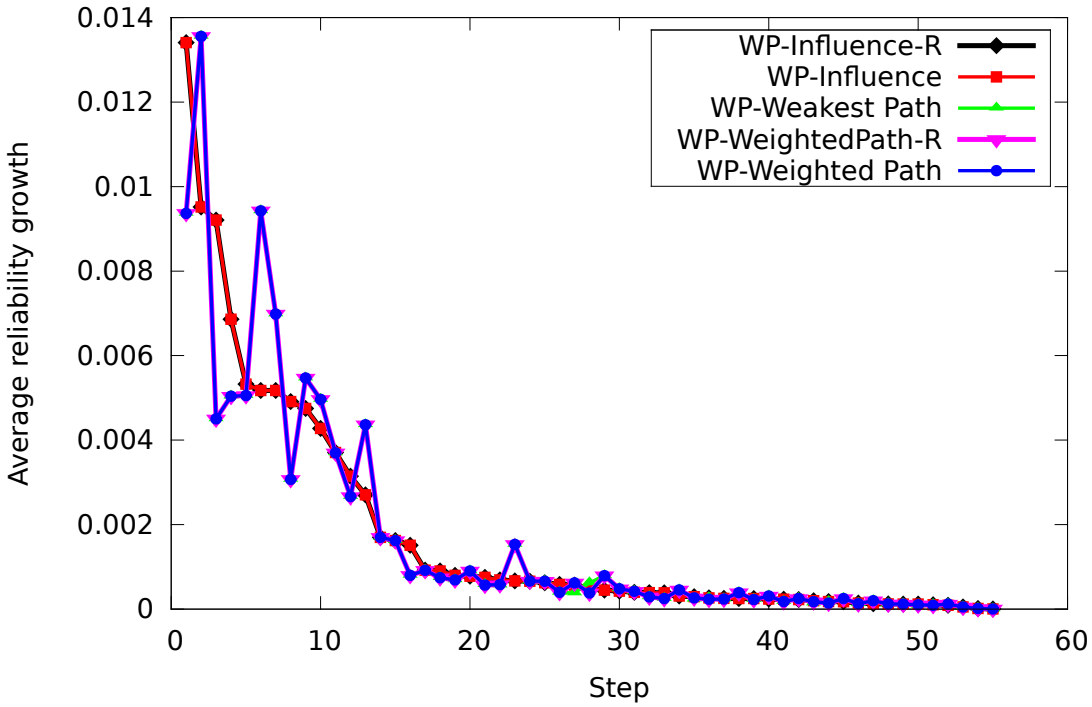
Algorithm	Measure	0.99, inc. 0.0099	0.7 - 0.9, inc. 20%
WP-Influence	*MAE*	0.0	0.002723361530364
	RMSE	0.0	0.003611978665526
	MAPE	0.0	0.005941509852968
	RMSPE	0.0	0.008380357062730
WP-WeakestPath	*MAE*	0.001501165784937	0.001685861609996
	RMSE	0.002290580703696	0.001872527226902
	MAPE	0.001569833508413	0.003241068988812
	RMSPE	0.002427607087499	0.003652803365054
WP-WeightedPath-R	*MAE*	0.001491084021603	0.001683250178144
	RMSE	0.002288827868637	0.001901861773540
	MAPE	0.001559689322406	0.003228963430391
	RMSPE	0.002425931070373	0.003701181695178
WP-WeightedPath	*MAE*	0.001491084021603	0.001683250178144
	RMSE	0.002288827868637	0.001901861773540
	MAPE	0.001559689322406	0.003228963430391
	RMSPE	0.002425931070373	0.003701181695178

WP-Influence-R algorithm in accuracy, since its error measures are equal to 0. On the other hand, in *experiment B* where there is a greater difference in replacement component reliabiltiy, the *WP-Influence* becomes less stable. This is due to errors in reliability sensitivity analysis that get aggregated when computing expression (8.3). In fact, *WP-Influence* prooves to be less accurate than *WP-WeakestPath* and *WP-WeightedPath* as it achieves respectively 1.92 and 1.89 times higher *RMSE*. Furthermore, *WP-WeakestPath* and *WP-WeightedPath* are very closely matched. Specifically, the *WP-WeightedPath* has lower *MAE* indicating that it on average provides more accurate recomendations. However, *WP-WeakestPath* achieves lower *RMSE* indicating that *WP-WeightedPath* can provide recommendations with significantly lower accuracy. The effect of these recommendations is more prononced since *RMSE* is a quadratic measure. Furthermore, unlike in the previously described results, the *WP-WeightedPath* as accurate as *WP-WeightedPath-R* in both experiments. Results indicate that since workflow constructs are less complex then in the artificial data set, the differences in atomic component reliability in *experiment B* are not relevant enough to impact the quality of recommendations. Finally, by observing the percentage error measures, it can be concluded that all algorithms are more accurate in *experiment A* where reliability values of atomic components are higher. These results is in accordance with the prevously described experiments.

To give a better insight into behavior of the recommendation algorithms, the average relia-



(a) Average reliability



(b) Average reliability growth

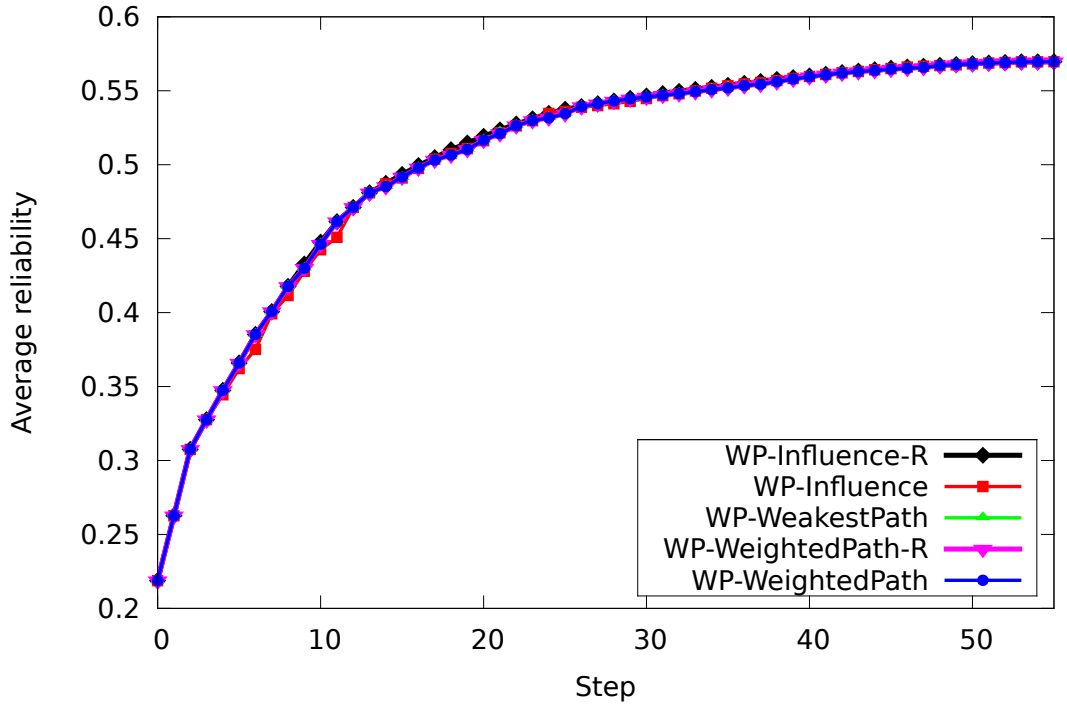
Figure 8.24: Reliability growth results: initial 0.99, increase 0.0099.

bility growth is presented for both experiments in Figure 8.24b and Figure 8.25b. Unlike in the application-wise improvement scenario, the reliability growth curves intersect *WP-Influence-R* graph more than once. In fact, the heuristic algorithms oscillate around the best solution, pro-

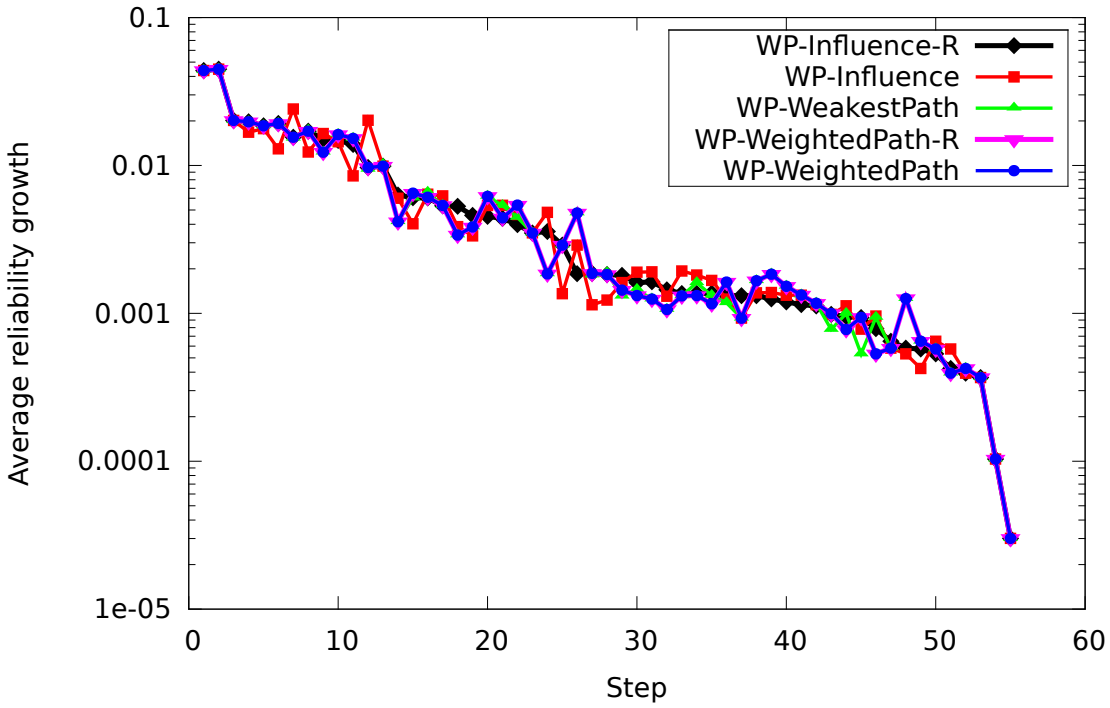
vided by the *WP-Influence-R* algorithm. This behavior indicates that the heuristic algorithms manage to correct the error in component recommendation and maintain the quality of solution close to the *WP-Influence-R* algorithm. The results for both experimental setups also indicate that the *WP-WeightedPath* displays a more stable behavior than the *WP-WeakestPath* algorithm since it deviates slightly less from the curve of the more accurate *WP-Influence-R* algorithm. This behavior is particularly visible in Figure 8.25b, for the improvement interval $[40, \ldots, 50]$. Furthermore, the results show how *WP-Influence* matches the accuracy of *WP-Influence-R* in *experiment A*, while it becomes less stable in *experiment B*.

Figure 8.26 shows the average number of composition improvement steps required to reach a certain reliability threshold—denoted on the x-axis as the total reliability growth. As for the artificial data set, in data-set-wise improvement scenario lower values of $AvgSt$ do not necesserily imply higher accuracy. They rather indicate that the recommened weak points are accurate for the most of the applications in the data set. Thus, more accurate recommendation algorithms will properly identify application subset that has the greatest impact on the overall reliability. This effect is visible in results of both experiments. For instance, in *experiment A* (Figure 8.26a), the most accurate *WP-Influence-R* requires 5.87% more improvement steps to reach the reliability threshold 0.02 than the *WP-WeakestPath*. On the other hand, for the latter improvement steps, the *WP-Influence-R* provides recommendation solutions that are highly accurate for the most of the data set. Secifically, it requires 7.83 less improvement steps to reach the reliablity threshold 0.2 than the *WP-WeakestPath*. A similar effect if visible in the results of *experiment B* (Figure 8.26b). Finally, it can be observed that in *experiment A* there is a drop in $AvgSt$ for reliability threshold 0.14, an effect not visible in previously presented results. This drop occures when the number of applications that can reach a certain threshold drops, while other applications that can reach the threshold do not cause a significant increase in the number of required improvement steps.

In order to evaluate the impact of the additional overhead t_o on the applicability of the weak point recommendation algorithms, results of average growth rate for $t_o \in [0ms, 5ms]$ are shown in Figure 8.27a.In experimental setup the initial reliability of all components is set equally to 0.9 and the final reliability is set to 0.99 (10% increase). The average growth rate, shown on the z-axis in logarithmic scale, is calculated for each of the 54 possible improvement steps, out of which 30 are displayed to make the results more readable. The results indicate low dependence on the overhead t_o as the algorithms are closely matched in their performance. As in case of



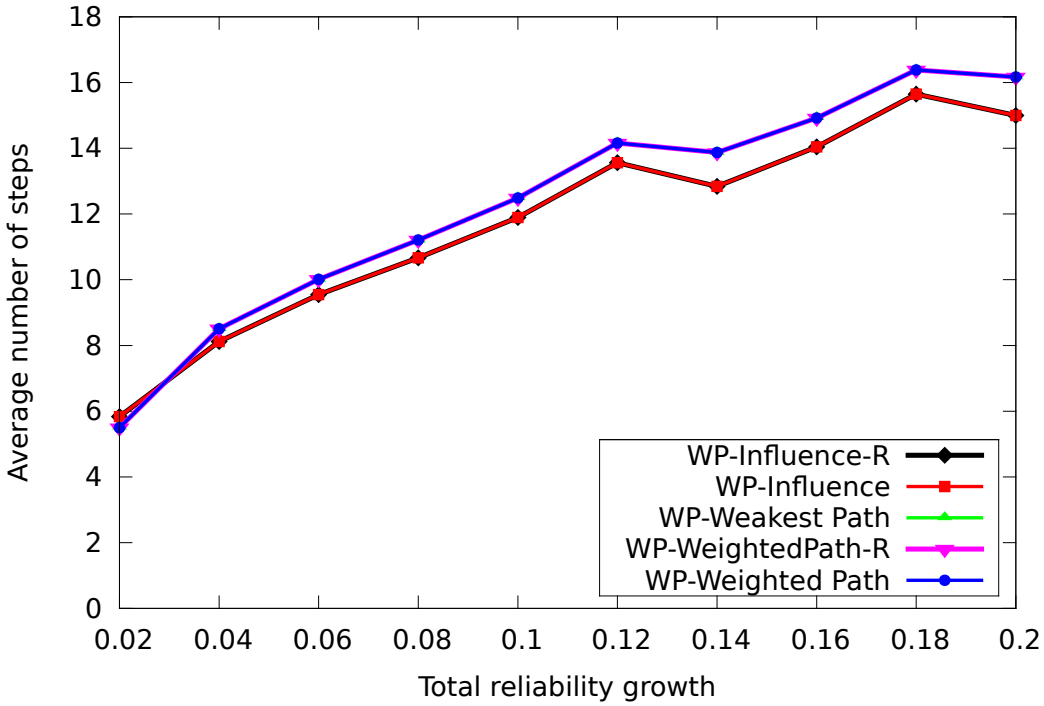
(a) Average reliability



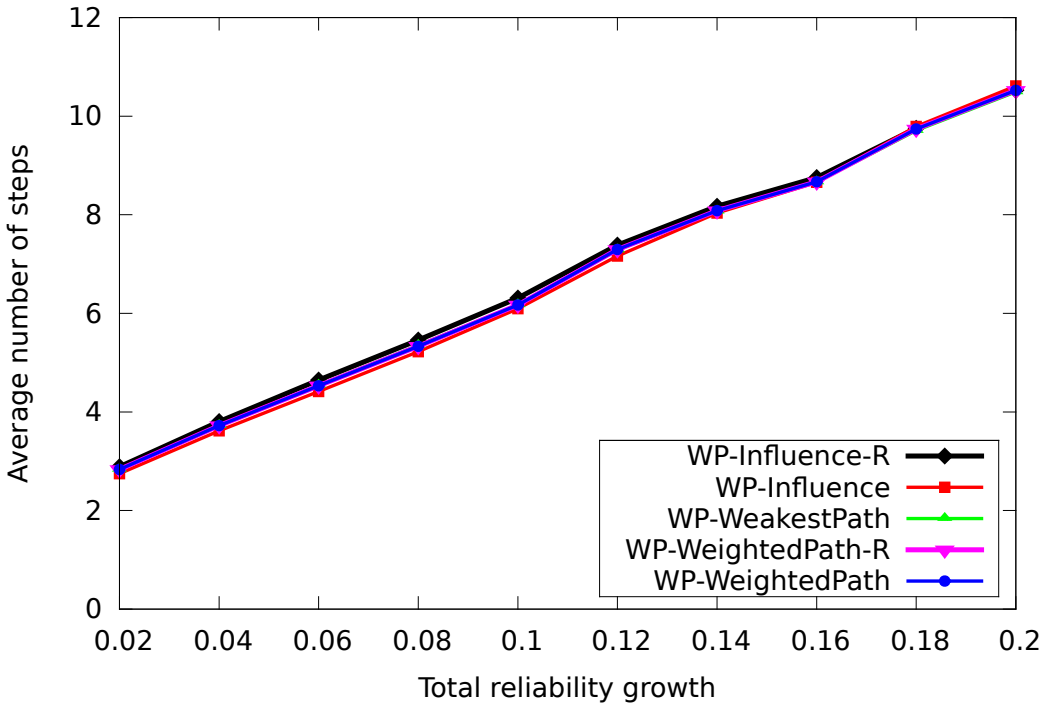
(b) Average reliability growth

Figure 8.25: Reliability growth results: initial 0.7 - 0.9, increase 20%.

the artificial data set, the results are less conclusive regarding the algorithms' applicability than in the application-wise scenario. This is due to the fact that the surfaces intersect on multiple occasions, causing discontinuities regardless of the increase in t_o.



(a) Average number of steps: initial 0.99, increase 0.0099



(b) Average number of steps: initial 0.7 - 0.9, increase 20%

Figure 8.26: Average number of steps to reach a reliability threshold.

This behaviour can be better observed in Figure 8.27b for the entire improvement interval. The graph shows a projection of surface in Figure 8.27a onto the x-y plane. For example, a discontinuity independent of t_o between *WP-Influence* and *Wp-WeakestPath* can be observed in

the 10^{th} improvement step. This means the *WP-WeakestPath* matches the performance of the *WP-Influence* algorithms in the 10^{th} improvement step, since the curves will not intersect for any value of t_o. This claim is proven by expressions (8.16 - 8.18). The results confirm that the heuristic graphs manage to match the performance of the *WP-Influence* algorithm during the improvement process making the decision on the algorithm applicability less conclusive.

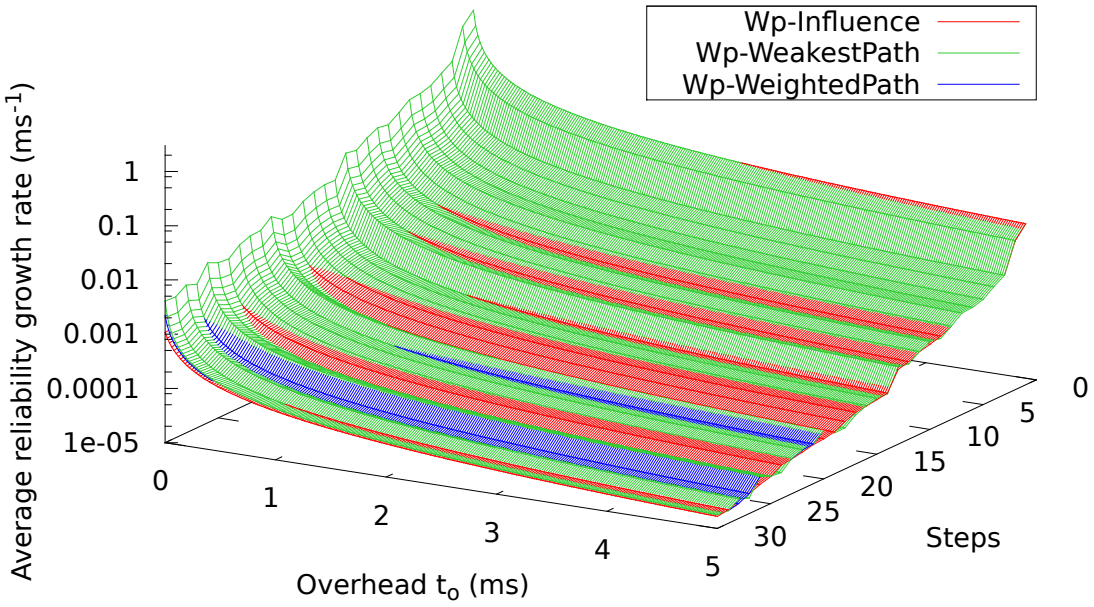
8.6 Impact of Atomic Component Dependency

To evaluate the impact of atomic component dependency on weak point recommendation accuracy, the artificial data set models generated in *experiment A* were extended with additional dependencies. Specifically the models were extendend with a number of unidirectional dependencies, as described in Section 5.2. To form the dependancies, component pairs were randomly chosen with an uniform probability distribution. For each component pair, component S_i was made dependent on component S_j so that by improving S_j, reliability of S_i would increase 10%.

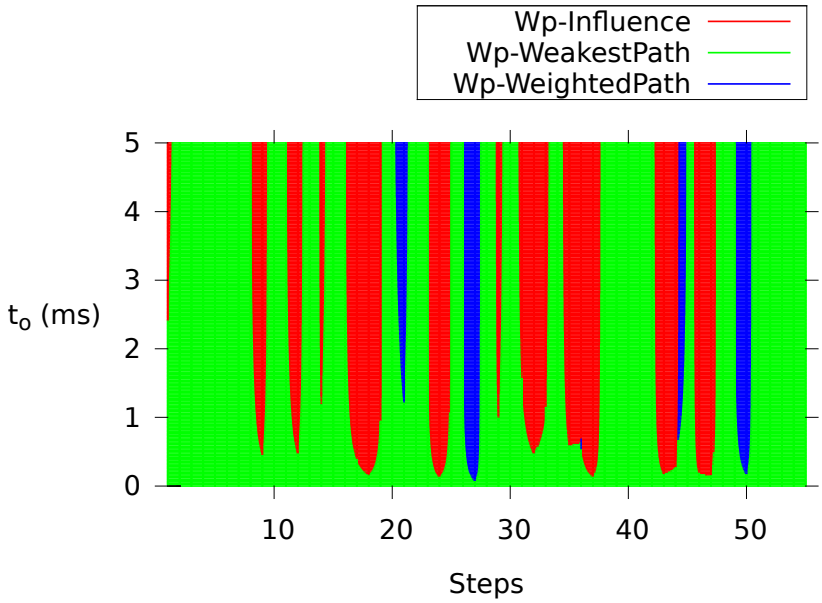
Two experiments were made for each recommendation algorithm. In the first experiment, weak points were recommended using the model that takes into account dependencies (+*D*). On the other hand, in the second experiment, recommendations were performed using the original models that do not take into account dependences (-*D*). In order to evaluate how using less accurate models (-*D*) impacts the recommendation algorithms' accuracy, reliability growth for weak point lists generated in the second experiment was calculated using the extended models (+*D*). In other words, weak point lists derived using less accurate models (-*D*) were evaluated on more accurated reliability models (+*D*).

The experiments were performed for two different setups. In the first setup (Figure 8.28a), 5 dependent component pairs were randomly introduced into the workflow, while in the second setup Figure (8.28b), 10 dependent component pairs were introduced. The presented results clearly indicate that the achieved $AvgRel$ is lower for the -*D* experiments. Thus, it can be concluded that by ignoring atomic component dependence, the recommendation accuracy decreases.

Furthermore, the results in Figure 8.28 show that *WP-Influence* is more susceptible to inaccuracies in (-*D*) models then the heuristic algorithms, as it strongly depends on accurately estimating the influence of each individual atomic service. We further supprt this claim by



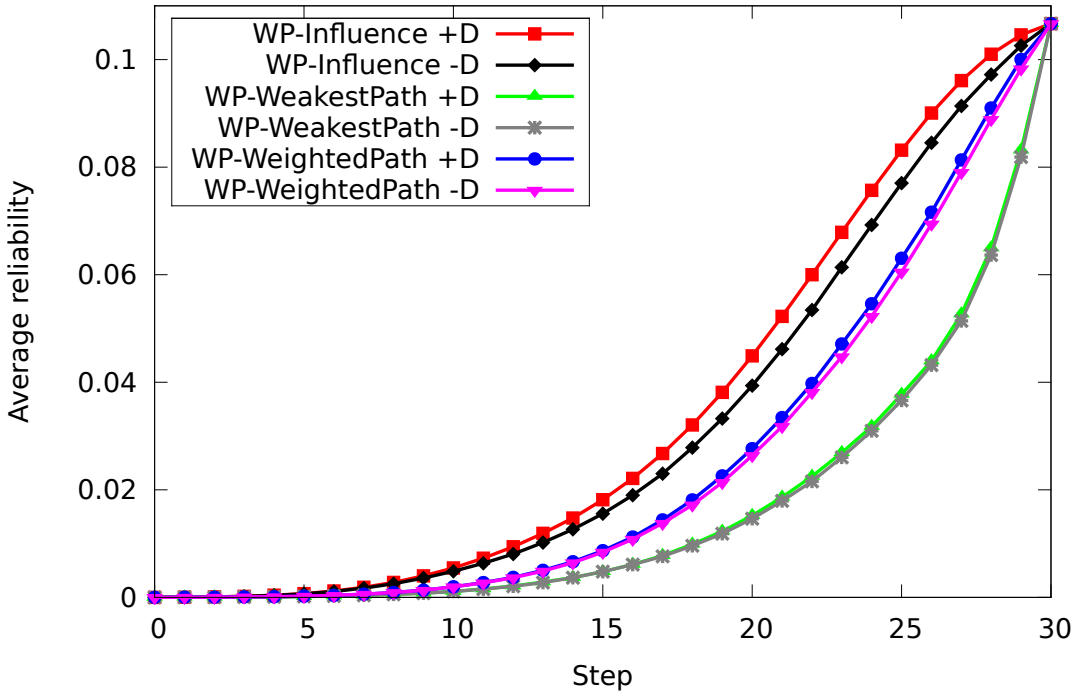
(a) Average growth rate for t_o $0ms - 5ms$



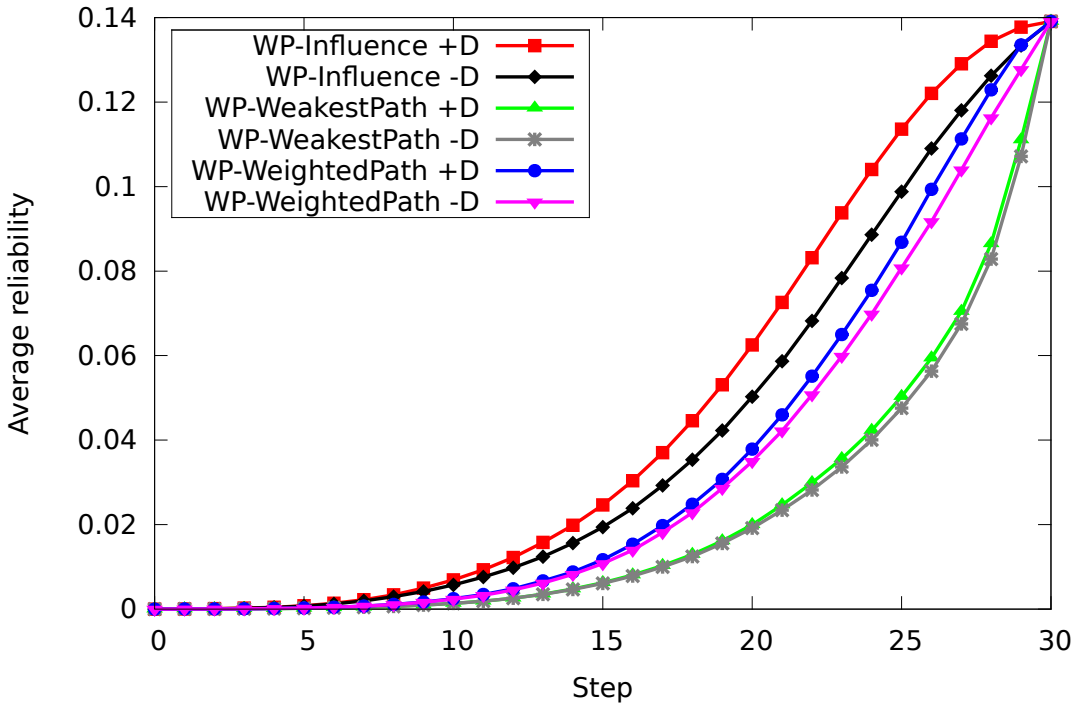
(b) Best performing algorithm in relation to conducted replacement steps and overhead t_o

Figure 8.27: Impact of the overhead t_o on the $AvgGrRate$

computing the *RMSPE* error measure to compare the difference of *WP Influence* algorithm in experiment (+*D*) and all the recommendation algorithms in experiments (-*D*). Thus, $x_{2,i}$ in expression 8.15 is equal to $AvgRel$ of *WP-Influence* +*D* and $x_{2,i}$ is equal to the $AvgRel$ of



(a) 5 dependent component pairs



(b) 10 dependent component pairs

Figure 8.28: Impact of atomic component reliability on $AvgRel$

recommendation algorithms in (*-D*) experiments. The results are presented in Figure 8.29, having the number of introduced dependent component pairs on the x-axis. The results clearly indicate that *RMSPE* for *WP-Influence* grows steeper than for the heuristic algorithms. Thus,

it can be concluded that heuristic algorithms are highly applicable in cases when strong atomic component mutual dependence is expected, but is not plausable to include it into the model.

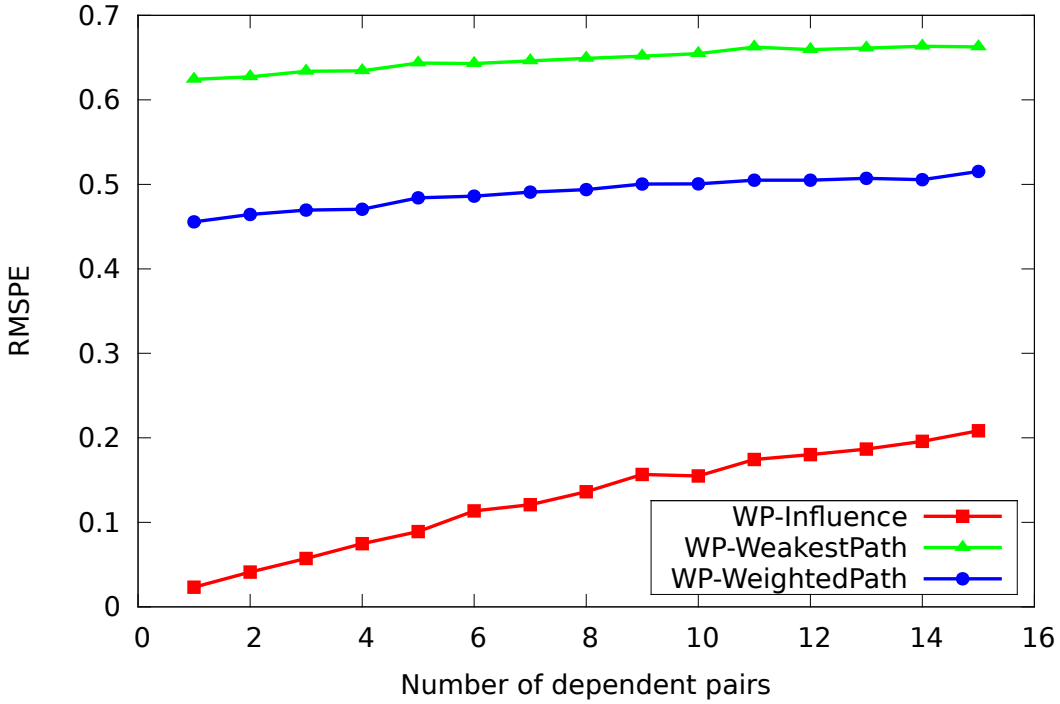


Figure 8.29: Difference between *WP-Influence* +*D* and -*D* experiments.

8.7 Summary of Results

In order to establish how the presented weak point recommendation algorithms behave in specific working conditions, two fundamental properties were evaluated: accuracy of the recommended solution and computational performance. The experiments were conducted on two distinct data sets: an artificially generated data set and a real-world data set consistent of Yahoo Pipes composite applications. The results presented in this chapter are summarized throughout the rest of this section.

Experimental setup

The performance of weak point recommendation algorithms was examined in relation to reliability models' size. In order to regulate reliability model size, a series of model data sets were generated, each data set containing models built out of a different number of atomic building components. In case of the real-world data set, distinct reliability model data sets were created

by sorting the applications according to their number of atomic building components, i.e. in that case the number of models was not equal in each data set. The recommendation algorithm's performance was evaluated based on the execution time required to compute a single weak point recommendation list. Specifically, average execution time measure for each particular data set was computed, as defined in Section 8.2. In order to experimentally confirm complexity analysis of weak point recommendation algorithms, presented in Section 6.4, two experiments were conducted with a different range in the number of atomic building components used to construct a particular reliability model data set.

The accuracy of weak point recommendation algorithms was evaluated for two distinct scenarios. In the application-wise component improvement scenario (Section 8.1.1), weak points were recommended for each application separately, based on its own weak point list. On the other hand, in the data-set-wise component improvement scenario (Section 8.1.2), weak points are recommended by aggregating all the individual weak point lists of every reliability model in the data set. In that case, the best average component for the entire data set is recommended. For each of the described evaluation scenarios two experiments were conducted with varying initial reliability values of atomic components. Experiments were conducted in multiple steps, called the improvement steps. At each improvement step a list of weak points was computed. Then, the most influential component was improved by increasing its reliability. This procedure was repeated until all the atomic building components were improved. For each improvement step average reliability, average reliability growth, total reliability growth and average reliability growth rate measures were computed as defined in Section 8.2.

Performance

For experimental setup with a lower range in the number of atomic building components (5 - 50), the results indicate a linear increase of average execution time with the increase in the number of atomic components. In addition, the *WP-Influence* algorithm displays the steepest increase of average execution time, while the average execution time of *WP-WeakestPath* algorithm increases with the lowest gradient. The findings are consistent for both the artificial and real-world data set, as shown by the graphs in Figures 8.3 and 8.19. On the other hand, experiment with a higher range in the number of atomic components (100 - 3000) was conducted only for the artificial data set, as there is no representative sample for the Yahoo Pipes data set beyond 15 atomic components. The experiments show that for a higher number of atomic components,

logarithmic factors in expressions defining computational complexity of the weak point recommendation algorithms become more significant. These findings are supported by the graphs presented in Figures 8.4 and 8.5. In conclusion, the results of all the conducted experiments decisively confirm that the *WP-Influence* algorithm is the most computationally demanding algorithm. Furthermore, regarding the heuristic algorithms, the *WP-WeightedPath* algorithm is decisively more computationally demanding than the *WP-WeakestPath* algorithm. Thus, the presented results are in accordance with the computational complexity analysis presented in Section 6.4.

Accuracy: Application-Wise Evaluation Scenario

In case of the application-wise evaluation scenario for the artificial data set, the results of both experiments clearly indicate that the *WP-Influence* algorithm recommends weak points with greater accuracy than the heuristic algorithms. This claim holds for each performed weak point improvement step. Furthermore, the results of all performed experiments confirm that the *WP-WeightedPath* algorithm is less accurate than the *WP-Influence* algorithm but more accurate than the *WP-WeakestPath* algorithm. These claims are supported by the results presented in Figures 8.3, 8.7 and 8.8. The same results regarding the recommendation algorithms' accuracy have been reached in case of the real-world data set. However, the difference in accuracy is less pronounced than in case of the artificial data set as workflow constructs in the real-world data set are less complex and, thus, define less influence paths per input random variable. In such cases, the accuracy of heuristic weak point recommendation algorithms more closely matches the accuracy of the *WP-Influence* algorithm. These claims are supported by the results presented in Figures 8.20, 8.21 and 8.22.

Accuracy: Data-Set-Wise Evaluation Scenario

Results for the data-set-wise evaluation scenario yield slightly different conclusions. In case of both data sets, it is clear that the *WP-Influence* provides recommendations with the greatest accuracy. This claim is supported by the results presented in Figures 8.11, 8.12, and 8.13 for the artificial data set and in Figures 8.24, 8.25, and 8.26 for the real-world data set. The heuristic algorithms closely match the performance of the *WP-Influence* algorithm. If fact, at certain improvement steps, heuristic algorithms perform as accurate as the *WP-Influence* algorithm. This is due to the fact that in case of the data-set-wise evaluation scenario the solution space is

aggregated. Specifically, the solution space for the data-set-wise evaluation scenario contains $n-i$ possible components at each step, where n is the number of composite applications and i is the ordinal number of the weak point improvement step. On the other hand, for the application-wise evaluation scenario the solution space is much larger as it contains $n(n-i)$ possible weak point selections at each improvement step. The experiment results clearly show that in case of aggregated solution space, the reliability growth achieved by applying the heuristic algorithms oscillates around the best solution provided by the *WP-Influence* algorithm. Furthermore, it is less conclusive which of the heuristic algorithms is more accurate as the *WP-WeakestPath* can provide a more accurate solution than the *WP-WeightedPath* algorithm. Therefore, error measures defined in Section 8.2.3 were applied to determine the difference between the heuristic algorithms and the *WP-Influence* algorithm. The computed measures for both data sets are presented in Table 8.1 for the artificial data set and in Table 8.4 for the real-world data set. The presented results show that the computed error measures for *WP-WeightedPath* algorithm are lower than for the *WP-WeakestPath* algorithm, indicating that the *WP-WeightedPath* algorithm more closely matches the *WP-Influence* algorithm. Therefore, it can be concluded that for an average case, the *WP-WeightedPath* algorithm is more accurate than the *WP-WeakestPath* algorithm.

Impact of additional computational overhead

Finally, in order to establish how can the presented weak point recommendation algorithms best be used in specific working conditions, the average reliability growth rate measure was computed, as defined in Section 8.2. The measure shows which algorithm achieves best reliability growth in relation to spent system resources and the number of component improvement steps. The computational resources, measured as average execution time, were varied by introduction of a fixed step-wise computational overhead. The results for both artificial and real-world data sets in the application-wise evaluation scenario are presented in Figures 8.10 and 8.23. The results indicate compact regions that denote when it is most efficient to use a certain recommendation algorithm in relation to the performed number of improvement steps and additional computational overhead.

Furthermore, the results for average reliability growth rate in case of the data-set-wise evaluation scenario are presented in Figures 8.14 and 8.27. Unlike in case of the application-wise evaluation scenario, the surfaces denoting when it is best to use a certain algorithm are discon-

tinued. The discontinuities appear when a certain algorithm matches the performance of the *WP-Influence* algorithm, as proven by the expression (8.18). Thus in case of the data-set-wise component improvement it is less conclusive which algorithm should be used at certain working conditions. This results also indicate that using the least computationally demanding algorithm, the *WP-WeakestPath* algorithms, yields a low decrease in achieved accuracy, making it highly applicable.In conclusion, the results confirm that in case of data-set-wise scenario, heuristic algorithms can be used to replace the *WP-Influence* algorithm entirely.

Impact of Atomic Component Dependency

The results presented in Figure 8.28 confirm that by ignoring mutual dependency of atomic components, the reliability models are less precise and yield less accurate weak point list recommendations. Furthermore, the results suggest that *WP-Influence* algorithm is more susceptible to introduction of atomic component dependencies. These conclusions are further supported by the results shown in Figure 8.29. Therefore, it can be concluded that heuristic algorithms are highly applicable in cases when existence of atomic component dependencies is not represented by the reliability model.

In conclusion, the experimental results conclusively confirm the premises stated in Chapter 6 regarding both accuracy and performance characteristics of the weak point recommendation algorithms. Regarding weak point recommendation accuracy, the *WP-Influence* is more accurate than the heuristic algorithms. Furthermore, the *WP-WeightedPath* algorithm, which takes into account existence of multiple influence paths, proves to be more accurate than the *WP-WeakestPath* algorithm. Moreover, the results also confirm computational complexity analysis presented in Section 6.4. The *WP-Influence* algorithm proves to be more computationally demanding then the heuristic algorithms. In addition, more accurate *WP-WeightedPath* algorithm proves to be more computationally demanding than the less accurate the *WP-WeakestPath* algorithm.

Chapter 9

Conclusion

The goal of this thesis is to put forward a reliability attainment mean that supports sustainable development of complex consumer applications. The presented solution is a design time iterative reliability management method. Its underlaying motivation is that by focusing reliability improvement actions, e.g. introduction of fault tolerance mechanisms, to the most significant application's building components, the required reliability level can be reached while spending less design and run time resources.

The selective improvement is performed by ordering the composite application's building components into weak point lists by their influence on the overall reliability. Thus, a more significant weak point is considered to be the one whose improvement results in a greater increase of the overall reliability. The iterative reliability management method, as defined in Chapter 4, consists of performing consecutive steps of estimating application's reliability, recommending weak point lists, and performing weak point improvements, until the required reliability level has been reached. Each of the method's steps needs to scale well with the increase in complexity of the constructed application, as well as be incorporated into a general consumer computing environment. To that end, the achieved scientific contributions of this thesis are: (1) reliability model for composite consumer applications, (2) scalable method for consumer application reliability management based on a suite of heuristic weak point recommendation algorithms, (3) evaluation of the weak point recommendation algorithm suite, and (4) architecture of a consumer assistant for development of reliable consumer applications.

Reliability model for composite consumer applications (1), presented in Chapter 5, is a probabilistic graphical model based on belief networks. Specifically, the introduced model enables architecture-based reliability estimation, as its goal is to identify reliability-critical ele-

ments of the consumer application's architecture. Furthermore, the presented approach is path based, meaning that the impact of each individual building component on the overall application reliability is modeled as superposition of multiple influence paths, derived by observing the application's workflow. This approach was chosen in favor of state based models and other optimization techniques, as they do not scale well with the increase in application size, i.e. number of building components and workflow complexity. Furthermore, since performance of weak point recommendation algorithms depends on the model size, additional optimizations to reduce the number of model paths were introduced. Specifically, in case of multiple sequential component invocations, influence paths leading from input model variables are aggregated using the repeat nodes.

The introduced reliability model can be used to recommend application's weak points by isolating influence paths leading from each individual input random variable, i.e. by assessing its individual influence on the overall application reliability. The second contribution of this thesis is a scalable method for weak point recommendation (2). The first hypothesis of the introduced approach is that by utilizing solely the graphical structure of the reliability model, it is possible to make the recommendation method less dependent on the number of input random variables. To that end, two heuristic algorithms *WP-WeightedPath* and *WP-WeakestPath* have been designed, as presented in Chapter 6. The heuristic algorithms are less computationally demanding than the *WP-Influence* algorithm that assesses the exact influence of an individual building component. These findings are confirmed by the computational complexity analysis presented in Section 6.4. Furthermore, the heuristic algorithms are also expected to be less accurate than the *WP-Influence* algorithm. However, *WP-WeightedPath* is expected to be more accurate than the *WP-WeakestPath* as it considers existence of multiple influence paths, making it also more computationally demanding. Having in mind the stated algorithm properties, the second hypothesis of the proposed approach is that by considering additional step-wise computational overheads of the reliability management method, through proper selection of weak point recommendation algorithms the overall method computational efficiency can be improved. For instance, in cases when a low additional step-wise overhead exists, it is more efficient to use the heuristic algorithms, as reaching final reliability in a greater number of improvement steps would result in a lower consumption of resources.

Feasibility of the weak point recommendation method, based on a suite of heuristic recommendation algorithms, is confirmed by an extensive evaluation (3), whose results are presented

in Chapter 8. The experiments were conducted for an artificial data set, as well as a real-world data set consistent of Yahoo Pipes reliability models. Regarding computational performance, the results conclusively confirm that heuristic algorithms are less computationally demanding than the *WP-Influence* algorithm. For instance, in case of the artificial data sets, that contain applications with up to 50 atomic components, the average execution time grows linearly with the number of atomic components. In those conditions, the average execution time of the *WP-Influence* algorithm grows with a 105.3 and 48.4 times higher gradient then for *WP-WeakestPath* and *WP-WeightedPath* algorithms respectively. Similarly, in case of the real-world data set, that contains applications with up to 10 atomic components, the average execution time of the *WP-Influence* algorithm grows with a 46.5 and 31.3 times higher gradient then for *WP-WeakestPath* and *WP-WeightedPath* algorithms respectively. For a higher number of atomic components, the logarithmic factors in the computational complexity expressions gain more significance, making the difference between the algorithms' execution times greater.

Regarding weak point recommendation accuracy, two distinct evaluation scenarios were considered: application-wise and data-set-wise component improvement. The former scenario considers improvements based on application's individual weak point list, while the latter considers improvements based on an aggregated weak point list, generated by combining all the weak point lists in the data set. The accuracy was evaluated by comparing the recommended solution with an optimal one. To that end, standard error measures were calculated, e.g. root mean square error *RMSE*. For the application-wise scenario, the results for both the artificial and real-world data sets conclusively confirm that *WP-Influence* is on average the most accurate algorithm at each conducted improvement step. In fact, it achieves several orders of magnitude lower *RMSE* than the heuristic algorithms. Furthermore, the results confirm expectations that *WP-WeightedPath* is more accurate than the *WP-WeakestPath* algorithm. Specifically, the *WP-WeightedPath*s achieves up to 2.95 times lower *RMSE* than the *WP-WeakestPath* in the artificial data set experiments and up to 5.41 times lower *RMSE* in the real-world data set experiments. On the other hand, in case of the data-set-wise scenario, heuristic algorithms match the *WP-Influence* algorithm more closely. In fact, at certain improvement steps, the heuristic algorithms completely match the accuracy of the *WP-Influence* algorithm. However, by taking into account the entire application improvement interval, the *WP-Influence* remains the most accurate algorithm. Furthermore, the *WP-WeightedPath* remains on average more accurate the *WP-WeakestPath* algorithm, as can be confirmed by the presented standard error measures. Re-

sults of evaluation clearly confirm the hypothesis that computational performance of weak point recommendation can be made less dependent on the number of application's building components at the cost of recommendation accuracy.

Further evaluations have been performed to assess the algorithms' applicability in relation to additional step-wise overhead of the reliability management method. The results for average growth rate (achieved reliability growth divided by spent computational resources) for application-wise scenario in case of both the artificial and real-world data sets clearly indicate the conditions under a certain algorithm achieves best performance. The results expectedly show that with the increase of additional overhead, the application interval of more accurate algorithms grows. On the other hand, in case of the data-set-wise scenario, the results are less conclusive. Since in this case, the solution space is severely reduced, heuristic algorithms can at certain steps match the *WP-Influence* in accuracy. This in turn makes the algorithm applicability independent of the increase in the additional overhead. For that reason, the heuristic algorithms can be considered to be highly applicable in case of the data-set-wise improvement, and, thus can be used to replace the *WP-Influence* algorithm entirely. In conclusion, the results of evaluation confirm the hypothesis that by determining recommendation algorithms' applicability in relation to the additional computational overhead, the reliability management method can be made more computationally efficient.

The final thesis contribution is focused on incorporating the reliability management method into the consumer computing environment (4). In order to support the method, consumer computing environment was extended both with assistant and programmable elements, as defined in Chapter 7. The consumer application reliability assistant was designed to implement the main processes of reliability management method. Based on the method definition, a set of operations needed to be performed by a consumer were identified. By taking into account the identified operations, the architecture of the consumer assistant was designed. Furthermore, in order to ensure the consumer assistant runs efficiently in distributed environments, application modules suitable for computational offloading were identified. By enabling computational offloading of demanding operations into a cloud infrastructure, the designed assistant can be deployed on devices with limited system resources, like mobile devices. Finally, an example of consumer application reliability assistant implemented as a web widget was presented. Furthermore, in order to enable application improvements through introduction of redundancy, programmable elements that support both backward and forward recovery fault tolerance methods were de-

signed. Like in case of the assistant element, consumer performed operations were identified and used to define the programmable components' architectures. Finally, implementations of programmable elements as web widgets were presented. Specifically, in case of backward recovery, check point and restart, and recovery blocks methods were implemented as widgets. On the other hand, in case of forward recover, n-version programming method was implemented. By combining the defined widgets, more complex fault tolerance methods, like consensus recovery blocks, can be constructed.

In conclusion, the main contribution of this thesis is a novel reliability management method that enables construction of reliable composite consumer applications. The presented research results advance the state-of-the-art in dependable computer systems as no comprehensive solution that considers construction of very large dependable component-based applications exists. Thus, apart from the consumer applications, the presented approach is applicable to other component-based systems, like service-oriented systems. Possible future research directions include both improving the reliability model and the weak point recommendation algorithms. In case of the reliability model, optimizations can be developed to further reduce the number of influence paths, consequently making the weak point recommendation procedures more efficient. On the other hand, heuristics of weak point recommendation algorithms can be further fine-tuned to get more accurate results, while retaining properties high computational efficiency. Finally, the presented data-set-wise evaluation scenario can be adapted to optimize resource consumption in cloud-based infrastructures.

Bibliography

[1] ITU, "The World in 2013: ICT Facts and Figures", http://www.itu.int/en/ITU-D/Statistics/Documents/facts/ICTFactsFigures2013.pdf, 2013.

[2] O'Reilly, T., What is Web 2.0. O'Reilly Media, 2009, [Online]. Available: http://books.google.hr/books?id=NpEk_WFCMdIC

[3] Mashable, "Google Play Hits 1 Million Apps", http://mashable.com/2013/07/24/google-play-1-million/, 2013.

[4] Nielsen, "A Year of Change And Growth in U.S. Smartphones", http://www.nielsen.com/us/en/newswire/2012/state-of-the-appnation-%C3%A2%C2%80%C2%93-a-year-of-change-and-growth-in-u-s-smartphones.html, 2013.

[5] Computerworld, "Programming ability is the new digital divide: Berners-Lee", http://www.computerworld.co.nz/article/452521/programming_ability_new_digital_divide_berners-lee/, 2012.

[6] "SOAP version 1.2", http://www.w3.org/TR/2007/REC-soap12-part0-20070427/, 2007.

[7] Fielding, R. T., "Architectural styles and the design of network-based software architectures", Ph.D. dissertation, University of California, Irvine, 2000, aAI9980887.

[8] Velte, T., Velte, A., Elsenpeter, R., Cloud Computing, A Practical Approach. Mcgraw-hill, 2009, [Online]. Available: http://books.google.hr/books?id=mf0LMXve2gEC

[9] Srbljic, S., Skvorc, D., Popovic, M., "Programming languages for end-user personalization of cyber-physical systems", AUTOMATIKA – Journal for Control, Measurement, Electronics, Computing and Communications, Vol. 53, No. 3, 2012, pp. 294-310.

[10] yi Hong, J., ho Suh, E., Kim, S.-J., "Context-aware systems: A literature review and classification", Expert Systems with Applications, Vol. 36, No. 4, 2009, pp.

8509 - 8522, [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0957417408007574

[11] Skvorc, D., "Consumer programming", Ph.D. dissertation, University of Zagreb, Faculty of electrical engineering and computing, 2010.

[12] Srbljic, S., Skvorc, D., Skrobo, D., "Widget-oriented consumer programming", AUTOMATIKA – Journal for Control, Measurement, Electronics, Computing and Communications, Vol. 50, No. 3–4, 2009, pp. 252–264.

[13] Popovic, M., "Consumer program synchronization", Ph.D. dissertation, University of Zagreb, Faculty of electrical engineering and computing, 2011.

[14] Kaisler, S., Armour, F., Espinosa, J., Money, W., "Big data: Issues and challenges moving forward", in System Sciences (HICSS), 2013 46th Hawaii International Conference on, 2013, pp. 995-1004.

[15] Su, X., Khoshgoftaar, T. M., "A survey of collaborative filtering techniques", Adv. in Artif. Intell., Vol. 2009, Jan. 2009, pp. 4:2–4:2, [Online]. Available: http://dx.doi.org/10.1155/2009/421425

[16] Van Meteren, R., Van Someren, M., "Using content-based filtering for recommendation", in Proceedings of the Machine Learning in the New Information Age: MLnet/ECML2000 Workshop, 2000.

[17] Alag, S., Collective Intelligence in Action, ser. Manning Pubs Co Series. Manning Publications Company, 2008, [Online]. Available: http://books.google.hr/books?id=0zQ3PQAACAAJ

[18] Quinn, A. J., Bederson, B. B., "Human computation: a survey and taxonomy of a growing field", in Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, ser. CHI '11. New York, NY, USA: ACM, 2011, pp. 1403–1412, [Online]. Available: http://doi.acm.org/10.1145/1978942.1979148

[19] Brabham, D. C., Crowdsourcing. MIT Press, 2013.

[20] Von Ahn, L., "Human computation", Ph.D. dissertation, Carnegie Mellon University, Pittsburgh, PA, USA, 2005, aAI3205378.

[21] Wang, F.-Y., Carley, K., Zeng, D., Mao, W., "Social computing: From social informatics to social intelligence", Intelligent Systems, IEEE, Vol. 22, No. 2, 2007, pp. 79-83.

[22] Ye, S., Lang, J., Wu, F., "Crawling online social graphs", in Web Conference (APWEB), 2010 12th International Asia-Pacific, 2010, pp. 236-242.

[23] Avizienis, A., Laprie, J.-C., "Dependable computing: From concepts to design diversity", Proceedings of the IEEE, Vol. 74, No. 5, 1986, pp. 629-638.

[24] Avizienis, A., Laprie, J.-C., Randell, B., Landwehr, C., "Basic concepts and taxonomy of dependable and secure computing", Dependable and Secure Computing, IEEE Transactions on, Vol. 1, No. 1, 2004, pp. 11-33.

[25] Avizienis, A., Laprie, J.-C., Randell, B., "Fundamental concepts of dependability", 2001.

[26] Laprie, J.-C., "Dependable computing and fault tolerance : Concepts and terminology", in Fault-Tolerant Computing, 1995, Highlights from Twenty-Five Years., Twenty-Fifth International Symposium on, 1995, pp. 2-.

[27] Lie, C. H., Hwang, C. L., Tillman, F. A., "Availability of maintained systems: A state-of-the-art survey", A I I E Transactions, Vol. 9, No. 3, 1977, pp. 247-259, [Online]. Available: http://www.tandfonline.com/doi/abs/10.1080/05695557708975153

[28] Mi, J., "Some comparison results of system availability", Naval Research Logistics (NRL), Vol. 45, No. 2, 1998, pp. 205–218, [Online]. Available: http://dx.doi.org/10.1002/(SICI)1520-6750(199803)45:2<205::AID-NAV5>3.0.CO;2-C

[29] Musa, J. D., Iannino, A., Okumoto, K., Software reliability: measurement, prediction, application. New York, NY, USA: McGraw-Hill, Inc., 1987.

[30] Kuo, W., Zuo, M., Optimal Reliability Modeling: Principles and Applications. Wiley, 2003, [Online]. Available: http://books.google.hr/books?id=vdZ4Bm-LnHMC

[31] Lyu, M. R., Ed., Handbook of software reliability engineering. Hightstown, NJ, USA: McGraw-Hill, Inc., 1996.

[32] "Information technology security evaluation criteria (itsec), provisional harmonised criteria", 1991.

[33] Jacob, J., “The basic integrity theorem”, in Computer Security Foundations Workshop IV, 1991. Proceedings, 1991, pp. 89-97.

[34] Ghezzi, C., Jazayeri, M., Mandrioli, D., Fundamentals of Software Engineering, 2nd ed. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2002.

[35] Siewiorek, D. P., Swarz, R. S., Reliable computer systems (3rd ed.): design and evaluation. Natick, MA, USA: A. K. Peters, Ltd., 1998.

[36] Sommerville, I., Software Engineering: (Update) (8th Edition), 8th ed. Addison Wesley, Jun. 2006, [Online]. Available: http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20&path=ASIN/0321313798

[37] Pinheiro, E., Weber, W.-D., Barroso, L. A., “Failure trends in a large disk drive population”, in 5th USENIX Conference on File and Storage Technologies (FAST 2007), 2007, pp. 17-29, [Online]. Available: http://research.google.com/archive/disk_failures.pdf

[38] Laprie, J., Arlat, J., Beounes, C., Kanoun, K., “Hardware- and software-fault tolerance”, in ESPRIT 90. Springer Netherlands, 1990, pp. 786-789, [Online]. Available: http://dx.doi.org/10.1007/978-94-009-0705-8_63

[39] Kadav, A., Renzelmann, M. J., Swift, M. M., “Tolerating hardware device failures in software”, in Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles, ser. SOSP ’09. New York, NY, USA: ACM, 2009, pp. 59–72, [Online]. Available: http://doi.acm.org/10.1145/1629575.1629582

[40] Park, J., Kim, H.-J., Shin, J.-H., Baik, J., “An embedded software reliability model with consideration of hardware related software failures”, in Software Security and Reliability (SERE), 2012 IEEE Sixth International Conference on, 2012, pp. 207-214.

[41] Cock, D., Carpenter, A., “A proposed hardware fault simulation engine”, in Design Automation. EDAC., Proceedings of the European Conference on, 1991, pp. 570-574.

[42] Weglarz, E., Saluja, K., Mak, T. M., “Testing of hard faults in simultaneous multithreaded processors”, in On-Line Testing Symposium, 2004. IOLTS 2004. Proceedings. 10th IEEE International, 2004, pp. 95-100.

[43] Wei, J., Rashid, L., Pattabiraman, K., Gopalakrishnan, S., “Comparing the effects of intermittent and transient hardware faults on programs”, in Dependable Systems and Networks Workshops (DSN-W), 2011 IEEE/IFIP 41st International Conference on, 2011, pp. 53-58.

[44] Chillarege, R., Bhandari, I., Chaar, J., Halliday, M., Moebus, D., Ray, B., Wong, M.-Y., “Orthogonal defect classification-a concept for in-process measurements”, Software Engineering, IEEE Transactions on, Vol. 18, No. 11, 1992, pp. 943-956.

[45] Cristian, F., “Understanding fault-tolerant distributed systems”, Commun. ACM, Vol. 34, No. 2, Feb. 1991, pp. 56–78, [Online]. Available: http://doi.acm.org/10.1145/102792.102801

[46] Wespi, A., Debar, H., Dacier, M., Nassehi, M., “Fixed- vs. variable-length patterns for detecting suspicious process behavior”, J. Comput. Secur., Vol. 8, No. 2,3, Aug. 2000, pp. 159–181, [Online]. Available: http://dl.acm.org/citation.cfm?id=1297828.1297830

[47] Avizienis, A., “Design of fault-tolerant computers”, in Proceedings of the November 14-16, 1967, fall joint computer conference, ser. AFIPS ’67 (Fall). New York, NY, USA: ACM, 1967, pp. 733–743, [Online]. Available: http://doi.acm.org/10.1145/1465611.1465708

[48] Avizienis, A., Chen, L., “On the implementation of N-version programming for software fault tolerance during execution”, in Proceedings of the IEEE International Computer Software and Applications Conference, 1977, pp. 149–155.

[49] Fox, A., Patterson, D., “Self-repairing computers”, SCIENTIFIC AMERICAN, Vol. 281, No. 11, 2003.

[50] Deswarte, Y., Blain, L., Fabre, J. C., “Intrusion tolerance in distributed computing systems”, in Research in Security and Privacy, 1991. Proceedings., 1991 IEEE Computer Society Symposium on, 1991, pp. 110-121.

[51] Deswarte, Y., Kanoun, K., Laprie, J.-C., “Diversity against accidental and deliberate faults”, in Computer Security, Dependability and Assurance: From Needs to Solutions, 1998. Proceedings, 1998, pp. 171-181.

[52] Arnold, T. F., “The concept of coverage and its effect on the reliability model of a repairable system”, Computers, IEEE Transactions on, Vol. C-22, No. 3, 1973, pp. 251-254.

[53] Avresky, D., Arlat, J., Laprie, J.-C., Crouzet, Y., “Fault injection for formal testing of fault tolerance”, Reliability, IEEE Transactions on, Vol. 45, No. 3, 1996, pp. 443-455.

[54] Bouricius, W. G., Carter, W. C., Schneider, P. R., “Reliability modeling techniques for self-repairing computer systems”, in Proceedings of the 1969 24th national conference, ser. ACM ’69. New York, NY, USA: ACM, 1969, pp. 295–309, [Online]. Available: http://doi.acm.org/10.1145/800195.805940

[55] Ellison, R., Ellison, R. J., Fisher, D., Fisher, D. A., Linger, R. C., Linger, R. C., Lipson, H. F., Lipson, H. F., Longstaff, T., Longstaff, T., Mead, N., Mead, N. R., “Survivable network systems: An emerging discipline”, CERT, Tech. Rep., 1997.

[56] Fabre, J., Nicomette, V., Perennou, T., Stroud, R., Wu, Z., “Implementing fault tolerant applications using reflective object-oriented programming”, in Fault-Tolerant Computing, 1995. FTCS-25. Digest of Papers., Twenty-Fifth International Symposium on, 1995, pp. 489-498.

[57] Forrest, S., Hofmeyr, S. A., Somayaji, A., Longstaff, T. A., “A sense of self for unix processes”, in In Proceedings of the 1996 IEEE Symposium on Security and Privacy. IEEE Computer Society Press, 1996, pp. 120–128.

[58] Hosford, J. E., “Measures of dependability”, Operations Research, Vol. 8, No. 1, 1960, pp. 53–64.

[59] Huang, Y., Kintala, C., Kolettis, N., Fulton, N., “Software rejuvenation: analysis, module and applications”, in Fault-Tolerant Computing, 1995. FTCS-25. Digest of Papers., Twenty-Fifth International Symposium on, 1995, pp. 381-390.

[60] Carzaniga, A., Gorla, A., Pezzè, M., “Handling software faults with redundancy”, in Architecting Dependable Systems VI, ser. Lecture Notes in Computer Science, Lemos, R., Fabre, J.-C., Gacek, C., Gadducci, F., Beek, M., Eds. Springer Berlin Heidelberg, 2009, Vol. 5835, pp. 148-171, [Online]. Available: http://dx.doi.org/10.1007/978-3-642-10248-6_7

[61] Landwehr, C. E., Bull, A. R., McDermott, J. P., Choi, W. S., “A taxonomy of computer program security flaws”, ACM Comput. Surv., Vol. 26, No. 3, Sep. 1994, pp. 211–254, [Online]. Available: http://doi.acm.org/10.1145/185403.185412

[62] Savolainen, P., Niemela, E., Savola, R., “A taxonomy of information security for service-centric systems”, in Software Engineering and Advanced Applications, 2007. 33rd EUROMICRO Conference on, 2007, pp. 5-12.

[63] Debar, H., Dacier, M., Wespi, A., “Towards a taxonomy of intrusion-detection systems”, Computer Networks, Vol. 31, No. 8, 1999, pp. 805 - 822, [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1389128698000176

[64] Ganek, A. G., Corbi, T. A., “The dawning of the autonomic computing era”, IBM Syst. J., Vol. 42, No. 1, Jan. 2003, pp. 5–18, [Online]. Available: http://dx.doi.org/10.1147/sj.421.0005

[65] Delac, G., Silic, M., Krolo, J., “Emerging security threats for mobile platforms”, in MIPRO, 2011 Proceedings of the 34th International Convention, 2011, pp. 1468-1473.

[66] Colbourn, C. J., McClary, D. W., “Locating and detecting arrays for interaction faults”, Journal of Combinatorial Optimization, Vol. 15, No. 1, 2008, pp. 17-48, [Online]. Available: http://dx.doi.org/10.1007/s10878-007-9082-4

[67] Chan, K., Bishop, J., Steyn, J., Baresi, L., Guinea, S., “A fault taxonomy for web service composition”, in Service-Oriented Computing - ICSOC 2007 Workshops, ser. Lecture Notes in Computer Science, Nitto, E., Ripeanu, M., Eds. Springer Berlin Heidelberg, 2009, Vol. 4907, pp. 363-375, [Online]. Available: http://dx.doi.org/10.1007/978-3-540-93851-4_36

[68] Nassu, B. T., Nanya, T., “Interaction faults caused by third-party external systems: a case study and challenges”, in Proceedings of the 5th international conference on Service availability, ser. ISAS’08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 59–74, [Online]. Available: http://dl.acm.org/citation.cfm?id=1788594.1788604

[69] Garvin, B., Cohen, M., “Feature interaction faults revisited: An exploratory study”, in Software Reliability Engineering (ISSRE), 2011 IEEE 22nd International Symposium on, 2011, pp. 90-99.

[70] Almeida, C., Verissimo, P., “Using light-weight groups to handle timing failures in quasi-synchronous systems”, in Real-Time Systems Symposium, 1998. Proceedings., The 19th IEEE, 1998, pp. 430-439.

[71] Taubenfeld, G., “Computing in the presence of timing failures”, in Proceedings of the 26th IEEE International Conference on Distributed Computing Systems, ser. ICDCS ’06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 16–, [Online]. Available: http://dx.doi.org/10.1109/ICDCS.2006.21

[72] Laranjeiro, N., Vieira, M., Madeira, H., “Predicting timing failures in web services”, in Database Systems for Advanced Applications, ser. Lecture Notes in Computer Science, Chen, L., Liu, C., Liu, Q., Deng, K., Eds. Springer Berlin Heidelberg, 2009, Vol. 5667, pp. 182-196, [Online]. Available: http://dx.doi.org/10.1007/978-3-642-04205-8_16

[73] Liang, C.-C., Wang, C.-H., Luh, H., Hsu, P.-Y., “Disaster avoidance mechanism for content-delivering service”, Comput. Oper. Res., Vol. 36, No. 1, Jan. 2009, pp. 27–39, [Online]. Available: http://dx.doi.org/10.1016/j.cor.2007.09.001

[74] Wang, N., Dong, B., “Fast failure recovery for reliable multicast-based content delivery”, in Network and Service Management (CNSM), 2010 International Conference on, 2010, pp. 505-510.

[75] Shen, Z., Wang, Q., “Failure detection, isolation, and recovery of multifunctional self-validating sensor”, Instrumentation and Measurement, IEEE Transactions on, Vol. 61, No. 12, 2012, pp. 3351-3362.

[76] Massoumnia, M.-A., Verghese, G. C., Willsky, A., “Failure detection and identification”, Automatic Control, IEEE Transactions on, Vol. 34, No. 3, 1989, pp. 316-321.

[77] Sprenkle, S., Gibson, E., Sampath, S., Pollock, L., “Automated replay and failure detection for web applications”, in Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering, ser. ASE ’05. New York, NY, USA: ACM, 2005, pp. 253–262, [Online]. Available: http://doi.acm.org/10.1145/1101908.1101947

[78] Ward, A., Glynn, P., Richardson, K., “Internet service performance failure detection”, SIGMETRICS Perform. Eval. Rev., Vol. 26, No. 3, Dec. 1998, pp. 38–43, [Online]. Available: http://doi.acm.org/10.1145/306225.306237

[79] Driscoll, K., Hall, B., Sivencrona, H., Zumsteg, P., “Byzantine fault tolerance, from theory to reality”, in Computer Safety, Reliability, and Security, ser. Lecture Notes in Computer Science, Anderson, S., Felici, M., Littlewood, B., Eds. Springer Berlin Heidelberg, 2003, Vol. 2788, pp. 235-248, [Online]. Available: http://dx.doi.org/10.1007/978-3-540-39878-3_19

[80] Greve, F., de Lima, M., Arantes, L., Sens, P., “A time-free byzantine failure detector for dynamic networks”, in Dependable Computing Conference (EDCC), 2012 Ninth European, 2012, pp. 191-202.

[81] Veronese, G., Correia, M., Bessani, A., Lung, L. C., Verissimo, P., “Efficient byzantine fault-tolerance”, Computers, IEEE Transactions on, Vol. 62, No. 1, 2013, pp. 16-30.

[82] Correia, M., Costa, P., Pasin, M., Bessani, A., Ramos, F., Verissimo, P., “On the feasibility of byzantine fault-tolerant mapreduce in clouds-of-clouds”, in Reliable Distributed Systems (SRDS), 2012 IEEE 31st Symposium on, 2012, pp. 448-453.

[83] Moniz, H., Neves, N., Correia, M., “Byzantine fault-tolerant consensus in wireless ad hoc networks”, Mobile Computing, IEEE Transactions on, Vol. PP, No. 99, 2012, pp. 1-1.

[84] Lamport, L., Shostak, R., Pease, M., “The byzantine generals problem”, ACM Trans. Program. Lang. Syst., Vol. 4, No. 3, Jul. 1982, pp. 382–401, [Online]. Available: http://doi.acm.org/10.1145/357172.357176

[85] Kropp, N., Koopman, P., Siewiorek, D., “Automated robustness testing of off-the-shelf software components”, in Fault-Tolerant Computing, 1998. Digest of Papers. Twenty-Eighth Annual International Symposium on, 1998, pp. 230-239.

[86] Tai, A., Meyer, J., Avizienis, A., “Performability enhancement of fault-tolerant software”, Reliability, IEEE Transactions on, Vol. 42, No. 2, 1993, pp. 227-237.

[87] Nikolaidis, E., Chen, S., Cudney, H., Haftka, R. T., Rosca, R., “Comparison of Probability and Possibility for Design Against Catastrophic Failure Under Uncertainty”,

Journal of Mechanical Design, Vol. 126, No. 3, 2004, p. 386, [Online]. Available: http://link.aip.org/link/JMDEDB/v126/i3/p386/s1&Agg=doi

[88] Ranade, S., Kolluru, R., Mitra, J., "Identification of chains of events leading to catastrophic failures of power systems", in Circuits and Systems, 2005. ISCAS 2005. IEEE International Symposium on, 2005, pp. 4187-4190 Vol. 5.

[89] Berson, A., Client/server architecture. McGraw-Hill, Inc., 1996.

[90] Taylor, R., Medvidovic, N., Dashofy, E., Software Architecture: Foundations, Theory, and Practice. Wiley, 2010, [Online]. Available: http://books.google.hr/books?id=j9pdGQAACAAJ

[91] Krafzig, D., Banke, K., Slama, D., Enterprise SOA: Service-Oriented Architecture Best Practices (The Coad Series). Upper Saddle River, NJ, USA: Prentice Hall PTR, 2004.

[92] Cusumano, M., "Cloud computing and saas as new computing platforms", Commun. ACM, Vol. 53, No. 4, Apr. 2010, pp. 27–29, [Online]. Available: http://doi.acm.org/10.1145/1721654.1721667

[93] Tsai, W. T., Zhang, D., Chen, Y., Huang, H., Paul, R., Liao, N., "A software reliability model for web services", in The 8th IASTED International Conference on Software Engineering and Applications, 2004, pp. 144–149.

[94] Zhao, S., Lu, X., Zhou, X., Zhang, T., Xue, J., "A reliability model for web services from the consumers' perspective", in Computer Science and Service System (CSSS), 2011 International Conference on, 2011, pp. 91-94.

[95] Zo, H., Nazareth, D. L., Jain, H. K., "Measuring reliability of applications composed of web services", in Proceedings of the 40th Annual Hawaii International Conference on System Sciences, ser. HICSS '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 278c–, [Online]. Available: http://dx.doi.org/10.1109/HICSS.2007.338

[96] Cortellessa, V., Grassi, V., "Reliability modeling and analysis of service-oriented architectures", in Test and Analysis of Web Services, Baresi, L., Nitto, E., Eds. Springer Berlin Heidelberg, 2007, pp. 339-362, [Online]. Available: http://dx.doi.org/10.1007/978-3-540-72912-9_12

[97] Goseva-Popstojanova, K., Trivedi, K. S., “Architecture-based approach to reliability assessment of software systems”, 2001.

[98] Thayer, R., Dorfman, M., System and software requirements engineering, ser. IEEE Computer Society Press tutorial. IEEE Computer Society Press, 1990, [Online]. Available: http://books.google.hr/books?id=r6VQAAAAMAAJ

[99] Jackson, A., Problem frames: analysing and structuring software development problems, ser. ACM Press Bks. Addison-Wesley, 2001, [Online]. Available: http://books.google.hr/books?id=j6hQAAAAMAAJ

[100] van Lamsweerde, A., “Goal-oriented requirements engineering: a guided tour”, in Requirements Engineering, 2001. Proceedings. Fifth IEEE International Symposium on, 2001, pp. 249-262.

[101] Pullum, L., Software Fault Tolerance Techniques and Implementation, ser. Artech House computing library. Artech House, Incorporated, 2001, [Online]. Available: http://books.google.hr/books?id=hqXvxsO5xz8C

[102] Meyer, B., “Applying "design by contract"”, Computer, Vol. 25, No. 10, Oct. 1992, pp. 40–51, [Online]. Available: http://dx.doi.org/10.1109/2.161279

[103] Freeman, E., Freeman, E., Sierra, K., Bates, B., Head First Design Patterns, ser. Head first series. O’Reilly Media, Incorporated, 2004, [Online]. Available: http://books.google.hr/books?id=LjJcCnNf92kC

[104] Gamma, E., Design Patterns: Elements of Reusable Object-oriented Software, ser. Addison-Wesley Professional Computing Series. Pearson Education, 2004, [Online]. Available: http://books.google.hr/books?id=12S_MKvhDl4C

[105] Boehm, B. W., “A spiral model of software development and enhancement”, Computer, Vol. 21, No. 5, May 1988, pp. 61–72, [Online]. Available: http://dx.doi.org/10.1109/2.59

[106] Meyer, B., “The new culture of software development”, J. Object Oriented Program., Vol. 3, No. 4, Oct. 1990, pp. 76–81, [Online]. Available: http://dl.acm.org/citation.cfm?id=98488.98504

[107] Horie, D., Kasahara, T., Goto, Y., Cheng, J., "A new model of software life cycle processes for consistent design, development, management, and maintenance of secure information systems", in Computer and Information Science, 2009. ICIS 2009. Eighth IEEE/ACIS International Conference on, 2009, pp. 897-902.

[108] Turpin, R., "A progressive software development lifecycle", in Engineering of Complex Computer Systems, 1996. Proceedings., Second IEEE International Conference on, 1996, pp. 208-211.

[109] Hierons, R. M., Bogdanov, K., Bowen, J. P., Cleaveland, R., Derrick, J., Dick, J., Gheorghe, M., Harman, M., Kapoor, K., Krause, P., Lüttgen, G., Simons, A. J. H., Vilkomir, S., Woodward, M. R., Zedan, H., "Using formal specifications to support testing", ACM Comput. Surv., Vol. 41, No. 2, Feb. 2009, pp. 9:1–9:76, [Online]. Available: http://doi.acm.org/10.1145/1459352.1459354

[110] Gaudel, M.-C., "Formal specification techniques", in Software Engineering, 1994. Proceedings. ICSE-16., 16th International Conference on, 1994, pp. 223-227.

[111] Lamsweerde, A. v., "Formal specification: a roadmap", in Proceedings of the Conference on The Future of Software Engineering, ser. ICSE '00. New York, NY, USA: ACM, 2000, pp. 147–159, [Online]. Available: http://doi.acm.org/10.1145/336512.336546

[112] Ambler, S., "A realistic look at object-oriented reuse", Softw. Dev., Vol. 6, No. 1, Jan. 1998, pp. 30–38, [Online]. Available: http://dl.acm.org/citation.cfm?id=291464.291468

[113] Pokkunuri, B. P., "Object oriented programming", SIGPLAN Not., Vol. 24, No. 11, Nov. 1989, pp. 96–101, [Online]. Available: http://doi.acm.org/10.1145/71605.71612

[114] Mitchell, J., Concepts in Programming Languages. Cambridge University Press, 2003, [Online]. Available: http://books.google.hr/books?id=7Uh8XGfJbEIC

[115] Bernhart, M., Reiterer, S., Matt, K., Mauczka, A., Grechenig, T., "A task-based code review process and tool to comply with the do-278/ed-109 standard for air traffic managment software development: An industrial case study", in High-Assurance Systems Engineering (HASE), 2011 IEEE 13th International Symposium on, 2011, pp. 182-187.

[116] Bernhart, M., Mauczka, A., Grechenig, T., "Adopting code reviews for agile software development", in Agile Conference (AGILE), 2010, 2010, pp. 44-47.

[117] Perpich, J. M., Perry, D. E., Porter, A. A., Votta, L. G., Wade, M. W., "Anywhere, anytime code inspections: using the web to remove inspection bottlenecks in large-scale software development", in Proceedings of the 19th international conference on Software engineering, ser. ICSE '97. New York, NY, USA: ACM, 1997, pp. 14–21, [Online]. Available: http://doi.acm.org/10.1145/253228.253234

[118] Raymond, E. S., The Cathedral and the Bazaar, 1st ed., O'Reilly, T., Ed. Sebastopol, CA, USA: O'Reilly & Associates, Inc., 1999.

[119] Feller, J., Fitzgerald, B. *et al.*, Understanding open source software development. Addison-Wesley London, 2002.

[120] Horwitz, S., Reps, T., Sagiv, M., Demand interprocedural dataflow analysis. ACM, 1995, Vol. 20, No. 4.

[121] Reps, T., Horwitz, S., Sagiv, M., "Precise interprocedural dataflow analysis via graph reachability", in Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages, ser. POPL '95. New York, NY, USA: ACM, 1995, pp. 49–61, [Online]. Available: http://doi.acm.org/10.1145/199448.199462

[122] Aho, A., Hopcroft, J., Ullman, J., The design and analysis of computer algorithms, ser. Addison-Wesley series in computer science and information processing. Addison-Wesley Pub. Co., 1974, [Online]. Available: http://books.google.hr/books?id=SJJQAAAAMAAJ

[123] Arora, S., Barak, B., Computational Complexity: A Modern Approach. Cambridge University Press, 2009, [Online]. Available: http://books.google.hr/books?id=nGvI7cOuOOQC

[124] Aho, A., Compilers: principles, techniques, & tools, ser. Addison-Wesley series in computer science. Pearson/Addison Wesley, 2007, [Online]. Available: http://books.google.hr/books?id=dIU_AQAAIAAJ

[125] Engler, D., Chelf, B., Chou, A., Hallem, S., "Checking system rules using system-specific, programmer-written compiler extensions", in Proceedings of the 4th conference on Symposium on Operating System Design & Implementation - Volume

4, ser. OSDI'00. Berkeley, CA, USA: USENIX Association, 2000, pp. 1–1, [Online]. Available: http://dl.acm.org/citation.cfm?id=1251229.1251230

[126] Cousot, P., Cousot, R., "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints", in Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, ser. POPL '77. New York, NY, USA: ACM, 1977, pp. 238–252, [Online]. Available: http://doi.acm.org/10.1145/512950.512973

[127] Cousot, P., "The role of abstract interpretation in formal methods", in Software Engineering and Formal Methods, 2007. SEFM 2007. Fifth IEEE International Conference on, 2007, pp. 135-140.

[128] Bernstein, P. A., Hadzilacos, V., Goodman, N., Concurrency control and recovery in database systems. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1987.

[129] Tanenbaum, A., Woodhull, A., Operating systems: design and implementation, ser. Prentice Hall Software Series. Pearson Prentice Hall, 2006, [Online]. Available: http://books.google.hr/books?id=RYU_AQAAIAAJ

[130] Meyer, B., "Dependable software", in Dependable Systems: Software, Computing, Networks, ser. Lecture Notes in Computer Science, Kohlas, J., Meyer, B., Schiper, A., Eds. Springer Berlin Heidelberg, 2006, Vol. 4028, pp. 1-33, [Online]. Available: http://dx.doi.org/10.1007/11808107_1

[131] Goldschlag, D., "Proving proof rules: a proof system for concurrent programs", in Computer Assurance, 1990. COMPASS '90, Systems Integrity, Software Safety and Process Security., Proceedings of the Fifth Annual Conference on, 1990, pp. 95-101.

[132] Duffy, D. A., Principles of automated theorem proving. New York, NY, USA: John Wiley & Sons, Inc., 1991.

[133] Dijkstra, E., A discipline of programming, ser. Prentice-Hall series in automatic computation. Prentice-Hall, Incorporated, 1976, [Online]. Available: http://books.google.ie/books?id=MsUmAAAAMAAJ

[134] Morgan, C., Programming from specifications, ser. Spectrum Book. Prentice Hall, 1990, [Online]. Available: http://books.google.hr/books?id=95dQAAAAMAAJ

[135] Fehnker, A., Huuck, R., "Model checking driven static analysis for the real world: designing and tuning large scale bug detection", Innovations in Systems and Software Engineering, Vol. 9, No. 1, 2013, pp. 45-56, [Online]. Available: http://dx.doi.org/10.1007/s11334-012-0192-5

[136] Baier, C., Katoen, J., Principles of Model Checking. Mit Press, 2008, [Online]. Available: http://books.google.ca/books?id=nDQiAQAAIAAJ

[137] Clarke, E., Grumberg, O., Peled, D., Model Checking. University Press Group Limited, 1999, [Online]. Available: http://books.google.ca/books?id=Nmc4wEaLXFEC

[138] Myers, G., Sandler, C., Badgett, T., Thomas, T., The Art of Software Testing, ser. Business Data Processing: A Wiley Series. Wiley, 2004, [Online]. Available: http://books.google.hr/books?id=86rz6UExDEEC

[139] Geras, A., "Leading manual test efforts with agile methods", in Agile, 2008. AGILE '08. Conference, 2008, pp. 245-251.

[140] Itkonen, J., Mantyla, M., Lassenius, C., "How do testers do it? an exploratory study on manual testing practices", in Empirical Software Engineering and Measurement, 2009. ESEM 2009. 3rd International Symposium on, 2009, pp. 494-497.

[141] Rushby, J., "Automated test generation and verified software", in Verified Software: Theories, Tools, Experiments, ser. Lecture Notes in Computer Science, Meyer, B., Woodcock, J., Eds. Springer Berlin Heidelberg, 2008, Vol. 4171, pp. 161-172, [Online]. Available: http://dx.doi.org/10.1007/978-3-540-69149-5_18

[142] Navabi, Z., "Deterministic test generation algorithms", in Digital System Test and Testable Design. Springer US, 2011, pp. 175-212, [Online]. Available: http://dx.doi.org/10.1007/978-1-4419-7548-5_6

[143] Hamzaoglu, I., Patel, J. H., "New techniques for deterministic test pattern generation", J. Electron. Test., Vol. 15, No. 1-2, Aug. 1999, pp. 63–73, [Online]. Available: http://dx.doi.org/10.1023/A:1008355411566

[144] Chevalley, P., Thevenod-Fosse, P., "Automated generation of statistical test cases from uml state diagrams", in Computer Software and Applications Conference, 2001. COMPSAC 2001. 25th Annual International, 2001, pp. 205-214.

[145] Sant, J., Souter, A., Greenwald, L., "An exploration of statistical models for automated test case generation", SIGSOFT Softw. Eng. Notes, Vol. 30, No. 4, May 2005, pp. 1–7, [Online]. Available: http://doi.acm.org/10.1145/1082983.1083256

[146] Song, N., Qin, J., Pan, X., Deng, Y., "Fault injection methodology and tools", in Electronics and Optoelectronics (ICEOE), 2011 International Conference on, Vol. 1, 2011, pp. V1-47-V1-50.

[147] Hierons, R., "Oracles for distributed testing", Software Engineering, IEEE Transactions on, Vol. 38, No. 3, 2012, pp. 629-641.

[148] Graves, T. L., Harrold, M. J., Kim, J.-M., Porter, A., Rothermel, G., "An empirical study of regression test selection techniques", ACM Trans. Softw. Eng. Methodol., Vol. 10, No. 2, Apr. 2001, pp. 184–208, [Online]. Available: http://doi.acm.org/10.1145/367008.367020

[149] Abran, A., Bourque, P., Dupuis, R., Moore, J. W., Eds., Guide to the Software Engineering Body of Knowledge - SWEBOK. Piscataway, NJ, USA: IEEE Press, 2001.

[150] Binder, R., Testing Object-oriented Software Testing: Models, Patterns, and Tools, ser. Object Technology Series. ADDISON WESLEY Publishing Company Incorporated, 2000, [Online]. Available: http://books.google.hr/books?id=P3UkDhLHP4YC

[151] Reifer, D., "Software failure modes and effects analysis", Reliability, IEEE Transactions on, Vol. R-28, No. 3, 1979, pp. 247-249.

[152] Gofuku, A., Koide, S., Shimada, N., "Fault tree analysis and failure mode effects analysis based on multi-level flow modeling and causality estimation", in SICE-ICASE, 2006. International Joint Conference, 2006, pp. 497-500.

[153] Koller, D., Friedman, N., Probabilistic Graphical Models: Principles and Techniques, ser. Adaptive Computation and Machine Learning. Mit Press, 2009, [Online]. Available: http://books.google.hr/books?id=7dzpHCHzNQ4C

[154] Yi, H., Jiang, C., Hu, H., Cai, K.-Y., Mathur, A., "Using markov-chains to model reliability and qos for deployed service-based systems", in Computer Software and Applications Conference Workshops (COMPSACW), 2011 IEEE 35th Annual, 2011, pp. 356-361.

[155] Mohan, K., Verma, A., Srividya, A., Rao, G., Gedela, R., "Early quantitative software reliability prediction using petri-nets", in Industrial and Information Systems, 2008. ICIIS 2008. IEEE Region 10 and the Third international Conference on, 2008, pp. 1-6.

[156] Alam, M., Al-Saggaf, U., "Quantitative reliability evaluation of repairable phased-mission systems using markov approach", Reliability, IEEE Transactions on, Vol. 35, No. 5, 1986, pp. 498-503.

[157] Brall, A., Hagen, W., Tran, H., "Reliability block diagram modeling - comparisons of three software packages", in Reliability and Maintainability Symposium, 2007. RAMS '07. Annual, 2007, pp. 119-124.

[158] Distefano, S., Puliafito, A., "Dependability evaluation with dynamic reliability block diagrams and dynamic fault trees", IEEE Trans. Dependable Secur. Comput., Vol. 6, No. 1, Jan. 2009, pp. 4–17, [Online]. Available: http://dx.doi.org/10.1109/TDSC.2007.70242

[159] Xing, L., Amari, S., "Fault tree analysis", in Handbook of Performability Engineering, Misra, K., Ed. Springer London, 2008, pp. 595-620, [Online]. Available: http://dx.doi.org/10.1007/978-1-84800-131-2_38

[160] Geymayr, J., Ebecken, N. F. F., "Fault-tree analysis: a knowledge-engineering approach", Reliability, IEEE Transactions on, Vol. 44, No. 1, 1995, pp. 37-45.

[161] Ramamoorthy, C. V., Ho, G. S., Han, Y. W., "Fault tree analysis of computer systems", in Proceedings of the June 13-16, 1977, national computer conference, ser. AFIPS '77. New York, NY, USA: ACM, 1977, pp. 13–17, [Online]. Available: http://doi.acm.org/10.1145/1499402.1499407

[162] Xie, M., Software Reliability Modelling, ser. [Series on quality, reliability & engineering statistics. World Scientific Publishing Company Incorporated, 1991, [Online]. Available: http://books.google.hr/books?id=WSLzZc1ANqEC

[163] Gokhale, S. S., "Architecture-based software reliability analysis: Overview and limitations", IEEE Trans. Dependable Secur. Comput., Vol. 4, No. 1, Jan. 2007, pp. 32–40, [Online]. Available: http://dx.doi.org/10.1109/TDSC.2007.4

[164] Brosch, F., Koziolek, H., Buhnova, B., Reussner, R., "Architecture-based reliability prediction with the palladio component model", Software Engineering, IEEE Transactions on, Vol. 38, No. 6, 2012, pp. 1319-1339.

[165] Gokhale, S., Wong, W., Trivedi, K., Horgan, J., "An analytical approach to architecture-based software reliability prediction", in Computer Performance and Dependability Symposium, 1998. IPDS '98. Proceedings. IEEE International, 1998, pp. 13-22.

[166] Meedeniya, I., Moser, I., Aleti, A., Grunske, L., "Architecture-based reliability evaluation under uncertainty", in Proceedings of the joint ACM SIGSOFT conference – QoSA and ACM SIGSOFT symposium – ISARCS on Quality of software architectures – QoSA and architecting critical systems – ISARCS, ser. QoSA-ISARCS '11. New York, NY, USA: ACM, 2011, pp. 85–94, [Online]. Available: http://doi.acm.org/10.1145/2000259.2000275

[167] Brito, P. H. S., Lemos, R. d., Martins, E., Moraes, R., Rubira, C. M. F., "Architectural-based validation of fault-tolerant software", in Proceedings of the 2009 Fourth Latin-American Symposium on Dependable Computing, ser. LADC '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 103–110, [Online]. Available: http://dx.doi.org/10.1109/LADC.2009.22

[168] Mason, D. V., "Probabilistic program analysis for software component reliability", Ph.D. dissertation, University of Waterloo, Waterloo, Ont., Canada, Canada, 2002, aAINQ85339.

[169] Yacoub, S. M., Cukic, B., Ammar, H. H., "Scenario-based reliability analysis of component-based software", in Proceedings of the 10th International Symposium on Software Reliability Engineering, ser. ISSRE '99. Washington, DC, USA: IEEE Computer Society, 1999, pp. 22–, [Online]. Available: http://dl.acm.org/citation.cfm?id=851020.856175

[170] Cheung, R., "A user-oriented software reliability model", Software Engineering, IEEE Transactions on, Vol. SE-6, No. 2, 1980, pp. 118-125.

[171] Littlewood, B., “A reliability model for markov structured software”, SIGPLAN Not., Vol. 10, No. 6, Apr. 1975, pp. 204–207, [Online]. Available: http://doi.acm.org/10.1145/390016.808441

[172] Siegrist, K., “Reliability of systems with markov transfer of control”, Software Engineering, IEEE Transactions on, Vol. 14, No. 7, 1988, pp. 1049-1053.

[173] Gokhale, S., Trivedi, K., “Reliability prediction and sensitivity analysis based on software architecture”, in Software Reliability Engineering, 2002. ISSRE 2003. Proceedings. 13th International Symposium on, 2002, pp. 64-75.

[174] Laprie, J.-C., Kanoun, K., “X-ware reliability and availability modeling”, IEEE Trans. Softw. Eng., Vol. 18, No. 2, Feb. 1992, pp. 130–147, [Online]. Available: http://dx.doi.org/10.1109/32.121755

[175] Zheng, Z., Lyu, M., “Ws-dream: A distributed reliability assessment mechanism for web services”, in Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008. IEEE International Conference on, june 2008, pp. 392 -397.

[176] De Barros, M., Shiau, J., Shang, C., Gidewall, K., Shi, H., Forsmann, J., “Web services wind tunnel: On performance testing large-scale stateful web services”, in Dependable Systems and Networks, 2007. DSN ’07. 37th Annual IEEE/IFIP International Conference on, june 2007, pp. 612 -617.

[177] Weyuker, E., Vokolos, F., “Experience with performance testing of software systems: issues, an approach, and case study”, Software Engineering, IEEE Transactions on, Vol. 26, No. 12, 2000, pp. 1147-1156.

[178] Denaro, G., Polini, A., Emmerich, W., “Early performance testing of distributed software applications”, SIGSOFT Softw. Eng. Notes, Vol. 29, No. 1, Jan. 2004, pp. 94–103, [Online]. Available: http://doi.acm.org/10.1145/974043.974059

[179] Xie, J., Ye, X., Li, B., Xie, F., “A configurable web service performance testing framework”, in High Performance Computing and Communications, 2008. HPCC ’08. 10th IEEE International Conference on, 2008, pp. 312-319.

[180] Musa, J. D., "Operational profiles in software-reliability engineering", IEEE Softw., Vol. 10, No. 2, Mar. 1993, pp. 14–32, [Online]. Available: http://dx.doi.org/10.1109/52.199724

[181] Cai, K., Software Defect and Operational Profile Modeling, ser. Chinese studies in information science. Kluwer Academic Publishers, 1998, [Online]. Available: http://books.google.hr/books?id=lYsLSf2fcXwC

[182] Cacheda, F., Carneiro, V., Fernández, D., Formoso, V., "Comparison of collaborative filtering algorithms: Limitations of current techniques and proposals for scalable, high-performance recommender systems", ACM Trans. Web, Vol. 5, No. 1, Feb. 2011, pp. 2:1–2:33, [Online]. Available: http://doi.acm.org/10.1145/1921591.1921593

[183] Zheng, Z., Ma, H., Lyu, M., King, I., "Qos-aware web service recommendation by collaborative filtering", Services Computing, IEEE Transactions on, Vol. 4, No. 2, april-june 2011, pp. 140 -152.

[184] Zheng, Z., Lyu, M., "Collaborative reliability prediction of service-oriented systems", in Software Engineering, 2010 ACM/IEEE 32nd International Conference on, Vol. 1, may 2010, pp. 35 -44.

[185] Silic, M., Delac, G., Krka, I., Srbljic, S., "Scalable and accurate prediction of availability of atomic web services", Services Computing, IEEE Transactions on, Vol. PP, No. 99, 2013, pp. 1-1.

[186] Silic, M., Delac, G., Srbljic, S., "Prediction of atomic web services reliability based on k-means clustering", in Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, ser. ESEC/FSE 2013. New York, NY, USA: ACM, 2013, pp. 70–80, [Online]. Available: http://doi.acm.org/http://dx.doi.org/10.1145/2491411.2491424

[187] Eles, P., Izosimov, V., Pop, P., Peng, Z., "Synthesis of fault-tolerant embedded systems", in Proceedings of the conference on Design, automation and test in Europe, ser. DATE '08. New York, NY, USA: ACM, 2008, pp. 1117–1122, [Online]. Available: http://doi.acm.org/10.1145/1403375.1403644

[188] Huang, J., Huang, K., Raabe, A., Buckl, C., Knoll, A., “Towards fault-tolerant embedded systems with imperfect fault detection”, in Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE, 2012, pp. 188-196.

[189] Avizienis, A., Kelly, J. P. J., “Fault tolerance by design diversity: Concepts and experiments”, Computer, Vol. 17, No. 8, Aug. 1984, pp. 67–80, [Online]. Available: http://dx.doi.org/10.1109/MC.1984.1659219

[190] Kelly, J., McVittie, T., Yamamoto, W., “Implementing design diversity to achieve fault tolerance”, Software, IEEE, Vol. 8, No. 4, 1991, pp. 61-71.

[191] Ammann, P. E., Knight, J. C., “Data diversity: An approach to software fault tolerance”, IEEE Trans. Comput., Vol. 37, No. 4, Apr. 1988, pp. 418–425, [Online]. Available: http://dx.doi.org/10.1109/12.2185

[192] Nakagawa, T., Yasui, K., “Optimal testing-policies for intermittent faults”, Reliability, IEEE Transactions on, Vol. 38, No. 5, 1989, pp. 577-580.

[193] Contant, O., Lafortune, S., Teneketzis, D., “Diagnosis of intermittent faults”, Discrete Event Dynamic Systems, Vol. 14, No. 2, Apr. 2004, pp. 171–202, [Online]. Available: http://dx.doi.org/10.1023/B:DISC.0000018570.20941.d2

[194] Bondavalli, A., Chiaradonna, S., di Giandomenico, F., Grandoni, F., “Threshold-based mechanisms to discriminate transient from intermittent faults”, Computers, IEEE Transactions on, Vol. 49, No. 3, 2000, pp. 230-245.

[195] Saleh, A., Patel, J., “Transient-fault analysis for retry techniques”, Reliability, IEEE Transactions on, Vol. 37, No. 3, 1988, pp. 323-330.

[196] Sosnowski, J., “Transient fault tolerance in redundant systems”, in CompEuro ’92 . ’Computer Systems and Software Engineering’,Proceedings., 1992, pp. 280-285.

[197] Jha, S., Li, W., Seshia, S., “Localizing transient faults using dynamic bayesian networks”, in High Level Design Validation and Test Workshop, 2009. HLDVT 2009. IEEE International, 2009, pp. 82-87.

[198] Vaidyanathan, K., Trivedi, K., “A comprehensive model for software rejuvenation”, Dependable and Secure Computing, IEEE Transactions on, Vol. 2, No. 2, 2005, pp. 124-137.

[199] Musuvathi, M., Qadeer, S., Ball, T., Basler, G., Nainar, P. A., Neamtiu, I., "Finding and reproducing heisenbugs in concurrent programs", in Proceedings of the 8th USENIX conference on Operating systems design and implementation, ser. OSDI'08. Berkeley, CA, USA: USENIX Association, 2008, pp. 267–280, [Online]. Available: http://dl.acm.org/citation.cfm?id=1855741.1855760

[200] Wilfredo, T., "Software fault tolerance: A tutorial", NASA, Tech. Rep., 2000.

[201] Abbott, R. J., "Resourceful systems for fault tolerance, reliability, and safety", ACM Comput. Surv., Vol. 22, No. 1, Mar. 1990, pp. 35–68, [Online]. Available: http://doi.acm.org/10.1145/78949.78951

[202] Goodenough, J. B., "Exception handling: issues and a proposed notation", Commun. ACM, Vol. 18, No. 12, Dec. 1975, pp. 683–696, [Online]. Available: http://doi.acm.org/10.1145/361227.361230

[203] Sawadpong, P., Allen, E., Williams, B., "Exception handling defects: An empirical study", in High-Assurance Systems Engineering (HASE), 2012 IEEE 14th International Symposium on, 2012, pp. 90-97.

[204] Taylor, D., Morgan, D., Black, J., "Redundancy in data structures: Improving software fault tolerance", Software Engineering, IEEE Transactions on, Vol. SE-6, No. 6, 1980, pp. 585-594.

[205] Parhami, B., "Voting algorithms", Reliability, IEEE Transactions on, Vol. 43, No. 4, 1994, pp. 617-629.

[206] Eckhardt, D. E., Lee, L. D., "A theoretical basis for the analysis of multiversion software subject to coincident errors", IEEE Trans. Softw. Eng., Vol. 11, No. 12, Dec. 1985, pp. 1511–1517, [Online]. Available: http://dx.doi.org/10.1109/TSE.1985.231895

[207] Scott, R., Gault, J., McAllister, D., "Fault-tolerant software reliability modeling", Software Engineering, IEEE Transactions on, Vol. SE-13, No. 5, 1987, pp. 582-592.

[208] McAllister, D., Sun, C.-E., Vouk, M., "Reliability of voting in fault-tolerant software systems for small output-spaces", Reliability, IEEE Transactions on, Vol. 39, No. 5, 1990, pp. 524-534.

[209] Lamport, L., “Fast paxos”, Distributed Computing, Vol. 19, 2006, pp. 79-103, 10.1007/s00446-006-0005-x.

[210] Brilliant, S. S., Knight, J. C., Leveson, N. G., “The consistent comparison problem in n-version software”, SIGSOFT Softw. Eng. Notes, Vol. 12, No. 1, Jan. 1987, pp. 29–34, [Online]. Available: http://doi.acm.org/10.1145/24574.24575

[211] Issarny, V., Tartanoglu, F., Romanovsky, A., Levy, N., “Coordinated forward error recovery for composite web services”, in Reliable Distributed Systems, 2003. Proceedings. 22nd International Symposium on, 2003, pp. 167-176.

[212] BrzeziŃski, J., Helary, J.-M., Raynal, M., “Semantics of recovery lines for backward recovery in distributed systems”, Annales Des Télécommunications, Vol. 50, No. 11-12, 1995, pp. 874-887, [Online]. Available: http://dx.doi.org/10.1007/BF03005244

[213] Ziv, A., Bruck, J., “An on-line algorithm for checkpoint placement”, IEEE Trans. Comput., Vol. 46, No. 9, Sep. 1997, pp. 976–985, [Online]. Available: http://dx.doi.org/10.1109/12.620479

[214] Hiroyama, S., Dohi, T., Okamura, H., “Comparison of aperiodic checkpoint placement algorithms”, in Advanced Computer Science and Information Technology, ser. Communications in Computer and Information Science, Tomar, G., Chang, R.-S., Gervasi, O., Kim, T.-h., Bandyopadhyay, S., Eds. Springer Berlin Heidelberg, 2010, Vol. 74, pp. 145-156, [Online]. Available: http://dx.doi.org/10.1007/978-3-642-13346-6_14

[215] Xu, J., Randell, B., Object-oriented Construction of Fault-tolerant Software, ser. Technical report series. University of Newcastle upon Tyne, Computing Science, 1993, [Online]. Available: http://books.google.hr/books?id=p2AAMwEACAAJ

[216] Pradhan, D. K., Ed., Fault-tolerant computer system design. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1996.

[217] Randell, B., Xu, J., of Newcastle upon Tyne. Computing Science, U., Recovery blocks, ser. Technical report series. University of Newcastle upon Tyne, Computing Science, 1994, [Online]. Available: http://books.google.hr/books?id=9rYFMwEACAAJ

[218] Hoare, T., “Recovery blocks”, in Dependable and Historic Computing, ser. Lecture Notes in Computer Science, Jones, C., Lloyd, J., Eds. Springer Berlin Heidelberg, 2011, Vol.

6875, pp. 261-266, [Online]. Available: http://dx.doi.org/10.1007/978-3-642-24541-1_19

[219] Tyrrell, A. M., "Recovery blocks and algorithm-based fault tolerance", in EUROMICRO 96. Beyond 2000: Hardware and Software Design Strategies., Proceedings of the 22nd EUROMICRO Conference, 1996, pp. 292-299.

[220] Kim, K. H., Welch, H. O., "Distributed execution of recovery blocks: An approach for uniform treatment of hardware and software faults in real-time applications", IEEE Trans. Comput., Vol. 38, No. 5, May 1989, pp. 626–636, [Online]. Available: http://dx.doi.org/10.1109/12.24266

[221] Avizienis, A., "The n-version approach to fault-tolerant software", IEEE Trans. Softw. Eng., Vol. 11, No. 12, Dec. 1985, pp. 1491–1501, [Online]. Available: http://dx.doi.org/10.1109/TSE.1985.231893

[222] Chen, L., Avizienis, A., "N-version programminc: A fault-tolerance approach to rellablllty of software operatlon", in Fault-Tolerant Computing, 1995, Highlights from Twenty-Five Years., Twenty-Fifth International Symposium on, 1995, pp. 113-.

[223] Bondavalli, A., Chiaradonna, S., Di Giandomenico, F., Xu, J., "An adaptive approach to achieving hardware and software fault tolerance in a distributed computing environment", J. Syst. Archit., Vol. 47, No. 9, Mar. 2002, pp. 763–781, [Online]. Available: http://dx.doi.org/10.1016/S1383-7621(01)00029-7

[224] Grnarov, A., Arlat, J., Avizienis, A., "On the performance of software fault-tolerance strategies", in Proc. 10th International Symposium on Fault-Tolerant Computing, Kyoto, Japan, 1980, pp. 251–253.

[225] Laprie, J.-C., Béounes, C., Kanoun, K., "Definition and analysis of hardware- and software-fault-tolerant architectures", Computer, Vol. 23, No. 7, Jul. 1990, pp. 39–51, [Online]. Available: http://dx.doi.org/10.1109/2.56851

[226] Hwang, S.-Y., Lim, E.-P., Lee, C.-H., Chen, C.-H., "Dynamic web service selection for reliable web service composition", Services Computing, IEEE Transactions on, Vol. 1, No. 2, april-june 2008, pp. 104 -116.

[227] Liao, C.-F., Jong, Y.-W., Fu, L.-C., “Toward reliable service management in message-oriented pervasive systems”, Services Computing, IEEE Transactions on, Vol. 4, No. 3, july-sept. 2011, pp. 183 -195.

[228] Zeng, L., Benatallah, B., Ngu, A., Dumas, M., Kalagnanam, J., Chang, H., “Qos-aware middleware for web services composition”, Software Engineering, IEEE Transactions on, Vol. 30, No. 5, may 2004, pp. 311 - 327.

[229] Aime, M., Lioy, A., Pomi, P., “Automatic (re)configuration of it systems for dependability”, Services Computing, IEEE Transactions on, Vol. 4, No. 2, april-june 2011, pp. 110 -124.

[230] Alrifai, M., Risse, T., “Combining global optimization with local selection for efficient qos-aware service composition”, in Proceedings of the 18th international conference on World wide web, ser. WWW ’09. New York, NY, USA: ACM, 2009, pp. 881–890.

[231] Fujii, K., Suda, T., “Semantics-based context-aware dynamic service composition”, ACM Trans. Auton. Adapt. Syst., Vol. 4, No. 2, May 2009, pp. 12:1–12:31.

[232] Hanen, H., Bourcier, J., “Dependability-Driven Runtime Management of Service Oriented Architectures”, in PESOS - 4th International Workshop on Principles of Engineering Service-Oriented Systems - 2012, Zurich, Suisse, Jun. 2012.

[233] Mabrouk, N. B., Beauche, S., Kuznetsova, E., Georgantas, N., Issarny, V., “Qos-aware service composition in dynamic service oriented environments”, in Proceedings of the 10th ACM/IFIP/USENIX International Conference on Middleware, ser. Middleware ’09. New York, NY, USA: Springer-Verlag New York, Inc., 2009, pp. 7:1–7:20.

[234] Kalasapur, S., Kumar, M., Shirazi, B., “Dynamic service composition in pervasive computing”, Parallel and Distributed Systems, IEEE Transactions on, Vol. 18, No. 7, july 2007, pp. 907 -918.

[235] Alrifai, M., Risse, T., Nejdl, W., “A hybrid approach for efficient web service composition with end-to-end qos constraints”, ACM Trans. Web, Vol. 6, No. 2, Jun. 2012, pp. 7:1–7:31.

[236] Zeng, L., Ngu, A., Benatallah, B., Podorozhny, R., Lei, H., "Dynamic composition and optimization of web services", Distributed and Parallel Databases, Vol. 24, No. 1-3, 2008, pp. 45-72, [Online]. Available: http://dx.doi.org/10.1007/s10619-008-7030-7

[237] Belaïd, D., Mukhtar, H., Ozanne, A., Tata, S., "Dynamic component selection for sca applications", in Software Services for e-Business and e-Society, ser. IFIP Advances in Information and Communication Technology, Godart, C., Gronau, N., Sharma, S., Canals, G., Eds. Springer Berlin Heidelberg, 2009, Vol. 305, pp. 272-286, [Online]. Available: http://dx.doi.org/10.1007/978-3-642-04280-5_22

[238] Liu, W., Wong, W., "Discovering homogenous service communities through web service clustering", in Proceedings of the 2008 AAMAS international conference on Service-oriented computing: agents, semantics, and engineering, ser. SOCASE'08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 69–82.

[239] Liu, X., Huang, G., Mei, H., "Discovering homogeneous web service community in the user-centric web environment", IEEE Transactions on Services Computing, Vol. 2, 2009, pp. 167-181.

[240] Zhang, X., Yin, Y., Zhang, M., Zhang, B., "Web service community discovery based on spectrum clustering", in Computational Intelligence and Security, 2009. CIS '09. International Conference on, Vol. 2, dec. 2009, pp. 187 -191.

[241] Perryea, C., Chung, S., "Community-based service discovery", in Web Services, 2006. ICWS '06. International Conference on, sept. 2006, pp. 903 -906.

[242] Brnsted, J., Hansen, K., Ingstrup, M., "Service composition issues in pervasive computing", Pervasive Computing, IEEE, Vol. 9, No. 1, jan.-march 2010, pp. 62 -70.

[243] Al-Masri, E., Mahmoud, Q. H., "Investigating web services on the world wide web", in Proceedings of the 17th international conference on World Wide Web, ser. WWW '08. New York, NY, USA: ACM, 2008, pp. 795–804.

[244] Liang, Q., Wu, X., Lau, H. C., "Optimizing service systems based on application-level qos", Services Computing, IEEE Transactions on, Vol. 2, No. 2, april-june 2009, pp. 108 -121.

[245] Tang, M., Ai, L., “A hybrid genetic algorithm for the optimal constrained web service selection problem in web service composition”, in Evolutionary Computation (CEC), 2010 IEEE Congress on, 2010, pp. 1-8.

[246] Elnozahy, E. N. M., Alvisi, L., Wang, Y.-M., Johnson, D. B., “A survey of rollback-recovery protocols in message-passing systems”, ACM Comput. Surv., Vol. 34, No. 3, Sep. 2002, pp. 375–408.

[247] Mansour, H., Dillon, T., “Dependability and rollback recovery for composite web services”, Services Computing, IEEE Transactions on, Vol. 4, No. 4, oct.-dec. 2011, pp. 328 -339.

[248] Brereton, P., Budgen, D., “Component-based systems: A classification of issues”, Computer, Vol. 33, No. 11, Nov. 2000, pp. 54–62, [Online]. Available: http://dx.doi.org/10.1109/2.881695

[249] Balen, H., Elenko, M., Jones, J., Palumbo, G., Distributed Object Architectures with CORBA, ser. SIGS reference library series. Cambridge University Press, 2000, [Online]. Available: http://books.google.hr/books?id=M0aIf-3g_sgC

[250] Delac, G., Silic, M., Srbljic, S., “A Reliability Improvement Method for SOA-Based Applications”, manuscript submitted for publication, 2012.

[251] Delac, G., Silic, M., Srbljic, S., “Reliability modeling for soa systems”, in MIPRO, 2012 Proceedings of the 35th International Convention, 2012, pp. 847-852.

[252] Syu, Y., Ma, S.-P., Kuo, J.-Y., FanJiang, Y.-Y., “A survey on automated service composition methods and related techniques”, in Services Computing (SCC), 2012 IEEE Ninth International Conference on, 2012, pp. 290-297.

[253] Christensen, E., Curbera, F., Meredith, G., Weerawarana, S., “Web service definition language (wsdl)”, W3C, Tech. Rep., Mar. 2001, [Online]. Available: http://www.w3.org/TR/wsdl

[254] Budiselic, I., “Component recommendation for development of composite consumer applications”, Ph.D. dissertation, University of Zagreb, Faculty of electrical engineering and computing, 2014.

[255] Vladimir, K., “Peer tutoring in consumer computing”, Ph.D. dissertation, University of Zagreb, Faculty of electrical engineering and computing, 2013.

[256] Silic, M., “Reliability prediction of consumer computing applications”, Ph.D. dissertation, University of Zagreb, Faculty of electrical engineering and computing, 2013.

[257] Winkler, M., Springer, T., Trigos, E., Schill, A., “Analysing dependencies in service compositions”, in Service-Oriented Computing. ICSOC/ServiceWave 2009 Workshops, ser. Lecture Notes in Computer Science, 2010, Vol. 6275, pp. 123-133.

[258] Kumar, K., Liu, J., Lu, Y.-H., Bhargava, B., “A survey of computation offloading for mobile systems”, Mob. Netw. Appl., Vol. 18, No. 1, Feb. 2013, pp. 129–140, [Online]. Available: http://dx.doi.org/10.1007/s11036-012-0368-0

[259] Piedad, F., Hawkins, M., High Availability: Design, Techniques, and Processes, ser. Enterprise computing series. Prentice Hall PTR, 2001, [Online]. Available: http://books.google.hr/books?id=kHB0HdQ98qYC

[260] Delac, G., Silic, M., Vladimir, K., “Reliability sensitivity analysis for yahoo! pipes mashups”, in Information Communication Technology Electronics Microelectronics (MIPRO), 2013 36th International Convention on, 2013, pp. 851-856.

List of Figures

List of Tables

List of Algorithms

Biography

Goran Delač was born in 1984 in Zagreb, Croatia where he attended primary and secondary school. In 2003 he enrolled the undergraduate study in computer science at the University of Zagreb, Faculty of Electrical Engineering and Computing (FER), during which he was twice awarded the annual best student award "*Josip Lončar*". In 2007, as an intern at Ericsson Nikola Tesla, Zagreb, he participated in the "*SIP/WS Interworking Triggering Gateway*" project that resulted in a patent application. After graduating in 2008, he joined the Department of Electronics, Microelectronics, Computer, and Intelligent Systems at University of Zagreb, FER, as a research assistant. At the same time, he enrolled the doctoral study in computer science at the University of Zagreb, FER under supervision of Professor Siniša Srbljić. He participated in "*Computing Environments for Ubiquitous Distributed Systems* and "*End-User Tool for Gadget Composition*" research projects, supported by the Ministry of Science, Education and Sports of the Republic of Croatia and Google Inc., USA. In 2011, he was granted a software engineering internship at Tuenti Technologies S.L., Madrid, Spain, where he participated in the *MVNO* project aimed at integrating mobile phone and social network services. His research interests span the area of dependable computing, distributed systems, and service oriented computing. Goran Delač is an author of 3 category A journal papers, 1 category B journal paper and 5 conference papers.

List of Publications

Journal Papers

1. Silic, M., Delac, G., Srbljic, S., “Scalable and Accurate Prediction of Availability of Atomic Web Services”, IEEE Transactions on Services Computing, 2013, accepted paper.

2. Delac, G., Budiselic, I., Zuzak, I., Skuliber, I., Stefanec, T., “A Methodology for SIP and

SOAP Integration Using Application-Specific Protocol Conversion", ACM Transactions on the Web, Vol. 6, No. 4, November 2012, pp. 15:1 – 15:28

3. Zuzak, I., Budiselic, I., Delac, G., "A Finite-State Machine Approach for Modeling and Analyzing RESTful Systems", Journal of Web Engineering, Vol. 10, No. 4, December 2011, pp. 353-390

4. Zuzak, I., Budiselic, I., Delac, G., " Formal Modeling of RESTful Systems Using Finite-State Machines", Lecture Notes in Computer Science (Web Engineering, 11th International Conference, ICWE 2011), 6757/2011 (2011), pp. 346-360.

Conference Papers

1. Silic, M., Delac, G., Srbljic, S., "Prediction of Atomic Web Services Reliability Based on k-Means Clustering", Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russia, 2013, pp. 70-80.

2. Delac, G., Silic, M., Vladimir, K., "Reliability Sensitivity Analysis for Yahoo! Pipes Mashups", Proceedings of the 36rd International Convention , MIPRO 2013, pp. 851-856.

3. Delac, G., Silic, M., Srbljic, S., "Reliability Modeling for SOA Systems", Proceedings of the 35rd International Convention , MIPRO 2012, pp. 988-993.

4. Delac, G., Silic, M., Krolo, J., "Emerging Security Threats for Mobile Platforms", Proceedings of the 34rd International Convention , MIPRO 2011 pp. 1468 -14673.

5. Silic, M., Krolo, J., Delac, G., "Security Vulnerabilities in Modern Web Browser Architecture", Proceedings of the 33rd International Convention , MIPRO 2010, pp. 1240-1245.

Životopis

Goran Delač rođen je 1984. godine u Zagrebu gdje pohađa osnovnu i srednju školu. 2003. godine upisuje diplomski studij računarstva na Fakultetu elektrotehnike i računarstva, Sveučilišta u Zagrebu (FER) tijekom kojeg mu je dva puta dodijeljeno priznanje "*Josip Lončar*" za najbolje studente. U sklopu usavršavanja u kompaniji Ericsson Nikola Tesla, Zagreb, 2007. godine sudjeluje u projektu "*SIP/WS Interworking Triggering Gateway*" koji rezultira patentnom prijavom. Diplomirao je 2008. godine nakon čega se zapošljava kao znanstveni novak na Zavodu za elektroniku, mikroelektroniku, računalne i inteligentne sustave, FER. Iste godine upisuje doktorski studij računarstva na Sveučilištu u Zagrebu, FER, pod mentorstvom prof. dr. sc. Siniše Srbljića. Sudjeluje na projektima "*Računalne okoline za sveprisutne raspodijeljene sustave*" i "*End-User Tool for Gadget Composition*" podržane od strane MZOŠ-a i kompanije Google Inc., SAD. U sklopu usavršavanja u kompaniji Tuenti Technologies S.L., Madrid, Španjolska, 2011. godine, sudjeluje na projektu *MVNO*, čiji je cilj bila integracija usluga mobilnog operatera i društvene mreže. Njegovo područje istraživanja obuhvaća oslonjivo računarstvo, raspodijeljeno računarstvo te računarstvo zasnovano na uslugama. Goran Delač autor je 3 rada u časopisima kategorije A, 1 rada u časopisu kategorije B te 5 radova na međunarodnim konferencijama.

Popis objavljenih djela

Rad u časopisima

1. Šilic, M., Delač, G., Srbljić, S., “Scalable and Accurate Prediction of Availability of Atomic Web Services”, IEEE Transactions on Services Computing, 2013, rad prihvaćen za objavljivanje.

2. Delač, G., Budiselić, I., Žužak, I., Skuliber, I., Štefanec, T., “A Methodology for SIP and

SOAP Integration Using Application-Specific Protocol Conversion", ACM Transactions on the Web, Vol. 6, No. 4, Studeni 2012, str. 15:1 – 15:28

3. Žužak, I., Budiselić, I., Delač, G., "A Finite-State Machine Approach for Modeling and Analyzing RESTful Systems", Journal of Web Engineering, Vol. 10, No. 4, Prosinac 2011, str. 353-390

4. Žužak, I., Budiselić, I., Delač, G., " Formal Modeling of RESTful Systems Using Finite-State Machines", Lecture Notes in Computer Science (Web Engineering, 11th International Conference, ICWE 2011), 6757/2011 (2011), str. 346-360.

Rad na konferencijama

1. Šilić, M., Delač, G., Srbljić, S., "Prediction of Atomic Web Services Reliability Based on k-Means Clustering", Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russia, 2013, str. 70-80.

2. Delač, G., Šilić, M., Vladimir, K., "Reliability Sensitivity Analysis for Yahoo! Pipes Mashups", Proceedings of the 36rd International Convention , MIPRO 2013, str. 851-856.

3. Delač, G., Šilić, M., Srbljić, S., "Reliability Modeling for SOA Systems", Proceedings of the 35rd International Convention , MIPRO 2012, str. 988-993.

4. Delač, G., Šilić, M., Krolo, J., "Emerging Security Threats for Mobile Platforms", Proceedings of the 34rd International Convention , MIPRO 2011 str. 1468 -14673.

5. Šilic, M., Krolo, J., Delač, G., "Security Vulnerabilities in Modern Web Browser Architecture", Proceedings of the 33rd International Convention , MIPRO 2010, str. 1240-1245.