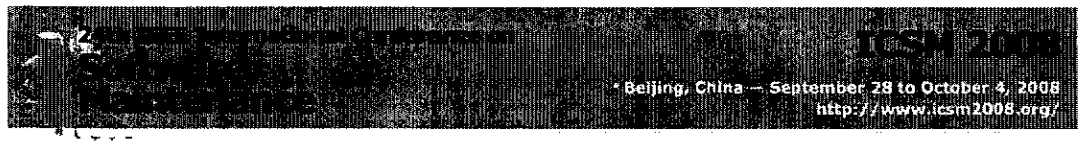
Proceedings of the
2008 IEEE International Conference on Software Maintenance
September 28–October 4, 2008
Beijing, China

IEEE Cat. No. CFP08079-CDR
ISBN 978-1-4244-2614-0
ISSN 1063-6773

Proceedings of the
2008 Frontiers of Software Maintenance
September 30–October 2, 2008
Beijing, China

IEEE Cat. No. CFP0830E-CDR
ISBN 978-1-4244-2655-3
Library of Congress 2008905197



Welcome from the General Chairs

It is our pleasure to welcome you to Beijing, the bustling capital of the People's Republic of China, for the *24th IEEE International Conference on Software Maintenance* (ICSM 2008). ICSM is the premier international event in the software maintenance field for participants from academia, government, and industry to share ideas and experiences for solving critical software maintenance problems.

This is the first time ICSM has been held in China. Beijing is a modern, friendly, world-class city of more than 15 million people, and host to the 2008 Summer Olympics. With numerous scenic and cultural attractions, visiting Beijing and China for ICSM 2008 promises to be an unforgettable experience.

We first started publicizing the ICSM 2008 conference at ICSE 2006 in Shanghai – well over two years ago. Since then we have done our utmost to ensure that as many people as possible knew about ICSM 2008. The badges we created (shown at right) and wore at events around the world for the last few years are a testament to this roving publicity effort. Jens Krinke, Gustavo Rossi, and Qianxiang Wang also worked as Publicity Chairs to raise awareness of the conference.

ICSM 2008 is actually a full week of events. It includes the main ICSM technical and social programs, the two co-located events: the *8th IEEE International Working Conference on Source Code Analysis and Manipulation* (SCAM 2008) and the *10th IEEE International Symposium on Web Site Evolution* (WSE 2008), and several new additions to the main ICSM program: the *Software Technology and Engineering Practice* (STEP) workshops, the *Frontiers of Software Maintenance* (FoSM) track, and the special *Far East Track* sessions.

Putting together a large event like ICSM 2008 takes the sustained effort of a large group of volunteers. First and foremost, we would like to thank the Program Chairs, Hong Mei and Ken Wong, for their excellent technical program. They worked diligently throughout the entire process to solicit submissions, manage the complex review procedure, and carefully select the final proposals for inclusion in the conference program.

We would also like to particularly thank Ken Wong and his team at the University of Alberta for designing and maintaining the excellent ICSM 2008 conference Web site. They kept the site current at all times – in both English and Chinese!

The Finance Chair, Dave Binkley, was a huge help to the conference by managing our finances. This is the first time that ICSM has shared a joint budget and administrative structure with the co-located events, which made an already challenging task even more complicated. Dave also worked closely with Elliot Chikofsky and the Reengineering Forum for the online conference reservation system, which includes handling transactions in multiple currencies between several organizations in different countries.

The team from Peking University, led by Lu Zhang and Qianxiang Wang, superbly handled the local arrangements for the conference. They worked closely with Ingrid Zhang of the Beijing Friendship Hotel and Shihong Huang of Florida Atlantic University to ensure that the hotel

contract was in place, that the meeting facilities were setup and ready, and that the social events were smoothly organized for our enjoyment.

The ICSM 2008 proceedings were more work than usual this year because they also include a separate volume for papers from the new *Frontiers of Software Maintenance* track. We are very appreciative of Hausi Müller for leading the FoSM effort, and for the Proceedings Chair, Andrea De Lucia, for working with Causal Productions and the IEEE Computer Society to ensure that the papers for both ICSM and FoSM were collected, processed, and published on time.

ICSM 2008 is made richer for the participation of our two co-located events: SCAM 2008 (led by Giulio Antoniol) and WSE 2008 (led by Massimiliano Di Penta). Each of these events required considerable organization in their own right. We are pleased that coordinating these efforts to form a unified maintenance week worked so well.

We would be remiss in not thanking several groups that provided the conference with behind-the-scenes guidance. The first is the ICSM Steering Committee, led by Rainer Koschke. The second is the IEEE Computer Society staff, and in particular Tom Baldwin, for their aid in managing all aspects of ICSM 2008. This year there were a great many unique issues that had to be addressed to host the conference in China. We could not have done it without their help.

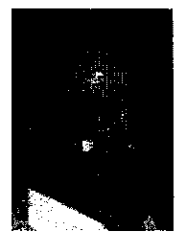
ICSM continues to enjoy the sponsorship of the IEEE Computer Society. This year we were also financially aided by our donors: OW2 Consortium, Beijing Simpleware Tech, and Shandong CVICSE Middleware. All aspects of the conference benefit from the time and effort provided by our supporting organizations: Peking University (China), the Florida Institute of Technology (USA), the University of Alberta (Canada), the China Computer Federation, and Kent State University (USA).

Last but not least, we would like to thank you – the participants at ICSM 2008 who have traveled to China to participate in this groundbreaking event. Enjoy the ICSM 2008 technical and social program in the magnificent atmosphere of Beijing!

Fuqing Yang

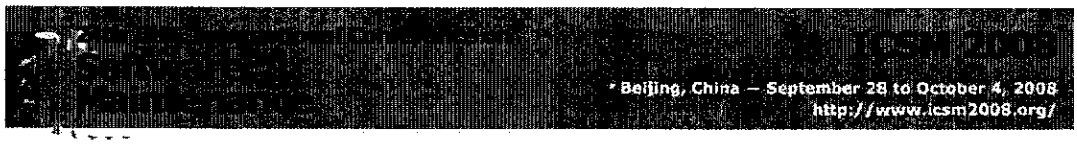
Peking University
China

Scott Tilley



Florida Institute of Technology
USA

General Chairs, ICSM 2008



Welcome from the Program Chairs

The International Conference on Software Maintenance (ICSM) has grown since its start in 1983 to become the premier international forum for researchers and practitioners to address key challenges facing the software maintenance community. With an exciting program scheduled over five days, ICSM 2008 is the centerpiece of a full week of software maintenance events.

ICSM 2008 received 156 research paper submissions from all over the world. Based on 472 reviews from 78 Program Committee members and another 62 external referees, we selected 40 high-quality papers for publication in the proceedings and presentation in 14 balanced technical sessions at the conference. As staple events at ICSM, we also have the Doctoral Symposium to highlight ongoing or recently completed Ph.D. research, the Industry Track with reports of industrial experiences, and Tool Demonstrations to exhibit innovative research prototypes.

Special attractions in the program this year are *Software Technology and Engineering Practice* (STEP) Workshops, the *Frontiers of Software Maintenance* (FoSM) series, the Far East Track, and the Education Session. Three STEP workshops provide superb occasions to discuss research in service-oriented systems, advanced tool construction, and open-source communities. The FoSM series of 16 invited paper presentations gather leading international experts to overview the past, present, and future of key research areas related to software maintenance. The Far East Track adds a distinctive, regional element to ICSM 2008, focusing on software maintenance research, practices, and challenges from Far East Asia. The Education Session offers an informal discussion of issues in training the next generation of software maintainers.

Last, but not least, we are excited to have two keynote addresses. Harry Sneed (University of Regensburg, Germany and ANECON GmbH, Austria), a long-time practitioner and contributor to ICSM, offers his insights on outsourcing software maintenance work. Ji-Feng He (East China Normal University, China), a distinguished leader in software research across China, describes an approach to address testing and maintenance problems using formal methods. Over the years, both of these speakers have made significant strides in the evolution of software systems.

A strong and diverse conference program would not be possible without the combined efforts of many people. We are deeply grateful to all those who contributed. The quality of the papers hinges on a systematic process of paper submission, review, and final editing. Thanks go to all the authors for their valuable contributions across all tracks. Also, we were impressed by the diligent and thoughtful reviews from the Program Committee members and external referees on the research papers. Indeed, they returned every assigned review. We thank them for all their insightful comments and suggestions, on behalf of all the authors as well as ourselves.

The program is elevated by the feats of several track chairs and their reviewers based on 28 submissions. We wish to acknowledge Gerardo Canfora and Kostas Kontogiannis for organizing a splendid Doctoral Symposium, Jochen Hartmann and Vipul Shah for creating an engaging Industry Track, Rudolf Ferenc and Holger Kienle for producing a stimulating set of Tool Demonstrations, Dennis Smith and Ladan Tahvildari for establishing the STEP Workshops, and Liz Burd and Mehmet Orgun for arranging the Education Session. Having these track chairs

spread around four continents posed a coordination challenge, but lends an international perspective.

Invited contributions take an extra effort to recruit participants and organize their papers. We are particularly thankful of Hausi Müller, who was instrumental in receiving most commitments for the FoSM series as early as September 2007, during ICSM 2007 in Paris. His enthusiasm and belief in the maintenance research community made a convincing argument to feature FoSM prominently in the conference with special time slots and a separate volume. We also extend our gratitude to Hongji Yang and Ying Zou for selecting, contacting, and following through with six speakers from across the Far East.

To handle the many paper submissions, we are indebted immensely to Michael Collard. Behind the scenes, his care, responsiveness, guidance, and judgment were key to a smoothly running electronic submission, review, and notification process. He developed several customizations to improve the flexibility and usability of the online system. Also, we acknowledge the Department of Computer Science at Kent State University (USA) for their continuous support of the server on which the system runs.

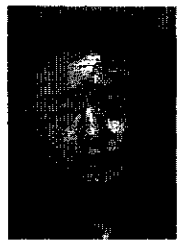
On the conference Web site and publicity materials, we are particularly appreciative of the volunteer work in Chinese translation by Xin Li, Zhenchang Xing, and Dabo Sun at the University of Alberta (Canada) and Shihong Huang of Florida Atlantic University (USA). Their knowledge of software engineering along with Chinese culture was crucial to representing ICSM effectively to research communities in China.

At the conference itself, we wish to thank all those who helped to moderate discussions diplomatically, and keep the various activities running on time, including session chairs, workshop organizers, and again the track chairs.

As Program Chairs, we are involved in many aspects of organizing a conference, involving coordination with chairs in Publicity, Proceedings, Local Arrangements, and Finance. This is only made achievable by General Chairs overseeing and supporting the whole team. In particular, we thank Fuqing Yang for her leadership, and Scott Tilley for his incredible fortitude in making it all work.

Best wishes, enjoy the conference, and have fun in Beijing!

Hong Mei



Peking University
China

Kenny Wong

University of Alberta
Canada

Program Chairs, ICSM 2008

Conference Committee

General Chairs

Fuqing Yang, Peking University, China
Scott Tilley, Florida Institute of Technology, USA

Program Chairs

Hong Mei, Peking University, China
Kenny Wong, University of Alberta, Canada

STEP Workshops and Panels Chairs

Dennis Smith, Carnegie Mellon University / Software Engineering Institute, USA
Ladan Tahvildari, University of Waterloo, Canada

Doctoral Symposium Chairs

Gerardo Canfora, University of Sannio, Italy
Kostas Kontogiannis, National Technical University of Athens, Greece

Industry Track Chairs

Jochen Hartmann, BMW, Germany
Vipul Shah, Tata Consultancy Services, India

Education Track Chairs

Elizabeth Burd, Durham University, UK
Mehmet Orgun, Macquarie University, Australia

Tool Demonstrations Chairs

Rudolf Ferenc, University of Szeged, Hungary
Holger Kienle, University of Victoria, Canada

Frontiers of Software Maintenance Chair

Hausi Müller, University of Victoria, Canada

Far East Track Chairs

Hongji Yang, De Montfort University, UK
Ying (Jenny) Zou, Queen's University, Canada

Publicity Chairs

Jens Krinke, FernUniversität in Hagen, Germany
Gustavo Rossi, National University of La Plata, Argentina
Qianxiang Wang, Peking University, China

Finance Chair

David Binkley, Loyola College in Maryland, USA

Local Arrangements Chair

Lu Zhang, Peking University, China

Cultural Liaison

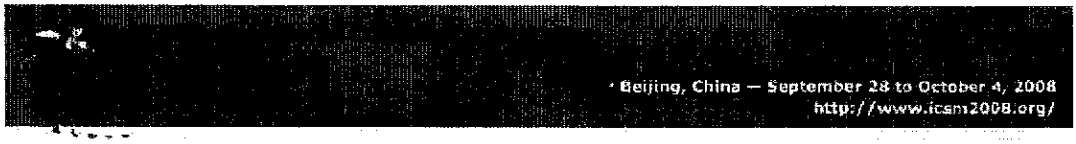
Shihong Huang, Florida Atlantic University, USA

Proceedings Chair

Andrea De Lucia, University of Salerno, Italy

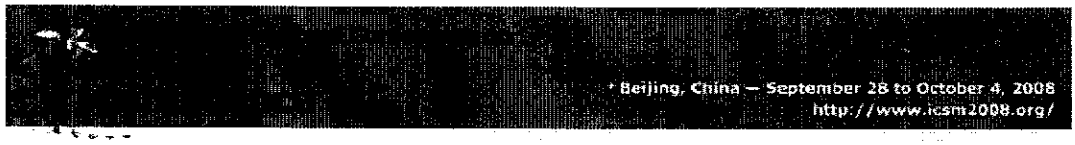
Submissions Chair

Michael Collard, Kent State University, USA



Steering Committee

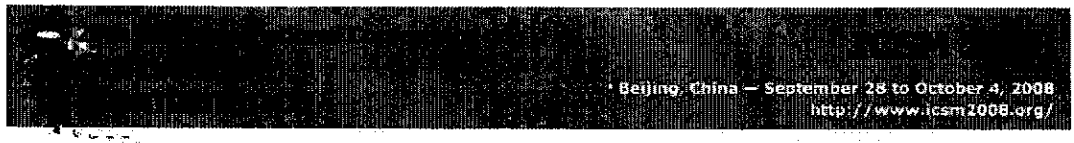
Rainer Koschke, University of Bremen, Germany (Chair)
Andrian Marcus, Wayne State University, USA
Panagiotis Linos, Butler University, USA
Spiros Mancoridis, Drexel University, USA
Harry Sneed, ANECON GmbH, Austria
Paolo Tonella, FBK-irst, Italy



Program Committee

Nicolas Anquetil, École des Mines de Nantes, France
Giuliano Antoniol, École Polytechnique de Montréal, Canada
Teresa Baldassarre, University of Bari, Italy
Francoise Balmas, Université Paris 8, France
Ira Baxter, Semantic Designs, Inc., USA
Dirk Beyer, Simon Fraser University, Canada
David Binkley, Loyola College in Maryland, USA
Marcela Fabiana Genero Bocco, University of Castilla-La Mancha, Spain
Jim Buckley, University of Limerick, Ireland
Yuanfang Cai, Drexel University, USA
Danilo Caivano, University of Bari, Italy
Gerardo Canfora, University of Sannio, Italy
Ned Chapin, InfoSci, Inc., USA
William Cheng-Chung Chu, Tunghai University, Taiwan
Tony Cox, Dalhousie University, Canada
Thomas Dean, Queen's University, Canada
Giuseppe Di Lucca, University of Sannio, Italy
Massimiliano Di Penta, University of Sannio, Italy
Damiano Distante, University of Sannio, Italy
Stéphane Ducasse, INRIA Lille Nord Europe, France
Juan Fernández-Ramil, Open University, UK and University of Mons-Hainaut, Belgium
Harald Gall, University of Zurich, Switzerland
Keith Gallagher, Durham University, UK
Daniel German, University of Victoria, Canada
Tudor Girba, University of Bern, Switzerland
Michael Godfrey, University of Waterloo, Canada
Yann-Gaël Guéhéneuc, Université de Montréal, Canada
Tibor Gyimóthy, University of Szeged, Hungary
Abdelwahab Hamou-Lhadj, Concordia University, Canada
Mark Harman, King's College London, UK
Ahmed Hassan, Queen's University, Canada
Daqing Hou, Clarkson University, USA
Katsuro Inoue, Osaka University, Japan
Stan Jarzabek, National University of Singapore, Singapore
Mira Kajko-Mattson, Stockholm University and Royal Institute of Technology, Sweden
Holger Kienle, University of Victoria, Canada
Sunghun Kim, Massachusetts Institute of Technology, USA
Paul Klint, CWI and University of Amsterdam, Netherlands
Rainer Koschke, University of Bremen, Germany
René Krikhaar, VU University Amsterdam, Netherlands

Jens Krinke, FernUniversität in Hagen, Germany
Yvan Labiche, Carleton University, Canada
Ralf Lämmel, University of Koblenz-Landau, Germany
Michele Lanza, University of Lugano, Switzerland
Mingshu Li, Chinese Academy of Sciences, China
Jian Lu, Nanjing University, China
Atif Memon, University of Maryland, USA
Nabor Mendonça, University of Fortaleza, Brazil
Maurizio Morisio, Politecnico di Torino, Italy
Hausi Müller, University of Victoria, Canada
Tien Nguyen, Iowa State University, USA
Liam O'Brien, NICTA, Australia
Martin Pinzger, University of Zurich, Switzerland
Lori Pollock, University of Delaware, USA
Juergen Rilling, Concordia University, Canada
Kamran Sartipi, McMaster University, Canada
Jelber Sayyad Shirabad, University of Ottawa, Canada
Carolyn Seaman, University of Maryland and Fraunhofer Center Maryland, USA
Dennis Smith, Carnegie Mellon University / Software Engineering Institute, USA
Harry Sneed, ANECON GmbH, Austria
Eleni Stroulia, University of Alberta, Canada
Tarja Systä, Tampere University of Technology, Finland
Ladan Tahvildari, University of Waterloo, Canada
Paolo Tonella, FBK-irst, Italy
Mariella Tortorella, University of Sannio, Italy
Arie van Deursen, Delft University of Technology, Netherlands
Giuseppe Visaggio, University of Bari, Italy
Robert Walker, University of Calgary, Canada
Feng-Jian Wang, National Chiao-Tung University, Taiwan
Andreas Winter, Johannes Gutenberg University of Mainz, Germany
Qing Xie, Accenture Technology Labs, USA
Tao Xie, North Carolina State University, USA
Zhenchang Xing, University of Alberta, Canada
Baowen Xu, Southeast University, China
Hongji Yang, De Montfort University, UK
Andy Zaidman, Delft University of Technology, Netherlands
Jianjun Zhao, Shanghai Jiao Tong University, China
Ying (Jenny) Zou, Queen's University, Canada



Additional Reviewers

Mithun Acharya, North Carolina State University, USA
Pasquale Ardimento, University of Bari, Italy
Arun Bahulkar, Tata Consultancy Services, India
Tibor Bakota, University of Szeged, Hungary
Hamid Abdul Basit, Lahore University of Management Sciences, Pakistan
Ken Bauer, University of Alberta, Canada
Olga Baysal, University of Waterloo, Canada
Alexandre Bergel, INRIA Lille Nord Europe, France
Mario Luca Bernardi, University of Sannio, Italy
Vilmos Bilicki, University of Szeged, Hungary
Thierry Bodhuin, University of Sannio, Italy
Nicola Boffoli, University of Bari, Italy
Amancio Bouza, University of Zurich, Switzerland
Engin Bozdag, Delft University of Technology, Netherlands
Chih-Hung Chang, Hsiuping Institute of Technology, Taiwan
Marta Cimitile, University of Bari, Italy
Azin Dehmoobed, McMaster University, Canada
Simon Denier, Université de Montréal, Canada
Lajos Jenő Fülöp, University of Szeged, Hungary
Tamas Gergely, University of Szeged, Hungary
Adnane Ghannem, Université de Montréal, Canada
Bashar Gharaibeh, Iowa State University, USA
Giacomo Ghezzi, University of Zurich, Switzerland
Salima Hassaine, Université de Montréal, Canada
Lile Hattori, University of Lugano, Switzerland
Abram Hindle, University of Waterloo, Canada
JeeHyun Hwang, North Carolina State University, USA
Patricia Jablonski, Clarkson University, USA
Judit Jász, University of Szeged, Hungary
Priya Jayaratna, McMaster University, Canada
Yue Jia, King's College London, UK
Juanjuan Jiang, Tampere University of Technology, Finland
Zhen Ming Jiang, Queen's University, Canada
Foutse Khomh, Université de Montréal, Canada
Adam Kiezun, Massachusetts Institute of Technology, USA
Patrick Knab, University of Zurich, Switzerland
Heng Boon Kui, National University of Singapore, Singapore
Vinay Kulkarni, Tata Research Development and Design Centre, India
Nuo Li, North Carolina State University, USA
Chih-Wei Lu, Hsiuping Institute of Technology, Taiwan

Francesco Mazzone, University of Sannio, Italy
Tom Mens, University of Mons-Hainaut, Belgium
Ali Mesbah, Delft University of Technology, Netherlands
Rimon Mikhaiel, University of Alberta, Canada
Ravindra Naik, Tata Research Development and Design Centre, India
Javier Perez, University of Valladolid, Spain and University of Mons-Hainaut, Belgium
Nam Pham, Iowa State University, USA
Li Ruan, Chinese Academy of Sciences, China
Anna Ruokonen, Tampere University of Technology, Finland
Haihao Shen, Shanghai Jiao Tong University, China
István Siket, University of Szeged, Hungary
Mike Smit, University of Alberta, Canada
Kunal Taneja, North Carolina State University, USA
Suresh Thummalapenta, North Carolina State University, USA
Niels Veerman, VU University Amsterdam, Netherlands
László Vidács, University of Szeged, Hungary
Harald Wertz, Université Paris 8, France
Richard Wettel, University of Lugano, Switzerland
Shujian Wu, Chinese Academy of Sciences, China
Michael Würsch, University of Zurich, Switzerland
Hua Xiao, Queen's University, Canada
Lizi Xie, Chinese Academy of Sciences, China
Da Yang, Chinese Academy of Sciences, China
Sai Zhang, Shanghai Jiao Tong University, China
Yuzhu Zhao, Chinese Academy of Sciences, China
Lijie Zou, University of Waterloo, Canada

Sponsors

Sponsoring Organizations

IEEE

IEEE Computer Society

TCSE

IEEE Computer Society Technical Council on Software Engineering

In Cooperation With

Reengineering Forum (REF)

Corporate Donors

Shandong CVICSE Middleware, China

OW2 Consortium, China

Beijing Simpleware Tech, China

Supporting Organizations

Peking University, China

Florida Institute of Technology, USA

University of Alberta, Canada

KENT STATE
UNIVERSITY

Kent State University, USA

China Computer Federation, China

Testing-based Assessment Process for Upgrading Component Systems*

Andres Flores
GIISCo Group
Departamento de Ciencias de la Computación
Universidad Nacional del Comahue
Neuquen, Argentina
aflores@uncoma.edu.ar

Macario Polo Usaola
Alarcos Group
Escuela Superior de Informática
Universidad de Castilla-La Mancha
Ciudad Real, Spain
macario.polo@uclm.es

Abstract

Updating component-based systems demand a careful treatment due to stability risks. Replacement components must be analysed on behaviour equivalence for the provided functionality. Our proposal complements the conventional compatibility analysis with black-box testing criteria as a support for substitutability. The aim is to analyse functions of data transformation encapsulated on components, i.e. their behaviour. This is reflected by the Observability testing metric. For a piece under substitution, a Component Behaviour Test Suite is built for being later applied on candidate replacement components. The approach is supported through a tool, testooj, which is focused on testing Java components.

1 Introduction

Maintenance of systems assembled from components (i.e. component-based systems) involves updates by replacing existing pieces with upgrades or totally new components. This entails a highly risky scenario where functioning systems stability can be seriously undermined [19, 33, 7]. The evolutive nature of software and the impact of changes makes substitutability become a serious issue. Whether there can be a certain control on versions of a component, under successive releases changes may be spread across most of the codified functions and structures producing a massive difference with respect to the original component. This is even harder when components are acquired from different vendors, where system integrators cannot control deployment, and do not know if the same environment (e.g. compiler, and compiling options) were used on components that are supposed to be alike, or even on successive releases [24].

Many approaches describe the impact of changes based on *revision numbers* with the pattern '*major.minor.micro*', where the *major* number implies an incompatible change. Well-known component frameworks (DCE, CORBA, OSGi [30, 27, 32]) and package deployment systems [9, 3] comply with this schema. Although very helpful, users cannot completely rely on revision numbers to maintain a system stability. Some component frameworks may have capabilities to make automatic adjustments without affecting the system. Others like Java, which present static type checking and binding, may raise an exception upon a (theoretically) safe-change on the definition type expected by one client's compiled code [5].

The main concern for a system integrator is therefore to identify if new releases or new acquired components can safely replace pieces from a component-based system already deployed and in-use. With that intent, we propose in this paper a Process for Assessing Components Substitutability. That is, to find out if a replacement component can substitute another from a component system.

The compatibility analysis is mainly based on the Observability testing metric [14, 20]. The central idea is to observe the operational behaviour of a component (i.e. its output as a function of its input). Analysing the expected input and output data, and how data is transformed into another, provides a reliable way to compare behaviour from components – i.e. to achieve semantic analysis.

To address this approach, specific testing coverage criteria have been selected in order to design an adequate Test Suite TS as a representation of behaviour for components, viz. a Component Behaviour Test Suite. Such TS is developed for a piece under substitution, to be later exercised on candidate replacements to observe behaviour equivalence.

Automation of the whole Assessment Process is currently supported for the Java framework through a tool, *testooj* [31], from where Test Case generation is done rig-

*This work is supported by UCLM–Indra Software Labs. (Mixed Center of Research and Development) and projects: CyTED–CompetiSoft (506AC0287), UNCo–ISUCSoft (04-E0XX), and UCLM-PRALIN (PAC-08-0121-1374)

ously through automated steps and conditions. Since it is assumed the usual unavailability of component internal aspects (e.g. source code), only interfaces information is considered for the whole process. By means of *introspection* (i.e. reflection on Java and .Net frameworks) interfaces from a component and its candidate replacement are extracted to be used on different phases of the process. The tool additionally integrates well-known testing frameworks like JUnit and MuJava [21, 26], from where the Component Behaviour TS is easily validated in the development phase and later effectively executed against candidate components for compatibility analysis. A .Net version of the same tool has been partially implemented as well, and will be updated to include the remainder phases of the process.

The paper is organised as follows. Section 2 presents an overview of the whole approach. Section 3 describes aspects of the Component Behaviour TS. Section 4 presents the Syntactic Evaluation. Section 5 describes the Testing-based Semantic Evaluation. Section 6 presents results of an experiment. Section 7 presents some related work. Conclusions and future work are presented afterwards.

2 Testing-based Assessment Process for Substitutability

Given a component C and a candidate replacement K, each one accepts a certain input on their services, from where an internal transformation function returns a specific output. Not only input and output should be equivalent on both components, but mainly the right generation of a particular output from a specific input, which is referred to as *functional mapping*. The Observability testing metric [14, 20] is particularly focused on analyzing the functional mapping performed by a component, which helps to understand its behaviour. This may be used to expose a potential compatibility between components – as discussed in [1, 6]. Exploring functional mappings could be extensive, but focusing on certain aspects and representative data result more efficient and is also highly effective. This is basically addressed through a specific selection of testing coverage criteria.

Our proposal of Substitutability Assessment Process consists of three main phases, which are depicted in Figure 1. Being C an original component and K a candidate replacement, the whole process involves the following:

1^{st} Phase. A test suite TS is generated with the purpose to represent behaviour aspects from a component C. This TS complies with certain criteria which help describing different facets of interactions of component C with others components into a software system. Notice that the goal of such TS is not to find faults but to represent behaviour. This will be fully explained in Section 3.

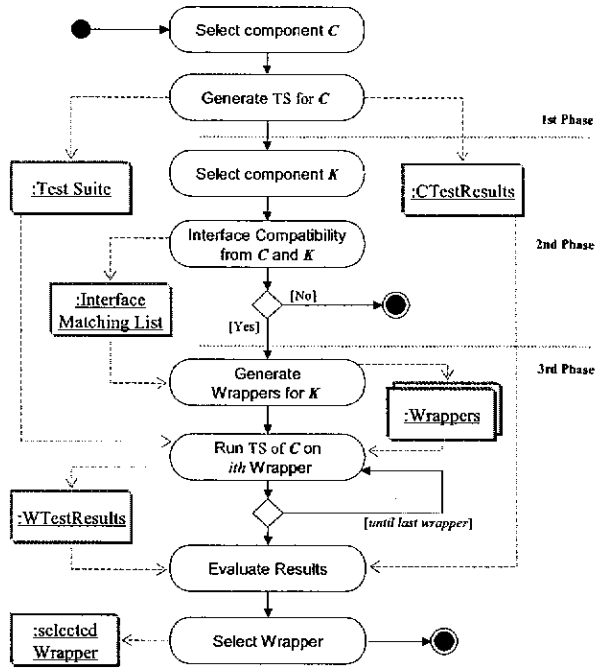


Figure 1. Testing-based Compatibility Approach

2^{nd} Phase. Interfaces offered by C and the candidate K are compared syntactically. If K offers an interface compatible to that of C, then K is passed to the next phase. The analysis considers if the set of services from K contains the services offered by C. At this stage, there can be compatibility even though services from C and K have different names, different order in the parameters, etc. The outcome of this phase is a mapping list where each service from C may have a correspondence with one or more services from K. Details of this phase are given in Section 4.

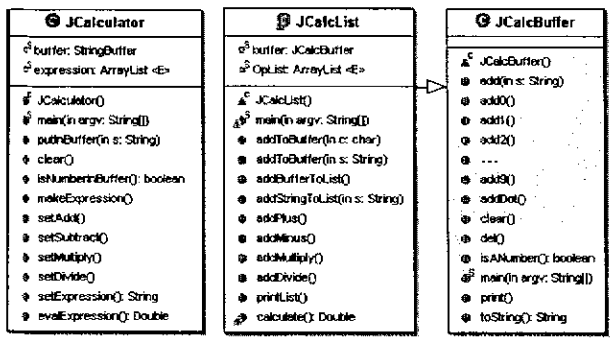
3^{rd} Phase. Component K which has passed the interface compatibility must be evaluated at a semantic level. This implies to execute the TS generated for C in the first phase, against K. The purpose is to find the true service correspondences from the list generated in the second phase. Hence, from such a list is generated a set of wrappers (W) for K. The ultimate goal is to find a wrapper $w_i \in W$ to be placed instead of C to allow current C's clients to safely call the K's interface. To achieve this, each $w_i \in W$ is taken at a time as the target class under test by running the TS from C. After the whole set W has been tested then results from each execution are analysed to expose the degree at which can be concluded that a compatibility has been found. This also implies that at least one wrapper $w_i \in W$ can be selected as the most suitable to allow tailoring K to be integrated into the system as a replacement for C.

Next sections provide detailed information on each step. The application of the process is illustrated by means of the following case study.

2.1 Case Study

The case study is initially based on a Java calculator, `JCalc`, which has been downloaded from *http://sourceforge.net*, and whose main classes are shown in Figure 2(b). Then a new component called `JCalculator` has been created as a variation from `JCalc`, as can be seen in Figure 2(a). For illustrative purposes, `JCalculator` will be considered as the original component, which makes `JCalc` to become a candidate replacement component.

In order to give a conclusive decision on compatibility between `JCalculator` and `JCalc`, the first phase of the Substitutability Assessment Process must be initiated. This involves the creation of a Component Behaviour TS for `JCalculator`, which later will be used for the semantic evaluation. Following is explained how this phase proceeds.



(a) JCalculator (b) JCalc

Figure 2. Original component (a), and replacement (b)

3 Component Behaviour Test Suite

In order to build a Test Suite TS as a behavioural representation of components, specific coverage criteria for component testing has been selected. The goal of this TS is to check that a candidate component *K* coincides on behaviour with a given original component *C*. Therefore, each test case in TS will consist of a set of calls to *C*'s services, from where the testing results are saved on a repository for determining acceptance or refusal when the TS is applied against component *K*. Following are listed some relevant component coverage notions [20], explaining the strategy for their implementation on the approach.

- *all–methods* [16]. Components are accessed through their interfaces, which are composed of a set of methods or services. The criterion requires that every interface must be executed at least once – i.e. every service from each interface is invoked at least once. In [20, 35] this is called *all-interface*.
- *all–events* [36]. An event is an incident in which the resulting effect is the invocation of an interface. Events could be synchronous (e.g. direct calls to services) or asynchronous (e.g. triggering exceptions) [34]. The criterion requires that every event (synchronous or asynchronous) from a component must be covered by some test. Thus this criterion covers *all-exceptions* described in [16, 35].
- *all–context-dependence* [36]. Events can have sequential dependencies on each other causing distinct behaviours according to the order in which they (i.e. services or exceptions) are called. The criterion requires to traverse each operational sequence at least once.
- *all–content-dependence* [36]. An interface may change values that affect behaviour of other(s). This coverage corresponds to a sort of data flow analysis strategy.

In case of *all–content-dependence* two cases apply: *intra-* or *inter-component* interface dependence. That is, dependence either to services inside the same interface or to external services (from interfaces of other components). Inter-component interface dependence requires to design tests with a client and a server component. Similarly for events in case of *all–context-dependence* coverage [35].

Our Component Behaviour TS concerns intra-component dependence, to ease evaluating components without extra environmental requisites. In particular, we implement *all–context-dependence* for which operational sequences are represented by using *regular expressions*, with the alphabet comprised of signatures from components services. This help to describe a general pattern referred to as the "*protocol of use*" for a component interface [22, 28].

Specific coverage criteria for regular expressions has been proposed in [25], which expose the relation with the component coverage criteria previously presented, and mainly explain why regular expressions are an adequate implementation strategy on this approach.

Considering coverage for component service invocation, that is the *all–methods* criterion, a proper coverage is addressed through the *all–alphabets* criterion for regular expressions. For example, the test suite `{abc}` satisfies the alphabets coverage for the regular expression `a*b(b|c)`. Additionally, the set of operators for regular expressions (e.g. `'|'`, `'*'`, etc) help describing every case of service operational sequences. Thus, the so-called *all–operators* criterion is almost equivalent to *all–context-dependence*. However, it is required to force *all–exceptions* to provide coverage for *all–events*. In our approach this is done explicitly. For example the test suite `{bb, abc}` satisfies the operators coverage for the regular expression above.

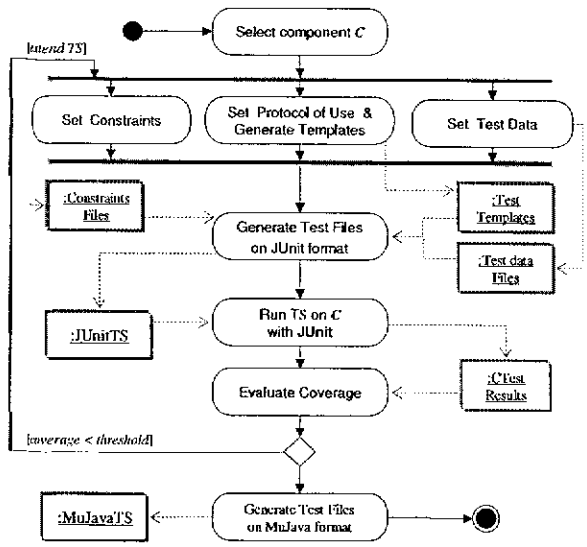


Figure 3. Generation of Component Behaviour Test Suite

Operational sequences can also be derived from Finite State Machines (FSM), which in fact are widely used in the Testing field for representing behaviour and deriving test cases [4, 28, 22]. In our context, service signatures would be represented on edges of an FSM to describe different operational sequences. Of course FSMs can also be used for complex representations of component behaviour with its abstract states and the way they are reached and traversed. Since FSMs can be actually represented by regular expressions, equivalence or subsumes relations can be found on criteria from both notations. For example the FSM's *all–transitions* criterion (called *all–edges* in [25]) subsumes *all–alphabets*. However *all–operators* has been defined as a wider criterion thus subsuming *all–transitions*.

The reflection mechanism of the Java framework allows to extract elements from a component interface to be able to automate Test Case generation. Thus, we may count with service signatures for the alphabet of regular expressions. Also, exceptions extracted from services help to strengthen the representation of components behaviour, which then are used to satisfy the *all–exceptions* criterion. In this way, the regular expression based approach is properly complemented to achieve the *all–context-dependence* criterion.

In fact, some exceptions require the component being on a specific state only reachable after previous executions of other events (e.g. invocations to certain services), therefore operational sequences (context-dependence) are usually the only strategy to get a proper coverage.

By means of the case study presented in the previous section, is following explained how the procedure to build the Component Behaviour TS is carried out, and how it deals with the analysis concerning coverage criteria.

3.1 Test Suite for JCalculator

To build a Component Behaviour TS for `JCalculator`, some steps supported by the *testooj* tool [31] must be done, as can be seen on Figure 3. Test cases can be generated on two formats: JUnit and MuJava [21, 26]. Initially the TS is generated on JUnit format to be validated by its execution against the original component (`JCalculator`). The need is to get 100% successful results since the TS is designed to have configurations of test cases that either do not fail or raise controlled exceptions. Only thus, the TS achieves the goal of representing behavioural facets of the original component. After the TS has been properly validated, then a TS on MuJava format will be derived to be used on the third phase of the process, to ease the involved analysis tasks – this is fully explained on Section 5.1.

One of the initial settings for building the TS, according to Figure 3, implies to set the *protocol of use* (in the form of a regular expression). For `JCalculator` could be as follows.

```
JCalculator putInBuffer [(setAdd | setSubtract |
    setMultiply | setDivide) putInBuffer]+
        setExpression evalExpression
```

The *testooj* tool makes use of the *java.util.regex.Pattern* class to get a set of *test templates* describing operational sequences. They are generated according to the expected length for expressions (sequences) derived from the regular expression. In this case, the minimum length would be 6, which produces four sequences with only one math service, that helps to cover *all–alphabets* criterion. However, to achieve *all–operators* it is required a minimum length of 8, which cause an additional iteration for the `'+'` operator. Thus, 20 templates were finally generated where sixteen of them are comprised of two math services (two iterations).

Next steps involve some other settings, like *test values*, where Figure 4 shows how the values (`1, 2, 3`) were loaded for the only parameter of `putInBuffer` service. They will be used in pairs according to the *protocol of use* (i.e. one value before and after a call to a math service). Figure 4 also shows how to edit *constraints* (assertions) in the pre-/post-code areas, that are later inserted before and after the call to a corresponding selected service. Some reserved words are provided to manipulate the called services. The word *obtained* represents the instance of the component under test (CUT). Arguments for parameters are referenced with *argX* – e.g. *arg1* and *arg2* for the two calls to `putInBuffer`. At the right bottom of Figure 4 is also shown how for an *exception* of a service can be set that it must be thrown with a specific test value. However, no exceptions were modelled for `JCalculator` here.

Test cases on JUnit format require to include an oracle, for which operations of the *Assert* class from the JUnit framework help to check the state of the CUT. In the

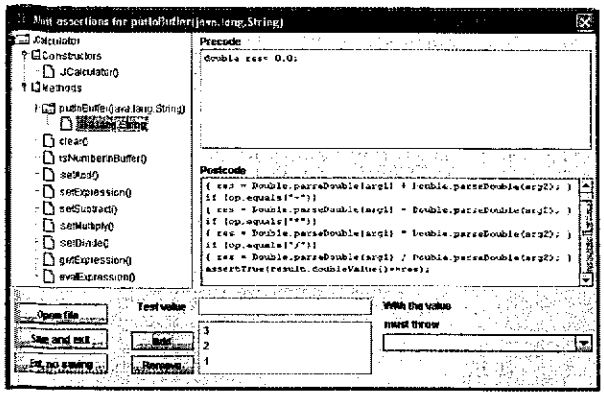


Figure 4. Constraints, Exceptions and Test Values

```
public String testTS_1_1() {
   try {
     JCalculator obtained=null;
     obtained =new JCalculator();
     java.lang.String arg1=(java.lang.String) "3";
     obtained.putInBuffer(arg1);
     String op = "+";
     obtained.setAdd();
     java.lang.String arg2=(java.lang.String) "3";
     obtained.putInBuffer(arg2);
     obtained.setExpression();
     java.lang.Double result=obtained.evalExpression();
     return result.toString();
   }
    catch (Exception e) {
    return e.toString();
   }
}
```

Figure 5. MuJava version of Test Cases for JCalculator

postcode of `evalExpression` service was used *assertTrue* – as shown on Figure 4. After this, *test values* are used in combinations with the 20 *test templates* (operational sequences) and *constraints* files (pre/post-code). Four algorithms are provided by *testooj* to produce the combinations: *each choice* [2], *antirandom* [23], *pairwise* [8], and *all combinations* [18]. The last one was applied in this case study.

Each combination becomes a test case, in the form of a method inside a test driver file. For `JCalculator`, 468 test cases were generated into a class called `JUnitJCalculator`. After this the TS is validated against `JCalculator`, where *testooj* launches the JUnit tool. Test cases are evaluated according to the included *Assert* operation, thus producing a binary result: either success or failure.

The java class `JUnitJCalculator` represents the Component Behaviour TS for `JCalculator` – i.e. the goal to be accomplished on this initial phase. Then a version of the TS on MuJava format can be derived to be used on the third phase of the process. The `MuJavaJCalculator` class was generated with minimal variations: neither pre/post-code is necessary nor oracle (since now the methods return a String). Figure 5 shows the test method `testTS_1_1` on MuJava format, which exercises the `setAdd` math service with the test value 3 on both arguments.

In the following section is explained the second phase of the process, which applies when a candidate replacement component must be integrated into the system.

4 Interface Compatibility

This phase takes place when a component K is considered as a potential replacement for a given component C into a system. This particular evaluation is focused on components interfaces, which are compared at a syntactic level. Four levels are defined for services when comparing interfaces syntactically:

- *Exact Match.* Two services under comparison must have identical signature. This includes service name, return type, and for both parameters and exceptions: amount, type and order.
- *Near-Exact Match.* Similar to previous, though on parameters and exceptions it is relaxed the order into the list, and for service names it is observed likely substrings equivalence.
- *Soft-Match.* Two cases may apply: (1) similar to previous, though service name is ignored and for exceptions it is relaxed to only identify the existence of any; (2) implies subtyping equivalence for return and parameters, and either equality or substring equivalence for service names, and for exceptions: equality of amount, type and order.
- *Near-Soft Match.* Similar to case (1) on the previous level, though considering subtyping equivalence for return and parameters at this level.

Data type equivalence concerns the subsumes relationship or subtyping (written $<:$) [37, 17], which is implemented for built-in types in this approach according to the *direct* subtyping (written $<_1$) from the Java language [17]. Therefore types on services from s_K must have at least as much precision as types on C. For instance for an *int* type, the corresponding type cannot be lower on precision like *short* or *byte* (among numerical types).

The outcome of this step is a matching list characterising each correspondence according to the four cases above. Figure 6 shows algorithms for this step. As can be seen, for each service s_C in C, the algorithm saves a list of services from K which are compatible to s_C. For example, let be C with three services s_{Ci}, $1 \leq i \leq 3$, and K with five services s_{Kj}, $1 \leq j \leq 5$. After the procedure the returning matching list (HashMap) might result as follows:

$\{ (s_{C1}, \{s_{K1}, s_{K2}, s_{K5}\}), (s_{C2}, \{s_{K2}, s_{K4}\}), (s_{C3}, \{s_{K3}\}) \}$

When comparing interfaces, the number of services offered by a candidate component may be equal or greater than the original's. However, it is enforced that every ser-

vice of an original component must have a correspondence in the matching list. Whether a mismatch is found for any original service, the process requires a decision from an integrator. This could be either to provide a manual service matching in order to follow with the process or to stop by concluding the incompatibility of the candidate component.

```
HashMap buildInterfaceCompatibility(Class C, Class K)
   HashTable result= empty
   foreach method_C in C.getMethods()
      Array compatibles= empty;
      result.put(method_C, compatibles);
      loadCompatibleMethods(method_C, result, C, K)
   endForeach
   return result
end

void loadCompatibleMethods (Method method_C, HashMap result,
                            Class C, Class K)
   foreach method_K in K.getMethods()
      Array compatibles = result.get(method_C)
      if exact_match(method_C, method_K)
          compatibles.add("exact", method_K)
      elseif near_match(method_C, method_K, C, K)
          compatibles.add("near", method_K)
      elseif soft_match(method_C, method_K, C, K)
          compatibles.add("soft", method_K)
      elseif near_soft_match(method_C, method_K, C, K)
          compatibles.add("n_soft", method_K)
      endIf
   endForeach
end
```

Figure 6. Interface Matching Algorithms

As can be seen on Figure 6, the algorithms try to find matches initiating with higher compatibility levels to then follow with the weaker ones (i.e. from *exact* to *near-soft*). It is very important to identify strong constrained matches because the outcome of this phase means a pre-analysed knowledge from components under evaluation, which is used as a basis for the next phase to finally get a conclusive result about compatibility.

In an object-oriented framework like Java, there exists a set of methods that are inherited from the *Object* class [17], which are always present (unless inheritance is not considered on the evaluation). These methods may help finding matching when some of them are conveniently overridden, however they usually do not provide interesting aspects to participate on a comparison. Thus, the option could be to omit those methods in a first try. In case no match is found for a given component service, such *Object* methods could then be considered to observe the results of the matching procedure.

4.1 JCalculator-JCalc Interface Matching

When running the Interface Matching between `JCalculator` and `JCalc`, the results reveal that all services from `JCalculator` have found a match – as can be can be seen on Table 1. For example service `putInBuffer` has a *near-exact–match* with service `addToBuffer` (due to the substring equivalence) and also two *soft-matches*, also shown in Figure 7. Another two `JCalculator` services obtained a unique correspondence by means of a *soft-match*, where service `getExpression` has a match with the `toString` service (from the *Object* class). Moreover, four other services obtained a unique *near-exact–match* and three of them also obtained 21 *soft-matches*. Service `setAdd` obtained 6 *near-exact–matches* and 16 *soft-matches*. Service `clear` obtained an *exact-match* and 21 *soft-matches*. The reminder two obtained 22 *soft-matches*.

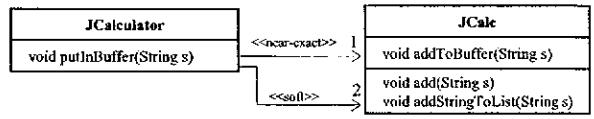


Figure 7. Near-exact and soft-match for `putInBuffer` service

The matching list obtained on this phase gives the chance to discover a potential component compatibility by providing information for the next phase which involves the test-based semantic compatibility.

Exact	Near-Exact	Soft	Near-Soft	(Amount) Services	
1				(7)	getClass, toString, wait, etc.
1	1	20		(2)	notify, notifyAll
1		21		(1)	clear
	1			(1)	isNumberInBuffer
	1	2		(1)	putInBuffer
	1	21		(2)	setMultiply, setDivide
	6	16		(1)	setAdd
		1		(2)	evalExpression, getExpression
		22		(2)	setExpression, setSubtract

Table 1. Summary of Interface Compatibility for JCalculator-JCalc

5 Behaviour Compatibility

This phase does not only may give a differentiation from syntactic similar services, but mainly assures that interface correspondences also match at the semantic level. This means the purpose is finding services from a candidate replacement component that expose a similar behaviour with respect to the original component. In the approach of this paper, this implies to exercise the Component Behaviour Test Suite, generated in the first phase of the process, against the upgrade or replacement component.

The automation of this phase is based on the matching information from the Interface Compatibility analysis, which is used to build the wrapper set W for the candidate replacement. Each wrapper will be a class which can replace the original component, since includes the same interface and even the same class name. A wrapper thus behaves as an

adapter (i.e. an *adapter pattern* [15]) simply forwarding requests to the candidate component. The size of W comes from combinations of services matching. Instead of simply making a blind combination, it is possible to get a reduced amount through the previous syntactic evaluation.

The wrapping approach thus makes use of concerns from *interface mutation* [16, 10] by applying operators to change service invocations and also to change parameter values. The former is done through the list of matching services. The later, by varying arguments on parameters with the same type, on a particular correspondence from the matching list. Nevertheless, the amount of correspondences can be reduced by taking the highest compatibility level obtained in the Interface Compatibility. For instance the `clear` service from the case study with an *exact-match* and other lower compatibility levels, then option is to take only such correspondence and omitting the rest. Thus, this service does not produce additional wrappers.

In summary, the total amount of wrappers is the result from a product of all services from C which have more than one correspondence to K services and those who have parameter correspondences. Whether the size of the wrapper set is too high, an integrator may decide to manually set the correspondences to build only one wrapper, based on the knowledge provided by the Interface Compatibility. For this the *testooj* tool also provides with the ad-hoc utilities. In case no success is obtained with the generated wrapper, another correspondences could be applied, or even decide to change to an automatic building of a bigger set of wrappers.

After building wrappers, the testing step may proceed by taking each wrapper as the target testing component and executing the Component Behaviour TS. Figure 8 shows three steps of the process: (1) how test results are obtained from the original component C, (2) how wrappers of candidate component K are tested against the TS generated for C, and (3) how results from tested wrappers are compared with those from C. Test cases evaluation is done by means of the returned String value. Thus, the evaluation on each test case gives a binary result: either success or failure. The percentage of successful tests from each wrapper determines its acceptance or refusal, that is either killing the wrapper (as a mutation case) or allowing it to survive. The greater the killed wrappers the better, because it might facilitate making decisions on compatibility for the component under evaluation.

5.1 Running JCalculator's TS on JCalc

In order to initiate the Behaviour Compatibility between `JCalculator` and `JCalc` it is required to build the wrappers set W according to the syntactic matching list generated in the second phase of Interface Compatibility. The highest level of compatibility has been then considered for

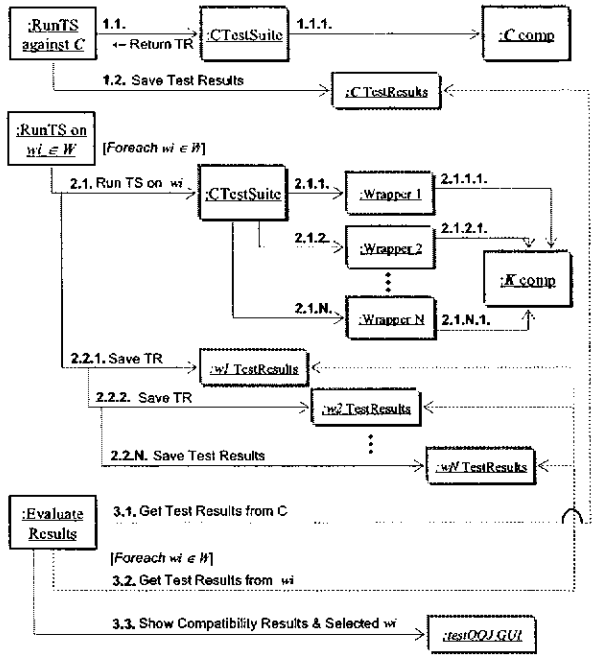


Figure 8. Running TS and Evaluating Results

building the wrappers on this case study. In this case the size of W is $6 * 22 * 22 = 2904$, since only three services from `JCalculator` involved a matching with more than one service from `JCalc`.

After that, the next step is to run the Component Behaviour TS saved on file `MuJavaJCalculator` on each wrapper from W in order to evaluate the semantic compatibility. For this the *testooj* tool provides with an *executor* facility, which is based on the MuJava framework. The executor takes the testing file and iterates through the wrappers list. After this a "*Result Analysis*" utility can show the wrappers that failed the tests when comparing with the original component `JCalculator`. Those failed wrappers correspond to killed mutants, since the application of the *interface mutation* technique. Table 2 shows a summary of results where only one wrapper passed successfully the tests. This means only one wrapper may survive (as a mutation case) which ease to make decisions whether to accept of discard the candidate replacement component – i.e. `JCalc` in this case study.

Although, the range of successful results is quite high, even the highest percentage below 100% (i.e. 77,77%) has a distinguishable difference, which makes easy to recognise that they corresponds to faulty versions of the target wrapper – the one with the true matching, here recognised with the 100%. In case of the wrapper that obtained 77,77% of success, the only wrong matching applied involved the service `setSubtract` to service `del` from `JCalc` – instead

Case	% Success	Wrappers
1	100	1
2	77,77	1
3	75	20
4	72,22	20
5	69,44	6
6	52,77	80
7	50	163
9	8,33	1290
10	0	1323
Total		2904

Table 2. Results of running JCalculator's TS on JCalc

Case	% Success	Wrappers
1	100	1
2	16,66	90
9	8,33	20
10	0	170
Total		281

Table 3. Results of running the 2nd set of wrappers

of `addMinus` which implies the true matching.

The survivor wrapper not only help discovering compatibility between `JCalculator` and `JCalc`, but it also represents the artefact an integrator requires when tailoring the candidate component (`JCalc` in this case study) to be effectively assembled into the system.

5.2 Size Reduction of Wrappers Set

The set of wrappers could grow on size pretty high according to matching cases identified on the Interface compatibility phase. Nevertheless, most of them certainly will correspond to faulty versions of a target wrapper – the one which can describe the true service matching. This means, many wrappers into the set, in fact do not qualify as interesting artefacts to be considered on an evaluation.

For example, the amount of wrappers for the developed case study, could grow over a thousand million when only *soft-matches* are considered. Instead, the wrapper set has a size of only 2904, which is far lower from that hypothetical initial size. This reduces the testing effort and increases performance for the process.

Reduction strategies are applied on the Interface compatibility phase, with the intend to increase the amount of higher levels of compatibility – i.e. *exact-matches* cases or at least *near-exact–matches*. Hence, the practical approach that was implemented involves to analyse service names in order to find substring correspondences – as mentioned on Section 4. In the case study for example, for services `setMultiply` and `setDivide` there is a *near-exact–match* with `addMultiply` and `addDivide` respectively. Even `setAdd` service obtained a less amount of correspondences (6 instead of 22), by using such strategy.

A second group of 281 wrappers has been built concerning *interface mutation* cases not considered on the set of 2904 wrappers. This means, lower compatibility levels were applied this time. The results after running the Component Behaviour TS for `JCalculator` can be seen on Table 3, where one wrapper passed successfully the tests and the rest obtained either zero or a very low percentage of success. This means there is another wrapper which could actually survive (as a mutation case). This second survivor wrapper has the only difference of a matching on `putInBuffer` service to service `add(String s)` (see Figure 7) – which in fact represents a true matching as well, though actually inherited from a superclass in the `JCalc` component.

This second survivor wrapper could pass unrecognised in a normal process when only high compatibility levels are considered. Nevertheless, the important goal is being able to properly recognise a semantic compatibility, which was perfectly achieved with the first set of wrappers, that was even closer to find survivors on most of their members. This second set of 281 wrappers on the contrary, besides the survivor was too far from finding survivors. This means that the first set was based on a stronger basis, which thus expose the importance of the Interface Compatibility procedure.

6 Additional Experiment

Another experiment was carried out to observe the effectiveness of the process. This time, the components were download from the SIR repository [1] [11], which is a public repository intended to be used as a benchmark for testing experiments. The java package `JTopas` was selected, which provides a generic, multi-purpose tokenizer for "readable" text (e.g. source code, HTML, XML, ASCII text), to be integrated into a parser. `JTopas` is also available at *http://jtopas.sourceforge.net*. For this experiment has been selected the `PluginTokenizer` class which represents the main functionality and makes use of the rest of the classes in the package. The whole project of `JTopas` includes 4 versions, from where *version*0 (zero) has been considered as the original component, and the other three versions as the candidate components.

The first phase of the Substitutability Assessment Process was then initiated to build the Component Behaviour TS for *version*0 of `PluginTokenizer`. Together with the project available at the SIR, a test suite on JUnit format is also provided, which was used as a base for learning about the component to develop the corresponding TS. Thus, the initial step for describing the *protocol of use* (to represent operational sequences) has been done to achieve an adequate TS for uncovering the required testing coverage criteria – as discussed on Section 3. As a result from this step, three *test templates* were generated.

[1] The SIR (Software-artifact Infrastructure Repository), http://esquared.unl.edu/sir

The downloaded test suite from the project also provided with a set of *test data*, which consists of 14 HTML files to be "tokenized". These test data were then combined with the three *test templates* to generate a TS comprising 42 test cases (on JUnit format) which was saved on a file called `JUnitPluginTokenizer`.

After that, the TS was run against *version*0 of `PlugInTokenizer`, that is the original component, in order to validate the TS. Since results were successful the next step was to derive a version of the TS on MuJava format which was saved on a file called `MuJavaPluginTokenizer`, to be used on the third phase of the process.

The second phase was then initiated, that is the Interface Compatibility, which gave only *exact-matches* as results. In fact, from *version*1 the reminder versions have a bigger interface than *version*0, which can be perfectly discovered if the order is inverted. Since only *exact-matches* has been found, then only one wrapper need to be generated. The only additional concern may involve those services whose parameter list has a size bigger than one. Some of them include parameters with the same (or equivalent) types, which might be located on a different order (into the parameter list) for different versions. However, since those components correspond to successive versions (i.e. upgrades), the initial assumption is that no such changes have been done to those services – in which there is no externally apparent change. This is even more clear, when the major changes that were observed involve the addition of extra services into the interface.

One wrapper was generated for each of the three remaining versions, from where the execution of the TS gave 100% successful results. Therefore no need of generating other wrappers is required to make a decision on compatibility for the set of upgrades.

Whether hypothetically the generated wrappers would give unsuccessful results, the next option would be among the combinations of parameters for those whose type is identical. Since 4 services involve two alike parameters and 3 others involve six alike parameters, the amount of wrappers by considering that option can actually grow to 3456.

7 Related Work

Regression testing is closely related to our goals, which as explained in [29] generally try to apply reduction strategies on a TS in order to improve efficiency without losing safety – i.e. exposing expected faults on targeted pieces. This is achieved by identifying parts affected by changes on successive versions and recognising "dangerous" testing factors – e.g. paths, transitions, branches, sentences, etc. However, such reduction strategies are based on some knowledge about the changed pieces, that is, source code (white-box) or specifications (black-box). Our approach, on the other hand, assume no existence of other information but the one accessible through the reflection mechanism. Besides, candidate replacements are not assumed to be actual new versions of an original component. Therefore, no identification could be done of changed pieces, which thus expose the usefulness of our approach, which is trying to distinguish behaviour compatibility between an original component and an a priori unknown candidate replacement.

The goals of the work in [24] are very similar to ours. The approach takes an existing TS for a system, which is executed and through a monitoring mechanism component data and interaction models are built dynamically. Later for a component under substitution, those models are used to derive a reduced TS for compatibility purposes. Although the approach seems to be robust, there is still a crucial aspect concerning the adequacy of the initial TS, which could indeed afect reliability of the compatibility analysis. That is why our approach has been conceived with an initial phase focused on the design of a specific TS. Yet, actually any previously developed TS could be used, while evaluating its adequacy. Even a TS designed from specific models could be applied by making minimal adjustments.

Other important related work is summarised in [20] where approaches concerning BIT (Built-in Testing), testable architectures, metadata-based, and user's specification-based testing are properly covered. A main initial difference with those approaches concerns the underlying purpose, which implies to assure a proper component execution. For this most them are based on strategies to find faults. However, our approach has a completely different purpose, far from trying to find faults, the procedures intend to observe a compatibility on behaviour. This is achieved through valid configurations of test cases, i.e. those that do not fail during testing. Even for exceptions the intent is being able to recognise their presence at specific and controlled circumstances.

8 Conclusions

The approach presented in this paper is focused on the maintenance stage where component-based systems require being updated by replacing certain components with new releases (upgrades) or completely different software units (i.e. components from a different vendor). The proposal is a Process for Assessing Components Substitutability which makes use of testing coverage criteria to describe components behaviour with the purpose to analyse compatibility on candidate replacement components. Therefore, this process integrates two aspects: evaluation of compatibility and testing tasks, which therefore reduces effort for system integrators and additionally provides a support on reliability. The *testooj* tool gives automation support for each phase of the process, which helps reducing time and effort and

also reinforces control to achieve a rigorous approach. Our current work concerns test selection for the Component Behaviour TS, for which we are applying prioritization strategies to structure a manageable set of test cases. This can ease understanding on component behaviour, to thus facilitate explaining levels of compatibility.

References

[1] R. Alexander and M. Blackburn. Component Assessment Using Specification-Based Analysis and Testing. Technical Report SPC-98095-CMC, Software Productivity Consortium, Herndon, Virginia, USA, May 1999.

[2] P. Ammann and A. Offutt. Using Formal Methods to derive Test Frames in Category-Partition Testing. In *9th IEEE COMPASS*, pages 69–80, Gaithersburg, MD, USA, 1994.

[3] E. Bailey. *Maximum RPM*. Sams, 1997.

[4] R. Binder. *Testing Object Oriented Systems - Models, Patterns and Tools*. Addison-Wesley, 2000.

[5] P. Brada and L. Valenta. Practical Verification of Component Substitutability Using Subtype Relation. In *32nd EUROMICRO-SEAA*, Dubrovnik, Croatia, August 2006.

[6] A. Cechich and M. Piattini. Early detection of COTS component functional suitability. *Information and Software Technology*, 49(2):108–121, 2007.

[7] A. Cechich, M. Piattini, and A. Vallecillo. *Component-based Software Quality: Methods and Techniques*, volume 2693 of *LNCS*. Springer-Verlag, 2003.

[8] J. Czerwonka. Pairwise Testing in Real World. In *24th PNSQC*, pages 419–430, Portland, OR, US, October 2006.

[9] Debian Project. Debian Packaging Manual, 2007. http://www.debian.org/doc/.

[10] M. Delamaro, J. Maldonado, and A. Mathur. Interface Mutation: An Approach for Integration Testing. *IEEE Transactions on Software Engineering*, 27(3):228–247, March 2001.

[11] H. Do, S. Elbaum, and G. Rothermel. Supporting Controlled Experimentation with Testing Techniques: An infrastructure and its Potential Impact. *Empirical Software Engineering: An International Journal*, 10(4):405–435, 2005.

[12] A. Flores and M. Polo. Towards Software Component Substitutability through Black-Box Testing. In *5th Workshop STV'07, during ICSSEA'07*, pages 111–120, Paris, France, December 2007. Fraunhofer IRB Verlag.

[13] A. Flores and M. Polo. Testing-based Component Assessment for Substitutability. In *10th ICEIS'08*, Barcelona, Spain, June 2008. INSTICC Press.

[14] R. S. Freedman. Testability of Software Components. *IEEE Transactions on Software Engineering*, 17(6):553–564, June 1991.

[15] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[16] S. Gosh and A. P. Mathur. Interface Mutation. *Software Testing, Verification and Reliability*, 11:227–247, 2001. http://www.interscience.wiley.com.

[17] J. Gosling, B. Joy, G. Steele, and G. Bracha. *Java™ Language Specification*. Sun Microsystems, Inc. Addison-Wesley, US, 3rd. edition, 2005. http://java.sun.com/docs/books/jls/third_edition/html/j3TOC.html.

[18] M. Grindal et.al. Combination Testing Strategies: a survey. *Software Testing, Verification and Reliability*, 15(3):167–199, 2005. http://www.interscience.wiley.com.

[19] G. Heineman and W. Council. *Component-Based Software Engineering - Putting the Pieces Together*. Addison-Wesley, 2001.

[20] Jaffar-Ur Rehman, M. et.al. Testing Software Components for Integration: a Survey of Issues and Techniques. *Software Testing, Verification and Reliability*, 17(2):95–133, June 2007. http://www.interscience.wiley.com.

[21] JUnit Home Page. JUnit.org Resources for Test Driven Development, 2008. http://www.junit.org/home.

[22] S. H. Kirani and W.-T. Tsai. Method Sequence Specification and Verification of Classes. *Journal of Object-Oriented Programming*, 7(6):28–38, 1994.

[23] Y. Malaiya. Antirandom Testing: Getting the most out of Black-box Testing. In *IEEE ISSRE*, pages 86–95, Toulouse, France, 1995.

[24] L. Mariani, S. Papagiannakis, and Pezzè. Compatibility and Regression Testing of COTS-component-based software. In *IEEE ICSE*, pages 85–95, Minneapolis, USA, May 2007.

[25] L. Mariani, M. Pezze, and D. Willmor. Generation of Integration Tests for Self-Testing Components. In *Workshop ITM-FORTE*, LNCS 3236, pages 337–350, Toledo, Spain, October 2004. Springer-Verlag.

[26] μJava Home Page. Mutation system for Java programs, 2008. http://www.cs.gmu.edu/ offutt/mujava/.

[27] OMG. CORBA Components - Version 3.0. Technical Report formal/02-06-65, Object Management Group, Inc., June 2002. http://www.omg.org.

[28] OMG. Unified Modeling Language: Superstructure version 2.0. Technical report, Object Management Group, Inc., 2005. http://www.omg.org.

[29] Orso, A. et.al. Using Component Metadata to Regression Test Component-based Software. *Software Testing, Verification and Reliability*, 17:61–94, May 2006. http://www.interscience.wiley.com.

[30] M. Peterson. *DCE: A Guide to Developing Portable Applications*. McGraw-Hill, 1995.

[31] M. Polo, S. Tendero, and M. Piattini. Integrating Techniques and Tools for Testing Automation. *Software Testing, Verification and Reliability*, 16(1):1–37, 2006. http://www.interscience.wiley.com.

[32] The OSGi Alliance. OSGi Service Platform, Release 4, August 2005. http://www.osgi.org/.

[33] B. Warboys et.al. An Active-Architecture Approach to COTS Integration. *IEEE Software*, pages 20–27, July/August 2005.

[34] Y. Wu, M.-H. Chen, and J. Offutt. Uml-based integration testing for component-based software. In *2nd ICCBSS'03*, pages 251–260, Ottawa, Canada, February 2003. LNCS 2580, Springer-Verlag.

[35] Y. Wu, D. Pan, and M.-H. Chen. Techniques of Maintaining Evolving Component-based Software. In *16th IEEE ICSM*, page 236, San Jose, CA, USA, October 2000.

[36] Y. Wu, D. Pan, and M.-H. Chen. Techniques for Testing Component-based Software. In *7th IEEE ICECCS*, pages 222–232, Skovde, Sweden, June 2001.

[37] A. M. Zaremski and J. Wing. Specification Matching of Software Components. *ACM Transactions on Software Engineering and Methodology*, 6(4), October 1997.