Leszek A. Maciaszek
Joaquim Filipe (Eds.)

Communications in Computer and Information Science 410

Evaluation of Novel Approaches to Software Engineering

7th International Conference, ENASE 2012
Warsaw, Poland, June 2012
Revised Selected Papers

Springer

Communications in Computer and Information Science 410

Leszek A. Maciaszek Joaquim Filipe (Eds.)

Evaluation of Novel Approaches to Software Engineering

7th International Conference, ENASE 2012
Warsaw, Poland, June 29-30, 2012
Revised Selected Papers

Springer

Volume Editors

Leszek A. Maciaszek
Wrocław University of Economics, Poland
and
Macquarie University, Sydney, NSW, Australia
E-mail: leszek.maciaszek@mq.edu.au

Joaquim Filipe
INSTICC and IPS, Estefanilha, Setúbal, Portugal
E-mail: joaquim.filipe@estsetubal.ips.pt

ISSN 1865-0929 e-ISSN 1865-0937
ISBN 978-3-642-45421-9 e-ISBN 978-3-642-45422-6
DOI 10.1007/978-3-642-45422-6
Springer Heidelberg New York Dordrecht London

Library of Congress Control Number: 2013955916

CR Subject Classification (1998): D.2, F.3, D.3, H.4, K.6

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

Preface

This CCIS volume contains the papers of the 7th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE) held in Wrocław, Poland, sponsored by the Institute for Systems and Technologies of Information, Control and Communication (INSTICC) and co-organized by the Wrocław University of Economics (WUE).

There is a well-focused research community around the ENASE conferences. The mission of ENASE is to be a prime international forum in which to discuss and publish research findings and IT industry experiences with relation to the evaluation of novel approaches to software engineering. By comparing novel approaches with established traditional practices and by evaluating them against software quality criteria, the ENASE conferences advance knowledge and research in software engineering, identify the most hopeful trends, and propose new directions for consideration by researchers and practitioners involved in large-scale software development and integration.

Overall, for the 7th ENASE in Wrocław we received 54 papers from 27 countries, of which 11 papers were accepted for publication and presentation as full papers. The papers were submitted for blind reviews to 82 renowned ENASE 2012 Program Committee members. With multiple reviews for each paper, the final decision of acceptance/rejection was taken by the PC Chair Prof. Leszek Maciaszek. A few borderline papers were subjected to extra considerations and discussions before decisions were reached.

The ENASE 2012 acceptance ratio for full papers published in this volume was 20.4% (based on 54 submissions), but considering that 9 papers were formally submitted in the category of "position" (rather than "regular") papers, a more truthful ratio for full papers is 24% (based on 45 submissions).

ENASE 2012 was held in conjunction with the 14th International Conference on Enterprise Information Systems ICEIS 2012. The two conferences shared in joint plenary sessions the same list of keynote speakers. The prominent list of keynotes consisted of the following professors:

- Schahram Dustdar, Vienna University of Technology, Austria
- Dimitris Karagiannis, University of Vienna, Austria
- Steffen Staab, University of Koblenz-Landau, Germany
- Pericles Loucopoulos, Loughborough University, UK
- Yannis Manolopoulos, Aristotle University, Greece

September 2013

Joaquim Filipe
Leszek Maciaszek

Organization

Conference Chair

Joaquim Filipe	Polytechnic Institute of Setúbal/INSTICC, Portugal

Program Chair

Leszek Maciaszek	Wroclaw University of Economics, Poland/Macquarie University ∼ Sydney, Australia

Organizing Committee

Helder Coelhas	INSTICC, Portugal
Andreia Costa	INSTICC, Portugal
Patrícia Duarte	INSTICC, Portugal
Bruno Encarnação	INSTICC, Portugal
Raquel Pedrosa	INSTICC, Portugal
Vitor Pedrosa	INSTICC, Portugal
Cláudia Pinto	INSTICC, Portugal
Susana Ribeiro	INSTICC, Portugal
Pedro Varela	INSTICC, Portugal

Program Committee

Guglielmo De Angelis, Italy
Colin Atkinson, Germany
Costin Badica, Romania
Ghassan Beydoun, Australia
Maria Bielikova, Slovak Republic
Piotr Bubacz, Poland
Dumitru Burdescu, Romania
Wojciech Cellary, Poland
Rebeca Cortazar, Spain
Massimo Cossentino, Italy
Bernard Coulette, France
Marcelo d'Amorim, Brazil
Philippe Dugerdil, Switzerland
Angelina Espinoza, Spain
Joerg Evermann, Canada
Maria João Ferreira, Portugal
Agata Filipowska, Poland
Maria Ganzha, Poland
Juan Garbajosa, Spain
Alessandro Garcia, Brazil
Cesar Gonzalez-Perez, Spain
Jeff Gray, USA
Hans-Gerhard Gross, The Netherlands
Rene Hexel, Australia
Emily Hill, USA
Benjamin Hirsch, UAE
Charlotte Hug, France
Bernhard G. Humm, Germany

Zbigniew Huzar, Poland
Akira Imada, Belarus
Stefan Jablonski, Germany
Slinger Jansen, The Netherlands
Monika Kaczmarek, Poland
Robert S. Laramee, UK
Xabier Larrucea, Spain
George Lepouras, Greece
Francesca Lonetti, Italy
Pericles Loucopoulos, UK
Graham Low, Australia
Jian Lu, China
André Ludwig, Germany
Ivan Lukovic, Serbia
Leszek Maciaszek, Poland/Australia
Lech Madeyski, Poland
Leonardo Mariani, Italy
Sascha Mueller-Feuerstein, Germany
Johannes Müller, Germany
Anne Hee Hiong Ngu, USA
Andrzej Niesler, Poland
Janis Osis, Latvia
Marcin Paprzycki, Poland
Oscar Pastor, Spain
Dana Petcu, Romania
Naveen Prakash, India
Lutz Prechelt, Germany
Elke Pulvermueller, Germany
Rick Rabiser, Austria
Lukasz Radlinski, Poland
Gil Regev, Switzerland
Artur Rot, Poland
Radoslaw Rudek, Poland
Francisco Ruiz, Spain
Krzysztof Sacha, Poland
Motoshi Saeki, Japan
Sreedevi Sampath, USA
Heiko Schuldt, Switzerland
Onn Shehory, Israel
Jerzy Surma, Poland
Jakub Swacha, Poland
Stephanie Teufel, Switzerland
Rainer Unland, Germany
Olegas Vasilecas, Lithuania
Krzysztof Wecel, Poland
Michael Whalen, USA
Igor Wojnicki, Poland
Kang Zhang, USA

Auxiliary Reviewers

Fabiano Ferrari, Brazil
Stefan Hanenberg, Germany
Reinhold Kloos, Germany
Ioan Lazar, Romania
Otavio Lemos, Brazil

Invited Speakers

Schahram Dustdar	Vienna University of Technology, Austria
Dimitris Karagiannis	University of Vienna, Austria
Steffen Staab	University of Koblenz-Landau, Germany
Pericles Loucopoulos	Loughborough University, UK
Yannis Manolopoulos	Aristotle University, Greece

Table of Contents

Papers

Papers

Dynamic Symbolic Execution Guided by Data Dependency Analysis for High Structural Coverage

TheAnh Do, A.C.M. Fong, and Russel Pears

School of Computing and Mathematical Sciences, Auckland University of Technology, Auckland, New Zealand
{theanh.do,afong,russel.pears}@aut.ac.nz

Abstract. Dynamic symbolic execution has been shown to be an effective technique for automated test input generation. When applied to large-scale and complex programs, its scalability however is limited due to the combinatorial explosion of the path space. We propose to take advantage of data flow analysis to better perform dynamic symbolic execution in the context of generating test inputs for maximum structural coverage. In particular, we utilize the chaining mechanism to (1) extract precise guidance to direct dynamic symbolic execution towards exploring uncovered code elements and (2) meanwhile significantly optimize the path exploration process. Preliminary experiments conducted to evaluate the performance of the proposed approach have shown very encouraging results.

Keywords: Dynamic Symbolic Execution, Automated Test Input Generation, Software Testing, Data Flow Analysis.

1 Introduction

Testing is a widely adopted technique to ensure software quality in software industry. For about 50% of the total software project costs are devoted to testing. However, it is labour-intensive and error-prone. An attempt to alleviate those difficulties of manual testing is to develop techniques to automate the process of generating test inputs. For over the last three decades, techniques have been proposed to achieve this goal, ranging from random testing [5, 18, 19], symbolic execution [3, 10, 15, 23], search-based testing [16], the chaining approach [12], to dynamic symbolic execution [13, 21, 8].

Among these techniques, dynamic symbolic execution has been gaining a considerable amount of attention in the current industrial practice [9]. It intertwines the strengths of random testing and symbolic execution to obtain the scalability and high precision of dynamic analysis, and the power of the underlying constraint solver. One of the most important insights of dynamic symbolic execution is the ability to reduce the execution into a mix of concrete and symbolic execution when facing complicated pieces of code, which are the critical obstacle to pure symbolic execution. While effective, the fundamental scalability issue of dynamic symbolic execution is how to handle the combinatorial explosion of the path space, which is extremely large or infinite in sizable and complex programs. Dynamic symbolic execution therefore, if performed in a way to exhaustively explore all feasible program paths, often ends up

L.A. Maciaszek and J. Filipe (Eds.): ENASE 2012, CCIS 410, pp. 3–15, 2013.

with small regions of the code explored in practical time, leaving unknown understanding about the unexplored.

In fact, covering all feasible paths of the program is impractical. Besides, testing large programs and referring to sophisticated criteria can often be out of the limit of a typical testing budget. In the practice of software development, therefore, high code coverage has been long advocated as a convenient way to assess test adequacy [20, 6]. Specifically, the testing process must ensure every single code element in the program is executed at least once. In this context, dynamic symbolic execution can be conducted so as to cover all code elements rather than exploring all feasible program paths. This may lead to a significant reduction in the number of paths needed to explore. However, the question of "*how can we derive precise guidance to perform dynamic symbolic execution towards achieving high code coverage?*" becomes important. This question emphasizes two aspects: high coverage achievements and minimal path explorations. The second aspect is essential in the sense that the cost of performing dynamic symbolic execution is expensive, especially in large programs, any technique helping achieve high code coverage must optimize path explorations to be applicable within resources available e.g., CPU, memory and time.

To answer this question, we propose to apply data flow analysis to better perform dynamic symbolic execution in the test input generation process. Particularly, we utilize the chaining approach [12] to pull out precise guidance in order to direct dynamic symbolic execution towards *effectively* and *efficiently* exploring code elements. Specifically, given a test goal (an unexplored code element e.g., statement or branch), the chaining approach first performs data dependency analysis to identify statements that affect the execution of the test goal, and then it uses these statements to form sequences of events that are to be executed prior to the execution of the test goal. The advantage of doing this is twofold: (1) it precisely focuses on the cause of getting the test goal to be executed and (2) it slices away code segments that are irrelevant to the execution of the test goal. As we will show in the evaluation, these two strengths enable dynamic symbolic execution to achieve higher code coverage and at the same time significantly optimize the number of path explorations required to unclose high-complexity code.

The paper is organized as follows. Section 2 introduces dynamic symbolic execution and highlights the path space explosion problem. Section 3 provides a brief survey of related work. Section 4 illustrates the chaining approach. Section 5 explains the prototype implementation and discusses the experimental study. We discuss research issues and future work in Section 6, and conclude the paper in Section 7.

2 Dynamic Symbolic Execution

The key idea behind dynamic symbolic execution [13] is to start executing the program under test with concrete values while gathering symbolic predicates of corresponding symbolic values along the execution. By negating one symbolic predicate and solving the path constraint with an off-the-shelf constraint solver, it can obtain a new input to steer the execution along an alternative program path. This process is often performed in an attempt to exhaustively and systematically explore all feasible paths of the program. Dynamic symbolic execution hence outperforms "classical"

symbolic execution through being able to simplify complex constraints, and deal with complex data structures and native calls of real world programs.

To perform dynamic symbolic execution, code of the program is instrumented in a way that concrete execution can be executed simultaneously with symbolic execution. So, while the former drives the execution, the latter maintains a symbolic memory *S*, which maps memory addresses to symbolic expressions, and a symbolic path constraint *PC*, which is a first-order quantifier-free formula over symbolic expressions. In this way, once an expression is evaluated, it is evaluated both concretely and symbolically, and both physical memory and symbolic memory are updated accordingly. Similarly, once a conditional statement *if (e) then S1 else S2* is executed, *PC* is updated according to the "*then*" or "*else*" branch taken. If the "*then*" branch is taken, *PC* becomes $PC \wedge \sigma(e)$; otherwise, it is $PC \wedge \neg\sigma(e)$, where $\sigma(e)$ denotes the symbolic predicate obtained by evaluating e in symbolic memory. As a result, the symbolic path constraint *PC* presenting a symbolic execution of the program is as follows:

$$PC = \sigma_1 \wedge \ldots \wedge \sigma_{i-1} \wedge \sigma_i \wedge \sigma_{i+1} \wedge \ldots \wedge \sigma_n \quad (1)$$

Every single symbolic predicate of *PC* represents one possibility to execute the program along an alternative path. That is one can randomly pick up a predicate, e.g., σ_i, negate it, and then form the constraint system $(\sigma_1 \wedge \ldots \wedge \sigma_{i-1} \wedge \neg\sigma_i)$ to be solved by the underlying constraint solver. The satisfiability of the constraint system results in an input that the execution of the program with this input will follow the previous path up to the corresponding conditional statement of the negated predicate, but afterwards change the flow of control to the other branch. Consequently, for every symbolic path constraint, the number of program paths can be 2^n or be exponential in the number of symbolic predicates.

```
Node   typedef enum {false, true} bool;
       #define N 20
(s)    bool CheckArray(int A[N]) {
         int i;
(1)      bool success = true;
(2)      for (i = 0; i < N; i ++) {
(3)        if (A[i] != 25)
(4)          success = false;
         }
(5)      if (success) {
(6)        // target
         }
(e)    }
```

Fig. 1. The *CheckArray* function checks if all elements of an input array equal 25

In practice, the number of symbolic predicates of the program is often extremely large (or even infinite), especially in the presence of loops and/or recursions, causing dynamic symbolic execution to face with the combinatorial explosion of the path space. The *CheckArray* function in Figure 1 could be a good example to illustrate this phenomenon. It takes input an array of 20 elements and check if all elements equal 25. This yields 2^{20} (=1,048,576) paths with just 20 symbolic predicates. In practice,

this problem becomes worse as the input of programs can be a stream of data with too large (or unknown) size. In case of *check_ISBN* and *check_ISSN* in the set of test subjects, for instance, both functions take an array with (4093+3) tokens, which give rise to approximately $2^{(4093+3)}$ paths, making dynamic symbolic execution ill-suited for the goal of covering all code elements of programs. It is therefore necessary to devise appropriate search strategies to guide dynamic symbolic execution to achieve high code coverage within a minimal testing budget. In the next section, we provide a brief survey of related test input generation techniques based on dynamic symbolic execution to assess the current state of research.

3 Related Work

Cadar et al. give a preliminary assessment of the use of (modern) symbolic execution in academia, research labs and industry in which the authors emphasize "*A significant scalability challenge for symbolic execution is how to handle the exponential number of paths in the code*" [9]. In the context of using dynamic symbolic execution to generate test inputs for maximum code coverage, tools being in favour of depth-first explorations such as DART [13] and CUTE [21] deeply widen the program path space but lack the ability to forward the execution to further unexplored control flow points. These approaches when executed against large programs for finite time achieve very low code coverage. Pex [22] is an automated structural testing tool developed at Microsoft Research. It combines a rich set of basic search strategies and gives a fair choice among them. While the combination helps maximize code coverage through attempting different program control flows, discovering code elements may require specific guidance of control and data flow analysis. Fitnex [25] further makes Pex more guided by using fitness functions to measure the improvement of the path exploration process. The main obstacle of this approach is the flag problem [4], where fitness functions face a flat fitness landscape, giving no guidance to the search process. Flags, however, are widely used in real world software [4]. CREST [7] is an extensible platform for building and experimenting with heuristics for achieving high code coverage. Among search heuristics implemented in CREST, CfgDirectedSearch is shown more effective than the others through the reported experimental data. This search strategy leverages the static control flow of the program under test to guide the search down short static paths to unexplored code. Theoretically, the control flow guidance may be imprecise since the execution of code elements may require data dependencies going beyond short paths and/or being calculated in dynamic paths. In fact, when testing large-scale programs, the intertwinement of control flow guidance and random branch selection is more efficient than the control flow guidance alone in terms of coverage improvements and performance optimizations [11].

Obviously, with sizable and complex programs, the difficulty of using dynamic symbolic execution to generate test inputs for maximum code coverage is how to mine for appropriate paths to guide the search process towards exposing unexplored code elements among the many possible program paths. In the next section, we introduce the chaining mechanism in an attempt to address this issue.

4 The Chaining Approach

The chaining approach [12] was proposed to make use of data dependency analysis to guide the search process. The basic idea is to identify statements leading up to the goal structure, which may influence the outcome of the test goal. Those statements are sequences of events that the search process must walk along to target the test goal. The chaining approach can hence be considered as a slicing technique [24, 14] which simplifies programs by focusing on selected aspects of semantics. As a result, the chaining approach can provide precise guidance since it forces the consideration of data flows, and it is effective since it slices away irrelevant code segments to the execution of the test goal. These two strengths can guide the search process into potentially unexplored but promising areas of the path space to unclose high-complexity code.

We illustrate the core of the chaining mechanism using again the function *Check-Array* in Figure 1 in which the test goal is to cover branch (5, 6). For this, the chaining mechanism first generates the following initial event sequence $E_0 =<(s, \emptyset), (6, \emptyset)>$ where each event is a tuple $e_i = (n_i, S_i)$ where n_i is a program node and S_i is a set of variables referred to as a constraint set. Now suppose that the search process fails to find an input array with all elements equal 25 to execute the target branch, moving from node 5 to node 6. Node 5 is hence considered to be a *problem node*. Formally, a problem node refers to a conditional statement for which the search process within a fixed testing budget cannot find inputs to execute an intended branch from this node. The chaining mechanism then performs data flow analysis in respect of this problem node to identify statements that define data for variables used in the conditional expression. In this case, the conditional expression consists of variable *success*, which is defined at nodes 1 and 4. Two event sequences are constructed accordingly, E_1 and E_2, based on the initial event sequence.

$$E_1 =<(s, \emptyset), (1, \{success\}), (5, \emptyset), (6, \emptyset)>$$

$$E_2 =<(s, \emptyset), (4, \{success\}), (5, \emptyset), (6, \emptyset)>$$

Notice that for every two adjacent events in an event sequence, $e_i = (n_i, S_i)$ and $e_{i+1} = (n_{i+1}, S_{i+1})$, there must exist a path from n_i to n_{i+1} along which all variables in S_i are not redefined. Such a path allows the effect of a definition statement to be transferred up to the target structure. Obviously, the sequence E_2 cannot help to explore the test goal as the value of *success* variable is *false*, which leads to the execution of the "*else*" branch instead. The event sequence E_1, on the other hand, guides the search process to first reach node 1 from the function entry, which sets the value of *success* variable to the desired *true* value to explore branch (5, 6), and then continues from node 1 to node 5. When moving to node 5, the value of *success* variable may be killed at node 4 if branch (3, 4) is executed. If so, the search process is guided to change the flow of control at node 3 to execute the "*else*" branch, which prevents *success* variable from being set to the unwanted *false* value. This guidance is continuously refined throughout the *for* loop to preserve the constraint set {*success*} of event (1, {*success*}) while reaching to event (5, Ø). By doing so, the value of all elements in the input array is altered to 25, providing the desired input to expose the test goal, branch (5, 6).

We now formalize the process of creating a new event sequence from an existing sequence E. Let E = <e_1, …, e_{i-1}, e_i, e_{i+1}, …, e_m> be an event sequence. Suppose the search process driven by this event sequence guides the execution up to event e_i and a problem node p is encountered between events e_i and e_{i+1}. Let d be a definition statement of problem node p. Two events are generated, e_p = (p, Ø) and e_d = (d, D(d)), corresponding to the problem node and its definition. A new event sequence is now created by inserting these two events into sequence E. Event e_p is always inserted between e_i and $_{ei+1}$. However, event e_d, in general, may be inserted in any position between e_1 and e_p. Suppose the insertion of event e_d is between events e_k and e_{k+1}. The following event sequence is then created:

$$E' = <e_1, \ldots, e_{k-1}, e_k, \mathbf{e_d}, e_{k+1}, \ldots, e_{i-1}, e_i, \mathbf{e_p}, e_{i+1}, \ldots, e_m>$$

Since new events are added to the sequence, the implication of data propagation may be violated. This requires modifications of the associated constraint sets of involved events. The update is done in the following three steps:

(1) $S_d = S_k \cup D(d)$

(2) $S_p = S_i$

(3) $\forall j, k + 1 \leq j \leq i, S_j = S_j \cup D(d)$

In the first step, the constraint set S_d of event e_d is initialized to the union of D(d) and the constraint set of the preceding event e_k. This modification ensures that the constraint set S_k of event e_k is preserved up to event e_{k+1} while getting through the newly inserted event e_d. The second step also imposes the same requirement on event e_p by assigning S_i to its constraint set. In the final step, all constraint sets of events between e_{k+1} and e_i are modified by including a variable defined at d. By doing this, the chaining mechanism guarantees to propagate the effect of the definition at node d up to the problem node p.

Created event sequences may be organized in the form of a tree referred to as a *search tree*. The initial event sequence E_0 represents the root of the tree. Other levels of the tree are formed by event sequences created when problem nodes are encountered. One tree node represents a possibility to unclose the test goal. The search process when following an event sequence attempts to adjust the execution to move from one event to another without violating the constraint set in the previous event. This suggests a systematic mechanism to propagate the effect of all possible data facts up to the target goal structure. When intertwined with dynamic symbolic execution, this search process specifically involves picking up relevant symbolic predicates, negating them, and then forming constraint systems to be solved by the underlying constraint solver for inputs. The back and forth interaction between the search process and dynamic symbolic execution is continuously refined till a desired test input is found to expose the test goal. Dynamic symbolic execution directed by the precise guidance of the chaining mechanism hence can potentially enhance the test input generation process for both coverage improvements and program exploration optimizations. In the next section, we describe a prototype implementation and detail a preliminary evaluation to properly assess these capabilities of our proposed approach.

5 Experimental Study

5.1 Prototype Implementation

We have implemented our proposed approach as a search heuristic, called STIG (Scalable Test Input Generation), on the CREST platform [7], an automatic test input generation tool for C, based on dynamic symbolic execution. Since CREST unrolls decisions with multiple conditions as an equivalent cascade of single condition decisions and converts every single conditional statement into a form of *if (e) then S1 else S2*, branch coverage achieved by CREST is comparable to condition coverage in the original program. For any specification clothed in forms of assertion calls, i.e., *assert (e)*, we transform it into a conditional statement *if (!e) error()* to check for errors. In this case, the test goal is to explore the function call *error*.

Table 1. Percentage of branch coverage achieved by search strategies on 15 test subjects

Subject	loc	br	Random	DFS	CREST	Pex	Fitnex	STIG
sample	31	12	42	92	92	92	92	100
testloop	17	8	13	88	88	100	100	100
hello_world	37	32	34	56	91	91	91	100
valves_nest2	32	12	**17**	**42**	**42**	100	100	100
valves_nest5	62	30	**7**	**17**	**17**	100	100	100
valves_nest10	112	60	**3**	**8**	**8**	100	100	100
valves_nest20	212	120	**2**	**4**	**4**	80	100	100
valves_nest50	512	300	**1**	**2**	**2**	**39**	**58**	100
valves_nest100	1012	600	**0**	**1**	**1**	**20**	**25**	100
netflow	28	6	83	83	83	100	100	100
moveBiggestInFront	37	6	100	83	83	100	100	100
handle_new_jobs	37	6	67	100	100	83	100	100
update_shps	52	10	60	90	90	100	100	100
check_ISBN	78	52	83	83	83	96	83	98
check_ISSN	78	52	83	83	94	96	83	98
Average	156	87	40	56	59	87	89	**100**$^{\approx}$

Notice: *loc* number of lines of code, *br* number of branches
Random random input search, *DFS* depth-first search

5.2 Experimental Setup

To evaluate the effectiveness of our proposed approach, we chose a set of test subjects and conducted experiments to compare STIG with two widely adopted search strategies, random input search and depth-first search, and with three test input generation tools CREST [7], Pex [22] and Fitnex [25]. For CREST, we chose the control-flow graph directed search strategy (CfgDirectedSearch) which was confirmed the "best" search algorithm through the reported experimental data. The test input generation CREST tool used in the whole paper thus refers to this chosen search strategy. For Pex, it implements dynamic symbolic execution to generate test inputs for .NET code, supporting languages C#, VisualBasic, and F#. Besides, Fitnex is an extension of Pex.

Table 2. Measurements of number of program explorations performed by search strategies

Subject	Random	DFS	CREST	Pex	Fitnex	STIG
sample	1000	1000	1000	1000	1000	13
testloop	1000	1000	1000	26	27	22
hello_world	1000	1000	1000	1000	1000	55
valves_nest2	_	1000	1000	92	71	11
valves_nest5	_	1000	1000	236	125	26
valves_nest10	_	1000	1000	338	741	51
valves_nest20	_	1000	1000	1000	689	101
valves_nest50	_	1000	1000	1000	1000	251
valves_nest100	_	1000	1000	1000	1000	501
netflow	1000	2	1000	5	5	2
moveBiggestInFront	15	1000	1000	4	4	2
handle_new_jobs	1000	2	2	1000	99	34
update_shps	1000	1000	1000	5	7	4
check_ISBN	1000	1000	1000	**234**	**313**	51
check_ISSN	1000	1000	1000	**234**	**313**	45
Average	891	867	934	478	426	**78**

Notice: *Random* random input search, *DFS* depth-first search

The test subjects selected include the *sample*, *testloop*, and *hello_world* functions. The first function was adopted from the work of Ferguson and Korel [12] while the last two functions were adopted from the work of Xie et al. [25]. The next six test subjects, *valves_nest<i>* with *i = {2, 5, 10, 20, 50, 100}*, were from the work of Baluda et al. [2]. These test subjects were employed in literature to illustrate the essence of individual exploration problems of dynamic symbolic execution. Thus we want to check if STIG by using data flow analysis is able to tackle those exploration problems. In particular, for the *valves_nest<i>* programs where, as reported [2], DFS, CREST and KLEE [8] all revealed their worst performance and scored very low coverage, we expected STIG with the ability to precisely identify the root cause of the execution of hard-to-reach code elements could potentially guide the search process to achieve maximum coverage within minimal exploration effort. Notice that the *hello_world* function in this evaluation was modified to check an input array which must start with "Hello", end with "World!" and contain only spaces. This modification makes the function more difficult for search strategies to cover all its code coverage. The rest of the test subjects were mentioned in the work of Binkley et al. [4]. These functions come from open-source programs, we hence want to evaluate the capability of STIG in dealing with the high complexity of real world programs as compared to the others'. For the sake of experiments, for some functions, we just extracted part of their code. All the test subjects are in C code, to make comparison with Pex and Fitnex, we converted them to C# code.

All experiments in the evaluation were run on 3GHz CoreTM2 Duo CPU with 4GB of RAM and running Ubuntu GNU/Linux for Random, DFS, CREST and STIG, and Windows for Pex and Fitnex.

The first purpose of the evaluation is to evaluate the capability of each tool (or search strategy) in achieving high code coverage, hence it is fair to set up a fixed testing budget for all. For these relatively small programs, we chose 1000 runs as the limit to run every test subject on each tool. We measured the percentage of branch coverage obtained. The results are shown in Table 1. The second purpose is to evaluate the capability of each tool in optimizing the path exploration process. As mentioned, this is an important criterion to evaluate the effectiveness of any test input generation tool based on dynamic symbolic execution since the cost of performing dynamic symbolic execution is expensive, minimizing the number of path explorations is necessary to make the technique applicable in practice. For this, besides the first stop condition, 1000 runs, we also stopped tools when all branch coverage of the experimenting test subject was met. The results are given in Table 2.

5.3 Experimental Results

Table 1 and Table 2 summarize the statistics we obtained after we carried out the experiments. It is clear from the statistics that Random is the worst approach to test input generation with the lowest average coverage (40%) obtained but the highest average number of runs (891 runs) exploited. Remarkably, on the test subjects *valves_nest<i>*, Random with its *careless* input generation mechanism run out of memory due to being trapped into too long loop iterations with large amounts of memory requested. We stopped Random after 25 runs for these test subjects and recorded the coverage achievement. Besides, we did not accumulate these numbers of runs into its average run number since it does not make sense to make comparison. These test subjects appear to be the biggest hurdle of current test input generation tools. For instance, Random could not reach beyond 2 branches on any of these test subjects.

DFS is an instance of using dynamic symbolic execution to systematically explore all feasible paths of the program. It lacks the ability to forward the execution to further unexplored control flow points and hence achieved very low branch coverage within the fixed testing limit. However, since DFS relies on the power of the underlying constraint solver, it must obtain higher coverage (56% on average) than Random. For CREST, it had 14 out of 15 cases failed to achieve full coverage. For these cases, the test subjects contain branches that require precise guidance of data flow analysis to be covered. Only CREST utilizes the static control flow and thus is not effective. As shown, on the selected test subjects, CREST achieved little coverage higher than DFS. However, this is was not the case when looking at exploration optimizations. On the test subjects *valves_nest<i>*, DFS and CREST both revealed their worst capability in coping with the combinatorial explosion of the path space to achieve high coverage.

Pex and Fitnex achieved quite similar average coverage results, 87% and 89%, respectively. While Pex failed in 8 cases to achieve full coverage, Fitnex had fewer 2 cases. In cases of *check_ISBN* and *check_ISSN*, both Pex and Fitnex automatically terminated after 234 and 313 runs, respectively, although all coverage was not achieved. The comparison thus favours these tools with respect to path explorations. The results obtained by both Pex and Fitnex are better than Random, DFS and CREST in terms of coverage achievements and exploration optimizations. This highlights the power of bringing several search strategies together as well as the power of fitness

functions in test input generation. However, on the test subjects *valves_nest50* and *valves_nest100*, both the approaches failed to expose high complexity code.

For STIG, it failed to achieve 100% coverage in 2 cases, *check_ISBN* and *check_ISSN*. We manually investigated these test subjects and realized that the two functions contain one unreachable branch which was resulted from the instrumentation step where our tool normalizes every *if* statement to have the form *if (e) then S1 else S2*. Currently, STIG is not able to deal with infeasible code. But an interesting observation when we conducted experiments on these test subjects is that even though we set the testing limit to 1000 runs, STIG stopped the exploration process after 51 runs for *check_ISBN* and 45 runs for *check_ISSN*. This means the search process considered all possible combinations of data flows but none could help to explore the test goal. This suggests evidence that this code element is infeasible. We refer this situation to *saturated data propagation* and are working to give a formal proof for identifying infeasible code through exploring data flows. Noticeably, on the six test subjects *valves_nest<i>*, while Random ran out of memory, DFS and CREST both revealed problematic issues in their search mechanism, Pex failed to reach even 50% for *valves_nest50* and *valves_nest100*, and for these two test subjects Fitnex also succumbed due to overly long execution traces. STIG did not only scored 100% coverage but also efficiently minimized the exploration process, namely using less than one run to explore one branch on average.

It is worth mentioning that on average STIG achieved the highest coverage (100% if infeasible code is not counted) and maintained a significantly small number of program explorations (78 runs compared to 426 and 478 of Fitnex and Pex, respectively, and 934 of CREST) on the selected test subjects. This shows the capability of utilizing data flow analysis to guide dynamic symbolic execution in the test input generation process.

In this evaluation, we did not conduct experiments to compare our proposed approach STIG with the well-known dynamic symbolic execution KLEE tool [8], and with the ARC-B tool [2]. The latter was proposed to combine dynamic symbolic execution and abstraction refinement to mitigate the combinatorial explosion of the path space for coverage improvements and infeasible code identification. However, having a look at the experimental data reported in Baluda et al. [2], specifically on the sequence of test subjects *valves_nest<i>*, we observe that the KLEE tool, like DFS and CREST, obtained very low coverage, covering only 5 branches in spite of the increasing size of the test subjects. In case of *valves_nest100*, DFS, CREST and KLEE all did not go beyond 1% coverage. For the ARC-B tool, when compared to STIG again on the test subjects *valves_nest<i>*, it demanded considerable numbers of program explorations to acquire the same coverage as STIG did, almost 6 times greater. Apart from that, on the test subject *valves_nest100*, ARC-B reached 71% coverage after 10,000 runs, STIG did not only scored 100% coverage but also successfully accomplished the search process after 501 runs, or approximately 20 times fewer.

6 Discussion

The Chaining Approach. The chaining approach we utilized in this work is a test input generation technique [12], which relies on a local search method called the

alternating-variable method to find test inputs but this is performed largely randomly. In addition, the chaining mechanism itself mainly focuses on propagating the effect of definition statements to the target structure but lacks the ability to consider at a definition it may need to perform certain computations to satisfy the target predicate. This limitation was *partially* addressed in the work of McMinn & Holcombe [17] and has been strengthened by STIG to be able to intertwine with dynamic symbolic execution.

Complexity. The cost of applying the chaining mechanism comes from two facets. One is performing data flow analysis to identify definition statements (or formally *reaching definitions* [1]) of problem nodes. This is a maximum fixedpoint algorithm operated statically on the source code of the program prior to dynamic symbolic execution. The algorithm complexity is the product of the height of the lattice and the number of nodes in the program flow graph, which is minor compared to the very expensive cost of performing dynamic symbolic execution. The other is the cost of performing dynamic symbolic execution with the guidance of event sequences. This cost results actually in the number of runs that STIG requires to execute the program, which was confirmed significantly smaller than other search strategies and tools. In fact, we observed from the experiments that CREST and STIG both executed the test subjects within a matter of a few seconds. Pex and Fitnex, however, consumed a considerable amount of time on all the test subjects.

Evaluation. The evaluation was conducted in a set of relatively small test subjects. However, these test subjects already demonstrated problematic exploration issues in current approaches to dynamic symbolic execution. In the future work, we aim to extend the proposed approach and conduct experiments on large test subjects to properly assess the validity of our proposal and observations. We believe that when testing sizeable and complex programs, where the path space is too large to systematically exhaustively explore, the ability to break down the path space and to precisely guide the search process by centralizing on selected aspects of semantics of our proposed approach is essential in optimizing the very expensive cost of performing dynamic symbolic execution to maximize coverage achievements and enhance error-detection capabilities.

7 Conclusions

Achieving high code coverage is an important goal of software testing. Dynamic symbolic execution based techniques hold much promise to make this goal achievable. When applied to real world software, the scalability of dynamic symbolic execution, however, is limited due to the extremely large program path space. In this paper, we have proposed to apply data flow analysis to effectively and efficiently perform dynamic symbolic execution for maximum code coverage. The proposed approach alleviates the combinatorial path space explosion by guiding the search process to focus on code segments that truly affect the execution of uncovered code. The experimental evaluation shows that STIG is effective in maximizing code coverage, optimizing path explorations, and providing useful evidence to identify infeasible code elements. In most of the experiments, STIG achieves higher coverage with significantly small path explorations than popular state-of-the-art test case generation tools.

Acknowledgements. We thank Kiran Lakhotia and Mauro Baluda for sending us source code of test subjects used in their work, references [4] and [2], respectively. We are grateful to Nikolai Tillmann and Tao Xie for their help on Pex [22] and Fitnex [25].

References

1. Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools, 2nd edn. Addison Wesley (2008)
2. Baluda, M., Braione, P., Denaro, G., Pezze, M.: Enhancing structural software coverage by incrementally computing branch executability. Software Quality Journal 19(4), 725–751 (2011)
3. Boyer, R.S., Elspas, B., Levitt, K.N.: SELECT–a formal system for testing and debugging programs by symbolic execution. In: Proceedings of the International Conference on Reliable Software, pp. 234–245. ACM, New York (1975)
4. Binkley, D.W., Harman, M., Lakhotia, K.: FlagRemover: a testability transformation for transforming loop-assigned flags. ACM Transactions on Software Engineering and Methodology 20(3) (2011)
5. Bird, D., Munoz, C.: Automatic generation of random self-checking test cases. IBM Systems Journal 22(3), 229–245 (1983)
6. British Standards Institute. BS 7925-1 Vocabulary of Terms in Software Testing (1998)
7. Burnim, J., Sen, K.: Heuristics for scalable dynamic test generation. In: Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering, pp. 443–446. IEEE Computer Society, Washington, DC (2008)
8. Cadar, C., Dunbar, D., Engler, D.: KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation. USENIX Association, Berkeley (2008)
9. Cadar, C., Godefroid, P., Khurshid, S., Păsăreanu, C.S., Sen, K., Tillmann, N., Visser, W.: Symbolic execution for software testing in practice: preliminary assessment. In: Proceedings of the 33rd International Conference on Software Engineering, pp. 1066–1071. ACM, New York (2011)
10. Clarke, L.A.: A system to generate test data and symbolically execute programs. IEEE Transactions on Software Engineering 2(3), 215–222 (1976)
11. Do, T.A., Fong, A.C.M., Pears, R.: Scalable automated test generation using coverage guidance and random search. In: Proceedings of the 7th International Workshop on Automation of Software Test, pp. 71–75 (2012)
12. Ferguson, R., Korel, B.: The chaining approach for software test data generation. ACM Transactions on Software Engineering and Methodology 5(1), 63–86 (1996)
13. Godefroid, P., Klarlund, N., Sen, K.: DART: directed automated random testing. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 213–223. ACM, New York (2005)
14. Horwitz, S., Reps, T., Binkley, D.: Interprocedural slicing using dependence graphs. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 35–46. ACM, New York (1988)
15. King, J.C.: Symbolic execution and program testing. Communications of the ACM 19(7), 385–394 (1976)
16. McMinn, P.: Search-based software test data generation: a survey. Software Testing, Verification & Reliability 14(2), 105–156 (2004)

17. McMinn, P., Holcombe, M.: Evolutionary Testing Using an Extended Chaining Approach. Evolutionary Computation 14(1), 41–64 (2006)
18. Offut, J., Hayes, J.: A Semantic Model of Program Faults. In: Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 195–200. ACM, New York (1996)
19. Pacheco, C.: Directed Random Testing. PhDDissertation, MIT Department of Electrical Engineering and Computer Science, Cambridge, Massachusetts (2009)
20. RTCA, Inc.: Document RTCA/DO-178B. U.S. Department of Transportation, Federal Aviation Administration, Washington, D.C (1993)
21. Sen, K., Marinov, D., Agha, G.: CUTE: a concolic unit testing engine for C. In: ESEC/FSE-13, pp. 263–272. ACM, New York (2005)
22. Tillmann, N., de Halleux, J.: Pex–white box test generation for.NET. In: Beckert, B., Hähnle, R. (eds.) TAP 2008. LNCS, vol. 4966, pp. 134–153. Springer, Heidelberg (2008)
23. Visser, W., Păsăreanu, C.S., Khurshid, S.: Test input generation with Java PathFinder. In: Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 97–107. ACM, New York (2004)
24. Weiser, M.: Program slicing. In: Proceedings of the 5th International Conference on Software Engineering, pp. 439–449. IEEE Press, Piscataway (1981)
25. Xie, T., Tillmann, N., Halleux, P.D., Schulte, W.: Fitness-guided path exploration in dynamic symbolic execution. In: Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks, pp. 359–368 (2009)

Using Roles as Units of Composition

Fernando Barbosa[1] and Ademar Aguiar[2]

[1] Escola Superior de Tecnologia, Instituto Politécnico de Castelo Branco, Av. do Empresário, Castelo Branco, Portugal
fsergio@ipcb.pt
[2] Departamento Informática, Faculdade de Engenharia da Universidade do Porto, Rua Dr. Roberto Frias, Porto, Portugal
ademar.aguiar@fe.up.pt

Abstract. A single decomposition strategy cannot capture all aspects of a concept, so we need to extend Object Oriented Decomposition (today most used strategy). We propose roles as a way to compose classes that provides a modular way of capturing and reusing those concerns that fall outside a concept's main purpose, while being a natural extension of the OO paradigm. Roles have been used successfully to model the different views a concept provides and we want to bring that experience to the programming level. We discuss how to make roles modular and reusable. We also show how to compose classes with roles using JavaStage, our role supporting language. To validate our approach we developed generic and reusable roles for the Gang of Four patterns. We developed reusable roles for 10 out of 23 patterns. We also were able to use some of these roles in JHotDraw framework.

Keywords: Modularity, Inheritance, Roles, Composition, Reuse.

1 Introduction

To deal with the complexities of any problem we normally use abstractions. In Object-Oriented (OO) languages, classes are the usual abstraction mechanism. Each class represents a specific concept. A single decomposition technique, however, cannot capture all possible views of the system [1] and each concept may be viewed differently depending on the viewer: a river may be a food resource to a fisherman, a living place to a fish, etc. Roles can accommodate these different views.

Roles were introduced by Bachman and Daya [2] but several role models have since been proposed. But the definitions, modeling ways, examples and targets are often different [3,4]. The research on roles has focused largely on its dynamic nature [5,6,7], modeling with roles [8] and relationships [9].

Role modeling, by decomposing the system into smaller units than a class, has proved to be effective, with benefits like improved comprehension, documentation, etc [10]. However, no language supports such use of roles. To overcome this fact we'll focus our role approach in class composition and code reuse.

We propose roles as a basic unit we can compose classes with. A role defines state and behavior that are added to the player class. Roles provide the basic behavior for concerns that are not the class's main concern, leading to a better modularization.

L.A. Maciaszek and J. Filipe (Eds.): ENASE 2012, CCIS 410, pp. 16–32, 2013.

A class can then be seen either as being composed from several roles or as an undivided entity.

To maximize role reuse we'll use modularity principles as guidelines. We intend to develop roles as modular units, making them reusable. We argue that developing roles independently of their players will make them much more reusable. To express our ideas we created JavaStage, an extension to Java. We will use JavaStage in the examples so we will give a brief introduction so examples are clear.

To show that roles can be reusable modules we show that it is possible to build a role library. We started our role library with the analysis of the Gang of Four (GoF) design patterns [11]. We were able to develop generic roles for 10 patterns. We were also able to use some of those roles in the JHotDraw Framework.

We can summarize our paper contributions as: a way of composing classes using roles as modular units; a java extension that supports roles; a role library based on the GoF patterns.

This paper is organized as follows: Section 2 gives a brief description of decomposition problems. Section 3 discusses how to enhance role reuse. Section 4 shows how to compose classes with roles using JavaStage. Our roles for the GoF patterns are debated in Section 5. Related work is presented in Section 6 and Section 7 concludes the paper.

2 Decomposition Problems

How do we decompose a system? There still isn't a definitive answer and there are many decomposition techniques. The most used today is Object Oriented Decomposition, but some argue that a single decomposition strategy cannot adequately capture all the system's details [1].

Consequences of using a single decomposition strategy are crosscutting concerns. They appear when several modules deal with the same problem, which is outside their main concern, because one cannot find a single module responsible for it. This leads to scattered, replicated code.

Because a module must deal with a problem that is spread by several others, changes to that code will, quite probably, affect other modules. Independent development is thus compromised, evolution and maintenance are a nightmare because changes to a crosscutting concern need to be done in all modules.

We will tackle this problem by using roles as a building block for classes. We put the crosscutting concern in a role and the classes play the role. Any changes to the concern are limited to the role, improving maintenance and reducing change propagation. The crosscutting concerns become more modular.

2.1 Multiple Inheritance

To overcome decomposition restrictions some languages use multiple inheritance. But multiple inheritance also has multiple problems, caused mostly by name collisions when a class inherits from two or more superclasses that have methods with the same signature or fields with the same name. It can even occur when a class inherits twice

from the same superclass – the diamond problem. Different languages provide different solutions (virtual classes in C++) and others simply forbid it like Java.

Java uses interfaces when a class must be seen as conforming to another type. Interfaces only declare constants and methods signatures so they have no state or method implementations. This may result in the code duplication in classes implementing the same interface but with different superclasses.

It is usual to start an inheritance hierarchy with an interface and then a superclass providing the default behavior for that hierarchy. We argue that the default implementation should be provided by a role and the superclass plays that role. This way we can reuse the basic behavior whenever we need to, thus preventing the use of multiple inheritance. This is depicted in Fig. 1. The example shows a Figure hierarchy with an interface and a role at the top. The DefaultFigure class implements the interface and plays the role. All its subclasses inherit this default behavior. The ImageFigure, a subclass from another hierarchy, also becomes part of the Figure hierarchy by implementing the Figure interface. It also plays the BasicFigure role so it has the same default behavior every DefaultFigure subclass has.

2.2 Aspect Oriented Programming

There are other attempts to remove crosscutting concerns, like Aspect Oriented Programming (AOP) [12]. However AOP is not close to OO and requires learning many new concepts. And while the modularization of crosscutting concerns is the flagship of AOP several authors disagree [13,14].

Concepts like pointcuts and advices are not easy to grasp, and their effects are more unpredictable than any OO concept. A particular one is the fragile pointcut [15]: simple changes in a method can make a pointcut either miss or incorrectly capture a joint point thus incorrectly introducing or failing to introduce the required advice.

AOP obliviousness [16] means that the class is unaware of aspects and these can be plugged or unplugged as needed. This explains why some dynamic role languages use AOP. But it also brings comprehensibility problems [17]. To fully understand the system we must know the classes and the aspects that may affect them. This is a major drawback when maintaining a system, since the dependencies aren't always explicit and there isn't an explicit contract between both parts.

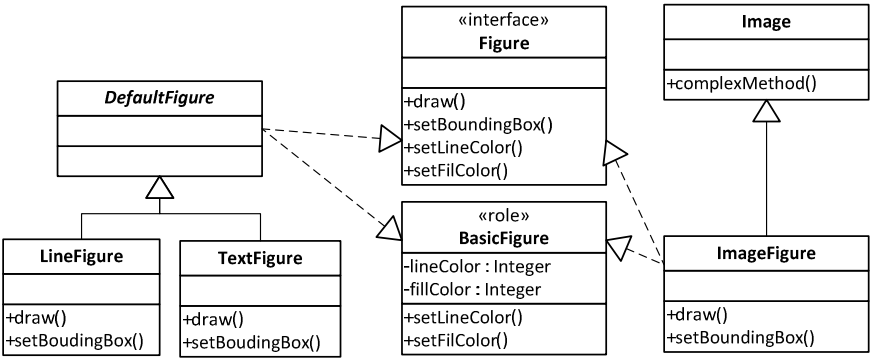


Fig. 1. Example of a Figure hierarchy with both an interface and a role as top elements

With roles all dependencies are explicit and the system comprehensibility is increased compared to the OO version [8]. Roles do not have AOP obliviousness because the class is aware of the roles it plays. Any changes to the class do not affect the role, if the contract between them stays the same.

Our approach does not replace AOP. They are different and approach different problems. We believe that for modeling static concerns our approach is more suitable while AOP is better suited for pluggable and unpplugable concerns.

2.3 Traits

Classes' composition using alternatives to multiple inheritance have been proposed such as mixins [18, 19] and traits [20, 21, 22]. Traits have one advantage over mixins and single inheritance: the order of composition is irrelevant. Traits have first appeared in smalltalk but some attempts have been made into bringing traits in to java-like languages [23, 24].

Traits can be seen as a set of methods that provide common behavior. Traits are stateless, the state is supplied by the class that uses it and accessed by the trait through required methods, usually accessors. Trait's methods are added to the class that uses them. The class also provides glue code to compose the several traits it uses. Traits can impose restrictions on the classes that use them, but cannot impose restriction on the classes it interacts with.

Traits don't have visibility control, meaning that a trait cannot specify an interface and all trait methods are public, even auxiliary ones. Since traits cannot have state then this is a minor problem, but it does limit a class-trait interface.

Traits have a flattening property: a class can be seen indifferently as a collection of methods or as composed by traits, and that a trait method can be seen as a trait method or as a class method.

In our approach a class can be seen as being composed from several roles or as an undivided entity. This is not to be confused with the flattening property of traits. A super reference in a trait refers to the super of the class that uses the trait, while a super reference in the role refers to the super role.

Our roles can have state, visibility control and their own inheritance hierarchy while traits cannot. In our approach the order of role playing is also irrelevant except for a specific conflict resolution, but it is so to facilitate development and can be overridden by the developer or by the compiler.

3 Reusing Roles

This section is dedicated to what we believe are the factors that will enhance role reuse: independent evolution of roles and players, role configuration, and roles being used as components for classes.

3.1 Roles as Modules

Modularization [25] is one of the most important notion in software development. Breaking a system into modules allows each module to be independently developed,

shortening development time. Other advantages are better comprehensibility, enhanced error tracing, etc, but the one developers treasure most is the modules' high reusability. It allows library development and libraries reduce the amount of code one must write to build a system.

A key concept is encapsulation. When a module is encapsulated changes in the module, even drastic ones, do not affect other modules. A module has an interface and an implementation. The interface defines how clients interact with the module, and it shouldn't change much along the module life-cycle as clients must be aware of the changes and change their own implementation accordingly.

Modules interact with each other but intra-modules interactions are more intense than inter-modules interactions. Intra-modules interactions may require a specialized interface. To cope with this, most languages declare different levels of access, usually private, protected and public.

To maximize role reuse we have to enable the independent evolution between roles and players. If we treat a role as a module and the player as another module then we can strive for a greater independence between them. Thus roles must provide an interface and ensure encapsulation.

Providing an interface is simple if we use roles as first class entities. Encapsulation and independent development raises a few issues. We must consider that roles only make sense when played by a class. But classes cannot have access to role members and vice-versa. If they did roles and classes could not be developed independently, because any change to the role structure could cause changes in the class and vice-versa. Therefore roles and classes must rely solely on interfaces.

3.2 Dropping the PlayedBy Clause

Many role approaches focus on the dynamic use of roles: extending objects by attaching roles. Roles are usually bounded to a player by a playedBy clause that states who can play the role. In dynamic situations where roles are developed for extending existing objects this is acceptable, even if it restricts role reuse, but not in static situations.

Using an example derived from [26] we show, in Fig. 2, a Point class representing a 2D Cartesian coordinate and a Location role that provides a view of Point as a physical location on a map. We also present a PolarPoint class that represents a coordinate in polar coordinates. The role could apply to both classes but the playedBy forbids it as these classes are not related. Making one a subclasse of the other would violate the "is a" rule of inheritance. We could use a Coordinate interface with getX and getY methods with both classes implementing that interface. This cannot be done in a dynamic context where both classes are already developed and cannot be modified.

Our purpose is to use roles as building blocks and not for extending objects. This is a totally different way of viewing role-class relationships. Our roles are meant to be used to compose classes so roles are developed without knowledge of all classes that can play them. Thus using the playedBy clause would limit role reusability. In the example, if we develop a role for both classes the role must state that it needs the player to have getX and getY methods. Some form of declaring these requirements must be used but not by using a playedBy clause.

```
class Point {
  int x, y;
  Point(int x, int y){this.x= x; this.y= y;}
  int getX( ) { return x; }
  int getY( ) { return y; }
}
class PolarPoint {
  int r;  double beta;
  PolarPoint(int r, double b) { this.r = r; beta = b; }
  int getX(){return (int)(r*Math.cos(beta));}
  int getY(){return (int)(r*Math.sin(beta));}
}
role Location playedBy Point {
  string getCountry() {
    int x = performer.getX();
    int y = performer.getY();
    String country = "PT";  // converting point to a country name
    return country;
  }
}
```

Fig. 2. A Point class, a Location role playable by it and a PolarPoint class that could also play the Location role

3.3 The Need to Rename Methods

Methods names are specific to an interaction. For example, Observer [11] describes an interaction between subjects and observers. It is used in many systems with minor changes, usually the methods used to register an observer with a subject and the update methods used by the subject to notify its observers. A Subject role for a MouseListener instance of the pattern would define methods like addMouseListener, or removeMouseListener. That role could not be reused for a KeyListener instance which uses methods like addKeyListener or removeKeyListener.

A method's name must indicate its purpose, so a name like addListener reduces comprehensibility, and can limit the class to play only one subject role. Thus, renaming methods expands role reusability. A class that plays a role must ensure a specific interface, but that interface should be configurable, at least in what respects to method names. Some languages [7] use a "rename" clause that allow classes to rename a role method. If the role interface is big this task is tedious and error prone. We need a more expedite way of doing this.

Roles also interact with other objects. Again method names are important. For example, each subject has a method that calls the observer's update method. In the Java AWT implementation of the pattern there are several methods like mousePressed, mouseReleased, etc. The rename clause is not usable here because it applies only to the role methods. We need a mechanism that allows fast renaming for role methods and methods called by the role.

3.4 Summary

For roles to be fully reusable then they must: provide an interface; ensure encapsulation; be developed independently from its players; state requirements player must fulfill; provide a method renaming mechanism that enables the role to be configured by the player.

4 Composing Classes Using Roles

To support roles we developed JavaStage, an extension to Java. Examples in this paper have been compiled with our JavaStage compiler. We will not discuss JavaStage's syntax in detail but it will be perceptible from the examples and we will explain it briefly so that examples are understandable.

4.1 Declaring Roles

A role may define fields, methods and access levels. A class can play any number of roles, and can even play the same role more than once. We refer to a class playing a role as the player of that role.

When a class plays a role all the non private methods of the role are added to the class. To play a role the class uses a plays directive and gives the role an identity, as shown in Fig. 3. To refer to the role the class uses its identity.

As an example we will use the Subject role from the Observer pattern. Consider a Figure in a drawing application. Whenever the Figure changes, the drawing must be updated so the figure plays the role of an observer's subject. Being a subject is not the Figure main concern so it's wise to develop a subject role, shown in Fig. 3, to capture that concern and let figures play it. In the code we omitted access modifiers for simplicity, but they should be used.

```
role FigureSubject {
  Vector<FigureObserver> observers = new Vector<FigureObserver>();
  void addFigureObserver( FigureObserver o)  { observers.add( o );    }
  void removeFigureObserver(FigureObserver o){ observers.remove( o ); }
  protected void fireFigureChanged( ){
    for( FigureObserver o : observers )    o.update( );
  }
}
class DefaultFigure implements Figure {
  plays FigureSubject figureSbj;
  void moveBy(int dx, int dy) {
    // code for moving the figure
    // firing change, using role identity
    figureSbj.fireFigureChanged();
  }
}
```

Fig. 3. A Figure subject role for an instance of the observer pattern and a class playing it

4.2 Stating Role Requirements

A role does not know who will be its players but may need to exchange information with them so it must require the player to have a particular interface. We do that using a requirements list. The list can include required methods from the player but also required methods from objects the role interacts with. The list states the method owner and the method signature. To indicate that the owner is the player we use the Performer keyword. Performer is used within a role as a place-holder for the player's type. This enables roles to declare fields and parameters of the type of the player. This is shown in Fig. 4 which shows a singleton role.

4.3 Method Renaming

We developed a renaming mechanism, to enhance role reuse and facilitate role configuration, which allows method names to be easily configured. Each name may have three parts: one configurable and two fixed. Both fixed parts are optional so the name can be fully configurable by the player. The configurable part is bounded by # as shown next.

```
fixed#configurable#fixed
```

The name configuration is done by the player in the plays clause as depicted in Fig. 5. To play the role the class must define all configurable methods.

```
public role Singleton {
  requires Performer implements Performer();
  private static Performer single = null;
  public static Performer getInstance( ){
    if( single == null )  single = new Performer();
    return single;
  }
}
```

Fig. 4. A Singleton role requiring its player to have a default constructor

```
public role GenericSubject<ObserverType> {
  requires ObserverType implements void #Fire.update#();
  public void add#Observer#( ObserverType o){  observers.add( o ); }
  protected void fire#Fire#( ){
    for( ObserverType o : observers )  o.#Fire.update#( );
  }
}
class DefaultFigure implements Figure {
  plays GenericSubject<FigureObserver>
     ( Fire = FigureChanged,       Fire.update = figureChanged,
       Observer = FigureObserver       ) figureSbj;
  plays GenericSubject<FigureHandleObserver>
     ( Fire = FigureHandleChanged, Fire.update = figureHandleChanged,
       Observer = FigureHandleObserver ) figHandleSbj;
 public void moveBy( int dx, int dy) { figureSbj.fireFigureChanged(); }
}
```

Fig. 5. The generic subject role with configurable methods and a class playing that role twice

We can take our figure subject role and make it more generic with this renaming mechanism. In Fig. 5 we show how we can use method renaming to make our subject role more generic. It also shows a class playing that role as a FigureObserver subject and as a FigureHandlerObserver subject.

It is possible for many methods to share the same configurable part. This way we can rename several methods with a simple configuration. This is useful when several methods share a common part. One such example is the Subject role, where the several methods that manage the observers usually contain the type of the observer.

4.4 Multiple Versions of a Method

It's possible to declare several versions of a method using multiple definitions of the configurable name. This enables methods to have multiple names. More important though, is that it also enables methods with the same structure to be defined just once. This is useful when several methods have the same structure but each calls different methods.

We can expand FigureObserver to include more update methods to specify which change occurred, like figureMoved. Such plays clause would be:

```
plays GenericSubject<FigureObserver>
  ( Fire = FigureChanged, Fire.update = figureChanged,
    Fire = FigureMoved,   Fire.update = figureMoved,
    Observer = FigureObserver  ) figureSbj;
```

4.5 Making Use of Name Conventions

Another feature of our renaming strategy is the class directive. When class is used as a configurable part it will be replaced by the name of the player class. This is useful in inheritance hierarchies because we just need to place the plays clause in the superclass and each subclass gets a renamed method. It does imply that calls will rely on name conventions.

```
role VisitorElement<VisitorType> {
 requires VisitorType implements void visit#visitor.class#(Performer t);

  void accept#visitor#( VisitorType v ){
    v.visit#visitor.class#( performer );
  }
}
class DefaultFigure {
  plays VisitorElement<FigureVisitor>( visitor = Visitor ) visit;
  // … rest of class code
}
class LineFigure extends DefaultFigure {
    // no Visitor pattern code
}
interface FigureVisitor {
    void visitLineFigure( LineFigure f );
    void visitTextFigure( TextFigure f );
    //…
}
```

Fig. 6. The VisitorElement role, a class Sigure that plays the role, a subclass from the Sigure hierarchy and the Visitor interface

One such case is the Visitor pattern. This pattern defines two roles: the Element and the Visitor. The Visitor declares a visit method for each Element. Each Element has an accept method with a Visitor as an argument that calls the corresponding method of the Visitor. Visitor's methods usually follow a name convention in the form of visitElementType. We used this property in our VisitorElement role, as shown in Fig. 6. The example shows it being used in a Figure hierarchy with figures as Elements. It also shows that Figure subclasses don't have any pattern code, because they will get an acceptVisitor method that calls the correct visit method.

4.6 Roles Playing Roles or Inheriting from Roles

Roles can play roles but can also inherit from roles. When a role inherits from a role that has configurable methods it cannot define them. When a role plays another role it must define all its configurable methods.

For example managing observers is a part of a more general purpose concern that is to deal with collections. We can say that the subject role is an observer container and develop a generic container role and make the subject inherit from the container.

If the FigureSubject role can be played by several classes then we'll create a FigureSubject based on GenericSubject. Because we need to rename the role methods the FigureSubject role must play the generic Subject role and define all its methods. DefaultFigure would then use FigureSubject without any configuration.

Both situations are depicted in Fig. 7.

4.7 Conflict Resolution

Class methods have precedence over role methods. Conflicts may arise when a class plays roles that have methods with the same signature. When conflicts arise the compiler issues a warning. The conflict can be resolved by redefining the method and

```
role GenericContainer<ThingType> {
 Vector<ThingType> ins =  new Vector<ThingType>();
 void add#Thing#( ThingType t )           { ins.add( t );  }
 void insert#Thing#At(ThingType t,int i)  { ins.insertElementAt(t, i);}
  protected Vector<ThingType> get#Thing#s(){ return ins; }
}
role GenericSubject<ObserverType> extends GenericContainer<ObserverType> {
  requires ObserverType implements void #Fire.update#();
  protected void fire#Fire#( ){
    for( ObserverType o : get#Thing#s() )   o.#Fire.update#( );
  }
}
role FigureSubject {
  plays GenericSubject<FigureObserver>
    ( Fire = FigureChanged,  Fire.update = figureChanged,
      Fire = FigureMoved,    Fire.update = figureMoved,
      Thing = FigureObserver ) figureSbj;
}
class DefaultFigure implements Figure {
  plays FigureSubject figureSbj;
}
```

Fig. 7. Role inheritance and role playing roles

calling the intended method. This is not mandatory because the compiler uses, by default, the method of the first role in the plays order and role methods override inherited methods. This may seem a fragile rule, but we believe it will be enough for most cases. Even if a conflicting method is later added to a role the compiler does issue a warning so the class developer is aware of the situation. He can solve the situation as he wishes and not as imposed by the role or superclass' developers.

Conflicts do not arise when a player plays the same role more than once if it renames the role methods or role methods have different parameter types. There is never a conflict between role fields as the role identity is required to access a role field. Problems like the diamond problem in multiple inheritance are thus avoided.

5 Towards a Role Library

5.1 Roles in Design Patterns

To start our role library we analyzed the 23 GoF patterns [11]. They are a good starting point because of its wide use. If we create roles for these patterns then our approach will have impact on many of today frameworks and applications.

Each pattern defines a number of collaborating participants. Some participants can be seen as roles while others cannot. This distinction is made in [27] by considering the roles defining or superimposed. For each pattern we took the roles of each participant and focused on similar code between pattern instances to find reusable code. We present our results by groups of patterns. They were grouped by similarities between implementation or problems. We've built a sample scenario for each pattern but will not discuss them, due to space constraints.

Singleton, Composite, Observer, Visitor. Singleton, Observer and Visitor were already discussed. Composite uses the Container role. Each composite maintain a collection of child components and implements the operations defined by the component hierarchy. Children management is common between instances, so we reused the Container role. Component operations are instance dependent and not suitable for generalization, even if they mostly consist in iterating through the children and performing the operation on each child.

Factory Method, Prototype. With these patterns we developed roles that provide a greater modularity and dynamicity not present in traditional implementations. The use of the class directive for renaming is common to these roles.

Factory Method defines an interface for creating an object, but let subclasses decide which class to instantiate. Implementation of this pattern is instance dependent. There is, however, a variation whose purpose is to connect parallel class hierarchies: each class from a hierarchy delegates some tasks to a corresponding class of another hierarchy. Each class has a method that creates the corresponding class object (product). We moved the creation of the product to a creator class, which provides methods to create all products, one method each. Classes just call the right method in the creator. One advantage is the modularization of the pattern as the association between

classes is made in a single class not on a class by class basis. Future changes are made to the creator class only. Because the creation process is in a single class we can dynamically change the creator. We developed a role that allows the specification of the factory method that creates the object of the parallel class. The method uses the class directive so the plays clause is used only in the top class. It implies the use of naming conventions, but that is a small price to pay for the extra modularity. We also developed a role with a fixed creator, when dynamic creators aren't needed.

The Prototype pattern specifies the kind of objects to create using a prototypical instance, and creates new objects by cloning this prototype. The prototype class has a clone method that produces a copy of the object. Every class has its own cloning method but it may not be sufficient because the clone method may do a shallow copy where a deep copy is needed, or vice-versa. The client should choose how the copy is made. We developed a role that moves the creation of the copy to another class, as we did for FactoryMethod. That class is now responsible for creating the copies of all classes used as prototypes and thus may choose how to make the copy. Because it uses the class directive Prototype subclasses don't need to declare the clone method.

Flyweight, Proxy, State, Chain of Responsibility. Roles developed for these patterns are basically management methods. They are useful as they provide the basic pattern behavior and developers need to focus only on the specifics of their instance.

Flyweight depends on small sharable objects that clients manipulate and on a factory of flyweights that creates, manages and assures the sharing of the flyweights. The concrete flyweights are distinct but many flyweight factories have a common behavior: verify if a flyweight exists and, if so, return it or, if not, create, store and then return it. Our flyweight factory role manages the flyweights. Players supply the flyweight creation method.

In Proxy a subject is placed inside one object, the proxy, which controls access to it. Some operations are dealt by the proxy, while others are forwarded to the subject. Which methods are forwarded or handled are instance dependent as is the creation of the subject. Forwarding and checking if the subject is created or accessible is fairly similar between instances. Our proxy role stores the subject reference and provides the method that checks if the subject exists and triggers its creation otherwise.

The State pattern allows an object to alter its behavior when its internal state changes. There are almost no similarities in this pattern because each instance is unique. Our role is responsible for keeping the current state and for state transitions. The state change method terminates the actual state before changing to, and starting, the new state.

Chain of Responsibility avoids coupling the sender of a request to its receiver. Each object is chained to another and the request is passed along the chain until one handles it. Implementations of this pattern often use a reference to the successor and methods to handle or pass the request. Each instance differs in how the request is handled and how each handler determines if it can handle the request. Some implementations use no request information, others require some context information and in others the request method returns a value. We developed a role for each variation.

Abstract Factory, Adapter, Bridge, Decorator, Command, Strategy. The code for these patterns is very similar between instances but we could not write a role for any. For example, many abstract factories have methods with a return statement and the creation of an object. However the object's type and how it is created are unique. Adapter instances are similar in the way the Adapter forwards calls to the adaptee, but the call parameters and return types vary for each method.

Builder, Façade, Interpreter, Iterator, Mediator, Memento, Template Method. These patterns showed no common code between instances, because they are highly dependent on the nature of the problem. For example, an iterator is developed for a concrete aggregate and every aggregate has a unique way to traverse.

5.2 Summary

We developed roles for a total of 10 patterns out of 23, which is a good outcome, especially because every developed role is reusable in several scenarios. We believe that our Subject role, for example, will be useful for a large number of Observer instances. There are also additional advantages in some roles, like a better modularity in Factory Method and Prototype. Some roles are limited in their actions, like State but are highly reusable, nevertheless.

From our study there are a few patterns that do not gain from the use of roles. These roles are quite instance specific and the classes built for their implementation are dedicated and are not reusable outside the pattern. There are a few patterns that could benefit from using roles to emulate multiple inheritance and provide a default implementation to operations done in a class inheritance hierarchy, like Abstract Factory and Decorator. We also found similar code between instances that we could not put into a role. This was the case of patterns that forwarded method calls, like Adapter, Decorator and Proxy. However the variations were not supported by roles because they were in the methods return type and parameters types and number.

5.3 Testing Role-Player Independency

In order to asses if our roles are independent of their players we took the sample scenarios that illustrated its use and built a dependency structure matrix (DSM) for each. We use our sample of the Observer role and its DSM as an example of that work.

For an Observer sample we developed a Flower class that notifies its observers when it opens, as shown in Fig. 8. Flower plays the FlowerSubject role, which is the Subject role configured to this particular scenario. As an observer we developed a Bee class that when notified prints a message saying it is seeing an open flower. The code for the bee, observer interface and the flower event are not shown for simplicity. The FlowerSubject role is not really necessary as the Flower could configure the Subject role directly but it is good practice to do so.

From that sample we obtained the DSM of Fig. 9. Here we can find that there is no dependency between the Subject role and the Flower class and that the FlowerSubject depends only on the Subject role and not vice-versa. If we group the classes into

modules as shown in the figure we can see that the module where the role is included does not depend on any other module. It shows that the flower module is dependent from the role module via the role. It also shows that the Flower module does not depend on its concrete observers, as expected from the observer pattern. The Subject role is therefore independent of its players as could be inferred from the use of the subject role in a total of 3 examples in this paper alone. We may also add that we also used that same role in the JHotDraw Framework.

5.4 Reusing Roles in Real Systems

We also tested our roles using the JHotDraw framework. From this study we were able to use the Observer role often, the State role and the Visitor role. This seems to indicate that the developed roles are independent from their players and reusable. We plan, in future work, to study other open source systems to fully assess this.

```
public role FlowerSubject {
  plays Subject<FlowerObserver,FlowerEvent>(
            Thing=FlowerObserver,
            Fire=Open, Fire.update=flowerOpened ) sbj;
}
public class Flower {
   plays FlowerSubject flwrSubject;
   private boolean opened = false;
   public void open(){
     opened = true;
     fireOpen( new FlowerEvent( this ) );
   }
}
```

Fig. 8. The FlowerSubject role and the Flower class from our subject role sample

Name		1	2	3	4	5	6	7	8
EventType	1								
ObserverType	2	1							
Subject	3	1	1						
FlowerEvent	4							1	
FlowerObserver	5				1				
FlowerSubject	6	1			1	1			
Flower	7				1		1		
Bee	8				1	1		1	

Fig. 9. DSM of the Observer role sample

6 Related Work

To our knowledge there's never been an attempt to implement roles as static types and as components of classes. Riehle [10] lays the foundations for role modeling using static roles. He proved role usefulness in the various challenges frameworks are

faced with, like documentation, comprehensibility, etc. He does not propose a role language, but simply explains how roles could be used in some languages.

Chernuchin and Dittrich [28] use the notion of natural types and role types that we followed. They also described ways to deal with role dependencies which we didn't consider as it would introduce extra complexity to the role language. They suggest programming constructs to support their approach but no role language has emerged.

Chernuchin and Dittrich [29] compared five approaches for role support in OO languages. They were multiple inheritance, interface inheritance, the role object pattern, object teams and roles as components of classes. They used criteria such as encapsulation, dependency, dynamicity, identity sharing and the ability to play the same role multiple times. Roles as components of classes compared fairly well and the only drawback, aside dynamicity, was the absence of tools that supported it. With JavaStage that drawback is eliminated.

Object Teams [5] is an extension to Java that uses roles as first class entities. They introduce the notion of team. A team represents a context in which several classes collaborate. Even though roles are first class entities they are implemented as inner classes of a team and are not reusable outside that team. Roles are also limited to be played by a specific class.

EpsilonJ [7] is another java extension that, like Object Teams, uses aspect technology. In EpsilonJ roles are also defined as inner classes of a context. Roles are assigned to an object via a bind directive. EpsilonJ uses a requires directive similar to ours. It also offers a replacing directive to rename methods names but that is done on an object by object basis when binding the role to the object.

PowerJava [6] is yet another java extension that supports roles. In PowerJava roles always belong to a so called institution. When an object wants to interact with that institution it must assume one of the roles the institution offers. To access specific roles of an object castings are needed. Roles are written for a particular institution, therefore we cannot reuse roles between institutions.

7 Conclusions

We presented a way of composing classes using roles. With roles we are able to capture the concerns that are not the class main concern and modularize them. We presented an, hitherto missing, language that supports roles as components of classes and showed how we can use it to compose classes. Moreover we showed that roles can be made reusable to a great extent. The result was the development of generic roles for 10 GoF patterns.

References

1. Tarr, P.L., Ossher, H., Harrison, W.H., Sutton, Jr., S.M.: N degrees of separation: Multi-dimensional separation of concerns. In: International Conference on Software Engineering (1999)
2. Bachman, C.W., Daya, M.: The role concept in data models. In: Proceedings of the 3rd International Conference on Very Large Databases, pp. 464–476 (1977)

3. Graversen, K.B.: The nature of roles - A taxonomic analysis of roles as a language construct, Ph. D. Thesis, IT University of Copenhagen, Denmark (2006)
4. Steimann, F.: On the representation of roles in object-oriented and conceptual modeling. Data & Knowledge Engineering 35(1), 83–106 (2000)
5. Herrmann, S.: Programming with Roles in ObjectTeams/Java. In: AAAI Fall Symposium: "Roles, An Interdisciplinary Perspective" (2005)
6. Baldoni, M., Boella, G., van der Torre, L.: Interaction between Objects in power-Java. Journal of Object Technologies 6, 7–12 (2007)
7. Tamai, T., Ubayashi, N., Ichiyama, R.: Objects as Actors Assuming Roles in the Environment. In: Choren, R., Garcia, A., Giese, H., Leung, H.-f., Lucena, C., Romanovsky, A. (eds.) SELMAS. LNCS, vol. 4408, pp. 185–203. Springer, Heidelberg (2007)
8. Riehle, D., Gross, T.: Role Model Based Framework Design and Integration.'. In: Proceedings of the, Conference on Object-Oriented Programming Systems, Languages, and Applications (1998)
9. Pradel, M.: Explicit Relations with Roles - A Library Approach. In: Workshop on Relationships and Associations in Object-Oriented Languages, RAOOL (2008)
10. Riehle, D.: Framework Design: A Role Modeling Approach, Ph. D. Thesis, Swiss Federal Institute of technology, Zurich (2000)
11. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley (1995)
12. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An overview of AspectJ. In: Lindskov Knudsen, J. (ed.) ECOOP 2001. LNCS, vol. 2072, pp. 327–335. Springer, Heidelberg (2001)
13. Steimann, F.: The paradoxical success of aspect-oriented programming". In: Proceedings of the 21st Annual Conference on Object-Oriented Programming Languages, Systems, and Applications, OOPSLA 2006 (2006)
14. Przybyłek, A.: Systems Evolution and Software Reuse in Object-Oriented Programming and Aspect-Oriented Programming. In: Bishop, J., Vallecillo, A. (eds.) TOOLS 2011. LNCS, vol. 6705, pp. 163–178. Springer, Heidelberg (2011)
15. Koppen, C., Störzer, M.: PCDiff, 2004: Attacking the fragile pointcut problem. In: European Interactive Workshop on Aspects in Software, Berlin, Germany (2004)
16. Filman, R.E., Friedman, D.P.: Aspect-oriented programming is quantification and obliviousness. In: Workshop on Advanced Separation of Concerns at OOPSLA (2000)
17. Griswold, W.G., Sullivan, K., Song, Y., Shonle, M., Tewari, N., Cai, Y., Rajan, H.: Modular Software Design with Crosscutting Interfaces. IEEE Software 23(1), 51–60 (2006)
18. Bracha, G., Cook, W.: Mixin-Based Inheritance. In: Proceedings of the OOPSLA/ECOOP, Ottawa, Canada, pp. 303–311. ACM Press (1990)
19. Bracha, G.: The programming language jigsaw: mixins, modularity and multiple inheritance. PhD thesis, University of Utah (1992)
20. Scharli, N., Ducasse, S., Nierstrasz, O., Black, A.P.: Traits: Composable units of behavior. In: Cardelli, L. (ed.) ECOOP 2003. LNCS, vol. 2743, pp. 248–274. Springer, Heidelberg (2003)
21. Ducasse, S., Schaerli, N., Nierstrasz, O., Wuyts, R., Black, A.:: Traits: A mechanism for fine-grained reuse. Transactions on Programming Languages and Systems (2004)
22. Black, A., Scharli, N.: Programming with traits. In: Proceedings of the International Conference on Software Engineering (2004)
23. Quitslund, P., Black, A.: Java with traits - improving opportunities for reuse. In: Proceedings of the 3rd International Workshop on Mechanisms for Specialization, Generalization and Inheritance (2004)

24. Smith, C., Drossopoulou, S.: *chai*: Traits for java-like languages. In: Gao, X.-X. (ed.) ECOOP 2005. LNCS, vol. 3586, pp. 453–478. Springer, Heidelberg (2005)
25. Parnas, D.L.: On the criteria to be used in decomposing systems into modules. Commun. ACM 15(12), 1053–1058 (1972)
26. Ingesman, M.D., Ernst, E.: Lifted Java: A Minimal Calculus for Translation Polymorphism. In: Proceeding of the International Conference on Objects, Models, Components and Patterns, Zurich, Switzerland (2011)
27. Hannemann, J., Kiczales, G.: Design Pattern Implementation in Java and AspectJ. In: Proceedings of the 17th Conference on Object-Oriented Programming, Seattle, USA (2002)
28. Chernuchin, D., Dittrich, G.: Role Types and their Dependencies as Components of Natural Types. In: 2005 AAAI Fall Symposium: Roles, , An Interdisciplinary Perspective (2005)
29. Chernuchin, D., Lazar, O.S., Dittrich, G.: Comparison of Object-Oriented Approaches for Roles in Programming Languages. In: Papers from the, Fall Symposium (2005)

Extending Service Selection Algorithms with Interoperability Analysis

Paweł Kaczmarek

Gdańsk University of Technology,
Faculty of Electronics, Telecommunications and Informatics,
Narutowicza 11/12 Str., Gdańsk, Poland
pkacz@eti.pg.gda.pl
http://www.eti.pg.gda.pl

Abstract. Application development by integration of existing, atomic services reduces development cost and time by extensive reuse of service components. In Service Oriented Architecture, there exist alternative versions of services supplying the same functionality but differing in Quality of Service (QoS) attributes, which enables developers to select services with optimal QoS. Existing algorithms of service selection focus on the formal model of the composite service refraining from interoperability issues that may affect the integration process. In this paper, the author proposes a methodology that extends existing service selection algorithms by introducing additional constraints and processing rules representing interoperability. Two computational models are considered: the graph-based model and the combinatorial model. The extensions enable a straightforward application of a wide range of existing algorithms. The paper also describes a system that implements chosen service selection algorithms together with appropriate extensions for interoperability analysis.

Keywords: Service Oriented Architecture, Interoperability, Service Integration, Business Process Management, Quality of Service.

1 Introduction

Application development in Service Oriented Architecture (SOA) assumes that alternative services realizing the same functionality are available. The services differ in non-functional, Quality of Service (QoS) attributes such as performance, reliability and price. Typically, application development intends to optimize the QoS, which means maximization of desired attributes, such as performance, while simultaneously meeting constraints imposed on the final application, such as maximum price or minimum reliability [1], [2].

Existing methods and algorithms of service selection, however, do not analyze technical interoperability that may limit the choice of alternative services despite theoretical conformance of functionality and correctness of QoS analysis. Although services typically use the Web services standards (WS*) to exchange information, there are many threats for effective integration. The runtime platforms that host services support only selected standards and versions from the wide set of WS* standards. Even if runtime platforms are compatible in supported standards, interoperability is not guaranteed

L.A. Maciaszek and J. Filipe (Eds.): ENASE 2012, CCIS 410, pp. 33–47, 2013.

because of standard inconsistencies and vendor-specific extensions [3], [4]. Service selection methods additionally require mechanisms to represent and guarantee interoperability in the final application.

Considering the problem, the author proposes a methodology that includes interoperability analysis in service selection algorithms. The data structures of service selection algorithms are extended with dedicated constraints that represent interoperability between services. The constraints result from interoperability rates that consider runtime environments that host the services and WS* standards that are used by the services. In the methodology, two computational representations are analyzed: the combinatorial model together with algorithms based on the integer linear programming (ILP) problem and the graph-based model together with algorithms based on single source shortest path selection. The proposed extensions enable a straightforward application of a wide range of existing algorithms as the extensions preserve the general structure of data models.

The proposed solution was implemented in a web-based system that supports service selection in development of workflow applications as a representative case of the composite service concept. The system maintains a knowledge base of runtime environments interoperability. Chosen service selection algorithms together with appropriate extensions for interoperability analysis were implemented in the system.

The rest of the paper is organized as follows. The next section presents and analyses background knowledge used in the methodology. Sect. 3 describes interoperability notation adjusted for service selection purposes. Sect. 4 presents extensions for selection algorithms in the graph-based and combinatorial models. System implementation is presented in Sect. 5. Finally, Sect. 6 presents related work and Sect. 7 concludes the paper.

2 System Model and Background

Service selection during development of composite services is a mature research discipline that has already developed models and solutions for various application areas. In this research, existing models are extended with issues specific for interoperability analysis, considering that interoperability may limit the choice in concrete situations.

The formal model of a composite service may be applied in concrete application areas such as: development of WS*-based services using the SOA approach, development of cloud-based applications and business process modeling (BPM) and execution using workflow management systems.

2.1 Composite Service Model

Typically, the model of a composite service (CS) takes the following assumptions [1], [5], [2]:

- A composite service consists of N atomic operations composed using composition structures (e.g.: loops, conditions, forks, joins). Each operation is realized by a service from a service class ($S_1, ..., S_N$), a service class is also called an abstract service.

- For each service class S_i, there exist one or more atomic services (s_{ij}) that realize the same functionality, but differ in non-functional attributes. Each service s_{ij} is associated with a QoS vector $q_{ij} = [q_{ij}^1, ..., q_{ij}^n]$ of non-functional attributes, e.g.: service price, performance and security.
- There is a defined utility function (F) that is the evaluation criteria of service selection on its QoS attributes. The utility function depends on user preferences considering the non-functional attributes. The user intends to minimize attributes such as price and maximize performance and reliability with given weights for each attribute.

The QoS value of the final CS depends on service selection and the structure of the CS. The calculation considers probability of execution in multiple execution paths, loop unfolding and others. Considering that QoS values are expressed in different metric types, a form of normalization is performed to calculate the utility function F. Attributes receive positive and negative weights depending on whether they are desired or undesired as QoS of the service as described in detail in [6], [1].

Let pos denote the number of attributes to be maximized and neg denote the number of attributes to be minimized, then:

$$F_{ij} = \sum_{a=1}^{pos} w_a * (\frac{q_{ij}^a - \mu^a}{\sigma^a}) + \sum_{b=1}^{neg} w_b * (1 - \frac{q_{ij}^b - \mu^b}{\sigma^b}) \quad (1)$$

where we denote: w - attribute weight, μ - average, σ - standard deviation. Additionally, $0 < w_a, w_b < 1$ and $\sum_{a=1}^{pos} w_a + \sum_{b=1}^{neg} w_b = 1$ [1].

Additionally, it is required that the overall QoS value of the composite service does not exceed a specified limit (e.g. a price constraint), which is known as the QoS constraint and defined as:

$$Q_c = [Q_c^1, Q_c^2, ..., Q_c^m] \quad (2)$$

Service selection process intends to select services such that

1. The utility function F of the composite service is maximized
2. Q_c constraints are satisfied for the composite service ($Q_{cs}^a \leq Q_c^a, \forall_{Q^a \in Q_c}$)

2.2 Existing Approaches to Service Selection

The service selection problem with QoS constraints has exponential computational complexity in the general case [5], [15]. Different solutions have been proposed that either narrow service structure to make it feasible for optimal calculations or apply a heuristic algorithm. Existing solutions, however, do not integrate the concept of interoperability analysis with service selection based on QoS attributes.

This work intends to introduce interoperability constraints to existing algorithms without causing the necessity of major changes in algorithm structures. In order to cover a possibly wide range of algorithms, the following features are considered:

- Computational Model. Two well-known models are used in service selection: the combinatorial model - 0-1 multidimensional multichoice knapsack problem (MMKP), e.g.: [7], [2], and the graph model - multiconstraint optimal path selection (MCOP), e.g.: [12], [8]. The work [1] analyzes both approaches.

Table 1. Selected selection algorithms and their characteristics

Algorithm	Computational model	Optimization scope and accuracy
BBLP, WS_IP [1]	Combinatorial	Global optimal
MCSP [1]	Graph	Global optimal
RWSCS_KP [2]	Combinatorial	Global heuristic
WS_HEU, WFlow [1]	Combinatorial	Global heuristic
MCSP-K [1]	Graph	Global heuristic
SEWSCP [7]	Combinatorial	Local optimal
ACO4WS [8]	Graph	Global heuristic (AI)
PSO [9]	Graph	Global heuristic (AI)
DaC [10]	Combinatorial	Local optimal
LOSS/GAIN [11]	Graph	Global heuristic
GA [12]	Graph	Global heuristic (AI)
FastHeu [13]	Graph	Local heuristic (DYN)
MILP BeesyCluster [13]	Graph	Local optimal (DYN)
DIST_HEU [5]	Combinatorial	Local optimal
QCDSS[14]	Graph	Global heuristic (AI)

- Accuracy. Optimal algorithms are applicable for small services, e.g. [1] proposes an algorithm based on the branch-and-bound linear programming. For large services, heuristic algorithms are proposed, such as the shortest-path based MCSP-K algorithm [1] or the combinatorial-based RWSCS_RP algorithm [2]. Artificial intelligence (AI) heuristic methods are also used, e.g. the ant-colony based ACO4WS algorithm [8] and particle swarm optimization [9], [14]. Some solutions propose dynamic (DYN) optimization [13].
- Optimization Scope. The reduction of optimization scope is another method of solving the problem. In this approach, the composite service is divided into subservices that are analyzed locally, typically using an optimal algorithm, e.g.: SEWSCP [7], DIST_HEU [5] and DaC [10].

Table 1 shows selected algorithms and their characteristics.

In the MMKP/combinatorial model [2], [1], [16], service classes are mapped to item groups, while atomic services are mapped to items in the groups. Each atomic service (s_{ij}) requires resources q_{ij} and supplies profit F_{ij}. The Q_c constraints are treated as available resource in the knapsack. The problem is formulated formally as:

$$\begin{aligned} Max & \sum_{i=1}^{N} \sum_{j \in S_i} F_{ij} x_{ij} \\ subject\ to & \sum_{i=1}^{N} \sum_{j \in S_i} q_{ij}^a * x_{ij} \leq Q_c^a \ (a = 1, ...m) \\ & \sum_{j \in S_i} x_{ij} = 1 \\ & x_{ij} \in \{0, 1\}\ i = 1, ..., N,\ j \in S_i \end{aligned} \tag{3}$$

where $x_{ij} = 1$ if service s_{ij} is selected for the solution and $x_{ij} = 0$ otherwise. The problem may be solved using integer linear programming (ILP) methods [15].

In the graph-based model [1], every atomic service is a node in the graph and potential transitions between services are considered as links. If a service class S_i sends data to a service class S_k then there is a link between every service from S_i to every service from S_j. The node that represents a service s is assigned with QoS attributes of the service and consequently with service utility function F. QoS attributes of a service are added to every link incoming to the node that represents the service. The multiconstraint optimal path problem intends to find a path such that the utility function F is maximized while meeting the Q_c constraints.

If QoS constraints are removed from the selection problem, a simplified model is created, in which services are selected considering solely the utility function F. In this case, local optimization for each service class S_i may be used to select the service that supplies the highest utility value. Despite the simplification, interoperability needs to be considered in this case too, as the communication model of the final composite service remains unchanged.

2.3 Interoperability in Service Integration

Service integration in SOA requires resolution of interoperability issues. Web services standards (WS*) [17] were proposed and accepted as the main solution for achieving the task. Currently, there exist approximately fifty standards in different versions apart from the basic SOAP and WSDL. Standards constitute the WS* stack [18], in which standards for an extended functionality rely on lower level ones. Main functional groups include the areas of: addressing (WS-Addressing), security (e.g. WS-Security), reliable messaging (e.g WS-Reliablility), transactions (e.g WS-Coordination, WS-AtomicTransactions) and others. Business process modeling and execution standards (e.g. Business Process Execution Language [19]) leverage existing WS* standards assuming that lower layer standards may be used for communication with atomic services. Despite the general success of WS*, effective service integration using WS* is still a challenging task [4], [3].

General purpose interoperability metrics have been proposed as universal means of interoperability rating. Typically, the metrics specify interoperability levels that are associated with a required scope of data exchange [20]. For example, Layers of Coalition Interoperability (LCI) define nine layers: Physical, Protocol, Data/Object Model, Information, Knowledge/Awareness, Aligned Procedures, Aligned Operations, Harmonized/Strategy Doctrines, and Political Objectives. Sect. 6 presents more detailed discussion concerning existing solutions and methods.

3 Interoperability Notation for Service Selection

In our work, interoperability is analyzed on the communication protocol level, which corresponds to Level 1 (Connected) in the LISI metric or Level 2 (Data/Object) in the LCI metric [20]. We assume that if services are interoperable on that level, it is possible to exchange data between them in the context of CS construction.

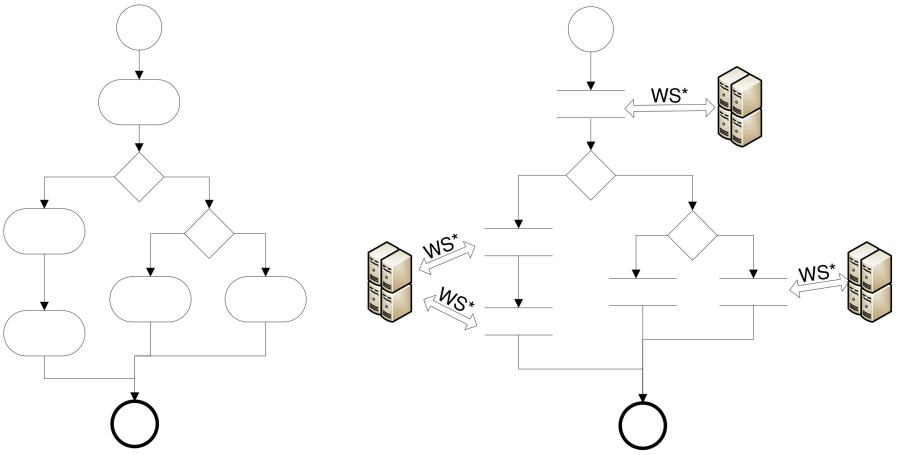


Fig. 1. Differences between message passing model (a) and remote method invocation model (b)

For the purpose of this work, we make the following assumptions:

- There exist different runtime environments (denoted $R = \{r_1, ..., r_P\}$ that may host services. Each service s is hosted in a runtime environment $r(s) \in R$ that is unambiguously determined by the atomic service s.
- Each runtime environment supports a known subset of existing WS* standards and their versions.
- Each service s requires some WS* standards (in specified versions) to operate correctly. The standards are supported by $r(s)$.

The number of runtime environments is limited, considering that they are application servers supplied by leading industrial companies or communities. Examples of runtime environments include: IBM WebSphere AS, Microsoft IIS, Apache Tomcat/Axis and Oracle Glassfish.

If a service s communicates remotely with a service s' than it is required that s' can interoperate with s. We rate interoperability binary as either interoperable (if data can be transfered correctly) or non-interoperable (if data can not be transfered). Assuming that s sends data to s', let $q^{comm}(s, s')$ denote interoperability modeled as an attribute of communication cost between s and s', defined as follows:

$$\begin{aligned} q^{comm}(s, s') &= 0 \; if \; s, s' \; can \; interoperate \\ q^{comm}(s, s') &= 1 \; if \; s, s' \; can \; not \; interoperate \end{aligned} \tag{4}$$

In this notation, interoperability cost 0 denotes that services are interoperable while interoperability cost 1 denotes that they are not interoperable.

The q^{comm} value is given a priori and depends on conditions encountered in the real environments that host the services. Information about service interoperability is taken from two complementary sources:

- Experimental Verification. A test connection is established between s and $r(s')$ to check if data can be transfered correctly within required standards.

– Theoretical Calculation. The calculation compares standards required by s and supported by $r(s')$. r' must support standards required by s if s and s' are to be interoperable. The method is easier to apply but less reliable than the first one, as declared standard compliance does not inferr compatibility [4].

The flow of data between services depends on the applied communication model. Two well-known models are considered [21] as shown in Fig. 1:

– remote method invocation model (RMI)
– message-passing model (MPM)

In the RMI model, there is a central runtime environment that hosts the composite service, denoted ($r(CS)$), and the runtime manages data exchange during invocation of remote atomic services. The model is typically used in the BPM technology by Business Process Execution Language (BPEL) [19] and Windows Workflow Foundation (the XPDL format). In the message-passing model, services send data directly from one to another without a central coordination point. The Business Process Modeling Notation standard [22] uses this model to describe the logic of a business process.

4 Algorithm Extensions

The proposed extensions of existing models with interoperability constraints enable a possibly straightforward application of existing selection algorithms. We consider both RMI and MPM, as well as graph-based and combinatorial-based algorithms.

4.1 Extensions for Remote Method Invocation Model

In the RMI model, we assume that CS may be hosted on one of alternative runtime environments. Let $R^{cs} = \{r_1^{cs}, ..., r_T^{cs}\}$ denote the set of alternative environments for CS. Each service runtime environment $r(s)$ must interoperate with the chosen runtime environment for the CS. The presented algorithm is applicable if a workflow is defined using the BPEL or XPDL standards.

Let $q^{comm,cs}(s, cs, r^{cs})$ be defined analogously to q^{comm} that is

$$\begin{aligned} q^{comm,cs}(s, cs, r^{cs}) &= 0 \; if \; s \; can \; interoperate \; with \\ &\qquad cs \; hosted \; on \; r^{cs} \\ q^{comm,cs}(s, cs, r^{cs}) &= 1 \; if \; s \; can \; not \; interoperate \; with \\ &\qquad cs \; hosted \; or \; r^{cs} \end{aligned} \tag{5}$$

Let $Q^{comm,cs}$ be the set of interoperability information of $q^{comm,cs}(s_{ij}, cs, r_t^{cs})$ for each s_{ij} $i = 1..N$, $j \in S_i$, $r_t^{cs} \in R^{cs}$.

The processing is done as shown in Algorithm 1. The algorithm analyzes in a loop each alternative runtime environment of CS. For each r^{cs}, it performs a pre-filtering of services that are interoperable with the r^{cs}. Then, one optimal solution is calculated for each r^{cs} using an algorithm (SEL_ALG) chosen from existing selection algorithms. Both combinatorial-based and graph-based selection algorithms are applicable in this

Algorithm 1. Service selection using an existing algorithm and interoperability constraints (RMI model)

input: $R^{cs}, S_1, ...S_N, Q^{comm,cs}, F, SEL\_ALG$ - an existing algorithm
output: $[s_{1a_1}, ..., s_{Na_N}]$, $a_i \in S_i$ //optimal selection
  $tmpSelection = \varnothing$ //list of optimal selections for each $r^{cs} \in R^{cs}$
  for all ($r_t^{cs} \in R^{cs}$) **do**
    for all ($S_i, i = 1, ..., N$) **do**
      $S_i^{r_t^{cs}}$ = select all s_{ij} from S_i such that $q^{comm,cs}(s_{ij}, r_t^{cs}) = 0$
      if ($S_i^{r_t^{cs}} == \varnothing$) **then**
        //there are no services in S_i that can interoperate with r_t^{cs},
        //no feasible solution for runtime environment r_t^{cs}
        continue with next r_t^{cs}
      end if
    end for
    $tmpSelection+ = SEL\_ALG\ ((S_i^{r_t^{cs}}, i = 1..N))$
  end for
  $maxTmpUtility = 0$
  for all ($sel \in tmpSelection$) **do**
    $tmpUtility = F(sel)$
    if ($tmpUtility > maxTmpUtility$) **then**
      $maxTmpUtility = tmpUtility; result = sel$
    end if
  end for
  return $result$

invocation model. After processing of all r^{cs}, the optimal solution for the whole set R^{cs} is selected Interoperability constraints introduce a relatively low computational complexity in the RMI model. Selection of interoperable services depends on $O(sizeof(S) * sizeof(R^{sc}))$. In practice, the size of $R^{sc} = T$ may be considered as constant because it does not depend on CS size and there is a limited number of industrially-running environments that may host services. Generation of $tmpSelection$ takes $T * O(SEL\_ALG)$, which does not exceed the computational complexity of SEL_ALG. The final step (selection of the best solution from R^{sc}) takes less than $T * timeof(F)$, which is certainly lower that SEL_ALG, as the algorithms invoke F for each service and for the whole workflow. Therefore, $O(SEL\_ALG)$ is the factor that determines the final computational complexity of Algorithm 1.

4.2 Extensions for Message Passing, Combinatorial Model

In this model, if a service s communicates with a service s', then $r(s)$ must interoperate with $r(s')$. The model is applicable for workflows defined using the BPMN standard. As stated in Sect. 3 $q^{comm}(s_{ij}, s_{kl})$ denotes interoperability between s_{ij}, s_{kl}, $s_{ij} \in S_i$, $s_{kl} \in S_k$, $i, k = 1, ..., N$, $j \in S_i$, $l \in S_k$. We extend the existing combinatorial model and its corresponding integer linear programming representation with additional constraint equations that represent interoperability:

$$\sum_{j \in S_i} \sum_{l \in S_k} q^{comm}_{s_{ij}, s_{kl}} * (x_{ij} * x_{kl}) = 0 \tag{6}$$

for all pairs S_i, S_k such that S_i sends data to S_k in the structure of CS. It is required that if two services are selected ($x_{ij} = 1$ and $x_{kl} = 1$), then their interoperability communication cost $q^{comm}_{s_{ij},s_{kl}} = 0$. If two service classes do not pass data, the q^{comm} value is insignificant and may be ignored.

The number of added equations is equivalent to the number of edges in the composite service graph, which depends linearly on the number of services (N). The maximum number of elements in each equation is $Max(sizeof(S_i)) * Max(sizeof(S_i))$ for $i = 1...N$. We assume that the maximum number of alternative services for one service class does not exceed a constant ($MaxS_i$).

Equation 6, however, introduces non-linearity, so the problem is changed from ILP to mixed integer nonlinear programming (MINLP), which makes it significantly more difficult to solve. We transform the equation into a separable programming problem, which is a special class of MINLP that is relatively easily solvable by existing methods [15]. In order to achieve the separable programming form, we perform the following transformation:

$$\begin{aligned} & x_{ij} * x_{kl} = y^2_{1,ijkl} - y^2_{2,ijkl} \\ where \;\; & y^2_{1,ijkl} = 0.5(x_{ij} + x_{kl}) \\ & y^2_{2,ijkl} = 0.5(x_{ij} - x_{kl}) \end{aligned} \tag{7}$$

for each $j \in S_i$, $l \in S_k$. After the transformation, additional y variables are added to problem formulation. The number of y variables equals the total number of elements in Equation 6 and has the complexity of $O(N * MaxS_i * MaxS_i)$. The newly created model is more complex than the original one, but it remains solvable by existing methods such as branch-and-bound or implicit enumeration [15]. The model is applicable for service selection methods that leverage (N)LP to find an optimal solution, that is for methods based on local optimization (DaC [10]) or applied to small composite services.

4.3 Extensions for Message Passing, Graph-Based Model

In this model, we extend the composite service graph with additional information regarding interoperability constraints. Analogously to the combinatorial approach, the model is applicable for workflows defined using the BPMN standard. We use the previously defined q^{comm} attribute to represent interoperability communication cost. The processing is done as follows:

- Assuming that a service class S_i communicates with a service class S_k, the $q^{comm}(s_{ij}, s_{kl})$ attribute is assigned to each link from a service $s_{ij} \in S_i$ to a service $s_{kl} \in S_k$. The attribute has the value 0 if services are interoperable and the value 1 in the other case.
- The Q_c vector is extended with an additional element Q^{comm}_c that represents interoperability, that is:

$$\begin{aligned} & Q'_c = [Q^1_c, Q^2_c, ..., Q^m_c, Q^{comm}_c] \\ & Q^{comm}_c = 0 \end{aligned} \tag{8}$$

where Q_c^{comm} for a service selection (a $path$ in the graph) is calculated as follows:

$$Q_c^{comm}(path) = \sum_{link \in path} q_{link}^{comm} \tag{9}$$

that is: every edge on the selected path must have the communication cost 0.

The proposed model does not change the graph structure and the structure of Q_c constraints. Therefore, it is possible to apply both optimal and heuristic existing algorithms that are based on graph structure processing, such as MSCP, MSCP_K or ACO4WS [1] [8].

In this model, interoperability analysis requires much lower computational cost as compared with the combinatorial approach. One additional attribute is added to the QoS vector on the service level and on the constraints level. This requires additional computations during graph preparation and during service selection, but does not change the computational complexity of algorithms. Therefore, the graph-based algorithms are more adequate for interoperability analysis in the message passing model.

5 System Implementation

As a part of the research, we implemented the WorkflowIntegrator system that enables selection of services during composite service development. The aim of the system is to support developers in the selection process depending on given requirements on utility function, QoS attributes, and interoperability constraints. The system is available online at: http://kask.eti.pg.gda.pl/WorkflowFTI

Fig. 2 shows a screenshot of the WorkflowIntegrator system with summary results of an exemplary service selection using cost and availability as selection attributes.

We use workflow applications as a concrete implementation of the composite service concept, which enables us to leverage existing solutions in service description and invocation.

In the workflow methodology, a workflow application corresponds to a complex services, while a service available through Web services corresponds to an atomic service. Concrete runtime environments are Workflow Management Systems (WfMS) for the workflow application, and Application Servers for atomic services. Three main standards are used for service description: Business Process Execution Language (BPEL), Business Process Modeling Notation (BPMN) and the Windows Workflow Foundation (XPDL format).

The system supplies the ASInteroperability module that stores general purpose interoperability ratings of runtime environments [23] available also at:

http://kask.eti.pg.gda.pl/ASInteroperability

The ASInteroperability module contains a systematic description of existing WS* standards, their versions, configuration options and acceptable values for the options. For example WS-AtomicTransaction anticipate the Commit Protocol option with values: Completion, Two-Phase Commit (Volataile), Two-Phase Commit (Durable). Popular existing runtime environments have been described in the system including: name, vendor, version and used libraries. The runtime description model enables dynamic association between runtime environments and dependant libraries.

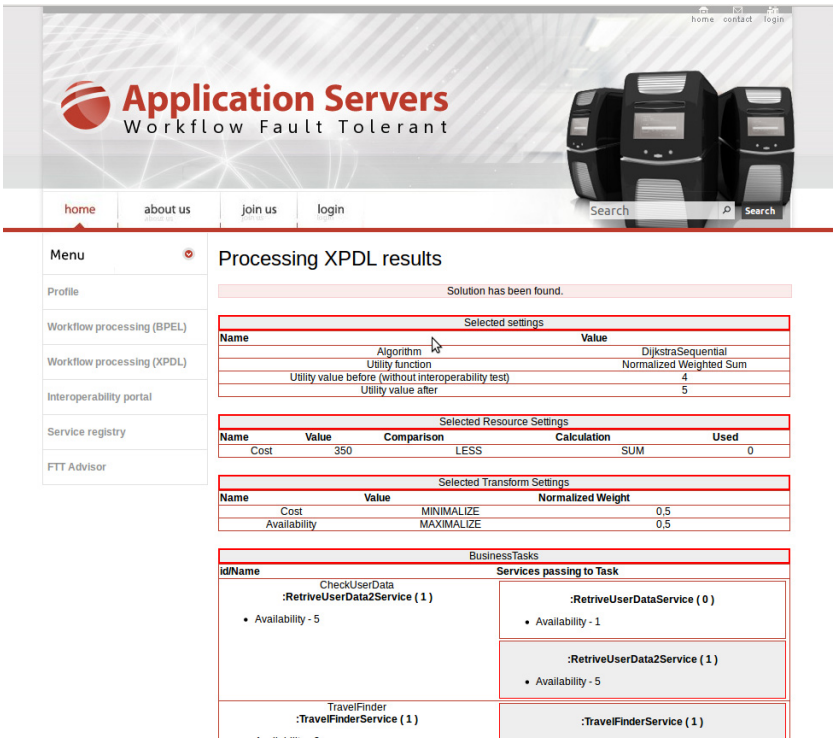


Fig. 2. A screenshot of the WorkflowIntegrator system with summary results of an exemplary service selection

Interoperability rates are given to concrete configurations in which integration is attempted. Each configuration covers specification of: the two integrated runtime environments, used WS* standards, standard versions, significant configuration options and values settled for the options. Rates are given by developers who have already attempted to establish a concrete integration and share the knowledge with the community.

Additionally, a service registry is defined that contains specification of services together with their location, QoS attributes, the runtime environment that hosts the service, and used WS* standards. Additionally, it is possible to specify groups of alternative services, which is the main function of the registry in the context of service selection during service composition. Groups of alternative services are configured manually by developers of the composite service.

Using ASInteroperability and Service Registry, the module of Workflow Fault Tolerant Integrator (WorkflowFTI) supports design of optimal workflow applications depending on the following information acquired from the developer:

1. The developer defines groups of alternative services that correspond to service classes in the CS model. Each service contains description of QoS attributes, used WS* standards and the application server that hosts the service.

2. The developer specifies the utility function that should be optimized in the form of weights for QoS attributes and the method for attribute aggregation (average, sum, max, min and others).
3. The developer specifies which selection algorithm should be used.
4. The user supplies a defined workflow application together with initially assigned services for tasks. The assignment aims at specifying which service group should be used for processing each task.

After the configuration, WorkflowFTI uses the chosen algorithm to select appropriate services and returns the result. During the selection, it considers the relevant information such as: interoperability constraints, QoS attributes, utility function and others.

The system contains prototypical implementations of representative existing algorithms extending them with relevant interoperability processing. Both RMI and message passing models are analyzed during the work. The BPEL standard was used as a representation of the RMI model, while the BPMN standard (using the XPDL format) was used as a representation of the message passing model. In the message passing model, interoperability between runtime environments of individual services is analyzed during the selection process. The proposed methodology was implemented using both optimal and approximate existing algorithms. Optimal algorithms include: explicit enumeration based on ILP and a simplified version of the MSCP algorithm. Additionally, we implemented a greedy algorithm as an example of the approximate approach. In the RMI model, services are pre-filtered considering their interoperability with each WfMS, and services supplying the best QoS are selected. We verified interoperability analysis using an optimal algorithm based on explicit enumeration, and an approximate algorithm based on local optimization.

The system is implemented in the .NET 3.5 environment and hosted on Microsoft Internet Information Service. We used the MSSQL database for storing data regarding interoperability description and service registry. A web interface is supplied for users intending to design a workflow application or verify interoperability between application servers. Internally, the system uses Web services to integrate ASInteroperability, Service Registry and WorkflowFTI.

6 Related Work

Sect. 2 presented core concepts regarding the problem addressed in this paper. Apart from the background, various detailed solutions have been proposed. [24] presents an interoperability assessment model for service composition using five interoperability levels: signature, protocol, semantic, quality and context. The assessment is based on a formal evaluation metric that assigns a rate in the range $[0, 1]$ for each level depending on defined conditions. The work [25] addresses a similar issue assuming that components-off-the-shelf are integrated. The work specifies groups of attributes that are used for interoperability analysis, for example: control input technique, data communication protocols, encapsulation of behavior and others. Using the model, authors of that paper propose an interoperability assessment tool. Both solutions do not analyze QoS attributes and constraints, such as price, performance, and reliability in service composition.

The Web Services Interoperability (WS-I) Organization [18] has been established to refine existing standards and promote integration of heterogeneous environments. The organization issues WS-Profiles that impose additional constraints on communication formats, which enables higher interoperability. [26] surveys standards and solutions related to business process modeling and execution. The work discusses correlation between standards related to business process modeling and formal methods of representation. [13] presents the BeesyCluster execution environment that enables definition and execution of business processes with service replacement and dynamic optimization.

The concept of Functional Quality of Service (FQoS) is introduced in [27]. FQoS describes functional attributes of Web services using mathematical methods to manipulate and quantify those attributes. Authors analyze FQoS on three levels: term similarity, tree similarity and text similarity. [28] proposes to use Petri nets for compatibility analysis during composition of BPEL applications. Authors present a detailed association between Petri net concepts and BPEL control constructs and define a formal procedure of Web services composition. The papers focus mainly on service compatibility analysis rather than selection based on non-functional QoS attributes.

[29] and [30] present frameworks that enable interoperability rating and analysis in Web services-based systems. Both papers propose service metamodels that cover core service description concepts applicable on the enterprise or on the P2P level. Using the metamodels, interoperability of services is rated in the considered scope. [31] is another work that rates interoperability in Web services-based solutions. The work proposes a formal description model that covers peers and operations, such as send, receive and transition. Results achieved using the proposed methods may be beneficial in our work as an additional source of information about service interoperability.

7 Conclusions

The presented methodology enables application of existing service selection algorithms in cases where interoperability issues must be considered. The remote method invocation model suits best to the proposed solution allowing us to use any of the existing algorithms with minor development and computational effort. The message passing model favors graph-based selection algorithms (both optimal and heuristic) allowing us to preserve the general structure of algorithm data model. The application of the methodology in the message passing model for combinatorial algorithms faces difficulties because of non-linearity of constraint equations. The problem is solvable by existing methods, but requires additional computational effort. As a part of the research, implementation work was performed for a representative set of composite service models and algorithm approaches, which verified the presented approach.

Extension of heuristic algorithms based on the combinatorial model is an interesting area of future work, considering that the algorithms may use proprietary data processing that does not rely directly on the defined ILP representation of the problem. Application of the methodology in industrial projects is another area of future work. The current work covered implementation of the methodology in representative algorithms, but the use of it in industrial projects may give further guidelines for extensions and

adjustments. The projects may further supply interesting statistical data, such as the ratio of interoperable services, market share of runtime environments, and differences in QoS attributes of services, which will be beneficial in further improvements of service selection methods.

Acknowledgements. The work was supported in part by the Polish Ministry of Science and Higher Education under research projects N N519 172337. The author would like to thank students of the Faculty of ETI for implementation and testing work regarding the system for service selection.

References

1. Yu, T., Zhang, Y., Lin, K.J.: Efficient algorithms for web services selection with end-to-end qos constraints. ACM Transactions on the Web 1 (2007)
2. Cao, H., Feng, X., Sun, Y., Zhang, Z., Wu, Q.: A service selection model with multiple qos constraints on the mmkp. In: IFIP International Conference on Network and Parallel Computing (2007)
3. Fisher, M., Lai, R., Sharma, S., Moroney, L.: Java EE and .NET Interoperability: Integration Strategies, Patterns, and Best Practices. FT Prentice Hall (2006)
4. Egyedi, T.M.: Standard-compliant, but incompatible?! Computer Standards & Interfaces 29(6), 605–613 (2007)
5. Alrifai, M., Risse, T., Dolog, P., Nejdl, W.: A scalable approach for qoS-based web service selection. In: Feuerlicht, G., Lamersdorf, W. (eds.) ICSOC 2008. LNCS, vol. 5472, pp. 190–199. Springer, Heidelberg (2009)
6. Zeng, L., Benatallah, B., Dumas, M., Kalagnanam, J., Sheng, Q.Z.: Quality driven web services composition. In: Proceedings of the 12th International Conference on World Wide Web, WWW 2003 (2003)
7. Hong, L., Hu, J.: A multi-dimension qos based local service selection model for service composition. Journal of Networks 4 (2009)
8. Wang, X.L., Jing, Z., Zhou Yang, H.: Service selection constraint model and optimization algorithm for web service composition. Information Technology Journal (2001)
9. Xia, H., Chen, Y., Li, Z., Gao, H., Chen, Y.: Web service selection algorithm based on particle swarm optimization. In: Eighth IEEE International Conference on Dependable, Autonomic and Secure Computing (2009)
10. Yu, J., Buyya, R., Tham, C.K.: Cost-based scheduling of workflow applications on utility grids. In: IEEE International Conference on e-Science and Grid Computing, e-Science (2006)
11. Sakellariou, R., Zhao, H., Tsiakkouri, E., Dikaiakos, M.D.: Scheduling workflows with budget constraints. In: Integrated Research in GRID Computing (CoreGRID Integration Workshop 2005, Selected Papers) (2007)
12. Cao, L., Li, M., Cao, J.: Using genetic algorithm to implement cost-driven web service selection. Multiagent and Grid Systems - An International Journal 3 (2007)
13. Czarnul, P.: Modeling, run-time optimization and execution of distributed workflow applications in the jee-based beesycluster environment. The Journal of Supercomputing, 1–26 (2010)
14. Chun-hua, H., Xiao-hong, C., Xi-ming, L.: Dynamic services selection algorithm in web services composition supporting cross-enterprises collaboration. Cent. South Univ. Technol (2009)
15. Bradley, S.P., Hax, A.C., Magnati, T.L.: Applied Mathematical Programming. Addison-Wesley (1977)

16. Martello, S., Toth, P.: Algorithms for knapsack problems. Annals of Discrete Mathematics (1987)
17. Booth, D., Haas, H., McCabe, F., Newcomer, E., Champion, M., Ferris, C., Orchard, D.: Web Services Architecture, Working Group Note. W3C (2004)
18. WS-I: Interoperability: Ensuring the Success of Web Services. Web Services Interoperability Consortium (2004)
19. Oasis: Web Services Business Process Execution Language Version 2.0 (2007)
20. Ford, T., Colombi, J., Graham, S., Jacques, D.: A survey on interoperability measurement. In: 12th International Command and Control Research and Technology Symposium (ICCRTS) Adapting C2 to the 21st Century (2007)
21. Tanenbaum, A.S., van Steen, M.: Distributed Systems Principles and Paradigms. Prentice Hall (2002)
22. OMG: Business Process Model and Notation 2.0 Beta 1 Specification. Object Modeling Group (2009), http://www.omg.org/cgi-bin/doc?dtc/09-08-14
23. Kaczmarek, P.L., Nowakowski, M.: A developer's view of application servers interoperability. In: Wyrzykowski, R., Dongarra, J., Karczewski, K., Waśniewski, J. (eds.) PPAM 2011, Part II. LNCS, vol. 7204, pp. 638–647. Springer, Heidelberg (2012)
24. Fang, J., Hu, S., Han, Y.: A service interoperability assessment model for service composition. In: IEEE International Conference on Services Computing, pp. 153–158 (2004)
25. Bhuta, J., Boehm, B.: Attribute-based cots product interoperability assessment. In: Sixth International IEEE Conference on Commercial-off-the-Shelf (COTS)-Based Software Systems (2007)
26. van der Aalst, W.M.P., ter Hofstede, A.H.M., Weske, M.: Business process management: A survey. In: van der Aalst, W.M.P., ter Hofstede, A.H.M., Weske, M. (eds.) BPM 2003. LNCS, vol. 2678, pp. 1–12. Springer, Heidelberg (2003)
27. Jeong, B., Cho, H., Lee, C.: On the functional quality of service (fqos) to discover and compose interoperable web services. Expert Systems with Applications 36 (2009)
28. Tan, W., Fan, Y., Zhou, M.: A petri net-based method for compatibility analysis and composition of web services in business process execution language. IEEE Transactions on Automation Science and Engineering 6 (2009)
29. Ullberg, J., Lagerström, R., Johnson, P.: A framework for service interoperability analysis using enterprise architecture models. In: IEEE SCC (2), pp. 99–107 (2008)
30. Tsalgatidou, A., Athanasopoulos, G., Pantazoglou, M.: Interoperability among heterogeneous services: The case of integration of p2p services with web services. Int. J. Web Service Res. 5, 79–110 (2008)
31. Fu, X., Bultan, T., Su, J.: Synchronizability of conversations among web services. IEEE Transactions on Software Engineering 31, 1042–1055 (2005)

Correlation of Business Activities Executed in Legacy Information Systems

Ricardo Pérez-Castillo[1], Barbara Weber[2], and Mario Piattini[1]

[1] Instituto de Tecnologías y Sistemas de Información (ITSI), University of Castilla-La Mancha, Paseo de la Universidad 4, 13071, Ciudad Real, Spain
{ricardo.pdelcastillo,mario.piattini}@uclm.es
[2] University of Innsbruck, Technikerstraße 21a, 6020, Innsbruck, Austria
barbara.weber@uibk.ac.at

Abstract. Reverse engineering techniques are commonly applied for discovering the underlying business processes. These techniques often rely on event logs recorded by process-aware information systems. Apart from these, there are many non-process-aware systems without mechanisms for recording events. Certain techniques for collecting events during the execution of non-process-aware systems have been proposed to enable the discovery of business processes from this kind of systems. In these techniques the correlation of events into their execution instance constitutes a big challenge since the business process definitions supported by non-process systems are implicit. This paper presents a correlation algorithm which works together a technique for obtaining event logs from non-process-aware systems. The event correlation algorithm is applied to the events dataset collected at runtime to discover the best correlation conditions. Event logs are then built using such conditions. The applicability of the proposal is demonstrated through a case study with a real-life system.

Keywords: Enterprise Modeling, Business Process, Event Correlation, Legacy Information Systems.

1 Introduction

Current companies must continuously evolve to maintain their competitiveness levels. Keeping this in mind, process modelling is essential for companies to be able to understand, manage and adapt their business processes [17]. Despite this important fact, a vast amount of companies do not model their business processes. When these companies decide in favor of business process modelling, they have two main options: (i) modelling from scratch by business experts, which is time-consuming and error-prone; (ii) using business process mining techniques to discover business processes from system execution information [15]. The focus of this paper is on the second option since it takes into account the business knowledge embedded in enterprise information systems.

Business process mining techniques allow for extracting information from process execution logs –known as event logs [15]. Event logs contain information about the start and completion of activities and the resources executed by the processes [2].

L.A. Maciaszek and J. Filipe (Eds.): ENASE 2012, CCIS 410, pp. 48–63, 2013.

These events logs are often recorded by process-aware information systems (PAIS) (e.g., Enterprise Resource Planning (ERP) or Customer Relationship Management (CRM) systems). The process-aware nature of PAIS facilitates direct events recording during process execution. However, not all information systems are process-aware. In fact, there is a vast amount of enterprise information systems which are non-process-aware (termed as traditional systems in this paper) though they could also benefit from the application of process mining techniques.

Previous works made a particular effort for the registration of event logs from traditional systems. Firstly, main challenges involved in the collection of event logs from traditional systems were identified [11]. Secondly, a particular technique to obtain event logs from traditional systems was also developed [12]. This technique first injects statements into the source code to instrument it. The instrumented system is then able to record some events, which are finally analysed and organized in an event log. This technique has already been applied to real-life traditional systems with promising results [12, 13]. However, the accuracy of the previous technique is limited, since events were correlated in different process instances by using some simple heuristics. In fact, event correlation is a key challenge, i.e., each event must be assigned to the correct instance, since it is possible to have various instances of the business process running at the same time. This paper therefore addresses the weaknesses on the previous technique by providing an enhanced technique, which adapts and applies an existing event correlation algorithm [8].

The new technique relies on human interaction to firstly identify some candidate correlation attributes. The candidate attribute values are then recorded with each event by means of an instrumented version from a traditional system. After that, event correlation algorithms proposed in a related work [8] are adapted and applied to this intermediate information to discover the sub-set of correlation attributes and conditions. The correlation set is finally used to obtain accurate process instances in the final event log. For a smoother introduction into industry, this technique has been developed by using database-stored intermediate information as well as a set of algorithms implemented as stored procedures. Since the technique is tool-supported, a case study involving a real-life traditional system has been conducted to demonstrate the feasibility of the proposal. The empirical validation results show the technique is able to obtain the set of correlation attributes allowing appropriate event logs in a moderate time.

The paper is organized as follows: Section 2 summarizes related work; Section 3 presents in detail the proposed technique; Section 4 conducts a case study with an author management system; and Section 5 discusses conclusions and future work.

2 Related Work

Event correlation is an issue of growing importance in the process mining field due to the increasing heterogeneity and distribution of enterprise information systems. In addition, there are various ways in which process events could be correlated. In fact, many times event correlation is subjective and most proposals employ correlation heuristics [6]. Most techniques assess some indicators and check if they are under or above a heuristic threshold to discard non-promising correlation attributes.

For example, Burattin et al., [1] propose an approach consisting of the introduction of a set of extra fields, decorating each single activity log. These attributes are used to carry the information on the process instance. Algorithms are designed using relation algebra notions, to extract the most promising case IDs from the extra fields. Other techniques proposals, as in [14], are based in the use of algorithms to discover correlation rules by using assessments of statistic indicators (e.g., variance of attribute values) from datasets. Similarly, *Ferreira et al.* [3] propose a probabilistic approach to find the case ID in unlabeled event logs. Motahari-Nezhad et al., [8] propose a set of algorithms to discover the most appropriate correlation attributes and conditions (e.g., conjunctive and disjunctive conditions grouping two or more correlation attributes) from the available attributes of web services interaction logs.

This paper presents an improvement of a previous technique to retrieving event logs from traditional systems [12], by adapting and applying the algorithm to discover the correlation set provided by Motahari-Nezhad et al., [8]. While this algorithm is applied to web services logs, the current approach adapts the algorithm to be applied in traditional information systems for obtaining event logs. The feasibility of this approach is empirically validated by means of a case study.

Moreover, there are proposals addressing the distribution of heterogeneous event logs. For example, *Hammoud et al.* [5] presents a decentralized event correlation architecture. In addition, *Myers et al.* [9] apply generic distributed techniques in conjunction with existing log monitoring methodologies to get additional insights about event correlation. Decentralized event correlation approaches remain however outside of the scope of this paper.

3 Event Correlation

Event correlation deals with the definition of relationships between two or more events so to point out events belonging to a same business process execution (i.e., process instance). Event correlation is very important in traditional information systems, since the definitions of the executed business processes are not explicitly identified [11]. Fig. 1 shows an overview of the event correlation challenge. Each business process can be executed several times. Each execution is known as process instance. Events collected during system execution must be correlated into the correct instance.

This paper presents a technique to obtain event logs from traditional systems, paying special attention to the event correlation improvement regarding previous work. The technique consists of three stages. Firstly, the technique records events from the execution of traditional systems. During this stage the technique allows experts to identify candidate correlation attributes, whose runtime values will then be collected together with each event. As a result, events and their respective attributes are then stored in a database in an intermediate format (cf. Section 3.1). Secondly, the algorithm proposed by [8] is adapted and applied with the event datasets so to discover the most appropriate set of attributes and conditions for the events correlation (cf. Section 3.2). Finally, the third stage applies an algorithm based on the correlation set in order to correlate each event with its corresponding process instance (cf. Section 3.3). As a result, a standard-format event log is obtained from the source traditional system.

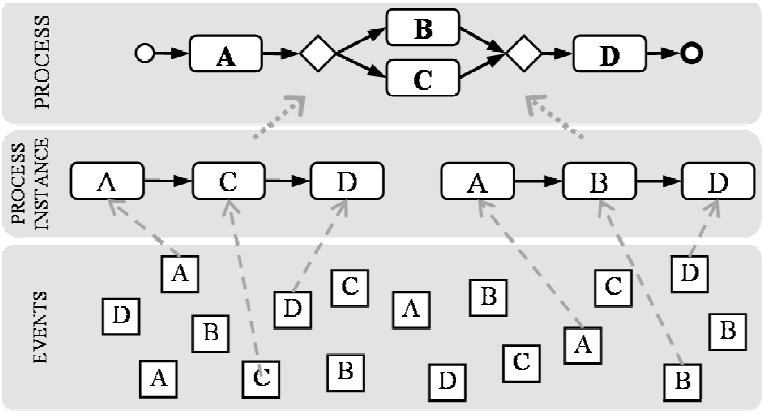


Fig. 1. Event correlation overview

3.1 Event Collection

The event collection stage is in charge of the suitable generation and storage of events throughout system execution. Since traditional information systems do not have any in-built mechanism to record events about executed business processes, this stage instruments information systems to record events. The instrumentation is semi-automated by a parser that syntactically analyzes the source code and injects statements in particular places of the code to record events during system execution.

This work follows the *'a callable unit / a business activity'* approach proposed by *Zou et al.* [18]. Callable units are the generic elements (e.g., Java methods, C or COBOL procedures, etc.) in which the parser injects statements to record an event corresponding to the execution of a business activity. Despite this fact, not all the executions of callable units have to be recorded as events. Some callable units such as fine-grained or technical callable units do not correspond to events and must be discarded. The injection in the exact place is consequently supported by some information provided by experts. Such experts (i) delimit business processes with the start and end callable units of each process; (ii) establish the boundaries of non-technical source code to be instrumented; and finally, (iii) they identify those code elements that can be treated as candidate correlation attributes.

This stage is supported by an improved version of the tool presented in [12], which supports the identification and addition of candidate correlation attributes. Selection of candidate correlation attributes is a key task, since an incomplete list of candidate attributes may lead to a non-suitable correlation. This stage provides experts with all the possible selectable attributes. These attributes are every parameter appearing in callable units as well as the output and fine-grained callable units that are invoked within those callable units, which are considered to be collected as events. The information about candidate correlation attributes is injected together with a statement enabling event collection.

Fig. 2 provides an example of the results obtained after instrumenting a Java method of the system under study (cf. Section 4). The two tracing statements (see highlighted statements) are injected at the beginning and at the end of the body of the

method. Those candidate correlation attributes that are present in a method (e.g., a parameter or variable) are automatically injected in the tracing statements, i.e., the respective variables are in the set of parameters of the invocation to the method 'writeDBEvent' (see Fig. 2). However, not all correlation attributes defined by experts are present in all methods (e.g., due to the absence of a particular variable). In this case, the respective parameter of the method 'writeDBEvent' (i.e., the tracing statement) is an empty string (" "). As a result, during execution of this method, the runtime value (or an empty value) will be recorded together with the name of the attribute and the event (the name of the method representing the business activity).

```
public class SocioFacadeDelegate {
    [...]
    public static void saveAuthor(AuthorVO author) throws InternalErrorException {
        writeDBEvent("SocioFacadeDelegate.saveAuthor", "Author Management", "", "start",
          false, false, -1, false, 2, 8, "", "" + author.getId(), "" + author.isHistorico(),
          "" + author.getNumeroSocio(), "", "" + author.getCotas());
        try {
            SaveAuthorAction action = new SaveAuthorAction(author);
            PlainActionProcessor.process(getPrivateDataSource(), action);
        } catch (InternalErrorException e) {
           throw e;
        } catch (Exception e) {
           throw new InternalErrorException(e);
        }
        writeDBEvent("SocioFacadeDelegate.saveAuthor", "Author Management", "", "complete",
          false, true, -1, false, 2, 8, "", "" + author.getId(), "" + author.isHistorico(),
          "" + author.getNumeroSocio(), "", "" + author.getCotas());
    }
    [...]
}
```

Fig. 2. Example of code instrumentation

The instrumented system is then normally executed and -when an injected statement is reached- it records events together with the value of all the candidate correlation attributes available in that callable unit. Unlike other similar techniques, it does not build an event log on the fly during system execution [12, 13]. Instead, this technique stores all the information about events and their candidate correlation attributes in a database. A relational database context facilitates the implementation of faster algorithms to discover the correlation set from large datasets (cf. Section 3.2).

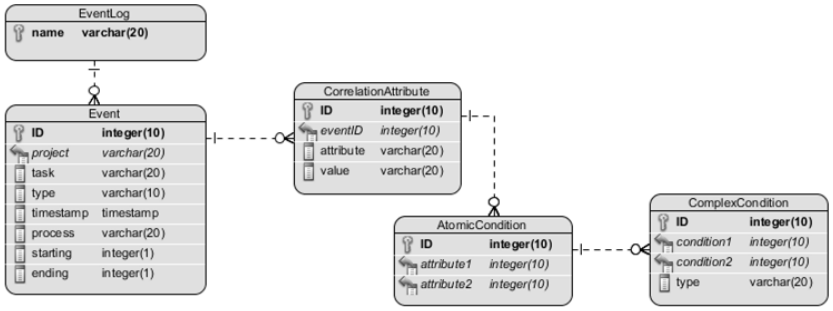


Fig. 3. Database schema for event and attributes

Fig. 3 shows the database schema used to represent the intermediate event information. The EventLogs table is used to represent different logs obtained from different source systems. The Events table contains all the different events collected, including the executed task, type (start or complete), originator, execution timestamp, two columns to indicate if the executed task is the initial or final task of a process, as well as the process name. CorrelationAttributes is a table related to Events and contains the runtime values of candidate correlation attributes.

Candidate correlation attributes are combined by means of correlation conditions which are then used to correlate events. We differentiate two kinds of correlation conditions: atomic and complex conditions. Firstly, atomic conditions represent key-based conditions which compare two correlation attributes. For example, condition1:attribute1 =attribute2 signifies that two events will be correlated if the value of attribute1 of the first event is equal to the value of attribute2 of the second event under evaluation. These conditions are stored in the AtomicConditions table (see Fig. 3). Secondly, complex conditions evaluate two different conditions at the same time that are combined by a logic operator, e.g., conjunction (AND) or disjunction (OR) [8]. For example, condition3: condition1 AND condition2 evaluated for two events signifies that both condition1 and condition2 must be met for the couple of events at the same time. Table Complex-Conditions represents this information in the database schema (see Fig. 3).

3.2 Discovering Correlation Attributes

After collection of events and candidate correlation attributes, an adaptation of the algorithm described in [8] is applied to discover the most appropriate correlation set. This set consists of a set of atomic conditions (i.e., equal comparisons between couples of attributes) as well as a set of complex conditions (i.e., conjunctive comparisons between couples of atomic conditions). The technique does not consider disjunctive conditions since these conditions are only needed for heterogeneous systems to detect synonyms of certain correlation attributes. The algorithm (see Fig. 4) first considers all the possible combinations of attributes involved in atomic conditions (lines 1-3), and it then prunes the non-interesting conditions (i.e., less promising conditions) using the three following heuristics.

Heuristic 1. When attributes of atomic conditions are the same, the distinct ratio must be above alpha or distinct to one (line 4). Distinct Ratio (1) indicates the cardinality of an attribute in the dataset regarding its total number of non-null values. When it is under alpha or one it signifies that the attribute contains a global unique value and the atomic condition with this attribute can therefore be pruned. Alpha (2) is the threshold to detect global unique values and it indicates how much values vary regarding the size of the dataset.

$$\boldsymbol{DistinctRatio}(\boldsymbol{a_i}) = \frac{distinct(a_i)}{nonNull(a_i, a_j)} \tag{1}$$

$$\boldsymbol{\alpha} = \frac{distinct_{MAX}(a_i)}{NumberOfEvents} \tag{2}$$

Heuristic 2. If the atomic condition is formed by two different attributes it is pruned when the shared ratio is above the alpha threshold (line 5). In a similar way the previous heuristic, Shared Ratio (3) represents the number of distinct shared values regarding their non-null values for both attributes.

$$\boldsymbol{SharedRatio}(a_i, a_j) = \frac{distinct(a_i, a_j)}{max\left(distinct(a_i), distinct(a_j)\right)} \quad (3)$$

```
Input: Attributes; Events
Output: AC: the set of atomic conditions; CC: the set of conjunctive conditions
1:  for a ∈Attributes ∧ b ∈Attributes do
2:      AC← "a = b"
3:  end for
4:  AC←AC – { c | c.attribute1 = c.attribute2 and
    ( DistinctRatio (c.attribute1) < α or DistinctRatio (c.attribute1) = 1 ) }
5:  AC←AC – { c | c.attribute1 ≠ c.attribute2 and
    SharedRatio (c.attribute1, c.attribute2) <α }
6:  AC←AC – { c | PIRatio (c.attribute1, c.attribute2) < α or
    PIRatio (c.attribute1, c.attribute2) >β }
7:  N0←AC; N1← { }
8:  k ← 1
9:  for c1 ∈ Nk-1 and c2 ∈ Nk-1 do
10:     Nk← "c1 ∧ c2"
11: end for
12: while Nk≠ { } do
13:     Nk←Nk – { c | ConjNumberPI ( Nk.condition1, Nk.condition2 ) ≤
        NumberPI( Nk.condition1.attribute1, Nk.condition1.attribute2 ) or
        ConjNumberPI( Nk.condition1, Nk.condition2 ) ≤
        NumberPI( Nk.condition2.attribute1, Nk.condition2.attribute2 ) }
14:     Nk←Nk – { c | ConjPIRatio ( Nk.condition1, Nk.condition2 )<α or
        ConjPIRatio( Nk.condition1, Nk.condition2 ) > β }
15:     CC←CC ∪ Nk
16:     for c1 ∈ Nk and c2 ∈ Nk do
17:         Nk+1← "c1 ∧ c2"
18:     end for
19:     k ← k+1
20: end while
```

Fig. 4. Algorithm to discover the correlation set

Heuristic 3. Atomic conditions are pruned when the process instance ratio (PIRatio) is under alpha or above beta (line 6). This heuristic checks that the partitioning of the future log does not only have one or two big instances, or many short instances. PIRatio (4) is measured as the number of process instances (NumberPI) divided into non-null values for both attributes. In turn, NumberPI (5) is heuristically assessed as the distinct attribute values for all the different couples of events (executed in a row) containing both attributes. This is the first difference, since the previous algorithm first calculates a set of correlated event pairs, and then it computes a recursive closure over that set [8]. In contrast, our algorithm estimates NumberPI by considering the number of possible pairs of correlated events. This change has been made taking into account that the recursive closure evaluation is time-consuming (the complexity of graph closure algorithms in literature is often O(2n) since they check each pair of nodes for the remaining of pairs). On the contrary, the expected results using this

proposal can be considered as heuristic approximation with a lower computational cost (i.e., O(n) since this technique only evaluates the list of event pairs). This is the first difference regarding the algorithm proposed by *Motahari-Nezhad et al.* [8].

$$\boldsymbol{PIRatio}(\boldsymbol{a_i}, \boldsymbol{a_j}) = \frac{|NumberPI(a_i, a_j)|}{nonNull(a_i, a_j)} \tag{4}$$

$$\begin{aligned}\boldsymbol{NumberPI}(\boldsymbol{a_i}, \boldsymbol{a_j}) = \{v : \exists\, e, e' \epsilon Events | e.attribute1 = a_i \\ e'.attribute2 = a_j\ v = e.attribute1.value \\ = e'.attribute2.value\ \ e.timestamp > e'.timestamp\ \}\end{aligned} \tag{5}$$

Moreover, Beta (6) which can be established between 0.25 and 1 is used as another threshold to evaluate the average length of the outgoing instances. For instance, a beta value of 0.5 (as is usually used) signifies that conditions leading to process instances with length above or equal to the half of the total events would be discarded.

$$\boldsymbol{\beta} \epsilon\ [0.25, 1] \tag{6}$$

After atomic conditions are filtered out, the algorithm (see Fig. 4) builds all the possible conjunctive conditions based on the combination of outgoing atomic conditions (lines 7-11). These conditions are then pruned by applying two heuristics (lines 13-14). After that, new conjunctive conditions by combining the remaining previous conditions are iteratively evaluated (lines 15-19).

The first heuristic (line 13) applied to filter out conjunctive conditions is based on the monotonicity of the number of process instances. This is the second difference with regard to [8], since this algorithm considers the number of process instances (but not the length of instances) to evaluate the monotonicity heuristic. This heuristic is based on the idea that the number of process instances for a conjunctive condition is always higher than the number for their simpler conditions in isolation. Conjunctive conditions that do not increase the number of process instances are therefore pruned, since they are subsumed in their atomic conditions. The number of process instances obtained though conjunctive conditions (ConjNumberPI) (7) is based on (5), which is defined for simple conditions, and is measured by intersecting both component conditions of the conjunctive one.

$$\boldsymbol{ConjNumberPI}(\boldsymbol{c_i}, \boldsymbol{c_j}) = NumberPI(c_i.a_1, c_i.a_2) \cap NumberPI(c_j.a_1, c_j.a_2) \tag{7}$$

The algorithm also applies the same heuristic about the partitioning of the log to the conjunctive conditions (line 14). Thereby, the ratio of process instances is also evaluated for conjunctive conditions (*ConjPIRatio*) (8). When *ConjPIRatio* is under alpha or above beta threshold the conjunctive condition is discarded.

$$\boldsymbol{ConjPIRatio}(\boldsymbol{c_i}, \boldsymbol{c_j}) = \frac{|ConjNumberPI(c_i, c_j)|}{nonNull(c_i.a_1,\ c_i.a_2,\ c_j.a_1,\ c_j.a_2)} \tag{8}$$

In conclusion, the proposed algorithm adapts the algorithms provided by [8] thus adjusting to traditional systems. There are two main changes as seen above in this section: (i) the way in which the expected number of process instances is calculated for each condition; and (ii) the monotonicity heuristic which only takes into account the length of the estimated process instances.

```
Input: Events; AC: the set of atomic conditions; CC: the set of conjunctive conditions
Output: Log: the final event log
1:process; instance
2:for e1 ∈ Events ∧e2 ∈ Events ∧ e1.proces = e2.process
  ∧ e1.timestamp ≤ e2.timestamp do
3:   if e1.starting=true ∧ ∀n ∈Log.processes.name, process.name=n then
4:     process.name ← e1.process
5:   end if
6:   for ac ∈AC ∧cc ∈CCdo
7:     if ∃i, e1.attributes[ i ].name = ac.attribute1 ∧
       ∃i', e2.attributes[ i' ].name = ac.attribute2 ∧
       e1.attributes[ i ].value = e2.attributes[ i' ].value ∧
       ∃j, e1.attributes[ j ].name = cc.condition1.attribute1 ∧
       ∃j', e2.attributes[ j' ].name = cc.condition1.attribute2 ∧
       e1.attributes[ j ].value = e2.attributes[ j' ].value ∧
       ∃k, e1.attributes[ k ].name = cc.condition2.attribute1 ∧
       ∃k', e2.attributes[ k' ].name = cc.condition2.attribute2 ∧
       e1.attributes[ k ].value = e2.attributes[ k' ].value then
8:         instance.id ←e1.attributes[ i ].value + e2.attributes[ i' ].value +
           e1.attributes[ j ].value + e2.attributes[ j' ].value +
           e1.attributes[ k ].value + e2.attributes[ k' ].value
9:       instance.events←instance.events∪ {e1, e2}
10:        process.instances←process.instances∪ {instance}
11:      end if
12:  end for
13:  Log.processes←Log.processes∪ [4]
14: end for
```

Fig. 5. Algorithm to discover process instances

3.3 Discovering Process Instances

The last stage, after obtaining the correlation set, attempts to discover process instances using the correlation set in order to build the final event log, which will be written following the MXML (Mining XML) format [16]. MXML is a notation based on XML and is the most common format used by most process mining tools.

Fig. 5 shows the algorithm to correlate all the events of the intermediate dataset in its own process instance within the event log. The algorithm explores all the candidate events pairs, i.e., those pairs that belong to the same process and which were executed in a row (line 2). When an event was recorded as the start point of a process, the target process takes this name (lines 3-5). For each candidate event pair, all the atomic and conjunctive conditions of the correlation set are evaluated (line 7). If the event pair meets all the conditions, then it is a correlated event pair and these events are then put into the respective process instance, and in turn, the instance is added to the process (lines 9-10). Process instances are previously identified by means of the specific values of the events' attributes involved in the correlation set (line 8). Each process found during the event pair exploration, together with all the discovered process instances, is finally added to the event log (line 13).

Business process can be subsequently discovered from the MXML event logs by applying different well-known techniques and algorithms developed from the business process mining field.

4 Case Study

This section presents a case study conducted with a real-life information system. The whole set of artifacts involved in the study are online available in [10]. The object of the study is the proposed technique and the purpose of the study is to demonstrate the feasibility of the technique in terms of accuracy. The main research question therefore is established as MQ. Additionally, the study evaluates two secondary research questions: AQ1 and AQ2.

MQ. - *Can the technique obtain correlation sets for generating event logs from a traditional system which could on their turn be used to discover the business processes supported by the system?*

AQ1. - *How well does this technique perform compared to the previously developed technique?*

AQ2. - *How much time does the technique require to discover correlation sets regarding the size of datasets?*

AQ1 evaluates the gain of this new approach over the previous one employing an isolated source code object as correlation set. For the evaluation of this secondary question, the result of this study is compared with the result obtained in a previous study that validated the previous approach using the same traditional information system [13]. AQ2 assesses the time spent on discovering correlations sets in order to know if the technique is scalable to a large dataset.

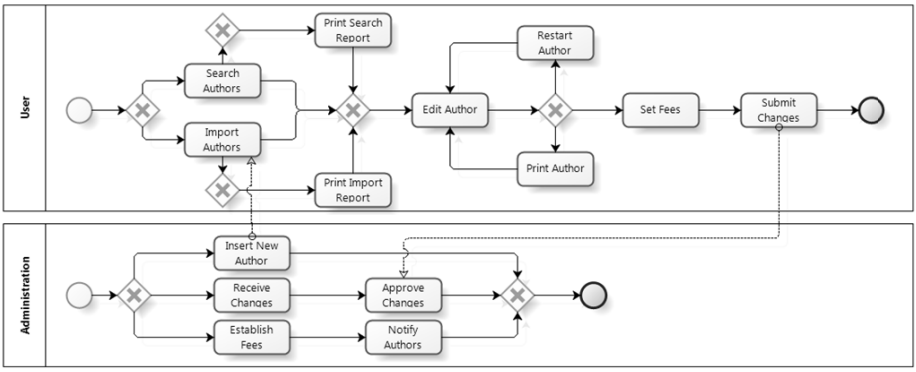


Fig. 6. Reference business process model of the AELG-members system

In order to answer the main research question, the study follows a qualitative research approach by comparing a reference model and the obtained one. The study first considers the business process model previously provided by business experts (the reference model). Secondly, the study obtains an event log (using the discovered

correlation set) and compares the business processes instances collected in the log together with the reference business process model. The comparison between models evaluates the degree of conformance of the obtained model regarding the reference one, which is done by scoring the number of business activities in common with the reference business process model.

4.1 Case under Study

The traditional (non-process-aware) information system under study was AELG-members, which supports the administration of an organization of Spanish writers. From a technological point of view, AELG-members is a Java standalone application with an architecture that follows the traditional structure on three layers: (i) the domain layer supporting all the business entities and controllers; (ii) the presentation layer dealing with the user interfaces; and (iii) the persistency layer handling data access. The total size of the legacy system is 23.5 KLOC (thousands of lines of source code).

Fig. 6 shows the business process supported by the system under study, which is considered as the reference business process model. The main business activities carried out by the writers' organization, including author registration, importing author information from different sources, cancellation of memberships, author information management and payment of fees.

4.2 Execution Procedure

For the execution of the case study, all the stages of the proposed technique were semi-automated by different tools. The steps carried out during the study execution were the following:

1. The AELG-members system was instrumented through the Event Traces Injector tool [10] which was modified to support the addition of candidate correlation attributes by experts. Six attributes were selected to be collected together with events (see Table 1). Some attributes regarding the identification of author were first selected due to the business processes focuses on this entity. Other attributes related to fees were also selected since experts expect process instances end when annual fees of an author are paid.

2. The instrumented version of AELG-members was normally executed in the same production environment. The execution consisted of storing events and candidate correlation attributes in a SQL Server 2005 database until significant datasets to conduct the study were collected. Three different sizes of dataset (above 2000, 7000 and 15000 events) were considered to test different configurations.

3. The algorithm for discovering the correlation set (Fig. 4) was then applied to the datasets. Unlike in previous stages, this algorithm was implemented by means of a set of stored procedures using PL/SQL which executes a set of queries from datasets. Since the beta threshold (Eq. 4) can be chosen by business experts [8], the algorithm was applied with four different values: 0.25, 0.5, 0.75 and 1. The four different correlation sets obtained for each configuration are shown in Table 2. An event log was

obtained for each different correlation set by means of the algorithm presented in Fig. 5, which was also implemented through PL/SQL procedures.

4. The four event logs were finally analyzed and compared with the reference model. For this purpose, the ProM tool [16] was used to discover the respective business process models for each log. The study particularly used the genetic mining algorithm of ProM since it is the most accurate one [7]. Finally, the conformance of each business process model with the reference model was analyzed according to the aforementioned qualitative research approach (cf. Section 4.1).

Table 1. Candidate correlation attributes selected

Attribute ID	Java Class	Output Method
1	FeeVO	getIdAuthor
2	AuthorVO	getId
3	AuthorVO	isHistoric
4	AuthorVO	getMemberNumber
5	PublicAuthorVO	getId
6	AuthorVO	getFees

Table 2. Correlation sets and time obtained in each case

	Events	Correlation Attributes	β=0.25	β=0.5	β=0.75	β=1
Correlation Sets	2432	10412	A	C	C	C
	7608	33278	A	C	C	C
	15305	74136	B	C	D	D
Time (s)	2432	10412	12	15	16	15
	7608	33278	41	56	55	55
	15305	74136	113	150	151	147

Table 3. Correlation sets. Numbers 1 to 5 refer to attribute IDs of Table 1. Letters *o* to *s* refer to atomic conditions.

Atomic Conditions	A	o : 1=1	p : 2=2	q : 4=4	r : 6=6	
	B	o : 1=1	p : 2=2	q : 4=4	r : 6=6	
	C	o : 1=1	p : 2=2	q : 4=4	r : 6=6	s : 5=5
	D	o : 1=1	p : 2=2	q : 4=4	r : 6=6	s : 5=5
Complex Conditions	A	o ∧ q	p ∧ q			
	B	o ∧ q	p ∧ q	r ∧ q	r ∧ p	
	C	o ∧ q	p ∧ q	r ∧ q	r ∧ p	
	D	o ∧ q	p ∧ q	r ∧ q	r ∧ p	o ∧ s

4.3 Analysis of Results

Table 2 and Table 3 summarize the results obtained after cases study execution, showing the correlation sets (A, B, C and D) obtained for each combination of dataset (2432, 7608 and 15305 events) and beta value (0.25, 0.5, 0.75 and 1). Table 2 also shows the time spent on discovering each correlation set as well as the particular atomic and conjunctive conditions of each set.

After obtaining the corresponding event log and discovering the respective business process for the AELG-Members system, it was perceived that the most accurate correlation set was ‘A’. The set ‘A’ leads to the business process with the highest

conformance degree (93%). This means the business process discovered using set 'A' had the highest number of business activities in common with the reference business process model (13 from the 14 tasks).

The same conclusion can be stated by analyzing the conditions of correlation set 'A' (see Table 3). Set 'A' is less restrictive (compared to the other sets) and contains fewer correlation conditions. Despite this fact, it contains all the atomic conditions necessary to evaluate the identity of each writer (i.e., getIdAuthor, getId and get MemberNumber). Additionally, set 'A' also contains the atomic condition to know when a fee is paid (i.e., getFees), which signifies that a particular process instance ends for a writer.

Moreover, regarding complex conditions of correlation set 'A', there is a conjunctive condition linking FeeVo.getIdAuthor together with AuthorVO.getId, which signifies that the managed fees must correspond to the same writer of a particular process instance. Finally, set 'A' also works well because the categorical correlation attribute AuthorVO.isHistoric was properly discarded, since these kinds of attributes (e.g., Boolean variables) split the datasets into only two instances.

The remaining correlation sets (B, C and D) are similar to correlation set 'A', since all those sets contain all correlation conditions of 'A'. However, those sets incorporate more conditions, and although they provide alternative event correlations, they are more restrictive. This means that some process instances obtained using 'A' could be split in two or more instances in case sets B, C or D were used as the correlation set instead of set 'A'. These sets led to conformance values between 64% and 86%, which respectively correspond to 9 and 12 tasks in common with the 14 tasks of the reference model.

Regarding the evaluation of AQ1, in the previous case study with the same system [13], the Java class 'AuthorVO' was selected as the classifier to collect correlation information during the system instrumentation stage. During system execution, the runtime values of the AuthorVO objects were used to correlate events. As a result, all the process instances in the event log were obtained with all the events regarding each writer. Unlike the current approach, not all the different executions of the reference business process (see Fig. 6) for each author were detected. For example, every time a writer pays the annual fee, it should be detected as the end of a process instance. This kind of aggregation works by using any correlation set obtained with the current approach. However, as per the previous approach, not all the events of the same writer could be grouped into fine-grained process instances, since the sole information to correlate events was AuthorVO objects.

The conformance degree in the previous case study with the same system was 77% [13], while the degree obtained with the proposed technique is 93% (obtained with set 'A'). In fact, the business process obtained with the previous technique was complex and visually intricate due to several crossing sequence flows. As a result, AQ1 can be positively answered.

In order to demonstrate the feasibility of the proposal, the time spent on discovering correlation sets was analyzed according to the additional question AQ2. Outlier beta values such as 1, and especially 0.25 lead to shorter times (see Table 2). This is due to the fact that outlier values allow the algorithm to quickly prune non-promising correlation sets, saving much time. Anyway, it should be noted that the time regarding the beta value is approximately linear. Besides, regarding the number of events, the

time is non-linear. The time is lower for smaller datasets and higher for larger ones. It seems the trend of the time follows a quadratic function. This is due to the fact that every event must be checked for all the remaining events according to the proposed algorithm.

In conclusion, the main research question can be positively answered. This means that the technique is able to correlate events from traditional system, and in turn, it produces a gain regarding techniques previously developed. However, the time spent on discovering correlation sets is quadratic, and huge datasets may be time-consuming.

4.4 Threats to the Validity

The most important threat is the fact that the code could be poorly instrumented. The obtained results clearly depend on the candidate correlation attributes selected at the beginning of the study. If business experts select an incomplete or erroneous set of candidate correlation attributes, the outgoing results could be quite different. In order to mitigate this threat we propose repeating the study using an iterative approach in which experts can select or remove some candidate correlation attributes according to the results obtained for each iteration. This way, the list of candidate correlation attributes can be iteratively refined.

Moreover, correlation sets do not always have to be obtained under lower beta values (e.g., 0.25). A lower beta value often implies a more restrictive correlation set and vice versa. The beta threshold can therefore be established by business experts depending on the constraint degree to be applied to the particular set of candidate correlation attributes. This threat can be addressed by repeating the study with different cases and different beta values.

5 Conclusions

This paper presents a technique to discover the correlation set in order to generate event logs from traditional (non-process aware) information systems. This challenge is important in traditional systems since (i) they do not have any in-built mechanism to record events and (ii) captured events do not have any reference to the process instance they belong to.

The technique consist of three stages: (i) the selection of candidate correlation attributes and injection of statements into the source code to collect events during system execution; (ii) the discovery of the correlation set from collected events; and (iii) the generation of the final event logs by correlating events using the discovered correlation conditions.

All the stages of the technique are semi-automated, making it possible to validate the proposal by conducting a case study with a real-life system. The study demonstrates the feasibility of the technique to discover correlation sets that lead to well-formed event logs and the gain regarding previous techniques in terms of accuracy. The main implication of the results is that this technique contributes to the application of well-proven techniques and algorithms from the process mining field. So far, such

business process mining techniques work with event logs that are often obtained only from process-aware information systems.

The work-in-progress focuses on conducting another case study with a healthcare information system to obtain strengthened conclusions about empirical validation. Moreover, concerning the selection of candidate correlation attributes, a mechanism to analyse source code and provide business experts with some insights about the most appropriate attributes will be developed.

Acknowledgements. This work has been supported by the *FPU Spanish Program*; by the R+D projects funded by *JCCM*: ALTAMIRA (PII2I09-0106-2463), GEODAS-BC (TIN2012-37493-C03-01), and MAESTRO.

References

1. Burattin, A., Vigo, R.: A framework for Semi-Automated Process Instance Discovery from Decorative Attributes. In: IEEE Symposium on Computational Intelligence and Data Mining (CIDM 2011), Paris, France, pp. 176–183 (2011)
2. Castellanos, M., Medeiros, K.A.D., Mendling, J., Weber, B., Weitjers, A.J.M.M.: Business Process Intelligence. In: Cardoso, J.J., van der Aalst, W.M.P. (eds.) Handbook of Research on Business Process Modeling, pp. 456–480. Idea Group Inc. (2009)
3. Ferreira, D.R., Gillblad, D.: Discovering process models from unlabelled event logs. In: Dayal, U., Eder, J., Koehler, J., Reijers, H.A. (eds.) BPM 2009. LNCS, vol. 5701, pp. 143–158. Springer, Heidelberg (2009)
4. Fluxicon Process Laboratories, XES 1.0 Standard Definitio (Extensible Event Stream) (2009), `http://www.xes-standard.org`
5. Hammoud, N.: Decentralized Log Event Correlation Architecture. In: Proceedings of the International Conference on Management of Emergent Digital EcoSystems, pp. 480–482. ACM, France (2009)
6. McGarry, K.: A Survey of Interestingness Measures for Knowledge Discovery. Knowl. Eng. Rev. 20(1), 39–61 (2005)
7. Medeiros, A.K., Weijters, A.J., Aalst, W.M.: Genetic Process Mining: An Experimental Evaluation. Data Min. Knowl. Discov. 14(2), 245–304 (2007)
8. Motahari-Nezhad, H.R., Saint-Paul, R., Casati, F., Benatallah, B.: Event Correlation for Process Discovery From Web Service Interaction Logs. The VLDB Journal 20(3), 417–444 (2011)
9. Myers, J., Grimaila, M.R., Mills, R.F.: Adding Value to Log Event Correlation Using Distributed Techniques. In: Proceedings of the Sixth Annual Workshop on Cyber Security and Information Intelligence Research, pp. 1–4. ACM, Oak Ridge (2010)
10. Pérez-Castillo, R.: Experiment Results about Assessing Event Correlation in Non-Process-Aware Information Systems (2012), `http://alarcos.esi.uclm.es/per/rpdelcastillo/CorrelationExp.html#correlation` (cited February 09, 2012)
11. Pérez-Castillo, R., Weber, B., García-Rodríguez de Guzmán, I., Piattini, M.: Toward obtaining event logs from legacy code. In: Muehlen, M.z., Su, J. (eds.) BPM 2010 Workshops. LNBIP, vol. 66, pp. 201–207. Springer, Heidelberg (2011)
12. Pérez-Castillo, R., Weber, B., García Rodríguez de Guzmán, I., Piattini, M.: Generating Event Logs from Non-Process-Aware Systems Enabling Business Process Mining. Enterprise Information System Journal 5(3), 301–335 (2011)

13. Pérez-Castillo, R., Weber, B., García Rodríguez de Guzmán, I., Piattini, M.: Process Mining through Dynamic Analysis for Modernizing Legacy Systems. IET Software Journal 5(3), 304–319 (2011)
14. Rozsnyai, S., Slominski, A., Lakshmanan, G.T.: Discovering Event Correlation Rules for Semi-Structured Business Processes. In: Proceedings of the 5th ACM International Conference on Distributed Event-based System, pp. 75–86. ACM, New York (2011)
15. van der Aalst, W., Weijters, A.J.M.M.: Process Mining, in Process-aware information systems: bridging people and software through process technology. In: Dumas, M., van der Aalst, W., Ter Hofstede, A. (eds.), pp. 235–255. John Wiley & Sons, Inc. (2005)
16. Van der Aalst, W.M.P., Van Dongenm, B.F., Günther, C., Rozinat, A., Verbeek, H.M.W., Weijters, A.J.M.M.: ProM: The Process Mining Toolkit. In: 7th International Conference on Business Process Management BPM, - Demonstration Track, pp. 1-4. Springer, Ulm, Germany(2009)
17. Weske, M.: Business Process Management: Concepts, Languages, Architectures, Leipzig, Germany, p. 368. Springer, Heidelberg (2007)
18. Zou, Y., Hung, M.: An Approach for Extracting Workflows from E-Commerce Applications. In: Proceedings of the Fourteenth International Conference on Program Comprehension, pp. 127–136. IEEE Computer Society (2006)

Detection of Infeasible Paths: Approaches and Challenges

Sun Ding and Hee Beng Kuan Tan

Division of Information Engineering, Block S2, School of Electrical & Electronic Engineering,
Nanyang Technological University, 639798, Singapore
{ding0037,ibktan}@ntu.edu.sg

Abstract. Each execution of the program follows one path through its control flow graph. In general, a program has a large number of such paths. Most of the programs have an infinite number of these paths. Regardless of the quality of the program and the programming language used to develop it, in general, a sizable number of these paths are infeasible—that is no input can exercise them. Detection of these infeasible paths has a key impact in many software engineering activities including code optimization, testing and even software security. This paper reviews the approaches for detecting infeasible paths, discusses the challenges and proposes to revisit this important problem by considering also empirical aspect in conjunction to formal program analysis.

Keywords: Survey, Path Infeasibility, Symbolic Evaluation, Program Analysis, Software Testing.

1 Introduction

Control flow graph (CFG) is the standard model to represent the execution flow between statements in a program. In the CFG of a program, each statement is represented by a node and each execution flow from one node to another is represented by a directed edge, where this edge is out-edge of the former node and the in-edge of the latter node. Each path through the CFG from the entry node to the exit node is a logic path in the program. In order for an execution to follow a path in the CFG, the input submitted to the program must satisfy the constraint imposed by all the branches that the path follows. An infeasible path is a path in the CFG of a program that cannot be exercised by any input values. Figure 1 shows an infeasible path p = (entry, 1, 2, 3, 4, 5, 6, exit) in a CFG. This is because we cannot find any input x satisfying $x \geq 0$ and $x < 0$ jointly.

The existence of infeasible paths has major impact to many software engineering activities. Code can certainly be optimized further if more infeasible paths can be detected during the process of optimization. In software testing, the structural test coverage can be much accurately computed if infeasible paths can be detected more accurately. In the automated generation of structured test cases, much time can be saved if more infeasible paths can be detected. In code protection, it can also help in code deobfuscation to identify spurious paths inserted during obfuscation. In software verification, detecting and eliminating infeasible paths will help to enhance the verification precision and speed. There are many more areas like security analysis [37],

L.A. Maciaszek and J. Filipe (Eds.): ENASE 2012, CCIS 410, pp. 64–78, 2013.

web application verification [30]; [29]; [28]; [27], database application design [34]; [35] that can be helped by the detection of infeasible paths.

To detect infeasible paths in real programs, one needs to deal with complex data structures and dependency. Additional effort is required to formally present them in symbolic expressions or constraints for further verification by heuristics, predefined rules or even standard theorem provers. If the verification returns negative results (e.g.: "Invalid" answer from theorem provers), the path is then considered as infeasible. Such verification model is undecidable in general. But it is still possible to have practical approaches that are not theoretically complete to detect infeasible paths.

The purpose of this article is to familiarize the reader with the recent advances in infeasible paths detections and its related applications. Concepts and approaches will be introduced informally, with citations to original papers for those readers who preferring more details. Information about tools and implementation is also introduced. The paper is organized as below: the literals for infeasible paths detection is reviewed in section 2. Information of tool implementation is introduced in section 3. We discussed remaining problems and future challenges in section 4. Section 5 summarizes the entire paper.

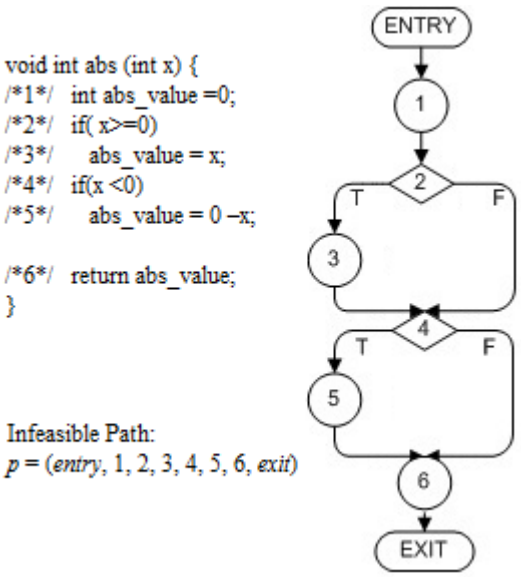


Fig. 1. An infeasible path

2 Detection of Infeasible Path

During software developing or testing, infeasible path detection usually appears as an essential step for the final project goal. A variety of approaches have been proposed. Based on the ways that they detect infeasible paths, we classify them into six types: (1) data flow analysis; (2) path-based constraint propagation; (3) property sensitive data flow

analysis; (4) syntax-based approach; (5) infeasibility estimation; (6) gene-ralization of infeasible paths. These approaches differ in their detection precision, computational cost and relevantly suitable applications. We review these approaches by introducing their main features, strength and weaknesses and the related applications.

2.1 Data Flow Analysis

Classic Data flow analysis is a technique over CFG to calculate and gather a list of mappings, which maps program variables to values at required locations in CFG. Such list of mappings is called flow fact. A node with multiple in-edges is defined as a merge location, where flow facts from all of its predecessor nodes are joined together. Due to the joining operation, a variable may be mapped to a set of values instead of a single value. If a node has multiple out-edges (predicate node), each of these out-edges is defined as a control location. Flow fact is split and traversed to the successor nodes at control locations. Due to the splitting operation, a variable may be mapped to an empty set of values [25].

Data flow analysis is a common approach for detecting infeasible paths. In this type of approach, each control location would be checked when they are traversed. An infeasible path is detected when any variable is mapped to an empty set of values at a control location. In Figure 2, suppose we only consider the flow fact about variable *sum*. Here *sum* is an integer variable initialized as 0. Therefore the flow fact is initialized as [0, 0] after node1. The flow fact is traversed transparently through node2, node3 and reaching node5, after which it is split as two: one as [0, 0] flowing to node6 and the other as an empty set flowing to node7. It is then concluded that any path passing through (1, 2, 3, 5, 7) is infeasible. Approaches based on data flow analysis are often useful for finding a wide variety of infeasible paths. In the above example, the checking at node5 can detect a family of infeasible paths, which all containing the sub part (1, 2, 3, 5, 7).

However, classic data flow analysis scarifies the detection precision, which causes some infeasible paths wrongly identified as feasible. It is important to note that the flow fact computed at a control location L is essentially an invariant property — a property that holds for every visit to L. Therefore two things will cause the loss of the detection precision: First, the correlated branches are ignored and flow facts are propagated across infeasible paths. Second, by the joining operation at merging location, flow facts from different paths are joined, leading to further over-approximation [16]. To explain this point, consider the example program shown in Figure 2. The flow fact of variable i is initialized as [0, 0]. After passing node3, it is split as two: one as [0, 0] on the TRUE branch and the other as an empty set on the FALSE branch. However by simply keeping track of all possible variable values at node5, the two different flow facts are joined. The flow fact from node5 flowing to node6 is over-approximated as [0, 0]. Hence we cannot directly infer that node4 cannot be executed in consecutive iterations of the loop. Therefore path such as (entry, 1, 2, 3, 4, 5, 6, 2, 3, 4) cannot be inferred as infeasible, which actually is. The most typical and well cited approach for detecting infeasible paths based on data flow analysis is from Gustafsson et al., [21]; [19]; [20]. Approaches based on data flow analysis are path insensitive.

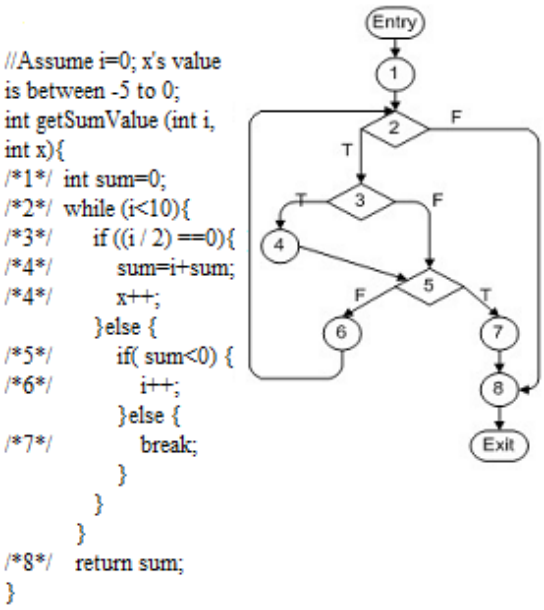


Fig. 2. Infeasible path detection with data flow analysis

Other similar approaches include work from Altenbernd [1], detecting infeasible paths by searching for predicates with conflict value range while traversing CFG in an up-bottom style. This approach depends on knowing execution time of each basic block in advance. The basic block refers to a piece of continuing nodes except predicate nodes. At each merge location, only the in-edge with longest execution time will be remained for further consideration. All flow facts will be checked at each control location. Those branches that with variables mapped to an empty value set will be detected as infeasible and excluded for further consideration. Dwyer et al. [14] proposed to adopt data flow analysis approach to check consistency violation in concurrent systems. They construct a Trace-Flow Graph (TFG), which is a modified version of CFG for concurrent systems. Variables in TFG are mapping to a set of possible system properties like sequence of event calling, synchronization of critical section. A consistency violation is found when a corresponding path is identified as infeasible in the TFG.

Approaches based on data flow analysis do not require providing a prepared set of paths. It searches for infeasible paths directly based on CFG. So they are often applied to estimate the maximum execution time for a procedure, called WCET: worst-case execution time [15] which is essential in designing real time systems. Firstly they help tighten the estimated result of WCET analysis by removing the influences from infeasible paths in the case that these paths are the longest ones. Secondly, they are useful in deriving loop upper bound in WCET analysis.

2.2 Path-Based Constraint Propagation

Path-based propagation approaches apply symbolic evaluation to a path to determine its feasibility. These approaches carry a path sensitive analysis for each individual

path in a given path set by comparing with approaches based on data flow analysis. Through symbolic evaluation, they propagate the constraint that a path must satisfy and apply theorem prover to determine the solvability of the constraint. If the constraint is unsolvable, the path is then concluded as infeasible. These approaches have high precision of detection but with heavy overhead. They are usually applied in code optimization and test case generation in which accuracy is essential. Figure 3 gives a general overview of these approaches.

In propagating constraint along a given path, either backward propagation [2]) or forward propagation strategy [3] could be adopted to extract the path constraint under the supported data types. Forward propagation traverses the path from entry to exit and performs symbolic execution on every executable statement. Intermediate symbolic values are stored for subsequent use. It can detect infeasible paths early by detecting contradicting constraints early. It is also more straight forward and thus easier to implement, especially in the case of dealing with arrays or containers like List and HashMap. [41]. However the storage of intermediate values may grow very fast and cause this strategy not scalable for large program. Backward strategy applies a bottom-up traversing manner by collect path constraint first and later only search for values correlated with path constraint. The space of the intermediate storage is largely reduced.

Based on the underlying domain theories of the constraint solver or theorem pro-er [39], constraint solving determines the solvability of the propagated constraint. The power and precision of path-based constraint propagation approaches depend on the power of constraint solver. Hence, they are sound except on those cases in which existing constraint solvers have problems (e.g., floating point problems).

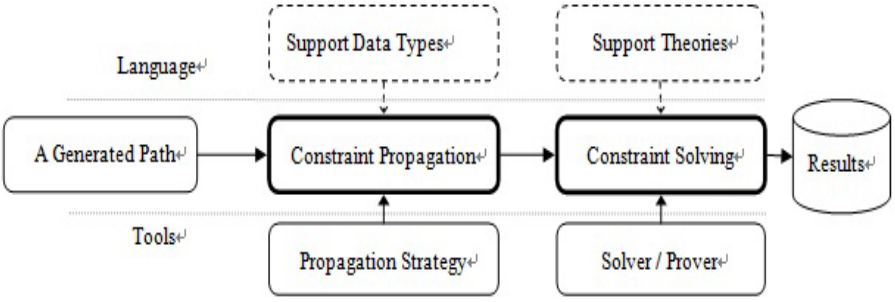


Fig. 3. An overview of path-based constraint propagation approach

As mentioned by Williams [43] recently, though constraint resolution is very efficient most of the time, it is actually NP-complete and it is undecidable to know which kind of constraint will take "too long" to be solved. Therefore it is not always possible for these approaches to determine the feasibility of a path automatically. Furthermore every time execution the constraint solver, there is a risk of causing the timeout exception.

Figure 4 illustrates an example of path-based constraint propagation approaches in general. Consider the target path p = (entry, 1, 2, 3, 6, 7, exits). After propagation along the path using backward propagation strategy, the constraint finally becomes $((x<2)$ AND $(x+1)>7)$. The resulting constraint is submitted to a constraint solver for evaluation. As the result is unsolvable, therefore, this path is identified as infeasible.

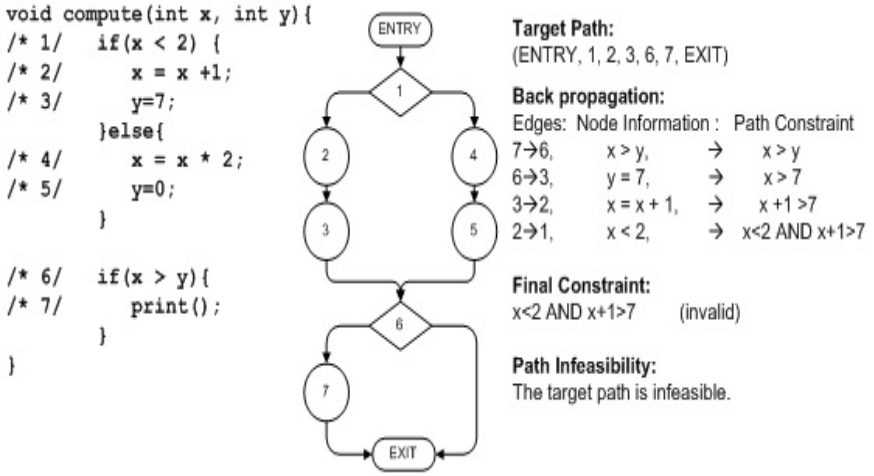


Fig. 4. An illustration of the path-based constraint propagation approach

Among the path-based constraint propagation approaches, Bodik et al., [5] observed that many infeasible paths were caused by branch correlation and data flow analysis based approaches are overly conservative to handle them. Starting from a predicate node, they address this problem by backward propagation along a path to accumulate path constraint and consecutively evaluating the constraint with predefined rules. The path traversed is identified as infeasible if the associated constraint is determined by the predefined rules as unsolvable. If the constraint is determined by the predefined rules as solvable, the path is identified as feasible. If the solvability of the constraint cannot be determined by the predefined rules, the feasibility of the path is therefore undetermined. Goldbeg et al., [18] approached the problem with proposed a more general model. For each targeting path, its infeasibility is determined by the corresponding path constraint. The path constraint is the conjunction of each branch condition along the path after substituting every referenced variable's definition. An independent constraint solver named KITP is invoked to evaluate the path constraint. If the path constraint is evaluated as unsolvable, then the path is identified as infeasible. Goldbeg et al. explicitly specified a domain, on which that constraint solver KITP could work. The domain limits the considered data types as Integer and Real and limits the constraint type as linear.

By equipping different constraint solvers, other approaches are able to detect infeasible paths with constraints over other domains. Ball and Rajamani [4]used a binary decision diagram (BDD) as the prover to identify infeasible paths for Boolean programs. Zhang and Wang [46] used an ad-hoc SMT solver to detect infeasible paths with path constraints over both Boolean and numerical linear domains. Bjørner et al., [6] detected infeasible paths over the domain of String by using a SMT solver called Z3 (2008) which is able to manipulate and process those string functions in path constraints.

The advantage of the path-based constraint propagation approaches is the precision with which we can detect infeasible program paths. The difficulty in using full-fledged path-sensitive approaches to detect infeasible paths is the huge number of program paths

to consider. In summary, even though path-based constraint propagation approaches are more accurate for infeasible path detection, they suffer from a huge complexity.

Constraint propagation approaches are often used in areas requiring high accuracy like test case generation and code optimization. In test case generation, paths will be firstly evaluated their infeasibility. Infeasible paths will be filtered away from test data generation to save resources and time. For example, Korel [26] checked the path infeasibility before generating test case at the last predicate to avoid unnecessary computation. Other similar examples are like work from Botella et al., [7], and work from Prather and Myers [38]. In code optimization, def-use pairs along those infeasible paths are eliminated for enhancing the efficiency of code [5].

2.3 Property Sensitive Data Flow Analysis Approach

Both data flow analysis and constraint propagation approach have strength and weakness. This section introduces the hybrid approach that combine both of them together under the framework of partial verification. The latter refers to the verification against a list of given properties to check instead of verifying all system properties. Property is an abstract term covering variables, functions or special data structures like pointer in C/C++.

With a given list of properties, approaches of this type have similar routine with approaches using classic data flow analysis except two modifications. First the flow fact is updated at location *L* only when *L* contains properties correlated operations. Second at merge locations, equal values for the same property from different flow facts will be merged as one; but different values for the same property from different flow facts will be separately recorded instead of joining them together. Same with approaches using classic data flow analysis, infeasible paths would be detected if any property is mapped to an empty value set at a control location. To illustrate this, let us go back to the example in Figure 2. Suppose variable *sum* and *i* are specified as the two properties. At the TRUE branch of node5, sum is mapped to an empty set. A family of infeasible paths containing (1, 2, 3, 5, 7) are detected as efficient as using classic data flow analysis. However at node5, the flow facts for *i* will be recorded separately: fl_node3 and fl_node4. Therefore, this time, we are able to detect infeasible paths such as (entry, 1, 2, 3, 4, 5, 6, 2, 3, 4) because only fl_node4 will be considered in the consecutive iterations of the loop.

Among this type of approaches, one well cited work is from Das et al., [11]. They proposed an approach called ESP, whose main idea is introduced in the last paragraph, to enhance the precision of data flow analysis while also guaranteeing the verification could be controlled in polynomial time. They later extended ESP to a more abstract and general model. Work from Dor et al., [13] extended ESP for better performance over C/C++, especially for better cooperating with pointer and heap. Other similar work may include work from Fischer et al., [16] and Cobleigh et al., [8].

The advantage here is that this type of approaches achieves a good balance between precision and scalability. However, it brings difficulty in specifying properties accurately. There is also a risk of detection failure because of losing a precise tracking of some properties, which having complex data structures.

2.4 Syntax-Based Approach

Many infeasible paths are caused by the conflicts that can be identified from using solely syntax analysis. Syntax-based approaches take advantage from these characteristics. They define syntax for such conflicts as patterns or rules. Syntax analysis is applied to detect infeasible paths through using rules or recognizing patterns.

The more noticeable recent approach is proposed by Ngo and Tan [36]; [35]; [34]. They identified the four syntactic patterns, identical/complement-decision pattern, mutually-exclusive-decision pattern, check-then-do pattern, looping-by-flag pattern. These patterns model the obvious conflicts between the value of a variable set and the value of the same variable asserted by the predicate of a branch or between predicates of branches, in a path. For example, the predicates at branches (2, 3) and (4, 5) in Figure 1 are $x \geq 0$ and $x < 0$ respectively. These two predicates have obvious conflict and can be detected from syntax analysis. Hence, the path p = (entry, 1, 2, 3, 4, 5, 6, exit) in Figure 1 is clearly infeasible. Based on the four patterns identified, they developed an approach to detect infeasible paths from any given set of paths. Through the use of these patterns, the approach can avoid the expensive symbolic evaluation by just using solely syntax analysis to detect infeasible paths through recognizing these patterns. In opposing to other approaches, their approaches were proposed as an independent approach. They have also conducted an experiment on code from open-source systems and found their approach can detect a large proportion of infeasible paths.

Among those earlier syntax-based approaches, one well cited work is from Hedley and Hennell [24]. They proposed to use heuristics rules to detect four types of infeasible paths: infeasible paths caused by consecutive conditions, infeasible paths caused by inconsistency between test and definition, infeasible paths caused by constant loop control variable and infeasible paths caused by constant loop times. These rules are quite efficient as they solely based on syntax analysis. Later experiment from Vergilio et al., [42] showed that by a fast scan, Hedley and Hennell's rules were able to correctly identify nearly half paths as infeasible in a path set with 880 sample paths.

The advantage of syntax-based approaches is that they can avoid the expensive symbolic evaluation by applying solely syntax analysis to achieve some efficiency. However, these approaches may report a small number of infeasible paths that are actually feasible as they just based on syntax analysis alone. That is, they suffer from the possibility of reporting false-positive results.

Syntax-based approaches detect infeasible paths from a set of paths, so they rely on well-constructed paths set. They are often used for a fast scan during the early testing stage. They are also used as the first step for code optimization or coverage estimation to avoid the influence of infeasible paths during the later analysis.

2.5 Infeasibility Estimation Approach

Early researchers have found the problems caused by infeasible paths and it was very hard to achieve a satisfied detecting result. Therefore they managed to build statistical metrics to estimate the number of infeasible paths in a given procedure based on certain static code attributes. The most famous work is from Malevris et al., [31].

They stated that "the number of predicates involved in a path being a good heuristic for assessing a path's feasibility". The greater the number of predicates exist in a path, the greater the probability for the path being infeasible. They further concluded a regression function $f_q=Ke^{-\lambda q}$ to represent the above relationship, in which K and λ are two constants, q stands for the number of predicate nodes involved in a given path, while f_q stands for the possibility of this path being feasible. Later, Vergilio et al. [42] validated the above results over a broader selection of programs and extended the work to involve in more static code attributes, such as: number of nodes, number of variables and number of definitions.

The advantage of such metrics is that they are easy to implement and provide a fast way to predict path infeasibility within a confidence level [42]. However, it is an approach of rough estimation rather than accurate detection. The accuracy of the regression function also biased over different test programs and different programming language.

2.6 Generalization of Infeasible Paths Approach

When a path is infeasible in a CFG, all other paths that contain the path are also clearly infeasible. Based on this simple concept, Delahaye proposed to generate all infeasible paths from a given set of seed infeasible paths [12]. It provides a convenient way to generate error seeded models for further testing, especially for legacy programs or combined as a component in regression testing.

3 Tools Implementation

In most program optimization, software analysis and testing tools, infeasible path detection usually appears as an important component. Best to our knowledge, there is no independent tool particularly designed for it. In this section, we introduce related existing tools based on above approaches. We also brief the implementation details of the approaches described in last section to help those who want to detect infeasible paths in their own applications.

We select 14 relevant tools and analyze them in this section. These tools are summarized in Table 1.

3.1 Data Flow Analysis Approach

The most completed work is from Gustafsson et al., [21]; [19]; [20]; [22]. In order to detect infeasible paths over complex programs, they decompose a program into several linked scopes. The latter is a set of program statements within a call of one procedure or an execution of a loop iteration. It is statically created so that calls to a function or a loop at different call sites will be marked and analyzed separately. This brings in higher precision but more expensive computation cost. The scope graph is hierarchical representation of the structure of a program which is used to describe the interaction relationships between scopes.

Table 1. Useful Tools for Implementation

Name	Description	Link	Supported Language
Code analysis and optimization			
Soot	A Java Optimization Framework.	http://www.sable.mcgill.ca/soot/	Java
CodeSurfer	Code analyser	www.grammatech.com/products/codesurfer/	C/C++
Pixy	Code and security analyser for PHP	http://pixybox.seclab.tuwien.ac.at/pixy/	PHP
Automated Test Case Generation			
CUTE	Test case generation for C/C++	http://cute-test.com/wiki/cute/	C/C++
jCute	Test case generation for JAVA	http://cute-test.com/wiki/jcute/	JAVA
CREST	Test case generation for C	http://code.google.com/p/crest/	C
Pex	Structural testing framework for .Net	http://research.microsoft.com/en-us/projects/pex/	C#, C++, VB.net
eToc	Path selection with genetic algorithm and test case generation for C/C++, Java	http://www.informatik.hu-berlin.de/forschung/gebiete/sam/Forschung%20und%20Projekte/aktuelle-forschung-und-projekte/softwarekomponenten-entwicklung/eodl-projects/etoc/etoc	Java, C/C++
Theorem Prover			
Z3	SMT prover	http://research.microsoft.com/en-us/um/redmond/projects/z3/	C/C++
Lp_Solver	Linear constraint solver	http://lpsolve.sourceforge.net/5.5/	C/C++, JAVA, PHP, Matlab
BLAST	Lazy abstraction software verification	http://mtc.epfl.ch/software-tools/blast/index-epfl.php	C
Verification and Error detection			
ESC-Java	Error checking for annotated Java program	http://secure.ucd.ie/products/opensource/ESCJava2/	Java
SLAM	Verify critical properties for C/C++	http://research.microsoft.com/en-us/projects/slam/	C/C++
LCLint	Static analysis for C/C++	http://www.splint.org/guide/sec1.html	C/C++

Data flow analysis based on abstract interpretation will be performed over each scope separately to compute the live variables set. A recorder is created to store the infeasible paths detected within each scope. Among the work of Gustafsson et al., only primary data types are mentioned. There has been no corresponding open source toolkit published. For readers planning to code based on this type of approach, they could utilize existing data flow analysis framework to find out variables mapped to an empty value set at certain control locations and detect infeasible paths accordingly.

For example, the sub package Spark in Soot can perform intra or inter data flow analysis over Java procedures. Other available toolkits are, for example, CodeSufer for C/C++, Pixy for PHP.

3.2 Path-Based Constraint Propagation Approach

Approaches based on constraint propagation and solving often appear in tools of automated test case generation. The typical example is concolic testing [40]. Before test data is generated, the target path will be tested its infeasibility by submitting the path constraint to theorem prover. If it is infeasible, the last predicate condition will be reversed to stand for a new path containing the opposite branch. The details could be found in the following tools: CUTE and jCUTE which are available as binaries under a research-use only license by Urbana-Champaign for C and JAVA; CREST, which is an open-source solution for C comparable to CUTE; Microsoft Pex, which is publicly available as a Microsoft Visual Studio 2010 Power Tool for the NET Framework.

Readers, who are interested in implementing this type of approaches in their own application, can first apply those data flow analysis tools to propagate along a path and generate path constraint in symbolic expressions. Later the constraint could be submitted to theorem provers. Available tools of the latter include Z3, which is a SMT solver from Microsoft; Lp_Solver, which is an open source linear constraint solver under GNU license; BLAST, which is a prover often used to verify abstract program.

3.3 Property Sensitive Data Flow Analysis Approach

Das et al., [11] proposed a tool called ESP, which is a research project for partial verification under Microsoft. ESP is able to construct CFG and perform property sensitive analysis in either intra-procedure or inter-procedure mode. The verification for given properties at control location is handled by its build-in rules for primary data types in C/C++. But the tool provides interface to replace the build-in rules with standard theorem provers for more complex analysis. There are also several other tools based on this type of approaches, which are like ESC-Java, SLAM, LCLint.

3.4 Syntax-Based Approach

Syntax-based approach is easy to implement because the heuristics are concluded from code and expressed in a straight forward style for implementation. Ngo and Tan [36] implemented an automated test case generation system called jTGEN for automated test data generation for Java. The system consists of an infeasible path detector that based on heuristic code-patterns, a code parser based on Soot, a path selector and a test case generator based on eToc. The system uses a genetic algorithm to select a set of paths. These paths are checked against heuristic code patterns and only feasible paths will be remained for test case generation.

4 Limitation of Current Solutions

Software grows fast in both size and complexity in current trend, more paths and constraints are encountered in the detection of infeasible paths, therefore using traditional symbolic evaluation based approaches for infeasible path detection encounter scalability issue. For approaches of path sensitive analysis, there is a need to limit the number of the targeting paths. The simplest way is to set an upper bound to limit the paths number. Possible effort could be applying intelligent approach like genetic algorithm to guide the path selection [45]: by choosing proper fitness function, only paths with high suspicion of infeasibility would be remained for further processing. Another attempt is from Forgács and Bertolino [17]) who utilized program slicing: By reducing a program to a slice of the variables and statement concerned, the detection of infeasible paths is therefore made simpler.

Theoretically, it is believed that program complexity will highly raise the difficulty of detecting infeasible path. Because the path may contains long data dependency, complex data types, side-effect functions, and non-linear operators. It will be with high cost to develop a general model to cover them. It is also not possible to determine the infeasibility of all cases. As infeasible path detection could be viewed as a type of model abstraction and verification. Snifakis recently suggested [10] that general verification theory would be of theoretical interest only. By contrast, a compositional study for particular classes of properties or systems would be highly attractable.

5 Challenges and Potential Practical Solution

Detection of infeasible paths remains an important problem in software engineering. Current approaches mainly apply full symbolic evaluation instrumented with constraint solvers to detect infeasible paths. Symbolic evaluation is inherently computational expensive. Furthermore, theoretically, constraints imposed by branches that a path follows can be in any form. Therefore, it is unsolvable to determine the infeasibility of a path in general. These are the major challenges currently.

However, theoretical limitation does not always imply practical limitation. Despite the theoretical limitation, one might still develop a good practical solution if there are useful practical characteristics one can take advantage.

More specifically, we propose to investigate the characteristics of constraints imposed by the branches that may lead a path in real system code to be infeasible. If one can empirically identify interesting characteristics that majority of these constraints are possessing, one might be able to take advantage from them to establish good practical approaches to improve on both the precision and the proportion of infeasible paths that can be detected by current approaches. Clearly, systematic experiment instrumented with both automated and manual evaluation to examine the proportion of infeasible paths that an approach can detect is very difficult. However, researchers should still consider spending effort on this to provide the lacking empirical quantitative information on the proportion of infeasible paths that an approach can detect.

If these practical approaches can be invented and implemented to detect majority of the infeasible paths, it will provide major benefit to many related important applications such as code optimization, structural testing and coverage analysis.

6 Conclusions

We have reviewed existing approaches for the detection of infeasible paths. We have also discussed the challenges, the strengths and limitations of current approaches. Noticeably, all the existing approaches cannot detect majority of the infeasible paths efficiently. Most of the existing approaches were proposed under other approaches to solve another problem such as code optimization or test case generation, in which the detection of infeasible paths has great impact.

References

1. Altenbernd, P.: On the False Path Problem in Hard Real-Time Programs. In: Real-Time Systems, Euromicro Conference, p. 102 (1996)
2. Balakrishnan, G., Sankaranarayanan, S., Ivančić, F., Wei, O., Gupta, A.: SLR: Path-Sensitive Analysis through Infeasible-Path Detection and Syntactic Language Refinement. In: Alpuente, M., Vidal, G. (eds.) SAS 2008. LNCS, vol. 5079, pp. 238–254. Springer, Heidelberg (2008)
3. Ball, T., Rajamani, S.K.: The SLAM project: debugging system software via static analysis. SIGPLAN Not. 37, 1–3 (2002)
4. Ball, T., Rajamani, S.K.: Bebop: a path-sensitive interprocedural dataflow engine. Presented at the Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, Snowbird, Utah, United States (2001)
5. Bodik, R., Gupta, R., Soffa, M.L.: Refining data flow information using infeasible paths. SIGSOFT Softw. Eng. Notes 22, 361–377 (1997)
6. Bjørner, N., Tillmann, N., Voronkov, A.: Path Feasibility Analysis for String-Manipulating Programs. Presented at the Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems: Held as Part of the Joint European Conferences on Theory and Practice of Software, York, UK (2009)
7. Botella, B., Delahaye, M., Hong-Tuan-Ha, S., Kosmatov, N., Mouy, P., Roger, M., Williams, N.: Automating structural testing of C programs: Experience with PathCrawler. In: Automation of Software Test, ICSE Workshop, pp. 70–78 (2009)
8. Cobleigh, J.M., Clarke, L.A., Ostenveil, L.J.: The right algorithm at the right time: comparing data flow analysis algorithms for finite state verification. Presented at the Proceedings of the 23rd International Conference on Software Engineering, Toronto, Ontario, Canada (2001)
9. Concolic testing, `http://en.wikipedia.org/wiki/Concolic_testing` (retrieved)
10. Edmund, M., Clarke, E., Allen, E., Joseph, S.: Model Checking: Algorithmic Verification and Debugging. Communications of the ACM 52, 74–84 (2009)
11. Das, M., Lerner, S., Seigle, M.: ESP: path-sensitive program verification in polynomial time. SIGPLAN Not. 37, 57–68 (2002)
12. Delahaye, M., Botella, B., Gotlieb, A.: Explanation-Based Generalization of Infeasible Path. Presented at the Proceedings of the 2010 Third International Conference on Software Testing, Verification and Validation (2010)
13. Dor, N., Adams, S., Das, M., Yang, Z.: Software validation via scalable path-sensitive value flow analysis. SIGSOFT Softw. Eng. Notes 29, 12–22 (2004)

14. Dwyer, M.B., Clarke, L.A., Cobleigh, J.M., Naumovich, G.: Flow analysis for verifying properties of concurrent software systems. ACM Trans. Softw. Eng. Approachol. 13, 359–430 (2004)
15. Ermedahl, A.: A modular tool architecture for worst-Case execution time Analysis. PHD thesis, Uppsala University, Dept. of Information Technology, Uppsala University, Sweden (June 2003)
16. Fischer, J., Jhala, R., Majumdar, R.: Joining dataflow with predicates. SIGSOFT Softw. Eng. Notes 30(5), 227–236 (2005)
17. Forgács, I., Bertolino, A.: Feasible test path selection by principal slicing. SIGSOFT Softw. Eng. Notes 22, 378–394 (1997)
18. Goldberg, A., Wang, T.C., Zimmerman, D.: Applications of feasible path analysis to program testing. Presented at the Proceedings of the 1994 ACM SIGSOFT International Symposium on Software Testing and Analysis, Seattle, Washington, United States (1994)
19. Gustafsson, J.: Worst case execution time analysis of Object-Oriented programs. In: The Proceedings of the Seventh International Workshop on Object-Oriented Real-Time Dependable Systems, San Diego, CA, USA, pp. 0071–0071 (2002)
20. Gustafsson, J., Ermedahl, A., Sandberg, C., Lisper, B.: Automatic derivation of loop bounds and infeasible paths for WCET analysis using abstract execution. Presented at the Proceedings of the 27th IEEE International Real-Time Systems Symposium (2006)
21. Gustafsson, J.: Analyzing execution-time of Object-Oriented programs using abstract interpretation. PhD thesis, Department of Computer Systems, Information Technology, Uppsala University (2000)
22. Gustafsson, J., Ermedahl, A., Lisper, B.: Algorithms for Infeasible Path Calculation. In: Sixth International Workshop on Worst-Case Execution Time Analysis, Dresden, Germany (2006)
23. Hampapuram, H., Yang, Y., Das, M.: Symbolic path simulation in path-sensitive dataflow analysis. SIGSOFT Softw. Eng. Notes 31, 52–58 (2005)
24. Hedley, D., Hennell, M.A.: The cause and effects of infeasible paths in computer programs. Presented at the Proceedings of the 8th International Conference on Software Engineering, London, England (1985)
25. Khedker, U., Sanyal, A., Karkare, B.: Data flow analysis: theory and practice. Taylor and Francis (2009)
26. Korel, B.: Automated test data generation for programs with procedures. SIGSOFT Softw. Eng. Notes 21, 209–215 (1996)
27. Liu, H., Tan, H.B.K.: Covering code behavior on input validation in functional testing. Information and Software Technology 51(2), 546–553 (2009)
28. Liu, H., Tan, H.B.K.: Testing input validation in web applications through automated model recovery. Journal of Systems and Software 81(2), 222–233 (2008)
29. Liu, H., Tan, H.B.K.: An Approach for the maintenance of input validation. Information and Software Technology 50, 449–461 (2008)
30. Liu, H., Tan, H.B.K.: An approach to aid the understanding and maintenance of input validation. In: Proceedings 22nd International Conference on Software Maintenance (ICSM 2006), pp. 370–379 (September 2006)
31. Malevris, N., Yates, D.F., Veevers, A.: Predictive metric for likely feasibility of program paths. Information and Software Technology 32, 115–118 (1990)
32. McMinn, P.: Search-based software test data generation: a survey: Research Articles. Softw. Test. Verif. Reliab. 14, 105–156 (2004)

33. Moura, L.D., Bjorner, N.: Z3: an efficient SMT solver. Presented at the Proceedings of the Theory and practice of software. In: 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, Budapest, Hungary (2008)
34. Ngo, M.N., Tan, H.B.K.: Applying static analysis for automated extraction of database interactions in web applications. Information and Software Technology 50(3), 160–175 (2008)
35. Ngo, M.N., Tan, H.B.K.: Heuristics-based infeasible path detection for dynamic test data generation. Inf. Softw. Technol. 50, 641–655 (2008)
36. Ngo, M.N., Tan, H.B.K.: Detecting Large Number of Infeasible Paths through Recognizing their Patterns. In: Proceedings ESEC-FSE 2007, Joint European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering, pp. 215–224. ACM Press (2007)
37. Padmanabhuni, B., Tan, H.B.K.: Defending against Buffer-Overflow Vulnerabilities. IEEE Computer 44(11), 53–60 (2011)
38. Prather, R.E., Myers, J.P.: The Path Prefix Software Engineering. IEEE Trans. on Software Engineering (1987)
39. Robinson, A.J.A., Voronkov, A.: Handbook of Automated Reasoning, vol. II. North-Holland (2001)
40. Sen, K., Marinov, D., Agha, G.: CUTE: a concolic unit testing engine for C. SIGSOFT Softw. Eng. Notes 30, 263–272 (2005)
41. Tahbildar, H., Kalita, B.: Automated Software Test Data Generation: Direction of Research. International Journal of Computer Science and Engineering Survey 2, 99–120 (2011)
42. Vergilio, S., Maldonado, J., Jino, M.: Infeasible paths within the context of data flow based criteria. In: The VI International Conference on Software Quality, Ottawa, Canada, pp. 310–321 (1996)
43. Williams, N.: Abstract path testing with PathCrawler. Presented at the Proceedings of the 5th Workshop on Automation of Software Test, Cape Town, South Africa (2010)
44. Wilhelm, R., Engblom, J., Ermedahl, A., Holsti, N., Thesing, S., Whalley, D., Bernat, G., Ferdinand, C., Heckmann, R., Mitra, T., Mueller, F., Puaut, I., Puschner, P., Staschulat, J., Stenstrom, P.: The worst-case execution-time problem–overview of approaches and survey of tools. ACM Trans. Embed. Comput. Syst. 7, 1–53 (2008)
45. Xie, T., Tillmann, N., Halleux, P.D., Schulte, W.: Fitness-Guided Path Exploration in Dynamic Symbolic Execution. Presented at the IEEE/IFIP International Conference on Dependable Systems & Networks, Lisbon (2009)
46. Zhang, J., Wang, X.: A constraint solver and its application to path feasibility analysis. International Journal of Software Engineering and Knowledge Engineering 11, 139–156 (2001)

A Formal Monitoring Technique for SIP Conformance Testing

Xiaoping Che, Felipe Lalanne, and Stephane Maag

Telecom SudParis, CNRS UMR 5157, 9 rue Charles Fourier, 91011 Evry Cedex, France
{xiaoping.che,felipe.lalanne,
stephane.maag}@telecom-sudparis.eu

Abstract. Formal approaches provide many keys to efficiently test the conformance of communicating protocols. Active and passive testing techniques are two main sets of these approaches. Compare to active testing, passive testing techniques are used whenever the system cannot be interrupted, or its interfaces are unavailable to access. Under such conditions, communication traces are extracted from points of observation and compared with the expected conformance requirements formally specified as properties. This paper presents a novel monitoring approach, aiming at formally specifying protocol properties in order to check them on real execution traces. A prototype is developed and experienced based on the algorithms defined in a previous paper. Experiments are presented through a set of IMS/SIP properties and numerous execution traces in order to evaluate and assess our technique. The relevant verdicts and discussions are provided at the end.

Keywords: Formal Methods, Passive Testing, Monitoring, Protocols, IMS/SIP.

1 Introduction

Two main types of formal approaches can be applied to test the conformance of communicating protocols: active and passive testing. While active testing techniques are based on the analysis of the protocol answers when it is stimulated, the passive ones focus on the observation of input and output events of the implementation under test (IUT) in run-time. They are called *passive* since they do not modify the runtime behavior of the IUT. Although passive testing does lack some of the advantages of active techniques, such as test coverage, it provides an effective tool for fault detection when the access to the interfaces of the system is unavailable, or in already deployed systems, where the system cannot be interrupted. In order to check conformance of the IUT, the record of the observation during runtime (called *trace*) is compared with the expected behavior, defined by either a formal model [1] (when available) or as a set of formally specified properties [2] obtained from the requirements of the protocol.

In the context of black-box testing of communicating protocols, executions of the system are limited to communication traces, i.e. inputs and outputs to and from the IUT. Since passive testing approaches derive from model-based methodologies [3], such input/output events are usually modeled as: a *control* part, an identifier for the event belonging to a finite set, and a *data* part, a set of parameters accompanying the control

L.A. Maciaszek and J. Filipe (Eds.): ENASE 2012, CCIS 410, pp. 79–94, 2013.

part. In these disciplines, properties are generally described as relations between control parts, where a direct causality between inputs and outputs is expected (as in finite state-based methodologies) or a temporal relation is required. In modern message-based protocols (e.g. SIP [4]), while the control part still plays an important role, data is essential for the execution flow. Input/output causality cannot be assured since many outputs may be expected for a single input. Moreover when traces are captured on centralized services, many equivalent messages can be observed due to interactions with multiple clients. That is why temporal relations cannot be established solely through control parts. Furthermore, although the traces are finite, the number of related packets may become huge and the properties to be verified complex.

In this work, we present a passive testing approach for communicating protocols based on the formal specification of the requirements and their analysis on collected runtime execution traces. The related works are present in Section 2. In Section 3, a Horn based logic is defined to specify the properties to be verified. Both the syntax and a three-valued semantics are provided. An algorithm has been defined in a previous work [5] to evaluate the satisfaction of properties on off-line traces. Our approach has been implemented and relevant experiments are depicted in Section 4. Finally, we discuss and conclude our paper in Section 5.

2 Related Work

Formal methods for conformance testing have been used for years to prove correctness of implementations by combining test cases evaluation with proofs of critical properties. In [3] the authors present a description of the state of the art and theory behind these techniques. Passive testing techniques are used to test already deployed platforms or when direct access to the interfaces is not available. Most of these techniques only consider control portions [1,6], data portion testing is approached by evaluation of traces in state based models, testing correctness in the specification states and internal variable values. Our approach, although inspired by it, is different in the sense that we test critical properties directly on the trace, and only consider a model (if available) for potential verification of the properties. Our research is also inspired from the runtime monitoring domain. Though runtime monitoring techniques are mainly based on model checking while we do not manipulate any models, some proposed languages to describe properties are relevant for our purpose. The authors of [7] provide a good survey in this area.

In [2], an invariant approach taking into account control parts has been presented. In [8], the authors have defined a methodology for the definition and testing of time extended invariants, where data is also a fundamental principle in the definition of formulas and a packet (similar to a message in our work) is the base container data. In this approach, the packet satisfaction to certain events is evaluated. However, data relation between multiple packets is not allowed.

Although closer to runtime monitoring, the authors of [9] propose a framework for defining and testing security properties on Web Services using the Nomad [10] language. As a work on Web services, data passed to the operations of the service is taken into account for the definition of properties, and multiple events in the trace can be

compared, allowing to define, for instance, properties such as "Operation op can only be called between operations login and logout". Nevertheless, in Web services operations are atomic, that is, the invocation of each operation can be clearly followed in the trace, which is not the case with network protocols, where operations depend on many messages and sometimes on the data associated with the messages. In [11], the authors propose a logic for runtime monitoring of programs, called **EAGLE**, that uses the recursive relation from LTL $\mathbf{F}\phi \equiv \phi \vee X\phi$(and its analogous for the past), to define a logic based only on the operators next (represented by $\bigcirc$) and previous (represented by $\odot$). Formulas are defined recursively and can be used to define other formulas. Constraints on the data variables and time constraints can also be tested by their framework. However, their logic is propositional in nature and their representation of data is aimed at characterizing variables and variable expressions in programs, which makes it less than ideal for testing message exchanges in a network protocol.

3 Formal Passive Testing Approach

3.1 Basics

A message in a communication protocol is, using the most general possible view, a collection of data fields belonging to multiple domains. Data fields in messages, are usually either *atomic* or *compound*, i.e. they are composed of multiple elements (e.g. a URI sip: name@domain.org). Due to this, we also divide the types of possible domains in *atomic*, defined as sets of numeric or string values[1], or *compound*, as follows.

Definition 1. *A compound value v* of length $k > 0$, is defined by the set of pairs $\{(l_i, v_i) \mid l_i \in L \wedge v_i \in D_i \cup \{\epsilon\}, i = 1...k\}$,where $L = \{l_1, ..., l_k\}$ is a predefined set of labels and D_i are data domains, not necessarily disjoint.

In a *compound* value, in each element (l, v), the label l represents the functionality of the piece of data contained in v. The length of each compound value is fixed, but undefined values can be allowed by using ϵ (null value). A *compound domain* is then the set of all values with the same set of labels and domains defined as $\langle L, D_1, ..., D_k\rangle$. Notice that, D_i being domains, they can also be either *atomic* or *compound*, allowing for recursive structures to be defined. Finally, given a network protocol P, a compound domain M_p can generally be defined, where the set of labels and element domains derive from the message format defined in the protocol specification. A *message* of a protocol P is any element $m \in M_p$.

Example. A possible message for the SIP protocol, specified using the previous definition is

$m = \{(method,$ **'INVITE'**$), (status, \epsilon), (from,$ 'john@b.org'$), (to,$ 'paul@b.org'$), (cseq,$ $\{(num, 10), (method,$ **'INVITE'**$)\})\}$

representing an **INVITE** request (a call request) from *john@b.org to paul@b.org*. Notice that the value associated to the label cseq is also a compound value, {(num,10), (method, **'INVITE'**}.

[1] Other values may also be considered atomic, but we focus here, without loss of generality, to numeric and strings only.

Accessing data inside messages is a basic requirement for the current approach. In order to reference elements inside a compound value, the syntax $v.l_1, l_2, ..., l_n$ is used, where v is a compound value, and l_i are labels. In the previous example, $m.cseq.num$ references the value associated with the label num inside the value for $cseq$ (the value 10). If the reference does not exist, it is associated to ϵ.

A *trace* is a sequence of messages of the same domain (i.e. using the same protocol) containing the interactions of an entity of a network, called the *point of observation* (P.O), with one or more peers during an indeterminate period of time (the life of the P.O).

Definition 2. Given the domain of messages M_p for a protocol P. A trace is a sequence $\Gamma = m_1, m_2, ...$ of potentially infinite length, where $m_i \in M_p$.

Definition 3. Given a trace $\Gamma = m_1, m_2, ...$ a *trace segment* is any finite sub-sequence of Γ, that is, any sequence of messages $\rho = m_i, m_{i+1}, ..., m_{j-1}, m_j (j > i)$, where ρ is completely contained in Γ (same messages in the same order). The order relations $\{<, >\}$ are defined in a trace, where for $m, m' \in \rho, m < m' \Leftrightarrow pos(m) < pos(m')$ and $m > m' \Leftrightarrow pos(m) > pos(m')$ and $pos(m) = i$, the position of m in the trace $(i \in \{1, ..., len(\rho)\})$.

As testing can only be performed in trace segments, in the rest of the document, trace will be used to refer to a trace segment unless explicitly stated.

3.2 Syntax and Semantics for the Tested Properties

A syntax based on Horn clauses is used to express properties. The syntax is closely related to that of the query language Datalog, described in [12], for deductive databases, however, extended to allow for message variables and temporal relations. Both syntax and semantics are described in the current section.

Syntax. Formulas in this logic can be defined with the introduction of terms and atoms, as defined below.

Definition 4. A *term* is either a constant, a variable or a *selector variable*. In BNF: $t ::= c \mid x \mid x.l.l...l$ where c is a constant in some domain (e.g. a message in a trace), x is a variable, l represents a label, and $x.l.l...l$ is called a *selector variable*, and represents a reference to an element inside a compound value, as defined in Definition 1.

Definition 5. An *atom* is defined as

$$A ::= \overbrace{p(t, ..., t)}^{k} \mid t = t \mid t \neq t$$

where t is a term and $p(t, ..., t)$ is a predicate of label p and arity k. The symbols = and $\neq$ represent the binary relations "equals to" and "not equals to", respectively.

In this logic, relations between terms and atoms are stated by the definition of clauses. A *clause* is an expression of the form $A_0 \leftarrow A_1 \wedge ... \wedge A_n$, where A_0, called the head of the clause, has the form $A_0 = p(t_1^*, ..., t_1^k)$, where t_1^* are a restriction on terms for the head of the clause (t^* = c | x). $A_1 \wedge ... \wedge A_n$ is called the body of the clause, where A_i are atoms.

A formula is defined by the following BNF:

$$\phi ::= A_1 \wedge ... \wedge A_n \mid \phi \rightarrow \phi \mid \forall_x \phi \mid \forall_{y>x} \phi \mid \forall_{y<x} \phi \mid \exists_x \phi \mid \exists_{y>x} \phi \mid \exists_{y<x} \phi$$

where $A_1, ..., A_n$ are atoms, $n \geq 1$ and x, y are variables. Some more details regarding the syntax are provided in the following

- The $\rightarrow$ operator indicates causality in a formula, and should be read as *"if-then"* relation.
- The $\forall$ and $\exists$ quantifiers, are equivalent to its counterparts in predicate logic. However, as it will be seen on the semantics, here they only apply to messages in the trace. Then, for a trace $\rho, \forall_x$ is equivalent to $\forall(x \in \rho)$ and $\forall_{y<x}$ is equivalent to $\forall(y \in \rho; y < x)$ with the '$<$' indicating the order relation from Definition 3. These type of quantifiers are called *trace temporal quantifiers*.

Semantics. The semantics used on this work is related to the traditional Apt–Van Emdem–Kowalsky semantics for logic programs [13], however we introduce an extension in order to deal with messages and trace temporal quantifiers. We begin by introducing the concept of substitution (as defined in [14]).

Definition 6. A *substitution* is a finite set of bindings $\theta = \{x_1/t_1, ..., x_k/t_k\}$ where each t_i is a term and x_i is a variable such that $x_i \neq t_i$ and $x_i \neq x_j$ if $i \neq j$.

The *application* $x\theta$ of a substitution θ to a variable x is defined as follows.

$$x\theta = \begin{cases} t & \text{if } x/t \in \theta \\ x & \text{otherwise} \end{cases}$$

The application of a particular binding *x/t* to an expression *E* (atom, clause, formula) is the replacement of each occurrence of x by t in the expression. The application of a substitution θ on an expression *E*, denoted by $E\theta$, is the application of all bindings in θ to all terms appearing in *E*.

Given $K = \{C_1, ..., C_p\}$ a set of clauses and $\rho = m_1, ..., m_n$ a trace. An *interpretation*[2] *in logic programming* is any function *I* mapping an expression *E* that can be formed with elements (clauses, atoms, terms) of *K* and terms from ρ to one element of $\{\top, \bot\}$. It is said that *E* is true in *I* if *I(E)* = $\top$.

The **semantics of formulas** under a particular interpretation *I*, is given by the following rules.

– The expression $t_1 = t_2$ is true, iff t_1 equals t_2 (they are the same term).
– The expression $t_1 \neq t_2$ is true, iff t_1 is not equal to t_2 (they are not the same term).
– A ground $atom$[3] $A = p(c_1, ..., c_k)$ is true, iff $A \in I$.
– An atom A is true, iff every ground instance of A is true in I.
– The expression $A_1 \wedge ... \wedge A_n$, where A_i are atoms, is true, iff every A_i is true in I.
– A clause $C : A_0 \leftarrow B$ is true, iff every ground instance of C is true in I.
– A set of clauses $K = \{C_1, ..., C_p\}$ is true, iff every clause C_i is true in I.

[2] Called an *Herbrand Interpretation*.
[3] An atom where no unbound variables appear.

An interpretation is called a *model* for a clause set $K = \{C_1, ..., C_p\}$ and a trace ρ if every $C_i \in K$ is true in I. A formula ϕ is true for a set K and a trace ρ (true in K,ρ, for short), if it is true in $every$ model of K, ρ. It is a known result [13] that if M is a $minimal$ model for K, ρ, then if $M(\phi) = \top$, then ϕ is true for K, ρ.

The general semantics of formulas is then defined as follows. Let K be a clause set, ρ a trace for a protocol and M a minimal model, the operator M defines the semantics of formulas.

$$\hat{M}(A_1 \wedge ... \wedge A_n) = \begin{cases} \top & \text{if } M(A_1 \wedge ... \wedge A_n) = \top \\ \bot & \text{otherwise} \end{cases}$$

The semantics for trace quantifiers requires first the introduction of a new truth value '?' (inconclusive) indicating that no definite response can be provided. The semantics of quantifiers $\forall$ and $\exists$ is defined as follows

$$\hat{M}(\forall_x \phi) = \begin{cases} \bot & \text{if } \exists\theta \text{ with } x/m \in \theta \text{ and } m \in \rho, \\ & \text{where } \hat{M}(\phi\theta) = \bot \\ ? & \text{otherwise} \end{cases}$$

$$\hat{M}(\exists_x \phi) = \begin{cases} \top & \text{if } \exists\theta \text{ with } x/m \in \theta \text{ and } m \in \rho, \\ & \text{where } \hat{M}(\phi\theta) = \top \\ ? & \text{otherwise} \end{cases}$$

Since ρ is a finite segment of an infinite execution, it is not possible to declare a '$\top$' result for $\forall_x \phi$, since we do not know if ϕ may become '$\bot$' after the end of ρ. Similarly, for $\exists_x \phi$, it is unknown whether ϕ becomes true in the future. Similar issues occur in the literature of passive testing [2] and runtime monitoring [15], for evaluations on finite traces. This issue is further detailed in one of our technical reports [5].

The rest of the quantifiers are detailed in the following, where x is assumed to be found as a message previously obtained by $\forall_x$ or $\exists_x$

$$\hat{M}(\forall_{y>x} \phi) = \begin{cases} \bot & \text{if } \exists\theta \text{ with } y/m \in \theta, \\ & \text{where } \hat{M}(\phi\theta) = \bot \text{ and } m > x \\ ? & \text{otherwise} \end{cases}$$

$$\hat{M}(\exists_{y>x} \phi) = \begin{cases} \top & \text{if } \exists\theta \text{ with } y/m \in \theta, \\ & \text{where } \hat{M}(\phi\theta) = \top \text{ and } m > x \\ ? & \text{otherwise} \end{cases}$$

The semantics for $\forall_{y<x}$ and $\exists_{y<x}$ is equivalent to the last two formulas, exchanging $>$ by $<$. Finally, the truth value for $\hat{M}(\phi \rightarrow \psi) \equiv \hat{M}(\phi) \rightarrow \hat{M}(\psi)$.

3.3 Evaluation Complexity

An algorithm for evaluation of formulas is provided in [5]. The algorithm uses a recursive procedure to evaluate formulas, coupled with a modification of SLD (Selective

Linear Definite-clause) resolution algorithm [16] for evaluation of Horn clauses. In the same work it is shown that the worst-case time complexity for a formula with k quantifiers is $\mathcal{O}(n^k)$ to analyze the full trace, where n is the number of messages in the trace. Although the complexity seems high, this corresponds to the time to analyze the complete trace, and not for obtaining individual solutions, which depends on the type of quantifiers used. For instance for a property $\forall_x p(x)$, individual results are obtained in $\mathcal{O}(1)$, and for a property $\forall_x \exists_y q(x,y)$, results are obtained in the worst case in $\mathcal{O}(n)$. Finally, it can also be shown that a formula with a '$\rightarrow$' operator, where Q are quantifiers

$$\underbrace{Q \ldots Q}_{k}(\underbrace{Q \ldots Q}_{l}(A_1 \wedge \ldots \wedge A_p) \rightarrow \underbrace{Q \ldots Q}_{m}(A'_1 \wedge \ldots \wedge A'_q))$$

has a worst-case time complexity of $\mathcal{O}(n^{k+max(l,m)})$, which has advantages with respect to using formulas without the '$\rightarrow$' operator. For instance, evaluation of the formula $\forall_x(\exists_y p(x,y) \rightarrow \exists_z q(z))$ has a complexity of $\mathcal{O}(n^2)$, while the formula $\forall_x \exists_y \exists_z (p(x,y) \wedge q(z))$ has a complexity of $\mathcal{O}(n^3)$ in the worst case.

4 Experiments

The concepts described in the Section 3, along with the above mentioned evaluation algorithm (for a reason of space, the reader interested in this algorithm is invited to see [5]), form part of our implemented framework.[4] The implementation has been performed using Java and is composed of two main modules, as shown by the Figure 1.

The *trace processing* module receives the raw traces collected from the network exchange, and converts the messages from the input format into a list of messages compatible with the clause definitions. Although the module can be adapted to multiple input formats, in our experiments, the input format used was PDML, an XML format that can be obtained from Wireshark[5] traces. In the XML, data values are identified by a `field` tag, representing an individual data element in the message (a header, a parameter). Each sub-element in the target message is related to a field in the XML by its name, for instance, the '`status.line`' message element with the XML field '`sip.Status-Line`'. In the XML, fields are grouped by protocol, which also allows the tool to filter messages not relevant to the properties being tested.

The *tester* module takes the resulting trace from the trace evaluation along with the clause set and the formula to test, and it returns a set of satisfaction results for the formula in the trace, as well as the variable bindings and the messages involved in the result. The results from the experiments are presented in the following.

4.1 IP Multimedia Subsystem Services and SIPp

The IMS (IP Multimedia Subsystem) is a standardized framework for delivering IP multimedia services to users in mobility. It aims at facilitating the access to voice or multimedia services in an access independent way, in order to develop the fixed-mobile

[4] Available at `http://www-public.int-evry.fr/~lalanne/datamon.html`
[5] `http://www.wireshark.org`

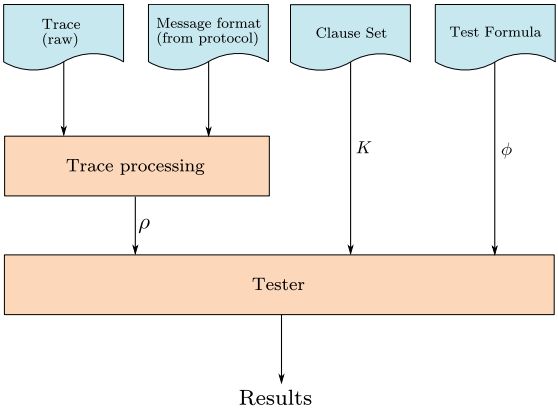


Fig. 1. Architecture for the framework

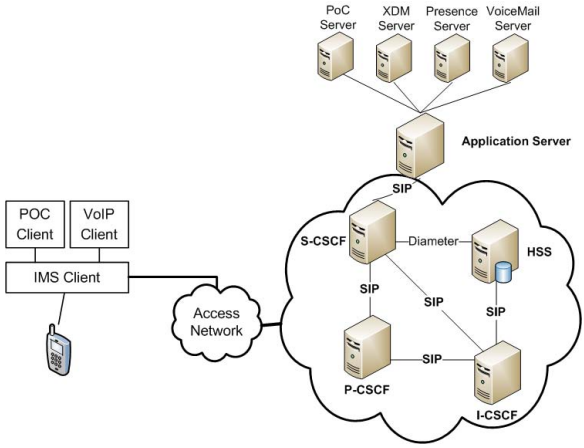


Fig. 2. Core functions of IMS framework

convergence. To ease the integration with the Internet world, the IMS heavily makes use of IETF standards.

The core of the IMS network consists on the Call Session Control Functions (CSCF) that redirect requests depending on the type of service, the Home Subscriber Server (HSS), a database for the provisioning of users, and the Application Server (AS) where the different services run and interoperate. Most communications with the core network and between the services are done using the Session Initiation Protocol [4]. Figure 2 shows the core functions of the IMS framework and the protocols used between the different entities. The Session Initiation Protocol (SIP) is an application-layer protocol that relies on request and response messages for communication, and it is an essential part for communication within the IMS framework. Messages contain a header which provides session, service and routing information, as well as a body part (optional) to complement or extend the header information.

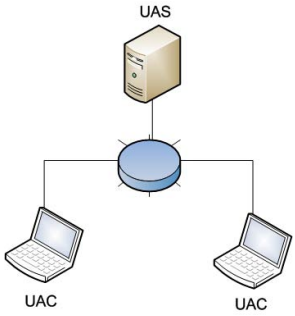


Fig. 3. Architecture of our LAN

For the experiments, traces were obtained from SIPp [17]. SIPp is an Open Source implementation of a test system conforming to the IMS, and it is also a test tool and traffic generator for the SIP protocol, provided by the Hewlett-Packard company. It includes a few basic user agent scenarios (UAC and UAS) and establishes and releases multiple calls with the INVITE and BYE methods. It can also read custom XML scenario files describing from very simple to complex call flows (e.g. subscription including SUBSCRIBE and NOTIFY events). The traces obtained from SIPp contain all communications between the client and the SIP core. Based on these traces and properties extracted from the SIP RFC, tests were performed using our above mentioned methodology and tool. Tests were performed using a prototype implementation of the formal approach mentioned above, using an algorithm we developed and described in [5].

4.2 Architectures

In the experiments, we designed a simulation on Local Area Network (LAN) architecture for testing. For ensuring the accuracy and authenticity of the results, we construct the environment by using real laptops. The LAN architecture is an environment containing several UACs, which can be used to test the correctness, robustness and reliability under tremendous number of calls. The observation points being on the UAS (Fig.3). The HW configuration of UAS is a CPU- Intel Core i5-2520M 2.50 GHz, 4GB DDR3 and the ones of UACs: CPU- AMD Atholon 64 X2 5200+, 2GB DDR2 and CPU- Intel Core2 Duo T6500 2.10 GHz, 2GB DDR2.

4.3 Properties

In order to formally design the properties to be passively tested, we got inspired from the TTCN-3 test suite of SIP [18] and the RFC 3261 of SIP [4]. We designed seven properties for the experiments, for the evaluation of each property we used a set of traces containing $\{500, 1000, 2000, \ldots, 512000$ packets$\}$ in order to get exhaustive results.

1. For every Request there must be a Response. This property can be used for a monitoring purpose, in order to draw further conclusions from the results. Due to the issues related to testing on finite traces for finite executions, a $fail$ results can never be

given for this context. However $inconclusive$ results can be provided and conclusions may be drawn from further analysis of the results (for instance if the same type of message is always without a response). The property evaluated is as follows:

$$\forall(request(x) \wedge x.method! = \textbf{'ACK'} \rightarrow \exists_{y>x}(nonProvisional(y) \wedge responds(y,x)))$$

where $nonProvisional(x)$ accepts all non provisional responses (non-final responses, with $status \geq 200$) to requests with method different than **ACK**, which does not require a response. The results from the evaluation on the traces are shown on Table 1. As expected, most traces show only true results for the property evaluation, but inconclusive results can also be observed. Taking a closer look at trace 10, the inconclusive verdict corresponds to **REGISTER** message, with an Event header corresponding to a conference event [19], this message is at the end of the trace, which could indicate that the client closed the connection before receiving the **REGISTER** message. The same phenomenon can be observed on the other traces (2,4,5 and 7). The last trace with question mark is too huge to be executed due to the limitation of the computer memory, the tool crashed after four hours execution.

Table 1. For every request there must be a response

Trace	No.of messages	Pass	Fail	Inconclusive	Time(s)
1	500	150	0	0	0.941
2	1000	318	0	1	1.582
3	2000	676	0	0	2.931
4	4000	1301	0	1	5.185
5	8000	2567	0	1	10.049
6	16000	5443	0	0	20.192
7	32000	10906	0	1	39.016
8	64000	21800	0	0	84.015
9	128000	43664	0	0	155.903
10	256000	87315	0	1	382.020
11	450000	153466	0	0	1972.720
12	512000	?	?	?	?

2. No Session can be Initiated without a Previous Registration. This property can be used to test that only users successfully registered with the SIP Core can initiate a PoC session (or a SIP call, depending on the service). It is defined using our syntax as follows

$$\forall_x(\exists_{y>x} sessionEstablished(x,y) \rightarrow \exists_{u<x}(\exists_{v>u} registration(u,v)))$$

where $sessionEstablished$ and $registration$ are defined as

$$sessionEstablished(x,y) \leftarrow x.method = \textbf{'INVITE'} \wedge y.statusCode = 200 \wedge responds(y,x)$$
$$registration(x,y) \leftarrow request(x) \wedge responds(y,x) \wedge$$
$$x.method = \textbf{'REGISTER'} \wedge y.statusCode = 200$$

However, the analysis of the results depends on the following condition: did the trace collection begin from a point in the execution of the communication before the user(s) registration took place? If the answer is positive, then inconclusive results can be treated as a possible fault in the implementation, otherwise, only inconclusive verdicts can be given. Unfortunately, in the traces collected such condition does not hold, therefore a

Table 2. No session can be initiated without a previous registration

Trace	No.of messages	Pass	Fail	Inconclusive	Time
1	500	60	0	0	13.690s
2	1000	109	0	0	57.117s
3	2000	182	0	1	207.841s
4	4000	405	0	0	869.322s
5	8000	785	0	0	1.122h
6	16000	1459	0	0	5.660h
7	32000	2905	0	0	27.282h
8	64000	5863	0	1	136.818h
9	128000	?	?	?	?

definitive verdict cannot be provided. However it can be shown that the property and the framework allows to detect when the tested property holds on the trace, as Table 2 illustrates.

From the results on Table 2, it can also be seen that the evaluation of this property is much more time consuming than the one on Table 1. By extrapolation from the same time complexity, the trace 9 will take about 23 days for the evaluation where the same trace took only 155s in property 1. Although this is expected given the complexity of the evaluation (n^2 for the first property vs. n^4 in the current one), the current definition of the property is also quite inefficient, and shows a limitation of the syntax. During evaluation, all combinations of x and y are tested until $sessionEstablished(x, y)$ becomes true, and then all combinations of u and v are evaluated until $registration(u, v)$ becomes true. It would be more efficient to look first for a message with method **INVITE**, then look if the invitation was validated by the server as a response with status 200 to then attempt to look for a registration. This could be achieved, for instance, by allowing quantifiers on the clause definitions. But, the syntax as currently specified does not allow that type of definition.

3. Subscription to Events and Notifications. In the presence service, a user (the watcher) can subscribe to another user's (the presentity) presence information. This works by using the SIP messages **SUBSCRIBE**, **PUBLISH** and **NOTIFY** for subscription, update and notification respectively. These messages also allow the subscription to other types of events other than presence, which is indicated in the header Event on the SIP message. It is desirable then to test that whenever there is a subscription, a notification MUST occur upon an update event. This can be tested with the following formula

$$\forall_x(\exists_{y>x}(subscribe(x, watcher, user, event) \wedge update(y, user, event)) \\ \rightarrow \exists_{z>y} notify(z, watcher, user, event)))$$

where $subscribe$, $update$ and $notify$ hold on **SUBSCRIBE**, **PUBLISH** and **NOTIFY** events respectively. Notice that the values of the variables $watcher$, $user$ and $event$ may not have a value at the beginning of the evaluation, then their value is set by the evaluation of the $subscribe$ clause, shown in the following

$$subscribe(x, watcher, user, event) \leftarrow x.method = \textbf{`SUBSCRIBE'} \wedge watcher = x.from \\ \wedge user = x.to \wedge event = x.event$$

Here, the = operator, compares the two terms, however if one of the terms is an unassigned variable, then the operator works as an assignment. In the formula, the values

Table 3. Whenever an update event happens, subscribed users must be notified on the set of traces

Trace	No.of messages	Pass	Fail	Inconclusive	Time
1	500	3	0	0	10.412s
2	1000	7	0	0	42.138s
3	2000	10	0	0	160.537s
4	4000	19	0	0	632.192s
5	8000	30	0	0	2520.674s
6	16000	52	0	0	2.808h
7	32000	74	0	0	11.250h
8	64000	122	0	0	45.290h
9	128000	?	?	?	?

assigned on the evaluation of $subscribe$ will be then used for comparison in the evaluation of $update$. This is another way of defining formulas, different from using only message attributes.

The results of evaluating the formula are shown on Table 3. The results show no inconclusive results, although they also show that the full notification sequence is quite few in most traces. Notice that we are explicitly looking for a sequence $subscribe \rightarrow update \rightarrow notify$, however the sequence $subscribe \rightarrow notify$ can also be present for subscription to server events, therefore **SUBSCRIBE** and **NOTIFY** events might also appear on the trace.

Similarly to property 2, this property is quite inefficient in its evaluation, due to the same nesting of quantifiers. The evaluation time can be improved by rewriting the property as

$$\forall_x(update(x, user, event) \rightarrow (\exists_{y<x} subscribe(y, watcher, user, event) \\ \rightarrow \exists_{z>x} notify(z, watcher, user, event)))$$

which can be understood as: “if an update event is found, then if a previous subscription exists to such event, then a notification must be provided at some point after the update event”. The results of evaluating this property are shown on Table 4. Notice that for trace 1,3 and 7, a different number of true results are returned. This is due to the order of search given by the property, in the previous property, one pair **SUBSCRIBE - PUBLISH** was sufficient to return a result. In the current property, for each **PUBLISH** it will look for a matching **SUBSCRIBE**. Since for every subscription there can exist multiple updates, the number of true results differs.

Table 4. If an update event is found, then if a previous subscription exists, then a notification must be provided

Trace	No.of messages	Pass	Fail	Inconclusive	Time(s)
1	500	4	0	0	0.560
2	1000	7	0	0	1.158
3	2000	11	0	0	3.089
4	4000	19	0	0	6.164
5	8000	30	0	0	12.684
6	16000	52	0	0	25.416
7	32000	75	0	0	50.130
8	64000	122	0	0	99.372
9	128000	198	0	0	202.492
10	256000	342	0	0	394.756
11	512000	?	?	?	?

4. Every 2xx Response for INVITE Request must be responded with an ACK. This property can be used to ensure that when the IUT (UAC) has initiated an **INVITE** client transaction, either it is in the Calling or Proceeding state, on receipt of a Success (200 OK) response, the IUT MUST generate an **ACK** request. The **ACK** request MUST contain values for the Call-ID, From and Request-URI that are equal to the values of those header fields in the **INVITE** request passed to the transport by the client transaction. The To header field in the **ACK** MUST equal the To header field in the 2xx response being acknowledged, and therefore will usually differ from the To header field in the original **INVITE** request by the addition of the tag parameter. The **ACK** MUST contain a single Via header field, and this MUST be equal to the top Via header field (the field without the branch parameter) of the original **INVITE** request. The CSeq header field in the **ACK** MUST contain the same value for the sequence number in the original **INVITE** request, but the value of Method parameter MUST be equal to '**ACK**'. This evaluated property is as follows:

$$\forall_x(request(x) \wedge x.method = \textbf{INVITE} \rightarrow \exists_{y>x}(responds(y,x) \wedge success(y)) \\ \rightarrow \exists_{z>y}(ackResponse(z,x,y)))$$

where $success$ is defined as

$$success(y) \leftarrow y.statusCode >= 200 \wedge y.statusCode < 300$$

and $ackResponse$ is defined as

$$ackResponse(x,y,z) \leftarrow x.method = \textbf{ACK} \wedge x.Call-id = y.Call-id \\ \wedge x.CSeq = y.CSeq \wedge x.CSeq.method = \textbf{ACK} \wedge x.to = z.to \\ \wedge x.From = y.From \wedge x.Request-URI = y.Request-URI \wedge x.TopVia = y.TopVia$$

Table 5. Every 2xx response for **INVITE** request must be responded with an **ACK**

Trace	No.of messages	Pass	Fail	Inconclusive	Time
1	500	60	0	0	1.901s
2	1000	109	0	0	3.665s
3	2000	183	0	0	11.805s
4	4000	405	0	0	40.104s
5	8000	784	0	1	130.611s
6	16000	1459	0	0	522.050s
7	32000	2904	0	1	2237.442s
8	64000	5864	0	0	2.093h
9	128000	11555	0	1	8.630h
10	256000	23154	0	0	37.406h
11	450000	43205	0	0	142.568h
12	512000	?	?	?	?

The inconclusive messages observed in traces 5,7,9 are caused by the same phenomenon described in property 1. Besides, we observe a regular pattern in the results of this property: as the Table 2 and 5 illustrate, with the evaluation of same traces, the sum of Pass and Inconclusive verdicts of each trace in property 4 equal to the sums in property 2. This can be interpreted as the continuity of the transactions.

We are looking for a $sequence_a$: "**REGISTER** $\rightarrow$ 200 $\rightarrow$ **INVITE**" in property 2, on the other side, in property 4 we are searching a $sequence_b$: "**INVITE** $\rightarrow$ 200 $\rightarrow$ **ACK**".

As described in property 2, each **INVITE** must have a previous **REGISTER** message. We can infer a new sequence: "**REGISTER** $\rightarrow$ 200 $\rightarrow$ **INVITE** $\rightarrow$ 200 $\rightarrow$ **ACK**", which means each **ACK** message in the transaction must be corresponded to one **REGISTER** request. Since multiple user conference is not considered in the experiments, we could only conclude: under our particular architecture, the verdict numbers of $sequence_a$ should be equal to the ones of $sequence_b$.

5. Every 300-699 Response for INVITE Request must be responded with an ACK. Similar to the previous one, this property can be used to ensure that when the IUT (UAC) has initiated an **INVITE** client transaction, either it is in the Calling state or Proceeding state, on receipt of a response with status code 300-699, the client transaction MUST be transited to "Completed", and the IUT MUST generate an **ACK** request. The **ACK** MUST be sent to the same address and port which the original **INVITE** request was sent to, and it MUST contain values for the Call-ID, From and Request-URI that are equal to the values in the **INVITE** request. The To header field in the **ACK** MUST equal the To header field in the response being acknowledged. The **ACK** MUST contain a single Via header field, and this MUST be equal to the Via header field of the original **INVITE** request which includes the branch parameter. The CSeq header field in the **ACK** MUST contain the same value for the sequence number in the original **INVITE** request, but the value of Method parameter MUST be equal to '**ACK**'. Similarly to the property above, this property can be applied as:

$$\forall_x(request(x) \wedge x.method = \textbf{INVITE} \rightarrow \exists_{y>x}(responds(y,x) \wedge fail(y)) \\ \rightarrow \exists_{z>y}(ackResponse(z,x,y)))$$

where $fail$ is defined as

$$fail(y) \leftarrow y.statusCode >= 300 \wedge y.statusCode < 700$$

and $ackResponse$ is defined as

$$ackResponse(x,y,z) \leftarrow x.method = \textbf{ACK} \wedge x.Call-ID = y.Call-ID \wedge x.CSeq = y.CSeq \\ \wedge x.CSeq.method = \textbf{ACK} \wedge x.to = z.to \wedge x.From = y.From \wedge x.TopVia = y.TopVia \\ \wedge x.Request-URI = y.Request-URI$$

The only inconclusive verdict in trace 6 is due to the same phenomenon described in property 1. This property has the same evaluation time complexity as the previous one ($O(n^3)$), which should equal or be close to the ones in the property 4. However, the actual evaluation time does not respect it. From the Table 5 and 6, we can observe that the evaluation times of property 5 are always one higher level than the times of property 4. In order to clarify this issue, we can use the following formula to derive it. Assuming that, the processing time of a message is t, the total number of messages is n, the number of **INVITE** is k, there are x success responses and y fail responses. The evaluation time of property 4 can be derived as

$$T_e = x * t + \{\{n - [\tfrac{2x}{y} + \tfrac{(n-k-x-y)}{y}]\} + \{n - 2 * [\tfrac{2x}{y} + \tfrac{(n-k-x-y)}{y}]\} + \{n - 3 * [\tfrac{2x}{y} + \tfrac{(n-k-x-y)}{y}]\} \\ +, ..., +\{n - (y-1) * [\tfrac{2x}{y} + \tfrac{(n-k-x-y)}{y}]\} + \{n - y * [\tfrac{2x}{y} + \tfrac{(n-k-x-y)}{y}]\}\}$$

Table 6. Every 300-699 response for INVITE request must be responded with an ACK

Trace	No.of messages	Pass	Fail	Inconclusive	Time
1	500	10	0	0	3.445
2	1000	18	0	0	10.798
3	2000	49	0	0	34.331
4	4000	91	0	0	137.083
5	8000	165	0	0	557.803
6	16000	367	0	1	1950.656
7	32000	736	0	0	2.103h
8	64000	1403	0	0	8.498h
9	128000	2796	0	0	36.159h
10	256000	5513	0	0	145.088h
11	512000	?	?	?	?

where we know under regular condition $k = x + y$, and the equation becomes:

$$T_e = x * t + t * [n * y - (\sum_{a=1}^{y} a) * \frac{(2x+n-2k)}{y}] = t * [y^2 + \frac{(n-2)y}{2} + \frac{n}{2} + k - y]$$

This always gives a positive result and the *evaluation time is proportional to the value of y (the number of fails)*. Conversely in property 5, *the evaluation time is proportional to the value of x (the number of successes)*. Considering the success responses are 10 times more than the fail ones, the phenomenon that property 5 consumes more time than the property 4 can be well explained.

5 Conclusions and Perspectives

This paper introduces a novel approach to passive testing of network protocol implementation, with a particular focus on IMS services. This approach allows to define high-level relations between messages or message data, and then use such relations in order to define properties that will be evaluated on the trace. The property evaluation returns a $pass$, $fail$ or $inconclusive$ result for a given trace. To verify and test the approach, we design several properties for the evaluation. The approach has been implemented into a framework, and results from testing these properties on tremendous traces collected from IMS service.

The results are positive, the implemented approach allows to define and test complex data relations efficiently, and evaluate the properties successfully. As described in Discussion section, some improvements can be proposed as future works for performance testing. Moreover, we guess that some properties need, for various reasons as mentioned in this paper, to be specified using timers. In this objective, other perspectives are to enhance our syntax and semantics as well as study the more relevant way of formulating an RFC property.

Furthermore, the experiment results in Section 4 perfectly approved the suitability of our approach for conformance testing, which raise a further question: Is our approach suitable for benchmarking the performance of a protocol? As defined in the RFC1242 [20] and RFC2544 [21], a performance benchmark can be indicated as: Accessibility, Communication bandwidth, Maximum frame rate, Communication Latency and Frame loss rate.

We can build a passive performance benchmark system if all these norms are measurable. Currently, the result of property 1 already shown we could certainly test the

Accessibility and Frame loss rate by detecting the number of resent packets from the inconclusive verdicts. But if we want to test the Communication Latency, a $timer$ function needs to be added in order to test the arrival time, which will be the new direction of our future work.

References

1. Lee, D., Chen, D., Hao, R., Miller, R.E., Wu, J., Yin, X.: Network protocol system monitoring-a formal approach with passive testing. IEEE/ACM Transactions on Networking 14(2), 424–437 (2006)
2. Bayse, E., Cavalli, A., Nunez, M., Zaidi, F.: A passive testing approach based on invariants: application to the wap. Computer Networks 48(2), 247–266 (2005)
3. Hierons, R.M., Krause, P., Luttgen, G., Simons, A.J.H.: Using formal specifications to support testing. ACM Computing Surveys 41(2), 176 (2009)
4. Rosenberg, J., Schulzrinne, H., Camarillo, G., Johnston, A., Peterson, J.: Sip: Session initiation protocol (2002)
5. Maag, S., Lalanne, F.: A formal data-centric approach for passive conformance testing of communication protocols. Technical report, Telecom Sud-Paris - 110003-LOR, ISSN 1968-505X (2011)
6. Ural, H., Xu, Z.: An EFSM-based passive fault detection approach. In: Petrenko, A., Veanes, M., Tretmans, J., Grieskamp, W. (eds.) TestCom/FATES 2007. LNCS, vol. 4581, pp. 335–350. Springer, Heidelberg (2007)
7. Leucker, M., Schallhart, C.: A brief account of runtime verification. Journal of Logic and Algebraic Programming 78(5), 293–303 (2009)
8. Morales, G., Maag, S., Cavalli, A., Mallouli, W., Oca, E.D.: Timed extended invariants for the passive testing of web services. In: IEEE International Conference of Web Services (2010)
9. Cao, T.-D., Phan-Quang, T.-T., Felix, P., Castanet, R.: Automated runtime verification for web services. In: IEEE International Conference on Web Services, pp. 76–82 (2010)
10. Cuppens, F., Cuppens-Boulahia, N., Nomad, T.S.: A security model with non atomic actions and deadlines. IEEE (2005)
11. Barringer, H., Goldberg, A., Havelund, K., Sen, K.: Rule-based runtime verification. In: Steffen, B., Levi, G. (eds.) VMCAI 2004. LNCS, vol. 2937, pp. 44–57. Springer, Heidelberg (2004)
12. Abiteboul, S., Hull, R., Vianu, V.: Datalog and Recursion, 2nd edn. Addison-Wesley (1995)
13. Emden, M.V., Kowalski, R.: The semantics of predicate logic as a programming language. Journal of the ACM 23(4), 733–742 (1976)
14. Nilsson, U., Maluszynski, J.: Logic, programming and Prolog, 2nd edn. Wiley (1990)
15. Bauer, A., Leucker, M.: Runtime verification for ltl and tltl. ACM Transactions on Software Engineering and Methodology, 1–68 (2007)
16. Apt, K., Van Emden, M.: Contributions to the theory of logic programming. Journal of the ACM (JACM) 29(3), 841–862 (1982)
17. Hewlett-Packard: SIPp (2004), `http://sipp.sourceforge.net/`
18. ETSI: Methods for testing and specification (mts); conformance test specification for sip. (2004)
19. Rosenberg, J., Schulzrinne, H., Levin, O.: A session initiation protocol (sip) event package for conference state (2006)
20. Bradner, S.: Benchmarking terminology for network interconnection devices (1991)
21. Bradner, S., McQuaid, J.: Benchmarking methodology for network interconnect devices (1991)

Unified Modeling of Static Relationships between Program Elements

Ioana Şora

Department of Computer and Software Engineering, Politehnica University of Timisoara,
Bvd V. Parvan 2, 300223 Timisoara, Romania
ioana.sora@cs.upt.ro
http://www.cs.upt.ro/~ioana

Abstract. Reverse engineering creates primary models of software systems, representing their program elements and the relationships between them. According to the programming language or paradigm, these program elements are functions, variables, classes, methods, etc. The primary models are then used for different analysis purposes such as: impact analysis, modularization analysis, refactoring decisions, architectural reconstruction. Particularities of the programming language or paradigm do not affect the essence of these higher-level analysis tasks. Thus, having a unitary way of modeling program elements and their relationships is essential for the reuse and integration of higher-level analysis tools.

This work proposes a meta-model which provides a unitary way of describing static relationships between program elements. It defines a set of abstract concepts that can be mapped to program elements of both the object-oriented as well as the procedural programming paradigms. The proposed meta-model is validated by the implementation of model extraction tools from programs written in Java, C# (CIL) and ANSI C. The utility of the proposed meta-model has been shown by a number of different analysis applications that use it successfully.

Keywords: Reverse Engineering, Meta-models, Structural Dependencies.

1 Introduction

Reverse engineering of software systems represents the process of analyzing the subject system in order to identify the components of the system and the relationships between them and to create representations of the system in another form or at a higher level of abstraction [1].

The typical steps of a reverse engineering process are: first, primary information is extracted from system artifacts (mainly implementation); second, higher abstractions are created using the primary information. An abstraction process is guided by a particular analysis purpose, such as: design recovery, program comprehension, quality assessment, or as a basis for re-engineering. There is a lot of past, ongoing and future work in the field of reverse engineering [2].

The existence of that many contributions developing different reverse engineering analysis tools creates a new challenge the field: to be able to integrate different tools and to be able to reuse existing analysis infrastructures in different contexts. The research

L.A. Maciaszek and J. Filipe (Eds.): ENASE 2012, CCIS 410, pp. 95–109, 2013.

roadmap of [2] points out the necessity to increase tool maturity and interoperability as one of the future directions for reverse engineering. In order to achieve this, not only should be formats for data exchange unified, but most important common schema or meta-models for information representation must be adopted.

In our work, we propose a meta-model for representing language-independent and, moreover, programming paradigm independent, dependency structures. It arises from our experience of building ART (the Architectural Reconstruction Toolsuite). We started our work on ART for experimenting new approaches of architectural reconstruction of Java systems by combining clustering and partitioning [3]. Soon we wanted to be able to use our reconstruction tools on systems implemented in different languages. Moreover, some of the languages addressed by the architectural reconstruction problem belong to the object-oriented paradigm, such as Java and C#, while other languages such as C are pure procedural languages, but this must not be a relevant detail from the point of view of higher-level architectural reconstruction. Later, the scope of ART extended to support also several different types of program analysis, such as dependency analysis, impact analysis, modularization analysis, plagiarism detection at program design level. In order to support all of these goals, we have to extract and work with models that abstract the relevant traits of primary dependency relationships which occur in different languages, and are still able to serve different analysis purposes.

The remainder of this article is organized as follows. Section 2 presents background information about creating and using structural models in certain kinds of software reverse engineering applications. Section 3 presents the proposed meta-model for representing primary dependency structures. Section 4 presents implementation and usage of the proposed meta-model. Section 5 discusses the proposed meta-model in the context of related work.

2 Creating and Using Structural Models

In reverse engineering approaches, software systems are often modeled as graphs with nodes representing program elements and edges representing relations between these. Such graph-based modelling techniques can be used at different abstraction levels, as identified in [4]: Low-level graph structures, representing information at the level of Abstract Syntax Trees; Middle-level graph structures, representing information such as such as call graphs and program dependence graphs; High-level graph structures, representing architecture descriptions.

In our work we focus on analysis applications that need models at the middle level of abstraction, representing relationships between program elements and which do not include models of the control flow at instruction level.

According to the programming language, the program elements can be: files, packages, modules, classes, functions, variables, fields, methods, etc. The possible relationships vary according to the types of program elements: includes, inherits, uses, calls, contains, etc.

However, many analysis applications treat different (but similar) kinds of program elements in a unitary way, because they are equivalent from the point of view of their analysis purpose. For example, let us consider an application for architecture reconstruction by identifying layers through partitioning [5], [6] or subsystems through clustering

[7], [3]. The program elements involved in this kind of architectural reconstruction are classes, when the system is implemented in the object-oriented paradigm (Java, C#) or modules or files when applied to systems implemented in the structured programming paradigm (C). Relationships model in all cases code dependencies. However, in the object-oriented case, a dependency between two entities (classes) combines inheritance, method invocation, attribute accesses, while in the structured programming a dependency between two entities (modules) comes from accesses to global variables, function calls, use of defined types. Relationships can be characterized by their strength, leading to a weighted graph. Empirical methods may associate different importances to different kinds of relationships and define ways to quantify them. In all the cases, the architectural reconstruction application will work in the same manner over an abstract Dependency Structure Matrix (DSM) [6].

As another example of higher-level analysis application, we consider an application for refactoring of big, uncohesive modules or god-classes. The model used by this application will consist of fine-grained program entities, such as the members of a class, or elements of the same module. Relationships model facts such as a particular attribute or variable being accessed by a particular method or function or a given method or function being called from another method or function. Based on this kind of model of the internal structure of a component, one can identify big components with low internal cohesion, and, by applying clustering algorithms, one can find refactoring solutions such as splitting. Clustering will lead to identify several smaller and more cohesive parts that the big and not cohesive component can be split into. Again, the splitting decision operates on a abstract model (the internal DSM) of a component, which can be built in a similar way, in any situation (the component is a class or a module).

By putting together all these pieces, the extensible architecture of the ART tool-suite resulted as in Figure 1. The primary dependency structure models, built according to the UNIQ-ART meta-model, are the central element.

This approach introduces following major benefits:

- A primary dependency structure model abstracts code written in different programming languages and programming paradigms for which a language mapping and model extractor tool exists. All analysis tools are applied uniformly on extracted abstract models.
- A primary dependency structure model contains enough semantical information of the problem domain (structure of software systems)such that it is able to serve different analysis purposes.
- Different formats (at a syntactic level) can be used for representing the same model, and conversions between these formats can occur safely without information loss.

3 The UNIQ-ART Meta-model

3.1 Description of Our Meta-model

As concluded in the previous section, a reverse engineering toolsuite such as ART needs that primary dependency models are expressed according to an unique schema,

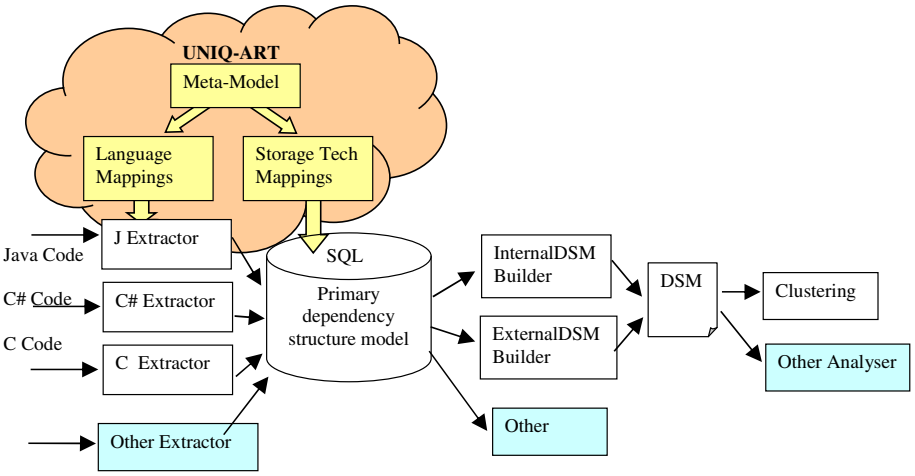


Fig. 1. Impact of UNIQ-ART on the architecture of the ART tool-suite

independent of the programming language in which the modeled systems have been implemented. We define an unique meta-model for representing structural program dependencies, further called the UNIQ-ART meta-model. UNIQ-ART defines the general modeling concepts and relationships, and does not impose any format for model representation. As depicted in Figure 1, different storage technologies and formats can be used (databases, XML files, etc), as long as they provide mappings for representing the elements of the meta-model.

Our general meta-modeling approach can be described as a 4-layered architecture, similar to the OMG's MOF [8], as depicted in Figure 2. The next subsubsections present details of these layers.

The General Approach (The Meta-Meta-layer). The meta-meta-level (Layer M3) contains the following concepts: *ProgramPart*, *AggregationRel*, *DependencyRel*. These are suitable for the analysis techniques of ART which need models able to represent different relationships between different program parts. The program parts refer to structural entities of programs, while the relationships are either aggregation relationships or dependency relationships.

Having the distinction between aggregation relationships and dependency relationships at the highest meta-level, this allows us to easily zoom-in and zoom-out the details of our models, (i.e. a dependency model between classes it can be easily zoomed-out at package level).

The UNIQ-ART Meta-layer. The goal of this meta-layer (Layer M2) in UNIQ-ART is to identify similar constructs in different languages and even different constructs that can be mapped to the same meta-representation. Certain language particularities which are not relevant for dependency-based analyses are lost in this process, but it is a reasonable trade-off when taking into account the fact that it creates a large reuse potential for different analysis tools. The following paragraphs detail how we define

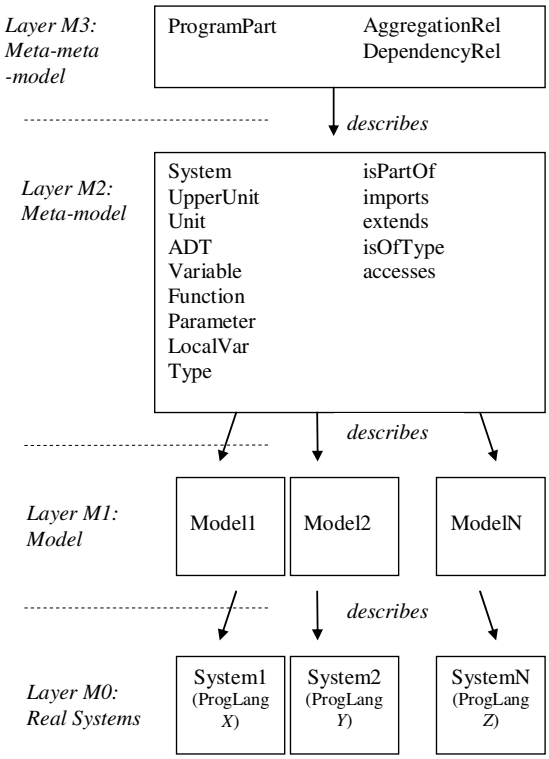


Fig. 2. The UNIQ-ART meta architecture

these concepts and highlight some of the more relevant aspects of their mapping to concrete programming languages with examples related to Java and C.

ProgramPart Instances and Aggregation Relationships in the Meta-layer. The meta-meta-concept *ProgramPart* has following instances at the M2 level: *System*, *UpperUnit*, *Unit*, *ADT*, *Variable*, *Function*, *Parameter*, *LocalVar*, *Type*. These are the types of structural program parts defined in our meta-model. They are mapped to concrete particular concepts of different programming languages.

A *System* is defined as the subject of a reverse engineering task (a program, project, library, etc).

An *Unit* corresponds to a physical form of organization (a file).

An *UpperUnit* represents a means of grouping together several *Units* and/or other *UpperUnits*. It corresponds to organizing code into a directory structure (C) or into packages and subpackages (Java) or by name spaces (C#).

An *ADT* usually represents an Abstract Data Type, either a class in object-oriented languages or a module in procedural languages. Abstract classes and interfaces are also represented as *ADTs*. An ADT is always contained in a single unit of code, but it is possible that a unit of code contains different ADT's, as it is possible for C# code to declare several classes in a single file. In the case of purely procedural languages, when

no other distinction is possible, the whole contents of a unit of code is mapped to a single *Default ADT*, as it is the case with C.

The relationships allowed between program parts at this level are aggregation relationships (*isPartOf*) and dependency relationships(*imports*, *extends*, *isOfType*, *calls*, *accesses*).

Between the different types of structural entities we consider following *isPartOf* aggregation relationships:

A *System* can be composed from several *UpperUnits*. Each *UpperUnit isPartOf* a single *System*.

An *UpperUnit* may contain several other *UpperUnits* and/or *Units*. Each *Unit isPartOf* a single *UperUnit*. There can be a *UpperUnit* which *isPartOf* a single another *UperUnit*.

A *Unit* may contain *ADTs*. Each *ADT isPartOf* a single *Unit*.

An *ADT* contains several *Variables*, *Functions*, *Types*. Each *Variable*, *Function* or *Type isPartOf* a single *ADT*.

An *ADT* corresponds to a module implementing an abstract data type, or to a class, or an interface. In case of procedural modules, the parts of the ADT (*Variables* and *Functions*) represent the global variables, functions and types defined here. In case of classes, the parts of the ADT (*Variables* and *Functions*) are mapped to the fields and methods of the class. Constructors are also represented by *Functions* in our meta-model.

A *Function* is identified by its name and signature. Representing instruction lists of functions is out of the scope of UNIQ-ART. The model records only whether functions declare and use local variables of a certain type. A *Function* contains several *Parameters* and/or *LocalVars*. *Parameter* objects represent both input and output parameters (returned types). Each *Parameter* or *LocalVar* object *isPartOf* a single *Function* object in a model.

Dependency Relationship Instances in the Meta-layer. Besides the aggregation relationships, program parts have also dependency relationships.

The dependency relationships considered here are: *imports*, *extends*, *isOfType*, *calls*, *accesses*.

Import relationships can be established at the level of a *Unit*, which may be in this relationship with any number of other *Units* or *UpperUnits*. It models the situations of using packages, namespaces or including header files.

An *ADT* may also *extend* other *ADTs*. This corresponds to the most general situation of multiple inheritance, where a class can extend several other classes. Extending an abstract class or implementing an interface in object oriented languages which allow this feature is also described by the extend relationship.

In the procedural paradigm, in a language such as C, we consider that the relationship established between a module and its own header file is very similar to implementing an interface and thus map it in our model in this category of *extends* relationships. In consequence, the fact of having a file include another file, will be generally mapped to an *import* relationship, but if the included file is its own header file (it contains declarations of elements defined in the importing file), the fact will be mapped to an *extend* relationship.

Each *Variable*, *Parameter* or *LocalVar* has a type either given by a *ADT*, a *Type* or a primitive construct built-in the language.

The interaction relationships *calls* and *accesses* and can be established between *Functions* and other *Functions* or between *Functions* and *Variables*.

The program parts of a system can establish relationships, as shown above, with other program parts of the same system. It is also possible that they establish relationships with program parts which do not belong to the system under analysis. For example, the called functions or accessed variables do not have to belong to units of the same system: it is possible that a call is made to a function which is outside the system under analysis, such as to an external library function. Another example is a class that is defined in the system under analysis but which extends another class that belongs to an external framework which is used. Such external program parts have to be included in the model of the system under analysis, because of the relationships established between them and some internal program parts. However, the models of such external program parts are incomplete, containing only information relevant for some interaction with internal parts. Units that are external to the system are explicitly marked as external in the model.

3.2 An Example

In this section we illustrate the layers M1 and M0 of UNIQ-ART (see Figure 2) on an example. Only for this presentation purposes, we assume that the modeled system, `LineDrawings`, is implemented in a language presenting both object-oriented and procedural characteristics - permitting both the definition of classes as well as the use of global variables and functions. In order to present as many features of UNIQ-ART in this single example, we combine features of the Java and C programming languages in the hypothetical language JC used in the example system `LineDrawings` shown below:

File Figures/SimpleFig.jc:

```
package Figures;

class Point {
   public int X, Y;
   public Point(int A, int B) {
     X = A; Y = B;
  }
    public void show() {
      //  ... implementation omitted
  }
}

public class Line {
  public Point P1, P2;
  public Line(int A, int B, int C, int D) {
    P1 = new Point(A, B);
    P2 = new Point(C, D);
```

```
  }
  public void draw() {
    Point P;
    for (P = P1; P != P2; P = nextPoint(P))
      P.show();
  }
  private Point nextPoint(Point P) {
    return new Point(P.X+1, P.Y+(P2.X-P1.X)/(P2.Y-P1.Y));
    }
}
```

File Program/Drawing.jc:

```
#includes "initgraph.h"
import Figures.SimpleFig;

void main(void) {
  fgraph();
  Line L1 = new Line(2, 4, 7, 9);
  Line L2 = new Line(5, 8, 12, 18);
  L1.draw();
  L2.draw();
    }
```

The model of the LineDrawings system is depicted in Figure 3.

The code of the example system is organized in two folders, `Figures` and `Program`, represented in the model as *UpperUnits*. These folders contain the source code files `SimpleFigures` and `Drawing`, which are represented by the two *Units*.

The `SimpleFigures` unit contains two classes, `Point` and `Line`, represented as *ADTs* belonging to the `SimpleFigures` *Unit*.

The class `Point` has two fields, X and Y, represented as *Variable* objects of the meta-model. Both are of a primitive type and thus not further captured by our model. The class `Point` has a constructor `init()` and a member function `show()`, both of them represented as *Function* objects of the meta-model. The constructor accesses both fields, represented by the *access* relationships. The `init()` function takes two parameters, which are of a primitive type and thus they introduce no *isOfType* relationships in our model.

The class `Line` has two fields, P1 and P2, represented in the model as Variables that are part of the `Line` *ADT*. These variables are of the type `Point` that has been already represented as the *ADT* `Point` as described above. This fact is reflected in the *isOfType* relationships. Class `Line` has a constructor `init()` and member functions `draw()` and `nextPoint()`. All functions access both fields, shown in the *access* relationships between them. The constructor of `Line` *calls* the contructor of `Point`. The function `draw()` has a local variable P which is of type `Point`, this is represented in the model by a *LocalVar* which *isPartOf* the *Function* `draw`. The *Function* `draw` *calls* functions `show()` defined in *ADT* `Point` and `nextPoint` defined in *ADT* `Line`.

The `Drawing` file defines no classes. According to the UNIQ-ART modeling conventions, in the model, the `Drawing` *Unit* contains a single `default` *ADT* which

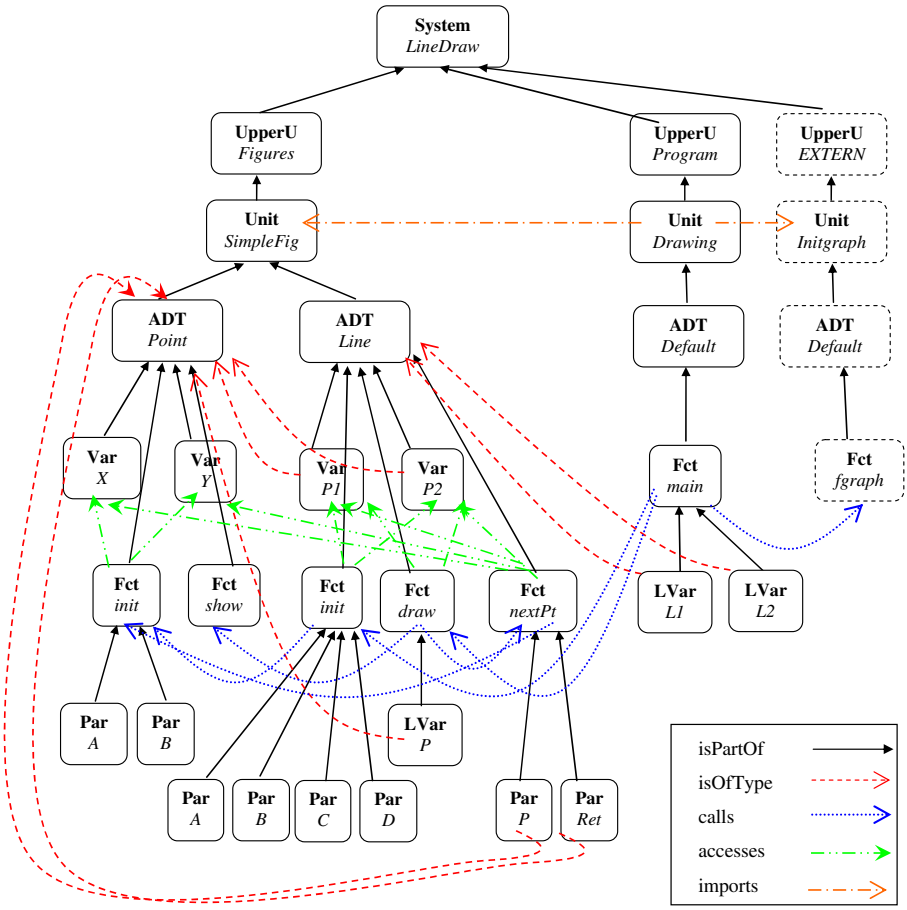


Fig. 3. Example of a model

contains only one global function, `main()`. The function `main()` *calls* the constructor of *ADT* `Line` and *calls* their `draw()` function.

The Function `main()` contains two *LocalVar* parts, each of them associated with the *ADT* `Line` through an *isOfType* relationship.

The function `main()` of the `Drawing` Unit also *calls* function `fgraph()` from an external library `Initgraph` (which is not part of the system under analysis, which is the system LineDraw). The Unit `Initgraph` has been not included in the code to be modeled thus it is considered external to the project under analysis, having only a partial model. We know only one function, `fgraph()`, out of all the elements belonging to this Unit, and only because it is called by a function that is part of our system.

This choosen example covered many of the features of UNIQ-ART. One important feature of UNIQ-ART which has not been covered by this example is the *extends* relationship which is used to model class inheritance, interface implementation or own header file inclusion. Also, another feature of UNIQ-ART not illustrated by this

example is modeling simple type definitions which are not ADTs (`typedef structs` in C for example).

4 Implementing and Using UNIQ-ART

4.1 Representing the UNIQ-ART Meta-Model

Models and their meta-models can be represented and stored in different ways: relational databases and their database schema, XML files and their schema, logical fact bases and their predicates, graphs. UNIQ-ART may support all these different storage technologies and formats through an adequate storage technology mapper (see Figure 1).

In our current implementation of ART, we have choosen to represent models through relational databases (MySQL) with an adequate schema of tables and relationships. Such a platform allows us to integrate model extractor tools from different languages and platforms. However, if interaction with other tools would require it so, a UNIQ-ART model can be easily mapped to other means of representation without loosing generality, for example a GXL [4] description could be easily generated starting from the contents of a database or directly by adapted model extractor tools.

Tables corresponds to each type of *ProgramPart*s (*Systems*, *UpperUnits*, *Units*, *ADTs*, *Functions*, *Parameters*, *LocalVars*, *Variables*, *TypeDefs*). The *isPartOf* relationship, which is defined for each structural element, is represented by foreign key aggregation (n:1 relationship). The *isOfType* relationship is defined only for *Variables*, *LocalVars* and *Parameters*. In these tables, there is first a discriminant between the cases whether it is a primitive type, an *ADT*, or a *Typedef* and, in the case of not primitives, there is a foreign key association with a row in the corresponding table, ADTs or TypeDefs (n:1). The other relationships - *extends*, *calls*, *accesses* - are of cardinality n:m and are represented by association tables.

4.2 Model-Extractor Tools

Model extractor tools for Java, C# and ANSI C are implemented in the ART toolsuite. Each model extractor is a completely independent tool, implemented using its own specific language, framework or technologies. Moreover, two of our model extractor tools work on compiled code (Java bytecode and Microsoft CIL) and can extract all the information they need.

The Java model extractor tool works on bytecode, processed with help of the ASM Java bytecode analysis framework (http://asm.ow2.org/).

Another model extractor tool works on managed .NET code, being able to handle code coming from any of the .NET programming languages (i.e. C#, VB, etc.) that are compiled into managed code (compiled into CIL - Common Intermediate Language). CIL is an object-oriented assembly language, and its object-oriented concepts are mapped onto the concepts of our meta-model. The model extractor tool uses Lutz Roeders Reflector for .NET for inspecting the CIL code (http://www.lutzroeder.com/dotnet/).

The model extractor tool for C works on source code. Our implementation runs first srcML [9] as a preprocessor in order to obtain the source code represented as a XML

file which can be easier parsed. The procedural concepts of C are mapped into concepts of our meta-model. We recall here only the more specific issues: Files are the Units of the model as well as the ADTs (each unit contains by default one logical unit); The relationship between two units, one that contains declarations of elements which are defined in the other unit, is equivalent with extending an abstract class.

All model extractor tools populate the tables of the same MySQL database with data entries. For each system that will be analyzed, the primary model is extracted only once, all the analysis tasks are performed by starting from the model stored in the database.

The model extractor tools have been used on a large variety of systems, implemented in Java, C# and C, ranging from small applications developed as university projects until popular applications available as open-source or in compiled form, until, as the benchmark for the scalability of the proposed approach, the entire Java runtime (rt.jar). The execution times needed for the extraction of the primary model and its storage in the database are, for an average-sized system of about 1000 classes, in the range of one minute. Taking into account that ART is a toolsuite dedicated to off-line analysis, like architectural reconstruction, these times are very reasonable once-for-a-system times. The extraction of the model and its storage in the database for a very large system (the Java runtime, having 20000 classes) took 24 minutes.

4.3 Applications Using UNIQ-ART

A primary dependency structure model stored in the database could be used directly by writing SQL queries. For example, such queries could retrieve the *ADT* containing the most functions, or retrieve the *Function* which is called by most other functions, etc.

Another way of using a primary model is with help of dedicated analysis tools, or deriving more specialized secondary models and using specialized tools to further analyze these. This corresponds to the scenario that is most characteristic to the ART toolsuite as it has been depicted in Figure 1.

Some of the most frequent used secondary models in ART are different types of DSMs (dependency structure matrixs). The primary model supports building different types of specialized DSM's by choosing which of the program elements are exposed as elements of the DSM. Each kind of DSM generates a different view of the system and can have different uses. The two most frequent used DSM's in ART are the *external DSM*(between logical units), and the *internal DSM* (between elements belonging to the same logical unit).

In case of the *external DSM*, the *ADTs* of the primary model are exposed as the elements(rows and columns) of the DSM. The DSM records as the relationship between its elements at column i and row j the composition of all interactions of elements related to ADT_i and ADT_j. In this composition of interactions can be summed up all or some of the following: Variables contained in ADT_i which are of a type defined in ADT_j, Functions contained in ADT_i that contain LocalVars or Parameters of a type defined in ADT_j, the number of Functions contained in ADT_j which are called by Functions contained in ADT_j, the number of functions which call functions from both of ADT_i and ADT_j, the fact that ADT_i extends ADT_j, etc. Also, this relationship can be quantified, the strength of the relationship results by applying a set of different (empirical) weights when composing interactions of the aforementioned kinds. Such a

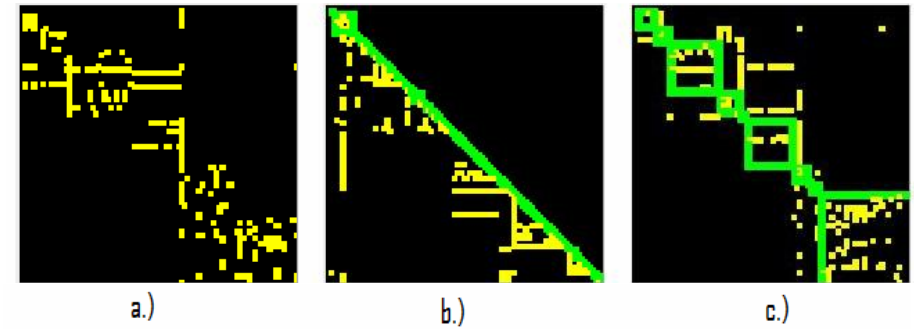


Fig. 4. Screenshots of ART. a.) Initial DSM b.) Layered DSM c.) Clustered DSM.

DSM is further analyzed by clustering for architectural reconstruction, by partitioning for identifying architectural layers, for detecting cycles among subsystems, for impact analysis, modularity analysis, etc. Some of our results were described in [3], where we improve clustering algorithms for software architecture reconstruction approaches starting from code. Subsystems are reconstructed by grouping together cohesive components and minimizing coupling between different subsystems.

In Figure 4 there are screenshots of ART representing an unordered DSM (case a.)), the DSM ordered as consequence of partitioning (case b.) and the DSM ordered as consequence of clustering (case c.).

In case of the *internal DSM* of a *ADT*, the program parts contained in the ADT become the elements (rows and columns) of the DSM. The rows and columns of the DSM correspond to *Variables* and *Functions* of a ADT. The existence of a relationship between the elements at column$i$ and row j is given by composing different possible interactions: direct call or access relationships between elements i and j, the number of other functions that access or call both of the elements i and j, the number of other variables accessed by both of the elements i and j, the number of other functions called by both of the elements i and j, etc. Such a DSM is used for analyzing the cohesion of an unit, and it can be used for taking refactoring decisions of splitting large and uncohesive units.

The times needed to build a secondary model (a DSM) in memory, starting from the primary model data stored in the database, are approximatively 10 times smaller than the time needed initially for extracting the primary model.

5 Related Work

There is a lot of previous and ongoing work in the fields of reverse engineering dealing with subjects like modularity checking [10], layering [5], detection of cyclic dependencies [11], impact analysis [12], clustering [7]. They all have in common the fact that they build and use some dependency models which are conceptually similar with Dependency Structure Matrix [6]. Our work in building ART [3] is also in this domain.

One problem that has been noticed early in the field is that existing tools and approaches are difficult to reuse in another context and that existing tools are difficult to integrate in order to form tool-suites. In order to achieve this, two aspects have to

be covered: first, formats for data exchange should be unified, and second, common schema or meta-models for information representation must be adopted.

Tools can be adapted to use a common data format. In [4], an infrastructure to support interoperability between tools of reverse engineering assumes that software reengineering tools are graph based tools that can be composed if they use a common graph interchange format, GXL (the Graph eXchange Language). GXL was developed as a general format for describing graph structures [13]. But GXL does not prescribe a schema for software data. GXL provides a common syntax for exchanges and features for users to specify their own schema.

For example, in the case of the typical ART scenario depicted in Figure 1, the clustering tool can require that DSM is represented in GXL format. This will make the clustering tool more reusable on different data. However, the GXL syntax does not capture anything about the semantics of the DSM, whether it is an external DSM or an internal DSM, as it was discussed in Section 4.3.

Establishing only the common exchange format does not offer the needed support for extracting models of a similar semantics out of different kinds of system implementations. In order to fully achieve this, common schema or meta-models for information representation must be used. It is with regard to this aspect that we introduce UNIQ-ART.

Reference schema for certain standard applications in reverse engineering have been developed. For example, such meta-models for C/C++ at the low detail (abstract syntax tree) level are proposed in [14] (the Columbus schema).

Some more general schema for language-independent modeling, address the family of object oriented systems. Examples include the UML meta-model [15] and the FAMIX meta-model [16]. They present similarities between them, as UML can be considered a standard for object-oriented concepts.

Compared by complexity of the meta-model and level of details that it is able to capture, UNIQ-ART is most similar with FAMIX [16]. FAMIX is used by a wide variety of re-engineering tools comprising software metrics evaluation and software visualization. FAMIX provides for a language independent representation of object-oriented source code, thus its main concepts are *Class*, *Method*, *Attribute*. FAMIX represent relationships between these also as entities *InheritanceDefinition*, *Access* and *Invocation*. It does not treat in any way procedural aspects (global variables and functions, user-defined types which are no classes) although probably it could be extended to do so. We defined the main concepts of UNIQ-ART and their language mappings in such a way that they *all* apply *transparently* to both object-oriented and procedural language concepts, since it was a main goal of the ART project to develop architectural reconstruction tools applicable for both object oriented and procedural systems.

Another meta-model is the Dagstuhl Middle Metamodel DMM [17]. It can represent models of programs written in most common object-oriented and procedural languages. But DMM provides the modeling capabilities for object-oriented and non-object-oriented modeling in an *explicit* way. DMM generalizes several concepts to achieve multi-language transparency, but not programming paradigm transparency. For example, in DMM a *Method* is a concept which is different from *Routine* or *Function*, although they are very similar, except for the fact that a *Method* has a relationship to a

Class. This leads to an increased complexity of the DMM model, which contains a big number of *ModelObject* types and *Relationship* types.

In contrast, our model started from the key decision to abstract away as many differences between the object-oriented and the procedural programming paradigm. The semantics of the dependency is the primordial criteria in establishing the mapping between a concrete program element and a model element. For example, UNIQ-ART makes no difference between functions and methods calls, variable of field accesses. It is also possible that the same program element or relationship is mapped to different model elements or relationships, according to the context of its usage. For example, the fact of having a C module including a header file can be mapped to different types of model relationships: if the header file contains the prototypes of the including module, the relationship is semantically similar to that of a class implementing an interface or extending an abstract class, thus it is mapped in the model to a relationship of type extends; otherwise, the situation of including a file merely represents a simple dependency.

As described in Section 3, the UNIQ-ART meta-model contains only nine different types of *ProgramParts* and six different types of relationships, all of them applying transparently to both object-oriented and procedural concepts. Since it operates with a small number of program part types, UNIQ-ART is lightweight and thus easy to use; however, it can be used by a number of different kinds of applications (architectural reconstruction of subsystems, identification of architectural layers, detection of cyclic dependencies, impact analysis, modularity analysis, refactoring of uncohesive modules by splitting, etc), which proves its modeling power and utility.

6 Conclusions

The UNIQ-ART meta-model defines a unitary way of representing primary dependency structures of programs written in object-oriented as well as procedural languages. All model entities introduced by UNIQ-ART are abstractions which hide the differences between object oriented and procedural concepts. The particularity of UNIQ-ART is that it identifies the similarities, from the point of view of the dependency semantics, between certain program parts of object oriented languages and procedural languages, and maps them to a common representation. This is different from other meta-modeling approaches which aggregate concepts of different programming languages and introduce distinct modeling concepts for each of them. We have implemented language mappings and model extractor tools for Java, C# (MS CIL) and ANSI C. The utility of the proposed meta-model has been shown by a number of different reverse engineering and analysis applications that use it successfully.

References

1. Chikofsky, E., Cross, J.H.I.: Reverse engineering and design recovery: a taxonomy. IEEE Software 7(1), 13–17 (1990)
2. Canfora, G., Di Penta, M.: New frontiers of reverse engineering. In: Future of Software Engineering, FOSE 2007, pp. 326–341 (May 2007)

3. Sora, I., Glodean, G., Gligor, M.: Software architecture reconstruction: An approach based on combining graph clustering and partitioning. In: 2010 International Joint Conference on Computational Cybernetics and Technical Informatics (ICCC-CONTI), pp. 259–264 (May 2010)
4. Kraft, N.A., Malloy, B.A., Power, J.F.: An infrastructure to support interoperability in reverse engineering. Information and Software Technology 49(3), 292–307 (2007); 12th Working Conference on Reverse Engineering
5. Sarkar, S., Maskeri, G., Ramachandran, S.: Discovery of architectural layers and measurement of layering violations in source code. J. Syst. Softw. 82, 1891–1905 (2009)
6. Sangal, N., Jordan, E., Sinha, V., Jackson, D.: Using dependency models to manage complex software architecture. In: Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications, OOPSLA 2005, pp. 167–176. ACM, New York (2005)
7. Mitchell, B.S., Mancoridis, S.: On the automatic modularization of software systems using the bunch tool. IEEE Transactions on Software Engineering 32, 193–208 (2006)
8. OMG: The metaobject facility specification (2011), `http://www.omg.org/mof/`
9. Maletic, J., Collard, M., Marcus, A.: Source code files as structured documents. In: Proceedings of 10th International Workshop on Program Comprehension, pp. 289–292 (2002)
10. Wong, S., Cai, Y., Kim, M., Dalton, M.: Detecting software modularity violations. In: Proceeding of the 33rd International Conference on Software Engineering, ICSE 2011, pp. 411–420. ACM, New York (2011)
11. Falleri, J.-R., Denier, S., Laval, J., Vismara, P., Ducasse, S.: Efficient retrieval and ranking of undesired package cycles in large software systems. In: Bishop, J., Vallecillo, A. (eds.) TOOLS 2011. LNCS, vol. 6705, pp. 260–275. Springer, Heidelberg (2011)
12. Wong, S., Cai, Y.: Predicting change impact from logical models. In: IEEE International Conference on Software Maintenance, pp. 467–470 (2009)
13. Winter, A.J., Kullbach, B., Riediger, V.: An overview of the GXL graph exchange language. In: Diehl, S. (ed.) Dagstuhl Seminar 2001. LNCS, vol. 2269, pp. 324–532. Springer, Heidelberg (2002)
14. Ferenc, R., Beszedes, A., Tarkiainen, M., Gyimothy, T.: Columbus - reverse engineering tool and schema for C++. In: Proceedings of International Conference on Software Maintenance, pp. 172–181 (2002)
15. OMG: The Unified Modelling Language (2011), `http://www.uml.org/`
16. Tichelaar, S., Ducasse, S., Demeyer, S., Nierstrasz, O.: A meta-model for language-independent refactoring. In: Proceedings of International Symposium on Principles of Software Evolution, pp. 154–164 (2000)
17. Lethbridge, T.C., Tichelaar, S., Ploedereder, E.: The Dagstuhl Middle Metamodel: A Schema For Reverse Engineering. Electronic Notes in Theoretical Computer Science 94, 7–18 (2004); Proceedings of the International Workshop on Meta-Models and Schemas for Reverse Engineering (ateM 2003)

Social Adaptation at Runtime

Raian Ali[1], Carlos Solis[2], Inah Omoronyia[3], Mazeiar Salehie[4], and Bashar Nuseibeh[4,5]

[1] Bournemouth University, U.K.
[2] Amazon
[3] University of Glasgow, U.K.
[4] Lero – University of Limerick, Ireland
[5] Open University, U.K.

Abstract. One of the main goals of software adaptation is that users get their dynamic requirements met efficiently and correctly. Adaptation is traditionally driven by changes in the system internally and its operational environment. An adaptive system has to monitor and analyse such changes and, if needed, switch to the right behaviour to meet its requirements. In this paper, we advocate another essential driver for adaptation which is the collective judgement of users on the different behaviours of a system. This judgement is based on the feedback iteratively collected from users at run-time. Users feedback should be related to their main interest which is the ability and quality of the system in reaching their requirements. We propose a novel approach to requirements-driven adaptation that gives the collective judgement of users, inferred from their individual feedback, a primary role in planning and guiding adaptation. We apply our approach on a case study and report on the results.

Keywords: Requirements-driven Adaptation, Requirements at Runtime, Social Adaptation.

1 Introduction

Self-adaptive systems are designed to autonomously monitor and respond to changes in the operational environment and the system own state [1]. Each change can trigger a different response to satisfy or maintain the satisfaction of certain design objectives [2]. Self-adaptive systems have basic design properties (called *self-**). Self-protection property means the ability to monitor security breaches and act to prevent or recover from their effects. Self-optimization means the ability to monitor the resources availability and act to enhance performance. Self-healing means the ability to monitor faults that have occurred or could occur and cure or prevent their effects. Finally, self-configuration means the ability to monitor changes related to all of the other properties and add, alter, or drop upon certain software entities [3].

Self-adaptivity is highly reliant on the system ability to autonomously monitor the drivers of adaptation (security breaches, the available resources, faults and errors, etc.). In [4], we argued that there are drivers which are unmonitorable by relying on solely automated means. The collective judgement of users on the validity and quality of each of the alternative behaviours a system supports, is an example of that. Such judgement is a primary driver for adaptation to decide upon what alternative is socially judged to

L.A. Maciaszek and J. Filipe (Eds.): ENASE 2012, CCIS 410, pp. 110–127, 2013.

be the best to meet users' requirements. In such settings, the feedback of individual users is the main ingredient to obtain and process targeting for a social and collective planing of adaptation. We define *Social Adaptation* as the system autonomous ability to obtain and analyse users' feedback and choose upon an alternative system behavior which is collectively shown to be the best for meeting requirements.

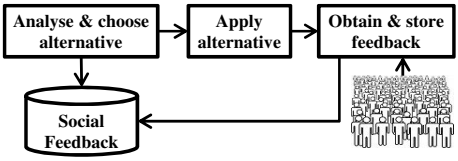
Social feedback can support a feasible and correct systems adaptation. This is particularly true for high-variability systems which incorporate a large number of alternatives. For such systems, a large number of users will be required to validate all of the system alternatives. Establishing such validation as a design time activity which is directed by designers (as often done in usability testing and user-centred design [5,6]) is highly expensive, time-consuming and hardly manageable. We define "social adaptation" as a kind of software adaptation that allows users to provide feedback at runtime and use it to choose the behaviour collectively shown to be the best for each different context. By involving users, i.e., by crowd-sourcing validation, social adaptation helps to evaluate and adapt high-variability systems rapidly and effectively.

Social adaptation advocates the repetitive collection and analysis of users feedback to keep adaptation up-to-date. Due to the dynamic nature of the world, users' judgement on the system could change over time. Thus, a traditional one step design-time system validation and customization might lead to judgements which are correct but only temporarily. For example, users' who currently like interaction via touch screens, might dislike it in the future when a better interaction technology is devised. Users' evaluation of a system alternative is not static and what is shown to be valid at certain stage of validation may become eventually invalid. Thus, the repetitive nature of social adaptation allows to accommodate the volatile nature of users judgement on a system.

Requirements are the natural subject of social feedback and, thus, social adaptation should be in the first place a requirements-driven adaptation. Upon the execution of a system alternative, users will be primarily concerned whether their requirements are met correctly. Users would also care about the degree of excellence of an execution against certain quality attributes. Consequently, social feedback concerns the validity and the quality of a system alternative as a way to meet users' requirement. This judgement is affectable by context which, thus, has to be monitored. The same system alternative could be judged differently when operating in different contexts. When a context C occurs, social adaptation analyses the feedback provided for each alternative when it was executed in C and choose the alternative which is collectively judged to suit C.

In this paper, we propose social adaptation which relies on the users' collective judgement of system alternatives as a primary driver for adaptation. We discuss the foundations and motivation of social adaptation. We provide a conceptualization of the main artefacts needed in a requirements model for systems adopting social adaptation. We develop analysis techniques to process social feedback and choose upon the system alternative which is collectively shown to be the most appropriate to meet a requirement. We develop a prototype CASE tool which supports our modelling and analysing, and evaluate our approach by applying it on a socially-adaptive messenger targeting to convey messages in the way collectively shown to best fit a certain context.

The paper is structured as follows. In Section 2 we define and motivate social adaptation and discuss the key differences between it and self-adaptation. In Section 3 we



Criteria	Self-Adaptation	Social Adaptation
Monitored	States and events	Users' judgments
Monitor	System	System and users
Target	System properties	Users' satisfaction
Decision Maker	Designers	Designers & users
Evolution	Designers	Designers & users

Fig. 1. Social Adaptation Loop (left), Social Adaptation vs. Self-Adaptation (right)

present the modelling artefacts of social adaptation from a requirements engineering perspective. In Section 4 we develop analysis algorithms to process social feedback and adapt the system at runtime. In Section 5 we evaluate of our approach in practice and present our CASE tool. In Section 6 we discuss the related work and in Section 7 we present our conclusions and future work.

2 Social Adaptation: Definition, Motivations and Foundations

Social adaptation is a distinguished kind of adaptation which responds to the social collective judgement about the correctness and efficiency of a system in meeting users' requirements. Social adaptation treats social feedback as a primitive driver for adaptation. Social feedback allows a runtime continuous evaluation of each of the alternative behaviours of a system. The alternative which is socially proved to be correct and more efficient will be the one to apply. In Fig.1 (left), we outline the social adaptation loop where the system analyses social feedback and decide upon which alternative behaviour to apply, applies it, and finally gets and stores the feedback of the users of that operation. We now discuss the key characteristic of social adaptation in comparison to traditional self-adaptation which is surveyed in [2] and we summarized that in Fig.1 (right):

- *The monitored* which represents the information to be monitored when deciding a suitable adaptation action. In self-adaptation, the system has to monitor changes in its internal and operational environment. Monitored events and states could indicate security breaches triggering a self-protecting action, or new settings of the available resources triggering a self-optimizing action, or faults triggering a self-healing action. In social adaptation, the system has to monitor the social judgement about its role in meeting requirements. Such judgements concern human opinions and conclusions rather than events and states in the system environment.
- *The monitor* which represents the entity that is responsible and capable of capturing the information which drives adaptation. In self-adaptation, the system enjoys an autonomous ability to monitor all information which is necessary for taking adaptation decisions which means that the monitor is a fully automated component [7]. Social adaptation, on the other hand, requires capturing its own kind of drivers, the users' judgement, which is un-monitorable by relying on solely automated means [4]. Social adaptation requires a socio-technical monitor which involves users' perception as an integral part of the system computation.
- *The adaptation target* which stands for the subject and the goal of adaptation. Self-adaptation provides a dynamic machinery to ensure that certain system properties

are maintained when changes happen. That is, the subject of adaptation is the system itself and the goal is to guarantee certain properties by preventing and acting against possible security breaches, faults, low performance, etc. On the other hand, social adaptation provides machinery to ensure that the system responds to the social collective judgement of each of its alternative behaviours Thus, the subject of social adaptation is the users' judgement of a system and its goal is to maximize the positive judgement about its role in meeting users' requirements.

- *The decision maker* which stands for the entity that takes adaptation decisions. In self-adaptation, decisions are planned by designers at design time so that adaptation is deterministic [8]. The system at runtime will monitor changes and apply a rationale developed by the designers and take an appropriate decision. In social adaptation, the decision makers are multiple. Designers plan adaptation at design time and leave the adaptation decision to be taken collaboratively by both users and systems. Users will interact with the system to provide their feedback and the system will analyse such feedback and adapt according to the rationale provided by the designers. Thus, adaptation is not determined by only designers' but it is also subject to users' collective decisions.
- *Evolution* which refers to the eventual changes in the adaptation rationale to keep adaptation up-to-date. Self-adaptive systems follow an evolution rationale, which is planned at design time, until the system apparently fails in taking correct adaptation decisions. When this happens, an evolution has to take place. To this end, the designers need to alter the system and make an evolved version able to cope with the new situation. Besides the possibility of being subject to a planned design time evolution, social adaptation incorporates runtime evolution driven by the users. A socially-adaptive system responds to the users' feedback which is itself likely to evolve over time, and thus the adaptation rationale itself evolves. In other words, the evolution of the adaptation rationale is driven by the evolution of users judgement about each system alternative with respect to meeting their requirements.

As a running example to illustrate the rest of this paper, we consider the case of a gallery-guide system designed to support the visitors of an art gallery. The system has two alternatives to convey information about art pieces to the visitors. *PDA-based system alternative*: by which the system will use the visitor's PDA to explain, possibly interactively, about an art piece. *Staff-supported system alternative*: by which the system will try to find a mentor to meet with the visitor and explain the art piece and also facilitate the meeting. The designers can not be fully certain of which alternative (PDA-based, Staff-supported) is more appropriate than the other and in which context. As a solution, the designers would need to build the system to obtain and benefit from the visitors feedback for taking such decision dynamically at runtime.

Users can judge and give feedback whether a system alternative leads to reach their requirements and its quality. We classify social feedback into 2 kinds:

- *System validity* which concerns whether a certain applied system alternative succeeds in meeting a certain requirement. Users are able to give feedback using requirements-related terms rather than software-related ones. For example, upon executing the PDA-based system alternative, a gallery visitor would only give a

Boolean answer saying "I could (not) ask for and get information". The visitors cannot generally explain how the complexity or simplicity of the HCI design prevented or facilitated the role of the system in the process of asking for and getting information.
- *System quality* which concerns the degree of excellence of a system alternative. For example, a visitor who had to wait for a long time to meet a mentor, would admit the delivery of information, i.e. the requirement is reached, but would probably give a negative assessment of some quality attributes like "comfort" and "quick".

We promote the social collective judgement on each system alternative, which relies on individual users' feedback, as a primitive irreplaceable driver for adaptation. Here we list three main reasons for that:

- *Un-monitorability of Judgements*. A user's judgement of validity and quality of a system alternative is a human opinion which is not always obtainable by relying solely on automated means and requires users to communicate it explicitly to the system [4]. For example, while the system can monitor if a requirement like "visitor is out of the gallery area when the gallery is to be closed" is met by sensing the visitor' location and movement, certain requirements cannot be monitored unless the system asks the users. The validity of the PDA-based system alternative with respect to meeting the requirement "visitor can ask for and get information" requires monitoring the visitor's judgement which is only possible if the user disclose it.
- *Repetitive Adaptation*. Certainty is not always achievable when designing a system. Moreover, certainty is not a static property. Thus, a one-stage validation of a system against its requirements may lead to a certainty at only the time of validation. When time passes, what was proved to be valid at the validation time may become invalid and vice versa. Consequently, we need a repetitive validation and adaptation to ensure up-to-date fitness between the system and the requirements it is meant to meet. Moreover, planned iterative design-time validation would be inadequate as changes that influence the validity and quality of the system are inherently unpredictable. The solution is to enable this process as a repetitive automated activity at runtime. Users can give their feedback and adaptation can take place while the system is operating. For example, at an early stage of the system life, the feedback of users in a certain environment could indicate that PDA-based system alternative is not valid and that it has a poor quality. When time passes and visitors become more familiar with the use of PDAs, it might become a valid and good quality alternative. In the future and when a new communication technology is devised, this system alternative may turn back to be inappropriate.
- *High-variability systems adaptation*. Variability, the existence of various system alternatives, is the cornerstone for adaptivity. Adapting highly-variable systems, which incorporate a large number of alternatives, necessitates asking the feedback of a large number of users. In traditional usability testing, user experience, and user-centric design, capturing users' feedback is usually performed at design time and involves a limited number of users. Applying these techniques on high-variability systems would not be feasible due to the number of users and cases and the amount of time needed to cover all possible alternatives and context variations. As a solution, social adaptation crowd-sources users at runtime to give feedback about

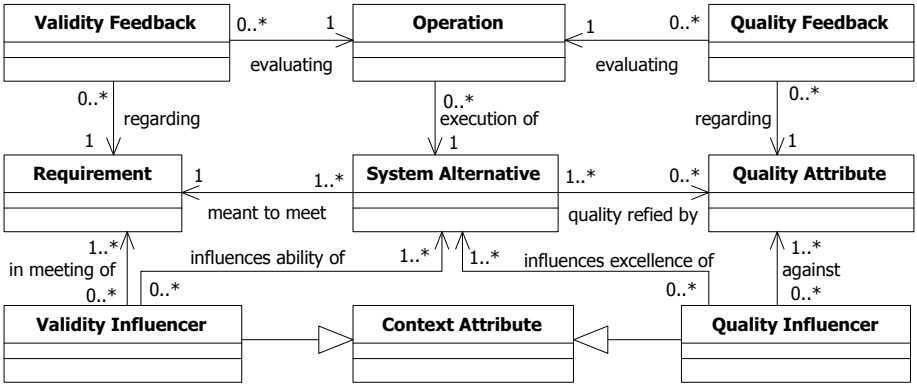


Fig. 2. Requirements modelling for Social Adaptation

validity and quality of each executed system alternative repetitively during the lifetime of software. For example, each of the alternatives of the gallery-guide system (PDA-based and Staff-supported) is just a high level description of a system alternative. Each of these two alternatives incorporates a large number of finer-grained functionalities and variations and context influences as well. To enable a continuous and open-to-the-crowd feedback, visitors should be asked for and enabled to give their feedback at runtime. Analysing and adapting to feedback autonomously at runtime supports a rapid and efficient adaptation.

3 Modelling Requirements for Social Adaptation

In this section, we explain a set of fundamental artefacts a requirements models should include when designing systems to enact social adaptation. Fig.2 shows the metamodel of these artefacts and Fig.3 shows an instance of it. It is worth pointing out that this model is meant to operate on the top of established requirements models which capture the relation between requirements and quality attributes and system alternatives. It is meant to extend such requirements models by the social feedback about this relation and the context influence on it. We will show an example of the application of our metamodel on a main-stream requirements model, the goal model, in Section 5.

The central horizontal part of the model captures the relation between the requirements and quality attributes on one side, and the designed system alternatives one the other. Requirement indicates an intention to reach a certain state of the world. System Alternative is a synthesis between human and automated activities designed to meet a requirement. A requirement could be reached via multiple system alternatives. Quality Attribute is a distinguished characteristic of the degree of excellence of a system alternative. In our framework, these three artefacts and the relations between them are specified by the designers at design time and are static and, thus, are not subject of monitoring at runtime. For examples, please see Fig.3.

Example 1. The statement "visitor can ask for and get information about any art piece" is a requirement statement that indicates a visitor's desire to reach a certain level of

Instance of the Model	
Requirement	R: visitor can ask for and get information about an art piece
System Alternatives	S_1: PDA-based (input handling, interactive presentation, video, etc.)
	S_2: Staff-supported (estimating time to meet, notifying staff, etc.)
Quality Attributes	Q_1: visitor is well-informed
	Q_2: visitor comfort
Validity Influencers	C_1: visitor' age, C_2: visitor' technology expertise level, influence the ability of S_1 to meet R
	C_3: estimated time for a staff to meet visitor, C_4: visitor has companions? Influence the ability of S_2 to meet R
Quality Influencers	C_1, C_2, and C_5: complexity of art piece information, influence the quality of S_1 with regards to Q_1
	C_1, C_2, and C_6: user movement status (moving, standing, sitting), influence the quality of S_1 with regards to Q_2
	C_7: staff expertise and C_8: staff ability to speak the visitor's native language, influence the quality of S_2 with regards to Q_1
	C_3 and C_9: existence of a free seat closed to the visitor's location, influence the quality of S_2 with regards to Q_2
Runtime Feedback Example	
Operations	$Operation_1$: execution of S_1. The values of its relevant context attributes are C_1= >65 years, C_2=low, C_5: low, C_6= standing.
	$Operation_2$: execution of S2. The values of its relevant context attributes are C3 = <5 min, C_4= alone, C_7= medium, C_8= fair, C_9= no
Validity Feedback	$Operation_1$.Validity_feedback= False (R is not reached)
	$Operation_2$.Validity _feedback= True (R is reached)
Quality Feedback	$Operation_1$. Quality_feedback is irrelevant (R was judged unreached)
	$Operation_2$.Quality_feedback(Q_1)= medium $Operation_2$.Quality_feedback(Q_2)= high

Fig. 3. An instance of the model of requirements artefacts for social adaptation shown in Fig.2

awareness and knowledge about an art piece in the gallery. To meet this requirement, different system alternatives can be designed (PDA-based, Staff-supported). The Staff-supported alternative has technical and social counterparts. The technical part includes enabling the visitor to specify a product via RFID technology, alerting the mentor via his PDA, guiding the mentor to meet visitor in certain meeting points. The social part includes that the mentor interacts with the visitor and explain to him about the art piece. "Visitor's comfort" while delivering information about an art piece is a quality attribute against it each of the system alternatives can be evaluated.

The lower part of the model stands for the context influence on the relation between the system alternatives on one side and the requirements and quality attributes on the other. Context Attribute is a distinguished characteristic of the environment within which the system operates. Validity Influencer is a context attribute that influences the ability of a system alternative to meet a requirement. Quality Influencer is a context attribute that influences the degree of excellence of a system alternative with respect to

a quality attribute. The context attributes, of both categories, are specified by designers at design time and monitored at runtime, i.e. the real values are obtained and stored at runtime. For examples, please see Fig.3.

Example 2. The context attributes "visitor's age" and "visitor's technology expertise level" are validity influencers which possibly affect the ability of PDA-based system alternative to enable visitor of querying and getting information. In other words, these attributes could influence whether a visitor finds PDA-based system alternative a valid means to meet his requirement of getting information about an art piece. The context attributes "the estimated time for a mentor to meet visitor", "the existence of a free seat close to the visitor's location" are context attributes that possibly affect the excellence of Staff-supported system alternative with respect to the quality attribute "visitor comfort".

The upper part of the model stands for the social feedback that express users' judgements about each operation of a system alternative. Operation is a single execution of a system alternative. Validity Feedback is a Boolean judgement given by a user concerning his evaluation whether an operation has led to meet his requirement. Quality Feedback is an assessment, reified by a numeric value, given by a user concerning his evaluation of an operation against a quality attribute. The social feedback, of both categories, is specified at design time by designers. The value of the feedback relevant to a specific system alternative is obtained from users and stored at runtime after an operation of that alternative is executed. For examples, please see Fig.3.

Example 3. An operation could start when a user specifies an art piece via RFID reader plugged in his PDA. Then the system chooses autonomously (based on analysing the historical feedback as we explain in Section 4) to follow a PDA-based system alternative and applies it. The correct execution of the functionalities this alternative incorporates (i.e. bug-free, no failure in network, etc.) does not mean that the requirement is reached. The visitor may not complete the interaction via his PDA or feel unable to interact with it, so the visitor will give a negative validity feedback. Now, let us suppose that the system followed a Staff-supported system alternative. After completing the operation, a visitor could state that he got the information (the validity feedback is positive) but it was a bit uncomfortable to wait for a mentor and thus the quality attribute "visitor comfort" could be evaluated to "medium".

In this work, and to enable the analysis which we propose in Section 4, we restrict the values of each context attribute to belong to an enumeration specified by the analyst. For example, the visitor age could be specified to be in {"<18", "between 18 and 25", "between 25 and 65", ">65"}. The validity feedback is already restricted to be a Boolean value. We also restrict the quality feedback value to be within a range of integers [0..n] where 0 stands for "the alternative has very bad quality against the quality attribute q" and n stands for "the alternative is excellent against the quality attribute q".

4 Social Adaptation Analysis

The main goal of obtaining social feedback is to support the system decision about the best alternative to apply for reaching users' requirements and overcome the designers' uncertainty about this decision cope with the changing trends of users over time. When the system has to meet a requirement, it has to choose an alternative to apply. The system

has to assess the collective judgement of each alternative of being a valid and a good-quality means to meet requirements. We propose to take into consideration different factors that together help for a holistic assessment of the collective judgement on the validity and quality of a system alternative. In what following, we discuss these factors taking examples of Fig.3:

- *Feedback Value*. This factor stands for the values given by the users when evaluating the validity and quality of each operation of a system alternative. Users provide the validity feedback by giving a Boolean answer reflecting their judgement whether the operation led to reach their requirements. Users evaluate the quality of each operation of a system alternative against each quality attribute by giving a value within a designated rank [0..n] where 0 means the lowest quality and n means the highest. These values are the basic factor in assessing the validity and the quality of a system alternative.
- *Feedback Relevance*. This factor stands for the meaningfulness of each of the users' validity and quality feedback when assessing a system alternative. This relevance will be interpreted as a weight for the feedback value which reifies the user' judgement of the validity and quality of a system alternative. We consider two sub-factors which influence the relevance of a feedback:
  - *Feedback Context*. This factor stands for the match between the context of a particular operation of a system alternative for which the feedback was given, and the current context where a decision about that alternative has to be taken. The validity of a system alternative and its quality against each quality attribute are affected by a set of context influencers as we explained earlier. The more the match between the values of these context influencers when the feedback was given and their values at the assessment time, the more relevant the feedback is. For example, suppose the system is assessing the validity of the PDA-based system alternative and that the current values for its validity influencers are (C_1: visitor' age) = "$>$ 65", (C_2: visitor's technology expertise level) = "low". Suppose we have two validity feedback F_1= "valid" and F_2= "invalid" and the values of contexts for F_1 and F_2 are C_1= "$>$65", C_2= "medium", and C_1= "$>$65", C_2= "low", respectively. Then the relevance of F_2 is higher than the relevance of F_1 because more context influencers match in F_2 than in F_1. Thus, and according to the feedback context factor, the alternative will be judged closer to "invalid" than "valid".
  - *Feedback Freshness*. This factor stands for the recentness of the feedback. Feedback relevance is proportional to its freshness. That is, the more recent the feedback is, the more meaningful. There could be several ways to compute the feedback freshness. One design decision could compute it by dividing its sequential number by the overall number of feedback of its kind. For example, suppose that PDA-based system alternative got two validity feedback, the earlier (with a sequential number 1) F_1= "invalid" and the later (with a sequential number 2) F_2= "valid". Thus, and according to the feedback freshness factor, the alternative will be judged closer to "valid" than "invalid".
- *User's Preferences*. This factor stands for the preferences of the user while assessing the overall quality of a system alternative. The analysis of the social feedback results in giving an overall assessment of each system alternative against each

```
Algorithm: Assessing Validity
Input:     s: System alternative
           C: {c, c is a validity influencer of s}
           C.Values: {(c,v), c in C and v = c's monitored value}
Output:    assessed validity of (s,r)
1.  OP:= {o ∈ s.Operations, o got a validity feedback}
2.  If |C| = 0 then
3.      RETURN Avg_Freshness (OP) *  (|{o in OP, o.validity_feedback= 'valid'}| / |OP|)
4.  Else
5.      relevant_validity_sum:= 0
6.      relevance_sum:= 0
7.      For i = 1 to |C| Do
8.        OP_Ci := Ø
9.        For each Ci ∈ 2^C and |Ci|= i
10.         OP_Ci = OP_Ci U {o in Op; ExactMatch(o.Ci.Values, C.Values)}
11.       EndFor
12.       validity_Ci:= |{o in OP_Ci; o.validity_feedback= 'valid'}| / | OP.Ci|
13.       relevance_Ci   := (i / |C| + Avg_Freshness(OP.Ci))/2
14.       relevant_validity_Ci := relevance_Ci   * validity_Ci
15.       relevant_validty_sum:= relevant_validity_sum + relevant_validity_Ci
16.       relevance_sum:= relevance_sum + relevance_Ci
17.     EndFor
18.     RETURN relevant_validity_sum/relevance_sum
19. EndIf
```

Fig. 4. Assessing Validity Algorithm

quality attribute. However, the assessment of the overall quality, i.e., the aggregated quality, of a system alternative may consider how the user appreciates each of these quality attributes. Similarly to the work in [9], we allow users to express their preferences by ranking the degree of importance of each of the quality attributes. For example, suppose that by analysing the historical quality feedback, the PDA-based alternative quality against Q_1= "visitor is well-informed" was assessed to 3 and against Q_2 = "visitor's comfort" was assessed to 2. Suppose that the Staff-supported alternative quality against Q_1 was assessed to 2 and against Q_2 to 3. If a user appreciates Q_1 more than Q_2 then PDA-based alternative overall quality will be assessed higher than that of the Staff-supported system alternative, and vice versa.

The algorithm Assessing Validity (Fig.4) computes the collective validity judgement of a system alternative based on the validity feedback users have provided in the past. It takes as input a system alternative s, the set of validity influencers C which affect the ability of s to meet the requirement it is designed to reach and the actual values of these influencers at the time of assessment $C.Values$. It gives as output an assessment of the validity of the statement *"s is a valid means to meet the requirement it is designed for"*. First, the algorithm identifies the operations OP of the system alternative s which got validity feedback from users (Line 1). If the system alternative has no validity influencers then the *context match* factor is irrelevant and the algorithm returns simply the

proportion of valid operations over the overall number of operations $\mid OP \mid$ multiplied (weighted) by the average freshness of the operations set OP (Lines 2-3).

When the system alternative has validity influencers, the algorithm iterates for each possible partial or complete match of the context validity influencers at the feedback time and the assessment time (Lines 7-17). For each combination of validity influencers C_i with a cardinality i, the algorithm identifies the set of operations OP_C_i whose validity influencers values ($o.C_i.Values$) match with the validity influencers values at the assessment time ($C.Values$) (Lines 9-11). The algorithm then computes the validity probability concerning the context matches of the cardinality i by dividing the number of valid operation of $Op.C_i$ by $\mid Op.C_i \mid$ (Line 12). The relevance of this probability is decided by both the cardinality of context match (i/$\mid C \mid$) and the value of the freshness factor (Avg_Freshness($Op.C_i$)), computed as we explained earlier (Line 13). The algorithm then multiplies the relevance with the computed validity to get the relevant (i.e., the weighted) validity (Line14). The algorithm then (Lines 15-16) accumulates the relevance and the relevant validity into the variables $relevant\_validity\_sum$ and $relevance\_sum$ (initiated at Lines 5-6). After the iteration goes through all partial and complete context matches, the algorithm gives the overall assessment by dividing the $relevant\_validity\_sum$ by the $relevance\_sum$ (Line 18).

Similarly to this algorithm, we have developed the algorithm Assessing Quality to calculate the collective judgement of the quality of a system alternative against a quality attribute. The main difference with regards to Assessing Validity algorithm is the consideration of quality feedback of only the operations with positive validity feedback as negative validity feedback makes any quality feedback irrelevant. Moreover, Assessing Quality deals with the average value of the quality feedback provided for an alternative against a quality attribute. Assessing the overall quality of an alternative considers also the users' preferences expressed via ranking the importance of each quality attributes. For the limitation of space, we have not included neither this algorithm nor the examples which explains both algorithms. For details, please see our technical report [10].

5 In Practice

To evaluate our framework, we have organized a lab session and invited 5 researchers specialized in requirements engineering and their research is in the area of requirements-driven adaptive systems engineering. We have explained our design principles of modelling requirements for socially-adaptive systems. We then explained the scenario of an adaptive messenger system which is able to deliver messages in different ways and asked the participants to draw a goal model, namely Tropos goal model [11], presenting its requirements together with the validity influencers (on the decompositions and means-end) and the quality influencers on the contribution links between any goal/task and a set of softgoals which evaluates all alternatives (please refer to our report in [10] for more details). The left part of Fig.5 shows a goal model built during the session; in the other part we extract one alternative and show its validity and quality influencers.

The main issues which were raised by the participants concerned the following. *Context monitorability*: besides the validity and quality judgement, the values of certain context attributes might not be monitorable by relying solely on automated means and

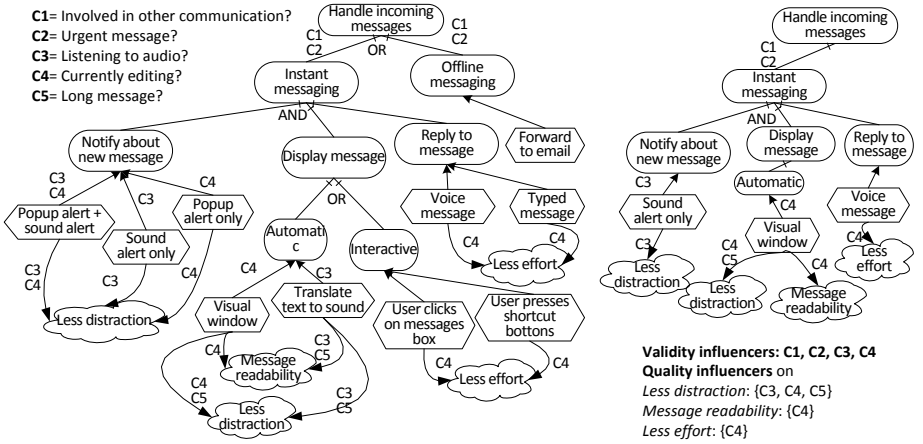


Fig. 5. A Goal model (left) and an alternative of it with quality and validity influencers (right)

may require users to act as monitors. *Quality and context attributes identification*: users should be given the chance to define extra quality and context attributes which were not considered by the designers. *Context influence (un)certainty*: the design should also consider that some context influences are already known and our approach should accommodate both certain and uncertain context influences. *Runtime evolution of the requirements model*: the analysis should be enriched to also decide parts of the model which should be removed when collectively proved to be invalid or having very low quality and also help the analyst by indicating loci in the model where an evolution should take place. *Feedback relevance*: we need mechanisms to exclude feedback which are not significant or inconsistent taking into consideration several factors like the user' history and pattern of use. We need to handle these limitations of our approach to maximize its applicability and operations and broaden and optimize its automated analysis.

To evaluate the social adaptation analysis proposed in Section 4, we have asked 15 users to provide validity and quality feedback about 6 alternatives of a prototype messenger in 3 different contexts. For the quality feedback we have considered 2 quality attributes (Q_1: less distraction, Q_2: less effort). Then we have run the validity and the quality assessment analysis for the different alternatives in the different contexts taking as input the set of obtained users' feedback. Then, we have surveyed a set of 3 other users (testing users) and compared their feedback to the collective judgements obtained by running the validity and quality analysis algorithms.

To evaluate the validity assessment analysis, we have considered 3 alternatives and 3 different contexts where the validity assessment algorithm has given high probability for validity ($>$85 %). We have asked the 3 users to provide their validity feedback (thus 27 feedback were provided in total) and out of which 23 feedback had the value "valid" and 4 had the value "invalid". This shows a good match between the collective validity judgement of the 15 initial users, computed by the validity assessment algorithm, and the judgement of each of the testing users. It is worth pointing out that the consensus of users about the validity is more likely to be achieved than the quality due to the nature of the decision about validity which has a clear-cut criteria to be judged upon.

Table 1. For each context, the rank of the different alternatives (mismatches are in bold font)

Q_1 :	C_1				C_2				C_3			
LD	Sys	U_1	U_2	U_3	Sys	U_1	U_2	U_3	Sys	U_1	U_2	U_3
A_1	2	2	2	2	2	2	**1**	2	2	2	2	2
A_2	1	1	1	1	1	1	**2**	1	1	1	**3**	**3**
A_3	3	3	3	3	3	3	3	3	3	3	**1**	**1**

To evaluate the quality assessment analysis, we have considered 3 other alternatives and 3 different contexts and asked the 3 testing users to provide the quality feedback of each alternative in each of the contexts against Q_1 and Q_2. Table 1, show the ranking of the automated analysis assessment and the testing users assessment of the 3 different messenger system alternatives (A_1, A_2, A_3) against the quality attribute Q_1: "Less Distraction" (LD), in 3 different contexts (C_1, C_2, C_3). The acronym "Sys" stands for the assessment given by the algorithm Assess Quality and U_i stand for each testing user. For example, and taking the first data column, the automated analysis of the users' feedback to obtain the quality assessment of A_1, A_2, and A_3 alternatives against the quality attribute "Less Distraction" (LD) within the context C_1 indicated that A_2 has the best quality, and A_1 has the second and A_3 has the lowest quality. The ranking the automated analysis gave in this case, matched the ranking made based on the quality assessment each testing user gave. In the context C_2, the user U_2 gave a ranking different from the one given by the automated analysis and we highlight the mismatching results. The same for U_2 and U_3 for the context C_3. As shown in the table, the matching between the collective quality judgement computed by the quality assessment algorithm and the testing users feedback was good enough (21 out of 27). For the quality attribute Q_2= "Less Effort", the number of matches between the automated collective judgement and testing users was also good (18 matches out of 27 comparisons).

There are several threats to validity concerning our evaluation. The first threat concerns the small size scenario which we used (the messenger system) and the relatively small number of participants who gave feedback. Larger scale experiment would maximize the credibility of our results. The second is the kind of practitioners who modelled the scenario who already had a good expertise in requirements modelling and self-adaptive systems. Communicating our principles and guidelines to novice practitioners might raise other concerns related to the practitioners understandability and acceptability of our framework. The third is the relatively short period of time for getting feedback which makes it hard to evaluate certain things such as the evolution of social trends. The fourth is that our users committed to provide feedback which might not be the case with real users. Thus, we still need to study users' acceptance of our approach.

We have implemented a tool for specifying goal models which conform with our metamodel of Fig.2. The goal of our tool is to support designers in modelling requirements for social adaptation proposed in Sec. 3 and performing the analysis proposed in Sec. 4. The metamodel is defined using the Eclipse Modelling Framework (EMF)[1], which permits to define metamodels using the Ecore modelling language. EMF also generates tree model editors and Java source code for representing, persisting and manipulating those models. Using the EMF model editor, we can define a goal model

[1] `http://eclipse.org/modelling/emf/`

with any number of alternatives, quality attributes, quality influencers, and validity influencers. In the model editor we can specify which validity and quality influencers affect a given alternative. Our metamodel representation also includes feedback classes; therefore an alternative can contain the feedback provided by users of the system. Using the EMF generated source code, we have implemented our proposed algorithm for assessing the collective validity and quality of a goal model alternative. More details are in [10]. This implementation can take a goal model and can compute the validity and quality of the variants according to the provided user feedback. We have also used our tool for computing the quality and validity of the models used in the evaluation of the algorithm.

6 Related Work

There are several research areas highly related to Social Adaptation. We are mainly concerned about addressing challenges in the research in software-engineering for adaptive systems and, particularly, requirements-driven adaptation. Our proposed approach enriches requirement-driven adaptation research by a systematic approach to incorporate users' collective judgement of systems behaviours, perceived as means to meet requirements, as a primary adaptation driver. This helps for a more holistic adaptation which overcomes the limitation of automated means to judge if requirements (functional and non-functional) are being met [4]. It also allows for reflecting the collective voice of users' rather than relying on designers' judgements which could be, or eventually become, invalid. This systematic approach is inline with an increasing trend to involve users in the adaptation loop [12]. We concretely position users as a feedback providers and specify such a feedback in terms of requirements and provide mechanisms to process it and assess the social collective judgement of system behaviours which ultimately represents the human role in the adaptation loop.

In their seminal work, Fickas and Feather [13] highlight the importance of requirements monitoring at runtime as a basic and essential step for planning and enacting adaptation. This is a fundamental principle of our approach. Cheng et al. note that in requirement models uncertainty have not been explicitly addressed in traditional requirements engineering [12]. We address uncertainty by involving users in evaluating the system alternatives against their capability to meet requirements so that certainty is achieved based on the perception of users regarding the actual operation. Souza et al. [14] note that the (partial) un-fulfilment of requirements triggers adaptation. They introduce awareness requirements to refer to success, failure, performance and other properties of software requirements (i.e. meta-requirements) and propose that the system should monitor changes in these properties and decide upon when and what adaptation should take place. We argued that certain information can not be monitored by the system and require an explicit intervention of the users' set via feedback and provided a systematic way to realize that.

Baresi et al. [15] propose FLAGS (Fuzzy Live Adaptive Goals for Self-adaptive systems) for requirements-driven adaptation at runtime. FLAGS extends goal models mainly with adaptive goals which incorporate countermeasures for adaptation. When goals are not achieved by the current course of execution, adaptation countermeasures

are triggered. This approach can be potentially integrated with ours so that when social collective judgement indicates some failures, a certain adaptation goals should be activated. Qureshi and Perini [16] emphasize on flexibility of requirements refinement and provide a method that supports the runtime refinement of requirements artefacts as a repetitive activity performed collaboratively between the users and the application itself. We plan to benefit from this approach to incorporate users in the modelling process at runtime and not only the judgement of the system different behaviours.

Bencomo et al. [17] advocate that adaptation is planned either in a pre-defined way at design time or via an evolvable and reflexive response to some monitored parameters at runtime. The gap between goals and the system has to be bridged so that the system adaptation is guided by goals and the adaptation correctness is judge by the fulfilment of goals (requirements reflection). In another work related to requirements reflection, Sawyer et al. [8] discuss that runtime representation of requirements model, synchronizing the model with the architecture, dealing with uncertainty, multiple objective decision making, and self-explanation are areas need to be considered in realizing a requirements-aware system. We supplement these works with users perception as part of the system monitor so that we empower adaptivity and dealing with design-time uncertainty.

In self-adaptation [1,2] and autonomic computing, software can monitor and analyse changes in its internal and operational environment and plan and execute an appropriate response. Besides the research on purely-automated adaptation, there is an increasing trend to involve users in the adaptation loop as we advocate in this paper. As we mentioned earlier, we present social adaptation to overcome self-adaptation limitations in monitoring and analysing another driver of adaptation which is the users' evaluation of the role of the system as a means for reaching the requirements. Moreover, social adaptation relies on the collective feedback provided by an open community of users so that adaptation is accelerated and its feasibility is increased.

Social adaptation is complementary to personalization and customizing software to individuals [9,18]. Personalization deals with various aspects of system design such as the user interfaces [19,20], information content [21,22], etc. Social adaptation tailors a system to the collective judgement of its users' community while personalization customizes a system to the characteristics of individuals. That is, social adaptation is about socializing a system instead of personalizing it. While socialization clearly does not replace personalization, it is essential when the system is highly variable and the individual users use the system for relatively limited number of times and the system validity and quality is subject to frequent changes. In such settings, customizing software would better aggregate the judgements made by users who used the system in the past and benefit of that to maximize the satisfactions of the current users.

Recent research has explored the role of context in requirements (e.g., [23], [24]) and the elicitation of contextual requirements [25]. In these works, the relationship between context and requirements are not evolvable, i.e., there is a certain degree of certainty when analysing context influence on requirements. Such assumption could be valid when dealing with well-known system scenarios where the relationship between the system and its environment is predictable at design time. Social adaptation serves

when there is uncertainty in specifying this relations. It makes it subject to validation and evolution driven by the users feedback about the quality and the validity of system different behaviours in different contexts. We have discussed social adaptation in the context of software product lines and proposed Social Software Product Lines that take users feedback on products quality as a main driver for runtime derivation/reconfiguration [26].

7 Conclusions and Future Work

Adaptivity is a demand to maintain the validity and quality of software over time. Adaptation is driven by certain categories of changes in the system internal state and its operational environment (security breaches, faults and errors, available resources, context, etc.). We advocated another essential driver for adaptation; the collective judgement of users on each behaviour of a system. We proposed social adaptation to refer to the continuous process of obtaining users feedback and analysing it to choose upon the behaviour which is collectively judged to best fit a certain context. The main ingredient to get the collective judgement is the feedback of individual users. Feedback reflects users' main interest which is the validity and quality of a behaviour in meeting requirements.

Social adaptation is meant to enrich self-adaptation by accommodating users perception as a part of the system computation and their collective judgement as a primary adaptation driver. Our inspiring principle is the wisdom-of-crowds [27] which keeps the door open for decisions formulated collectively by users. This helps us to overcome several limitations of purely-automated adaptation. Users judgement of the role of the system in meeting their requirements is essential for adaptation and it is unmonitorable by relying solely on automated means. Social adaptation relies on users collaboration to explicitly communicate it to the system. High-variability software makes it expensive and even infeasible to make validation as a design time activity performed in lab settings and directed by designers. Furthermore, and due to the dynamic nature of the world, validity is not a static property and should be iteratively repeated and the system should be altered accordingly. Social adaptation treats users as a system collaborators who act as continuous evaluators so that validation becomes feasible and remains up-to-date.

As a future work, we plan to enrich our conceptual modelling framework to capture constructs which help to better judge the relevance of a feedback like the pattern of use and the history of a user. Moreover, we need to devise techniques to involve users in collectively taking harder decisions at runtime such as altering the requirements model itself by adding (removing) requirements, context and quality attributes which are relevant (irrelevant). However, we always need to address the challenge of balancing between the users effort and computing transparency which is the essence of adaptive systems. We also need to work on incentives such as rewarding active users as a part of the social adaptation engineering. We need to further validate our approach on more complex systems and settings and improve our CASE tool to better facilitate social adaptation modelling and analysis.

Acknowledgements. This work was supported, in part, by Science Foundation Ireland grant 10/CE/I1855 to Lero - the Irish Software Engineering Research Centre (www.lero.ie), the EU Seventh Framework Programme (FP7/2007-2013) under grants no 258109 (FastFix) and European Research Council (ERC) Advanced Grant 291652 - ASAP.

References

1. Laddaga, R.: Self-adaptive software. Technical Report 98-12, DARPA BAA (1997)
2. Salehie, M., Tahvildari, L.: Self-adaptive software: Landscape and research challenges. ACM Transactions on Autonomous and Adaptive Systems 4, 14:1–14:42 (2009)
3. Murch, R.: Autonomic computing. IBM Press (2004)
4. Ali, R., Solis, C., Salehie, M., Omoronyia, I., Nuseibeh, B., Maalej, W.: Social sensing: when users become monitors. In: Proceedings of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE 2011, pp. 476–479. ACM (2011)
5. Dumas, J.S., Redish, J.C.: A Practical Guide to Usability Testing, 1st edn. Intellect Books, Exeter (1999)
6. Vredenberg, K., Isensee, S., Righi, C.: User-Centered Design: An Integrated Approach. Prentice Hall PTR (2001)
7. Maalej, W., Happel, H.J., Rashid, A.: When users become collaborators: towards continuous and context-aware user input. In: Proceeding of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications, OOPSLA 2009, pp. 981–990. ACM (2009)
8. Sawyer, P., Bencomo, N., Whittle, J., Letier, E., Finkelstein, A.: Requirements-aware systems: A research agenda for re for self-adaptive systems. In: Proceedings of the 2010 18th IEEE International Requirements Engineering Conference, RE 2010, pp. 95–103 (2010)
9. Hui, B., Liaskos, S., Mylopoulos, J.: Requirements analysis for customizable software goals-skills-preferences framework. In: Proceedings of the 11th IEEE International Conference on Requirements Engineering, pp. 117–126 (2003)
10. Ali, R., Solis, C., Omoronyia, I., Salehie, M., Nuseibeh, B.: Social adaptation: When software gives users a voice. Technical Report Lero-TR-2011-05, Lero. University of Limerick. Ireland (2011)
11. Bresciani, P., Perini, A., Giorgini, P., Giunchiglia, F., Mylopoulos, J.: Tropos: An agent-oriented software development methodology. Autonomous Agents and Multi-Agent Systems 8(3), 203–236 (2004)
12. Cheng, B.H.C., Giese, H., Inverardi, P., Magee, J., de Lemos, R.: Software engineering for self-adaptive systems: A research road map. In: Software Engineering for Self-Adaptive Systems, pp. 1–26 (2008)
13. Fickas, S., Feather, M.S.: Requirements monitoring in dynamic environments. In: Proceedings of the Second IEEE International Symposium on Requirements Engineering, RE 1995 (1995)
14. Silva Souza, V.E., Lapouchnian, A., Robinson, W.N., Mylopoulos, J.: Awareness requirements for adaptive systems. In: Proceeding of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2011, pp. 60–69. ACM (2011)
15. Baresi, L., Pasquale, L., Spoletini, P.: Fuzzy goals for requirements-driven adaptation. In: Proceedings of the 2010 18th IEEE International Requirements Engineering Conference, RE 2010, pp. 125–134 (2010)

16. Qureshi, N.A., Perini, A.: Requirements engineering for adaptive service based applications. In: Proceedings of the 2010 18th IEEE International Requirements Engineering Conference, RE 2010, pp. 108–111 (2010)
17. Bencomo, N., Whittle, J., Sawyer, P., Finkelstein, A., Letier, E.: Requirements reflection: requirements as runtime entities. In: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering, ICSE 2010, pp. 199–202. ACM, New York (2010)
18. Rahlff, O.W., Kenneth Rolfsen, R., Herstad, J.: Using personal traces in context space: Towards context trace technology. Personal Ubiquitous Comput 5, 50–53 (2001)
19. Weld, D.S., Anderson, C., Domingos, P., Etzioni, O., Gajos, K., Lau, T., Wolf, S.: Automatically personalizing user interfaces. In: International Joint Conference on Artificial Intelligence, pp. 1613–1619 (2003)
20. Schiaffino, S., Amandi, A.: User - interface agent interaction: personalization issues. Int. J. Hum.-Comput. Stud. 60, 129–148 (2004)
21. Teevan, J., Dumais, S.T., Horvitz, E.: Personalizing search via automated analysis of interests and activities. In: Proceedings of the 28th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, SIGIR 2005, pp. 449–456. ACM, New York (2005)
22. Chirita, P.A., Nejdl, W., Paiu, R., Kohlschütter, C.: Using odp metadata to personalize search. In: Proceedings of the 28th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, SIGIR 2005, pp. 178–185. ACM, New York (2005)
23. Ali, R., Chitchyan, R., Giorgini, P.: Context for goal-level product line derivation. In: The 3rd International Workshop on Dynamic Software Product Lines, DSPL 2009 (2009)
24. Ali, R., Dalpiaz, F., Giorgini, P.: A goal-based framework for contextual requirements modeling and analysis. Requir. Eng. 15, 439–458 (2010)
25. Seyff, N., Graf, F., Maiden, N.A.M., Grünbacher, P.: Scenarios in the wild: Experiences with a contextual requirements discovery method. In: Glinz, M., Heymans, P. (eds.) REFSQ 2009 Amsterdam. LNCS, vol. 5512, pp. 147–161. Springer, Heidelberg (2009)
26. Ali, R., Solis, C., Dalpiaz, F., Maalej, W., Giorgini, P., Nuseibeh, B.: Social software product lines. In: The 1st International Workshop on Requirements Engineering for Social Computing (RESC 2011), pp. 14–17. IEEE (2011)
27. Surowiecki, J.: The Wisdom of Crowds. Anchor (2005)

An Agent Oriented Development Process for Multimedia Systems*,**

Alma M. Gómez-Rodríguez, Juan Carlos González-Moreno, David Ramos-Valcárcel, and Francisco Javier Rodriguez-Martinez

Dept. de Informática.University of Vigo, Ed. Politécnico D-401,
Campus As Lagoas, Ourense E-32004, Spain
{alma,jcgmoreno,david,franjrm}@uvigo.es

Abstract. Requirements elicitation is a very difficult task when designing multimedia systems because of its special characteristics. The usual requirements specification process in multimedia development starts with the codification of prototype. After, it is analyzed by the customer to decide whether the application is to be constructed or not. This paper proposes a process specially defined for multimedia development that provides an initial prototype without human codification. The process combines the use of an agent oriented methodology and Alice (a multimedia oriented tool). In addition, a tool that provides support to this process is introduced. Both the process and its supporting tool are used in a simple example which demonstrates the suitability of the approach for multimedia prototype development.

1 Introduction

The modern devices, from computers to smartphones or tablets, introduce multimedia components in their applications [1], that has implied a growing relevance of such systems. Besides, multimedia developments are complex because they introduce multidisciplinary concepts from physic (GPS, gyroscope ...), mathematics, graphic desig, musics, scriptwriting, and software engineering [2].

Despite the importance of multimedia developments, there is not a public methodology or process devoted to multimedia development. Although there are multimedia private companies with productive development processes, they are not public because of the competitive advantage provided.

The special characteristics of multimedia systems make them suitable for applying the agent concept (for instance, games have characters which must show a certain degree of *intelligence* [3–5]). Agent Oriented Software Engineering (AOSE) has proof its capability for definition of complex distributed systems, where different agents cooperate to achieve certain goals. Moreover, a previous work [6] has shown the correspondence between AOSE concepts and multimedia concepts; for instance agent concept relates to game character, agent tasks to object functionalities, etc.

* Authors names are alphabetically ordered.

** This work has been supported by the project *Plataforma Tecnolóxica Intelixente de Xestión Avanzada para Produtores de Planta Ornamental de Galicia* with grant 10MRU007E of Xunta de Galicia.

L.A. Maciaszek and J. Filipe (Eds.): ENASE 2012, CCIS 410, pp. 128–141, 2013.

In AOSE, many methodologies have been proposed for systems development [7–10]. Among them, in this proposal, INGENIAS methodology has been selected attending to two basic reasons. The first one is that the different models and elements provided by the methodology were suitable for multimedia modeling, as it has been introduced in previous works [11, 12]. The second reason is that the methodology provides a tool that supports the development, called INGENIAS Development Kit (IDK) [13]. This tool automatically generates code from the system models, which can be used as an initial version of the system to be developed.

Multimedia systems are very intensive in animation, so the usual way of defining the requirements in such systems is to construct a storyboard or a prototype. Some previous works have addresed the idea of automatically generate that storyboard from a system description that uses Natural Language [14–16].

The approach presented in this paper is different, since it generates the animated storyboard from software engineering models. From these models, a particular tool, the Interactive StoryBoard Generator (ISBGen), obtains the system description using XML. This XML documents are used as inputs for Alice [17, 18], which visualizes these descriptions and provides them with 3D animation. Finally, the development process proposed and ISBGen is used in a simple multimedia development.

The remainder of the paper is organized as follows. Next sections introduce the technologies used in this work, as well as the relationships among them. Then, section 4 describes the ISBGen tool, which automatically generates the multimedia prototype. Section 5 approaches the development process for any multimedia system and applies it to a simple example showing the obtained results. The paper ends with the conclusions and future work.

2 Alice

Alice [17–19] is a programming environment which facilitates the creation of animations where a virtual world is populated with 3D objects (e.g., people, animals, vehicles ...). The tool is available freely and allows to create animated movies and simple video games. The tool has an interactive interface, where the user can drag and drop graphic tiles to create a program and allows to immediately see how it runs.

Alice has a huge library of predefined 3D objects. It includes static and dynamic objects which have a graphical aspect and, for dynamic ones, a predefined behavior. This characteristic makes it very suitable for making a prototype, because, in some cases, the developer only has to choose what object to include in the animation.

Other important advantage of Alice is that it follows an educative approach. This means that the tool is user-friendly, what makes easier to introduce changes in the animation. This characteristic is very interesting in this case, because it facilitates changes in the prototype and the feedback to INGENIAS models.

The tool uses Java and C# in its implementation and stores the definitions using XML documents. The data structures of both INGENIAS and Alice are based on XML, so that the integration is easy.

Moreover, a direct correspondence between the concepts used in the methodology and in Alice has been established in a previous work [6]. These conceptual relationships

constitute the formal basis for automatic generation of the storyboard. For instance, the agent concept of INGENIAS can be linked to Alice Object concept, or tasks of INGENIAS can be thought as Alice methods, etc.

3 INGENIAS and IDK

As it has been previously introduced, the methodology used in this proposal is INGENIAS and its supporting tool (IDK). INGENIAS [13] is a methodology created for the development of *Multi-Agent Systems* (MAS). Originally its purpose was the definition of a specific methodology for MAS development by integrating results from research in the area of agent technology, and from traditional Software Engineering Methodologies. The methodology covers analysis and design of MAS, and it is intended for general use, with no restrictions on application domain [20]. INGENIAS is based on the definition of a set of meta-models which introduce the elements to specify a MAS from five viewpoints: *agent* (definition, control and management of agent mental state), *interactions*, *organization*, *environment*, and *goals/tasks*. The meta-models proposed cover the whole life cycle and capture different views of the system. In the latest years, these meta-models have demonstrated their capability and maturity as supporting specification for the development of MAS [7].

The methodology can define either hardware or software agents, and has been applied in the creation of systems in several engineering fields, such as holonic manufacturing systems [21], multisensor surveillance systems [22], knowledge management systems [23], business workflow definition [24], simple games [7] and Product-Line Engineering environments [25].

From the very beginning, INGENIAS methodology provides a supporting CASE tool called IDK. This tool facilitates the construction of the different meta-models defined in the methodology and generates code from those definitions, or at least, code templates.

Recently the INGENIAS Agent Framework (IAF) has been proposed taking into account the experience in application of the methodology during several years enabling a full model driven development. This means that, following the guidelines of the IAF, an experienced developer can focus most of his/her effort in specifying the system, while the implementation is just a matter of automatically transforming the specification into code.

4 ISBGen

This paper presents the tool ISBGen, which provides a 3D interactive storyboard automatic generation from the elements of the meta-models defined using IDK. ISBGen is one of the contributions of this work and constitutes an important tool to be used in the process of multimedia storyboard generation, as it will be shown in the next section.

Figure 1 shows IDK with the ISBGen option. The central opened window shows the dialogue that appears when generating the storyboard from the meta-model definitions.

The tool facilitates the automatic production of the Alice file (with extension .a2w) that contains the storyboard in the suitable format. ISBGen translates the meta-models to an XML file that will be used as input in Alice. The use of XML provides a great

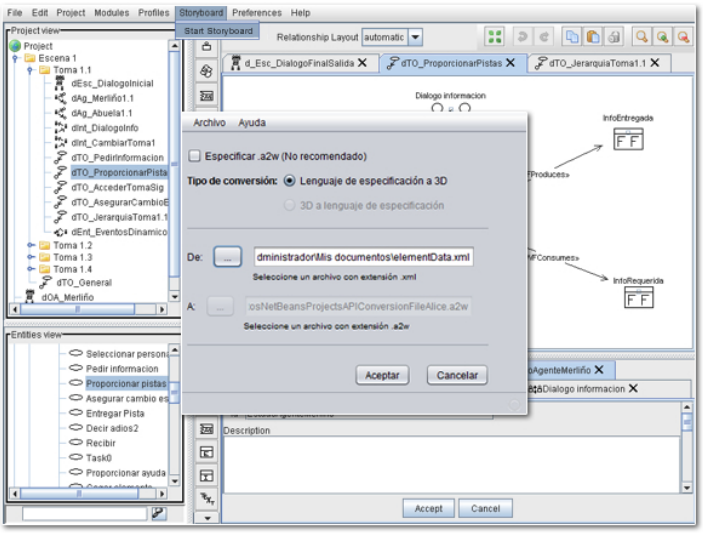


Fig. 1. ISBGen integrated in IDK

versatility for adapting the tool, if new characteristics appear in the models or the tools (IDK and Alice).

5 Development Process

5.1 Introduction

Traditional multimedia developments usually share two phases: pre-production and production. The first one includes the preliminary works oriented to define the basic characteristics of the system and to propose solutions to the most relevant functionalities. The phase of production will obtain the final product, developing in detail the sequences previously defined.

Habitually, in pre-production phase, the scriptwriters explain the basic idea using a storyboard which is evaluated by the project manager. The specifications are given after studying the script or storyboard that graphic designers make. From both resources, script and storyboard, a first prototype of the product can be modeled and an animated sequence is generated . This provides a good simulation of the final sequences to be developed and constitutes a natural way for the customer (producer, director, etc.) of verifying the model built.

5.2 Description

Our development process covers the pre-production phase and has as main aim the automatic generation of the storyboard. This process is based on the use of AOSE techniques adapted to multimedia systems, and on the utilization of ISBGen that produces fast simulation of the multimedia project at animation level (Storyboard Interactive).

At a high level of abstraction, the process consists of the cyclic application of the following steps:

1. The elements (object, tasks, goals, etc.) which constitute the system are defined using INGENIAS meta-models.
2. Using Alice, the graphical part of the elements (characters and objects in the scenery) is included. This object can be taken directly from Alice library or adapted from the available ones.
3. The ISBGen tool is used to generate in an automatic way the prototype.
4. The prototype is revised with the customer, the improvements are noted down and the process starts again from the first step. The changes imply only to redefine the meta-models.

Figure 2 shows a single iteration of the process previously described.

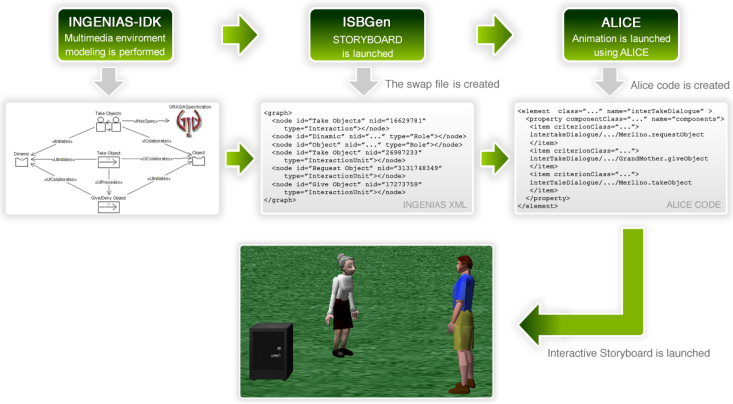


Fig. 2. Prototype generation process

Starting from a model constructed using IDK, an animated sequence, which constitutes a rapid interactive prototype that can be evaluated by the customer, is obtained. Also, the pre-production team can analyze the results, corroborate the implementation work that has been carried out and repair the deficiencies or optimize the features and animations.

Using this process, it is possible to show to the customer the idea that the scriptwriter has in mind, but starting with supporting actors (whose cost is significantly lower than protagonists). The prototype developed is a first test, that facilitates the study of all possible technical and graphics solutions to be applied in the final sequences. Moreover, the prototype permits to identify not obvious features and contributes to detect errors and problems that may appear in the future implementation.

The big advantage of this process is that, after this first model, a sketch of the system and the possible animated sequences that the script creates can be presented to the customer. Moreover, the feedback provided can be incorporated directly to the meta-models to improve the scene.

As it has been said before, the process is quick, because, the graphical part of most objects is obtained from Alice library. If it is the case that there is not a suitable object

in Alice for a particular element, the graphical part must be modeled in that moment. Nevertheless, the Alice objects can be adapted, if needed.

The process obtains as output a static model, which locates the elements inside the scene and defines what parameters should be modeled. This means establishing a scenery (a location) and natural components of the environment by default. These choices should reflect aspects such as the natural elements that interact with the environment: trees, plants, rivers,wind, rain . . . , or the lighting and its different types (radial, environment, parallel, etc.).

These elements are considered part of the environment and, as happens in virtual worlds like Second Life [26, 27] and OpenSim [28, 29], should have their own predefined attributes. Sometimes, the user should be able to act on such elements, but, in other moments, it is much more interesting that the elements have a random behavior, so that, the user can understand how they interact with the new world (like in some sites of virtual worlds).

The main aim of this part of the process is to fix the meta-model elements that are necessary to represent a multimedia scene and the other elements that are present on the scene but have not a relevant behavior. The agent paradigm provides an enormous potential to model these elements with undefined behavior, that perform the role of extras. The use of intelligent agents allows defining their behavior independently. Once this behavior has been defined, new not anticipated situations may appear from the interaction among agents, and these situations can be identified before interfering with the relevant ones.

5.3 Process Application

The system to be modeled as a basic example is a dialogue between two characters (named *Merlin* and *Grandmother*) who discuss how to give each other a particular object. This section explains step by step the results that can be achieved using the process previously described and the using the introduced tools (Alice, IDK and ISBGen).

Structure and Use Cases Models. The initial step of this process will be to define the **project structure**, that is, the number of scenes/shots that compose the whole system. For the case study, the dialog is composed of a single scene that, in turn, is divided in several shots (in the example of Figure 3 three shots are shown).

Next, a **use case diagram** must be modeled. Although this kind of diagram is not used in INGENIAS, we consider that it is of great utility in this moment of development. The diagram introduces the scenes which form part of the project and verifies the integrity of the structure previously defined. Figure 3 presents the use case diagram for the case study. The diagram shows a system with a scene consisting on three shots, but the example we are dealing with will be focused only in the first shot.

From this diagram, the ISBGen tool creates in Alice the objects defined in the diagram. Each use case corresponds to a shot and the tool introduces a number when generating the Alice object. These numbers indicate the order in which the use cases are going to be played. For instance, the model contained in Figure 3 will generate two objects in Alice: *shot1* and *shot11*. *Shot1* refers to ”Scene 1”, while *shot11* corresponds

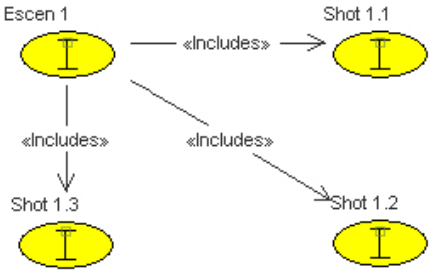


Fig. 3. Use Case Model

to the ”shot1.1” of INGENIAS model that is part of ”Scene1”’ (see the image in the central bottom part of Figure 4).

Organization Model. Following with the definition of the prototype, next step is defining the **Organization Model**, which describes the groups of agents, the functionality and the restrictions imposed on the agents. Before starting this diagram it is important to know the actors who interact with the system, their roles and, if the actors play different roles, how this roles will be grouped.

In Figure 4, the Organization Model of the case study is presented. The system consists of two groups of agents: *static* and *dynamic* agent. The agents belonging to the dynamic group are characterized by their capability to attend user interactions, while those of the static group do not require user intervention, so that they are completely autonomous, and act following their established goals.

This model has a direct correspondence with the 3D world modeled in INGENIAS and automatically generated by the tool, which can be seen in bottom part of Figure 4. ISBGen will generate, both the *Merlin* and the *Grandmother* agents from the diagram to Alice code. So, adding another agent to the virtual world would be as simple as including him in the organizational model and link him to a group of agents.

In the process, the next step is to define another **Organization Diagram**. In this case, the diagram details each of the shots associated to a particular scene. For instance, in the diagram of Figure 5, the interaction between agents that play the roles *dynamic* and *help* that fulfil the tasks *Request Information* and *Provide Clues*.

From this model, the communication between the two actors that play the roles *dynamic* and *help* can be extracted. The dynamic role selects the character to which it makes the request for information and, then the other role, through the establishment of a dialogue, provides the requested information. This dialogue makes that both roles achieve their established goals. The elements of this diagram are used to determine which tasks are involved in the interactions that occur during the execution of the system.

ISBGen does not use this diagram for code generation, sice since the tasks definition is performed later, and the information provided by this diagram is just illustrative for the developer. For this reason, this step can be skipped and it is only recommended to define these diagrams in large systems where this extra information can be useful.

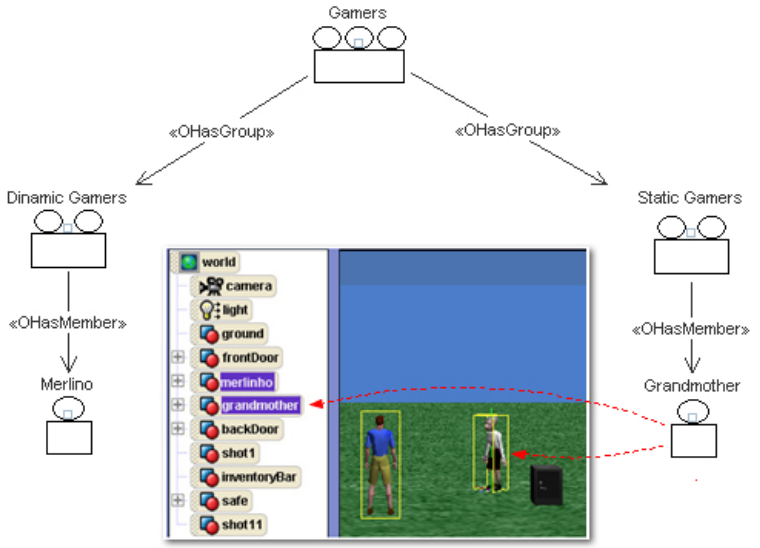


Fig. 4. General Organization Model

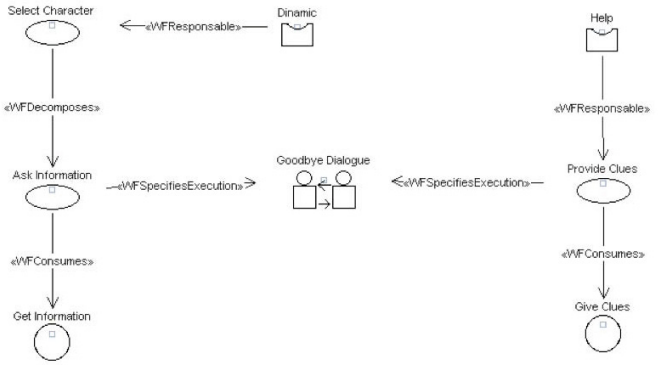


Fig. 5. Organization model for Shot 1.1

Tasks and Goals Model. Next phase consists in specifying the order in which the goals must be completed by agents. It is mandatory to take such decision before starting the execution of the system, and it is made through the definition of **Tasks and Goals Model**. The Tasks and Goals Model represents the relationships between goals, so that the order in which they must be achieved can be deduced. Moreover, the model may be used to break down the goals in the particular tasks that must be done to fulfil them. In the case study, the Tasks and Goals model has a sole goal: *to dialogue*, so that, the diagram is not showed here.

Task and Goals Model is very relevant in this process because, the hierarchical structure of the system is completed using the information that it contains. The methods and functions responsible for managing the lists of basic and secondary goals are automatically built by ISBGen.

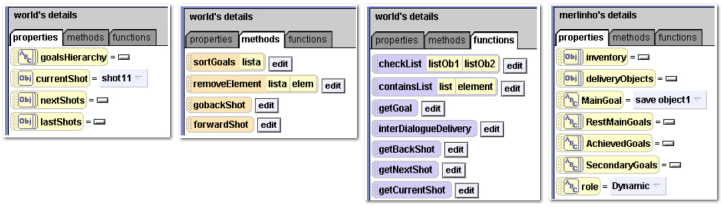


Fig. 6. Correspondences Alice - Task and Goal Model

From all the properties available in Alice, we have chosen the most suitable ones for defining agent characteristics. In particular, Figure 6 reflects in Alice the list of goals, methods, functions and properties associated with the case study. The properties panel (Properties) of the object *world* (the root of the project) -left image- is an ordered list of the main goals of the system called *GoalHierarchy*. It also contains the objects that reference the current, the next and the previous shot. In the methods panel (Methods) -second image- the methods for operating on the goals list and for modifying the current shot are shown. The third panel (functions) contains the methods needed to perform operations on agents, objects, shots and goals. Finally, the right panel displays the properties of the object *principal*, that is,the attributes of the main character, the list of objects and the list of goals, among others.

Agent Model. Next diagrams to be constructed are based on the **Agent Model**. An Agent Model must be done for each agent in the system. The model will show the capabilities and goals of each of the actors. The capabilities are represented by tasks, and each task must have an associated goal that must be achieved after execution. Summarizing, the agent model allows a detailed description of existing agents in the multimedia system.

An example of this kind of model is shown in Figure 7. From the definition of Agent Models, ISBGen generate the data related to the agent, its tasks and goals, the methods, and the list of goals to be met, which will be used in Alice. The figure reflects all the previous attributes for the main character of the case study. Each new feature of the 3D character of the system must be specified in the corresponding Agent Model as a task and, also, the goal accomplished by the task must be specified. For instance, to make an actor capable to move, in the Agent Model an agent with the tasks *move* and the goal *walk* must be defined. Once the information is used by ISBGen to generate the character, this will be reflected in the 3D world as a new method of the character of Alice, symbolizing the actor.

The bottom part of Figure 7 contain the properties, the methods and the functions for the main character of the case study, in particular the one called *Merlinho*. The properties panel represented contains the list of objects in the system, the ordered lists of goals and the role played by the agent selected. The second image -lower left- panel shows the object's methods, that is, the basic methods of the dynamic character, the movements and all the functionalities specified in the Tasks and Goal Model for the agents and tasks. The last image, bottom right, shows the functions panel, where the exchanges between the elements and the environment are shown. These functions represent the units of interaction described in the Interaction Model.

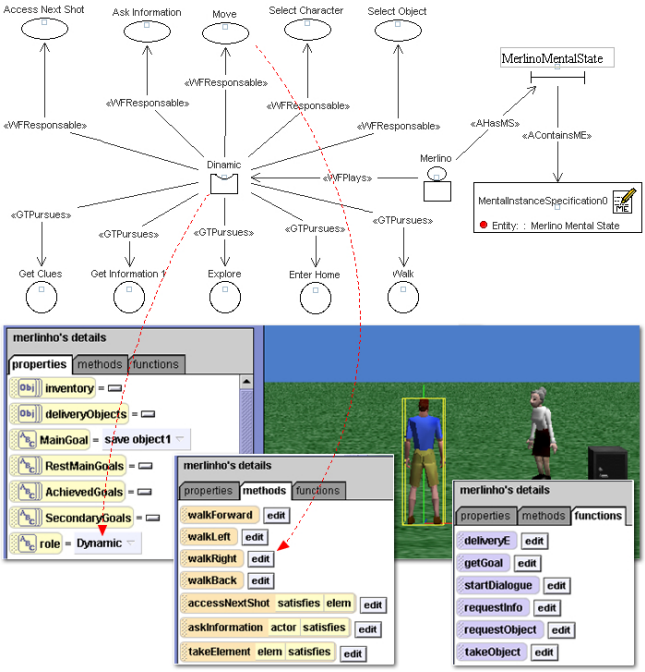


Fig. 7. Agent Model

Interaction Model. To complete the example of case study, it is necessary to use the Interaction Model. All the information exchanges that occur between two or more actors in the system should be reflected as interactions. This diagram represents the information exchanges, breaks down the actions of each interaction and specifies what is sent, and who is the sender and the receiver.

Figure 8 shows the model that defines the dialogue between the two agents, that is the core of the system. The bottom part of the figure introduces the ALICE definitions associated with the interaction units of the model. Each interaction unit appears in ALICE as a function associated with the agent that initiates the interaction.

Environment Model. The inclusion of events in the 3D world is done through the Environment Model, by identifying different internal applications that can be used and shared by the actors.

Finally, it will be necessary to describe in detail each of the tasks in the system using again Tasks and Goals Models. In these models the inputs and outputs that occur during the execution of a task and what are the initiators of such events should be described. The example in Figure 9 contains the description of the task *Provide clues* initiated by receiving certain information, called *RequiredInformation*. This task produces other task *GaveInformation*, which is used in the interaction, to complete the objective *Provide clues*. In addition, the task changes the mental status of the agent responsible for the task.

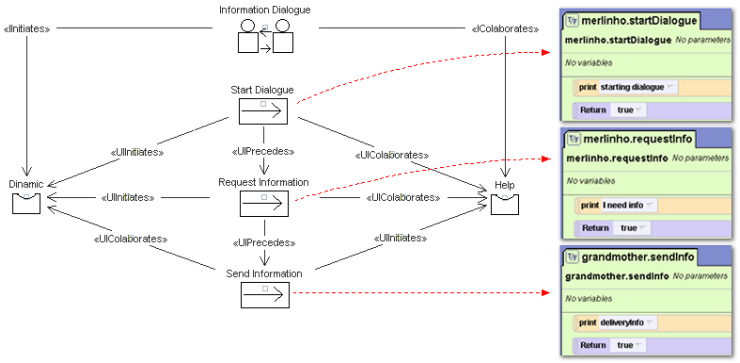


Fig. 8. Interaction Model

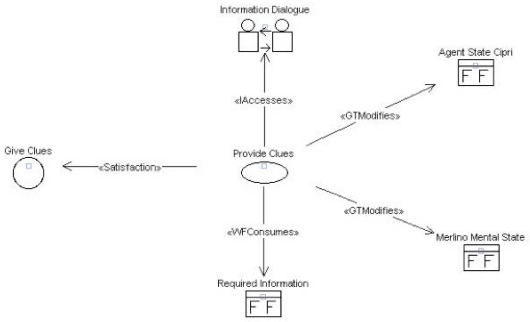


Fig. 9. Task and Goal Model

Review by the Customer. After modeling the environment, as stated before, an initial review in a meeting with the customer is carried out. This approach, though simple, allows to discuss this first prototype of the final product.

Any change to this prototype is low cost because the system is still in the very early stages of development. So that, this does not signify a cost beyond the predictions and is part of the process of product development. The customer can immediately see this first draft and confirm or change the specification of the generated prototype. Thus, the interaction developer/customer becomes really efficient, resulting in rapid prototypes that conform to the expected pattern. Attending the customer changes, a process of redesign, involving product improvement through optimization of the results of the prototype, is followed. With all this, a rapid definition of the expected product, which may be implemented with less risks, is obtained.

6 Conclusions

In the first steps of a multimedia project a high-level prototype makes easier the task of obtaining and validating the basic system's functionalities. The prototype makes available an overview of the multimedia product, and facilitates the feedback.

This increases the efficiency in the pre-production process, because all the changes needed are made at initial stages of development: unnecessary sequences can be eliminated or others are discovered. Moreover, the obtained models have more meaningful elements whose implementation is partially automated thanks to the tools offered. The use of ISBGen to obtain in an automatic way the storyboard from those definitions substantially reduces the time and costs of preproduction.

The use of a well-defined process supported by several tools introduces a certain degree of formalization in multimedia development, which usually lacks a formal process of software specification. In addition, the use of agent paradigm is valuable, as the concept of role, associated to a particular agent behavior, enhances the creation of intelligent autonomous behaviors that can discover new situations, not a priori defined, that may occur in the scenes. The use of ALICE, the modeling elements provided by the tool, as well as their by default behavior, makes it much easier to find objects that adapt to a particular situation to model.

From the example presented, it can be concluded that modeling a multimedia system is feasible using Agent Oriented methods. More over, the example has identified some limitations of the methodology for being used in this kind of developments. For instance, INGENIAS introduces many modeling elements that are not needed for multimedia definitions (like the sequence Model or the BDI mental state). In the future, we plan to pose a new methodology that overcomes such limitations. Besides, a new IDK profile for multimedia systems, which incorporates the proposed changes, will be defined.

Despite the satisfactory results of ISBGen tool, several improvements are intended to be made in the future. In particular, an interesting feature will be to be able to choose the graphical aspect of the agents. At the moment, the tool automatically assigns an Alice character for each of the agents in the system. In the future, the user will select for each agent its character from the ones available in Alice library.

The process proposed follows a top-down approach. Nevertheless, once the storyboard is obtained and presented to customer, it suffers changes. Offering a mechanism to automatically incorporate those changes to meta-models definitions, providing some kind of reengineering process, is, at the moment, under study.

Finally, we intend to integrate the NPL4INGENIAS tool [30, 12] in the process of development. NPL4INGENIAS is a tool that obtains the system requirements (using INGENIAS metamodels) from its description in Natural Language. The tool can be easily integrated in the process defined here and can simplify the obtention of a first model of the system to construct. A first experiment, which suggest this is possible, has been described in [16].

References

1. Songer, A.D., Diekmann, J.E., Karet, D.: Animation-based construction schedule review. Colorado, Construction Innovation: Information, Process, Management, 181–190 (2001)
2. Ramos-Valcárcel, D., Fajardo-Toro, C.H., Ojea, F.J.d.l.P.: Multimedia smart process (msp). In: 2011 6th Iberian Conference on Information Systems and Technologies (CISTI), vol. 1, pp. 320–325 (June 2011)

3. Masuch, M., Rueger, M.: Challenges in collaborative game design developing learning environments for creating games. In: Proceedings of the Third International Conference on Creating, Connecting and Collaborating through Computing, pp. 67–74. IEEE Computer Society, Washington, DC (2005)
4. Capra, M., Radenkovic, M., Benford, S., Oppermann, L., Drozd, A., Flintham, M.: The multimedia challenges raised by pervasive games. In: Proceedings of the 13th Annual ACM International Conference on Multimedia, MULTIMEDIA 2005, pp. 89–95. ACM, New York (2005)
5. Moreno-Ger, P., Fuentes-Fernández, R., Sierra-Rodríguez, J.L., Fernández-Manjón, B.: Model-checking for adventure videogames. Information and Software Technology 51(3), 564–580 (2009)
6. Ramos-Valcárcel, D.: Ingeniería de software orientada a agentes en el modelado de sistemas multimedia (in spanish). PhD thesis, Universidad de Vigo. Departamento de Informática. Escola Superior de Enxeñería Informática (2011)
7. Pavón, J., Gómez-Sanz, J.: Agent Oriented Software Engineering with INGENIAS. In: Mařík, V., Müller, J.P., Pěchouček, M. (eds.) CEEMAS 2003. LNCS (LNAI), vol. 2691, pp. 394–403. Springer, Heidelberg (2003)
8. Cuesta, P., Gómez, A., González, J., Rodríguez, F.J.: The MESMA methodology for agent-oriented software engineering. In: Proceedings of First International Workshop on Practical Applications of Agents and Multiagent Systems (IWPAAMS 2002), pp. 87–98 (2002)
9. Cossentino, M., Sabatucci, L.: Agent System Implementation. Agent-Based Manufacturing and Control Systems: New Agile Manufacturing Solutions for Achieving Peak Performance. CRC Press (2004)
10. Sturm, A., Dori, D., Shehory, O.: Single-model method for specifying multi-agent systems. In: AAMAS, pp. 121–128. ACM (2003)
11. Fuentes-Fernández, R., García-Magariño, I., Gómez-Rodríguez, A.M., González-Moreno, J.C.: A technique for defining agent-oriented engineering processes with tool support. Engineering Applications of Artificial Intelligence 23(3), 432–444 (2010)
12. Gómez-Rodríguez, A., González-Moreno, J.C., Ramos-Valcárcel, D., Vázquez-López, L.: Modeling serious games using aose methodologies. In: 2011 11th International Conference on Intelligent Systems Design and Applications (ISDA), pp. 53–58 (November 2011)
13. Pavón, J., Gómez-Sanz, J.J., Fuentes-Fernández, R.: IX. In: The INGENIAS Methodology and Tools, pp. 236–276. Idea Group Publishing (2005)
14. Ma, M.: Automatic conversion of natural language to 3D animation. PhD thesis, University of Ulster, Derry, Ireland (2006)
15. Kevin Glass, S.B.: Automating the creation of 3d animation from annotated fiction text. In: IADIS 2008: Proceedings of the International Conference on Computer Graphics and Visualization 2008, pp. 3–10 (2008)
16. Bolaño-Rodríguez, E., Moreno, J.C.G., Ramos-Valcárcel, D., López, L.V.: Using multi-agent systems to visualize text descriptions. In: PAAMS, pp. 39–45 (2011)
17. UVa User Interface Group: Alice: Rapid prototyping for virtual reality. IEEE Computer Graphics and Applications 15, 8–11 (1995)
18. Cooper, S., Dann, W., Pausch, R.: Alice: a 3-d tool for introductory programming concepts. Journal of Computing Sciences in Colleges 15(5), 107–116 (2000)
19. Conway, M., Audia, S., Burnette, T., Cosgrove, D., Christiansen, K., Deline, R., Durbin, J., Gossweiler, R., Koga, S., Long, C., Mallory, B., Miale, S., Monkaitis, K., Patten, J., Pierce, J., Shochet, J., Staack, D., Stearns, B., Stoakley, R., Sturgill, C., Viega, J., White, J., Williams, G., Pausch, R.: Alice: Lessons learned from building a 3d system for novices (2000)
20. Gómez-Sanz, J.J., Fuentes, R.: Agent oriented software engineering with ingenias. In: Garijo, F.J., Riquelme, J.-C., Toro, M. (eds.) IBERAMIA 2002. LNCS (LNAI), vol. 2527, Springer, Heidelberg (2002)

21. Botti, V., Giret, A.: ANEMONA: A Multi-agent Methodology for Holonic Manufacturing Systems. Springer Series in Advanced Manufacturing (2008)
22. Pavón, J., Gómez-Sanz, J.J., Fernández-Caballero, A., Valencia-Jiménez, J.J.: Development of intelligent multisensor surveillance systems with agents. Robotics and Autonomous Systems 55(12), 892–903 (2007)
23. Soto, J.P., Vizcaino, A., Portillo, J., Piattini, M.: Modelling a Knowledge Management System Architecture with INGENIAS Methodology. In: 15th International Conference on Computing (CIC 2006), Mexico City, Mexico, pp. 167–173 (2006)
24. Hidalgo, A., Gomez-Sanz, J., Mestras, J.: Workflow Modelling with INGENIAS methodology. In: 5th IEEE International Conference on Industrial Informatics (INDIN 2007), Vienna, Austria, vol. 2, pp. 1103–1108 (2007)
25. Cuesta, P., Gómez-Rodríguez, A., González-Moreno, J.C.: Agent oriented software engineering. In: Whitestein Series in Software Agent Technologies and Autonomic Computing, pp. 1–31. Springer (2007)
26. Kumar, S., Chhugani, J., Kim, C., Kim, D., Nguyen, A., Dubey, P., Bienia, C., Kim, Y.: Second life and the new generation of virtual worlds. Computer 41(9), 46–53 (2008)
27. Rymaszewski, M.: Second life: the official guide. John Wiley (2007)
28. Childers, B.: Run your own virtual reality with opensim. Linux J (March 2009)
29. Fishwick, P.: An introduction to opensimulator and virtual environment agent-based applications. In: Proceedings of the 2009 Winter Simulation Conference (WSC), pp. 177–183 (December 2009)
30. Gonzalez-Moreno, J.C., Vazquez-Lopez, L.: Design of multiagent system architecture. In: Ieee (ed.): IEEE International Workshop on Engineering Semantic Agent Systems, Turku (Finlandia), pp. 565–568. IEEE (2008)

A Semi-automated Approach towards Handling Inconsistencies in Software Requirements

Richa Sharma[1] and K.K. Biswas[2]

[1] School of Information Technology, IIT Delhi, India
[2] Department of Computer Science, IIT Delhi, India
sricha@gmail.com, kkb@cse.iitd.ernet.in

Abstract. Software Requirements expressed in the form of natural language are often informal and possibly vague. The need for formal representation of the requirements has been explored and addressed in various forms earlier. Of several recommended approaches, logical representation of requirements has been widely acknowledged to formalize the requirements languages. In this paper, we present courteous logic based representations for software requirements. We report the benefits of courteous logic based representations for handling inconsistencies in software requirements and take into account views of multiple stakeholders and the presuppositions. We show how courteous logic based representations can be used to ensure consistency as well as to uncover presuppositions in the requirements.

Keywords: Requirements Specification, Inconsistency, Presupposition, Knowledge Representation, Courteous Logic.

1 Introduction

Software Development commences with the phase comprising requirement engineering activities. Any software model, whether it is Waterfall model, Iterative or Agile, commences with tasks centred on the popularly known requirements phase. The requirements phase becomes crucial to the success of the software as this phase only serves as the basis for subsequent phases of software development. It would not be incorrect to say that establishing correct requirements is imperative even in agile model where developers work in close connection with the users; there is still need to clearly specify the requirements so that the requirements are well understood by the developer. The mode of expressing the requirements may vary from visual models to textual use-cases to user scenarios, but the underlying idea is that the requirements should be a good representation of the domain under study and should be well understood and agreed upon by all the stakeholders and the end-users. There is no precise definition as to what is meant by 'good' representation but with reference to features of good software requirements specification, requirements representations should be consistent, unambiguous and complete in nature [1]. It is a challenging task to draw upon the representations satisfying these qualities. The most common defect that arises in software requirements representation is that of *inconsistency*. The elicited requirements describing the functional specifications of the behaviour of the information system or the software application are frequently vague

L.A. Maciaszek and J. Filipe (Eds.): ENASE 2012, CCIS 410, pp. 142–156, 2013.

and incomplete; therefore, need enrichment during analysis for functional behaviour as well as non-functional properties. Requirements engineers need to examine these gathered requirements and to transform this "rough" sketch of requirements into a correct requirement specification [2]. As a result, new requirements are identified that should be added to the specification, or some of the previously stated requirements may need to be deleted to improve the specification. It becomes a critical task for requirements engineers to maintain the consistency of the set of specified requirements.

Our contribution in this paper is to demonstrate the courteous logic based requirements representations (requirements specification hereafter) as a worthwhile approach towards handling inconsistency in the requirements and the related issue of identifying presuppositions [3]. The use of logic to formalize the requirements has been acknowledged in earlier works too as logic offers proof-theoretic framework facilitating requirements validation and evolution. We are making use of a form of non-monotonic logic as requirements evolution becomes non-monotonic after an initial monotonic phase [2].

The rest of the paper is organized as follows. In section 2, we elaborate the inconsistency concern that motivated the use of courteous logic. Section 3 describes courteous logic in detail and how it can be used to address the inconsistency and presupposition concern in requirements. In section 4, we present an overview of related work followed by possible threats to the usage of formal logic by requirements analysts in section 5. Section 6 finally presents discussion and conclusion.

2 Inconsistency Concern

Several adequate definitions are available elucidating the inconsistency concern in software requirements [4], [5]. Of the two IEEE interpretations of inconsistency, namely internal inconsistency and the Software Requirements Specification not agreeing with some higher level document (traceability), we are interested in addressing the former one. Internal inconsistency arises when the specification of requirements contains conflicting or contradictory information about the expected behaviour of the system. The reason for conflicting information existence can be attributed to multiple views of the stakeholders. Internal inconsistency may also arise when some new requirement is added or an existing requirement is removed from the requirements specification. Internal inconsistency may also arise due to some implicit knowledge, referred to as presupposition that could not find explicit expression in the requirements set. These forms of inconsistency may be detected or remain undetected at the time of requirements analysis. It is often a matter of the domain expertise of the requirements analyst to detect the inconsistencies in the elicited requirements. If inconsistency is detected during requirements phase, then it can be corrected and software defects can be contained in requirements phase only. If undetected, then the problem permeates the subsequent phases as the programmer might resolve the concern arbitrarily during implementation. The undetected inconsistency is an indication that possibly the analyst has had a presupposition about the inconsistent information, or possibly it was overlooked. Presupposition being a linguistic phenomenon is related to the utterance and the intuitive meaning of the sentence [6]. In context of requirements, presuppositions significantly contribute to inconsistency and incompleteness problems.

In order to contain defects in the requirements phase and effectively manage the concern of inconsistency, certain degree of formalism is required in the requirements specification. It is essential to draw a formal model or representation of requirements that is a reflection of the expected observable behaviour of the system. Such a model or representation would allow identifying implicit or hidden inconsistency, which is otherwise difficult to identify as implicit inconsistency arises out of interpretation or consequence of the requirements. We can easily identify and take into consideration the explicit inconsistency arising because of multiple views of stakeholders, but no such easy approach is there with implicit inconsistency. We'll see in next section how courteous logic expressions address this requirements representation problem and obligates the analyst to explicitly specify the presuppositions too, if any.

3 Courteous Logic Based Solution

The idea of having requirements representation in a form that shows the expected observable behaviour of the system and the need to formalize requirements specifications has led to the use of logical representations of the requirements. We are more focused on non-monotonic reasoning for two reasons. First, to address the problem of inconsistency and explicit specification of presuppositions in the requirements; and secondly, real-world requirements correspond to human way of thinking and common-sense reasoning which is non-monotonic in nature. We found courteous logic based representation for requirements specification is suitable for our cause. We have already shown the adequacy of courteous logic for requirements specification in [7]. Here, we highlight the concern of undetected inconsistency and the associated concern of presupposition.

3.1 Courteous Logic

Courteous Logic [8] is a form of non-monotonic logic where consequence relation is not monotonic. It is based on prioritization of the rules. Courteous logical representation (CLP) is an expressive subclass of ordinary logical representation (OLP) with which we are familiar and, it has got procedural attachments for prioritized conflict handling. First Order Logic(FOL) beyond Logic Programming (LP) has not become widely used for two main reasons: it is pure belief language; it cannot represent procedural attachments for querying and actions and, it is logically monotonic; it can not specify prioritized conflict handling which are logically non-monotonic. The Courteous Logic Programming (CLP) extension of LP is equipped with classical negation and prioritized conflict handling. CLP features disciplined form of conflict-handling that guarantees a consistent and unique set of conclusions. The courteous approach hybridizes ideas from non-monotonic reasoning with those of logic programming. CLP provides a method to resolve conflicts that arise in specifying, updating and merging rules. Our CLP representations are based on IBM's CommonRules, available under free trial license from IBM alpha works [9].

In CLP, each rule has an optional rule label, which is used as a handle for specifying prioritization information. Each label represents a logical term, e.g., a logical 0-ary function constant. The "overrides" predicate is used to specify prioritization.

"overrides (lab1, lab2)" means that any rule having label "lab1" is higher priority than any other rule having label "lab2". The scope of what is conflict is specified by pair-wise mutual exclusion statements called “mutex's”. E.g., a mutex (or set of mutex's) might specify that there is at most one amount of discount granted to any particular customer. Any literal may be classically negated. There is an implicit mutex between p and classical-negation-of-p, for each p, where p is a ground atom, atom, or predicate.

An example illustrating the expressivity and non-monotonic reasoning of CLP: Consider following rules for giving discount to customer:

- If a customer has Loyal Spending History, then give him 5% Discount.
- If a customer was Slow to pay last year, then grant him No Discount.
- Slow Payer rule overrides Steady Spender.
- The amount of discount given to a customer is unique.

These rules are represented in CLP as following set of rulebase:

```
<steadySpender>
if shopper(?Cust) and spendingHistory(?Cust, loyal)
then
    giveDiscount(percent5, ?Cust);

<slowPayer>
if slowToPay(?Cust, last1year)
    then giveDiscount(percent0, ?Cust);

overrides(slowPayer, steadySpender);
```

As discussed above, the two types of customers are labeled as <steadySpender> and <slowPayer>; the predicate ‘overrides’ is used to override the discount attributed to slowpayer over the discount to be given to steadyspender in case a customer is found to be both steadySpender and slowPayer.

We found that courteous logic representations are closer to natural language representation of business rules in terms of commonly used ‘if - then’ rules and, can be easily interpreted by both the users and the developers. An experience with CLP shows that it is especially useful for creating rule-based systems by non-technical authors too [8]. Another advantage of CLP is computational scalability: inferencing is tractable (worst-case polynomial time) for a broad expressive case. By contrast, classical logic inferencing is NP-hard for this case.

3.2 Our Contribution

Requirements are usually expressed using Natural Language (NL) in requirements specification. We worked towards automated translation of requirements expressed in NL to Courteous Logic Form using Norm Analysis Patterns. Norms are the rules and patterns of behavior, formal or informal, explicit or implicit, existing within an organization or

society [11]. The requirements specifications represent the expectations, needs and desires of the stakeholders from the proposed information system. These specifications are, in fact, representative of norms for the domain of information system under consideration.

The choice of norm analysis patterns was made for two reasons: (a) Norms represent the behavioral patterns in an organization; (b) Norm Analysis Pattern representation is suitable for automated translation to courteous logic. We have made use of search algorithm to find norm analysis patterns ("If-then" form). Next, we applied lexical and syntactic analysis to the antecedent (If –part of the norm) and the consequent (then – part of the norm). This analysis helped us in translating norms to courteous logic formalism. Since our focus in this paper is handling inconsistencies in requirements, we will not delve deeper into our methodology. The approach has been discussed in detail in [12]. Though the translation of requirements in natural language is automated, still the observation of inference from courteous logic reasoning is manual and comparing those observations against real-world behavior is also manual. Therefore, we would prefer to call our approach as semi-automated.

3.3 Identifying Inconsistency and Presuppositions

In this sub-section, we bring forth the expressive and the reasoning power of courteous logic and, demonstrate how it proves effective in identifying instances of inconsistency and consequent presupposition through three case-studies in varying domains.

***Example 1* - Representing and Prioritizing Conflicting Views (Academic Grade Processing):** Consider the specification of students' grade approval process where the students' grades are approved by the course-coordinator, the department head and the dean. The expected behaviour of the system refers to the fact that at any point in time, approval from department head holds higher priority over course-coordinator; and approval from dean higher priority over department head and in turn, the course coordinator. Often, this observable behaviour is not captured in its essence in requirements specification. Instead, a process flow is captured stating that students' grades once entered in the system need to be approved by the course-coordinator, the department head and the dean.

This particular use-case served an excellent example of inconsistency as well as the presence of presupposition. In the absence of validation against explicit expected behaviour of the real-time system, the software system can possibly have an inconsistent state of grades subject to the arbitrary implementation done by the programmer. The pragmatic presupposition [6] associated with this use-case is that the process-flow describing approval by the course-coordinator, the department head and the dean refers to a sequential and prioritized flow with highest priority of the dean, followed by department head and then, course-coordinator. Consequently, programmer needs to take care of these details and should not take any decision arbitrarily. The courteous logic

specification of the requirements as stated in the given use-case translates to four labelled rules, namely *new, cdn, hod and dean* respectively:

```
<new>
   if assignGrades(?Regno, ?Year, ?Sem,        ?Group,       ?Sub,
?Point)
   then valStatus(new, ?Regno, ?Year,   ?Sem, ?Group, ?Sub);
<cdn>
   if approvedby(?Regno, ?Year, ?Sem,   ?Group,   ?Sub,   ?Point,
?Status,     coordinator)
   then valStatus(coordApproved, ?Regno,       ?Year,        ?Sem,
?Group, ?Sub);
<hod>
   if approvedby(?Regno, ?Year, ?Sem, ?Group,?Sub, ?Point, coor-
   dApproved, hod)
   then valStatus(hodApproved, ?Regno,  ?Year,   ?Sem,   ?Group,
?Sub);
<dean>
    if  approvedby(?Regno, ?Year, ?Sem, ?Group,?Sub, ?Point, ho-
dApproved, dean)
   then  valStatus(deanApproved, ?Regno,      ?Year,        ?Sem,
?Group, ?Sub);
```

In the expressions above, ?X represents a variable. The rule labelled as <*new*> specifies that when a student with registration number, ?Regno; year of study, ?Year; semester and group as ?Sem and ?Group respectively is assigned grades for points, ?Point in subject, ?Sub, then status of his grades would be *new*. The rule labelled as <*cdn*> specifies that grade status changes to *coordapproved* on approval by coordinator. Similarly, the rules labelled <*hod*> and, <*dean*> indicate the grade status on approvals by the department head and the dean respectively.

We verified the expected behaviour of the system in the absence of any prioritization of the rules as that information was not explicitly mentioned in the given use-cases. This use-case presented us with the case of implicit inconsistency present in the consequences of the above rules. We took three sample given facts for a student with registration number 2008CSY2658:

```
   assignGrades(2008CSY2658, 2009, even, 4, ai, c);
   approvedby(2008CSY2658, 2009, even, 4, ai, c, new, coordina-
tor);
   approvedby(2008CSY2658, 2009, even, 4, ai, c, new, hod);
```

Corresponding to these facts and the given labeled rules, the consequences inferred for grade status of this student was found to be taking three distinct values at one point in time:

```
valStatus(hodApproved, 2008CSY2658, 2009, even, 4, ai);
valStatus(new, 2008CSY2658, 2009, even, 4, ai);
valStatus(coordApproved, 2008CSY2658, 2009, even, 4, ai);
```

Since this is not practical for any status term to have multiple values assigned at one point of time, it represents an inconsistent state of the world. For all practical purposes, we can safely say that above specification is an inconsistent reflection of the real-world system. Since our specification is an executable model of the real-world, we could validate the specification against expected behaviour of the system and, reach the conclusion of inconsistency at an early stage of software development. Next, this observation pointed to the presence of some knowledge which is not yet put into words, i.e. *presupposition*. Further investigating and refining the requirements based on this observation and enriched knowledge, we added following rules:

```
overrides(cdn, new);                    overrides(hod, new);
overrides(dean, new);                   overrides(hod, cdn);
overrides(dean, cdn);                   overrides(dean, hod);
MUTEX
   valStatus(?Status1, ?Regno, ?Year, ?Sem, ?Group, ?Sub)
   AND
   valStatus(?Status2, ?Regno, ?Year, ?Sem, ?Group, ?Sub)
GIVEN
  notEquals( ?Status1, ?Status2 );
```

The overrides clause establishes the prioritizing relationship between the grade approval rules where the first argument holds higher priority above the second argument. The MUTEX specifies the scope of conflict, which is the grade status in our case. The overrides clause takes care of the possible conflict in the student's grade status. Validating the updated specification against the observable expected behaviour of the grade approval processing, we found our specification consistent as the consequence obtained was the expected one:

```
valStatus(hodApproved, 2008CSY2658, 2009, even, 4, ai);
```

The above example illustrates how conflicting information can be expressed with well-formed semantics of courteous logic. We present one more example below that illustrates expressing some default operation in a domain as well as exceptions to that processing.

Example 2* – *Representing Default and Exceptional Scenario Processing (Saving and Current Account Processing): Consider the account processing part of a bank customer where he can have more than one account. Let's consider that a bank customer can have a current account and a saving account. The customer can choose one of these accounts as default account for any transaction that he wants to carry out. The usual choice is current account but to keep the use-case generic, let us assume that customer has marked one of the accounts as default. The customer is free to select the other account for some of his transactions. In that case, the selected account processing should override the default processing. The natural language expression for such default operation and associated

exception can be easily understood by the involved stakeholders as well as developers. But what is often overlooked by developers is the implicit interpretation here – the account chosen for default processing should remain unaffected in case selection is made for the non-default account and often, this is uncovered till testing phase. Such overlooked implicit interpretation results in implicit internal inconsistency. Such a defect can be easily detected during RE phase if we have an executable model or representation of requirements that can sufficiently express the domain knowledge.

The courteous logic specification of the requirements as stated in the given account processing for deposit and withdrawal transactions by the customer translates to following rules:

```
<def>
 if deposit(?Txn, ?Client, ?Amount) and holds(?Client, ?Acct)
 and default(?Acct)
then addAmount(?Client, ?Acct, ?Amount);
<sel>
 if deposit(?Txn, ?Client, ?Amount) and holds(?Client, ?Acct)
 and option(?Client, ?Txn, sel, ?Acct)
then addAmount(?Client, ?Acct, ?Amount);
<def>
 if withdraw(?Txn, ?Client, ?Amount) and holds(?Client,
 ?Acct) and default(?Acct)
then subAmount(?Client, ?Acct, ?Amount);
 <sel>
 if withdraw(?Txn, ?Client, ?Amount) and holds(?Client,
 ?Acct) and option(?Client, ?Txn, sel, ?Acct)
 then subAmount(?Client, ?Acct, ?Amount);
```

The rule with label <*def*> indicates transaction processing from default account whereas the rule with label <*sel*> indicates processing from the selected account. For deposit type of transaction, the default rule (the first rule in the above expressions) indicates that if a client, ?Client is holding an account, ?Acct marked as default and he initiates a deposit transaction with a certain amount, ?Amount then that amount will get credited or added to his default account. Similar is the case with withdrawal transaction. The client can choose another account provided he is holding that account and chooses it for some transaction as expressed through rule labeled as <*sel*>. In this case, any deposit or withdrawal transaction would affect the selected account only. To verify that any such transaction will not affect the default account, we first tested the rules without any override clause using some sample data as:

```
holds(abc, acctabc10);                holds(abc, acctabc11);
default(acctabc10);
deposit(t1, abc, 1000);               deposit(t2, abc, 5000);
withdraw(t3, abc, 2000);              withdraw(t4, abc, 4000);
```

```
option(abc, t2, sel, acctabc11);
option(abc, t3, sel, acctabc11);
```

Corresponding to these facts and the given labeled rules, the consequences inferred for each of the transactions t1, t2, t3 and t4 were found to be:

```
addAmount(abc, acctabc10, 1000);
addAmount(abc, acctabc10, 5000);
addAmount(abc, acctabc11, 5000);
subAmount(abc, acctabc10, 2000);
subAmount(abc, acctabc11, 2000);
subAmount(abc, acctabc10, 4000);
```

The results obtained indicate that deposit to and withdrawal from default accounts are adequately expressed as the behavior of dummy transactions, t1 and t4 is same as the expected behavior. But on account selecting, both the selected and the default accounts are getting affected as transactions t2 and t3 represent the outcome of the requirement expression. Adding the clause for prioritizing the selected account and making the default account and the selected account mutually exclusive so that only one of these accounts is impacted by some operation, we got the desired output – one that matches the expected behavior in real-time scenario:

```
overrides(sel, def);
MUTEX
 addAmount(?Client, ?Acct1, ?Amount) AND
 addAmount(?Client, ?Acct2, ?Amount)
GIVEN    notEquals( ?Acct1, ?Acct2 );
MUTEX
 subAmount(?Client, ?Acct1, ?Amount) AND
 subAmount(?Client, ?Acct2, ?Amount)
GIVEN    notEquals( ?Acct1, ?Acct2 );
```

As elaborated in detail in example 1, the mutex clause in this case establishes the scope of conflict over the two accounts and, the override clause assigns priority to the selected account. This example highlights two things:

a) The labels used as handle to some rule are not mere tags that need to be different from each-other. These can be repeated and reused in same specification provided their intent is same wherever used.

b) The expressions without ‘override’ and ‘mutex’ clause were consistent with natural language specification of the requirements but, were inconsistent with expected behavior. The implicit presupposition that only the selected account should be affected was uncovered at requirements level owing to the executable nature of our specification.

Let's consider one more example to show that multiple views of stakeholders can also be conveniently expressed using courteous logic based requirements specifications.

Example 3* – *Representing and Prioritizing Views of Multiple Stakeholders (Corporate Event Processing): Consider a corporate action event announced on a security. If a client is holding the security on which event is announced, then that client is eligible to get the announced benefits of the event. These benefits can either be in the form of cash or stock or both. The types of benefits disbursed to the clients vary from one event type to another; it also depends on various other factors like base country of the security on which event is announced, the country of the customer; client opting for an option etc. Then, there can be multiple stakeholders having differing views like one particular stock market has rules that do not allow client to opt any option announced on event; whereas, clients from some other market can opt for event's announced operations, so on and so forth. We took a small subset of this large set of rules and gradually scaled the rules as well as the data to find that results are consistent with the actual observable expectations. This particular example served towards claiming scalability of courteous logic based requirements specifications. Our expressions could not only be easily validated against the real-world expected behavior but also these were small and compact making them easy to comprehend and verify against multiple real-world scenarios as shown below:

```
<cash>
if event(?EventId, ?Type, ?Security) and holds(?Client, ?Security) and opts(?Client, cash)
then distribute(?Client, ?EventId, cash);
<stock>
if event(?EventId, ?Type, ?Security) and holds(?Client, ?Security) and opts(?Client, stock)
then distribute(?Client, ?EventId, stock);
<both>
if event(?EventId, ?Type, ?Security) and holds(?Client, ?Security) and opts(?Client, both)
then distribute(?Client, ?EventId, both);

<divMtk1>
if event(?EventId, dividend, ?Security) and holds(?Client, ?Security) and baseCntry(?Security, Mkt1)
then distribute(?Client, ?EventId, stock);
<divMkt2>
if event(?EventId, dividend, ?Security) and holds(?Client, ?Security) and clientCntry(?Client, Mkt2)
then distribute(?Client, ?EventId, nothing);
```

```
<divMkt1Mkt5>
if event(?EventId, dividend, ?Security) and holds(?Client, ?Security) and baseCntry(?Security, Mkt1) and clientCntry(?Client, Mkt5)
then distribute(?Client, ?EventId, cash);
```

The rule with label as <*cash*> indicates that if an event, ?Event of some type, ?Type is announced on a stock, ?Security and a client, ?Client is holding that stock and he opt for '*cash*' option then he will receive the benefit of event in the form of cash as per the announced rules of the event. Similarly, use-cases with stock and both types of disbursements are represented through rules labelled as '*stock'* and '*both'* respectively. These are generic rules. Next, we have considered a hypothetical scenario where in stakeholders from stock market, *Mkt1* are of the view that if 'dividend' type of event is announced on the stock belonging to their nation, then all customers shall get event's benefits as stock only. This is represented in the rule labeled as <*divMkt1*>. The rule with label <*divMk2*> indicates that dividend event announced will not entail any benefits to clients from stock market *Mkt2*. The last rule is an exception to rule <*divMkt1*> - it says that if client hails from the stock market, *Mkt5* then he is eligible for benefit in the form of cash rather than stock.

The above-mentioned rules were then verified against facts from real-world as below:

```
event(11, dividend, samsung);
event(22, dividend, dell);
baseCntry(dell, US);

holds(abc, samsung);                    holds(abc, dell);
holds(xyz, dell);                       holds(pqr, dell);
clientCntry(xyz,Mkt2);
clientCntry(pqr, Mkt5);
opts(abc, both);
```

In the absence of any kind of prioritization amongst multiple views, we got the validation results as:

```
distribute(pqr, 22, cash);
distribute(pqr, 22, stock);
distribute(xyz, 22, nothing);
distribute(xyz, 22, stock);
distribute(abc, 11, both);
distribute(abc, 22, both);
distribute(abc, 22, stock);
```

These results are not in line with what actual happens in the stock market as one conclusion indicates no benefit to xyz for event 22; whereas next conclusion points out stock benefit to the same client on the same event. When the multiple views from stakeholders of different stock market were assigned priorities (that can be easily modified or updated later on too), the results obtained were as per the expected benefits disbursed to the client in stock market abiding terms and conditions:

```
overrides(divMkt1Mkt5,divMkt2)
overrides(divMkt1Mkt5,divMkt1)
overrides(divMkt1Mkt5,cash)
overrides(divMkt1Mkt5,stock)
overrides(divMkt1Mkt5,both)
```

and similar rules for rest of the markets including the generic ones:

```
overrides(both,stock);
overrides(both,cash);
overrides(stock, cash);
MUTEX
 distribute(?Client,?EventId,Value1) AND
distribute(?Client,?EventId,?Value2)
GIVEN   notEquals( ?Value1, ?Value2 );
```

Validating the facts gathered earlier against the set of labeled rules and the prioritized information, consistent and expected results were obtained as:

```
distribute(abc, 22, stock);
distribute(pqr, 22, cash);
distribute(abc, 11, both);
distribute(xyz, 22, nothing);
```

3.4 Observations

Courteous logic based representations can sufficiently express software requirements with reasoning and inferring mechanism as shown in [9]. We have explored the inconsistency and presupposition concern in detail in this paper. The advantage of the courteous logic based requirements specification lies in following observations:

1. It supports expressing conflicting information in a way that is subjected to prioritized conflict-handling. Any modification to the priority rules is a matter of changing the priorities.

2. Adding any information or removing any information during requirements evolution ensures consistency of the requirements specification.

3. Having inference mechanism based FOL, it allows validating the specifications against the expected observable behavior of the system.

4. It provides assistance in identifying presuppositions (tacit knowledge) in the specified requirements.

5. Earlier detection of defects as well as any disconnect between the client's intent and the requirements analyst's interpretation.

We also observe that courteous logic based requirements specifications have the potential to improve requirements elicitation, management and evolution. Point 5 leads to further noticing that earlier detection of problems can considerably reduce cost and time effort of the project and help in keeping project schedule on time. Further, courteous logic based requirements specifications can be helpful in identifying and preparing test-cases. The test-cases can be easily mapped to the rules present in the courteous logic based requirements specifications. The validation data can serve as the starting base to test data.

4 Related Work

The use of use of logic for requirements representation and resolving inconsistency has been acknowledged earlier too and has found its place in several authors' work. RML [10] is one of the earliest logic-based formal requirements modeling language. Revamping efforts on RML gave way to two more related languages – Conceptual Modeling Language (CML) [13] and Telos [14]. HCLIE language [15] is a predicate logic-based requirement representation language. It makes use of Horn clause logic, augmented with multiple inheritances and exceptions. HCLIE handles non-monotonic reasoning by not reporting any consequence in case any conflict arises. Description Logic has also been used for representing requirements in [16]. Description logic is an expressive fragment of predicate logic and is good for capturing ontology whereas, Horn clause logic is useful for capturing rules. Ordinary Logic Programming (OLP) lacks in conflict-handling or non-monotonic reasoning. Default Logic form of non-monotonic reasoning has also been used to reason about inconsistencies in requirements in [5]. The computational complexity of default logic is high and the expressions are not too easy and convenient to comprehend.

Business rules do present themselves with many instances of conflicts and presuppositions around them. Handling business rules along with constraints and related events and semantics of rules are of paramount importance. To capture business rules successfully, Horn clause logic needs to be augmented with non-monotonic (/commonsense) reasoning as has been discussed in [17] and, here we have presented courteous logic based representations towards the cause. Syntactical presuppositions have been addressed in [3]. We have presented logical expressions as a solution towards addressing semantic and pragmatic presuppositions. The underlying essence and rational of requirements and the corresponding contextual meaning can be understood via some executable model of requirements. We have presented such a model in the form of courteous logic based requirements representation.

5 Threats to Validation

The logical expressions do offer a mechanism to reason with the requirements and resolve relevant issues. Nevertheless, these formal logical expressions might pose usability problems to the stakeholders as well as those developers who are not fluent with the formal logic.

Even though the courteous logic representations are more English-like with less usage of arrows and symbols, still accepting a new approach is not very encouraging until its benefits are realized. Secondly, some amount of training would have to be imparted to the involved parties in requirements phase in order to make them understand the syntax and the querying mechanism. Third, translating the requirements corpus to the courteous logic expression would be time-consuming and would depend on the individual's expertise and skills. Automated conversion from natural language to these representations would certainly be an advantage but doing that itself poses natural language parsing challenge.

We still hope that benefits drawn from using non-monotonic formal representations of requirements will outweigh the threats. Requirements of varying domains are of varied nature and not one kind of model is able to sufficiently express all the aspects of the system. A domain which is rule-intensive and has multiple conflicting views and requirements would certainly be benefitted by courteous logic based specifications.

6 Conclusions

In this paper, we have successfully addressed the problem of identifying and analyzing the logical inconsistency in the software requirements using courteous logic based requirements specifications in our work. We have shown that these specifications, being an executable model of the system's requirements, not just identify implicit inconsistencies but also help in identifying and specifying presuppositions explicitly. The results from the case-studies are encouraging. Tractable inferencing and scalability of the representations are some of the motivating factors towards using courteous logic based specifications. We have also demonstrated that fixing the inconsistency by rule-prioritization does not entail a major change in the existing requirements specification. This aspect makes these expressions a suitable choice from software maintenance point of view. Though the requirements analysts might not find the idea of using courteous logic comfortable, but since these representations are relatively simpler; easy to comprehend and natural language like, we hope that with small amount of training these could be well-taken by the practitioners as well.

We further aim to refine the current proposed requirements representations and incorporate the second interpretation of inconsistency – traceability; and then develop a semantic framework for automated analysis of requirements. We see our framework as a foundation towards integrated framework for semantic software engineering.

References

1. IEEE Computer Society. IEEE Recommended Practice for Software Requirements Specification, IEEE Std 830 – 1998(R2009) (1998)
2. Zowghi, D.: On the Interplay Between Consistency, Completeness, and Correctness in Requirements Evolution. Information and Technology 45(14), 993–1009 (2003)
3. Ma, L., Nuseibeh, B., Piwek, P., Roeck, A.D., Willis, A.: On Presuppositions in Requirements. In: Proc. International Workshop on Managing Requirements Knowledge, pp. 68–73 (2009)
4. Tsai, J.J.P., Weigert, T., Jang, H.: A Hybrid Knowledge Representation as a Basis of Requirements Specifciation and Specification Analysis. IEEE Transaction on Software Engg. 18(12), 1076–1100 (1992)
5. Gervasi, V., Zowghi, D.: Reasoning about Inconsistencies in Natural Language Requirements. ACM Transactions on Software Engg. and Methodology 14(3), 277–330 (2005)
6. Levinson, S.C.: Pragmatics. Cambridge University Press (2000)
7. Sharma, R., Biswas, K.K.: Using Courteous Logic based representatiosn for Requirements Specifications. In: International Workshop on Managing Requirements Knowledge (2011)
8. Grosof, B.N.: Courteous Logic Programs: prioritized conflict handling for rules. IBM Research Report RC20836, IBM Research Division, T.J. Watson Research Centre (1997)
9. Grosof, B.N.: Representing E-Commerce Rules via situated courteous logic programs in RuleML. Electronic Commerce Research and Applications 3(1), 2–20 (2004)
10. Greenspan, S., Borgida, A., Mylopoulos, J.: A Requirements Modeling Language and its logic. Information Systems 11(1), 9–23 (1986)
11. Liu, K., Sun, L., Narasipuram, M.: Norm-based agency for designing Collaborative Information System. Information System 11, 229–247 (2001)
12. Sharma, R., Biswas, K.K.: Using Norm Analysis Patterns for Automated Requirements Validation. In: International Workshop on Requirements Patterns, Co-located with IEEE Conference on Requirements Engineering (RE 2012), Chicago, USA (2012)
13. Stanley, M.: CML: A Knowledge Representation Language with Applications to Requirements Modeling, M.Sc. Thesis, Dept Comp. Sc., University of Troronto (1986)
14. Mylopoulos, J., Borgida, A., Koubarakis, M.: Telos: Representing Knowledge about Information Systems. ACM Transactions on Information Systems (1990)
15. Tsai, J.J.-P.: andWeigert, T, HCLIE: a logic-based requirement language for new software engineering paradigms. Software Engineering 6(4), 137–151 (1991)
16. Zhang, Y., Zhang, W.: Description logic representation for requirement specification. In: Shi, Y., van Albada, G.D., Dongarra, J., Sloot, P.M.A. (eds.) ICCS 2007, Part II. LNCS, vol. 4488, pp. 1147–1154. Springer, Heidelberg (2007)
17. Borgida, A., Greenspan, S., Mylopoulos, J.: Knowledge Representation as the basis for Requirements Specifications. Computer 18(4), 82–91 (1985)

Deduction-Based Formal Verification of Requirements Models with Automatic Generation of Logical Specifications

Radosław Klimek

AGH University of Science and Technology,
al. A. Mickiewicza 30, 30-059 Krakow, Poland
rklimek@agh.edu.pl

Abstract. This work concerns requirements gathering and their formal verification using deductive approach. The approach is based on temporal logic and the semantic tableaux reasoning method. Requirements elicitation is carried out with some UML diagrams. A use case, its scenario and its activity diagram may be linked to each other during the process of gathering requirements. Activities are identified in the use case scenario and then their workflows are modeled using the activity diagram. Organizing the activity diagram workflows into design patterns is crucial and enables generating logical specifications in an automated way. Temporal logic specifications, formulas and properties are difficult to specify by inexperienced users and this fact can be a significant obstacle to the practical use of deduction-based verification tools. The approach presented in this paper attempts to overcome this problem. Automatic transformation of workflow patterns to temporal logic formulas considered as a logical specification is defined. The architecture of an automatic generation and deduction-based verification system is proposed.

Keywords: Requirements Engineering, UML, Use Case Diagram, Use Case Scenario, Activity Diagram, Formal Verification, Deductive Reasoning, Semantic Tableaux Method, Temporal Logic, Workflows, Design Patterns, Logical Modeling, Generating Formulas.

1 Introduction

Requirements engineering is an important part of software modeling. Requirements elicitation is an early phase in the general activity of the process of identifying and articulating purposes and needs for a new system. It should lead into a coherent structure of requirements and have significant impacts on software quality and costs. Software modeling enables better understanding of a domain problem and a developed system. Better software modeling is key in obtaining reliable software. Software reliability implies both correctness and robustness, i.e. it meets its functional (and non-functional) specifications and it delivers a service even if there are unexpected obstacles. The process of gathering requirements understood as an extra layer of abstraction improves not only the reliability of the software development phase but also enhances corporation communication, planning, risk management and enables cost reductions. System requirements are descriptions of delivered services in the context of operational constraints.

L.A. Maciaszek and J. Filipe (Eds.): ENASE 2012, CCIS 410, pp. 157–171, 2013.

Identifying software requirements of the system-as-is, gathering requirements and the formulation of requirements by users enables the identification of defects earlier in a life cycle. Thinking of requirements must precede the code generating act. UML, i.e. Unified Modeling Language [1,2], which is ubiquitous in the software industry can be a powerful tool for the requirements engineering process. UML use cases are central to UML since they strongly affect other aspects of the modeled system and, after joining the activity diagrams, may constitute a good instrument to discover and write down requirements. UML also appears to be an adequate and convenient tool for the documentation of the whole requirements process. What is more, the use case and activity diagrams may also be used in a formal-based analysis and the verification of requirements. Formal methods in requirements engineering may improve the process through a higher level of rigor which enables a better understanding of the domain problem and deductive verification of requirements. Temporal logic is a well established formalism for describing properties of reactive systems. It may facilitate both the system specifying process and the formal verification of non-functional requirements which are usually difficult to verify. The semantic tableaux method, which is a proof formalization for deciding satisfiability and which might be more descriptively called "satisfiability trees", is very intuitive and may be considered goal-based formal reasoning to support requirements engineering.

Maintaining software reliability seems difficult. Formal methods enable the precise formulation of important artifacts arising during software development and the elimination of ambiguity and subjectivity inconsistency. Formal methods can constitute a foundation for providing natural and intuitive support for reasoning about system requirements and they guarantee a rigorous approach in software construction. There are two important parts to the formal approach, i.e. formal specification and formal reasoning, though both are quite closely related to each other. Formal specification introduces formal proofs which establish fundamental system properties and invariants. Formal reasoning enables the reliable verification of desired properties. There are two well established approaches to formal reasoning and information systems verification [3]. The first is based on the state exploration and the second is based on deductive inference. During recent years there was particularly significant progress in the field of state exploration also called "model checking" [4]. However, model checking is an operational rather than analytic approach. Model checking is a kind of simulation for all reachable paths of computation. Unfortunately, the inference method is now far behind state exploration for several reasons, one of which is the choice of a deductive system. However, a more important problem is a lack of methods for obtaining system specifications which are considered as sets of temporal logic formulas and the automation of this procedure. It is not so obvious how to build a logical specification of a system, which in practice is a large collection of temporal logical formulas. The need to build logical specifications can be recognized as a major obstacle to untrained users. Thus, the automation of this process seems particularly important. Application of the formal approach to the requirements engineering phase may increase the maturity of the developed software.

1.1 Motivations, Contributions and Related Works

The main motivation for this work is the lack of satisfactory and documented results of the practical application of deductive methods for the formal verification of requirement models. It is especially important due to the above mentioned model checking approach that wins points through its practical applications and tools (free or commercial). Another motivation that follows is the lack of tools for the automatic extraction of logical specifications, i.e. a set of temporal logic formulas, which is an especially important issue because of the general difficulty of obtaining such a specification. However, the requirements model built using the use case and activity diagrams seems to be suitable for such an extraction. All of the above mentioned aspects of the formal approach seem to be an intellectual challenge in software engineering. On the other hand, deductive inference is very natural and it is used intuitively in everyday life.

The main contribution of this work is a complete deduction-based system, including its architecture, which enables the automated and formal verification of software models which covers requirements models based on some UML diagrams. Another contribution is the use of a non-standard method of deduction for software models. Deduction is performed using the semantic tableaux method for temporal logic. This reasoning method seems to be intuitive. The automation of the logical specification generation process is also an important and key contribution. Theoretical possibilities of such an automation are discussed. The generation algorithm for selected design patterns is presented. Prototype versions of inference engines, i.e. provers for the minimal temporal logic, are implemented.

The approach proposed in this work, i.e. how to build a requirements model through a well-defined sequence of steps, may constitute a kind of quasi methodology and is shown in Fig. 1. The loop between the last two steps refers to a process of both identifying and verifying new and new properties of a model. The work shows that formal methods can integrate requirements engineering with the expectation of obtaining reliable software. It encourages an abstract view of a system as well as reliable and deductive-based formal verification.

Let us discuss some related works. In work by Kazhamiakin [5], a method based on formal verification of requirements using temporal logic and model checking approach is proposed, and a case study is discussed. Work by Eshuis and Wieringa [6] addresses the issues of activity diagram workflows but the goal is to translate diagrams into a format that allows model checking. Work by Hurlbut [7] provides a very detailed survey of selected issues concerning use cases. The dualism of representations and informal character of scenario documentation implies several difficulties in reasoning about system behavior and validating the consistency between diagrams and scenarios description. Work by Barrett et al. [8] presents the transition of use cases to finite state machines. Work by Zhao and Duan [9] shows the formal analysis of use cases, however the Petri Nets formalism is used. In work by Cabral and Sampaio [10] a method of automatic generation of use cases and a proposed subset of natural languages to the algebraic specification is introduced. There are some works in the context of a formal approach and UML for requirements engineering but there is a lack of works in the area of UML models and deductive verification using temporal logic and the semantic tableaux method.

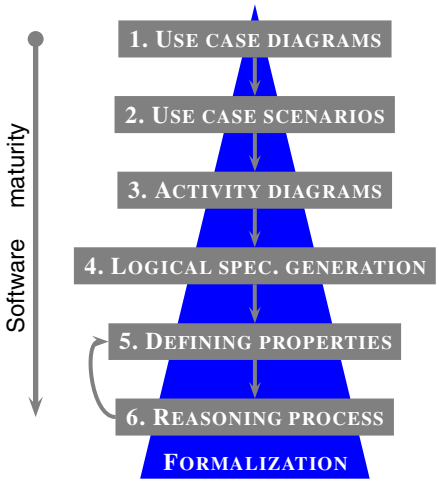


Fig. 1. Software requirements modeling and verification

2 Use Case and Activity Diagrams

This work relates to use case and activity diagrams which are well known, e.g. [1,11,2,12,13]. They are presented from a point of view of the approach adopted in this work, which clearly leads to the formal verification of a requirements model using the deductive method, c.f. Fig. 1.

The *use case diagram* consists of actors and use cases. *Actors* are objects which interact with a system. Actors create the system's environment and they provide interaction with the system. *Use cases* are services and functionalities which are used by actors. Use cases must meet actors' requirements. A use case captures some user visible functions. The main purpose of the use case diagram is to model a system's functionality. However, the diagram does not refer to any details of the system implementation since it is a rather descriptive technique compared with the other UML diagrams. On the other hand, each use case has its associated *scenario* which is a brief narrative that describes the expected use of a system. The scenario is made up of a set of a possible sequence of steps which enables the achievement of a particular goal resulting from use case functionality. Thus, the use case and the goal may be sometimes considered synonymous. The scenario describes a basic flow of events, and possible alternative flows of events.

From the point of view of the approach presented here it is important to draw attention to the requirement that every scenario should contain activities and actions of which individual scenario steps are built. An *activity* is a computation with its internal structure and it may be interrupted by an external event, while an *action* is an executable atomic computation which cannot be interrupted before its completion, c.f. [1]. Defining both the activity and the action results from efforts to achieve greater generality, i.e. to obtain both computations which can be interrupted and computations which cannot be interrupted.

Let us summarize these considerations more formally. In the initial phase of system modeling, in the course of obtaining requirements information, use case diagrams

UCD are built and they contain many use cases UC which describe the desired functionality of a system, i.e. $UC_1, ..., UC_i, ..., UC_l$, where $l > 0$ is a total number of use cases created during the modeling phase. Each use case UC_i has its own scenario. Activities or actions can and should be identified in every step of this scenario. The most valuable situation is when the whole use case scenario is not expressed in a narrative form but contains only the identified activities or actions. Thus, every scenario contains some activities $v_1, ..., v_i, ..., v_m$ and/or actions $o_1, ..., o_j, ..., o_n$, where $m \geq 0$ and $n \geq 0$. Broadly speaking every scenario contains $a_1, ..., a_k, ..., a_p$, where $p > 0$ and every a_k is an activity or an action. The level of formalization presented here, i.e. when discussing use cases and their scenarios, is intentionally not very high. This assumption seems realistic since this is an initial phase of requirements engineering. However, one of the most important things in this approach is to identify activities and actions when creating scenarios since these objects will be used when modeling activity diagrams.

The *activity diagram* enables model activities (and actions) and workflows. It is a graphical representation of workflow showing flow of control from one activity to another one. It supports choice, concurrency and iteration. The activity diagram AD consists of *initial* and *final states* (activities) which show the starting and end point for the whole diagram, *decision points*, activities, actions, e.g. input receiving action or output sending action, and swimlanes. The *swimlane* is useful for partitioning the activity diagram and is a way to group activities in a single thread. The activity diagram shows how an activity depends on others.

The nesting activities is permitted. Every activity is:

1. either an elementary activity, i.e. syntactically indivisible unit, or
2. one of the acceptable design patterns for activity workflows/diagrams.

From the viewpoint of the approach presented in this work it is important to introduce some restrictions on activity workflows. This relies on the introduction of a number of design patterns for activities that give all the workflows a structural form. *Pattern* is a generic description of structure of some computations, and does not limit the possibility of modeling arbitrary complex sets of activities. The following design patterns for activities, c.f. Fig. 2, are introduced: *sequence*, *concurrent fork/join*, *branching* and *loop-while* for iteration.

Let us summarize this section. For every use case UC_i and its scenario, which belongs to any use case diagram UCD, at least one activity diagram AD is created. Workflow for the activity diagram is modeled only using activities v_k and actions o_l which are identified when building a use case scenario. Workflows are composed only using the predefined design patterns shown in Fig. 2. When using these patterns one can build arbitrarily complex activity workflows.

3 Logical Background and Deduction System

Temporal logic and the semantic tableaux reasoning method are two basic instruments used in the presented approach. *Temporal logic* TL is a well established formalism for specification and verification, i.e. representing and reasoning about computations and software models, e.g. [14,15,16]. Temporal logic is broadly used and it can easily express liveliness and safety properties which play a key role in proving system

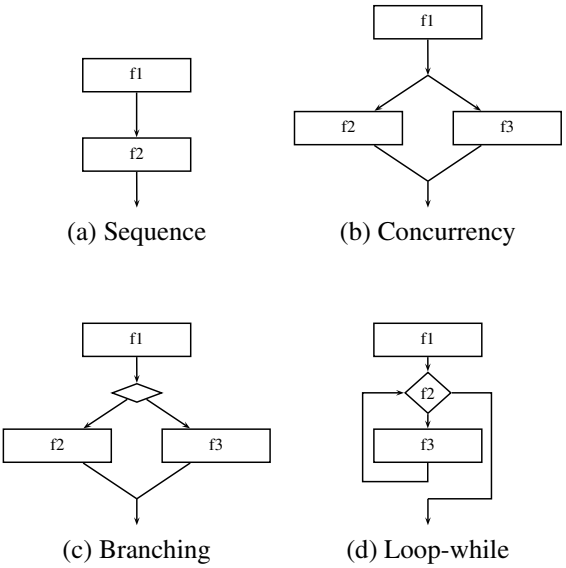


Fig. 2. Design patterns for workflow of activities

properties. Temporal logic exists in many varieties, however, considerations in this paper are limited to axiomatic and deductive systems for the *smallest temporal logic*, which is also known as temporal logic of the class K defined, e.g. [17], as a classical propositional calculus extension of axiom $\Box(P \Rightarrow Q) \Rightarrow (\Box P \Rightarrow \Box Q)$ and inference rule $\frac{\vdash P}{\vdash \Box P}$. This logic can be developed and expanded through the introduction of more complex time structure properties [18,19]. Examples of such enriched logics are: logic/formula D: (sample formula) $\Box p \Rightarrow \Diamond p$; logic/formula T: $\Box p \Rightarrow p$; logic/formula G: $\Diamond\Box p \Rightarrow \Box\Diamond p$; logic/formula 4: $\Box p \Rightarrow \Box\Box p$; logic/formula 5: $\Diamond p \Rightarrow \Box\Diamond p$; logic/formula B: $p \Rightarrow \Box\Diamond p$; etc. It is also possible to combine these properties and logics to establish new logics and relationships among them, e.g. KB4 $\Leftrightarrow$ KB5, KDB4 $\Leftrightarrow$ KTB4 $\Leftrightarrow$ KT45 $\Leftrightarrow$ KT5 $\Leftrightarrow$ KTB. However, it should be clear that considerations in this work are limited to the logic K and we focus our attention on the linear time temporal logic as it is sufficient to define most system properties. The following formulas may be considered as examples of this logic's formulas: $act \Rightarrow \Diamond rec$, $\Box(sen \Rightarrow \Diamond ack)$, $\Diamond liv$, $\Box\neg(evn)$, etc.

Semantic tableaux, or *truth tree*, is a decision procedure for checking formula satisfiability. The truth tree method is well known in classical logic but it can also be applied in the field of temporal logics [20]. It seems that it has some advantages in comparison with the traditional axiomatic approach. The method is based on formula decompositions. At each step of a well-defined procedure, formulas have fewer components since logical connectives are removed. At the end of the decomposition procedure, all branches of the received tree are searched for contradictions. Finding a contradiction in all branches of the tree means no valuation satisfies a formula placed in the tree root. When all branches of the tree have contradictions, it means that the inference

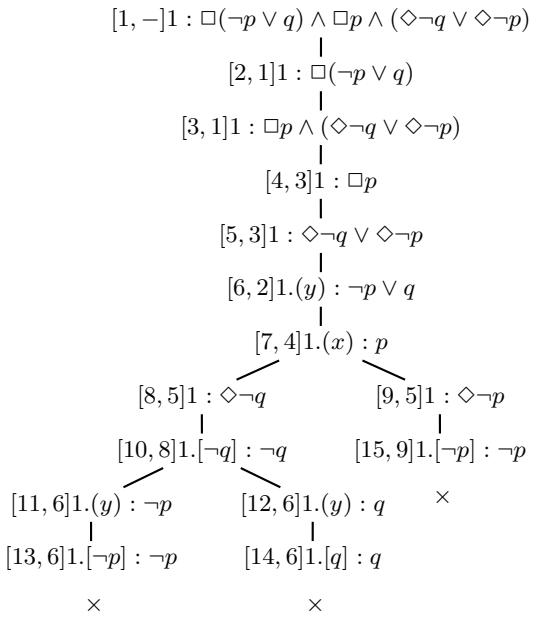


Fig. 3. The inference tree of the semantic tableaux method

tree is *closed*. If the negation of the initial formula is placed in the root, this leads to the statement that the initial formula is true. In the classical reasoning approach, starting from axioms, longer and more complicated formulas are generated and derived. Formulas become longer and longer step by step, and only one of them will lead to the verified formula. The method of semantic tableaux is characterized by the reverse strategy. The inference structure is represented by a tree and not by a sequence of formulas. The expansion of any tree branch may be halted after finding the appropriate sub-structure. In addition, the method provides, through so-called *open* branches of the semantic tree, information about the source of an error, if one is found, which is another and very important advantage of the method. The example of an inference tree for a smallest temporal logic formula is shown in Fig. 3. The purpose of this example is to demonstrate the reasoning and its specificity. The relatively short formula gives a small inference tree, but shows how the method works. Each node contains a (sub-)formula which is either already decomposed, or is subjected to decomposition in the process of building the tree. Each formula is preceded by a label referring to both a double number and a current world reference. The label $[i, j]$ means that it is the i-th formula, i.e. the i-th decomposition step, received from the decomposition transformation of a formula stored in the j-th node. The label "1 :" represents the initial world in which a formula is true. The label "$1.(x)$", where x is a free variable, represents all possible worlds that are consequent of world 1. On the other hand, the label "$1.[p]$", where p is an atomic formula, represents one of the possible worlds, i.e. a successor of world 1, where formula p is true. The decomposition procedure adopted and presented here, as well as labeling,

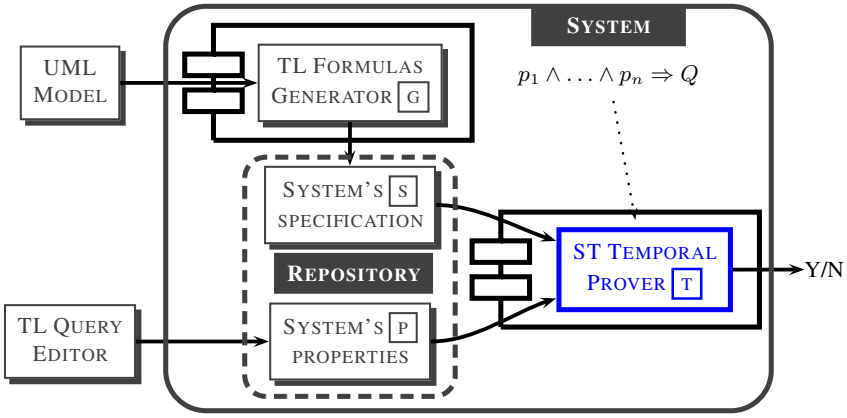


Fig. 4. Deduction-based verification system

refers to the first-order predicate calculus and can be found for example in the work of [21]. All branches of the analyzed trees are closed ($\times$). There is no valuation that satisfies the root formula. This consequently means that the formula[1] before the negation, i.e. $\neg(\Box(\neg p \vee q) \wedge \Box p \wedge (\Diamond\neg q \vee \Diamond\neg p))$, is always satisfied.

The architecture of the proposed inference system using the semantic tableaux method for the UML activity diagram workflows is presented in Fig. 4. The system works automatically and consists of some important elements. Some of them can be treated as a software components/plugins. The first component G generates a logical specification which is a set of a usually large number of temporal logic formulas (of class K). Formulas generation is performed automatically by extracting directly from the design patterns contained in a workflow model. The extraction process is discussed in section 4. Formulas are collected in the S module (repository, i.e. file or database) that stores the system specifications. It is treated as a conjunction of formulas $p_1 \wedge \ldots \wedge p_n = P$, where $n > 0$ and p_i is a specification formula generated during the extraction. The P module provides the desired system properties which are expressed in temporal logic. The easiest way to obtain such formulas is to use an editor and manually introduce the temporal logic formula Q. Such a formula(s) is/are identified by an analyst and describe(s) the expected properties of the investigated system. Both the system specification and the examined properties are input to the T component, i.e. *Semantic Tableaux Temporal Prover*, or shortly *ST Temporal Prover*, which enables the automated reasoning in temporal logic using the semantic tableaux method. The input for this component is the formula $P \Rightarrow Q$, or, more precisely:

$$p_1 \wedge \ldots \wedge p_n \Rightarrow Q \tag{1}$$

After the negation of formula 1, it is placed at the root of the inference tree and decomposed using well-defined rules of the semantic tableaux method. If the inference tree

[1] This formula and this inference tree originate from the master thesis by Łukasz Rola from AGH University of Science and Technology, Kraków, Poland. The thesis was prepared under the supervision of the author of this work.

is closed, this means that the initial formula 1 is true. Broadly speaking, the output of the T component, and therefore also the output of the whole deductive system, is the answer Yes/No in response to any introduction of a new tested property for a system modeled using UML. This output also realizes the final step of the procedure shown in Fig. 1.

The whole verification procedure can be summarized as follows:

1. automatic generation of system specification (the G component), and then stored (the S module) as a conjunction of all extracted formulas;
2. introduction of a property of the system (the P module) as a temporal logic formula or formulas;
3. automatic inference using semantic tableaux (the T component) for the whole complex formula 1.

Steps 1 to 3, in whole or individually, may be processed many times, whenever the specification of the UML model is changed (step 1) or there is a need for a new inference due to the revised system specification (steps 2 or 3).

4 Generation of Specification

Automated support for building any logical specification of a modeled system is essential for requirements acquisition and formal verification of properties. Such a specification usually consists of a large number of temporal logic formulas and its manual development is practically impossible since this process can be monotonous and error-prone. Last but not least, the creation of such a logical specification can be difficult for inexperienced analysts. Therefore, all efforts towards automation of this design phase are very desirable.

The proposed method and algorithm for an automatic extraction of a logical specification is based on the assumption that all workflow models for the UML activity diagrams are built using only well-known design patterns, c.f. Fig. 2. It should be noted that constructed activity workflow models are based on activities and actions identified when writing use case scenarios. The whole process of building a logical specification involves the following steps:

1. analysis of a workflow model of activities to extract all design patterns,
2. translation of the extracted patterns to a logical expression W_L which is similar to a well-known regular expression,
3. generation of a logical specification L from the logical expression, i.e. receiving a set of formulas of linear time temporal logic of class K.

Let us introduce all formal concepts necessary for the application of the above steps to illustrate the entire procedure in a more formal way.

The temporal logic *alphabet* consists of the following symbols: a countable set of atomic formulas p, q, r, etc., classical logic symbols like **true**, **false**, $\neg$, $\vee$, $\wedge$, $\Rightarrow$, $\Leftrightarrow$ and two linear temporal logic operators $\Box$ and $\Diamond$. (One can also introduce other symbols, e.g. parenthesis, which are omitted here to simplify the definition.) For this alphabet,

syntax rules for building well-formed logic formulas can be defined. These rules are based on BNF notation. The definition of *formula* of linear time temporal logic LTL includes the following steps:

- every atomic formula p, q, r, etc. is a formula,
- if p and q are formulas, then $\neg p$, $p \vee q$, $p \wedge q$, $p \Rightarrow q$, $p \Leftrightarrow q$ are formulas, too,
- if p and q are formulas, then $\Box p$, $\Diamond p$, are formulas, too.

Examples of valid, well-formed and typical formulas, restricted to the logic K, are the following formulas: $p \Rightarrow \Diamond q$ and $\Box\neg(p \wedge (q \vee r))$.

A set of LTL formulas describe temporal properties of individual design patterns of every UML activity diagram. This aspect is important as the approach presented here is based on predefined design patterns. An *elementary set* of formulas over atomic formulas a_i, where $i = 1, \ldots, n$, which is denoted $pat(a_i)$, is a set of temporal logic formulas $f_1, ..., f_m$ such that all formulas are well-formed (and restricted to the logic K). For example, an elementary set $pat(a, b, c) = \{a \Rightarrow \Diamond b, \Box\neg(b \wedge \neg c)\}$ is a two-element set of LTL formulas, created over three atomic formulas.

Suppose that there are predefined sets of formulas for every design pattern of the UML activity workflow from Fig. 2. The proposed temporal logic formulas should describe both safety and liveness properties of each pattern. In this way, $Sequence(a, b) = \{a \Rightarrow \Diamond b, \Box\neg(a \wedge b)\}$ describes properties of the Sequence pattern. Set $Concurrency(a, b, c) = \{a \Rightarrow \Diamond b \wedge \Diamond c, \Box\neg(a \wedge (b \vee c))\}$ describes the Concurrency pattern and $Branching(a, b, c) = \{a \Rightarrow (\Diamond b \wedge \neg\Diamond c) \vee (\neg\Diamond b \wedge \Diamond c), \Box\neg(b \wedge c)\}$ the Branching pattern.

Let us introduce some aliases for all patterns (Fig. 2): Seq as $Sequence$, $Concur$ as $Concurrency$, $Branch$ as $Branching$ and $Loop$ as $Loop{-}while$.

Every activity diagram workflow is designed using only predefined design patterns. Every design pattern has a predefined and countable set of linear temporal logic formulas. The workflow model can be quite complex and it may contain nesting patterns. This is one important reason why it is needed to define a symbolic notation which represents any potentially complex structure of the activity workflow. *Logical expression* W_L is a structure created using the following rules:

- every elementary set $pat(a_i)$, where $i = 1, \ldots, n$ and a_i is an atomic formula, is a logical expression,
- every $pat(A_i)$, where $i = 1, \ldots, n$ and where A_i is either
  - a sequence of atomic formulas a_j, where $j = 1, \ldots, m$, or
  - a set $pat(a_j)$, where $j = 1, \ldots, m$ and a_j is an atomic formula, or
  - a logical expression $pat(A_j)$, where $j = 1, \ldots, m$

  is also a logical expression.

The above defined symbolic notation is equivalent to a graphical one which is usually the most convenient for users when modeling a system. However, another rule describing a special case of a sequence of sequences is introduced:

- if $pat_1()$ and $pat_2()$ are logical expressions, where the empty parentheses means any arguments, then their concatenation $pat_1() \cdot pat_2()$, also noted $pat_1()pat_2()$, is also a logical expression.

```
                        /* ver. 15.10.2012
/* Activity Design Patterns
Sequence(f1,f2):
f1 => <>f2
[]~(f1 & f2)
SeqSeq(f1,f2,f3):
f1 => <>f2
f2 => <>f3
[]~ ((f1 & f2) | (f2 & f3) | (f1 & f3))
Concurrency(f1,f2,f3):
f1 => <>f2 & <>f3
[]~(f1 & (f2 | f3))
Branching(f1,f2,f3):
f1 => (<>f2 & ~<>f3) | (~<>f2 & <>f3)
[]~(f2 & f3)
Loop-while(f1,f2,f3):
f1 => <>f2
f2 & c(f2) => <>f3
f2 & ~c(f2) => ~<>f3
f3 => <>f2
[]~((f1 & f2) | (f2 & f3) | (f1 & f3))
```

Fig. 5. Predefined set of patterns and their temporal properties

This rule is redundant but the concatenation of sequences seems more convenient and in addition it provides concatenation of sequences as a sequence of three arguments. Thus, another predefined set $SeqSeq(a, b, c) = \{a \Rightarrow \Diamond b, b \Rightarrow \Diamond c, \Box\neg((a \wedge b) \vee (b \wedge c) \vee (a \wedge c))\}$ is introduced. It describes the concatenation properties of sequences of two patterns.

Any logical expression represents an arbitrary complex and nested workflow model for an activity diagram which was modeled using predefined design patterns. The logical expression allows representation of sets of temporal logic formulas for every design pattern. Thus, the last step is to define a logical specification which is generated from a logical expression. *Logical specification* L consists of all formulas derived from a logical expression, i.e. $L(W_L) = \{f_i : i > 0\}$, where f_i is a temporal logic formula. Generating logical specifications, which constitutes a set of formulas, is not a simple summation of formula collections resulting from a logical expression. Thus, the sketch of the generation algorithm is presented below.

The generation process of a logical specification L has two inputs. The first one is a logical expression W_L which is built for the activity workflow model. The second one is a predefined set P of temporal logic formulas for every design pattern. The example of such a set is shown in Fig 5. Most elements of the predefined P set, i.e. comments, two temporal logic operators, classical logic operators, are not in doubt. f_1, f_2 etc. are atomic formulas for a pattern. They constitute a kind of formal arguments for a pattern. $\Diamond f$ means that sometime (or eventually in the future) activity f is completed, i.e. the token left the activity. $c(f)$ means that the logical condition associated with activity f has been evaluated and is satisfied. All formulas describe both safety and

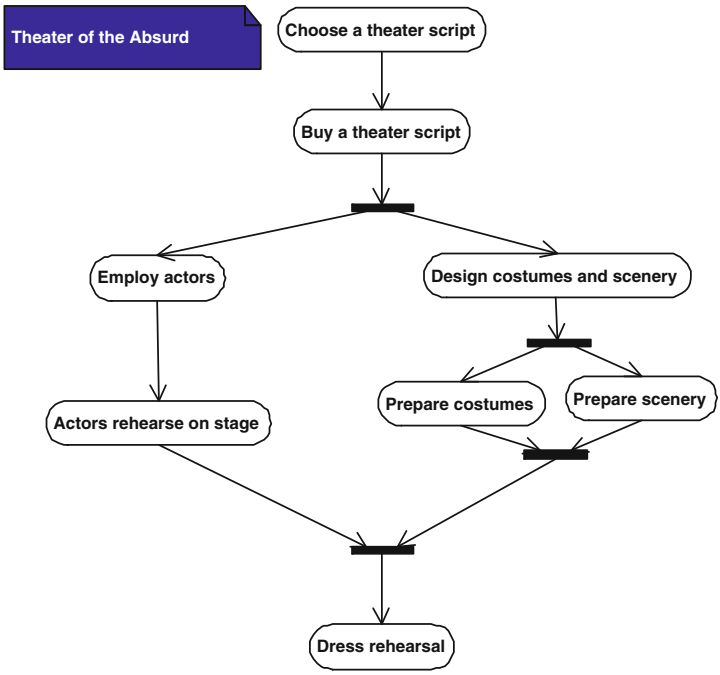


Fig. 6. A sample activity diagram and its workflow

liveness properties for a pattern.

The output of the generation algorithm is the logical specification understood as a set of temporal logic formulas. The sketch of the algorithm is as follows:

1. at the beginning, the logical specification is empty, i.e. $L = \emptyset$;
2. the most nested pattern or patterns are processed first, then, less nested patterns are processed one by one, i.e. patterns that are located more towards the outside;
3. if the currently analyzed pattern consists only of atomic formulas, the logical specification is extended, by summing sets, by formulas linked to the type of the analyzed pattern $pat()$, i.e. $L = L \cup pat()$;
4. if any argument is a pattern itself, then the logical disjunction of all its arguments, including nested arguments, is substituted in place of the pattern.

The above algorithm refers to similar ideas in work [22]. Let us supplement the algorithm by some examples. The example for the step 3: $SeqSeq(p, q, r)$, gives $L = \{a \Rightarrow \Diamond b, b \Rightarrow \Diamond c, \Box\neg((a \wedge b) \vee (b \wedge c) \vee (a \wedge c))\}$ and $Branch(a, b, c)$ gives $L = \{a \Rightarrow (\Diamond b \wedge \neg\Diamond c) \vee (\neg\Diamond b \wedge \Diamond c), \Box\neg(b \wedge c)\}$. The example for the step 4: $Concur(Seq(a, b), c, d)$ leads to $L = \{a \Rightarrow \Diamond b, \Box\neg(a \wedge b)\} \cup \{(a \vee b) \Rightarrow \Diamond c \wedge \Diamond d, \Box\neg((a \vee b) \wedge (c \vee d))\}$.

Let us consider a simple example to illustrate the approach, c.f. Fig. 6. This example illustrates the proposed verification method only for a single activity diagram (workflow). This is due to size constraints on the work. A model consisting of several activity diagrams obtained from multiple use case scenarios could also be considered. A single liveness property (3) is verified but it is possible to verify other properties, including

the safety property. When any formula describing activity is considered, e.g. $\Diamond p$, this means that the activity p is sometime (or eventually in the future) completed.

Suppose[2] that an owner of the "Theater of the Absurd", which has recently declined in popularity, wants to organize a new and great performance to begin a new and splendid period for the institution. Analyzing the scenario of the UML use case "Preparation of a theatrical performance" the following activities are identified: "Choose a theater script", "Buy a theater script", "Employ actors", "Actors rehearse on stage", "Design costumes and scenery" "Prepare costumes", "Prepare scenery" and "Dress rehearsal". The way of creating the use case scenario is not in doubt and therefore is omitted here. The scenario is usually given in a tabular form and its steps should include all the above identified activities. When building a workflow for the activity diagram, a combination of Sequence and Concurrency patterns is used. The logical expression W_L for this model is

$$\begin{aligned} SeqSeq(&ChooseScript, Concur(BuyScript,\\ &Seq(EmployActors, ActorsRehearse),\\ &Concur(Design, PrepareCostumes,\\ &PrepareScenery)), DressRehearsal) \end{aligned}$$

or after the substitution of propositions as letters of the Latin alphabet: a – Choose a theater script, b – Buy a theater script, c – Employ actors, d – Actors rehearse on stage, e – Design costumes and scenery, f – Prepare costumes, g – Prepare scenery, and h – Dress rehearsal, then logical expression W_L is

$$SeqSeq(a, Concur(b, Seq(c, d), Concur(e, f, g)), h)$$

From the obtained expression a logical specification L will be built in the following steps. At the beginning, the specification of a model is $L = \emptyset$. Most nested patterns are Seq and $Concur$. Sequence gives $L = L \cup \{c \Rightarrow \Diamond d, \Box\neg(c \wedge d)\}$, and then Concurrency gives $L = L \cup \{e \Rightarrow \Diamond f \wedge \Diamond g, \Box\neg(e \wedge (f \vee g))\}$. The next considered pattern is Concurrency for which the disjunction of arguments is considered $L = L \cup \{b \Rightarrow \Diamond(c \vee d) \wedge \Diamond(e \vee f \vee g), \Box\neg(b \wedge ((c \vee d) \vee (e \vee f \vee g)))\}$. The most outside situated pattern gives $L = L \cup \{a \Rightarrow \Diamond(b \vee c \vee d \vee e \vee f \vee g), (b \vee c \vee d \vee e \vee f \vee g) \Rightarrow \Diamond h, \Box\neg((a \wedge (b \vee c \vee d \vee e \vee f \vee g)) \vee ((b \vee c \vee d \vee e \vee f \vee g) \wedge h) \vee (a \wedge h))\}$.

Thus, the resulting specification contains formulas

$$\begin{aligned} L = \{&c \Rightarrow \Diamond d, \Box\neg(c \wedge d), e \Rightarrow \Diamond f \wedge \Diamond g,\\ &\Box\neg(e \wedge (f \vee g)), b \Rightarrow \Diamond(c \vee d) \wedge \Diamond(e \vee f \vee g),\\ &\Box\neg(b \wedge ((c \vee d) \vee (e \vee f \vee g))),\\ &a \Rightarrow \Diamond(b \vee c \vee d \vee e \vee f \vee g),\\ &(b \vee c \vee d \vee e \vee f \vee g) \Rightarrow \Diamond h,\\ &\Box\neg((a \wedge (b \vee c \vee d \vee e \vee f \vee g)) \vee\\ &((b \vee c \vee d \vee e \vee f \vee g) \wedge h) \vee (a \wedge h))\} \end{aligned} \tag{2}$$

[2] This example is an adaptation of another example from lectures by Prof. Tomasz Szmuc (AGH University of Science and Technology, Kraków, Poland).

The examined property can be

$$a \Rightarrow \Diamond h \tag{3}$$

which means that if the theater script is chosen then sometime in the future the dress rehearsal is completed. The whole formula to be analyzed using the semantic tableaux method is

$$\begin{aligned}((c \Rightarrow \Diamond d) \wedge (\Box\neg(c \wedge d)) \wedge (e \Rightarrow \Diamond f \wedge \Diamond g) \wedge \\ (\Box\neg(e \wedge (f \vee g))) \wedge \\ (b \Rightarrow \Diamond(c \vee d) \wedge \Diamond(e \vee f \vee g)) \wedge \\ (\Box\neg(b \wedge ((c \vee d) \vee (e \vee f \vee g)))) \wedge \\ (a \Rightarrow \Diamond(b \vee c \vee d \vee e \vee f \vee g)) \wedge \\ ((b \vee c \vee d \vee e \vee f \vee g) \Rightarrow \Diamond h) \wedge \\ (\Box\neg((a \wedge (b \vee c \vee d \vee e \vee f \vee g)) \vee \\ ((b \vee c \vee d \vee e \vee f \vee g) \wedge h) \vee (a \wedge h)))) \\ \Rightarrow (a \Rightarrow \Diamond h)\end{aligned} \tag{4}$$

Formula 2 represents the output of the G component in Fig. 4. Formula 4 provides a combined input for the T component in Fig. 4. Presentation of a full reasoning tree for formula 4 exceeds the size of the work since the tree contains many hundreds of nodes. The formula is true and the examined property 3 is satisfied in the considered activity model.

5 Conclusions

The work presents a new approach to the formal verification of requirements models using temporal logic and the semantic tableaux method. The paper presents the method for transformation of activity diagram design patterns to logical expressions which represent a model of requirements. The algorithm for generating logical specifications from a logical expression is proposed. The advantage of the whole methodology is that it provides an innovative concept for formal verification which might be done for any requirements model created using the UML use case and activity diagrams. It enables both receiving high-quality requirements and stable software systems as well as the transformation from verified goal-oriented requirements to reliable systems.

Future work may include the implementation of the logical specification generation module and the temporal logic prover. Another possibility is an extension of the deduction engine in order to support more complex logics to compare with the minimal temporal logic. Important issues are questions how to apply the approach in the real software development process and in the industry practice. The approach should results in a CASE software providing modeling requirements and software design. The purpose of the CASE software should include both the identification of activities and actions in use case scenarios, workflow modeling in activity diagrams as well as detecting and resolving possible inconsistencies of models being designed by different designers. All the above mentioned efforts will lead to user friendly deduction based formal verification of requirements engineering and user centered software development.

References

1. Rumbaugh, J., Jacobson, I., Booch, G.: The Unified Modeling Language Reference Manual. Addison Wesley (1999)
2. Pender, T.: UML Bible. John Wiley & Sons (2003)
3. Clarke, E., Wing, J., et al.: Formal methods: State of the art and future directions. ACM Computing Surveys 28(4), 626–643 (1996)
4. Clarke, E., Grumberg, O., Peled, D.: Model Checking. MIT Press (1999)
5. Kazhamiakin, R., Pistore, M., Roveri, M.: Formal verification of requirements using spin: A case study on web services. In: Proceedings of the Second International Conference on Software Engineering and Formal Methods (SEFM 2004), pp. 406–415 (2004)
6. Eshuis, R., Wieringa, R.: Tool support for verifying uml activity diagrams. IEEE Transactions on Software Engineering 30 (7), 437–447 (2004)
7. Hurlbut, R.R.: A survey of approaches for describing and formalizing use cases. Technical Report XPT-TR-97-03, Expertech, Ltd (1997)
8. Barrett, S., Sinnig, D., Chalin, P., Butler, G.: Merging of use case models: Semantic foundations. In: 3rd IEEE International Symposium on Theoretical Aspects of Software Engineering (TASE 2009), pp. 182–189 (2009)
9. Zhao, J., Duan, Z.: Verification of use case with petri nets in requirement analysis. In: Gervasi, O., Taniar, D., Murgante, B., Laganà, A., Mun, Y., Gavrilova, M.L. (eds.) ICCSA 2009, Part II. LNCS, vol. 5593, pp. 29–42. Springer, Heidelberg (2009)
10. Cabral, G., Sampaio, A.: Automated formal specification generation and refinement from requirement documents. Journal of the Brazilian Computer Society 14 (1), 87–106 (2008)
11. Fowler, M.: UML Distilled, 3rd edn. Addison-Wesley (2004)
12. Cockburn, A.: Writing Effective Use Cases. Addison-Wesley (2001)
13. Klimek, R., Szwed, P.: Formal analysis of use case diagrams. Computer Science 11, 115–131 (2010)
14. Emerson, E.: Temporal and Modal Logic. In: Handbook of Theoretical Computer Science. Volume B, pp. 995–1072. Elsevier, MIT Press (1990)
15. Klimek, R.: Introduction to temporal logic. AGH University of Science and Technology Press (1999) (in Polish)
16. Wolter, F., Wooldridge, M.: Temporal and dynamic logic. Journal of Indian Council of Philosophical Research XXVII(1), 249–276 (2011)
17. van Benthem, J.: Temporal Logic. In: Handbook of Logic in Artificial Intelligence and Logic Programming, pp. 241–350. Clarendon Press (1993–1995)
18. Chellas, B.F.: Modal Logic. Cambridge University Press (1980)
19. Pelletier, F.: Semantic tableau methods for modal logics that include the b and g axioms. Technical Report Technical Report FS-93-01, AAAI (Association for the Advancement of Artificial Intelligence) (1993)
20. d'Agostino, M., Gabbay, D., Hähnle, R., Posegga, J.: Handbook of Tableau Methods. Kluwer Academic Publishers (1999)
21. Hähnle, R.: Tableau-based Theorem Proving. ESSLLI Course (1998)
22. Klimek, R.: Towards formal and deduction-based analysis of business models for soa processes. In: Filipe, J., Fred, A. (eds.) Proceedings of 4th International Conference on Agents and Artificial Intelligence (ICAART 2012), Vilamoura, Algarve, Portugal, February 6-8, vol. 2, pp. 325–330. SciTePress (2012)

Author Index